SYNCHRONIZATION AND FAULT TOLERANCE TECHNIQUES IN CONCURRENT SHARED MEMORY SYSTEMS

by

Sahil Dhoked

APPROVED BY SUPERVISORY COMMITTEE:

Neeraj Mittal, Chair

Ramaswamy Chandrasekaran

Subbarayan Venkatesan

Andras Farago

This dissertation is dedicated to my mother,

Geeta Dhoked

She grew up being hearing-impaired

and struggled to receive proper education,

but is now proud to have her son achieve

one of the highest levels of formal education.

SYNCHRONIZATION AND FAULT TOLERANCE TECHNIQUES IN CONCURRENT SHARED MEMORY SYSTEMS

by

SAHIL DHOKED, BTech

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2022

ACKNOWLEDGMENTS

I owe this dissertation to my advisor and role model, Dr. Neeraj Mittal. His aptitude for problem solving with remarkable precision and quality is what I continually strive for. My accomplishments cannot fully incorporate the countless hours of brainstorming and field work we conducted behind the scenes. The invaluable lessons I have learnt during this time have and will continuously help me grow, both professionally and personally. I will be forever indebted to him, for his consistent encouragement, direction, insights, and nurturing, all of which have benefited me in developing the research acumen required to complete this work.

I would like to thank the committee members, Dr. R. Chandrasekaran, Dr. S. Venkatesan and Dr. Andras Farago for their challenging questions and key insights.

I am grateful to have had the opportunity to work with Dr. W. Golab, whom I consider an expert in the field. His knowledge in this field, and the broader perspective he provides during discussions, have been a great venue for me to learn and strive upon.

A special thanks to my parents, Geeta and Gumanmal Dhoked, and my brother, Hemant Dhoked for their unending love and motivation. They have always been a pillar of support upon which I have grown, personally and professionally. I would also like to express my gratitude to my sister, Shruti Jogani, and my aunt, Kavita Jain as well as their respective families for being there for me and making working in a foreign country easier.

I am grateful to The University of Texas at Dallas for this incredible opportunity and the exciting journey as I near the end of this stint. During this time, I got the opportunity to learn from classroom courses, network with like-minded peers and build precious friendships. I would like to thank all my instructors, friends and peers for accompanying me on this incredible journey.

March 2022

SYNCHRONIZATION AND FAULT TOLERANCE TECHNIQUES IN CONCURRENT SHARED MEMORY SYSTEMS

Sahil Dhoked, PhD
The University of Texas at Dallas, 2022

Supervising Professor: Neeraj Mittal, Chair

Mutual exclusion is one of the most commonly used techniques to handle contention in concurrent systems. Traditionally, mutual exclusion algorithms have been designed under the assumption that a process does not fail while acquiring/releasing a lock or while executing its critical section. However, failures do occur in real life, potentially leaving the lock in an inconsistent state. This gives rise to the problem of *recoverable mutual exclusion (RME)* that involves designing a mutual exclusion (ME) algorithm that can tolerate failures, while maintaining safety and liveness properties.

With the recent development of NVRAM (non-volatile random-access memory) technologies, there is renewed interest in the RME problem. The NVRAM technology is a combination of the low latency of traditional random-access memory with the high persistence of disk storage media. NVRAMs can be used to provide near-instantaneous recovery to many problems including the RME problem.

This work describes techniques for designing efficient algorithms to solve the RME problem under two different failure models, independent failure model and system-wide failure model, depending on whether processes fail independently or simultaneously. Additionally, especially for systems with low memory capacity, this work describes fault-tolerant techniques for reclaiming memory, in case there is no built-in support for garbage collection.

The primary measure of an RME algorithm is its performance. Performance of any ME algorithm, including an RME algorithm, is measured by the number of *remote memory references (RMRs)* made by a process—for acquiring and releasing a lock as well as recovering the lock structure after a failure. Loosely speaking, it represents the number of expensive shared memory instructions. In this work, two models of RMR computation are considered: (a) the CC model, and (b) the DSM model. The results mentioned in this work are applicable to both of these computation models.

For the independent failure model, this work presents a framework that transforms any algorithm that solves the RME problem into an algorithm whose performance (in terms of RMRs) can simultaneously *adapt* to (a) the number of processes competing for the lock, *as well as* (b) the number of failures that have occurred in the recent past, while maintaining the correctness and performance properties of the underlying RME algorithm.

Assume that, for n processes, the RMR complexity of the underlying RME algorithm is $R(n)$. Then, this framework yields an RME algorithm for which the RMR complexity is given by $\mathcal{O}(\min\{\ddot{c}, \sqrt{F+1},\ R(n)\})$, where $\ddot{c}$ denotes the *point contention* (number of active processes) and F denotes the number of failures in the *recent past*.

The system-wide failure model is a special case of the independent failure model that assumes that failures only occur simultaneously. For example, a power outage is a real life example of such a failure. This model makes a stronger assumption than just multiple independent failures. This assumption is leveraged with enhanced RME algorithms presented under this model. For the system-wide failure model, this work presents optimal RME algorithms (and related transformations) whose worst-case performance yield a $\mathcal{O}(1)$ RMR complexity.

The fault-tolerant memory reclamation algorithm provides novel techniques to bound the worst-case space complexity of RME algorithms. The techniques used are general enough that they may also be employed to bound the space complexity of other RME algorithms. Its RMR complexity is merely an additive factor of $\mathcal{O}(1)$.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Concurrent systems have been on the rise since the last few decades. One of the biggest examples of concurrent systems is the cloud system architecture. These systems facilitate multi-tasking and parallel computations, leading to an increased throughput and scalability in the system. Nowadays, even smaller devices like mobile phones contain multiple cores. However, the multi-process nature of concurrent systems is highly challenging to model and debug. Thus, these systems are highly prone errors like deadlocks and race conditions especially when in contention for shared resources.

One of the most commonly used techniques to handle contention in a concurrent system is to use *mutual exclusion (ME)*. The mutual exclusion problem was first defined by Dijkstra more than half a century ago in (Dijkstra, 1965). Using locks that provide mutual exclusion enables a process to execute its *critical section* (part of the program that involves accessing shared resources) in isolation without worrying about interference from other processes. This avoids race conditions, thereby ensuring that the system always stays in a consistent state and produces correct outcome under all scenarios.

Generally, algorithms for mutual exclusion are designed with the assumption that failures do not occur, especially while a process is accessing a lock or a shared resource. However, such failures can occur in the real world. A power outage or network failure might create an unrecoverable situation causing processes to be unable to continue. If such failures occur, traditional mutual exclusion algorithms, which are not designed to operate properly in the presence of failures, may fail to guarantee important safety and/or liveness properties (*e.g.*, system may deadlock). In many cases, such failures may have disastrous consequences. This gives rise to the *recoverable mutual exclusion (RME) problem*. The RME problem involves designing an algorithm that ensures mutual exclusion under the assumption that process

failures may occur at *any* point during their execution, but the system is able to recover from such failures and proceed without any adverse consequences.

Traditionally, concurrent algorithms use checkpointing and logging to tolerate failures by regularly saving relevant portion of application state to a persistent storage such as hard disk drive (HDD). Accessing a disk is orders of magnitude slower than accessing main memory. As a result, checkpointing and logging algorithms are often designed to minimize disk accesses. *Non-volatile random-access memory (NVRAM)* is a new class of memory technologies that, as per (Oukid and Lersch, 2018), combines the low latency and high bandwidth of traditional random access memory with the density, non-volatility, and economic characteristic of traditional storage media (*e.g.*, hard disk drive). Existing checkpointing and logging algorithms can be modified to use NVRAMs instead of disks to yield better performance, but, in doing so, one would not be leveraging the true power of NVRAMs (Narayanan and Hodson, 2012; Golab and Ramaraju, 2019). NVRAMs can be used to directly store implementation specific variables and, as such, have the potential for providing near-instantaneous recovery from failures.

Most of the application data can be easily recovered after failures by directly storing implementation-specific variables on NVRAMs. However, recovery of implementation variables alone is not enough. Processor state information such as contents of general and special purpose CPU registers (*e.g.*, program counter, condition code register, stack pointer, *etc.*) as well as contents of cache cannot always be recovered fully. In other words, recovery may be lossy, and, if not handled properly, a failure may cause the system to behave erroneously upon recovery. Due to this reason, there is a renewed interest in developing fast and dependable algorithms for solving many important computing problems in software systems vulnerable to process failures using NVRAMs. Using innovative methods, with NVRAMs in mind, this work aims to design efficient and robust fault-tolerant algorithms for solving mutual exclusion and other important concurrent problems.

In this work, the RME problem is studied extensively, considering both individual (independent) and simultaneous (system-wide) failures. The RME problem in the current form (for independent failures) was formally defined a few years ago by Golab and Ramaraju in (Golab and Ramaraju, 2016). Several algorithms have been proposed to solve this problem (Golab and Ramaraju, 2019; Golab and Hendler, 2017; Jayanti and Joshi, 2017; Jayanti et al., 2019; Chan and Woelfel, 2020; Dhoked and Mittal, 2020) as well as some of its variants (Jayanti and Joshi, 2019; Katzan and Morrison, 2021; Golab and Hendler, 2018). The RME problem for system-wide failures was defined in (Golab and Hendler, 2018). The system-wide failure model is a special case of the independent failure model that assumes that failures only happen simultaneously. A power outage is a real-life example of a system-wide failure, while a segmentation fault would be a real-life example of an independent failure. Note that the system-wide failure model makes a stronger assumption than just multiple independent failures which is further discussed in Chapter 8.

One of the most important measures of performance of an RME algorithm is the maximum number of *remote memory references (RMRs)* made by a process per critical section request in order to acquire and release the lock as well as recover the lock after a failure. Intuitively, RMR complexity captures the number of "expensive" shared memory steps performed by a process. Whether or not a memory reference is considered an RMR depends on the underlying memory model. The two most common memory models used to analyze the performance of an RME algorithm are the *cache-coherent (CC)* and *distributed shared memory (DSM)* models. Roughly speaking, a step is considered to incur an RMR in the CC model if it causes a memory location to be cached or a cached copy to be invalidated, and in the DSM model if it accesses data stored on a remote memory module. The CC model captures the working of the caching system used by hardware manufacturers to mask the high latency of memory, whereas the DSM model captures the NUMA (non-uniform memory access) effect observed in machines with large number of cores when memory is partitioned

into multiple modules. Hereafter, any RMR complexity bounds mentioned in the text are assumed to hold for both CC and DSM models *unless otherwise stated*.

Different RME algorithms provide different trade-offs in (RMR) performance guarantees under different failure scenarios. For example, one of the RME algorithms proposed in (Golab and Ramaraju, 2019) has an RMR complexity that grows linearly with the number of failures. Specifically, it has *optimal* RMR complexity of $\mathcal{O}(1)$ in the absence of failures, but its RMR complexity may grow *unboundedly* if failures occur repeatedly. On the other hand, the RME algorithm in (Jayanti et al., 2019) has an RMR complexity of $\mathcal{O}(\log n / \log \log n)$, where n is the number of processes in the system, irrespective of how many failures have occurred in the system (including the case when the system has not experienced any failures). The trade-offs in performance guarantees can be categorized under three different scenarios[1]: (a) No failures, (b) Limited number of failures, and (c) Arbitrarily large number of failures. Table 1.1 presents a comparison of the performance of known RME algorithms with respect to the above mentioned scenarios. Recently, Chan and Woelfel have proved a lower bound[2] of $\Omega(\log n / \log \log n)$ on the worst-case RMR complexity of any RME algorithm using currently available hardware instructions and practical word size of $\Theta(\log n)$ (Chan and Woelfel, 2021). A more detailed description of the related work is given later in Chapter 3.

Another noteworthy metric of an RME algorithm is its fairness property. Roughly speaking, fairness properties help "regulate" access to the critical section. This allows a fair chance for slower processes to access shared resources, despite failures. There have been multiple flavors of fairness properties defined in the literature (Golab and Ramaraju, 2019; Jayanti and Joshi, 2017; Dhoked and Mittal, 2020). Fairness requirements from an RME algorithm depend on external factors and are subject to interpretation.

[1]This discussion applies only for the independent failure model

[2]The lower bound results only hold for the individual failure model

Table 1.1. Comparison of known solutions to recoverable mutual exclusion problem with respect to RMR complexity under three different scenarios.

Algorithm	**RMR Complexity**			**Adaptive to Contention**
	No failures	**Limited failures**	**Arbitrary number of failures**	
Golab and Ramaraju's transformation for recoverability (Golab and Ramaraju, 2019, Section 4.1) using MCS lock	$\mathcal{O}(1)$	$\mathcal{O}(n+F)$	unbounded	No
Golab and Ramaraju's transformation for bounding RMR complexity (Golab and Ramaraju, 2019, Section 4.2) using MCS lock	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	No
Golab and Hendler's arbitration tree using k-port MCS lock*† (Golab and Hendler, 2017)	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	No
Jayanti and Joshi's wait-free recoverable lock (Jayanti and Joshi, 2017)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	No
Jayanti, Jayanti and Joshi's arbitration tree using k-port MCS lock (Jayanti et al., 2019)	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	No
Chan and Woelfel's array based recoverable lock‡ (Chan and Woelfel, 2020)	$\mathcal{O}(1)$	$\mathcal{O}(F+1)$	unbounded	No
Katzan and Morrison's abortable and recoverable lock (Katzan and Morrison, 2021)	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	$\mathcal{O}(\log n/\log\log n)$	Yes
The recoverable lock in this work	$\mathcal{O}(1)$	$\mathcal{O}(\sqrt{F+1})$	$\mathcal{O}(\log n/\log\log n)$	Yes

n: the number of processes in the system
F: the number of failures in the system
$*$: It has been recently shown in (Jayanti et al., 2019) that the algorithm is prone to deadlocks
†: RMR complexity measures only hold for the CC model
‡: RMR complexity is constant in the amortized case

Space consumption of RME algorithms poses another challenge, especially for certain embedded systems with low memory capacity. An RME algorithm that consumes (heap) memory dynamically needs to take steps in order to reclaim said memory. Memory reclamation, in single process systems without failures, follows a straightforward pattern. The process allocates "nodes" dynamically, consumes it, and then frees it once it has no more need of this node. Freed nodes may later be reused (as part of a different allocation) or returned to the operating system. However, due to some programmer error, if a node that is freed is later accessed by the process in the context of the previous allocation, it may cause some serious damage to the program and the operating system as well. In the context of multi-process systems, when a process frees a node, one may face the same issue without any programmer error. Even if the process that frees the node is able to guarantee that it will not access that node again, there may exist another process that is just about to access or dereference the node in the context of the old allocation.

In order to avoid the aforementioned error, a separate memory reclamation service is employed to ensure safe memory access and timely memory restoration. The course of memory reclamation in order to free a node proceeds in two steps. First, a node is **retired** by the underlying algorithm. The underlying algorithm needs to guarantee that once a node is retired, any *new* operations by any process may no longer be able to get access to the node. Second, the memory reclamation service **reclaims** the node once it is deemed to be "safe", *i.e.*, any operation by any process can no longer obtain access to the node[3]. A memory reclamation service is only responsible to provide a safe reclamation of a node once it is retired. On the other hand, in the case of RME, the responsibility of retiring the node is typically on the RME algorithm that needs to consume the memory reclamation service.

Prior works on memory reclamation (Michael, 2004; Fraser, 2004; McKenney and Slingwine, 1998; Arcangeli et al., 2003; Brown, 2015; Wen et al., 2018) provide safe memory

[3]in the context of the previous allocation

reclamation in the absence of failures, but are not trivially suited to account for failures and subsequent recovery using persistent memory. Moreover, most works focus on providing memory reclamation in the context of concurrent data structures. Design of memory reclamation for RME algorithms provides certain advantages due to the blocking nature of the RME problem.

1.1 Contributions

Two different models of failure are considered in this work, (a) the independent failure model, and (b) the system-wide failure model, depending on whether processes fail independently or simultaneously. The main contributions of this work include, but not limited to,

(i) An adaptive solution to RME under the independent failure model

(ii) A solution for memory reclamation/garbage collection for RME algorithms under both independent and system-wide failure models, and

(iii) An optimal solution to RME under the system-wide failure model

The first contribution (Dhoked and Mittal, 2020) is a *novel* framework that transforms any algorithm that solves the RME problem for the independent failure model into an algorithm that is simultaneously *adaptive* to (a) the number of processes competing for the lock, *as well as* (b) the number of failures that have occurred in the recent past, while having the same worst-case RMR complexity as that of the base RME algorithm. In particular, assume that the worst-case RMR complexity of a critical section request in the underlying RME algorithm is $R(n)$, where n denotes the number of processes in the system. Then, this framework yields an RME algorithm for which the worst-case RMR complexity of a critical section request is given by $\mathcal{O}(\min\{\ddot{c}, \sqrt{F+1},\ R(n)\})$, where $\ddot{c}$ denotes the maximum number of requests that are simultaneously active while the given request is outstanding (referred to

as *point contention*) and F denotes the number of failures that have occurred in the *recent past* of the given request (referred to as *failure-density*). Note that the RMR complexity of a request in this algorithm is high only when both point contention and failure-density of the request are high.

In addition to preserving the safety and liveness properties of the underlying RME algorithm (mutual exclusion, starvation freedom and bounded critical section re-entry), this transformation also maintains its other desirable properties such as *bounded exit* and *bounded recovery* as applicable. Roughly speaking, an RME algorithm satisfies the bounded exit property if a process is able to leave its critical section within a bounded number of its own steps unless it fails. It satisfies the bounded recovery property if a process is able to recover from a failure within a bounded number of its own steps unless it fails again.

The key idea behind the approach of the framework is to use a solution to a *weaker* variant of the RME problem, referred to as *weakly RME problem*, in which a failure may cause the mutual exclusion property to be violated temporarily albeit in a controlled manner, repeatedly as a "filter" to limit contention and achieve adaptability. To that end, this work presents an efficient algorithm for the weakly RME problem that has *optimal* RMR complexity of only $\mathcal{O}(1)$.

It is also shown that this RME algorithm is fair; in particular, it satisfies a variant of the first-come-first-served (FCFS) property, referred to as *CI-FCFS*. Intuitively, CI-FCFS guarantees first-come-first-served (FCFS) order among requests provided their recent past is failure-free.

The next major contribution of this work (Dhoked and Mittal, 2021) is a *novel* memory reclamation algorithm that bounds the space complexity of the weakly RME algorithm by $\mathcal{O}(n^2)$. This, in turn, bounds the space complexity of the algorithm generated by the framework by $\mathcal{O}(n^2 \cdot R(n) + S(n))$, where $S(n)$ denotes the space complexity of the underlying RME algorithm. This algorithm is general enough that it can be plugged into any RME

algorithm, while preserving all correctness properties and most desirable properties of the algorithm. On the other hand, it is specific enough to take advantage of assumptions made by RME algorithms. In particular, this algorithm may be blocking, but it is suitable in the context of the RME due to the very blocking nature of the RME problem.

This approach derives from prior works of EBR (Fraser, 2004) (epoch based reclamation) and QSBR (McKenney and Slingwine, 1998) (quiescent state based reclamation). However, unlike EBR and QSBR, where the memory consumption may grow unboundedly due to a slow process, this algorithm guarantees a bounded memory consumption.

Finally, for the system-wide failure model, this work presents an optimal solution where the worst-case RMR complexity of a critical section request is given by $\mathcal{O}(1)$. This solution does not rely upon an external failure detector and satisfies a fairness property known as failure robust fairness. Roughly speaking, an algorithm that satisfies failure robust fairness, guarantees every active process, an unbiased entry into the critical section despite continuous failures. Additionally, the solution only requires $\mathcal{O}(n^2)$ space, with no dynamic memory allocation.

The solution to RME under the system-wide failure model is divided into three major ideas. First, an $\mathcal{O}(1)$ RME algorithm that satisfies all correctness and desirable properties except bounded critical section re-entry and failure robust fairness properties. Lastly, two transformations that transforms this algorithm to add the bounded critical section re-entry and failure robust fairness properties. These two transformations do not incur any additional RMR overhead. Moreover, both the transformations are general enough to be employed by any algorithm that solves the RME problem under the system-wide failure model.

1.2 Organization of the Text

The rest of the text is organized as follows.

In Chapter 2, the system model is described and the RME problem is formally defined. A detailed description of the related work is given in Chapter 3. The weaker variant of the RME problem and its properties are defined in Section 4.1.

A highly efficient solution to the weaker variant of the RME problem with constant RMR complexity is presented in Section 4.2. In Section 4.3, this work presents a framework to transform any given RME algorithm into a new RME algorithm that preserves the worst-case RMR complexity of the original RME algorithm but has lower RMR complexity in the absence of failures. This transformation uses a solution to the weaker variant of the RME problem as a building block. Applying this transformation recursively, a new transformation is created in Section 4.4 that preserves the worst-case RMR complexity of the original RME algorithm and whose performance degrades sub-linearly with the number of "recent" failures (specifically, in proportion to $\sqrt{F}$). This transformation achieves the desired RMR complexity for each of the three scenarios mentioned earlier (as shown in Table 1.1). Chapter 5 offers a discussion on the fairness guarantees provided by the RME algorithms presented in this work (weakly as well as strongly recoverable).

In Chapter 6, this work presents a recoverable broadcast object with constant RMR complexity that allows a process to notify other processes that are waiting on it to reach a certain point in its execution. The broadcast object is used in the design of a recoverable memory reclamation algorithm that bounds the space-complexity of the (recursive) framework while maintaining its RMR complexity in Chapter 7. The broadcast object is also used in Chapter 8 to design a RME algorithm under the system-wide failure model that incurs $\mathcal{O}(1)$ RMRs and satisfies bounded critical section re-entry and failure robust fairness properties.

Finally, Chapter 9 presents the conclusions of this work and outlines directions for future research.

CHAPTER 2

SYSTEM MODEL AND PROBLEM FORMULATION

The model for this work closely follows the model used by Golab and Ramaraju (Golab and Ramaraju, 2019) in their work on recoverable mutual exclusion (RME).

2.1 System model

This work considers an asynchronous shared-memory system consisting of n unreliable processes labeled $p_1, p_2, \ldots, p_n$. Shared memory is used to store variables that can be accessed by any process. Besides shared memory, each process also has its own private memory that is used to store variables that can only be accessed by that process (*e.g.*, program counter, CPU registers, execution stack, *etc.*). Processes can only communicate by performing read, write and read-modify-write (RMW) instructions on shared variables. Processes are not assumed to be reliable and may fail.

A system execution is modeled as a sequence of process steps. In each step, some process either performs some local computation affecting only its private variables or executes one of the available instructions (read, write or RMW) on a shared variable or fails. Processes may run at arbitrary speeds and their steps may interleave arbitrarily. In any execution, between two successive steps of a process, other processes can perform an unbounded but finite number of steps.

2.2 Failure model

This work assumes the *crash-recover* failure model. A process may fail at any time during its execution by crashing. A crashed process recovers eventually and restarts its execution. A crashed process does not perform any steps until it has restarted. A process may fail multiple times, and multiple processes may fail concurrently.

Note that, upon restarting after a failure, the state of the lock as well as the underlying application utilizing the lock needs to be restored to a proper condition. This work focuses only on the recovery of the internal structure of the lock. Restoring the application state to its proper condition (using logs and/or persistent memory) is assumed to be the responsibility of the programmer and is beyond the scope of this work (Golab and Ramaraju, 2019; Golab and Hendler, 2017; Jayanti et al., 2019).

This work assumes that, upon crashing, a process loses the contents of its private variables, including but not limited to the contents of its program counter, CPU registers (general and special purpose) and execution stack. However, the contents of the shared variables remain unaffected and are assumed to persist despite any number of failures. When a crashed process restarts, all its private variables are reset to their initial values.

Processes that have crashed are difficult to distinguish from processes that are running arbitrarily slow. However, assume that every process is *live* in the sense that a process that has not crashed eventually executes its next step and a process that has crashed eventually recovers. In this work, a failure is considered to be associated with a single process. If a failure causes multiple processes to crash, unless otherwise stated, each process crash is treated as a separate failure.

2.3 Process execution model

A process execution is modeled using two types of computations, namely *non-critical section* and *critical section*. A critical section refers to the part of the application program in which a process needs to access shared resources in isolation. A non-critical section refers to the remainder of the application program.

If multiple processes access and modify shared resource(s) concurrently, it may lead to race conditions which may prevent the application from working properly and may possibly have disastrous consequences. To avoid such race conditions, a lock (or a mutual exclusion

(ME) algorithm) is used to enable each process to execute its critical section in isolation. At most one process can hold the lock at any time, and a process can execute its critical section only if it is holding the lock. The lock can be granted to another request only after the process (more specifically, request) holding the lock releases it after completing its critical section. Hereafter, the terms "mutual exclusion algorithm", "ME algorithm" and "lock" are used interchangeably.

Algorithm 2.1: Process Execution Model

```
1 while true do
2 |   Non-Critical Section (NCS)
3 |   Recover()
4 |   Enter()
5 |   Critical Section (CS)
6 |   Exit()
7 end while
```

The execution model of a process with respect to a lock is depicted in Algorithm 2.1. As shown, a process repeatedly executes the following five segments in order: NCS, `Recover`, `Enter`, CS and `Exit` (lines 2 to 6). The first segment, referred to as NCS, models the steps executed by a process in which it only accesses variables outside the lock. The second segment, referred to as `Recover`, models the steps executed by a process to perform any cleanup required due to past failures and restore the internal structure of the lock to a consistent state. The third segment, referred to as `Enter`, models the steps executed by a process to acquire the lock so that it can execute its critical section in isolation. The fourth segment, referred to as CS, models the steps executed by a process in the critical section where it accesses shared resources in isolation. Finally, the fifth segment, referred to as `Exit`, models the steps executed by a process to release the lock it acquired earlier in `Enter` segment.

It is assumed that, in the NCS segment, a process does not access any part of the lock or execute any computation that could potentially cause a race condition. Moreover, in the

`Recover`, `Enter` and `Exit` segments, a process accesses shared variables pertaining to the lock (and the lock only).

A process may crash at any point during its execution, including while executing the NCS, `Recover`, `Enter`, CS or `Exit` segment. Every crashed process upon restarting starts its execution from the beginning of the loop shown in algorithm 2.1, specifically from the beginning of the NCS segment. Note that any steps executed by a process to recover the application state are not explicitly modeled here. Specifically, both NCS and CS segments may consist of code in the beginning to recover relevant portions of the application state. Hereafter, this execution model only considers steps taken by a process during its `Recover`, `Enter` or `Exit` segments.

A history that follows the above execution model and its assumptions is known as a well-formed history. In the rest of this work, unless otherwise mentioned, only well-formed histories are considered.

In the rest of the text, the phrase "acquiring a recoverable lock," means "executing `Recover` and `Enter` segments (in order) of the associated RME algorithm." Likewise, the phrase "releasing a recoverable lock," means "executing the `Exit` segment of the associated RME algorithm."

Definition 2.1 (passage)**.** *A* passage *of a process is defined as the sequence of steps executed by the process from when it begins executing* `Recover()` *segment to either when it finishes executing the corresponding* `Exit()` *segment or experiences a failure, whichever occurs first.*

Definition 2.2 (failure-free passage)**.** *A passage of a process is said to be* failure-free *if the process has successfully executed the* `Recover`*,* `Enter` *and* `Exit` *segments of that passage without experiencing any failures.*

Definition 2.3 (super-passage)**.** *A* super-passage *of a process is a maximal non-empty sequence of consecutive passages executed by the process, where only the last passage of the process in the sequence can be failure-free.*

Definition 2.4 (failure-free super-passage)**.** *A super-passage of a process is said to be* failure-free *if it consists of exactly one passage.*

For ease of exposition, if a process has a super-passage in-progress (*i.e.*, not completed), then the process is said to have a pending or outstanding *request* for a critical section or super-passage. Note that a process may execute multiple failure-free CS segments during its super-passage. This is because a super-passage is considered to be complete only after the process has completed a failure-free passage, which includes `Exit` segment. All these CS segments are considered to be associated with the *same* request.

2.4 Problem definition

The RME problem is defined in terms of histories and its properties. A *history* is a collection of steps taken by processes. A process p is said to be *live* in a history H if H contains at least one step by p. It is assumed that every critical section is finite.

Definition 2.5 (fair history)**.** *A history H is said to be* fair *if (a) it is finite, or (b) if it is infinite and every live process in H either executes infinitely many steps or stops taking steps after a failure-free passage.*

Designing a recoverable mutual exclusion (RME) algorithm involves designing `Recover`, `Enter` and `Exit` segments such that the following correctness properties are satisfied.

Mutual Exclusion (ME) For any history H, at most one process is in its CS at any point in H.

Starvation Freedom (SF) Let H be an infinite fair history in which every process fails only a finite number of times during each of its super-passage. If a process p leaves the NCS segment in some step of H, then p eventually enters its CS segment[1].

[1]This form of starvation freedom is based on Jayanti, Jayanti and Joshi's definition (Jayanti et al., 2019)

Bounded Critical Section Re-Entry (BCSR) For any history H, if a process p crashes inside its CS segment, then, until p has reentered its CS segment at least once, any subsequent execution of `Recover` and `Enter` segments by p either complete within a bounded number of p's own steps or ends with p crashing.

Note that mutual exclusion is a safety property, and starvation freedom is a liveness property. The bounded critical section reentry is a safety as well as a liveness property. If a process fails inside its CS, then a shared object or resource (*e.g.*, a shared data structure) may be left in an inconsistent state. The bounded critical section reentry property allows such a process to "fix" the shared resource if needed before any other process can enter its CS (*e.g.*, (Golab and Ramaraju, 2019; Golab and Hendler, 2017; Jayanti et al., 2019)). This property loosely assumes that the CS segment is idempotent in the sense that, within the same super-passage, executing the CS segment multiple times, partially in some cases, is equivalent to executing it once completely. Our correctness properties are the same as those used in (Golab and Ramaraju, 2019; Golab and Hendler, 2017; Jayanti et al., 2019) and have been stated here for the sake of completeness. In addition to the correctness properties, it is also desirable for an RME algorithm to satisfy the following additional properties.

Bounded Exit (BE) For any infinite history H, any execution of the `Exit` segment by any process p either completes in a bounded number of p's own steps or ends with p crashing.

Bounded Recovery (BR) For any infinite history H, any execution of the `Recover` segment by process p either completes in a bounded number of p's own steps or ends with p crashing.

Note that the ME, SF, BCSR, BE, or BR properties are defined for histories. An algorithm is said to satisfy these properties if and only if every history generated by the algorithm satisfies the corresponding property.

2.5 Performance measures

The performance of RME algorithms is measured in terms of the number of *remote memory references (RMRs)* incurred by the algorithm during a *single* passage (*i.e.*, `Recover`, `Enter` and `Exit` segments). The definition of a remote memory reference depends on the memory model implemented by the underlying hardware architecture. In particular, this work considers the two most popular shared memory models:

Cache Coherent (CC) The CC model assumes a *centralized* main memory that acts as a global, shared store of variables. In addition, each process has a *local* cache memory. Whenever a process accesses (reads or writes) a variable, a copy of the variable is stored in the cache of that process. Any subsequent access to that variable is serviced using that cached copy as long as the copy is still valid. A write access causes other cached copies of the variable to be either updated or invalidated. In addition, it may also cause the copy stored in the main memory to be updated. In this model, a step incurs an RMR if it causes the contents of any of the caches to be modified (including invalidation).

Distributed Shared Memory (DSM) The DSM model assumes that the main memory is *partitioned* into multiple memory modules with one module attached to every process. A process can access a variable stored on any memory module, be it local or remote. However, accessing a variable stored on its local module is much faster than accessing the one stored on a remote module. In this model, a step incurs an RMR if it involves accessing a variable stored on a remote memory module.

Figures 2.1 and 2.2 demonstrate a pictorial abstraction of these models. In the rest of this work, if not explicitly specified, the RMR complexity measure of an algorithm applies to *both CC and DSM models*.

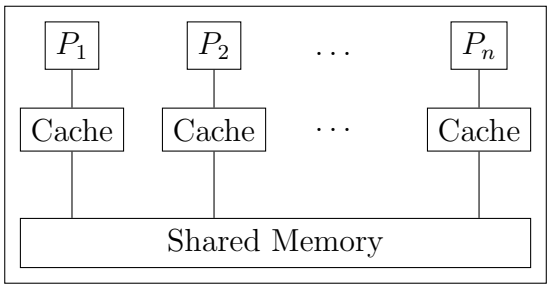

Figure 2.1. Pictorial representation of the cache coherent model

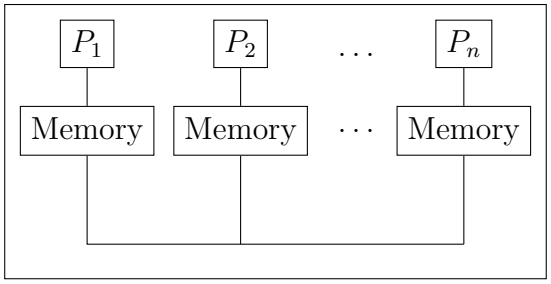

Figure 2.2. Pictorial representation of the distributed shared memory model

In this work, the RMR complexity of an RME algorithm is analyzed under three scenarios: (a) in the absence of failures (failure free RMR complexity), (b) in the presence of F failures (limited failures RMR complexity), and (c) in the presence of an unbounded number of failures (arbitrary failures RMR complexity).

Let $g(n, F)$ be a function of F and n, where $n \geq 1$ is the number of processes in the system and $F \geq 0$ is the number of failures that have occurred so far. Assume that $g(n, F)$ is a monotonically non-decreasing function of n and F since this function is going to be used to represent the worst-case RMR complexity of an RME algorithm. This work identifies several desirable performance measures applicable to an RME algorithm. To that end, first the following concepts are defined for the function $g(n, F)$.

PM 1. (Constantness) In the absence of failures, the function has a constant value independent of n. Formally, $g(n, 0) = \mathcal{O}(1)$.

PM 2. (Adaptiveness) In order to capture adaptiveness, a function $\Delta(n)$ is defined as:

$$\Delta(n) = \Big| \{F \mid g(n, F) < g(n, F + 1)\} \Big|$$

With limited number of failures, this work identifies three different cases.

(a) The function has a non-trivial dependence on F. Formally, $\Delta(n) = \Omega(1)$.

(b) The function has a strong dependence on F. Formally, $\Delta(n) = \omega(1)$.

(c) The function has a strong and sub-linear dependence on F. Formally, $\Delta(n) = \omega(1)$ and $g(n, F) = o(F)$.

PM 3. (Boundedness) With arbitrary number of failures, this work identifies two different cases:

(a) The function is finite-valued even as F tends to infinity. Formally, $\lim_{F \to \infty} g(n, F)$ is finite-valued.

(b) The function is bounded by a sub-logarithmic function of n. Formally, $\forall F$: $g(n, F) = \mathcal{O}(\log n / \log \log n)$.

Note that PM 2(b) implies PM 2(a), PM 2(c) implies PM 2(b) and PM 3(b) implies PM 3(a).

Consider an RME algorithm $\mathcal{A}$ and let $g(n, F)$ denote the *best known* bound on the worst-case RMR complexity of $\mathcal{A}$. Based on the performance measures satisfied by $g(n, F)$, $\mathcal{A}$ is called as:

1. Based on adaptiveness
   - *non-adaptive* if it does not satisfy PM 2(a).
   - *semi-adaptive* if it satisfies PM 2(a).
   - *adaptive* if it satisfies PM 2(b) (hence also PM 2(a).
   - *super-adaptive* if it satisfies PM 2(c) (hence also PM 2(b) and PM 2(a).
2. Based on boundedness
   - *unbounded* if it does not satisfy PM 3(a).
   - *bounded* if it satisfies PM 3(a).
   - *well-bounded* if it satisfies PM 3(b).

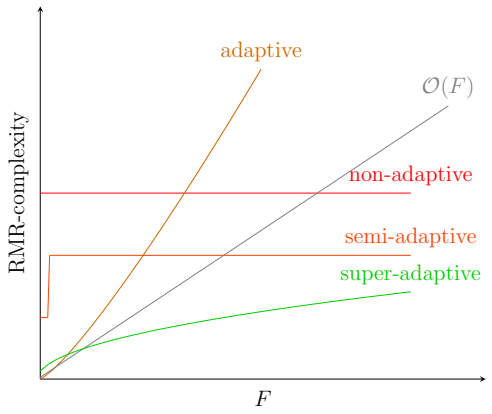

Figure 2.3. RMR complexity of adaptive algorithms

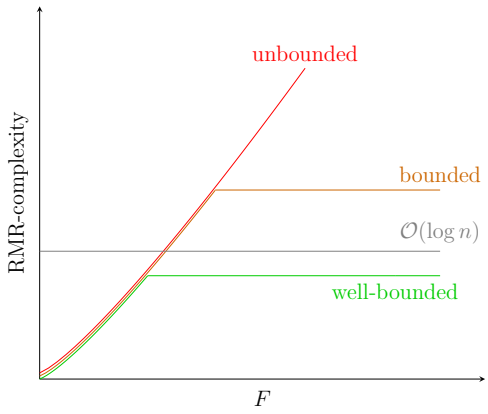

Figure 2.4. RMR complexity of bounded algorithms

Figure 2.3 shows how the RMR complexity of algorithms increases with number of failures. Figure 2.4 shows a comparison of RMR complexity of algorithms based on boundedness as the number of failures increase.

A comparison of the known RME algorithms with respect to the above performance measures PM 1 to PM 3 is shown in Table 2.1. As shown in Table 2.1, other existing RME algorithm are either non-adaptive, semi-adaptive or unbounded adaptive. The algorithm (Chapter 4) presented in this work is the first bounded-adaptive, as well as, well-bounded super-adaptive RME algorithm known for either memory model.

Note that the concept of adaptiveness (PM 2) depends on F, the number of failures in the system. As time passes, the number of failures may eventually grows unboundedly large. Often, certain failures that are too far in the past, do not affect the performance of RME algorithm. To quantify the impact of failures on the performance of an RME algorithm, this work uses the notion of *consequence interval* of a failure. Roughly speaking, it is used to capture the maximum duration for which the impact of the failure may be felt in the system. It is related to, but different from, the notion of k-failure-concurrent passage defined by Golab and Ramaraju in (Golab and Ramaraju, 2019). A more detailed comparison of the two notions is deferred to Section 5.2.

Table 2.1. Comparison of known solutions to the RME problem with respect to the six performance measures.

Algorithm	**Performance Measure**						**Classification**
	PM 1	**PM 2(a)**	**PM 2(b)**	**PM 2(c)**	**PM 3(a)**	**PM 3(b)**	
Golab and Ramaraju's transformation for recoverability (Golab and Ramaraju, 2019, Section 4.1) using MCS lock	✓	✓	✓	✗	✗	✗	unbounded adaptive
Golab and Ramaraju's transformation for bounding RMR complexity (Golab and Ramaraju, 2019, Section 4.2) using MCS lock	✓	✓	✗	✗	✓	✗	bounded semi-adaptive
Golab and Hendler's arbitration tree using k-port MCS lock* (Golab and Hendler, 2017)	✗	✗	✗	✗	✓	✓	well-bounded non-adaptive
Jayanti and Joshi's wait-free recoverable lock (Jayanti and Joshi, 2017)	✗	✗	✗	✗	✓	✗	bounded non-adaptive
Jayanti and Joshi's arbitration tree using k-port MCS lock (Jayanti et al., 2019)	✗	✗	✗	✗	✓	✓	well-bounded non-adaptive
Chan and Woelfel's array based recoverable lock† (Chan and Woelfel, 2020)	✗	✓	✓	✗	✗	✗	unbounded adaptive
Katzan and Morrison's abortable and recoverable lock (Katzan and Morrison, 2021)	✗	✗	✗	✗	✓	✓	well-bounded non-adaptive
Recoverable locks from this work [Chapters 4 and 8]	✓	✓	✓	✓	✓	✓	well-bounded super-adaptive

$*$: it has been shown in (Jayanti et al., 2019) that the algorithm is prone to deadlocks

†: RMR complexity is constant in the amortized case

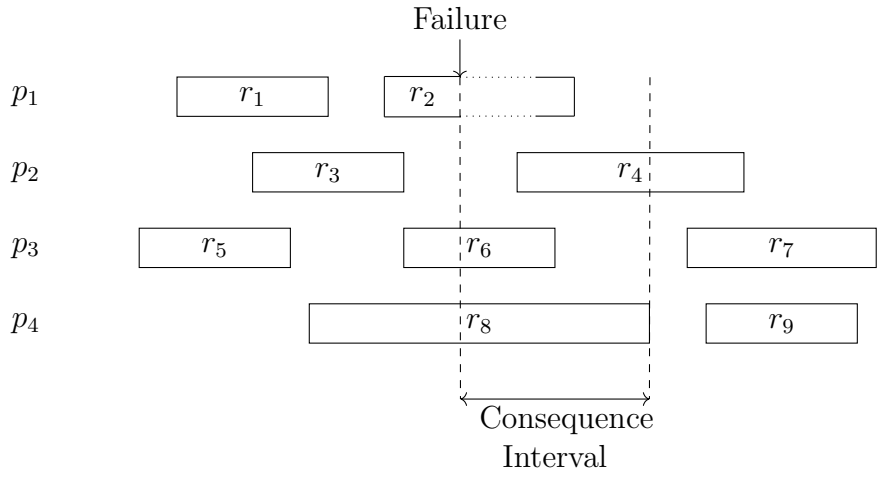

Figure 2.5. Illustration of the consequence interval of a failure

Definition 2.6 (consequence interval)**.** *The* consequence interval *of a failure in a history H is defined as the interval in time that starts from the onset of the failure and extends to the point in time when either every super-passage that started before this failure occurred in H has completed or the last step in H is performed, whichever happens earlier.*

An illustration of the consequence interval of a failure is provided in Figure 2.5. Intuitively, a failure is considered to be *recent* with respect to a time t if t is contained in the consequence interval of the failure. The following notion captures the "concentration" of failures in the recent past of a request.

Definition 2.7 (failure-density)**.** *The* failure-density *of a request is defined as the number of failures whose consequence interval overlaps with the super-passage of the request.*

This work presents an RME algorithm that is adaptive not only to failure-density (the main focus) but also to contention. To show the latter, this work uses the well-known notion of point contention defined as follows:

Definition 2.8 (point contention)**.** *The* point contention *of a request is defined as the maximum number of super-passages that are simultaneously in-progress at any point during the super-passage associated with the request.*

In the remainder of the text, unless stated otherwise, the term "adaptive" is used in the context of failure-density.

2.6 Synchronization Primitives

The RME algorithms described in this work assume that, in addition to regular read and write instructions, the system also supports atomic *compare-and-swap* (`CAS`) and *fetch-and-store* (`FAS`) read-modify-write (RMW) instructions.

Algorithm 2.2: Various read-modify-write instructions

```
 8 Function CAS(address, old, new)
 9 begin
10     if old = *address then
11         *address = new
12         return TRUE
13     else
14         return FALSE
15     end if
16 end
17 Function FAS(address, new)
18 begin
19     old = *address
20     *address = new
21     return old
22 end
```

A *compare-and-swap (*`CAS`*)* instruction takes three arguments: *address*, *old* and *new*; it compares the contents of a memory location (*address*) to a given value (*old*) and, only if they are the same, modifies the contents of that location to a given new value (*new*). It returns TRUE if the contents of the location were modified and FALSE otherwise.

A *fetch-and-store (*`FAS`*)* instruction takes two arguments: *address* and *new*; it replaces the contents of a memory location *address* with a given value *new* and returns the old contents of that location.

A pseudocode of these instructions is given in algorithm 2.2. Both instructions are commonly available in many modern processors such as Intel 64 (Intel, 2016) and AMD64 (AMD, 2019). Note that an RME algorithm may only one or none of these instructions.

CHAPTER 3

RELATED WORK

3.1 Recoverable Mutual Exclusion for Independent failures

Bohannon *et al.* (Bohannon et al., 1995, 1996) were the first ones to investigate the RME problem. However, their system model is different from the one assumed in this work. Specifically, in their system model, at least one process is reliable while other processes may be unreliable. Once an unreliable process fails, it never restarts. The reliable process is responsible for continuously monitoring the health of all other processes, and, upon detecting that an unreliable process has failed during its passage, it performs recovery by "fixing" the lock. The two RME algorithms differ in the way they implement the lock; the one in (Bohannon et al., 1995) uses test-and-set instruction whereas the one in (Bohannon et al., 1996) uses MCS queue-based algorithm.

Golab and Ramaraju formally defined the RME problem in (Golab and Ramaraju, 2016). Their formalization, especially the system model, has served as the basis for the subsequent work in this area, including this work. In (Golab and Ramaraju, 2016, 2019), Golab and Ramaraju also presented four different RME algorithms—a 2-process RME algorithm and three n-process RME algorithms. The first algorithm is based on Yang and Anderson's lock (Yang and Anderson, 1995), and is used as a building block to design an n-process RME algorithm. Both RME algorithms use only read, write and comparison-based primitives. The worst-case RMR complexity of the 2-process algorithm is $\mathcal{O}(1)$ whereas that of the resultant n-process algorithm is $\mathcal{O}(\log n)$. Both RME algorithms have optimal RMR complexity because, as shown in (Attiya et al., 2008; Anderson and Kim, 2002; Yang and Anderson, 1995), any mutual exclusion algorithm that uses only read, write and comparison-based primitives has worst-case RMR complexity of $\Omega(\log n)$. The remaining two algorithms are unbounded adaptive (with $f(x) = x$) and semi-adaptive (with $g(x) = x$), respectively (where f and g are as per the definitions of adaptivity and boundedness respectively from chapter 2).

Later, Golab and Hendler (Golab and Hendler, 2017) proposed an RME algorithm with sub-logarithmic RMR complexity of $\mathcal{O}(\log n / \log \log n)$ only under the CC model using MCS queue based lock (Mellor-Crummey and Scott, 1991a) as a building block. Note that MCS uses the `FAS` instruction, which is *not* a comparison-based RMW instruction, and thus the result does not violate the previously mentioned lower bound. Their algorithm does not satisfy the bounded exit property. Moreover, it has been shown to be vulnerable to starvation (Jayanti et al., 2019).

Ramaraju showed in (Ramaraju, 2015) that it is possible to design an RME algorithm with $\mathcal{O}(1)$ RMR complexity provided the hardware provides a special RMW instruction to swap the contents of two arbitrary locations in memory atomically. Unfortunately, at present, no known hardware supports such an instruction.

In (Jayanti and Joshi, 2017), Jayanti and Joshi presented an RME algorithm with $\mathcal{O}(\log n)$ RMR complexity. Their algorithm satisfies bounded (wait-free) exit and FCFS (first-come-first-served) properties.

In (Jayanti et al., 2019), Jayanti, Jayanti and Joshi proposed an RME algorithm that has sub-logarithmic RMR complexity of $\mathcal{O}(\log n / \log \log n)$. This is the best known RME algorithm as far as the worst-case RMR complexity is concerned that also satisfies bounded recovery and bounded exit properties.

Using a weaker version of starvation freedom, Chan and Woelfel (Chan and Woelfel, 2020) present a novel solution to the RME problem that incurs a constant number of RMRs in the amortized case, but its worst case RMR complexity may be unbounded.

A useful extension to the RME problem is when a process may decide to *abort* its request for critical section; this extension is referred to as the *abortable RME problem*. Recently, Katzan and Morrison (Katzan and Morrison, 2021) and Jayanti and Joshi (Jayanti and Joshi, 2019) have proposed efficient algorithms for solving the problem under both CC and DSM models. The algorithm by Jayanti and Joshi uses f-arrays and has $\mathcal{O}(\log n)$ RMR complexity (Jayanti and Joshi, 2019). On the other hand, the algorithm by Katzan and Morrison uses

a k-port abortable RME algorithm as a building block to design a sub-logarithmic abortable RME algorithm with RMR complexity of $\mathcal{O}(\log n / \log \log n)$ (Katzan and Morrison, 2021).

Recently, Chan and Woelfel have also proved a lower bound of $\Omega(\log n / \log \log n)$ on the RMR complexity of any RME algorithm using currently available hardware instructions and practical word size of $\Theta(\log n)$ (Chan and Woelfel, 2021).

3.2 Memory Reclamation

In (Michael, 2004), Michael uses *hazard pointers*, a wait-free technique for memory reclamation that only requires a bounded amount of space. Hazard pointers are special shared pointers that protect nodes from getting reclaimed. Such nodes can be safely accessed by the processes. Any node that is not protected by a hazard pointer is assumed to be safe to reclaim. Being shared pointers, hazard pointers are expensive to read and update.

In (Fraser, 2004), Fraser devises a technique called epoch based reclamation (EBR). As the name suggests, the algorithm maintains an epoch counter e and three limbo lists corresponding to epochs $e-1$, e and $e+1$. The main idea is that nodes retired in epoch $e-1$ are safe to be reclaimed in epoch $e+1$. This approach is not lock-free and a slow process may cause the size of the limbo lists to increase unboundedly.

In (McKenney and Slingwine, 1998), Mckenney and Slingwine present the RCU framework where they demonstrate the use of quiescent state based reclamation (QSBR). QSBR relies on detecting quiescent states and a grace period during which each thread passes through at least one quiescent state. Nodes retired before the grace period are safe to be reclaimed after the grace period. In (Arcangeli et al., 2003), Arcangeli et. al. make use of the RCU framework and QSBR reclamation for the System V IPC in the Linux kernel.

In (Brown, 2015), Brown presents DEBRA and DEBRA+ reclamation schemes. DEBRA is a distributed extension of EBR where each process maintains its individual limbo lists instead of shared limbo lists and epoch computation is performed incrementally. DEBRA+

relies on hardware assistance from the operating system to provide signalling in order to prohibit slow or stalled processes to access reclaimed memory.

Note that most of the aforementioned works on memory reclamation assume that processes do not crash. An equivalent algorithm for these algorithms under the *crash-recover* model is non-trivial.

3.3 Recoverable Mutual Exclusion for System-Wide Failures

In (Golab and Hendler, 2018), Golab and Hendler formally define the RME problem for system-wide failures (all processes fail and restart). Using this stronger assumption of system-wide failures, they design an $\mathcal{O}(1)$ RMR solution for the RME problem. Additionally, they present two algorithms to add BCSR and FRF properties to any existing RME algorithm for the system-wide failure model, each of which incur an $\mathcal{O}(1)$ RMR overhead.

They achieve these results using an external failure detector. This external failure detector is used to detect the current "epoch" number of the system. Roughly speaking, the epoch number counts the number of system-wide failures. Each process is assumed to know the epoch number upon recovering from failure.

CHAPTER 4

AN ADAPTIVE APPROACH TO RME

This chapter presents the framework to transform any algorithm that solves the RME problem into an adaptive one. The key idea is to leverage the gap between the lower bounds of the ME ($\mathcal{O}(1)$) and RME ($\mathcal{O}(\log n / \log \log n)$) problems; such that the RMR complexity gradually reaches the worst case as number of failures increase.

The approach of the framework is to use a solution to a *weaker* variant of the RME problem, referred to as *weakly RME problem*, in which a failure may cause the mutual exclusion property to be violated temporarily albeit in a controlled manner, repeatedly as a "filter" to limit contention and achieve adaptability. The weaker variant of the RME problem and its properties are defined in Section 4.1.

In Section 4.2, this work presents an efficient algorithm for the weakly RME problem that has *optimal* RMR complexity of only $\mathcal{O}(1)$. In Section 4.3, this work presents a framework to transform any given RME algorithm into a new RME algorithm that preserves the worst-case RMR complexity of the original RME algorithm but has lower RMR complexity in the absence of failures. This transformation uses a solution to the weaker variant of the RME problem as a building block. Applying this transformation recursively, a new transformation is created in Section 4.4 that preserves the worst-case RMR complexity of the original RME algorithm and whose performance degrades sub-linearly with the number of "recent" failures (specifically, in proportion to $\sqrt{F}$).

In particular, assume that the worst-case RMR complexity of a critical section request in the underlying RME algorithm is $R(n)$, where n denotes the number of processes in the system. Then, this framework yields an RME algorithm for which the worst-case RMR complexity of a critical section request is given by $\mathcal{O}(\min\{\ddot{c}, \sqrt{F+1},\ R(n)\})$, where $\ddot{c}$ denotes the maximum number of requests that are simultaneously active while the given request is outstanding (referred to as *point contention*) and F denotes the number of failures that have

occurred in the *recent past* of the given request (referred to as *failure-density*). Note that the RMR complexity of a request in this algorithm is high only when both point contention and failure-density of the request are high.

4.1 Weak Recoverability

To design a well-bounded super-adaptive RME algorithm, this work uses a solution to the *weaker* variant of the RME problem as a *building block* in which a failure may cause the ME property to be violated albeit only temporarily and in a controlled manner. This variant is referred to as the *weakly recoverable mutual exclusion problem.*

To formally define how long a violation of the ME property may last, the (previously defined) notion of consequence interval of a failure is used here.

Definition 4.1 (weakly recoverable mutual exclusion)**.** *An algorithm is a* weakly recoverable mutual exclusion algorithm *if, in addition to starvation freedom, it satisfies the following property: for any history H, if two or more processes are in their critical sections simultaneously at some point in H, then that point overlaps with the consequence interval of some failure.*

Roughly speaking, a weakly RME algorithm satisfies the ME property as long as no failure has occurred in the "recent" past. Hereafter, to avoid confusion, this work sometimes refers to the traditional recoverable mutual exclusion problem (respectively, algorithm) as defined in Section 2.4 as *strongly recoverable mutual exclusion* problem (respectively, algorithm).

The BE, BR and BCSR properties defined earlier in Section 2.4 are applicable to weakly RME problem as well.

Section 4.2 demonstrates that it is possible to design an optimal weakly recoverable mutual exclusion algorithm using existing hardware instructions whose worst-case RMR complexity is only $\mathcal{O}(1)$ under both CC and DSM models. In contrast, as proven in (Chan

and Woelfel, 2021), the worst-case RMR complexity of any strongly RME algorithm is $\Omega(\log n / \log \log n)$ under both CC and DSM models. This *gap* is exploited in this chapter to design an RME algorithm that is adaptive and bounded. To prove that the algorithm is also well-bounded super-adaptive, some additional properties of the weakly RME algorithm are utilized.

Note that not all failures may cause the ME property to be violated when using a weakly RME algorithm. To that end, the notion of sensitive instruction of an algorithm is defined.

Definition 4.2 (sensitive instruction)**.** *An instruction σ of a weakly RME algorithm is said to be* sensitive *if there exists a finite history H that satisfies the following conditions: (a) it contains exactly one failure in which a process crashes immediately after executing said instruction σ, and (b) two or more processes are in their CS at the end of H; it is said to be* non-sensitive *otherwise.*

Definition 4.3 (unsafe failure)**.** *A failure is said to be* unsafe *with respect to a weakly RME algorithm if it involves a process crashing while (immediately before or after) performing a sensitive instruction with respect to the algorithm; it is said to be* safe *otherwise.*

Note that, by definition, every instruction of a strongly RME algorithm is a non-sensitive instruction. As a result, every failure is safe with respect to a strongly RME algorithm.

The next notion limits the "degree" of violation (of the ME property) by a weakly RME algorithm if and when it occurs.

Definition 4.4 (responsive weakly recoverable mutual exclusion)**.** *A weakly recoverable mutual exclusion algorithm is* responsive *if, for all $k \geq 1$, it satisfies the following property: for any history H, if at least $k+1$ processes are in their critical sections simultaneously at some point in H, then that point overlaps with the consequence intervals of at least $\Omega(k)$ (unsafe) failures.*

4.1.1 Composite recoverable locks

The properties defined above are with respect to a *single* weakly recoverable lock. In order to construct a well-bounded super-adaptive (strongly) recoverable lock with desired performance characteristics, multiple weakly recoverable locks are used. A lock is called *composite* if it is employs one or more (weakly or strongly recoverable) locks. Composite locks might have several possible structures. For instance, the `Enter()` segment of one lock could be contained in the `Enter()` or CS segment of another lock or the CS segment of one lock may be contained in the NCS segment of another lock. An example of a composite lock is a lock based on the tournament algorithm (Golab and Ramaraju, 2019).

Note that, when multiple locks are involved, the notions defined earlier, namely consequence interval, sensitive instruction and unsafe failure, become *relative* to the specific lock. For example, a failure will have a different consequence interval with respect to each lock. An instruction may be sensitive with respect to one lock but non-sensitive with respect to another. Thus, in a composite lock, a failure may be unsafe with respect to one or more weakly recoverable locks.

Definition 4.5 (locality property)**.** *A composite (weakly or strongly) recoverable lock is said to satisfy the* locality property *if, for any instruction σ, σ is sensitive with respect to* at most one *of its component weakly recoverable locks.*

A composite lock whose component locks are all strongly recoverable trivially satisfies satisfies the locality property.

4.2 An Optimal Weakly Recoverable Lock

This section presents a weakly recoverable lock whose RMR complexity is $\mathcal{O}(1)$ per passage for all three failure scenarios under both CC and DSM models. This lock is based on the well-known MCS queue-based (non-recoverable) lock (Mellor-Crummey and Scott, 1991a).

The original lock did not satisfy the bounded exit property. Dvir and Taubenfeld proposed an extension to the original algorithm in (Dvir and Taubenfeld, 2017) to make the `Exit` segment wait-free. The augmented MCS lock, which satisfies bounded-exit property, is further extended to make it weakly recoverable.

4.2.1 Original MCS queue-based lock

Processes in the MCS mutual exclusion algorithm use queue nodes to synchronize their executions of CS segments. The algorithm maintains a first-come-first-served (FCFS) queue of outstanding requests using a linked-list of their associated nodes. A node contains two fields: (a) *next*, a reference to its successor node in the queue (if any), and (b) *locked*, a boolean variable used by a process to spin while waiting for its turn to enter its critical section. The queue itself is represented using a shared variable *tail* that contains a reference to the last node in the queue if non-empty and **null** otherwise.

To acquire the lock, a process first initializes its queue node by setting its *next* and *locked* fields to **null** and TRUE, respectively. It then appends the node to the queue by performing an `FAS` instruction on *tail* using the reference to its own node as an argument (to the instruction). Note that the instruction returns the contents of *tail* just before it is modified. If the return value is **null**, then it indicates that the lock is free and the process has successfully acquired the lock. If not, then it indicates that the lock is not free and the return value is the reference to the predecessor of the process' own node in the queue. In that case, it notifies the owner of the predecessor node of its presence. To that end, it stores the reference to its own node in the *next* field of the predecessor node, thereby creating a forward link between the two nodes. It then starts spinning on the *locked* field of its own node waiting for it to be reset to FALSE by the owner of the predecessor node as part of releasing the lock.

To release the lock, a process first tries to reset the *tail* variable to **null** (if *tail* still contains the reference to this process' node) using a `CAS` instruction. If the instruction returns TRUE,

then it implies that the queue does not contain any more outstanding requests and the lock is now free. On the other hand, if the instruction returns FALSE, then it implies that the queue contains at least one outstanding request and its own node is guaranteed to have a successor. It then waits until the *next* field of its own node contains a valid reference (a non-null value) indicating that a link has been created between its own node and its successor. Finally, it follows this link and resets the *locked* field in its successor node to FALSE.

4.2.2 Adding bounded exit property

The MCS original algorithm, as described above does not satisfy the bounded-exit property since a process leaving its critical section may have to wait until a link between its own node and its successor has been created.

To achieve the bounded-exit property, the original algorithm is augmented with a mechanism that allows a leaving process to notify the process next in line to acquire the lock that the lock is now free. To that end, a process, on leaving its critical section, attempts to store a special value (*e.g.*, reference to its own node) in the *next* field of its own node using a `CAS` instruction. Likewise, a link is also created using a `CAS` instruction instead of a simple write instruction as in the original algorithm. Both `CAS` instructions are designed to succeed only if the *next* field contains **null** value, thereby ensuring that the *next* field can only be modified once.

Thus, if the `CAS` instruction performed by a process leaving its critical section returns FALSE, then that process can conclude that the forward link has already been created. It then follows this link and resets the *locked* field of its successor node. On the other hand, if the `CAS` instruction performed by a process trying to create the link returns FALSE, then that process can infer that the lock is free and that it now holds the lock.

With this modification, unlike in the original MCS algorithm, a process cannot always reuse its own node for the next request after releasing the lock.

4.2.3 Adding weak recoverability

A pseudocode of the weakly recoverable lock is given in algorithms 4.1 and 4.2. The pseudocode uses the following shared variables. The first variable, *tail*, contains the address of the last node in the queue if the queue is non-empty and **null** otherwise. The next three variables, *state*, *mine* and *pred*, are arrays with one entry for each process. For the DSM model, the i-th entry of each of these array variables is local to process p_i. The variable $state[i]$ contains process p_i's current state with respect to the lock (explained later). The variable $mine[i]$ contains the address of the queue node associated with process p_i's most recent request. The variable $pred[i]$ contains the address of the predecessor node, if any, of process p_i after its node has been appended to the queue.

The *state* of a process with respect to a lock has five possible values, namely FREE, INITIALIZING, TRYING, INCS and LEAVING. At the beginning of a super-passage, the $state[i]$ of a process p_i, starts from FREE. It is changed to INITIALIZING at the end of `Recover` (line 69). It is changed to TRYING after 1. p_i has initialized $mine[i]$ with the address of a new node (line 37), 2. initialized the two fields of $mine[i]$ (lines 39 and 40) and finally 3. updated $pred[i]$ by setting it equal to $mine[i]$ (line 41). It is changed to INCS after p_i has acquired the lock. It is changed to LEAVING when p_i starts releasing the lock, and is changed to FREE again after p_i finishes releasing the lock and completes the super-passage.

This algorithm has only one sensitive instruction, namely the one involving the `FAS` instruction (line 46). Recall that a process uses this instruction to append its own node to the queue and also obtain the address of its predecessor node. If a process fails immediately after executing this instruction (line 46) but before it is able to persist its return value to the shared memory (line 47), there is no easy way to recover this address (of the predecessor) based on the current knowledge of the failed process. The queue continues to grow beyond this node, but it would be disconnected from the previous part of the queue, thereby creating

Algorithm 4.1: Pseudocode of process p_i for a weakly recoverable MCS-based lock with wait-free exit (Part 1 of 2)

```
   /* Data structure to represent nodes
      in a queue */
23 struct QNode {
       /* location used for spinning
          while waiting to enter CS */
24     locked: boolean variable
       /* reference to successor node */
25     next: reference to QNode
26 };
27 shared variables
       /* reference to the last node in
          the queue */
28     tail: reference to QNode, initially null
       /* remaining variables are
          arrays, whose i-th entry in the
          DSM model is local to process
          p_i */
       /* state of the process with
          respect to the lock */
29     state: array [1...n] of integer
        variables, all elements initially FREE
       /* state[i] ∈ {FREE, INITIALIZING,
          TRYING, INCS, LEAVING} */
       /* reference to my own node */
30     mine: array [1...n] of references to
        QNode, all elements initially null
       /* reference to predecessor's
          node */
31     pred: array [1...n] of references to
        QNode, all elements initially null
32 end
33 Function Enter()
34 begin
35     if state[i] = INITIALIZING then
36         if mine[i] = null then
37             mine[i] ← GETNEWNODE()
38         end if
           /* initialize QNode */
39         mine[i].next ← null
40         mine[i].locked ← TRUE
           /* to later determine whether
              FAS result was persisted */
41         pred[i] ← mine[i]
           /* advance the state */
42         state[i] ← TRYING
43     end if
44     if state[i] = TRYING then
45         if pred[i] = mine[i] then
               /* add my node to tail */
46             QNode* result ← FAS(tail,
                mine[i])
               /* persist FAS result */
47             pred[i] ← result
48         end if
49     if state[i] = TRYING then
50         if pred[i] ≠ null then
               /* create a forward link */
51             CAS(pred[i].next, null, mine[i])
52             if (pred[i].next = mine[i]) then
                   /* wait for pred */
53                 await not (mine[i].locked)
54             end if
55         end if
           /* advance the state */
56         state[i] ← INCS
57     end if
58 end
```

one more sub-queue. For an example, please refer to Figure 4.1. Thus, an unsafe failure occurs if a process fails immediately after executing the instruction at line 46.

Algorithm 4.2: Pseudocode of process p_i for a weakly recoverable MCS-based lock with wait-free exit (Part 2 of 2).

```
59 Function Recover()
60 begin
61     if state[i] = TRYING then
           /* the following two
              references are guaranteed
              to differ if FAS result was
              persisted */
62         if pred[i] = mine[i] then
               /* unsafe failure
                  potential; abort the
                  attempt */
63             Cleanup()
64         end if
65     else if state[i] = LEAVING then
           /* cleanup module was left
              incomplete */
66         Cleanup()
67     end if
68     if state[i] = FREE then
           /* advance the state */
69         state[i] ← INITIALIZING
70     end if
71 end
72 Function Exit()
73 begin
74     Cleanup()
75 end
76 Function Cleanup()
77 begin
       /* advance the state */
78     state[i] ← LEAVING
79     if mine[i] ≠ null then
           /* remove my node from the
              queue if it has no
              successor */
80         CAS(tail, mine[i], null)
           /* may not have a successor */
81         CAS(mine[i].next, null, mine[i])
           /* Check if successor exists
              in the queue */
82         if mine[i].next ≠ mine[i] then
               /* signal the successor to
                  stop spinning */
83             mine[i].next.locked ← FALSE
84         end if
85         pred[i] ← null
86         mine[i] ← null
87     end if
88     RETIRELASTNODE()
       /* advance the state */
89     state[i] ← FREE
90 end
```

If a process detects that it may have failed while executing the (FAS) instruction, it "relinquishes" its current node, informs its successor (if any) that the lock is now "free" using the wait-free signalling mechanism described earlier, retires the current node and retries acquiring the lock using a new node. This potentially creates multiple queues (or sub-queues) which may allow multiple processes to execute their critical sections concurrently, thereby violating the ME property. A node is relieved (either after failure or completion of critical section) by executing the Cleanup module (lines 77 to 89). This module can be invoked

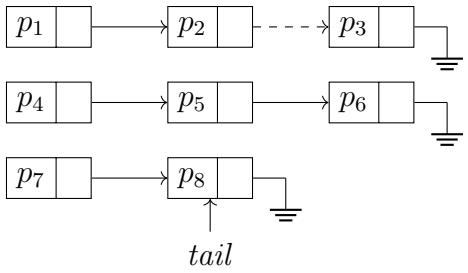

Figure 4.1. Processes $p_1 \ldots p_8$ successfully append their nodes to the tail of the queue using an `FAS` instruction. Processes p_4 and p_7 failed to store the outcome of the `FAS` instruction to persistent memory, and are thus unable to set the next field of the nodes of p_3 and p_6. Process p_3 has captured the address of the node of p_2 and is about to set the corresponding next field on the node of p_2. Effectively, three sub-queues are created due to failures of p_4 and p_7.

either from `Recover` or `Exit`. All other instructions of this algorithm, except for `FAS`, are non-sensitive. This is achieved by using the following techniques.

First, a process does not use the outcome of the `CAS` instruction used to modify the *next* field of a node (line 51 and line 81). After performing the `CAS` instruction on the *next* field, it reads the contents of the field again and determines its next step based on what it read. Note that, once initialized, the *next* field can only be modified once. This makes two steps involving the `CAS` instruction on the *next* field as *idempotent*; the effect of performing the `CAS` instruction multiple times if interrupted due to failures is the same as performing it once.

Second, portions of `Recover` and `Enter` are enclosed in if-blocks to be executed conditionally. Intuitively, the guard of an if-block represents the pre-condition that needs to hold before its body can be executed. The outermost if-blocks use guards based on the current *state* of the process, which is advanced only at the end of each block. The inner if-blocks use guards based on other variables. Except for the if-block containing the `FAS` instruction (which constitutes a sensitive instruction), all other if-blocks are idempotent; and can be executed repeatedly, without any adverse impact, starting from the evaluation of the guard,

if interrupted due to failures (lines 61 to 64, lines 65 to 67, lines 68 to 70, lines 35 to 43 and lines 50 to 55). Note that if the guard of an if-block does not hold, its body is not executed.

Third, similar to the case of the *next* field, a process does not use the outcome of the `CAS` instruction used to modify the *tail* pointer of the queue during the `Cleanup` module (line 80). After performing the `CAS` instruction on the *tail* pointer, irrespective of the outcome of the instruction, it blindly executes the remainder of the steps pertaining to signalling the successor node (lines 81 to 84). If the node has no successor, then the steps are redundant, but have no adverse impact even if the node has already been removed from the queue by an earlier `CAS` instruction.

The `GetNewNode` and `RetireLastNode` are "hooks" for the memory reclamation algorithm described later (Chapter 7). This algorithm guarantees that once a node is used by a process, the `RetireLastNode` is executed to completion at least once before invoking the `GetNewNode` method.

The algorithm for the weakly recoverable lock described in this section is referred to as WR-Lock. Note that lines 35 to 48, which do not contain any loop, constitute the *doorway* of WR-Lock whereas lines 49 to 57 constitute its *waiting* room.

4.2.4 Analysis and Proofs of Correctness

This work proves that WR-Lock is a responsive weakly recoverable mutual exclusion algorithm. The following some concepts and notations are defined to be used in the proofs. All definitions use the notion of *current* time.

The WR-Lock uses a queue node to synchronize among processes and serialize their critical section executions. In the absence of failures, a process *allocates* a new node at the beginning of a passage and *relieves* it at the end of that passage. However, in the presence of failures, a node allocated during one passage may be relieved during a different passage. Every node is *owned* by a (unique) process. Let $owner(u)$ denote the owner process of the

node u. A node becomes *active* once it has been *appended* to the queue by storing its address in the *tail* pointer using an `FAS` instruction (line 46). It follows from code inspection that:

Proposition 4.1. *A node is appended to the queue at most once.*

Note that active nodes can be *ordered* based on the sequence in which they were appended to the queue. The last active node to be appended to the queue is referred to as the *tail* node.

If a node v is appended to the queue immediately after a node u, then then v is the *successor* of u and there is an *edge* from v to u. The expression $succ(u)$ denotes the successor of node u and $edge(v, u)$ denotes the edge from node v to node u provided $v = succ(u)$. Intuitively, the edge $edge(v, u)$ captures the *wait-for* dependency that v has on u. In particular, $owner(v)$ cannot enter its CS until $owner(u)$ has relieved u. The set of active nodes along with the set of edges, as defined above, form a chain referred to as *global chain*.

A node is said to have been *relieved* if its owner has executed either the `CAS` instruction at line 81 successfully (to signal its successor process) or the write instruction at line 83 (to unlock the successor node).

A node is said to be *admissible* if it is active but not relieved.

The following proposition follows from code inspection:

Proposition 4.2. *A process can own at most one admissible node in the global chain. Further, if a process owns an admissible node in the global chain, then it has a super-passage in progress.*

Consider two nodes u and v such that $v = succ(u)$ and let $p_v = owner(v)$. The $edge(v, u)$ is said to be *admissible* if the following conditions hold. First, u is admissible. Second, when p_v appended v to the queue, the `FAS` instruction returned the address of u. Third, since p_v appended v to the queue, either p_v has not failed or has stored the address of u in the persistent memory by executing the write instruction at line 47. Intuitively, if the edge

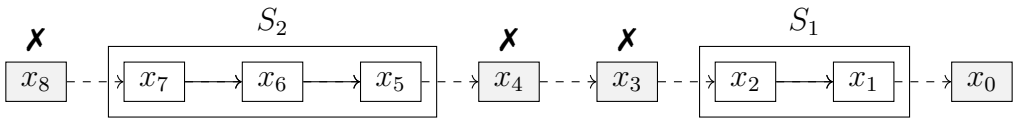

Figure 4.2. An illustration of the notions used in the correctness proof of WR-Lock. The global chain contains nine nodes, labeled x_0 to x_8. In the figure, (a) unshaded nodes represent admissible nodes while shaded nodes represent non-admissible nodes, and (b) solid lines represent admissible edges while dashed lines represent non-admissible edges. A node that was abandoned by its owner due to an unsafe failure has been labeled with '✗'. Nodes that belong to the same maximal admissible fragment or sub-queue have been enclosed in a rectangle. As shown, the global chain contains two sub-queues, given by $S_1 = \langle x_1, x_2 \rangle$ and $S_2 = \langle x_5, x_6, x_7 \rangle$. Nodes x_5 and x_7 are the front and rear nodes, respectively, of the sub-queue S_2. Finally, node x_8 is the tail node of the global chain.

$edge(v, u)$ is not admissible, then either $owner(u)$ has already relieved u or $owner(v)$ has lost the address of u.

Any *non-empty* connected set of nodes in the global chain is called a *fragment*. A fragment is said to be *admissible* if all its nodes as well as all its edges are admissible. An admissible fragment is said to be *maximal* if it is not strictly contained in another admissible fragment.

A *sub-queue* is defined as a maximal admissible fragment. Note that each sub-queue has a *front* node (no outgoing admissible edge) and a *rear* node (no incoming admissible edge).

Figure 4.2 gives an illustration of the concepts defined so far.

The following propositions follow from the above concepts.

Proposition 4.3. *Any two distinct sub-queues are node disjoint.*

Proposition 4.4. *If a process owns a node in some sub-queue, then it has a super-passage in progress. Moreover, if a process is in its CS, then it owns the front node of some sub-queue.*

Proposition 4.5. *Assume that every process fails only a finite number of times during each of its super-passage. If a process owns the front node of some sub-queue, then it eventually relieves the node.*

Proposition 4.6. *If a process owns the front node of some sub-queue and never fails thereafter in its super-passage currently in progress, then the super-passage eventually completes.*

Note that every active non-*tail* node has an *incoming* edge. Consider an admissible non-*tail* node u and let $v = succ(u)$. Note that, just after v is appended to the queue, the edge from v to u is admissible. Thus, if the edge from v to u is not admissible now, then it implies that $owner(v)$ failed before it was able to store the address of u in the persistent memory. If this happens, then the edge from v to u (whose status changed from admissible to non-admissible) is said to be *disrupted* by an unsafe failure.

The following proposition follows from above concepts.

Proposition 4.7. *The incoming edge to an admissible non-tail node can only be disrupted by an unsafe failure. Moreover, an unsafe failure can disrupt at most one edge in the global chain.*

Based on the above propositions, the following results hold about WR-Lock.

Lemma 4.8. *Given a history H and a non-negative integer k, if at least k processes are in their critical sections simultaneously, then the global chain contains at least k distinct sub-queues.*

Proof. The lemma follows from propositions 4.3 and 4.4. □

Theorem 4.9. *Given a history H, and a non-negative integer k, if $k+1$ processes are in their critical sections simultaneously, then there exists at least k unsafe failures whose consequence intervals are still in-progress.*

Proof. It follows from Lemma 4.8 that the global chain contains at least $k+1$ distinct sub-queues. Clearly, the rear node of all sub-queues, except possibly for one, is a *non-tail* node. It follows from proposition 4.7 that the incoming edge of every non-*tail* rear node was disrupted by a *unique* unsafe failure.

Now, consider an arbitrary sub-queue whose rear node, say u, is a non-*tail* node. Note that the global chain contains at least k such nodes. Let $v = succ(u)$ and let f be the unsafe

failure that disrupted $edge(v, u)$. Clearly, u is appended to the queue *before* v, and v is appended to the queue *before* f occurred. Using proposition 4.2, it implies that $owner(u)$ has a super-passage currently in progress that started before f occurred. In other words, the consequence interval of f is still active. □

Theorem 4.10. WR-Lock *satisfies the SF property.*

Proof. Assume, on the contrary, that WR-Lock does not satisfy the SF property. Then, there exists a fair history H in which every process fails only a finite number of times during each of its super-passage; and some super-passage, say Π, belonging to a process, say p, never completes. Let σ denote the first passage that p starts after recovering from its last failure in Π. By assumption, p never fails during σ. Clearly, p eventually advances to line 48 during σ at which point it is guaranteed to own an admissible node, say u, in the global chain. By applying proposition 4.5 repeatedly, it can be inferred that u eventually becomes the front node of its sub-queue. By applying proposition 4.6, it can be inferred that Π eventually completes—a contradiction. □

Theorem 4.11. WR-Lock *satisfies the BCSR property.*

Proof. Assume that some process, say p_i, fails while executing its CS, implying that $state[i] =$ InCS at the time of failure. Upon restarting, p_i executes `Recover` and `Enter` in that order. As the code inspection shows, since $state[i] =$ InCS, p_i only evaluates a constant number of if-conditions, all of which evaluate to FALSE, and is therefore able to proceed to the CS in a bounded number of its own steps. □

It follows from Theorems 4.9 to 4.11 that

Theorem 4.12. WR-Lock *is a responsive weakly recoverable mutual exclusion algorithm.*

Theorem 4.13. WR-Lock *satisfies the BR and BE properties.*

Proof. As the code inspection shows, execution of `Recover` and `Exit` does not involve any loops. Thus, a process can complete executing `Recover` and `Exit` within a bounded number of its own steps. Hence, WR-Lock satisfies the BR and BE properties. □

Theorem 4.14. *The RMR complexity of each of passage (*`Recover`*,* `Enter` *and* `Exit`*) of* WR-Lock *is* $\mathcal{O}(1)$*.*

Proof. As the code inspection shows, executions of `Recover` and `Exit` do not contain any loops and only contain a constant number of steps. The execution of `Enter`, however contains one loop at line 53, but otherwise contains a constant number of steps. This loop involves waiting on a **boolean** variable until it becomes TRUE and the variable can be written to only once. This incurs only $\mathcal{O}(1)$ RMRs in the CC model. In the DSM model, this variable is mapped to a location in local memory module. Hence, the RMR complexity of an execution of `Enter` is also $\mathcal{O}(1)$ in both CC and DSM models. □

4.3 A well-bounded semi-adaptive RME algorithm

This section describes a framework that uses other types of recoverable locks with certain properties as building blocks to construct a lock that is not only strongly recoverable but also well-bounded super-adaptive under both CC and DSM models. The construction of the (well-bounded super-adaptive) lock is split into two parts (Sections 4.3 and 4.4). The first part (Section 4.3) describes a basic framework to transform a bounded non-adaptive strongly recoverable lock to a bounded semi-adaptive strongly recoverable lock. This framework is later extended in Section 4.4 to make the lock super-adaptive while ensuring that it stays strongly recoverable and bounded.

The basic framework is based on the one used by Golab and Ramaraju in (Golab and Ramaraju, 2019, Section 4.2) to construct a strongly recoverable lock that is semi-adaptive. Specifically, in their framework, Golab and Ramaraju use two different types of strongly recoverable locks, referred to as base lock and auxiliary lock, along with two other components

to build another strongly recoverable lock, referred to as the *target* lock. The constructed *target* lock is bounded semi-adaptive based on the base lock that is unbounded adaptive and the auxiliary lock that is non-adaptive. They achieve this by customizing the base lock so that, upon detecting a failure, processes can abort their attempts and reset the (base) lock. In the presence of failures (even a single failure), the RMR complexity of the *target* lock is dominated by the overhead of aborting the attempt to acquire the base lock and then resetting the base lock, thereby making the lock semi-adaptive. In the rest of the text, the term "*target* lock" refers to the (strongly recoverable) lock being built.

4.3.1 Building blocks

This algorithm uses four different components as building blocks.

- *Filter lock:* A responsive weakly recoverable lock that provides mutual exclusion in the absence of failures. This algorithm uses an instance of the lock proposed in Section 4.2, which has $\mathcal{O}(1)$ RMR complexity for all three failure scenarios.
- *Splitter:* Used to split processes into *fast* or *slow* paths. If multiple processes navigate the splitter concurrently (which would happen only if an unsafe failure has occurred with respect to the filter lock), only one of them is allowed to take the fast path and the rest are diverted to the slow path. In other words, the splitter is *biased*. Intuitively, it can be viewed as a strongly recoverable *try* lock. It is implemented using an atomic integer and a `CAS()` instruction, which has $\mathcal{O}(1)$ RMR complexity for all three failure scenarios.
- *Arbitrator lock:* A *dual-port* strongly recoverable lock. Each port corresponds to a side. The two sides are referred to as Left and Right. At any time, at most one process should be allowed to attempt to acquire the lock from any side. However, *any two* of the n processes can compete to acquire the lock. This algorithm uses the implementation of the dual-port RME algorithm proposed by Golab and Ramaraju

in (Golab and Ramaraju, 2019, Section 3.1) (a transformation of Yang and Anderson's mutual exclusion algorithm to add recoverability), which has $\mathcal{O}(1)$ RMR complexity for all three failure scenarios.

- *Core lock:* a (presumably non-adaptive) strongly recoverable lock that assures mutual exclusion among processes taking the slow path. An instance of any of the existing RME algorithms may be used.

Note that the *target* lock satisfying BCSR, BR and BE properties is contingent upon the filter, arbitrator and core locks satisfying BCSR, BR and BE properties.

4.3.2 The execution flow

In order to acquire the *target* lock, a process proceeds as follows. It first waits to acquire the filter lock. Once granted, it navigates through the splitter trying to enter the fast path. If successful, it then attempts to acquire the arbitrator lock from the LEFT side. If one or more failures occur that are unsafe with respect to the filter lock, then multiple processes may acquire the filter lock simultaneously. If this results in contention at the splitter, then all but one processes are diverted to the slow path. If a process is forced to take the slow path, it attempts to acquire the core lock. Once granted, it then waits to acquire the arbitrator lock from the RIGHT side. Finally, once the process has successfully acquired the arbitrator lock, it is deemed to have acquired the *target* lock as well, and is now in the CS of the *target* lock.

A pictorial representation of the execution flow is depicted in Figure 4.3. Note that the pictorial representation depicts the two sides of the arbitrator lock as left and bottom, which actually correspond to the LEFT side and the RIGHT side of the arbitrator lock respectively.

In the absence of failures, every process takes the fast path, albeit one at a time. However, some processes do take the fast path even if their super-passage overlaps with the consequence interval of an unsafe failure with respect to the filter lock. Note that at most one process

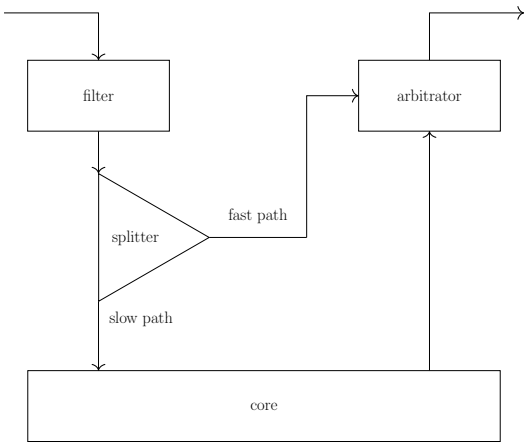

Figure 4.3. A pictorial representation of the framework.

can take the fast path at a time and at most one process can hold the core lock at a time. Any process that takes the fast path will always attempt to acquire the arbitrator lock from the LEFT side. Any process that takes the slow path and acquires the core lock will always attempt to acquire the arbitrator lock from the RIGHT side. Since the core lock is strongly recoverable, at most one process will try to acquire the arbitrator lock from each side at a time.

In order to release the *target* lock, a process simply releases its component locks in the reverse order in which it acquired them: the arbitrator lock, followed by the core lock (in case the process took the slow path), followed by the splitter and finally the filter lock.

The RMR complexity of the fast path is given by the sum of the RMR complexities of the filter lock, the splitter and the arbitrator lock. On the other hand, the RMR complexity of the slow path is given by the sum of the RMR complexities of the filter lock, the splitter, the core lock and the arbitrator lock.

For ease of exposition, the following terminology is used. Before a process is assigned a particular path, it is referred to as a *normal* process. It is classified as a *fast* process if it takes the fast path and a *slow* process otherwise. A slow process becomes a *medium-slow* process once it acquires the core lock.

Algorithm 4.3: Pseudocode of process p_i for the framework to design a semi-adaptive lock (Part 1 of 2)

```
91  shared variables
        /* filter lock */
92      F: n-process weakly recoverable lock
        /* splitter; stores the process
           that owns the fast path */
93      owner: integer variable, initially 0
        /* core lock */
94      C: n-process strongly recoverable lock
        /* arbitrator lock */
95      A: n-process dual-port strongly
          recoverable lock
        /* path of the process; in the
           DSM model, the i-th entry is
           local to process p_i */
96      path: array [1...n] of boolean
          variables, initially FAST
        /* path[i] ∈ {FAST, SLOW} */
97  end
98  definitions
99      side(path) = { LEFT   if path = FAST
                     { RIGHT  otherwise
100 end
101 Function Recover()
102 begin
        /* To ensure fair histories for
           each of the recoverable locks
           (F, C and A), the invocation
           of each Recover segment is
           executed just prior to the
           invocation of the respective
           Enter segment */
103 end
```

The pseudocode is given in Algorithms 4.3 and 4.4. The pseudocode closely follows the above description in text. The splitter is implemented using an integer (shared) variable, namely *owner*. The fast path is occupied if and only if *owner* has a non-zero value, in which case the value refers to the identifier of the process currently occupying the fast path. To take the fast path, a process attempts to store its own identifier in *owner* using a `CAS` instruction provided its current value is zero (line 109). If the attempt fails, the process changes its path type to slow (line 112). Note that a process resets its path type from slow to its default value of fast only after it has executed the `Exit` segment of the core lock at least once without encountering any failure (line 127).

In Golab and Ramaraju's framework, even if a process takes the fast path, it may still incur $\Omega(n)$ RMRs in the presence of even a single failure because of the overhead of aborting an attempt and then resetting the base lock, which is an expensive operation. In this framework, on the other hand, a process taking the fast path incurs only $\mathcal{O}(1)$ RMRs even

Algorithm 4.4: Pseudocode of process p_i for the framework to design a semi-adaptive lock (Part 2 of 2)

```
104 Function Enter()
105 begin
        /* acquire the filter lock */
106     F.Recover()
107     F.Enter()
108     if path[i] ≠ SLOW then
            /* not committed to the slow
               path; try the fast path */
109         CAS(owner, 0, i)
110     end if
111     if owner ≠ i then
            /* unable to take the fast
               path; commit to the slow
               path */
112         path[i] ← SLOW
            /* acquire the core lock */
113         C.Recover()
114         C.Enter()
115     end if
        /* acquire the arbitrator lock */
116     A.Recover(side(path[i]))
117     A.Enter(side(path[i]))
118 end
```

```
119 Function Exit()
120 begin
        /* release the arbitrator lock */
121     A.Exit(side(path[i]))
        /* Check which path was taken
           during Enter  */
122     if path[i] = SLOW then
            /* release the core lock */
123         C.Exit()
124     else
            /* empty the fast path */
125         owner ← 0
126     end if
        /* reset the path to default */
127     path[i] ← FAST
        /* release the filter lock */
128     F.Exit()
129 end
```

with arbitrary number of failures because the RMR complexity of acquiring the filter lock, followed by navigating the splitter to take the fast path and finally acquiring the arbitrator lock is only $\mathcal{O}(1)$ irrespective of the number of failures.

4.3.3 Analysis and Proofs of Correctness

The algorithm described in this section is referred to as SA-LOCK. The doorway of SA-LOCK is given by the doorway of its filter lock. When convenient, $\mathcal{F}$ and $\mathcal{C}$ are used to refer to the instances of filter and core locks, respectively, of SA-LOCK.

Theorem 4.15. SA-LOCK *satisfies the ME property.*

Proof. A process enters the CS segmentof SA-Lock after acquiring the arbitrator lock from one of the sides. The arbitrator lock satisfies the ME property as long as no more than one process attempts to acquire it from either side Left or Right at any time. The splitter ensures that, at any time, at most one process attempts to acquire the arbitrator lock from the Left side. The core lock ensures that, at any time, at most one process attempts to acquire the arbitrator lock from the Right side. Therefore, SA-Lock satisfies the ME property. □

Theorem 4.16. SA-Lock *satisfies the SF property.*

Proof. Assume that every process fails only a finite number of times during each of its super-passage.

All three locks used in the framework, namely filter, core and arbitrator, individually satisfy SF and BCSR properties. Also, navigating the splitter involves executing a constant number of instructions.

Consider an infinite fair history H, a process p that is live in H and a super-passage Π of p in H. Clearly, there exists a time after which p does not fail any more in Π. Let t denote the *earliest* such time. Consider the first passage that p starts after time t.

Note that either the SF or BCSR property of the filter lock guarantees that p eventually leaves the `Enter` segment of the filter lock and enters its CS segment. Process p then completes navigating the splitter within a bounded number of its own steps. Note that, if it was able to enter the fast (respectively, slow) path during an earlier passage of Π, then it is guaranteed to take the fast (respectively, slow) path again in this passage. Similar to the case of filter lock, it can be argued that p is guaranteed to enter the CS segment of the core lock if it takes the slow path during this passage. Likewise, it can be argued that p is guaranteed to enter the CS segment of the arbitrator lock, which in turn implies that p is guaranteed to enter the CS segment of the target lock. □

Theorem 4.17. SA-Lock *satisfies the BCSR property.*

Proof. If some process p_i is in the CS segment of SA-Lock, then it currently holds the filter lock and it either (a) acquired the arbitrator lock from the Left side by taking the fast path or (b) acquired the core lock first and then acquired the arbitrator lock from the Right side by taking the slow path.

If p_i fails in the CS segment of SA-Lock, it determines the path it took by checking the $path[i]$ variable and then retraces the same steps it had executed earlier. Since the filter lock, the core lock as well as the arbitrator lock satisfy the BCSR property, and the fact that the splitter is wait-free, p_i is guaranteed to be able to acquire the requisite locks and reenter the CS segment of SA-Lock within a bounded number of its own steps. Hence, SA-Lock satisfies the BCSR property. □

It follows from Theorems 4.15 to 4.17 that

Theorem 4.18. SA-Lock *is a strongly recoverable lock.*

Theorem 4.19. SA-Lock *satisfies the BR and BE properties.*

Proof. The `Recover` segment of SA-Lock is empty and hence it trivially satisfies the BR property.

As part of the `Exit` segment of SA-Lock, a process executes the `Exit` segment of the arbitrator lock, optionally followed by the `Exit` segment of the core lock, followed by the `Exit` segment of the filter lock. Since each of three locks individually satisfy the BE property, and the splitter is wait-free, it follows that SA-Lock also satisfies the BE property. □

Theorem 4.20 (SA-Lock is bounded semi-adaptive)**.** *The RMR complexity of any passage in a super-passage of* SA-Lock *is* $\mathcal{O}(1)$ *if the failure-density of the super-passage is zero and* $\mathcal{O}(R(n))$ *otherwise, where* $R(n)$ *denotes the worst-case RMR complexity of the core lock for* n *processes.*

Proof. In the absence of failures, *only one* process can successfully acquire the filter lock (Definition 4.1). This process navigates the splitter in $\mathcal{O}(1)$ steps, takes the fast path and acquires the arbitrator lock from the LEFT side (skipping the core lock along the way). The RMR complexity of the arbitrator lock is $\mathcal{O}(1)$. Thus, in this case, the RMR complexity of the *target* lock is given by $\mathcal{O}(1)$.

In the presence of failures, all n processes may be able to successfully acquire the filter lock and proceed to the splitter. Only one of these processes is allowed to take the fast path, which then attempts to acquire the arbitrator lock from the LEFT side. The remaining $(n-1)$ processes are diverted to the slow path and have to acquire the core lock and then acquire the arbitrator lock from the RIGHT side. Thus, in this case, the RMR complexity of the *target* lock is given by $\mathcal{O}(R(n))$. □

Theorem 4.21 (A well-bounded semi-adaptive lock)**.** *Assuming that Jayanti, Jayanti and Joshi's or Katzan and Morrison's RME algorithm (Jayanti et al., 2019; Katzan and Morrison, 2021) is used to implement the core lock. Then, the RMR complexity of any passage in a super-passage of* SA-LOCK *is* $\mathcal{O}(1)$ *if the failure-density of the super-passage is zero and* $\mathcal{O}(\log n / \log\log n)$ *otherwise.*

The rest of this section, is focused on proving an important lemma that is crucial to establish that the lock described in the next section (Section 4.4) is super-adaptive. Recall that the notions of super-passage and unsafe failures are relative to a specific lock.

Intuitively, the set of processes that attempt to acquire the core lock is strictly smaller than the set of processes that attempt to acquire the filter lock. Further, the size of the former set depends on the number of unsafe failures that have occurred with respect to the filter lock. To capture this formally, the following notations are defined. Let $\mathbb{P}(\ell, t)$ denote the set of processes that have a super-passage in progress with respect to lock ℓ at time t. Also, let $\mathbb{UF}(\ell, t)$ denote the set of all failures that are unsafe with respect to the lock ℓ

and whose consequence interval extends at least until time t. Further, if a process p has a super-passage in progress with respect to the target lock at time t, then $\Pi(p,t)$ denotes the super-passage of p with respect to the target lock at time t.

Recall that $\mathcal{F}$ and $\mathcal{C}$ refer to the filter and core locks, respectively, of SA-Lock.

Lemma 4.22. *Consider a time $t_{\mathcal{C}}$ such that $\mid \mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \mid > 0$. Then there exists time $t_{\mathcal{F}}$ with $t_{\mathcal{F}} \leq t_{\mathcal{C}}$ such that the following properties hold.*

(a) $\mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \subsetneq \mathbb{P}(\mathcal{F}, t_{\mathcal{F}})$,

(b) $\forall p \in \mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$, $\Pi(p, t_{\mathcal{F}}) = \Pi(p, t_{\mathcal{C}})$, *and*

(c) $\mid \mathbb{UF}(\mathcal{F}, t_{\mathcal{F}}) \mid \geq \mid \mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \mid$.

Proof. None of the processes in the set $\mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$ was able to take the fast path while navigating the splitter. Let q be the *last* process in $\mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$ to read the contents of the variable *owner* and t denote the time when it performed the read step. Clearly, $t \leq t_{\mathcal{C}}$. Furthermore, let r denote the process whose identifier was stored in *owner* when q read its contents. Let $t_{\mathcal{F}}$ be set to t. Now each property is proven one-by-one.

(i) Due to the arrangement of the locks, each process in the set $\mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$ holds the lock $\mathcal{F}$ at time $t_{\mathcal{F}}$, Moreover, process r also holds the lock $\mathcal{F}$ at time $t_{\mathcal{F}}$. In other words, $\mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \subseteq \mathbb{P}(\mathcal{F}, t_{\mathcal{F}})$, $r \in \mathbb{P}(\mathcal{F}, t_{\mathcal{F}})$ and $r \notin \mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$. Thus the property (a) holds.

(ii) Consider an arbitrary process $s \in \mathbb{P}(\mathcal{C}, t_{\mathcal{C}},)$. Assume, by the way of contradiction, that $\Pi(s, t_{\mathcal{F}}) \neq \Pi(s, t_{\mathcal{C}})$. This means that process s started a *new* super-passage after time $t_{\mathcal{F}}$. Since $s \in \mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$, process s read the contents of the variable *owner* some time after $t_{\mathcal{F}}$ but before $t_{\mathcal{C}}$. This contradicts the choice of $t_{\mathcal{F}}$. In other words, $\Pi(s, t_{\mathcal{F}}) = \Pi(s, t_{\mathcal{C}})$. Since s was chosen arbitrarily, it follows that for each $p \in \mathbb{P}(\mathcal{C}, t_{\mathcal{C}})$, $\Pi(r, t_{\mathcal{F}}) = \Pi(p, t_{\mathcal{C}})$. Thus the property (b) holds.

(iii) Let $\mid \mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \mid = k$. Thus, using property (a), it can be concluded that $\mid \mathbb{P}(\mathcal{F}, t_{\mathcal{F}}) \mid \geq k+1$. It follows from Theorem 4.9 that there exist at least k failures that are unsafe relative to the lock $\mathcal{F}$ and whose consequence interval overlaps with time $t_{\mathcal{F}}$. Thus, $\mid \mathbb{UF}(\mathcal{F}, t_{\mathcal{F}}) \mid \geq k = \mid \mathbb{P}(\mathcal{C}, t_{\mathcal{C}}) \mid$. Thus the property (c) holds.

This establishes the result. □

4.4 A well-bounded super-adaptive RME algorithm

This section presents a framework that extends the framework from Section 4.3 to make the lock super-adaptive while ensuring that it stays strongly recoverable and bounded. It uses the *gap* between the worst-case RMR complexity of implementing a weakly recoverable lock and that of implementing a strongly recoverable lock to achieve its goal. Finally, instantiating the framework with an appropriate well-bounded non-adaptive lock yields the desirable well-bounded super-adaptive lock.

4.4.1 The main idea

The main idea here is to *recursively* transform the core lock using instances of the semi-adaptive transformation from Section 4.3. The core lock is transformed repeatedly upto a height m that is equal to the worst-case RMR complexity of another strongly recoverable lock under arbitrary number of failures. The strongly recoverable lock now becomes the base case of the recursion. For ease of exposition, the core lock in the base case is referred to as the *base lock*.

Let NA-Lock() be a bounded (presumably non-adaptive but does not have to be) strongly recoverable lock, whose worst-case RMR complexity is $\mathcal{O}(R(n))$ for n processes. Let SA-Lock() denote an instance of the semi-adaptive lock described in Section 4.3. And, finally, let BA-Lock() denote the bounded super-adaptive lock being constructed [1]. The

[1] NA-Lock() is the base lock and BA-Lock() is the *target* lock

idea is to create $m = R(n)$ levels of SA-Lock() such that the core lock component of the SA-Lock() at each level is built using another instance of SA-Lock() for up to $m-1$ levels. Further, an instance of NA-Lock() is used at the base level (level m). Let SA-Lock $[i]$ denote the instance of SA-Lock() at level i. Formally,

$$\begin{aligned} \text{BA-Lock} &= \text{SA-Lock}[1] \\ \text{SA-Lock}[i].\text{core} &= \text{SA-Lock}[i+1] \qquad \forall i \in \{1, 2, \ldots, m-1\} \\ \text{SA-Lock}[m].\text{core} &= \text{NA-Lock} \end{aligned}$$

A pictorial representation of the execution flow of the recursive framework is depicted in Figure 4.4.

In order to acquire the *target* lock, a process starts at the first level as a normal process and waits to acquire the filter lock at level 1. It stays on track to become a fast process until an unsafe failure occurs with respect to the filter lock at the first level as a result of which multiple processes may be granted the (filter) lock simultaneously. All of these processes then compete to enter the fast path by navigating through the splitter. The splitter allows only one process to take the fast path at a time, and the rest are diverted to take the slow path. Note that a slow process is created at the first level only if an unsafe failure occurs with respect to the filter lock at the first level. All slow processes at the first level then move to the second level as normal processes. If no further failure occurs, then no slow process is created at the second level and all processes leave this level one-by-one as fast processes with respect to this level. Thus, only $\mathcal{O}(1)$ RMR complexity is *added* to the passages of all the affected processes until the impact of the first failure has subsided. However, if one or more slow processes are created at the second level, then it can be inferred that a *new* unsafe failure must have occurred with respect to the filter lock at the second level. All these slow processes at the second level then move to the third level as normal processes, and so on and so forth. At each level, a slow process, upon either acquiring the base lock or

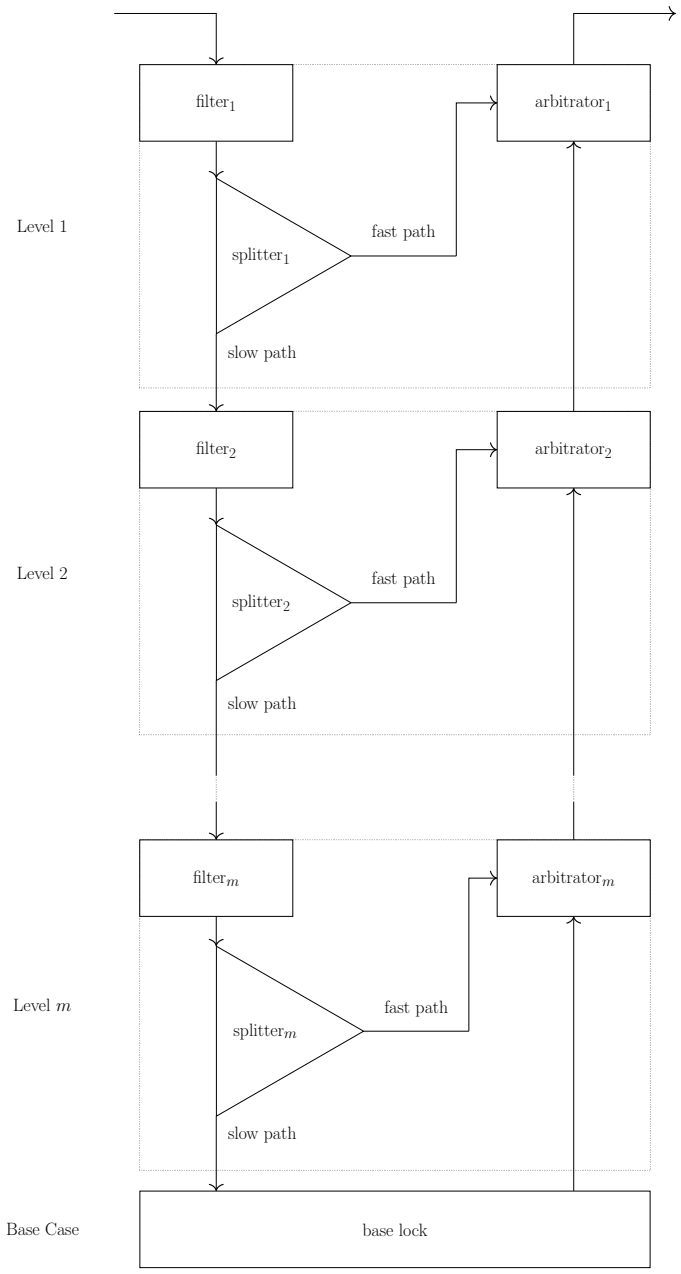

Figure 4.4. A pictorial representation of the recursive framework.

returning from the adjacent higher level (whichever case applies), becomes a medium-slow process. Irrespective of whether a process is classified as fast or medium-slow, it next waits to acquire the level-specific arbitrator lock. Once granted, it either returns to the adjacent lower level or, if at the initial level, is deemed to have successfully acquired the *target* lock.

Note that in this algorithm, at least k unsafe failures are required at any level to force k processes to be "escalated" to the next level. Each level except for the last one would add only $\mathcal{O}(1)$ RMR complexity to the passages of these process, thus making the *target* lock adaptive under limited failures. There is no further "escalation" of slow processes at the base level and a bounded (possibly non-adaptive) strongly recoverable lock is used to manage all slow processes at that point, thus bounding its RMR complexity under arbitrary number of failures as well.

As before, in order to release the *target* lock, a process releases its components locks in the reverse order in which it acquired them.

To prove that the *target* lock is well-bounded super-adaptive, two properties of the framework are utilized, namely, the weakly recoverable lock is responsive, and the *target* lock, which is a composite lock, satisfies the locality property.

4.4.2 Analysis and Proofs of Correctness

Let $\mathcal{F}_i$ and $\mathcal{C}_i$ denote the instances of the filter and core locks, respectively, at level i for $i = 1, 2, \ldots, m$.

Theorem 4.23. *For each level i with $1 \leq i \leq m$,* SA-Lock *[i] is a strongly recoverable lock.*

Proof. The proof is by backward induction on the level number of SA-Lock() starting from level m.

- ☐ Base case (SA-Lock $[m]$ is a strongly recoverable lock)

  Note that SA-Lock $[m]$ = NA-Lock(). By construction, NA-Lock() is a bounded strongly recoverable lock. Thus, SA-Lock $[m]$ is a strongly recoverable lock.

- ☐ Induction hypothesis (SA-Lock $[i+1]$ is a strongly recoverable lock for $1 \leq i < m$)

  To prove that SA-Lock $[i]$ is also a strongly recoverable lock. Note that SA-Lock $[i]$ is an instance of the semi-adaptive lock described in Section 4.3 with SA-Lock $[i+1]$

as its core lock. Since SA-Lock $[i+1]$ is a strongly recoverable lock, it follows from Theorem 4.18 that SA-Lock $[i]$ is also a strongly recoverable lock.

Thus, by induction, it can be concluded that SA-Lock $[i]$ is a strongly recoverable lock for each level $i = 1, 2, \ldots, m$. □

By construction, BA-Lock = SA-Lock [1]. Therefore,

Theorem 4.24. BA-Lock *is a strongly recoverable lock.*

Using induction similar to the one used in Theorem 4.23, one can show that

Theorem 4.25. BA-Lock *satisfies the BR and BE properties.*

To analyze the RMR complexity of a passage, the following results are proven first.

Theorem 4.26. BA-Lock *satisfies the locality property.*

Proof. BA-Lock uses three types of locks, namely filter, arbitrator and base; only the filter lock is weakly recoverable. There is one instance of the filter lock at each level. By construction, the `Enter` segments of any two instances of the filter lock do not overlap. The only sensitive instruction of the filter lock is the `FAS` instruction in its `Enter` segment. Therefore, BA-Lock satisfies the locality property. □

The following proposition follows by the construction of the recursive framework.

Proposition 4.27. *For each level i and time t with $1 \leq i < m$,*

$$\mathbb{P}(\mathcal{C}_i, t) = \mathbb{P}(\text{SA-Lock}(\textit{[i+1]}), t) = \mathbb{P}(\mathcal{F}_{i+1}, t)$$

Note that the set of processes that attempt to acquire the filter lock at any level becomes *progressively smaller* as the level number increases. Furthermore, the number of processes that are escalated to the next level depends on the number of unsafe failures experienced by the filter lock at the current level. This is captured by the next lemma.

Lemma 4.28. *Consider a process p, time t and level x, where $1 \leq x \leq m$, such that process $p \in \mathbb{P}(\mathcal{F}_x, t)$. Then there exist x times $t_1, t_2, \ldots, t_{x-1}, t_x$ with $t_1 \leq t_2 \leq \cdots \leq t_{x-1} \leq t_x = t$ such that the following properties hold. For each i with $1 \leq i < x$,*

(a) $\Pi(p, t_i) = \Pi(p, t)$,

(b) $\mathbb{P}(\mathcal{F}_i, t_i) \supsetneq \mathbb{P}(\mathcal{F}_{i+1}, t_{i+1})$, *and*

(c) $\left| \mathbb{UF}(\mathcal{F}_i, t_i) \right| \geq \left| \mathbb{P}(\mathcal{F}_{i+1}, t_{i+1}) \right|$.

Proof. The proof is by backward induction on i starting from $x-1$. In order to prove these results, the following auxiliary properties are used, which are part of the induction statement. For each i with $1 \leq i < x$,

(d) $\left| \mathbb{P}(\mathcal{F}_i, t_i) \right| > 0$, and

(e) $p \in \mathbb{P}(\mathcal{F}_i, t_i)$.

The proof of these five results is as follows:

□ Base case (properties (a)-(e) hold for $i = x - 1$)

By definition, $t_x = t$. By assumption, $p \in \mathbb{P}(\mathcal{F}_x, t_x)$. Applying proposition 4.27 obtains that $p \in \mathbb{P}(\mathcal{C}_{x-1}, t_x)$, thereby implying that $\left| \mathbb{P}(\mathcal{C}_{x-1}, t_x) \right| > 0$. Now applying Lemma 4.22 once, to infer that there exists a time, say t_{x-1} with $t_{x-1} < t_x$, such that the following properties hold.

(i) $\Pi(p, t_{x-1}) = \Pi(p, t_x)$, which, in turn, implies that $\Pi(p, t_{x-1}) = \Pi(p, t)$ because $t_x = t$ (property (a)).

(ii) $\mathbb{P}(\mathcal{F}_{x-1}, t_{x-1}) \supsetneq \mathbb{P}(\mathcal{C}_{x-1}, t_x)$, which, in turn, implies that $\mathbb{P}(\mathcal{F}_{x-1}, t_{x-1}) \supsetneq \mathbb{P}(\mathcal{F}_x, t_x)$ because $\mathbb{P}(\mathcal{C}_{x-1}, t_x) = \mathbb{P}(\mathcal{F}_x, t_x)$ (property (b)).

(iii) $\left| \mathbb{UF}(\mathcal{F}_{x-1}, t_{x-1}) \right| \geq \left| \mathbb{P}(\mathcal{C}_{x-1}, t_x) \right|$, which, in turn, implies that $\left| \mathbb{UF}(\mathcal{F}_{x-1}, t_{x-1}) \right| \geq \left| \mathbb{P}(\mathcal{F}_x, t_x) \right|$ (property (c)).

(iv) $\left| \mathbb{P}(\mathcal{F}_{x-1}, t_{x-1}) \right| > 0$ because $\mathbb{P}(\mathcal{F}_{x-1}, t_{x-1}) \supsetneq \mathbb{P}(\mathcal{F}_x, t_x) \supseteq \{p\}$ (property (d)).

(v) $p \in \mathbb{P}(\mathcal{F}_{x-1}, t_{x-1})$ because $\mathbb{P}(\mathcal{F}_{x-1}, t_{x-1}) \supsetneq \mathbb{P}(\mathcal{F}_x, t_x) \supseteq \{p\}$ (property (e)).

□ Induction hypothesis (assume that the properties (a)-(e) hold for some i, $1 < i < x$)

To prove that properties (a)-(e) also hold for $i-1$. Note that, by induction hypothesis, $\mid \mathbb{P}(\mathcal{F}_i, t_i) \mid > 0$. Thus, applying Lemma 4.22 once to infer that there exists time, say t_{i-1} with $t_{i-1} < t_i$, such that the following properties hold.

(i) $\Pi(p, t_{i-1}) = \Pi(p, t_i)$, which, in turn, implies that $\Pi(p, t_{i-1}) = \Pi(p, t)$ (property (a)).

(ii) $\mathbb{P}(\mathcal{F}_{i-1}, t_{i-1}) \supsetneq \mathbb{P}(\mathcal{C}_{i-1}, t_i)$, which, in turn, implies that $\mathbb{P}(\mathcal{F}_{i-1}, t_{i-1}) \supsetneq \mathbb{P}(\mathcal{F}_i, t_i)$ because $\mathbb{P}(\mathcal{C}_{i-1}, t_i) = \mathbb{P}(\mathcal{F}_i, t_i)$ (property (b)).

(iii) $\mid \mathbb{UF}(\mathcal{F}_{i-1}, t_{i-1}) \mid \geq \mid \mathbb{P}(\mathcal{C}_{i-1}, t_i) \mid$, which, in turn, implies that $\mid \mathbb{UF}(\mathcal{F}_{i-1}, t_{i-1}) \mid \geq \mid \mathbb{P}(\mathcal{F}_i, t_i) \mid$ (property (c)).

(iv) $\mid \mathbb{P}(\mathcal{F}_{i-1}, t_{i-1}) \mid > 0$ because $\mathbb{P}(\mathcal{F}_{i-1}, t_{i-1}) \supsetneq \mathbb{P}(\mathcal{F}_i, t_i) \supseteq \{p\}$ (property (d)).

(v) $p \in \mathbb{P}(\mathcal{F}_{i-1}, t_{i-1})$ because $\mathbb{P}(\mathcal{F}_{i-1}, t_{i-1}) \supsetneq \mathbb{P}(\mathcal{F}_i, t_i) \supseteq \{p\}$ (property (e)).

This establishes the lemma. □

The next corollary *quantifies* the number of processes that must be present at *each* of the lower levels for some process to be escalated to a certain level.

Corollary 4.29. *Consider a process p, time t and level x, where $1 \leq x \leq m$, such that process $p \in \mathbb{P}(\mathcal{F}_x, t)$. Let times $t_1, t_2, \ldots, t_{x-1}, t_x$ be as given by Lemma 4.28. Then, for each i with $1 \leq i < x$, $\mid \mathbb{P}(\mathcal{F}_i, t_i) \mid \geq\ x - i + 1$.*

The next corollary *quantifies* the number of unsafe failures that must occur with respect to the filter lock at each of the lower levels for some process to be escalated to a certain level.

Corollary 4.30. *Consider a process p, time t and level x, where $1 \leq x \leq m$, such that process $p \in \mathbb{P}(\mathcal{F}_x, t)$. Let times $t_1, t_2, \ldots, t_{x-1}, t_x$ be as given by Lemma 4.28. Then, for each i with $1 \leq i < x$, $\mid \mathbb{UF}(\mathcal{F}_i, t_i) \mid \geq\ x - i$.*

For the rest of this section, unless otherwise stated, assume that super-passage of a process and consequence interval of a failure are defined *relative to the target lock*.

Theorem 4.31. *Suppose a process p advances to level x at some time t during its super-passage, where $1 \leq x \leq m$. Then, there exist at least $x(x-1)/2$ failures that occurred at or before time t whose consequence interval overlaps with the super-passage of the process p.*

Proof. Let $t_1, t_2, \ldots, t_x$ be the times as given by Lemma 4.28. Since BA-Lock satisfies the locality property, the set of failures that are unsafe with respect to one instance of its filter lock is *disjoint* from the set of failures that are unsafe with respect to another instance of its filter lock. Formally,

$$\forall i, j : 1 \leq i, j \leq x \text{ and } i \neq j : \mathbb{UF}(\mathcal{F}_i, t_i) \cap \mathbb{UF}(\mathcal{F}_j, t_j) = \emptyset \quad \text{(pairwise disjoint property)}$$

Let Π denote the super-passage of p at time t. From the property (a) of Lemma 4.28, $\Pi = \Pi(p, t_1) = \Pi(p, t_2) = \ldots = \Pi(p, t_x)$. In other words, p is executing the same super-passage during the period $[t_1, t_x]$.

Let $\Phi(t)$ denote the set of all failures that occurred at or before time t and whose consequence interval overlaps with the super-passage Π. Note that the consequence interval of any failure with respect to the *target* lock contains the consequence interval of that failure with respect to any instance of its filter lock. This is because all pending requests for that instance of the filter lock are also pending requests for the *target* lock. Thus, $\forall i : 1 \leq i < x : \mathbb{UF}(\mathcal{F}_i, t_i) \subseteq \Phi(t)$. This in turn implies that

$$\bigcup_{i=1}^{x-1} \mathbb{UF}(\mathcal{F}_i, t_i) \subseteq \Phi(t) \qquad \text{(containment property)}$$

Therefore,

$$
\begin{aligned}
\left|\ \Phi(t)\ \right| \geq\ & \left|\ \bigcup_{i=1}^{x-1} \mathbb{UF}(\mathcal{F}_i, t_i)\ \right| && \text{(using containment property)} \\
=\ & \sum_{i=1}^{x-1} \left|\ \mathbb{UF}(\mathcal{F}_i, t_i)\ \right| && \text{(using pairwise disjoint property)} \\
=\ & \sum_{i=1}^{x-1} (x-i) && \text{(using Corollary 4.30)} \\
=\ & (x-1) + \cdots + 2 + 1 && \text{(expanding the sum)} \\
=\ & \frac{x(x-1)}{2} && \text{(algebra)}
\end{aligned}
$$

This establishes the result. □

The following results then follow from the above two theorems.

Theorem 4.32 (BA-Lock is bounded super-adaptive)**.** *The RMR complexity of any passage in a super-passage of* BA-Lock *is given by* $\mathcal{O}(\min\{\sqrt{k+1}, R(n)\})$*, where* k *denotes the failure-density of the super-passage and* $R(n)$ *denotes the RMR complexity of the base* NA-Lock() *for* n *processes.*

Corollary 4.33 (a well-bounded super-adaptive lock)**.** *Assume Jayanti, Jayanti and Joshi's (Jayanti et al., 2019) or Katzan and Morrison's RME algorithm (Katzan and Morrison, 2021) to implement the* NA-Lock*. Then the RMR complexity of any passage in a super-passage of* BA-Lock *is given by* $\mathcal{O}(\min\{\sqrt{k+1}, {}^{\log n}/_{\log\log n}\})$*, where* k *denotes the failure-density of the super-passage.*

The next theorem shows that BA-Lock is *adaptive to contention* as well.

Theorem 4.34. *Suppose a process* p *advances to level* x *at some time* t *during its super-passage, where* $1 \leq x \leq m$*. Then, there exist at least* $x-1$ *super-passages that are simultaneously in-progress along with the super-passage of the process* p *at or before time* t*.*

Proof. Let t_1 denote the time as postulated in the statement of Lemma 4.28. Clearly, $\Pi(p, t_1) = \Pi(p, t)$, $p \in \mathbb{P}(\mathcal{F}_1, t_1)$ and $\left| \mathbb{P}(\mathcal{F}_1, t_1) \setminus \{p\} \right| \geq x - 1$. □

This implies that BA-Lock is adaptive to both failures and contention as stated in the next theorem.

Theorem 4.35 (dual adaptivity)**.** *The RMR complexity of any passage in a super-passage of* BA-Lock *is given by* $\mathcal{O}(\min\{\ddot{c}, \sqrt{k+1}, R(n)\})$*, where* $\ddot{c}$ *denote the point-contention of the super-passage,* k *denotes the failure-density of the super-passage and* $R(n)$ *denotes the RMR complexity of the* NA-Lock() *for* n *processes.*

CHAPTER 5

FAIRNESS

An important desirable property satisfied by many ME algorithms is *fairness*. Intuitively, fairness ensures that no process is able to monopolize shared resources. To define fairness, the `Enter` segment is further partitioned into two segments—*doorway* followed by *waiting room*. A doorway consists of a *bounded* number of steps that a process executes in the beginning of its `Enter` segment, whereas the waiting room is the rest of the `Enter` segment. Further, it is often assumed that the `Recover` segment of a passage executed by a process is bounded, especially if the previous passage of the process, if it exists, was failure-free.

This work establishes a novel definition for fairness referred to as CI-FCFS. To that end, the notion of a exclusive passage is defined first. A failure-free passage is said to be *exclusive* if its associated super-passage is failure-free (and therefore it is the only passage of its super-passage).

CI-FCFS A history H is said to satisfy CI-FCFS if it satisfies the following property. Consider a pair of passages r_i and r_j in H belonging to processes p_i and p_j, respectively, such that (a) both r_i and r_j are exclusive passages, (b) p_i completes its doorway in r_i before p_j begins its doorway in r_j, and (c) r_i does not overlap with the consequence interval of any failure. Then, if p_j has started the CS segment in r_j, p_i must have started the `Exit` segment in r_i.

An RME algorithm is said to satisfy CI-FCFS if every history generated by the algorithm satisfies CI-FCFS. Note that, in the absence of failures, CI-FCFS reduces to traditional FCFS (first come first served) property.

This chapter first argues in Section 5.1 that both the weakly RME lock (Section 4.2), namely WR-Lock, and the strongly RME lock (Sections 4.3 and 4.4, namely SA-Lock, satisfy CI-FCFS. Then a comparison between two different notions of fairness is provided

in Section 5.2; *i.e.*, the CI-FCFS notion of fairness is compared with the notion of k-FCFS used by Golab and Ramaraju in (Golab and Ramaraju, 2019).

Note that BA-Lock is a special case of SA-Lock and thus would also satisfy CI-FCFS. Also, note that it is *not* required for the core lock of the framework (Section 4.3) to be fair.

5.1 Fairness proofs for RME Algorithms

5.1.1 Fairness of weakly RME

The doorway of WR-Lock consists of lines 35 to 48.

Theorem 5.1. WR-Lock *satisfies the CI-FCFS property.*

Proof. Consider a history H, and let r_i and r_j be passages of processes p_i and p_j respectively such that: (a) p_i completes its doorway in r_i before p_j starts its doorway in r_j, (b) both r_i and r_j are exclusive passages, and (c) r_i does not overlap with the consequence interval of any failure.

Assume, on the contrary, that there exists a time instant t in H such that, at time t, p_j is in the CS segment of r_j but p_j has not started the `Exit` segment of r_i. Note that, by time t, both p_i and p_j have completed the doorways for their respective passages. Let t_i and t_j denote the time instants when p_i and p_j, respectively, executed their `FAS` instructions (line 46 of algorithm 4.1). Further, let u_i and u_j denote the admissible nodes owned by p_i and p_j, respectively, at time t. Clearly, $t_i < t_j < t$.

Consider the following claim.

Claim 5.1.1. *As long as u_i is admissible, both u_i and u_j belong to a* single *sub-queue.*

The proof of the claim is by contradiction. Assume the claim doesn't hold. This implies that there exists a process p_k such that p_k (a) executed an `FAS` instruction at time t_k with $t_i < t_k < t_j$, but (b) experienced an unsafe failure f at time t_f with $t_k < t_f < t$. Since

$t_i < t_k$, $t_k < t_f$ and $t_f < t$, it follows that $t_i < t_f < t$. Clearly, r_i contains both t_i and t. This implies that r_j contains t_f and thus overlaps with the consequence interval of f. This is a contradiction. Thus the claim holds.

It follows from the claim that, until u_i stays admissible, u_j cannot become the front node of its sub-queue. In other words, p_j cannot enter the CS segment of r_j until p_i has started executing the `Exit` segment of r_i. □

5.1.2 Fairness of adaptive transformation

The proof that the target lock of the framework in Section 4.3 satisfies the CI-FCFS property assumes that the filter lock satisfies the CI-FCFS property.

Theorem 5.2. SA-Lock *satisfies the CI-FCFS property.*

Proof. Consider a history H, and let r_i and r_j be passages of processes p_i and p_j (with respect to the target lock) such that: (a) p_i completes its doorway in r_i before p_j starts its doorway in r_j, (b) both r_i and r_j are exclusive passages, and (c) r_i does not overlap with the consequence interval of any failure. Assume that p_j in the CS segment of r_j.

Let $r_i(\mathcal{F}) \subseteq r_i$ and $r_j(\mathcal{F}) \subseteq r_j$ denote the passages of p_i and p_j, respectively, with respect to the filter lock. Due to the arrangements of the locks, the following can be inferred. First, if the three conditions of the CI-FCFS property hold for r_i and r_j, then they also hold for $r_i(\mathcal{F})$ and $r_i(\mathcal{F})$. Second, if p_j is the CS segment of r_j, then it is also in the CS segment of $r_j(\mathcal{F})$. Third, if p_i has started the `Exit` segment of $r_i(\mathcal{F})$, then it has also started the `Exit()` segment of r_i.

Since WR-Lock satisfies CI-FCFS, p_i must have started the `Exit` segmentin $r_i(\mathcal{F})$ and thus must have started the `Exit` segmentin r_i. □

Since BA-Lock is a special case of SA-Lock, the following corollary follows.

Corollary 5.3. BA-Lock *satisfies the CI-FCFS property.*

5.2 Comparison with k-FCFS

Golab and Ramaraju define the concept of a k-failure-concurrent passage, where $k \geq 0$, to capture how a failure may impact the RMR complexity of a passage. The concept is defined recursively. Given a history, a passage of a process is said to be 0-failure-concurrent if it either ends with or begins after a crash of the process. It is said to be k-failure-concurrent if it is either $(k-1)$-failure-concurrent or its super passage overlaps with another super passage containing a $(k-1)$-failure-concurrent passage. Intuitively, the parameter k measures the "distance" of a passage from a 0-failure-concurrent passage in a given history. [1]

On the other hand, this work uses the concept of consequence interval of a failure. Given a history H, a passage is said to be *CI-concurrent* in H if its super-passage overlaps with the consequence interval of one or more failures in H.

A natural question to ask is how the concepts of k-failure-concurrent and CI-concurrent are related. This question is answered by showing that CI-concurrent is "stronger"[2] than 2-failure-concurrent and "incomparable" with 1-failure-concurrent[3]

Theorem 5.4. *Given a passage r in a finite history H, if r is CI-concurrent, then it is also 2-failure-concurrent.*

Proof. Given a passage r, let $\Pi(r)$ denote the super-passage associated with r.

Consider a passage r_i of process p_i that is CI-concurrent. By definition, $\Pi(r_i)$ overlaps with the consequence interval of a failure, say f. Let f involve the failure of process p_j while executing the passage r_j. Let $\mathbb{P}(f)$ denote the set of processes that have a super-passage in progress at the time when f occurred. Note that $\mathbb{P}(f) \neq \emptyset$ because $p_j \in \mathbb{P}(f)$. Let $p_k \in \mathbb{P}(f)$

[1] Note that a passage is exclusive if and only if it is not 0-failure-concurrent.

[2] Here stronger refers to the maximum duration for the impact of a failure

[3] 1-FC is stronger than 2-FC.

denote the process whose super-passage extends for the *longest* time in H. Finally, let r_k denote the *most recent* passage of p_k that started before f occurred. By definition, r_k must be in-progress when f occurred.

Thus, $\Pi(r_k)$ overlaps with $\Pi(r_j)$. This in turn implies that r_k is 1-failure-concurrent because r_j is 0-failure-concurrent. Also, note that $\Pi(r_k)$ strictly contains the consequence interval of f and hence overlaps with $\Pi(r_i)$. This implies that r_i is 2-failure-concurrent. □

To show that 1-failure-concurrent and CI-concurrent are incomparable properties, *i.e.*, neither implies the other, the following examples are provided.

Theorem 5.5. *There exists a history H and a passage r in H such that r is 1-failure-concurrent but not CI-concurrent.*

Proof. Proof by example.

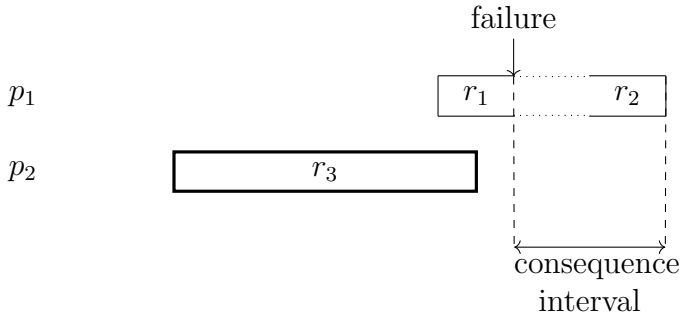

Figure 5.1. A passage (r_3) that is 1-failure-concurrent but not CI-concurrent

Consider the history shown in Figure 5.1. Passages r_1 and r_2 are the only 0-failure-concurrent passages in the history. Passage r_3 overlaps with r_1 and thus is 1-failure-concurrent, but is not CI-concurrent. □

Theorem 5.6. *There exists a history H and a passage r in H such that r is CI-concurrent but not 1-failure-concurrent.*

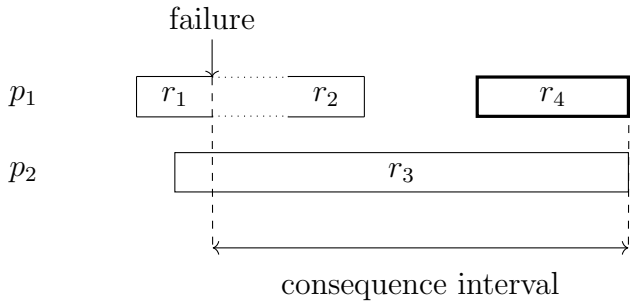

Figure 5.2. A passage (r_4) that is CI-concurrent but not 1-failure-concurrent

Proof. Proof by example.

Consider the history shown in Figure 5.2. Passages r_1 and r_2 are the only 0-failure-concurrent passages in the history. Passage r_4 overlaps with the consequence interval of the failure and is 2-failure-concurrent, but not 1-failure-concurrent. □

Finally, the next example shows that CI-concurrent is a strictly stronger property than 2-failure-concurrent, *i.e.*, the latter does not imply the former.

Theorem 5.7. *There exists a history H and a passage r in H such that r is 2-failure-concurrent but not CI-concurrent.*

Proof. Proof by example.

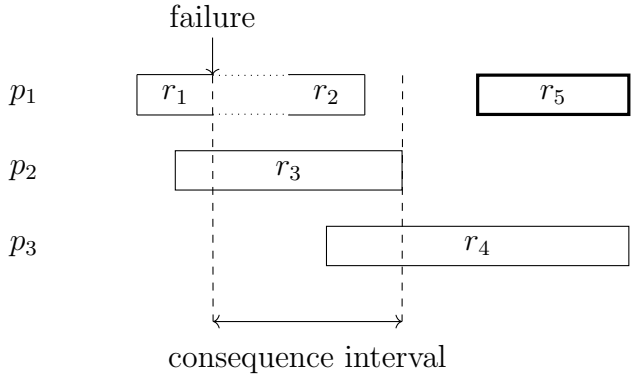

Figure 5.3. A passage (r_5) that is 2-failure-concurrent but not CI-concurrent

Consider the history shown in Figure 5.3. Passage r_2 begins after a failure which makes it 0-failure-concurrent. Passage r_4 overlaps with r_2, which makes it 1-failure-concurrent. Passage r_5 overlaps with r_4, which makes it 2-failure-concurrent. However, r_5 does not overlap with the consequence interval of any failure. □

Note that Golab and Ramaraju use the notion of k-failure concurrency to define a notion of fairness especially suited to RME algorithms, referred to as k-FCFS. Intuitively, k-FCFS guarantees FCFS among two passages provided none of them is k-failure-concurrent. The results above can be extended to establish the following relationship among different fairness properties. (a) CI-FCFS implies 2-FCFS, and (b) CI-FCFS and 1-FCFS are incomparable..

Incidentally, the framework in Section 4.3 yields an RME algorithm that not only satisfies the CI-FCFS property but also satisfies the 1-FCFS property.

Theorem 5.8. WR-Lock *satisfies the 1-FCFS property.*

Proof. The proof is analogous to the proof of Theorem 5.1 with the following modification. Let r_k denote the passage of p_k that ends with the failure f. Clearly, r_k is 0-failure-concurrent passage and overlaps with r_i. This implies that r_i is a 1-failure-concurrent passage—a contradiction. Thus the claim holds in this case as well. □

Theorem 5.9. SA-Lock *satisfies the 1-FCFS property.*

Proof. The proof is identical to that of Theorem 5.2. □

Thus it follows that:

Corollary 5.10. BA-Lock *satisfies the 1-FCFS property.*

CHAPTER 6

THE RECOVERABLE BROADCAST OBJECT

The recoverable broadcast object, hereafter denoted by BROADCAST, allows a designated process to notify and "wake up" one or more processes waiting for it to reach a certain point during its execution. It is a generalization of the SIGNAL object used by Jayanti, Jayanti and Joshi in (Jayanti et al., 2019) to design an RME algorithm with sub-logarithmic RMR complexity. It is also a generalization of the BARRIER object used by Golab and Hendler in (Golab and Hendler, 2018) to design an RME algorithm for system-wide failures with $\mathcal{O}(1)$ RMR complexity.

The SIGNAL object in (Jayanti et al., 2019) allows one process to notify and "wake up" another process waiting for it to reach a certain point during its execution. However, it can only be used once and different processes must wait on the object sequentially. On the other hand, a BROADCAST object can be used *repeatedly* and multiple processes can wait on the object *concurrently*.

The BARRIER object in (Golab and Hendler, 2018) is used to synchronize processes to wait at a certain point during their respective executions. The leader process releases the "BARRIER" and notifies other processes to "wake up" and resume execution. This object assumes that all processes fail together and may lead to a deadlock if processes fail independently. In that sense, the definition of BROADCAST object and its implementation can be viewed as a generalization of BARRIER object for the independent failure model.

6.1 Definition

The broadcast object, in essence, is a recoverable MRSW (Multi-Reader Single-Writer) *counter* object that stores a non-negative value and supports four operations, namely `SET`, `WAIT`, `WITHDRAW` and `READ`, with the following behavior.

1. SET() takes a positive number x as an argument and sets the counter value to x if its current value is smaller than x.
2. WAIT() also takes a positive number x as an argument and blocks until the counter value has advanced to at least x.
3. WITHDRAW() also takes a positive number x as an argument and revokes previous invocations of WAIT(x) if any.
4. READ() returns the current counter value.

A BROADCAST object is *owned* by a process and only the owner can change the value of the object. Thus, the SET operation can only be invoked by the owner process and the WAIT operation can only be invoked by a non-owner process.

An operation of a BROADCAST object is invoked by a process from *within* another (possibly RME) algorithm. The memory reclamation algorithm (Algorithms 7.1 and 7.2) from Chapter 7 uses the SET, WAIT and READ operations while the algorithms for BCSR transformation (Algorithms 8.4 and 8.5) and FRF transformation (Algorithms 8.6 and 8.7) from Chapter 8 utilize all of the SET, WAIT, READ and WITHDRAW operations of the BROADCAST object.

Given a history, each invocation of an operation of the BROADCAST object is referred to as a separate *instance* of the operation. An instance of an operation *terminates* when the calling process either crashes (while executing that instance) or returns (from the invocation). As such, if a process crashes while executing an operation, then, upon recovery it *does not* resume execution from where it failed. Rather, like in an RME algorithm, the process restarts from the beginning of the underlying algorithm. In the context of a BROADCAST object, this translates to the process possibly requiring another invocation of the same operation again at some point in the future. This behavior is formalized by requiring a history of

the underlying algorithm, *when limited to the steps relevant to the* BROADCAST *object*[1], to satisfy certain properties.

To ensure expected behavior from an implementation of the BROADCAST object, the history of the underlying algorithm should be well-formed[2] and legal (with respect to each BROADCAST object). Intuitively, a well-formed history guarantees that the BROADCAST operations are safe, while requirements on the history to be legal guarantee that the BROADCAST operations are live.

A history H is said to be *well-formed* with respect to a BROADCAST object if it satisfies the following conditions:

(W1) The owner process invokes the SET operation in an *incremental* manner. Specifically, let SET (x) and SET (y) be two consecutive instances of the SET operation in H. Then, $x \leq y \leq x + 1$.

(W2) The argument of every WAIT operation is "consistent" with the argument of its immediately preceding SET operation. Specifically, let x denote the argument of a WAIT operation in H, and let y denote the argument of its immediately preceding invocation of SET operation in H. Then, $x \leq y + 1$. In case no such preceding SET operation exists, the value of y is assumed to be the initial value of the counter.

An instance of an operation (SET, WAIT, WITHDRAW or READ) is referred to as *failure-free* if the process does not crash while executing the instance. An instance of WAIT, WITHDRAW or READ operation is said to have *completed-successfully* if it has terminated without failing. On the other hand, an instance of SET (x) is said to have *completed-successfully* if it has terminated (possibly with a failure) and an invocation of the READ operation by the owner immediately afterwards would return the value at least x. Note that an instance of SET

[1]This chapter considers histories to only consist of steps taken by processes *while* executing the operations of a BROADCAST object.

[2]The term well-formed history is overloaded here. It has a different meaning for the BROADCAST object as compared to well-formed histories for the RME problem

operation may have completed-successfully even though it was not failure-free if the owner process was able to execute "enough" steps before crashing.

The notion of a legal history is based on the notion of a run. Given a history H and a process p that is live in H, let $\mathcal{W}(H, p)$ denote the sequence of *all* instances of WAIT and WITHDRAW operations invoked by p in H. A non-empty sub-sequence λ of the sequence $\mathcal{W}(H, p)$ forms a *run* if it satisfies the following conditions:

(a) All instances in λ have the same argument.

(b) Two instances are consecutive in λ only if they are consecutive in $\mathcal{W}(H, p)$.

(c) Every non-last instance in λ terminates with a failure (*i.e.*, does not complete-successfully).

Note that the last instance in λ, in case λ is finite, may or may not terminate, and, if it does terminate, it may terminate with a failure.

A well-formed history H is said to be *legal* if it satisfies the following conditions:

(L1) If an instance (of an operation) invoked by a process p in H has not terminated, then eventually p either crashes or takes another step in H.

(L2) Every run in H contains a finite number of instances.

(L3) The last instance of every maximal run in H is failure-free.

The first condition (L1) implies that a history does not abruptly stop in the "middle" of an operation. The last two conditions (L2) and (L3) together imply that a process must repeatedly invoke an instance of WAIT or WITHDRAW operation with the "same" argument until it is able to execute a failure-free instance.

The algorithm for the WAIT operation under the DSM model uses helping as it uses a wakeup-chain to bound the worst-case RMR complexity by $\mathcal{O}(1)$. As such, it is guaranteed to complete-successfully only if the history is legal.

6.2 Correctness

Designing an implementation of the BROADCAST object involves designing the operations SET, WAIT, WITHDRAW and READ such that the following correctness properties are satisfied:

(P1) **Safety**: For any well-formed history H, an invocation of WAIT(x) by a process p_r ($r \neq w$) completes-successfully at some time t in H, only if process p_w has invoked SET(x) at least once before time t in H.

(P2) **Liveness**: Let H be a legal history in which at least one instance of SET(x) completes-successfully. Then every instance of WAIT(x) in H eventually terminates.

(P3) **Wait-Freedom**: Every instance of SET, WITHDRAW and READ operation in a legal history eventually terminates within a bounded number of the invoking process' steps (possibly with a failure).

The safety property (P1) ensures that the WAIT operation does not return spuriously while the liveness (P2) property guarantees that the WAIT operation does eventually return when appropriate.

Note that even though an invocation of SET(x) does not complete-successfully, an invocation of WAIT(x) may still complete-successfully without violating the safety property. Consider the following scenario. An instance of WAIT(m) terminates and a subsequent instance of READ returns $m-1$. This situation is permissible only if there exists an invocation of SET(m) before the WAIT(m) returns.

Additionally, note that that a legal history does not automatically guarantee the liveness property. Even though the last instance of a legal history is guaranteed to be failure-free and while the number of instances are guaranteed to be finite, it is not guaranteed that the last instance itself would be finite (or completes-successfully).

The wait-freedom (P3) property is also a liveness property that guarantees that the steps of the operations other than the WAIT operation are bounded in order to avoid any potential deadlock situations.

In addition to the correctness properties, it is also desirable for an implementation to satisfy the following performance property.

(P4) **RMR-Efficiency**: Each instance of READ, WITHDRAW, SET and WAIT operation has worst-case RMR complexity of $\mathcal{O}(1)$.

6.3 Implementation

The implementations of the BROADCAST object are described separately for CC and DSM models since the algorithms to achieve $\mathcal{O}(1)$ RMR complexity for all four operations are quite different for each model. The algorithm for the CC model is referred to as BROADCAST-CC, and the algorithm for the DSM model is referred to as BROADCAST-DSM.

6.3.1 CC model

It is relatively trivial to implement the BROADCAST object for the CC model in which all four operations have $\mathcal{O}(1)$ RMR complexity in the worst case as shown in Algorithm 6.1.

The algorithm uses a single shared MRSW **integer** variable *count*. In the SET(x) operation, the owner process writes x to the variable *count* if its current value is smaller than x (lines 139 and 140). In the WAIT(x) operation, a non-owner process spins until the variable *count* has a value of at least x (line 144). In the WITHDRAW(x) operation, a non-owner process does not need to perform any steps to revoke a call to WAIT(x). This is due to the fact that a non-owner process does not register its WAIT(x) request prior to spinning. In the READ() operation, the process simply returns the current value of *count* (line 135). Note that only the WAIT operation contains a loop. A process that invokes WAIT(x) incurs only $\mathcal{O}(1)$ RMRs while busy-waiting in the loop. This is guaranteed only for well-formed histories with the help of (W1) and (W2) conditions.

Algorithm 6.1: Pseudocode of process p_i for implementing BROADCAST object under the CC model

```
130 shared variables
        /* to model the counter */
131     count: integer variable, initially 1
132 end

    /* returns current counter value */
133 Function READ(): returns integer
134 begin
135     return count
136 end

    /* set the counter to a desired
       value; can only be invoked if p_i
       is the owner process */
137 Function SET( x: integer variable)
138 begin
139     if (count ≥ x) then return
140     count ← x
141 end

    /* wait until the counter reaches a
       desired value; can only be invoked
       if p_i is a non-owner process */
142 Function WAIT( x: integer variable)
143 begin
144     await count[i] ≥ x
145 end

    /* revoke previous invocations of
       WAIT(x); can only be invoked if p_i
       is a non-owner process */
146 Function WITHDRAW( x: integer variable)
147 begin
        /* CC model requires no extra
           work to withdraw the WAIT
           request */
148     return;
149 end
```

6.3.2 DSM model

The solution for the BROADCAST object under the CC model yields an unbounded RMR complexity for the WAIT operation under the DSM model. This is because, when $n \geq 3$, at least one non-owner process has to spin on a remote memory location. Another approach is for a non-owner process to spin on a local memory location, while the owner process is responsible for waking up all the waiting processes by updating the memory location of each spinning process. This approach yields $\mathcal{O}(1)$ RMR complexity for WAIT operation. However, the SET operation would yield $\mathcal{O}(n)$ RMR complexity.

This section describes an efficient algorithm to implement the BROADCAST object in the DSM model that incurs only $\mathcal{O}(1)$ RMRs for all four operations and uses $\mathcal{O}(n)$ space per BROADCAST object. The main idea is that, instead of notifying the spinning processes by itself, the owner process creates a *wake-up chain* in its local memory such that each process

in the chain is responsible for waking up *at most one* other process. The owner process then only needs to wake up the *first* process in the chain.

This technique is similar to the one used by Golab, *et al.* in (Golab et al., 2006) to derive a leader election algorithm with $\mathcal{O}(1)$ RMR complexity under the DSM model. However, their algorithm was designed to operate in a failure-free environment, whereas this algorithm is designed to be recoverable. Further, processes can use this BROADCAST object for synchronization multiple times, repeatedly, with no additional space overhead.

The pseudocode for the BROADCAST object under the DSM model is given in Algorithms 6.2 and 6.3. This algorithm uses two shared **integer** variables, namely A and B, and three shared **integer array** variables, namely *announce*, *target* and *wakeup*. Each array variable is of size n with one entry for each process. The variables A and B both store the counter value; they are updated at beginning and end, respectively, of a `SET()` operation. The array variables *announce* and *wakeup* are local to the owner process. However, the array variable *target* is distributed among all processes with the i-th entry local to process p_i. A process p_i uses $announce[i]$ to inform the owner process of its intention to wait until the counter value has advanced to a desired value and $target[i]$ to busy-wait until it is released (from spinning) by another process (note that $target[i]$ is local to p_i). Finally, the owner process uses the *wakeup* array to set up a wake-up chain, where p_i, upon waking up, is responsible for notifying the process whose identifier is stored in $wakeup[i]$, if any.

`WAIT` operation: Let x denote the input argument of the operation. The invoking process, say p_i, writes x to $target[i]$ and $announce[i]$ in order (lines 162 and 163), thereby informing the owner process of its intention to wait for the counter value to reach x. It then reads the value of the variable A (line 164) to check if the owner process has already invoked the `SET` operation with argument x. If so, it resets $target[i]$ to zero (line 165) so that it does not busy-wait in the next step. It then spins until $target[i]$ is reset to zero (line 167). Upon quitting the busy-waiting loop, it clears $announce[i]$ (line 168) and reads $wakeup[i]$ (line 169)

Algorithm 6.2: Pseudocode of process p_i for implementing BROADCAST object under the DSM model (Part 1 of 2)

```
150 shared variables
        /* two integer variables to model
           the counter */
151     A, B: integer variable, initially 1
        /* array for a non-owner process
           to inform the owner process of
           its intention to wait until
           the counter has advanced to a
           certain value; all entries are
           local to the owner process */
152     announce: array [1...n] of integer
          variable, all elements initially 0
        /* memory location for each
           non-owner process to spin on;
           the i-th entry is local to
           process p_i */
153     target: array [1...n] of integer
          variable, all elements initially 0
        /* array for the owner process to
           create a wake-up chain; all
           entries are local to the owner
           process */
154     wakeup: array [1...n] of integer
          variable, all elements initially 0
155 end
    /* returns current counter value */
156 Function READ(): returns integer
157 begin
158     return B
159 end
    /* wait until the counter reaches a
       desired value; can only be invoked
       if p_i is a non-owner process */
160 Function WAIT( x: integer variable)
161 begin
        /* initialize the location to
           spin on */
162     target[i] ← x
        /* announce the intent to wait */
163     announce[i] ← x
        /* no need to wait if the owner
           process has previously invoked
           SET() operation with argument
           value x */
164     if A ≥ x then
            /* reset the target location */
165         target[i] ← 0
166     end if
        /* spin until some process resets
           the target location */
167     await target[i] = 0
        /* revoke the announcement */
168     announce[i] ← 0
        /* check for next process to wake
           up */
169     k ← wakeup[i]
170     if k > 0 then
            /* wake up the next process in
               the chain */
171         CAS(target[k], x, 0)
172     end if
173 end
```

to determine the identifier of the next process to be notified, if any. Let $wakeup[i] = k$. If $k > 0$, and $target[k] = x$; then process p_i notifies process p_k by resetting $target[k]$ to zero using a CAS instruction (line 171). Note that the *wakeup* array may contain multiple chains;

Algorithm 6.3: Pseudocode of process p_i for implementing BROADCAST object under the DSM model (Part 2 of 2)

```
   /* set the counter to a desired
      value; can only be invoked if p_i
      is the owner process */
174 Function SET( x: integer variable)
175 begin
176     if B ≥ x then return
        /* update the first counter */
177     A ← x
        /* create the wake-up chain */
178     last ← 0
        /* scan the announce array */
179     for j ← 1 to n do
180         if announce[j] = x then
                /* assign p_j to wake-up the
                   last waiting process */
181             wakeup[j] ← last
182             if announce[j] = x then
                    /* affirm that p_j is
                       still waiting */
183                 last ← j
184             end if
185         end if
186     end for
187     if last > 0 then
            /* release the last process;
               other waiting processes get
               released one-by-one */
188         CAS(target[last], x, 0)
189     end if
        /* update the second counter */
190     B ← x
191 end
```

```
   /* revoke previous invocations of
      WAIT(x); can only be invoked if p_i
      is a non-owner process */
192 Function WITHDRAW( x: integer variable)
193 begin
        /* revoke the announcement */
194     announce[i] ← 0
195     if A < x then
            /* no need to wake any process
               since owner process has not
               yet created a wake-up chain
               with argument value x */
196         return
197     end if
        /* check for next process to wake
           up */
198     k ← wakeup[i]
199     if k > 0 then
            /* wake up the next process in
               the chain */
200         CAS(target[k], x, 0)
201     end if
202 end
```

this algorithm ensures that all processes in the *same chain* had invoked WAIT for the counter to reach the *same value*.

SET operation: Let x denote the input argument of the operation. The invoking process first registers its SET operation by writing x to the variable A (line 177). This ensures that any process that invokes WAIT(x) hereafter does not need to block. It then scans the *announce*

array looking for processes that may be waiting for the counter value to advance to x and chains all of them together (in a descending order of their identifiers) using the *wakeup* array (lines 178 to 186). Consider two processes p_ℓ and p_j with $\ell < j$ such that $announce[\ell] = x$, $announce[j] = x$ and, for each k with $\ell < k < j$, $announce[k] \neq x$. The owner process then sets $wakeup[j] = \ell$ indicating that process p_j, upon being released from spinning, is responsible for notifying process p_ℓ. At this point, if p_j has revoked its announcement (either via the WAIT operation at line 168 or the WITHDRAW operation at line 194, it may have missed reading the $wakeup[j]$ field, and thus create a situation in which p_ℓ spins indefinitely with no process responsible for notifying it. However, after writing ℓ to $wakeup[j]$, the owner process reads $announce[j]$ again (line 182) to ascertain that p_j has not already revoked its announcement. After building this chain, the owner process notifies the first process in the chain by clearing its entry in the *target* array (line 188), which then leads to a sequence of *cascading* notifications. Finally, it writes x to the variable B (line 190) to establish its SET operation completed-successfully.

WITHDRAW operation: Let x denote the input argument of the operation. The invoking process, say p_i, first resets $announce[i]$ to zero (line 194). It then reads the value of the variable A (line 195) to check if the owner process has already registered an invocation of the SET operation with argument x. If at this point $A < x$, it is guaranteed that p_i would not get added to the wakeup chain for x (unless p_i invokes another instance of WAIT(x)) . Otherwise, if $A \geq x$, then the owner process has registered its SET(x) operation and p_i may be part of a wakeup chain. Note that at this point, it is safe to wake up the next process in the wakeup chain. It then reads $wakeup[i]$ (line 198) to determine the identifier of the next process to be notified, if any. Let $wakeup[i] = k$. If $k > 0$, and $target[k] = x$; then process p_i notifies process p_k by resetting $target[k]$ to zero using a CAS instruction (line 200).

READ operation: It returns the value of the variable B (line 158). The main use of this operation is to determine the last SET() operation that completed-successfully, prior to invoking a new SET() operation.

All four operations are designed to be idempotent in the sense that executing an operation multiple times with the same argument, possibly partially in some cases, does not lead to any erroneous behavior.

6.4 Analysis and Proofs of Correctness

This section only focuses on proving the correctness of BROADCAST-DSM because the correctness proof for BROADCAST-CC is quite straightforward.

In the rest of this section, p_w to refers to the owner process of the BROADCAST object. The following proposition follows from the facts that only p_w can write to the variable A and p_w invokes the `SET` operation in an incremental manner (W1).

Proposition 6.1. *Given a well-formed history H, if the variable A has value at least x at time t, then there exists an invocation of* `SET(`x`)` *by p_w in H at or before time t.*

The next proposition follows from code inspection.

Proposition 6.2. *Given a well-formed history H, if process p_w executes the* `CAS` *instruction on* line 188 *with an expected value of x at some time t, then the variable A has a value that is at least x at time t in H.*

The same property also applies for a non-owner process as proven in Lemmas 6.3 and 6.4.

Lemma 6.3. *Given a well-formed history H, if a process p_r, where $r \neq w$, executes the* `CAS` *instruction on* line 200 *with an expected value of x at time t, then the variable A has a value that is at least x at time t in H.*

Proof. As the code inspection shows, p_r executes the `CAS` instruction on line 200 only if it does not return from the `WITHDRAW` operation at line 196 which can happen only if the condition $A < x$ on line 195 is false. Thus, the variable A must have a value that is at least x. □

Lemma 6.4. *Given a well-formed history H, if a process p_r, where $r \neq w$, executes the* `CAS` *instruction on* line 171 *with an expected value of x at time t, then the variable A has a value that is at least x at time t in H.*

Proof. It is sufficient to prove that the statement holds if p_r is the *first* non-owner process in H to invoke the `CAS` instruction on line 171 with the expected value of x. As the code inspection shows, p_r executes the `CAS` instruction only after quitting its busy-waiting loop, which can happen only *after* the entry $target[r]$ has been reset (to zero). The entry can be reset in only four ways. First, p_r resets $target[r]$ itself at line 165 because it read the value of A to be at least x earlier at line 164. Second, p_w resets $target[r]$ using the `CAS` instruction on line 188. In this case, it follows from proposition 6.2 that A has value at least x at the time p_w performed the `CAS` instruction. Third, another non-owner process, say p_s with $s \notin \{w, r\}$, resets $target[r]$ using the `CAS` instruction on line 171. By the assumption that p_r is the first non-owner process to invoke the `CAS` instruction line 171 with an expected value of x, the third case cannot occur. Fourth, another non-owner process, say p_s with $s \notin \{w, r\}$, resets $target[r]$ using the `CAS` instruction on line 200. As shown in Lemma 6.3, this can only happen if A has a value of at least x. □

Theorem 6.5. BROADCAST-DSM *satisfies the* safety (P1) *property.*

Proof. Let H be a well-formed history, in which an invocation of `WAIT(`x`)` by a process p_r, where $r \neq w$, completes-successfully at some time t in H.

As the code inspection shows, p_r quits the busy-waiting loop in the `WAIT()` operation after $target[r]$ has been reset to 0. This entry can only be reset in three ways. As shown below, each of these three cases implies that the value of A is at least x at the time the entry was reset. The statement then follows from proposition 6.1.

First, p_r resets $target[r]$ itself at line 165 because it read the value of A to be at least x earlier at line 164.

Second, p_w resets $target[r]$ using a CAS instruction on line 188. In this case, it follows from proposition 6.2 that the value of A is at least x at the time the entry was set.

Third, another non-owner process, say p_s, where $s \notin \{w, r\}$, resets $target[r]$ using a CAS instruction from the WAIT operation (on line 171) or from the WITHDRAW operation (on line 200). In this case, it follows from Lemma 6.4 or Lemma 6.3 respectively that the value of A is at least x at the time the entry was reset.

Then the result that process p_w has invoked SET(x) before time t in H follows from proposition 6.1. □

For the remainder of this section, the following notation needs to be defined. Suppose a process p is executing an instance I of an operation op, where $op \in \{$SET, WAIT, READ, WITHDRAW $\}$. Let m denote the line number belonging to op in the pseudocode shown in Algorithms 6.2 and 6.3. If the line is not part of the for-loop in SET(), then $t(I, m)$ is used to denote the time when p finished executing the line numbered m during I. Otherwise, $t(I, m, k)$ is used to denote the time when p finished executing the line for the iteration (of the for-loop) with j set to k during I. Finally, given a variable var, var^t is used to denote the value of var at time t.

Note that, in BROADCAST-DSM, the SET operation completes-successfully once it has successfully executed line 190.

Theorem 6.6. BROADCAST-DSM *satisfies the* liveness (P2) *property.*

Proof. Assume on the contrary, that BROADCAST-DSM does not satisfy the liveness (P2) property. Thus, there exists a legal history H in which at least one instance of SET(x) completes-successfully and at least one process in H invokes WAIT(x) and does not crash but the invocation never completes-successfully.

Let p_ℓ be the process with the largest identifier among such processes and let I_ℓ denote the corresponding instance of WAIT(x) in H invoked by p_ℓ that never terminates. This

also implies that p_ℓ eventually advances to the busy-waiting loop (line 167) during I_ℓ (from requirement (L1)) and gets stuck in the loop (since it is the only blocking step). Let I_w denote the *first* instance of `SET(`x`)` in H that p_w is able to complete-successfully. It follows from code inspection that p_w is able to write x to variable B (line 190) during I_w. Further, any subsequent invocation of `SET(`x`)` by p_w in H simply returns after ascertaining that variable B has value at least x, without creating any "new" wake-up chain.

For ease of exposition, unless otherwise stated, steps of processes p_w and p_ℓ are assumed to belong to instances I_w and I_ℓ, respectively.

Since p_ℓ never advances beyond the busy-waiting loop, $target[\ell]$ and $announce[\ell]$ are never reset hereafter. Formally,

$$\forall t : t \geq t(I_\ell, 164) : (announce^t[\ell] = x) \wedge (target^t[\ell] = x) \tag{6.1}$$

Clearly, p_ℓ reads the value of A *before* p_w writes value x to A. Formally,

$$t(I_\ell, 164) \ < \ t(I_w, 177) \tag{6.2}$$

Let $\mathcal{C} = \{c_1, c_2, \ldots, c_h\}$ denote the set of process indices in the wake-up chain created by p_w (in the reverse order of their identifiers). The next part proves that p_ℓ must be part of this wake-up chain, *i.e.*, $\ell \in \mathcal{C}$. Using (6.1) and (6.2), it can be inferred that

$$\forall t : t \geq t(I_w, 177) : (announce^t[\ell] = x) \wedge (target^t[\ell] = x) \tag{6.3}$$

This implies that, when p_w reads the value of $announce[\ell]$ twice at lines 180 and 182, it finds it to be x both times. Thus $\ell \in \mathcal{C}$.

It is now sufficient to prove that some process executes `CAS(`$target[\ell]$`, `x`, 0)` after time $t(I_w, 177)$, thereby resetting $target[\ell]$ to 0 and enabling p_ℓ to quit its busy-waiting loop. The remainder of the code after the busy-waiting loop is wait-free, thereby implying that I_ℓ eventually completes-successfully. This will contradict the original assumption that I_ℓ never completes-successfully. There are two cases to consider.

Case 1 ($\ell = c_1$) : It follows from code inspection that p_w proceeds to reset $target[c_1]$ to 0 using a `CAS` instruction, which is guaranteed to succeed due to (6.3).

Case 2 ($\ell = c_j$, where $j > 1$) : Since $p_{c_{j-1}}$ is part of the wake-up chain created by p_w, when p_w reads $announce[c_{j-1}]$ for the second time, p_w finds that $announce[c_{j-1}]$ has value x. This implies that I_w "overlaps" with one or more runs[3] invoked by $p_{c_{j-1}}$. Let $I_{c_{j-1}}$ denote the *last* instance of the *latest* run invoked by $p_{c_{j-1}}$ that "overlaps" with I_w. Without loss of generality assume that $I_{c_{j-1}}$ is maximal. Note that $I_{c_{j-1}}$ is guaranteed to exist because: (i) I_w is finite (by choice of I_w) and hence there is a last run, and (ii) every run in H has a finite number of instances (L2) and hence the last run has a last instance. Furthermore, $I_{c_{j-1}}$ is failure-free since the last instance of every maximal run in H is failure-free (L3). Note that $I_{c_{j-1}}$ may be instance of `WAIT`(x) or `WITHDRAW`(x) operation. The rest of the proof assumes the former case, but the proof can be easily adapted for the latter case.

Note that $c_{j-1} > \ell$. By the choice of p_ℓ, $I_{c_{j-1}}$ is finite and completes-successfully. Again, for ease of exposition, unless otherwise stated, steps of process $p_{c_{j-1}}$ are assumed to belong to instance $I_{c_{j-1}}$.

By the choice of $I_{c_{j-1}}$, once $p_{c_{j-1}}$ resets $announce[c_{j-1}]$ to 0, it is not set to x again at least until I_w terminates. Formally,

$$\forall t : t(I_{c_{j-1}}, 168) \leq t \leq t(I_w, 190) : announce^t[c_{j-1}] \neq x \tag{6.4}$$

It follows from code inspection that p_w writes c_j to $wakeup[c_{j-1}]$ before it reads $announce[c_{j-1}]$ for the second time. Formally,

$$t(I_w, 181, c_{j-1}) \ < \ t(I_w, 182, c_{j-1}) \tag{6.5}$$

[3]Recall that a run consists of `WAIT`(x) and `WITHDRAW`(x) operations.

Since $p_{c_{j-1}}$ is part of the wake-up chain created by p_w, when p_w reads $announce[c_{j-1}]$ for the second time, p_w finds that $announce[c_{j-1}]$ has value x. Thus, using (6.4), it can be inferred that

$$t(I_w, 182, c_{j-1}) \ < \ t(I_{c_{j-1}}, 168) \tag{6.6}$$

It also follows from code inspection that $p_{c_{j-1}}$ resets $announce[c_{j-1}]$ to 0 before it reads the value of $wakeup[c_{j-1}]$. Formally,

$$t(I_{c_{j-1}}, 168) \ < \ t(I_{c_{j-1}}, 169) \tag{6.7}$$

Combining (6.5), (6.6) and (6.7), derives that p_w writes c_j to $wakeup[c_{j-1}]$ before $p_{c_{j-1}}$ reads from $wakeup[c_{j-1}]$. Formally,

$$t(I_w, 181, c_{j-1}) \ < \ t(I_{c_{j-1}}, 169) \tag{6.8}$$

When p_w reads $announce[c_{j-1}]$ for the second time, it finds it to be x. Thus, using (W1), it can be inferred that

$$\forall t : t(I_w, 182, c_{j-1}) \leq t \leq t(I_{c_{j-1}}, 169) : announce^t[c_{j-1}] \in \{x, 0\} \tag{6.9}$$

Since any subsequent invocation of `SET(`x`)` by p_w (after I_w) does not create any new wake-up chain, it implies that p_w does not write to $wakeup[c_{j-1}]$ between $t(I_w, 181, c_{j-1})$ and $t(I_{c_{j-1}}, 169)$. As a result, when $p_{c_{j-1}}$ reads the value of $wakeup[c_{j-1}]$, it finds that it is set to c_j $(= \ell)$. It follows from code inspection that $p_{c_{j-1}}$ proceeds to execute `CAS(`$target[\ell]$`,` x`,` 0`)`, which is guaranteed to succeed due to (6.1).

Both cases arrive to a contradiction, thereby proving the statement. □

Theorem 6.7. BROADCAST-DSM *satisfies the* wait-freedom (P3) *property.*

Proof. It follows from code inspection that the `SET`, `WITHDRAW` and `READ` operations are wait-free. Thus, every instance of `SET`, `WITHDRAW` and `READ` operation in a legal history will eventually terminate (possibly with a failure). □

Theorem 6.8. BROADCAST-DSM *satisfies the* RMR-efficiency (P4) *property.*

Proof. It is trivial to see that READ incurs $\mathcal{O}(1)$ RMRs. The WITHDRAW operation has no loops and thus also incurs $\mathcal{O}(1)$ RMRs. The SET operation contains a loop from line 178 to line 186 where process p_w accesses variables *announce* and *wakeup*. Both these variables are local to process p_w and thus the SET operation incurs $\mathcal{O}(1)$ RMRs. Finally, the WAIT operation also contains only one loop at line 167 where only variable $target[i]$ is repeatedly accessed by process p_i. The variable $target[i]$ is local to process p_i and therefore the WAIT operation also incurs only $\mathcal{O}(1)$ RMRs. Thus, each instance of SET, WAIT, WITHDRAW and READ operation has worst-case RMR complexity of $\mathcal{O}(1)$. □

It can be easily verified that BROADCAST-CC also satisfies safety (P1), liveness (P2), wait-freedom (P3) and RMR-efficiency (P4) properties.

CHAPTER 7

RECLAIMING MEMORY FOR AN RME ALGORITHM

Note that the framework in Section 4.3 uses three types of locks—a weakly recoverable filter lock, a strongly recoverable n-process base lock and a strongly recoverable dual-port arbitrator lock. The first lock can be implemented using the weakly RME algorithm described in Section 4.2. The second lock can be implemented using sub-logarithmic RME algorithm proposed by Katzan and Morrison[1] in (Katzan and Morrison, 2021). Finally, the third lock can be implemented using Yang and Anderson's algorithm augmented to handle failures (Golab and Ramaraju, 2019). The RME algorithm by Katzan and Morrison has space complexity of $\mathcal{O}(n \log^2 n / \log \log n)$ under both CC and DSM models. The augmented Yang and Anderson's algorithm has $O(n)$ space complexity under both CC and DSM models. However, the weakly RME algorithm described in Section 4.2 (WR-Lock) has unbounded space complexity because, upon generating a request, it allocates a new queue node at run time (additional in case of unsafe failures). It is non-trivial to determine when the memory of these nodes can be reclaimed without causing the algorithm to misbehave due to dangling pointers while, at the same time, maintaining its $\mathcal{O}(1)$ RMR complexity.

This chapter describes an algorithm for memory reclamation that can be used to bound the space complexity of the augmented WR-Lock by $\mathcal{O}(n^2)$ under both CC and DSM models. The techniques used by the memory reclamation algorithm are generic enough to be used with other RME algorithms as well. However, this section primarily focuses on memory reclamation for the WR-Lock. The memory reclamation algorithm uses ideas from two well known approaches for memory reclamation, namely *epoch*-based reclamation (Fraser, 2004) and *quiescent*-state-based reclamation (Arcangeli et al., 2003; McKenney and Slingwine, 1998), with the added benefits that it (a) has bounded space complexity and, (b) is recoverable.

[1]The algorithm has the added feature of abortability, which is not used by the framework

The quiescent-state-based memory reclamation algorithms assume that processes exhibit the following behavior:

(Q1) Every process *alternates* between quiescent and non-quiescent states during its execution.

(Q2) A process can access a shared object *only* when it is in a non-quiescent state.

(Q3) A shared object has to be *retired* before its memory can be reclaimed.

(Q4) While in a non-quiescent state, a process cannot access any shared object that was retired *prior* to it entering its non-quiescent state.

This behavior can be leveraged to obtain the following general memory reclamation scheme: upon retiring a shared object, a process can reclaim the memory of the object safely after every process in the system has been in its quiescent state at least once since then. This main idea is used to design a memory reclamation algorithm that has the desired RMR and space complexities.

If processes do not fail, then quiescent and non-quiescent states of a process can be taken to be NCS segment and passage, respectively, and it can be verified that properties (Q1)–(Q4) hold. If processes can fail, then one possible approach is to consider a process to be in non-quiescent state when it is executing its super-passage. Although the properties (Q1)–(Q4) hold, however, this straightforward extension does not yield a memory reclamation algorithm with bounded space complexity, at least directly. This is because, every time a process experiences an unsafe failure, it needs a "fresh" queue node. As a result, a process may "consume" an arbitrarily large number of queue nodes during a single super-passage.

To obtain a memory reclamation algorithm with bounded space complexity, the super-passage of WR-Lock is modeled as a sequence of *attempts*. Loosely speaking, an attempt begins when a process requests allocation of a new node at line 37 (using the `GetNewNode()` function) and ends when it retires[2] the allocated node at line 88 (using the `RetireLastNode()`

[2]Note that the notion of "retire" is distinct from that of "relieve" (defined in Subsection 4.2.4). Also, note that a node is relieved before it is retired.

function). For every process, this algorithm uses two monotonically non-decreasing counters to *demarcate* the beginning and end of its attempt. The counters for process p_i are denoted by $start[i]$ and $finish[i]$; the latter is a BROADCAST object described in Chapter 6. They represent the number of attempts a process has started (respectively, finished) across all super-passages so far. A process increments its start counter just before it enters the doorway of WR-LOCK, and increments its finish counter at the completion of the `RETIRELASTNODE()` function.

If a process is not executing an attempt, it is said to be in a *quiescent* state, at which point its two counters will have the same value. At all other times, the finish counter of each process lags behind its start counter by exactly one. As desired, when a process is executing the doorway or waiting room of WR-LOCK as part of an attempt, it is in non-quiescent state.

Note that "attempt" and "passage" are related but different concepts. They provide two different ways to model the execution of a process *within its super-passage*. The boundaries of attempts and passages do *not* align. A failure ends a passage but not an attempt. A process may fail multiple times during an attempt. As such, an attempt of a process can overlap with multiple passages of that process, but a passage of a process can overlap with most two attempts of that process.

Definition 7.1 (useful attempt)**.** *An attempt of a process is said to be* useful *if the process takes at least one step of its CS segment during the attempt; otherwise, it is said to be useless.*

It can be shown that an attempt of a process becomes useless if and only if the process crashes while executing the `FAS()` instruction at line 46. Every completed super-passage in an infinite fair history consists of *at least one* useful attempt. In particular, the last attempt of a completed super-passage is useful. A node allocated during an attempt is said to be have been *retired* if the process has completed the execution of the `RETIRELASTNODE` function during that attempt at least once without crashing.

It can be easily verified that:

Proposition 7.1. WR-Lock *satisfies (Q1)–(Q4).*

Proposition 7.2. *A process consumes at most one queue node during an attempt.*

Note that, even after a node has retired, another process may still hold a reference to that node and may dereference it in the future. This memory reclamation algorithm relies on the notion of a *grace period* to determine when is it safe to reclaim the memory of a node after it has been retired (McKenney and Slingwine, 1998; Arcangeli et al., 2003; Hart et al., 2007). A time interval $[t, t']$ is said to be a *grace period* if, after time t', no process holds a reference to any node that was retired at or before time t. This work defines a related notion of allowance period as shown in Figure 7.1 to highlight the grace period *with respect to a retired node* as follows.

Definition 7.2 (allowance period)**.** *Given a history H and a node x, an interval $[t, t']$ in H is said to be the* allowance period *with respect to x if x was retired at time t and no process holds any reference to x after time t'.*

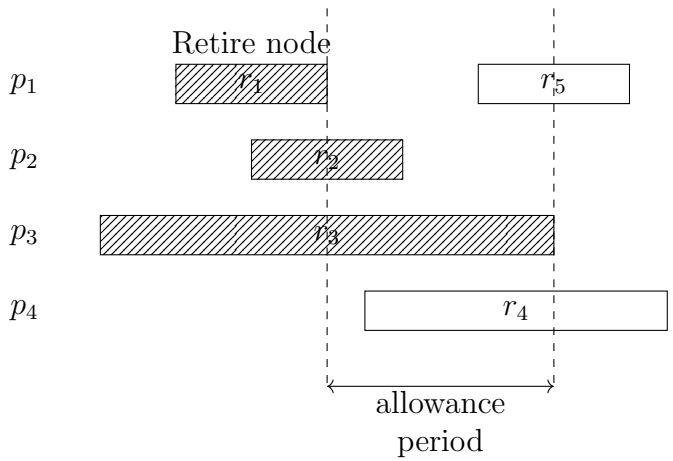

Figure 7.1. Illustration of the allowance period of a failure

It follows from proposition 7.1 that:

Lemma 7.3. *The allowance period with respect to a node retired at time t expires once every process has been in a quiescent state* at least once *after time t.*

Typically, in existing epoch based memory reclamation algorithms, a process uses a *non-blocking* method to *detect* that the allowance period of a node it retired earlier has expired. The length of the allowance period depends on the execution speeds of other processes. As such, it may extend arbitrarily long, during which period the process may retire additionally more (possibly unbounded number of) nodes. This makes it hard to achieve bounded space complexity.

This work uses a *blocking* approach to detect that the allowance period of a retired node has expired. It is feasible in this case because the underlying application, namely mutual exclusion, is inherently blocking (provided that any additional blocking does not create a deadlock). One of the main components of the memory reclamation algorithm is a routine to detect expiration of the allowance period with respect to a retired node (or, more precisely, a set of retired nodes), denoted by `APTD` *routine*, using a blocking synchronization that yields a bounded space complexity. In particular, the routine is designed to satisfy the following property:

Lemma 7.4. *Suppose an instance of* `APTD` *routine began at time t and completed at time t'. Then, the allowance period of every node retired at or before time t has expired at or after time t'.*

A simple approach to implement the `APTD` routine is as follows. A process, while in its quiescent state, waits for the *finish* counter of every process to "catch up" to its *start* counter, one-by-one. After the `APTD` routine completes, it follows from Lemma 7.4 that the process can safely reclaim the memory of any node it retired in the previous attempt. Due to its blocking nature, a process executes the `APTD` routine in the `Enter` segment, before starting a new attempt in order to maintain BR and BE properties. Executing the routine

in a quiescent state cannot create any deadlock since, both *start* and *finish* counters have the same value while a process is in quiescent state, and the process does not own any node in any sub-queue Subsection 4.2.4. Thus, no process can be busy-waiting on another process executing the `APTD` routine. This approach, however, increases the RMR complexity of each passage of WR-Lock to $\mathcal{O}(n)$.

7.1 Achieving constant RMR complexity

A possible approach to reduce RMR complexity is to *amortize* the overhead of executing the `APTD` routine over $\Omega(n)$ attempts. For example, a process can execute the `APTD` routine after every n attempts. Once the `APTD` routine completes, all nodes retired prior to the n attempts can be safely reclaimed. This, however, requires another set of n nodes that can be used to service requests for allocating queue nodes during these n attempts. Thus, to use this optimization, a process has to maintain *two* pools of nodes, each containing n nodes. While nodes in one pool are waiting for their allowance period to expire, nodes in the other pool can be used to serve requests for node allocation. The two pools are referred to as *active* and *backup* with obvious meaning. The role of the two pools is then *switched* after every n attempts. With this optimization, each passage of WR-Lock has $\mathcal{O}(1)$ RMR complexity in the *amortized case* but $\mathcal{O}(n)$ RMR complexity in the *worst case*.

However, there is a way to keep the worst-case RMR complexity of each passage at $\mathcal{O}(1)$. The main idea is to execute the `APTD` routine *incrementally* over $\Theta(n)$ attempts in such a way that it (a) is recoverable, (b) adds only $\mathcal{O}(1)$ RMRs to each passage of WR-Lock, (c) uses only $\mathcal{O}(n^2)$ space, and (d) maintains the CI-FCFS property of WR-Lock.

To maintain the fairness guarantee, a third counter is used. This counter, referred to as the *checkpoint* counter, is incremented after a process has completed the doorway of the original WR-Lock without crashing during a passage. The checkpoint counter for process

p_i is denoted by *checkpoint*$[i]$ and is basically a BROADCAST object described in Chapter 6. At any given time, the counters satisfy the following invariants for each process p_i:

$$start[i] - 1 \;\leq\; finish[i] \;\leq\; checkpoint[i] \;\leq\; start[i] \tag{7.1}$$

In case an attempt is rendered useless due to an unsafe failure, the checkpoint counter is incremented before the finish counter even though the process crashed while executing the doorway, in order to maintain the above invariant.

7.2 Phases and strides

To design a recoverable `APTD` routine that can be executed incrementally, the execution of the `APTD` routine is divided into *four* phases as follows:

- *Phase 1 (snapshot phase):* the process reads and records the value of the start counter of each process.
- *Phase 2 (catch-up phase):* the process waits for the checkpoint counter of each process to catch up to its start counter.
- *Phase 3 (yield phase):* the process waits for the finish counter of each process to catch up to its start counter.
- *Phase 4 (switch phase):* the process switches the role of the two pools.

The second phase is required to achieve the desired fairness guarantee. To enable incremental execution of the `APTD` routing, each phase is further divided into multiple *strides*[3]. Intuitively, a stride constitutes a "unit of execution" of the `APTD` routine and incurs $\mathcal{O}(1)$ RMRs under both CC and DSM models. The first three phases consist of n strides each and the fourth phase consist of two strides. Thus the routine as a whole consists of $3n+2$ strides. A *single* stride of the `APTD` routine is executed by invoking the function `ExecuteOneStride`.

[3]A stride means a long step, which is an apt description of what it denotes in this context.

Each process maintains a stride counter to keep track of the numbers of strides of the *current instance* of the `APTD` routine it has executed so far; the counter is reset during the switch phase as the designation of the two pools is flipped. The variable $stride[i]$ is used to denote the stride counter of process p_i. The pseudocode of the function `ExecuteOneStride` is shown in Algorithm 7.1. The function is designed in an idempotent manner, such that, executing it multiple times with the same value of the stride counter does not cause any undesirable behavior. Further, all repeated invocations of the `ExecuteOneStride` function, where the stride counter of the invoking process has the same value, are considered as essentially the same stride; until the value of the stride counter changes.

7.3 When to execute a stride?

The main idea is to execute one stride of the `APTD` routine "between" two consecutive attempts. Specifically, this memory reclamation algorithm satisfies the following property:

Proposition 7.5. *Suppose a process begins two consecutive attempts at times t and t' with $t < t'$. Then, it executes at least one stride of the* `APTD` *routine without failing between times t and t'.*

To ensure that the currently active pool does not run out of nodes at least until all strides of the `APTD` routine have been executed successfully, each pool now consists of $3n + 2$ nodes. When combined with proposition 7.5, this is necessary and sufficient because the `APTD` routine consists of $3n + 2$ strides and a process can consume at most one node in an attempt.

If fairness can be foregone, then a simple memory reclamation algorithm that meets all other desirable requirements (except for fairness) works as follows. A process executes one stride of the `APTD` routine in its `Enter` segment if it is in a quiescent state. Executing a stride of the `APTD` routine in quiescent state, specifically after finishing an attempt but before starting a new attempt, helps to easily avoid a deadlock.

7.4 Achieving fairness

A fairness property is typically defined with respect to a doorway, which is a *wait-free* piece of code that a process executes at the beginning of an `Enter` segment. As described earlier, a process executes a stride of the `APTD` routine at the *beginning* of its `Enter` segment when in a quiescent state before starting a new attempt. Since a stride may involve blocking (*i.e.*, busy waiting in a loop), it makes it infeasible to define the notion of doorway in the algorithm described so far.

This shortcoming is remedied as follows. If an attempt of a process is useful, then the process executes a stride of the `APTD` routine immediately after completing the doorway of WR-Lock rather than after completing that attempt. This stride is referred to as a *regular stride*. However, if an attempt of a process is useless, then the process executes a stride of the `APTD` routine immediately after completing that attempt as before (in quiescent state). This stride is referred to as a *penalty stride*. With this change, it can be verified that proposition 7.5 still holds.

Note that a process may execute both types of strides during its `Enter()` segment. Specifically, a process executes *at most one* regular and *at most one* penalty stride during its `Enter` segment implying that the (worst-case) RMR complexity of a passage remains $\mathcal{O}(1)$.

To ensure the BCSR property, the algorithm guarantees that a process executes a regular stride during an attempt exactly once. In other words, once it has completed a stride of the `APTD` routine, it does not execute another stride during that attempt even it were to fail and start a new passage (but same attempt); otherwise the process may not be able to reenter its CS segment within a bounded number of its own steps. To that end, whenever it starts a new attempt, it records the current value of its stride counter and executes a stride later only if the recorded and current values of its stride counter match. This also helps to avoid a deadlock by guaranteeing that a process that owns a node in a sub-queue never waits on a process that appended its node to the same sub-queue *after* its own. Specifically, it is proven

later that if a process p_i is waiting on a process p_j (either in the yield phase of the `APTD` routine or in the waiting room of WR-Lock) and attempts of both processes are useful, then p_i has executed its `FAS()` instruction after p_j.

This augmented weakly recoverable lock with bounded space complexity is referred to as WR-Lock-MR, and its pseudocode is given in Algorithms 7.1 and 7.2. To avoid repetition, only the pseudocode for the `Enter` segment, `GetNewNode` function and `RetireLastNode` function are provided. The pseudocode for `Recover` and `Exit` segments is same as that for WR-Lock and has been omitted.

First, consider the `Enter` segment. A process first executes a penalty stride, if applicable (lines 237 to 242). It then begins a new attempt, if in a quiescent state, and also records the current value of the stride counter (lines 243 to 246). Next, it executes the doorway of WR-Lock (line 247) and increments the checkpoint counter if lagging (line 248). It then executes a regular stride, if needed (lines 249 to 251). Finally, it executes the waiting room of WR-Lock (line 252).

Next, consider the `GetNewNode()` function. The stride counter is also used as an *index* into the active pool. So an invocation of `GetNewNode` simply returns the reference to the node stored at the corresponding location in the pool (line 257).

Finally, consider the `RetireLastNode` function. A process checks if the current attempt is useless, and, if so, sets the penalty flag and records the current value of the stride counter (line 260). It then increments the finish counter if lagging (line 265).

The doorway of WR-Lock-MR is defined as follows. Let B denote the maximum number of steps a process executes in the `Enter` segment in order to reach line 248 provided that the passage is exclusive. Recall that, in an exclusive passage, the process does not execute a penalty stride. In the absence of the penalty stride, the code in the `Enter` segment contains only one bounded loop[4], namely in the `Set()` operation (lines 179 to 186). The

[4]There is no loop for the implementation of the Broadcast object under the CC model.

Algorithm 7.1: Pseudocode of process p_i for bounding space complexity of WR-Lock described in Section 4.2 (Part 1 of 2)

```
203 shared variables
204     F : an instance of WR-Lock
        /* remaining variables are arrays, whose i-th entry in the DSM model is local to process p_i */
        /* number of attempts started */
205     start: array [1...n] of integer variable, all elements initially 1
        /* number of doorways finished */
206     checkpoint: array [1...n] of Broadcast object, all elements initially 1
        /* number of attempts finished */
207     finish: array [1...n] of Broadcast object, all elements initially 1
        /* two pools of 3n+2 nodes */
208     pool: array [1...n][0,1][1...3n+2] of QNode
        /* index of pool type */
209     currPool: array [1...n] of integer variable, all elements initially 0
210     bkupPool: array [1...n] of integer variable, all elements initially 1
        /* snapshot of start counter */
211     snapshot: array [1...n][1...n] of integer variable
        /* number of strides finished */
212     stride: array [1...n] of integer variable, all elements initially 1
        /* flag to penalize crashing */
213     penalty: array [1...n] of boolean variable, all elements initially FALSE
        /* last stride finished */
214     latest: array [1...n] of integer variable, all elements initially 1
215 end

        /* blocking method for APTD */
216 Function ExecuteOneStride()
217 begin
218     j ← (stride[i] − 1) mod n + 1
219     if stride[i] ∈ [1, n] then
            /* snapshot phase: snapshot start counters */
220         snapshot[i][j] ← start[j]
221     else if stride[i] ∈ [n + 1, 2n] and j ≠ i then
            /* catch-up phase: wait for doorway completion */
222         checkpoint[j].Wait(snapshot[i][j])
223     else if stride[i] ∈ [2n + 1, 3n] and j ≠ i then
            /* yield phase: wait for attempt completion */
224         finish[j].Wait(snapshot[i][j])
225     else if stride[i] ∈ [3n + 1, 3n + 2] then
            /* switch phase: switch current and backup pools */
226         if stride[i] = 3n + 1 then
                /* switch backup pool */
227             bkupPool[i] ← currPool[i]
228         else
                /* switch current pool */
229             currPool[i] ← 1 − bkupPool[i]
230         end if
231     end if
232     if stride[i] < 3n + 2 then
            stride[i] := stride[i] + 1
233     else stride[i] ← 1
234 end
```

Algorithm 7.2: Pseudocode of process p_i for bounding space complexity of WR-Lock described in Section 4.2 (Part 2 of 2)

```
235 Function Enter()
236 begin
237     if penalty[i] = TRUE then
            /* penalty stride */
238         if stride[i] = latest[i] then
                /* execute a stride of APTD
                   if not done already */
239             ExecuteOneStride()
240         end if
            /* register completion of the
               penalty stride */
241         penalty[i] ← FALSE
242     end if
243     if start[i] = finish[i].Read() then
            /* start a new attempt */
            /* record stride counter */
244         latest[i] ← stride[i]
            /* record start of attempt */
245         start[i] := start[i] + 1
246     end if
247     Execute the doorway of F (lines 35
          to 48 in Algorithm 4.1)
        /* register doorway completion */
248     checkpoint[i].Set(start[i])
        /* regular stride */
249     if stride[i] = latest[i] then
            /* execute a stride of APTD if
               not done already */
250         ExecuteOneStride()
251     end if
252     Execute the waiting room of F (lines 49
          to 57 in Algorithm 4.1)
253 end

254 Function GetNewNode()
255 begin
        /* return a reference to the node
           pointed to by the stride
           counter in the active pool */
256     return pool[i][currPool[i]][stride[i]]
257 end

258 Function RetireLastNode()
259 begin
        /* check if the attempt is
           inadmissible */
260     if start[i] ≠ checkpoint[i].Read() then
            /* attempt is being aborted
               due to a crash; set the
               penalty flag, record the
               stride counter and fix the
               checkpoint counter */
261         penalty[i] ← TRUE
262         latest[i] ← stride[i]
263         checkpoint[i].Set(start[i])
264     end if
        /* increment the finish counter
           if needed */
265     finish[i].Set(start[i])
266 end
```

doorway of WR-Lock-MR then consists of the first B steps of a process in the `Enter()` segment.

7.5 Analysis and Proofs of Correctness

Note that WR-Lock-MR uses $2n$ Broadcast objects for synchronization among processes given by *checkpoint*$[i]$ and *finish*$[i]$ for each $i = 1, 2, \ldots, n$. As such, the correctness of WR-Lock depends on the Broadcast objects behaving as expected (satisfying safety and liveness properties). In particular, the conditions mentioned in the statements of Theorems 6.5 to 6.7 should hold as and when needed. For example, Theorem 6.5 requires that the (sub)history with respect to a Broadcast object should be well-formed, while Theorem 6.6 requires that the (sub)history should be legal as well.

The correctness proof consists of two parts. The first part argues that the conditions required for a Broadcast object to behave correctly actually hold. The next part argues that WR-Lock-MR behaves correctly.

7.5.1 Broadcast objects pre-requisites

For each $i = 1, 2, \ldots, n$, process p_i owns the two Broadcast objects, *checkpoint*$[i]$ and *finish*$[i]$. As required, only process p_i invokes `Set()` operation on *checkpoint*$[i]$ and *finish*$[i]$, and, only process p_j, where $j \in \{1, 2, \ldots, n\} \setminus \{i\}$, invokes `Wait()` operation on *checkpoint*$[i]$ and *finish*$[i]$. For convenience, let $\mathcal{B}$ be the set of all Broadcast objects.

$$\mathcal{B} = \big\{ \textit{checkpoint}[i] \mid 1 \leq i \leq n \big\} \ \cup \ \big\{ \textit{finish}[i] \mid \mid 1 \leq i \leq n \big\}$$

Given a history H and a Broadcast object $b \in \mathcal{B}$, let $H \mid b$ denote the sub-history that consists of only those steps of H that processes take while executing an instance of one of the three[5] operations of b.

First, this (sub)section establishes that the assumption postulated in Theorem 6.5 holds.

Theorem 7.6. *Let H denote a history of* WR-Lock-MR*. Then, for every* Broadcast *object $b \in \mathcal{B}$, $H \mid b$ is well-formed with respect to b.*

[5]This algorithm only uses `Wait`, `Set` and `Read` operations of the Broadcast object.

Proof. Consider a BROADCAST object b owned by process p_i.

The property (W1) holds because (a) the argument passed to any instance of SET operation on *checkpoint*$[i]$ or *finish*$[i]$ counter is the current value of *start*$[i]$ counter, and (b) *start*$[i]$ counter has monotonically non-decreasing value.

The property (W2) holds because (a) the argument passed to any instance of WAIT operation on *checkpoint*$[i]$ or *finish*$[i]$ counter is the value of *start*$[i]$ counter read in the past, (b) *start*$[i]$ counter has monotonically non-decreasing value, and (c) the three counters of a process (*start*$[i]$, *checkpoint*$[i]$ and *finish*$[i]$) satisfy (7.1). □

This (sub)section next establishes that the assumptions postulated in Theorem 6.6 also hold. To that end, it needs to proven that, *under the assumptions made in the definition of the SF property*, (i) the (sub)history with respect to a BROADCAST object is legal, (ii) a checkpoint counter value eventually catches up to the corresponding start counter value and, (iii) a finish counter value eventually catches up to the corresponding start counter value. Each of the three conditions are now proven one-by-one.

A line ℓ of a function f is said to be *inevitable* if it satisfies the following property: if a process fails *while* executing the line ℓ, then, in every invocation of the function f by the (same) process thereafter, the line ℓ *lies* on *all* possible execution paths that can be taken by the process inside the function f, until the execution of the line has been completed-successfully. Typically, a line is considered to have completed-successfully if the execution of the line was failure-free, except in some cases as follows. Assume that line 239 or line 250 of the `Enter` function has been completed-successfully if only if the value of the stride counter is modified inside the function.

Proposition 7.7. *Lines 239 and 250 of the* `Enter` *function are inevitable.*

Proposition 7.8. *Lines 222 and 224 of the* EXECUTEONESTRIDE *function are inevitable.*

Based on these results, the desired results are proven.

Theorem 7.9. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of* WR-Lock-MR*. Then, for every* Broadcast *object $b \in \mathcal{B}$, $H \mid b$ is legal with respect to b.*

Proof. Consider a Broadcast object b owned by process p_i.

The property (L1) holds because, in an infinite fair history, a process can stop taking steps only *after* executing a failure-free passage.

The properties (L2) and (L3) hold because, as implied by propositions 7.7 and 7.8, if a process fails during an instance of `Wait` (x) operation, then it is guaranteed to repeatedly invoke instances of `Wait` (x) until the stride counter has changed. Moreover, a process can fail only a finite number of times during its super-passage, thereby implying that any run of `Wait()` operations in H is of finite length and ends with a failure-free instance. □

Theorem 7.10. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of* WR-Lock-MR *and consider a process p_i. Then, for every positive value x assumed by $start[i]$ counter in H, there exists at least one instance of* `Set` *(x) operation on $checkpoint[i]$ in H that completes-successfully.*

Proof. After a process *begins* an attempt, it has to execute only a bounded number of steps to increment its checkpoint counter unless it fails. However, by assumption, a process can fail only a finite number of times during each of its super-passage and hence during each of its attempt. □

The proof for the last assumption (the finish counter value eventually catches up to the corresponding start counter value) is more involved since a process may have to wait *after* incrementing the checkpoint counter value but *before* incrementing the finish counter value. A *wait-for dependency* is classified into three types:

(T1) Waiting for a checkpoint counter value to catch up to the associated start counter value (line 222 executed as part of penalty stride or regular stride).

(T2) Waiting for a finish counter value to catch up to the associated start counter value (line 224 executed as part of penalty stride or regular stride).

(T3) Waiting inside the waiting-room as part of WR-Lock (line 252).

Theorem 7.10 implies that waiting of type item (T1) is finite. So, is is sufficient to only consider the other two types of waiting and prove that they are finite as well. Note that, if an attempt of a process is rendered useless, then the process eventually abandons the queue node it allocated at the beginning of the attempt and increments its finish counter for that attempt. Therefore, it suffices to only focus on useful attempts and argue that every useful attempt eventually completes. To that end, the only situation to be concerned about is one in which a process p is waiting on another process q such that:

(A1) Both p and q are currently executing a useful attempt.

(A2) Both p and q have appended their respective nodes in the queue successfully using an `FAS` instruction.

The next lemma proves a crucial property common to both types of waiting.

Lemma 7.11. *If a process p is waiting on another process q such that (A1) and (A2) hold, then p executed its most recent* `FAS` *instruction* after *that of q.*

Proof. Clearly, the statement holds if p is waiting for q in the waiting room of WR-Lock. Thus, it is sufficient to focus on the case in which p is waiting for q in the yield phase of a penalty or regular stride. Let the useful attempts of p and q be denoted by A_p and A_q, respectively. Consider four instants of time as follows:

- ▷ t_q is the time when q executed the `FAS` instruction for A_q
- ▷ t_c is the time when p completed its stride for the first time during which it waited for the checkpoint counter of q to catch up to its start counter for A_q
- ▷ t_a is the time when p started the attempt A_p
- ▷ t_p is the time when p executed the `FAS` instruction for A_p

It is shown that $t_q < t_c < t_a < t_p$, thereby proving that $t_q < t_p$.

- ▷ $t_q < t_c$ because a process executes the `FAS` instruction *before* incrementing the checkpoint counter for a admissible attempt
- ▷ $t_c < t_a$ because a process executes only *one* regular stride of `APTD` routine during a useful attempt (and, in the regular stride for A_p, p is waiting for the finish counter of q to catch up to its start counter for A_q)
- ▷ $t_a < t_p$ because a process executes the `FAS` instruction *after* starting an attempt

This proves that the statement holds. □

Using the above lemma, the following can be proven.

Theorem 7.12. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of* WR-Lock-MR*. Then, every attempt in H eventually completes.*

Proof. Let $\mathcal{I}(t)$ denote the set of attempts in H that execute their `FAS` instructions at or before time t and have not completed by time t.

Consider an arbitrary attempt, say A in H. Clearly, a useless attempt has no busy-waiting loop and, thus, is guaranteed to complete. Therefore, assume that A is a useful attempt. Let p denote the process to which A belongs, and let t_A denote the time when p performs the `FAS` instruction during A to append its node to the queue.

Consider $\mathcal{I}(t_A)$. All attempts in $\mathcal{I}(t_A)$ can be ordered based on the sequence in which their `FAS` instructions are executed. Lemma 7.11 implies that, after time t, an attempt in $\mathcal{I}(t_A)$ can only busy-wait on another attempt in $\mathcal{I}(t_A)$. Using induction (on the order in which `FAS` instructions are performed) it can be proved that every attempt in $\mathcal{I}(t_0)$ eventually completes.

Since A was chosen arbitrarily, it can be inferred that every attempt in H eventually completes. □

Clearly, it follows that:

Corollary 7.13. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of* WR-Lock-MR *and consider a process p_i. Then, for every positive value x assumed by $start[i]$ counter in H, there exists at least one instance of* `Set` *(x) operation on $finish[i]$ in H that completes-successfully.*

In other words, all three assumptions required for Theorem 6.6 to apply hold.

7.5.2 Correctness of WR-Lock-MR

Only the SF and BCSR properties of the RME problem are focused on here because the proofs for other properties (ME, BE, BR and CI-FCFS) are almost identical to those for WR-Lock.

To establish the SF property, the following lemma about a penalty stride is proven first.

Lemma 7.14. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of* WR-Lock-MR *and consider a process p_i. Then every penalty stride executed by a process in H eventually terminates.*

Proof. A process executing a penalty stride may either wait in the catch-up phase or the yield phase of the `APTD` routine. The first type of wait is finite due to Theorem 7.10, while the second type is finite due to Corollary 7.13. □

Thus,

Lemma 7.15. WR-Lock-MR *satisfies the SF property.*

Proof. When executing a passage, a process may wait at three different points: (a) while performing a penalty stride (line 239), (b) while performing a regular stride (line 250), or (c) while executing the waiting room of WR-Lock (line 252) Lemma 7.14 establishes that the first type of waiting is finite. Theorem 7.12 establishes that the other two types of waiting are also finite. □

Lemma 7.16. WR-Lock-MR *satisfies the BCSR property.*

Proof. If a process crashes inside its critical section, then it implies that the process is currently executing a useful attempt. Upon restarting, it does not execute any stride of the `APTD` routine—penalty or regular, as reasoned below.

It does not execute a penalty stride because it is not in a quiescent state. It does not execute a regular stride because it would have already executed one prior to entering the critical section for the first time during this attempt. By design, it does not execute another regular stride until it completes the current attempt by executing the `Exit()` segment.

Thus, the BCSR property of the WR-Lock ensures the BCSR property of the WR-Lock-MR. □

Lemma 7.17. WR-Lock-MR *satisfies the CI-FCFS property.*

It now follows that:

Theorem 7.18. WR-Lock-MR *satisfies ME, SF, BCSR, BE, BR and CI-FCFS properties. Further, it has* $\mathcal{O}(1)$ *(worst-case) RMR complexity and* $\mathcal{O}(n^2)$ *space complexity.*

Theorem 7.19. *The space complexity of (augmented)* BA-Lock *is given by* $\mathcal{O}(n^2 \log n / \log \log n + S(n))$*, where* $S(n)$ *denotes the space complexity of the base* NA-Lock *for* n *processes.*

Corollary 7.20. *If Katzan and Morrison's RME algorithm (Katzan and Morrison, 2021) is used to implement the* NA-Lock*, then the space complexity of* BA-Lock *is given by* $\mathcal{O}(n^2 \log n / \log \log n)$.

CHAPTER 8

RME UNDER SYSTEM-WIDE FAILURES

The RME problem for system-wide failures was defined in (Golab and Hendler, 2018). The system-wide failure model is a special case of the independent failure model. As the name suggests, it assumes that all processes only fail simultaneously. For example, consider a system where all processes are reliable and failures only happen due to a power outage. The power outage causes all processes to crash. Note that the system-wide failure model makes a stronger assumption than just multiple independent failures.

This stronger assumption provides certain advantages such as lower RMR complexity and an option to opt out from the lock after a failure. As shown in (Chan and Woelfel, 2021), the RMR complexity of a solution to the RME problem for the *independent* failure model has a lower bound of $\mathcal{O}(\log n / \log \log n)$. However, using an external failure detector, Golab and Hendler (Golab and Hendler, 2018) design an $\mathcal{O}(1)$ RMR solution for the RME problem under *system-wide* failures. This raises the question whether there exists an algorithm with $\mathcal{O}(1)$ RMR complexity for the system-wide failure model that does not depend on an external failure detector. This work answers the previous question in the affirmative.

This chapter presents an optimal $\mathcal{O}(1)$ RMR solution to RME under the system-wide failure model, without relying on an external failure detector. Additionally, the system model is modified to accommodate an option to opt out from participating in the lock after a failure. The modified system model is presented in Section 8.1.

The RME solution is divided into three major ideas. First, an $\mathcal{O}(1)$ RME algorithm that satisfies all correctness and desirable properties except bounded critical section re-entry and failure robust fairness properties is presented in Section 8.2. Then, Section 8.3 presents a transformation to add the bounded critical section re-entry (BCSR) property to the above solution. Recall that BCSR is a correctness property that allows a process that crashed during its critical section to reenter its critical section immediately upon recovery. Finally,

Section 8.4 presents a transformation that adds the failure robust fairness (FRF) property to the resultant algorithm mentioned above. Intuitively, the FRF property (Golab and Hendler, 2018) guarantees an unbiased entry into the critical section for every active process despite repeated failures.

The BCSR and FRF transformations do not incur any additional RMR overhead ($\mathcal{O}(1)$ RMR overhead). Moreover, both these transformations are general enough to be employed by any algorithm that solves the RME problem under the system-wide failure model. Additionally, each transformation consumes $\mathcal{O}(n^2)$ space, with no dynamic memory allocation.

8.1 Model

The system model followed in this chapter closely follows the model described in Chapter 2, with a few exceptions. This section covers the main differences between the model from Chapter 2 and the model assumed here.

8.1.1 Failure model

The *crash-recover* failure model remains to be assumed here. A process may fail at any time during its execution by crashing. However, the model for system-wide failures assumes that if one process fails[1] at time t, then all processes have failed at the same time t. Every crashed process is assumed to recover eventually and restart its execution[2]. A crashed process does not perform any steps until it has restarted. There may be multiple system-wide failures, and a system-wide failure may occur at any point, independent of process execution.

Note that unlike the model in Chapter 2, this chapter does not considers a failure to be associated with a single process. Rather, this model assumes that a failure causes all the processes to crash.

[1]The terms "crash" and "fail" are used interchangeably

[2]Note that even though all processes crash at the same time, they may recover at different times

8.1.2 Process Execution Model

Algorithm 8.1: Process Execution Model under System-Wide failures

```
267 while true do
268 |   Non-Critical Section (NCS)
269 |   if condition then
270 |   |   Recover()
271 |   |   Enter()
272 |   |   Critical Section (CS)
273 |   |   Exit()
274 |   else
275 |   |   Withdraw()
276 |   end if
277 end while
```

The execution model of a process with respect to a lock is depicted in Algorithm 8.1. As shown, a process may repeatedly take one of two paths; (i) either it executes the NCS, `Recover`, `Enter`, CS and `Exit` segments (lines 268 to 273); or (ii) it executes NCS (line 268) and `Withdraw` (line 275) segments. The first path contains the five segments identical to the process execution model from Section 2.3. The second path contains two segments, the NCS segment followed by the `Withdraw` segment. The `Withdraw` segment, similar to the `Recover` segment, models the steps executed by a process to perform any cleanup steps required due to past failures (if any) and restore the internal structure of the lock to a consistent state. However, unlike the `Recover` segment, the process does not attempt to enter the CS after executing the `Withdraw` segment.

The selection of the execution path is modeled via a boolean variable *condition* (line 269). If *condition* is always set to TRUE, this execution model reduces to the execution model in Algorithm 2.1 from Chapter 2. The variable *condition* can only be set/reset in the NCS segment. Additionally, it is continued to be assumed that, in the NCS segment, a process does not access any part of the lock or execute any computation that could potentially cause a race condition; and, in the `Recover`, `Enter`, `Exit` and `Withdraw` segments, a process accesses shared variables pertaining to the lock (and the lock only).

The option to use `Withdraw` addresses an important limitation of the process execution model shown in Section 2.3. Consider the following scenario. A process crashes in the `Exit` segment after it finished executing its CS. Upon recovery, this process needs to execute the `Recover`, `Enter`, CS and `Exit` segments. This process may potentially be blocked in `Enter` and later is forced to re-execute its CS despite already completing it once. With the `Withdraw` option, such a process can remove any "dependency" on itself, without the need to block and re-execute the CS. The role of the `Withdraw` segment is typically more than that of the `Exit` segment. For instance, if a process fails in `Recover`, the `Withdraw` segment may need to undertake additional steps to recover the lock.

Every crashed process upon restarting starts its execution from the beginning of the loop shown in Algorithm 8.1, specifically from the beginning of the NCS segment. Note that even if a process crashes in the CS, and may potentially block other processes from entering their CS, it is not assumed to be in its CS until it completes executing its `Recover` and `Enter` segments. Hereafter, this execution model only considers steps taken by a process during its `Recover`, `Enter`, `Exit` or `Withdraw` segments.

The sequence of passages between any two successive system-wide failures is known as an *epoch*. Figure 8.1 demonstrates such an epoch between two successive system-wide failures.

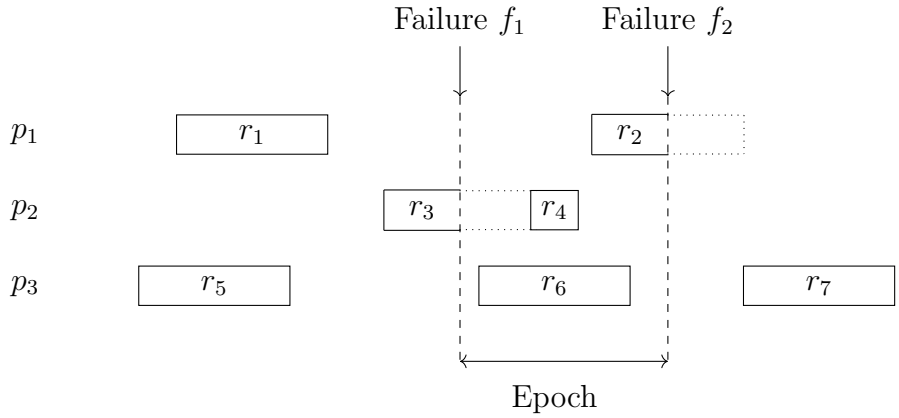

Figure 8.1. Illustration of an epoch between successive failures, f_1 and f_2

The definition of a passage needs to be modified to accommodate the `Withdraw` option. To that end, the terms CS-passage and Withdraw-passage have been defined.

Definition 8.1 (CS-passage)**.** *A* CS-passage *of a process is defined as the sequence of steps executed by the process from when it begins the execution of a* `Recover` *segment to either when it completes the execution of the corresponding* `Exit` *segment or experiences a failure, whichever occurs first.*

Definition 8.2 (Withdraw-passage)**.** *A* Withdraw-passage *of a process is defined as the sequence of steps executed by the process from when it begins the execution of a* `Withdraw` *segment to either when it completes the execution of the same* `Withdraw` *segment or experiences a failure, whichever occurs first.*

Definition 8.3 (passage)**.** *A* passage *of a process is defined as the sequence of steps executed by the process from when it begins the execution a CS-passage or a Withdraw-passage to when it completes*[3] *the execution of the corresponding CS-passage or Withdraw-passage.*

Definition 8.4 (failure-free passage)**.** *A passage of a process is said to be* failure-free *if the process has successfully executed a passage (CS-passage or Withdraw-passage) without experiencing any failure.*

Definition 8.5 (super-passage)**.** *A* super-passage *of a process is a maximal non-empty sequence of consecutive passages*[4] *executed by the process, where only the last passage of the process in the sequence can be failure-free.*

8.1.3 Problem definition

The definition of fair history remains the same and is stated here for completeness.

[3] perhaps with a failure

[4] The passages in a super-passage may be any combination of CS-passages and Withdraw-passages

Definition 8.6 (fair history)**.** *A history H is said to be* fair *if (a) it is finite, or (b) if it is infinite and every live process in H either executes infinitely many steps or stops taking steps after a failure-free passage.*

The definition of ME also remains the same. The definitions of SF and BCSR are modified as follows.

Mutual Exclusion (ME) For any history H, at most one process is in its CS at any point in H.

Starvation Freedom (SF) Let H be an infinite fair history in which every process fails only a finite number of times during each of its super-passage. If a process p starts a new passage in some step of H, then p eventually completes its passage.

Bounded Critical Section Re-Entry (BCSR) For any history H, if a process p crashes inside its CS segment, then, until p has either reentered its CS segment or invoked its `Withdraw` segment at least once, any subsequent execution of `Recover` and `Enter` segments by p either completes within a bounded number of p's own steps or ends with p crashing.

Note that a process may choose to execute a Withdraw-passage even if it had previously crashed inside the CS. In this case, the CS maybe left in an inconsistent state. However, this behavior does not violate the BCSR property and is permissible from an RME algorithm's standpoint. It is the responsibility, if any, of the underlying algorithm to maintain the consistency of the CS. Similarly, it is also permissible for a process to execute a Withdraw-passage immediately after it crashes at any point during a CS-passage; or vice versa.

In addition to the bounded recovery (BR) and bounded exit (BE) properties, it is also desirable for an RME algorithm to satisfy the following additional properties.

Failure Robust Fairness (FRF) For any fair history H containing infinitely many super-passages, there exists a bound b such that if a process p executes b consecutive

failure-free steps in the beginning of a CS-passage in H, then p eventually enters the CS segment or p invokes the `Withdraw` segment.

Bounded Withdraw (BW) For any infinite history H, any execution of the `Withdraw` segment by any process p either completes in a bounded number of p's own steps or ends with p crashing.

Note that the SF property assumes that the number of failures in a super-passage must be finite. A scenario may occur where there are infinitely many failures per super-passage, which unfairly blocks some process p_i while another process p_j successfully executes an infinite number of super-passages. In this case, even though an RME algorithm does satisfy the SF property, it is unfair. The FRF property addresses this limitation of the SF property[5]. The FRF property described here is an extension of the FRF property defined by Golab and Hendler (Golab and Hendler, 2018).

Note that the FRF property does not guarantee a FCFS ordering between processes. On the other hand, the fairness properties discussed in Chapter 5 do guarantee a FCFS ordering. However, these properties only apply if a "victim" process is not CI-concurrent or k-failure-concurrent. Thus, the FRF property is weaker but more applicable than previously discussed fairness properties (CI-FCFS and k-FCFS).

For algorithms that follow the Golab and Ramaraju model of process execution (Golab and Ramaraju, 2019), one may define a default implementation of the `Withdraw` segment as a sequence of `Recover`, `Enter` and `Exit` segments. This default implementation however may not support the BW property since the `Enter` segment may be blocking.

[5]Due to its stronger progress guarantees, some models may assume the FRF property to be a correctness property instead of a desirable property

8.2 MCS-SW

Any solution to the RME problem under the independent failure model also serves as a solution to the RME problem under the system-wide failure model. However, under the independent failure model, the per-passage RMR complexity has a lower bound of $\mathcal{O}(\log n / \log \log n)$. This section presents a solution to the RME problem under system-wide failures that has a $\mathcal{O}(1)$ RMR complexity and satisfies ME, SF, BR, BE, BW, 0-FCFS and CI-FCFS properties[6]. Additionally, the per process space complexity is $\mathcal{O}(1)$. This solution is called MCS-SW and is based on the well-known MCS queue-based lock (Mellor-Crummey and Scott, 1991a).

Golab and Hendler present a generic framework (Golab and Hendler, 2018) to transform any solution to the (non-recoverable) ME problem into a solution to the RME problem under system-wide failures, while also preserving the worst-case RMR complexity per passage. Their framework, in conjunction with the original MCS algorithm (Mellor-Crummey and Scott, 1991a) yields a $\mathcal{O}(1)$ RMR complexity algorithm. However, their framework assumes additional information from an external failure detector. The algorithm presented in this section achieves $\mathcal{O}(1)$ RMR complexity without the help of any additional external information.

8.2.1 Main Idea

The design of this algorithm is a modification of the well-known MCS queue-based lock (Mellor-Crummey and Scott, 1991a). In the absence of failures, this algorithm operates as per the MCS lock. In the event of a system-wide failure, all active processes abandon their nodes and reconstruct a new queue, effectively resetting the base lock. Additionally, this algorithm guarantees that each process has a maximum of two participating nodes at any

[6]Note that the solution does not satisfy the BCSR and FRF properties which are addressed later in Sections 8.3 and 8.4

instant. Therefore, it is able to recycle its node from a pool of two nodes[7]. In order to achieve bounded exit and bounded space, this algorithm incorporates techniques from Dvir and Taubenfeld's modification (Dvir and Taubenfeld, 2017) of the MCS lock.

8.2.2 Implementation

The pseudocode for the MCS-SW under the CC and DSM model is given in Algorithms 8.2 and 8.3. Processes in the MCS mutual exclusion algorithm use queue nodes (hereafter QNode) to synchronize their executions of CS segments. The algorithm maintains a first-come-first-served (FCFS) queue of outstanding requests using a linked-list of their associated nodes. Unlike the MCS algorithm from Section 4.2, a QNode in this algorithm contains the following two fields: (a) *next*, a reference to its successor node in the queue (if any), and (b) *pred*, a reference to the predecessor node (if any). The field *pred* is used by a process to spin while waiting for its turn to enter its critical section. The variable *tail* is a pointer that can hold a reference of type QNode. The queue itself is represented using variable *tail* that contains a reference to the last node in the queue if non-empty and **null** otherwise. The variable *mine* is an array of QNodes such that $mine[i]$, contains the address of the QNode associated with process p_i's most recent request. The variable *pool* is also an array of a pool containing two QNodes per process. Process p_i consumes the nodes in $pool[i]$ in an alternating manner each time it begins a new attempt to enter into CS. Finally, the variable *curr* is an array of integer. The value of the variable *curr* is either 0 or 1 and is used by process p_i to determine which node to consume from the nodes in $pool[i]$. For the DSM model, the i-th entry of the array variables *mine*, *pool* and *curr* is local to process p_i.

To acquire the lock, a process p_i first retrieves a node from the $pool[i]$ and initializes its *next* and *pred* fields to **null** and **null** respectively (lines 290 to 292). It then appends this

[7]It is safe to say that when a process requires a new node, it will only have at most one node participating in the algorithm

Algorithm 8.2: Pseudocode for the MCS lock that recovers under system wide failures (Part 1 of 2)

```
   /* Data structure to represent nodes
      in a queue */
278 struct QNode {
        /* reference to predecessor node;
           used for spinning while
           waiting to enter CS */
279     pred: reference to QNode, initially null
        /* reference to next node */
280     next: reference to QNode, initially null
281 };
282 shared variables
        /* reference to the last node in
           the queue */
283     tail: reference to QNode, initially null
        /* remaining variables are
           arrays, whose i-th entry in the
           DSM model is local to process
           p_i */
        /* pool of reusable nodes */
284     pool: array [1...n][0,1] of QNode, all
          elements initially {null, null}
        /* reference to my own nodes */
285     mine: array [1...n] of references to
          QNode, i-th element initially pool[i][0]
        /* index of node to use from the
           pool */
286     curr: array [1...n] of integer, all
          elements initially 0
287 end
288 Function Enter()
289 begin
        /* Retrieve a node from pool */
290     mine[i] ← pool[i][curr[i]]
        /* Initialize my node */
291     mine[i].next ← null
292     mine[i].pred ← null
        /* Append my node to the queue */
293     QNode * temp ← FAS(tail, mine[i])
        /* Store reference to
           predecessor's node */
294     mine[i].pred ← temp
295     if mine[i].pred ≠ null then
            /* Try linking predecessor's
               node to my node */
296         if CAS(mine[i].pred.next, null,
              mine[i]) then
                /* Link creation
                   successful; Wait for
                   predecessor to signal */
297             await mine[i].pred = null
298         end if
299     end if
300 end
```

node to the queue by performing an `FAS` instruction on *tail* (line 293) using the reference to its own node as an argument (to the instruction). Note that the instruction returns the contents of *tail* just before it is modified. This return value is stored in the field *pred* of the node $mine[i]$ (line 294). If the value of *pred* is **null** (line 295), then it indicates that the lock is free and the process has successfully acquired the lock. If not, then it indicates that the

Algorithm 8.3: Pseudocode for the MCS lock that recovers under system wide failures (Part 2 of 2)

```
301 Function Recover()
302 begin
303 |   Cleanup()
304 |   curr[i] ← 1 − curr[i]
305 end

306 Function Exit()
307 begin
308 |   Cleanup()
309 end

310 Function Withdraw()
311 begin
312 |   Cleanup()
313 end

314 Function Cleanup()
315 begin
    |   /* Remove my node from tail */
316 |   CAS(tail, mine[i], null)
    |   /* Disable successor process from
    |      creating a link */
317 |   CAS(mine[i].next, null, mine[i])
    |   /* Check if link exists between
    |      my node and successor node */
318 |   if mine[i].next ≠ mine[i] then
    |   |   /* Release the successor */
319 |   |   CAS(mine[i].next.pred, mine[i],
    |   |    null)
320 |   end if
321 end
```

lock is not free and the field *pred* contains the reference to the predecessor of the process' own node in the queue. In that case, it notifies the owner of the predecessor node of its presence. To that end, it attempts to store the reference to its own node in the *next* field of the predecessor node with a `CAS` instruction (line 296). The `CAS` would only be successful if the *next* field contains **null**. If the `CAS` is successful, a forward link is created between the two nodes. The process then starts spinning on the *pred* field of its own node (line 297) waiting for it to be reset to **null** by the owner of the predecessor node as part of releasing the lock. On the other hand, if the `CAS` instruction to create the forward link is unsuccessful, then the lock has been freed and that process now holds the lock.

Unlike the WR-Lock lock from Section 4.2, where a process aborts its attempt only if it fails at a certain point during its execution, a process in the MCS-SW lock always abandons its node upon recovering from a failure. Since, each process abandons its node upon recovering from a failure, the implementation of the `Recover`, `Exit` and `Withdraw` is

identical[8]. This is modeled with the help of a common `Cleanup()` module. Here, process p_i first tries to reset the *tail* variable (line 316) to **null** (if *tail* still contains the reference to this process' node) using a `CAS` instruction. Then, it attempts to store a special value (*e.g.*, reference to its own node) in the *next* field of its own node using a `CAS` instruction (line 317). This `CAS` instruction synchronizes with the `CAS` instruction from `Enter` on line 296. Both `CAS` instructions are designed to succeed only if the *next* field contains **null** value, thereby ensuring that the *next* field can only be modified once. Thus, if the `CAS` instruction performed by a process as part of the `Cleanup()` module is unsuccessful (line 318), then there exists a forward link to a successor node. It then follows this link and releases its successor by resetting the *pred* field of its successor node to **null** through a `CAS` instruction (line 319). The resetting of the *pred* field is done with a `CAS` instruction to allow node recycling as explained later.

In the absence of failures, the steps of `Recover` are inconsequential. When a system-wide failure occurs, let p_t be the last process to perform `FAS` on *tail* before the failure. Therefore, $tail = mine[t]$. Upon recovering from the failure, all the other processes would abandon their old nodes as a part of a `Recover` or `Withdraw` execution. Some of these processes may begin new attempts and perform `FAS` on the *tail*. Thus, a new queue is created with p_t at the head of the queue. The other processes would be blocked by p_t. Process p_t will eventually execute `Recover` or `Withdraw` after recovering from the failure, thereby unblocking all the other processes.

Note that if a process crashes during the execution of `Withdraw`, it will eventually execute `Cleanup` either from `Withdraw` again, or from `Recover`. The module `Cleanup` is idempotent and thus, the lock would remain in a consistent state.

Each process maintains a pool of two nodes. These nodes are safe to reuse during the same epoch as shown by Dvir and Taubenfeld (Dvir and Taubenfeld, 2017). Now consider

[8]except line 304 used for node recycling

the case when a system-wide failure f_1 occurs. If *tail* is **null** at the time of the failure, then the queue is empty and a new queue is created for subsequent requests. Otherwise, let $tail = mine[t]$, for some process p_t at the time of the failure. The queue for subsequent requests now continues to grow from this node. The node cycle does not get affected by this failure since abandoning of a node is equivalent to relinquishing a node during `Exit` without entering the CS[9].

Upon recovering failure f_1, it is guaranteed that each process releases its successor as part of the `Cleanup` module. Since each process abandons its node, releasing the successor is redundant except for the process p_t, whose node the *tail* points to, and where to queue continues to grow from.

Note that processes spin on the field *pred*, a reference of type QNode[10]. This is due to the following scenario. Let process p_2 be the successor to process p_1. Multiple failures cause p_1 to release p_2's node multiple times. When p_2 begins a new passage and enters the queue using the same node again, let p_3 be its predecessor. In this scenario, p_2's node is the successor to both p_1 and p_3's nodes, while the predecessor to p_2's node is p_3's node. Here, although p_1 erroneously attempts to release p_2's node, its `CAS` instruction at line 319 would be unsuccessful. This is due to the fact that the *pred* field contains the address of p_3's node.

8.2.3 Analysis and Proofs of Correctness

For the purposes of proofs of correctness, assume that the *pool* has an infinite supply of nodes and each request in the queue uses a distinct new node.

Theorem 8.1. MCS-SW *lock satisfies the ME property.*

[9]Note that node recycling is safe here since the system-wide failure model guarantees other processes would also fail and lose access to pending deference operation on the node. This strategy may not work for algorithms designed under the independent failure model, like the WR-Lock lock from Section 4.2.

[10]The original MCS lock uses a **boolean** type to spin on

Proof. By code inspection, every process performs `FAS` on *tail* prior to entering the CS (line 293). Let t_1 and t_2 be the times when processes p_1 and p_2 perform `FAS` on *tail* using nodes n_1 and n_2 respectively, during some epoch e. Without loss of generality, assume that $t_1 < t_2$. Further, assume that no process performs `FAS` on *tail* at time $t \in [t_1, t_2]$.

If p_1 removes n_1 from the queue by performing CAS on *tail* (line 316) before time t_2 then ME property holds.

Otherwise, $n_2.pred = n_1$. In this case, p_2 would try to link n_2 to n_1 (line 296). If unsuccessful, then p_1 must have disabled n_2's *next* field during `Exit` (line 317). Otherwise, p_2 must wait (at line 297) until p_1 releases n_2 (line 319). Since $n_2.pred = n_1$, only p_1 can unblock p_2. In either case, p_1 has completed its CS and begun its `Exit` segment before p_2 begins its CS. Thus, there exists a transitive order between processes entering the CS, based on the order of `FAS` performed on *tail*. Therefore, MCS-SW lock satisfies the ME property. □

Theorem 8.2. MCS-SW *lock satisfies the SF property.*

Proof. Proof by contradiction. Let H be an infinite fair history in which every process fails only a finite number of times during each of its super-passage. Assume that process p is starved and does not complete its last passage in its super-passage. Clearly p does not invoke `Withdraw` during its last passage (since MCS-SWsatisfies BW as shown later in Theorem 8.3). Thus, p is starved in the middle of a CS-passage. Let this passage belong to epoch e. Moreover, since this is p's last passage of its super-passage, there can be no failures during e, before p completes its passage.

Since H is a fair history, p is eventually going to perform `FAS` on *tail* (line 293). Without loss of generality, assume that every process that performs `FAS` on *tail* before p, eventually completes its super-passage (does not starve). Process p can only starve on line 297 where it waits for its predecessor to unblock it. Thus, (a) p's `FAS` would not have returned **null**,

and (b) p must have been successful in creating a forward link from its predecessor to its own node (line 296). Since p was successful in creating a forward link, its predecessor must have been unable to block the *next* pointer of its own node (line 317). When p's predecessor completes its `Exit` segment it will eventually unblock p on line 319 and so, p is not starved. This contradiction establishes the SF property for MCS-SW lock. □

Theorem 8.3. MCS-SW *lock satisfies the BR, BE and BW properties.*

Proof. As the code inspection shows, executions of `Recover`, `Exit` and `Withdraw` invoke the `Cleanup` module which does not involve any loops. Thus, a process can complete executing `Recover`, `Exit` and `Withdraw` within a bounded number of its own steps. Hence, WR-Lock satisfies the BR, BE and BW properties. □

Theorem 8.4. *The RMR complexity of any passage of the* MCS-SW *lock is* $\mathcal{O}(1)$.

Proof. As the code inspection shows, executions of `Recover`, `Exit` and `Withdraw` do not contain any loops and only contain a constant number of steps. The execution of `Enter`, however contains one loop at line 297, but otherwise contains a constant number of steps. This loop involves waiting on the *pred* field of a QNode, until it becomes **null** and the variable can be written to only by the actual predecessor. This incurs only $\mathcal{O}(1)$ RMRs in the CC model. In the DSM model, this variable is mapped to a location in local memory module. Hence, the RMR complexity of an execution of `Enter` is also $\mathcal{O}(1)$ in both CC and DSM models. □

Lemma 8.5. MCS-SW *lock does not satisfy the BCSR property.*

Proof. Consider the following scenario. Process p_1 is in CS and process p_2 is its successor. The variable *tail* points to p_2's node. At this moment, a system-wide failure occurs. Upon recovery p_2 abandons its node, during which it clears the *tail* pointer. When p_2 begins a new request, *tail* would be **null** and thus, p_2 enters CS and p_1 must wait to enter CS. Hence, MCS-SW lock does not satisfy the BCSR property. □

Lemma 8.6. MCS-SW *lock does not satisfy the FRF property.*

Proof. Proof by contradiction. Assume that the MCS-SW lock satisfies the FRF property, *i.e.*, there exists a bound b such that if

(a) there are infinitely many super-passages,

(b) process p executes b consecutive failure-free steps, and

(c) p does not invoke the `Withdraw` segment

then p will eventually enter its CS segment. Consider the following scenario.

Process p_1 acquires the lock. Process p_2 begins a CS-passage and attempts to acquire the lock. Process p_2 successfully executes at least b steps while attempting to acquire the lock[11]. Process p_1 releases the lock and completes its super-passage. However, before p_2 could acquire the lock, a system-wide failure occurs. Upon recovery, p_1 acquires the lock before p_2, and the scenario above repeats infinitely often. Thus, p_1 has undertaken infinitely many super-passages, p_2 has completed b consecutive failure-free steps and p_2 did not invoke the `Withdraw` segment. Yet, p_2 does not enter the CS segment. This is a contradiction. Hence, the MCS-SW lock does not satisfy the FRF property. □

Theorem 8.7. *A pool of two QNodes per process is sufficient to be reused between attempts.*

Proof. A node can be reused once it is guaranteed that either (a) no process has access to the node (in its previous context), or (b) a `CAS` instruction on any field of the node (in its previous context) is unsuccessful. Note that a process accesses its predecessor's node only during `Enter` and its successor's node only during `Cleanup`. Additionally, at the beginning of every passage, `Recover` invokes `Cleanup` which is useful only in case of a failure.

Let process p use two nodes, n_1 and n_2, during consecutive CS-passages r_1 and r_2 respectively. Consider the following notation. Let $pred(r)$ and $succ(r)$ denote the predecessor and successor of process p during passage r.

[11] These steps may involve spinning while waiting to acquire the lock

When p had executed passage r_2, it must have abandoned node n_1 (used in passage r_1) while executing `Recover`[12]. However, p's predecessor during r_1, $pred(r_1)$ and successor, $succ(r_1)$ may still have a reference to n_1.

Consider the scenario where after r_2, p reuses node n_1 while executing passage r_3. Two cases are possible here.

Case 1. r_2 ends with a failure: Upon recovery from a failure all processes abandon their nodes. Since a node only accesses its predecessor during `Enter`, $succ(r_1)$ does not access n_1. Process $pred(r_1)$ on the other hand does perform a `CAS` (line 319) as part of the `Cleanup` module. This `CAS` only succeeds if $n_1.pred$ points to the node of $pred(r_1)$. If p were to reuse n_1 during r_3, it will reset $n_1.pred$ to **null** as part of `Enter` (line 292), thereby rendering the `CAS` of $pred(r_1)$ on line 319 inconsequential. Thus, p can reuse node n_1.

Case 2. r_2 completes its passage successfully: Similar to the previous case, the resetting of $n_1.pred$ to **null** as part of `Enter` (line 292), renders the `CAS` of $pred(r_1)$ on line 319 inconsequential. However, $succ(r_1)$ may still access n_1. Without loss of generality, assume $succ(r_1) \neq p$. Since, r_2 completes successfully, $succ(r_1)$ must have abandoned/relinquished its node. Thus, $succ(r_1)$ does not access n_1 again. Hence, p can reuse node n_1.

In either case, p can safely reuse node n_1. Therefore, two QNodes are sufficient for every process. □

8.3 Transformation for BCSR

The RME algorithm presented in Section 8.2 does not satisfy the BCSR property by itself as shown in Lemma 8.5. This section presents a transformation that, when applied to the

[12]if not already relinquished while executing `Exit` during r_1

algorithm in Section 8.2, helps achieve the BCSR property. The transformation is general enough, that, under the assumption of system-wide failures, adds the BCSR property to any base RME algorithm[13]. Let the base RME algorithm be *baseCL*. Let the RME algorithm as a result of the transformation be referred to as the *target* lock. The *target* lock has the following advantages:

1. It preserves the ME, SF, BR, BW, CI-FCFS and k-FCFS properties of *baseCL*
2. It provides the BCSR property
3. It also satisfies the BE property if *baseCL* satisfies BE and BW properties
4. It preserves the per-passage RMR complexity of *baseCL*

The space complexity overhead of this transformation is $\mathcal{O}(n^2)$.

Golab and Ramaraju (Golab and Ramaraju, 2019) also present a framework to add the BCSR property to any base RME algorithm. Their framework applies to any RME algorithm under the assumption of independent failures; and thus also applies to algorithms under the system-wide failures assumption. However, their framework relies on the fact that the base lock satisfies CS continuity. Roughly speaking, CS continuity is the property that allows a process that crashed inside the CS segment, to recover and resume execution from the CS itself. The algorithm in Section 8.2 does not satisfy the CS continuity property and requires a separate transformation to achieve the BCSR property.

Golab and Hendler (Golab and Hendler, 2018) also present a framework to add the BCSR property to any base RME algorithm. Their algorithm does not require the CS continuity assumption from the base lock. However, their algorithm relies on an epoch number during every passage which in turn depends on an external failure detector.

The framework[14] presented below (in this section) does not require the CS continuity assumption from the base lock and does not depend on an external failure detector.

[13]Assuming that the base RME algorithm satisfies ME and SF correctness properties

[14]The words framework and transformation are used interchangeably in this work

8.3.1 Main Idea

In the absence of failures, this framework leverages the functionality of the base lock to navigate entry and exit of processes to the CS. If a failure does happen, but if no process is in the CS at the time of failure, the base lock is sufficient to manage accesses to the CS. However, to achieve the BCSR property, this framework keeps track of the current process in CS, if any. This allows a process to regain access to the CS, in the event of a failure. Consider the case when a process p_{cs} was in the CS at the time of the failure. Upon recovery from the failure, p_{cs} simply regains its access to the CS, while other processes navigate through the base lock to gain access to the CS. Let p_b be a process that successfully acquires the base lock. Here, p_b waits for p_{cs} to complete its CS. This waiting is executed with the help of a BROADCAST object from Chapter 6. Once p_{cs} completes its CS, it uses the BROADCAST object to signal p_b to enter the CS.

8.3.2 Implementation

The pseudocode for the BCSR transformation under the CC and DSM model is given in Algorithms 8.4 and 8.5. The `BCSR-Wait` module is used to abstract out the waiting required after acquiring the lock *baseCL*, if there is another process in the CS of the *target* lock. The algorithm maintains a shared pair of $\langle$**integer**, **integer**$\rangle$, CSSTATUS and four shared **array** variables, namely *start*, *finish*, *wait* and *skipRE*. Each array variable is of size n with one entry per process. For the DSM model, the i-th entry of each of these array variables is local to process p_i.

The variables *start* and *finish* are arrays of counters of type **integer** and BROADCAST object respectively. Process p_i uses the variable *start*$[i]$ to (loosely) count the number attempts to enter into CS. The algorithm guarantees that the value of the counter *start*$[i]$ would be distinct each time p_i enters into CS unless the entry is due to the BCSR property. The counter *finish*$[i]$ operates in synchronization with the *start*$[i]$ counter to register the end

Algorithm 8.4: Pseudocode for BCSR Transformation of a regular RME lock, *baseCL*, that only satisfies ME and SF properties (Part 1 of 2)

```
322 shared variables
      /* recoverable lock (without
         BCSR) under system-wide
         failures */
323   baseCL: instance of recoverable base
       lock
      /* variable to conduct BCSR */
324   CSStatus: pair of ⟨integer, integer⟩
       variable, initially ⟨⊥, ⊥⟩
      /* remaining variables are
         arrays, whose i-th entry in the
         DSM model is local to process
         p_i */
      /* counter to register beginning
         of distinct attempts */
325   start: array [1...n] of integer variable,
       all elements initially 1
      /* counter to register end of
         attempts */
326   finish: array [1...n] of Broadcast
       object, all elements initially 1
      /* variable that stores the id
         and start counter of process to
         wait on */
327   wait: array [1...n] of pair of
       ⟨integer, integer⟩ variable, all
       elements initially ⟨⊥, ⊥⟩
      /* flag when Recover and Enter of
         baseCL were skipped */
328   skipRE: array [1...n] of boolean
       variables, all elements initially FALSE
329 end
330 Function Recover()
331 begin
332   if CSStatus = ⟨i, _⟩ then
        /* Utilize BCSR */
333     skipRE[i] ← TRUE
334     return
335   end if
      /* Guaranteed to execute Recover
         and Enter of baseCL */
336   skipRE[i] ← FALSE
337   ⟨j, start_j⟩ ← wait[i]
338   if j ≠ ⊥ then
        /* Revoke previous Wait  */
339     finish[j].Withdraw(start_j)
340     wait[i] ← ⟨⊥, ⊥⟩
341   end if
      /* Register end of previous
         attempt if not done so */
342   finish[i].Set(start[i])
      /* Begin passage of baseCL */
343   baseCL.Recover()
344 end
345 Function Enter()
346 begin
347   if CSStatus = ⟨i, _⟩ then  return
348   baseCL.Enter()
      /* Wait if CS is occupied */
349   BCSR-Wait()
      /* Register attempt to CS */
350   start[i] ← start[i] + 1
      /* Enable BCSR */
351   CSStatus ← ⟨i, start[i]⟩
      /* Enter CS */
352 end
```

of its corresponding $start[i]$ attempt. Process p_i uses $finish[i]$ to signal any process that may be waiting after acquiring the base lock.

The variable CSStatus helps ascertain the BCSR property by keeping track of the current process in CS. It stores the id and the *start* counter of the process in CS; and $\langle \perp, \perp \rangle$

Algorithm 8.5: Pseudocode for BCSR Transformation of a regular RME lock, *baseCL*, that only satisfies ME and SF properties (Part 2 of 2)

```
353 Function Exit()
354 begin
        /* Disable BCSR */
355     if CSStatus = ⟨i, _⟩ then
356         CSStatus ← ⟨⊥, ⊥⟩
357     end if
        /* Register end of attempt */
358     finish[i].Set(start[i])
359     if ¬skipRE[i] then
            /* finish passage of baseCL */
360         baseCL.Exit()
361     else
            /* BCSR was utilized */
362         baseCL.Withdraw()
363     end if
364 end

    /* To wait for potential BCSR */
365 Function BCSR-Wait()
366 begin
367     ⟨j, start_j⟩ ← CSStatus
        /* Record upcoming Wait */
368     wait[i] ← ⟨j, start_j⟩
369     if j ≠ ⊥ then
            /* CS is not empty */
370         finish[j].Wait(start_j)
371     end if
372     wait[i] ← ⟨⊥, ⊥⟩
373 end

374 Function Withdraw()
375 begin
376     ⟨j, start_j⟩ ← wait[i]
377     if j ≠ ⊥ then
            /* Revoke previous Wait */
378         finish[j].Withdraw(start_j)
379         wait[i] ← ⟨⊥, ⊥⟩
380     end if
381     if CSStatus = ⟨i, _⟩ then
382         CSStatus ← ⟨⊥, ⊥⟩
383     end if
384     finish[i].Set(start[i])
385     baseCL.Withdraw()
386 end
```

if no process is in CS. The variable $wait[i]$ is a pair of $\langle$**integer**, **integer**$\rangle$, used by process p_i to record any upcoming wait operations it intends to perform. If $wait[i] = \langle j, start_j \rangle$ it implies that p_i intends to wait for the counter $finish[j]$ to reach the value $start_j$. Finally, the variable $skipRE[i]$, of type **boolean** is used by process p_i to indicate that it has invoked the BCSR property and has skipped performing the `Recover` and `Enter` sections of *baseCL* in the current passage.

In order to acquire the *target* lock, process p_i first checks the variable CSStatus to determine if it already holds the CS. If CSStatus contains i (the id of p_i), then p_i can

safely execute the CS; and thus returns from the `Enter` section (line 347). Otherwise, it acquires the lock *baseCL*[15] (line 348). Then, it invokes the `BCSR-Wait` module (line 349), where it

(a) reads and records the value of the variable CSStatus into $wait[i]$ (lines 367 and 368)

(b) checks if some process p_j is already in CS (line 369) based on the value read from the variable CSStatus (line 367)

(c) waits for the process p_j to signal its end of CS (line 370)

(d) clears the variable $wait[i]$ (line 372)

Once it returns from the `BCSR-Wait` module, p_i is presumed to hold access to the CS. It then increments the value of the $start[i]$ counter (line 350) to register its attempt, and stores $\langle i, start[i] \rangle$ (its id and the new value of the $start[i]$ counter) in the variable CSStatus (line 351) to indicate that it holds the CS, in order to enable the BCSR property. Then, p_i executes the CS.

The steps to release the *target* lock are roughly in a reverse order of the steps to acquire the lock. Process p_i first clears the variable CSStatus (line 356). Then, it registers the end of its attempt using the variable $finish[i]$ (line 358), thereby signalling any process waiting on line 370. Finally it releases the base lock, *baseCL* (line 360 or line 362). Note that there are two ways to release the lock *baseCL* as explained later.

The steps of `Recover` are inconsequential if executed at the beginning of a new super-passage. Here, process p_i only starts a new passage (and super-passage) with respect to *baseCL* on line 343. However, when process p_i recovers from a failure, it checks on line 332 whether CSStatus contains i (the id of p_i). If yes, it sets the variable $skipRE[i]$ to TRUE (line 333), to indicate that it is going to skip the `Recover` and `Enter` sections of *baseCL* and simply returns from the `Recover` of the *target* lock.

[15]Ideally, the `Enter` segment for the lock *baseCL* should follow immediately after the corresponding `Recover` segment. However, for ease of exposition, some steps are executed in between the invocations of *baseCL*.`Recover()` and *baseCL*.`Enter()`

Otherwise, if CSStatus does not contain i (the id of p_i), it performs the following steps. Firstly, it resets the variable $skipRE[i]$ to FALSE (line 336). Second, it checks if $wait[i] = \langle j, start_j \rangle$ for $j \neq \perp$ (lines 337 and 338), then p_i had crashed inside the `BCSR-Wait` module. In this case, it withdraws its attempt from the $finish[j]$ to ensure legal usage of the *finish* Broadcast object; and clears $wait[i]$ on line 340. Third, it sets $finish[i]$ to $start[i]$ on line 342 to either register the end of its previous attempt (if it crashed in `Exit` or in `Enter` before line 350 or did not crash at all) or abandon its previous attempt(if it crashed in `Enter` after line 350). Finally, p_i starts a new passage with respect to *baseCL* on line 343.

Note that at the time when process p_i recovers after a failure, if CSStatus contains i, then p_i skips execution of `Recover` and `Enter` of *baseCL* (lines 334 and 347). However, before setting CSStatus to $\langle i, start[i] \rangle$ (line 351), it must have executed `Recover` (line 343) and `Enter` (line 348) of the base lock in an earlier passage. In this case, process p_i has an incomplete super-passage with respect to the *baseCL*. In order to complete this super-passage, p_i withdraws its attempt from the base lock by executing *baseCL*.`Withdraw()` on line 362. Otherwise, if p_i had started a new passage with respect to the lock *baseCL*, it simply completes the passage by executing *baseCL*.`Exit()` on line 360. Process p_i uses the variable $skipRE[i]$ to determine how to complete the super-passage with respect to the lock *baseCL*. Thus, the passage structure for the lock *baseCL* remains consistent with the passage structure of the *target* lock, as per Algorithm 8.1. This ensures that the history remains well-formed with respect to *baseCL*; and if a history is fair with respect to the *target* lock, then it is also fair with respect to the lock *baseCL*.

The steps of the `Withdraw` for the *target* lock are a mix of the steps performed in `Recover` and `Exit` for the *target* lock. Firstly, process p_i withdraws its attempt, if any, to wait on the Broadcast object (lines 376 to 379) similar to the steps in `Recover` (lines 337 to 340). Next, it clears the variable CSStatus (line 382) and sets $finish[i]$ to $start[i]$ on line 384 similar to line 342. Finally, it withdraw its attempt from the base mutex *baseCL* on line 385

to maintain consistent passage structure for the lock *baseCL*. Note that the steps of `Recover` and `Withdraw` are similar enough that if p_i crashes at any point while executing the `Withdraw` for the *target* lock, then upon recovery, it may either re-execute the steps of `Withdraw`, or execute the steps `Recover` (followed by `Enter` and `Exit`) for the *target* lock.

8.3.3 Analysis and Proofs of Correctness

The BCSR transformation employs an RME lock and a BROADCAST object apart from primitive variable types. These objects operate under certain assumptions for their correctness. Therefore, this section is divided into three parts. First, this section proves that the assumptions of *baseCL* lock are fulfilled. Next, the assumptions of the BROADCAST object are fulfilled. Finally, this section proves all correctness, desirable and performance properties for the *target* lock.

Lemma 8.8. *Given a history H, every super-passage of baseCL lock in H is strictly contained within the super-passage of the target lock in H. Moreover, there exists at least one super-passage of the lock baseCL in every super-passage of the target lock.*

Proof. Assume that there are no super-passages in-progress at the beginning of H. Since lock *baseCL* cannot be accessed outside the *target* lock, it is sufficient to show that the end of a super-passage of the *target* lock must be preceded by the end of a super-passage of the lock *baseCL*.

A super-passage of the *target* lock ends only after a successful execution of either the `Exit` or the `Withdraw` segment. A successful execution of the `Withdraw` segment of the *target* lock implies a successful execution of the `Withdraw` segment of *baseCL* at line 385. Similarly, a successful execution of the `Exit` segment of the *target* lock implies a successful execution of the `Exit` segment of the *baseCL* at line 360 or the `Withdraw` segment of *baseCL* at line 362. In either case, the super-passage ends with respect to the lock *baseCL*. Therefore, the result holds. □

Corollary 8.9. *Given a history H, if H is well-formed with respect to the target lock, then H is also well-formed with respect to the base lock.*

Corollary 8.10. *At the end of a failure-free passage of the target lock, a process does not have any super-passage in-progress with respect to baseCL.*

Corollary 8.11. *Given a history H, if H is fair with respect to the target lock, then it is also fair with respect to baseCL.*

The following invariant can be observed upon code inspection.

Proposition 8.12. *At any instant of a given history H, $finish[i] \leq start[i] \leq finish[i] + 1$.*

The *finish* array is the only variable of type BROADCAST object. It is shown below in Lemmas 8.13 and 8.14 that histories of the *target* lock are well-formed and legal with respect to each BROADCAST object.

Lemma 8.13. *Given a history H of the target lock, and any process p_i, then H is well-formed with respect to the $finish[i]$* BROADCAST *object.*

Proof. H is well-formed because:

(W1) From code inspection, it can be observed that p_i advances the $start[i]$ counter in an incremental manner (line 350). From Proposition 8.12, it can be inferred that the `SET` operation on the $finish[i]$ counter is invoked in an incremental manner.

(W2) A `WAIT` operation is invoked on the object $finish[i]$ only as part of the `BCSR-Wait` module (line 370). The argument used for this `WAIT` operation is fetched from the variable CSSTATUS (line 367). This value is in turn assigned using the $start[j]$ counter (line 351). Thus, from Proposition 8.12, it can be inferred that if the last successful `SET` operation was invoked with argument x, then the argument to the `WAIT` operation is at most $x + 1$.

□

Lemma 8.14. *Let H be an infinite fair history of the target lock, where each process fails a finite number of times during its super-passage. Then, H is legal with respect to the finish[i]* BROADCAST *object.*

Proof. By code inspection, observe that if process p_i crashes while executing `SET(`x`)` for some x, it will eventually re-execute `SET(`x`)` as part of `Recover` (line 342), `Exit` (line 358) or `Withdraw` (line 384). Moreover, if a process p_j crashes while executing *finish*[i].`WAIT(`x`)` (line 370), for some x, then, upon recovery, it invokes *finish*[i].`WITHDRAW(`x`)`.

H is legal because:

(L1) H is fair. Thus, every process will continue to make progress.

(L2) Every run ends at the end of a super-passage of the *target* lock. Since, the number of passages in a super-passage are finite, every run in H contains a finite number of instances.

(L3) Similarly, the last instance of every maximal run in H is failure-free.

□

Since every history of the *target* lock is well-formed and legal with respect to each BROADCAST object; all operations of any BROADCAST object are assumed to eventually complete-successfully.

For the rest of the section, a process is said to be *enabled* to enter the CS, if it reaches a point in its execution after which it is guaranteed to enter the CS in a bounded number of its own steps[16]. Note that from code inspection, the following proposition can be established.

Proposition 8.15. *A process can modify the variable* CSSTATUS *during* `Enter` *only if it is enabled to enter the CS.*

[16] if there is no system-wide failure

Lemma 8.16. *If* CSStatus $= \langle \perp, \perp \rangle$ *at any instant during an epoch e, then at most one process can modify its contents till it is reset back to* $\langle \perp, \perp \rangle$.

Proof. In order to modify the contents of the variable CSStatus during `Enter` (line 351), processes need to first acquire the lock *baseCL* (line 348). At most one process can acquire this *baseCL* (ME property). Additionally, *baseCL* is released (line 360) after the variable CSStatus is reset (line 356). Thus, at most one process can modify the contents of the variable CSStatus until it is reset. □

Lemma 8.17. *If* CSStatus $= \langle j, start_j \rangle$ *at the beginning of an epoch e, then no other process* p_i, $i \neq j$, *can modify its contents until* p_j *resets it (as part of the* `Exit` *segment at line 356 or the* `Withdraw` *segment at line 382).*

Proof. In order to modify the contents of the variable CSStatus during `Enter` (line 351), process p_i first needs to execute the `BCSR-Wait` module (line 349). As part of the `BCSR-Wait` module, p_i waits till $finish[j]$ catches upto $start_j$ (line 370). The variable $finish[i]$ is updated (line 358 or line 384) only after the variable CSStatus is reset to $\langle \perp, \perp \rangle$ (line 356 or line 382) during `Exit` or `Withdraw`. Thus, the statement holds. □

Theorem 8.18. *The BCSR transformation preserves the ME property.*

Proof. Consider an epoch e. At the beginning of e, the variable CSStatus may contain $\langle \perp, \perp \rangle$ or $\langle i, start_i \rangle$ for some process p_i. In either case, from Lemma 8.16 or Lemma 8.17, at most one process may modify the contents of the variable CSStatus during e. Therefore, from Proposition 8.15, it implies that at most one process may be enabled to enter the CS during e, which in turn implies the ME property. □

Theorem 8.19. *The BCSR transformation preserves the SF property.*

Proof. Proof by contradiction. Assume there exists an infinite fair history H in which every process fails a finite number of times during each of its super-passage. Assume that some

process p does not complete the last passage in its super-passage (p starves forever). Since p does not fail during its last passage, there can be no failures hereafter as per the system-wide failure model.

Moreover, from Lemma 8.8 and Corollary 8.11, H is an infinite history with respect to the lock *baseCL* such that every process fails a finite number of times during each of its super-passage with respect to *baseCL*. Hence, p may not be starved at while executing any operation of *baseCL*.

The last passage of p may either be a Withdraw-passage or a CS-passage.

Case 1. p is starved during a Withdraw-passage:

The `Withdraw` segment does not contain any loops, but invokes *baseCL*.`Withdraw`. Even if the lock *baseCL* may not satisfy the BW property, it satisfies the SF property. Therefore, p cannot be starved during a Withdraw-passage.

Case 2. p is starved during a CS-passage:

During a CS-passage, p may potentially need to wait when it invokes *baseCL*.`Enter()` (line 348) or during `BCSR-Wait` (line 370). Since the lock *baseCL* satisfies the SF property, it cannot be starved during *baseCL*.`Enter()`. If p is starved during `BCSR-Wait`, it must have read the value $\langle j, start_j \rangle$ of variable CSStatus at line 367 to infer that some process p_j is in its CS. Since H is fair, p_j will eventually complete its CS and update the variable during `Recover` (line 342), `Exit` (line 358) or `Withdraw` (line 384). Thus, p cannot be starved during a CS-passage.

Hence p cannot be starved in either case. This is a contradiction. Therefore, the transformation preserves the SF property. □

Theorem 8.20. *The target lock that ensues from the BCSR transformation satisfies the BCSR property.*

Proof. Let process p be in the CS during some epoch e when a system-wide failure occurs. Before p enters the CS, it must have set CSStatus to contain its own id. At the beginning of epoch $e+1$, the variable CSStatus contains p's id. Lemma 8.17 states that no process can modify the contents of this variable until p resets it. When p recovers from its failure, it observes its id stored in the variable CSStatus (lines 332 and 347). Thus, p sets the value of *skipRE* to TRUE and enters the CS in a bounded number of its own steps. Hence, the BCSR property is guaranteed. □

Theorem 8.21. *The BCSR transformation preserves the BR and BW properties.*

Proof. As the code inspection shows, executions of `Recover` and `Withdraw` do not involve any loops, but invoke `Recover` and `Withdraw` respectively of *baseCL*. Therefore, the BCSR transformation preserves the BR and BW properties. □

Theorem 8.22. *The BCSR transformation satisfies the BE property if baseCL satisfies the BE and BW properties.*

Proof. As the code inspection shows, executions of `Exit` do not involve any loops, but invoke `Exit` or `Withdraw` of *baseCL*, depending on whether BCSR property was utilized. Therefore, the BCSR transformation satisfies the BE property if the lock *baseCL* satisfies the BE and BW properties. □

Theorem 8.23. *The BCSR transformation preserves the worst-case RMR complexity of the baseCL.*

Proof. As the code inspection shows, executions of `Recover`, `Exit` and `Withdraw` invoke operations of the *baseCL* and do not contain any extra loops. The execution of `Enter`, however contains one loop, in addition to `Enter` for *baseCL*. The loop in `Enter` is executed as part of the `BCSR-Wait` module at line 297, where a process waits for the process currently in CS. This loop is executed with the help of a Broadcast object, that, as per Theorem 6.8,

incurs $\mathcal{O}(1)$ RMRs. Moreover, any passage of the *target* lock contains at most one passage of *baseCL*. Hence, the BCSR transformation preserves the worst-case RMR complexity of the lock *baseCL*. □

Theorem 8.24. *The BCSR transformation preserves the FRF property.*

Proof. Assume that the lock *baseCL* guarantees the FRF property for every process that executes b consecutive failure-free steps. Consider a history H that contains infinitely many super-passages of the *target* lock. From Lemma 8.8, H contains infinitely many super-passages of *baseCL*. Consider a new bound $b + b'$ (for the *target* lock) such that b' represents the number of steps required to reach line 348. Assume that process p invokes $b + b'$ consecutive failure-free steps during some epoch e in H and does not invoke the `Withdraw` segment during that super-passage. Thus, p must have invoked b consecutive failure-free steps of *baseCL* in e. Moreover, p does not execute the `Withdraw` segment of the lock *baseCL* during `Recover` or `Enter`.

Since the lock *baseCL* satisfies the FRF property, p is eventually guaranteed to acquire it. Once p acquires *baseCL*, it is guaranteed to enter the CS. Hence, the BCSR transformation preserves the FRF property.

□

Note that the *target* lock satisfies the FRF property only when the lock *baseCL* satisfies the FRF property.

Corollary 8.25. *The BCSR transformation applied to the* MCS-SW *lock does not guarantee the FRF property.*

8.4 Transformation for FRF

The MCS-CSR lock, a result of the transformation presented in Section 8.3 applied to the MCS-SW lock (Section 8.2), does not satisfy the FRF property, as shown in Corollary 8.25

This section presents a transformation that, when applied to the MCS-CSR lock, achieves the FRF property. The transformation is general enough that, under the assumption of system-wide failures, adds the FRF property to any base RME algorithm[17]. Once again, let the base RME algorithm be *baseFL* and the resultant RME algorithm be the *target* lock. The *target* lock has the following advantages[18]:

1. It preserves the ME, SF, BCSR, BW and BE properties of *baseFL*
2. It provides the BR property even if *baseFL* does not satisfy the BR property
3. It provides the FRF property
4. It preserves the per-passage RMR complexity of *baseFL*

Golab and Hendler (Golab and Hendler, 2018) present a framework to add the FRF property to any base RME algorithm. Just like their BCSR transformation, their FRF transformation also relies on an epoch number during every passage which in turn depends on an external failure detector. The framework presented below (in this section) does not depend on an external failure detector.

8.4.1 Main Idea

The FRF property ensures that a "victim" process p_j is not denied entry into the CS while another process p_i is able to complete its super-passage infinitely often, given that p_j has taken a sufficient number of steps to register its CS request. Therefore, p_i may eventually have to wait for p_j during some super-passage. This mechanism is operated in two phases: (a) the snapshot phase, and (b) the wait phase. During the snapshot phase, p_i begins tracking the course of p_j's super-passage. In the wait phase, p_i blocks until p_j has completed the super-passage that p_i began tracking. Process p_i does this for every process p_j ($j \in [1 \ldots n]$, $j \neq i$).

[17]Assuming that the base RME algorithm satisfies ME and SF correctness properties

[18]Note that the transformation may lose other fairness properties like k-FCFS or CI-FCFS. These properties can be preserved by extending the FRF transformation as shown in Chapter 7

These phases are executed in different super-passages to avoid a circular wait. Moreover, in order to avoid RMR-explosion, each phase is further broken down into n strides, such that p_i executes one stride per super-passages. This mechanism is then repeated at the end of $2n$ strides.

8.4.2 Implementation

The pseudocode for the FRF transformation under the CC and DSM model is given in Algorithms 8.6 and 8.7. The `FRF-Wait` module abstracts out the waiting required to ensure the FRF property. The algorithm maintains five shared **array** variables, namely *start*, *finish*, *stride*, *latest* and *snapshot*. The array variables *start*, *finish*, *stride* and *latest* are of size n, with one entry per process. The array variable *snapshot* is also of size n with n entries per process. For the DSM model, the i-th entry of each of these array variables is local to process p_i.

Similar to the transformation for BCSR (Section 8.3), the variables *start* and *finish* are arrays of counters of type **integer** and BROADCAST object (from Chapter 6) respectively. The algorithm uses these two monotonically non-decreasing counters for every process to *demarcate* the beginning and end of its super-passages[19]. Process p_i uses the variable $start[i]$ to count the number super-passages it has started. The algorithm guarantees that the value of the counter $start[i]$ would be distinct for each super-passage of p_i. The counter $finish[i]$ operates in synchronization with the $start[i]$ counter to register the end of its corresponding $start[i]$ super-passage. At any time, $finish[i] \leq start[i] \leq finish[i]+1$. If $start[i]-finish[i] = 1$, then p_i has a super-passage in progress. Additionally, p_i uses the BROADCAST object $finish[i]$ to signal any process that may be waiting to maintain the FRF property.

[19]The *start* and *finish* counters can only approximate the boundary of a super-passage. In theory, a process may execute the last instruction of `Exit` and fail before returning, thereby rendering the super-passage incomplete. It would be impossible for an RME algorithm to detect that the next invocation of `Recover` is part of the same super-passage. However, for ease of exposition, this algorithm assumes that the *start* and *finish* counters demarcate the beginning and end of the super-passage.

Algorithm 8.6: Pseudocode for FRF Transformation of a regular RME lock, *baseFL* that satisfies ME and SF properties (and optionally BCSR property) (Part 1 of 2)

```
387 shared variables
        /* recoverable lock (without FRF)
           under system-wide failures */
388     baseFL: instance of recoverable base
         lock
        /* remaining variables are
           arrays, whose i-th entry in the
           DSM model is local to process
           p_i */
        /* counter to register beginning
           of super-passage */
389     start: array [1...n] of integer variable,
         all elements initially 1
        /* counter to register end of
           super-passage */
390     finish: array [1...n] of BROADCAST
         object, all elements initially 1
        /* variable to execute FRF-Wait()
           in a step-manner */
391     stride: array [1...n] of integer
         variable, all elements initially 1
        /* variable to ensure single
           execution of FRF-Wait() per
           super-passage */
392     latest: array [1...n] of integer
         variable, all elements initially 1
        /* stores a snapshot of how many
           super-passage other processes
           have started */
393     snapshot: array [1...n][1...n] of
         integer variable, all elements initially
         1
394 end
```

```
395 Function Recover()
396 begin
397     if start[i] = finish[i].READ() then
            /* #super-passages started =
               #super-passages finished */
            /* maintain last stride of
               FRF-Wait() executed */
398         latest[i] ← stride[i]
            /* Register beginning of a new
               super-passage */
399         start[i] ← start[i] + 1
400     end if
401 end
402 Function Enter()
403 begin
        /* Next if-block ensures a single
           execution of FRF-Wait() module
           per super-passage */
404     if latest[i] = stride[i] then
            /* Blocking if some process
               has registered but not
               fulfilled its attempt */
405         FRF-Wait()
406     end if
        /* Begin passage of baseFL */
407     baseFL.Recover()
408     baseFL.Enter()
409 end
```

Algorithm 8.7: Pseudocode for FRF Transformation of a regular RME lock, *baseFL* that satisfies ME and SF properties (and optionally BCSR property) (Part 2 of 2)

```
410 Function Exit()
411 begin
        /* Complete passage and
           super-passage of baseFL */
412     baseFL.Exit()
        /* Register end of super-passage;
           release waiting processes */
413     finish[i].SET(start[i])
414 end
415 Function Withdraw()
416 begin
        /* Begin Withdraw of baseFL */
417     baseFL.Withdraw()
        /* Withdraw any pending wait from
           FRF-Wait() */
418     j ← (stride[i] − 1) (mod n) + 1
419     if j ≠ i then
420         finish[j].WITHDRAW(snapshot[i][j])
421     end if
        /* Register end of super-passage;
           release waiting processes */
422     finish[i].SET(start[i])
423 end
    /* Blocking procedure to enforce FRF
       property */
424 Function FRF-Wait()
425 begin
        /* Index of process under
           consideration for FRF */
426     j ← (stride[i] − 1) (mod n) + 1
427     if stride[i] ∈ [1, n] then
            /* snapshot phase: take a
               snapshot of start counters
               of all processes */
428         snapshot[i][j] ← start[j]
429     else if (stride[i] ∈ [n + 1, 2n]) ∧ (j ≠ i)
          then
            /* wait phase: wait for other
               processes to complete their
               attempt */
430         finish[j].WAIT(snapshot[i][j])
431     end if
432     if stride[i] < 2n + 1 then
            /* Advance stride */
433         stride[i] ← stride[i] + 1
434     else
            /* Reset stride */
435         stride[i] ← 1
436     end if
437 end
```

The variables *stride* and *latest* are arrays of circular counters, each of type **integer**. These counters can only assume values in the range $[1,\ 2n]$. The variable $stride[i]$ is used by p_i to determine which one of the $2n$ strides of `FRF-Wait` to execute. The value of $stride[i]$ in the range $[1,\ n]$ corresponds to the snapshot phase, while the range $[n+1,\ 2n]$ corresponds to the wait phase. The algorithm guarantees that the value of $stride[i]$ is updated at the end of a failure-free execution of `FRF-Wait`. The variable $latest[i]$ captures the latest stride taken prior to executing an instance of `FRF-Wait`. It is used to ensure p_i executes exactly one stride per super-passage. Finally, the variable *snapshot* is a two dimensional array of type **integer**. During the snapshot phase, process p_i stores the observed value of $start[j]$ to $snapshot[i][j]$ and later uses this value to wait for $finish[j]$ to catch up.

In order to acquire the *target* lock, process p_i first checks if the current passage marks the beginning of a new super-passage (line 397). Note that at the beginning of a super-passage for process p_i, it is crucial that the values of $stride[i]$ and $latest[i]$ match. If the passage is the beginning of a new super-passage, p_i first updates the value of $latest[i]$ to match the value of $stride[i]$ (line 398) and then registers the beginning of a new super-passage by incrementing the counter $start[i]$ (line 399).

Next, it checks if the current passage if `FRF-Wait` has advanced the value of $stride[i]$ (line 404). If yes, it does not invoke the `FRF-Wait` module again and begins a new passage with respect to *baseFL* (lines 407 to 408). If not, p_i invokes an instance of `FRF-Wait` (line 405) before beginning a new passage with respect to *baseFL*. As part of the `FRF-Wait` module, p_i performs the following steps:

(a) Determines the id (j) of the process to snapshot or wait for (line 426)

(b) Determines whether to perform the snapshot phase or the wait phase (lines 427 and 429)

  (i) Snapshot phase: Records the value of the counter $start[j]$ to $snapshot[i][j]$ (line 428)

  (ii) Wait phase: Waits on the BROADCAST object $finish[j]$ to reach the value $snapshot[i][j]$ (line 430)

(c) Updates the value of $stride[i]$ (lines 433 and 435)

Once p_i acquires *baseFL*, it is deemed to be in the CS of the *target* lock. Note that once p_i begins a passage with respect to *baseFL*, it does not get blocked in `FRF-Wait` in that super-passage of the *target* lock.

In order to release the *target* lock, process p_i first releases *baseFL* (line 412). Then it updates the BROADCAST object $finish[i]$ (line 413) to mark the end of its super-passage and to release any blocking processes.

Note that the transformation is designed to have a nested structure, where the lock *baseFL* is only accessed after the completion of the `FRF-Wait` module. In the event of a failure, the recovery of process p_i depends on three possible cases:

(a) Process p_i crashed before completion of the `FRF-Wait` module[20] (lines 397 to 406).

In this case, there is no incomplete super-passage with respect to *baseFL*. Upon recovery, p_i executes all the pending steps of FRF before beginning a new passage as well as a new super-passage with respect to the lock *baseFL*.

(b) Process p_i crashed after completion of the `FRF-Wait` module but before registering the end of its super-passage (lines 407 to 412)

In this case, it is safe to assume that p_i crashed while executing a super-passage of *baseFL*. Upon recovery, p_i skips the steps required to perform FRF, with the help of if-blocks(lines 397 to 400 and lines 404 to 406) and executes a new passage with respect to *baseFL*.

(c) Process p_i crashed after registering the end of its super-passage (line 413)

In this case, the super-passage is presumed to have completed and subsequent passages are presumed to be part of a separate passage.

[20]The completion of the `FRF-Wait` module is detected by whether $stride[i]$ has been updated in the current super-passage.

However, if p_i wishes to `Withdraw` after a failure, it first invokes `Withdraw` for *baseFL* (line 417). Then, it withdraws any incomplete wait on the BROADCAST object *finish* (lines 418 to 421) that it may have invoked as part of the `FRF-Wait` module. Finally, it sets the counter *finish*[i] (line 422) to register the end of the super-passage. Note that the nested design of the algorithm allows p_i to begin a CS-passage, even if it crashes during Withdraw-passage.

8.4.3 Analysis and Proofs of Correctness

Note that the algorithm for the FRF transformation has a high overlap with the algorithm for memory reclamation from Chapter 7. Thus, the analysis and proofs of correctness of the two algorithms are similar. For completeness, the rest of this section states the major results and the reader is encouraged to refer to Section 7.5 for detailed proofs.

Lemma 8.26. *Let H denote a history of the target lock. Then, for every* BROADCAST *object b, $H \mid b$ is well-formed with respect to b.*

Lemma 8.27. *Assume that every process fails only a finite number of times in each of its super-passage. Let H denote an infinite fair history of the target lock. Then, for every* BROADCAST *object b, $H \mid b$ is legal with respect to b.*

Theorem 8.28. *The FRF transformation preserves the ME property.*

Theorem 8.29. *The FRF transformation preserves the SF property.*

Theorem 8.30. *The FRF transformation preserves the BCSR property.*

Theorem 8.31. *The FRF transformation preserves the BR, BE and BW properties.*

Theorem 8.32. *The FRF transformation preserves the worst-case RMR complexity of baseFL.*

Theorem 8.33. *The target lock as a result of the FRF transformation satisfies the FRF property.*

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 RME for Independent Failures

This work describes a general framework to transform any non-adaptive RME algorithm into a super-adaptive one without increasing its worst-case RMR complexity. In addition to the hardware instructions used by the underlying non-adaptive RME algorithm, the framework uses `CAS` and `FAS` RMW instructions, both of which are commonly available on most modern processors. When applied to the non-adaptive RME algorithm proposed by Jayanti, Jayanti and Joshi in (Jayanti et al., 2019) or Katzan and Morrison in (Katzan and Morrison, 2021), it yields a well-bounded super-adaptive RME algorithm that is simultaneously adaptive to (a) the number of processes competing for the lock, *as well as* (b) the number of failures that have occurred in the recent past, while having the same (asymptotic) worst-case RMR complexity as that of the base RME algorithm. Additionally, it is also shown that the RME algorithm obtained by applying this framework is fair and satisfies a variant of the FCFS property (CI-FCFS), even if the base RME algorithm is unfair.

This framework can be further improved upon as follows. A failed process, upon restarting, attempts to reacquire all the locks at every level it had advanced to, beginning from level one. As a result, the worst-case RMR complexity of a super-passage is given by $\mathcal{O}(F_0 \cdot \min\{\ddot{c}, \sqrt{F+1}, {}^{\log n}/_{\log\log n}\})$, where F_0 denotes the number of times the process fails while executing its (own) super-passage. Instead, the framework can be modified to allow a process to keep track of its level. With this modification, the worst case RMR complexity of a super passage reduces to $\mathcal{O}(F_0 + \min\{\ddot{c}, \sqrt{F+1}, {}^{\log n}/_{\log\log n}\})$.

Note that the framework needs to allocate memory dynamically on the heap which leads to an unbounded space consumption. To make the framework practical, this work also describes an extension to the framework to reclaim the memory of shared objects when no

longer needed. The memory reclamation algorithm bounds the worst-case space complexity of the framework, while maintaining all its desirable properties at the same time. The approach for the memory reclamation technique is general enough that it can be applied to other RME algorithms as well to bound their space complexity, such as Jayanti, Jayanti and Joshi's algorithm in (Jayanti et al., 2019).

9.2 RME for System-Wide Failures

For the system-wide failure model, this work presents an optimal RME algorithm that incurs $\mathcal{O}(1)$ RMRs for both the CC and DSM memory models. This algorithm does not depend on an external failure detector unlike the one by Golab and Hendler (Golab and Hendler, 2018). Additionally, this work presents two optimal transformations to add the BCSR and FRF properties to existing RME algorithm under the system-wide failure model.

The FRF transformation works in strides and phases, where a process takes snapshots in the first n strides (snapshot phase) and then waits for these snapshots in the next n strides (wait phase). Considering RMR complexity as a priority, these $2n$ strides are distributed over $2n$ attempts. This guarantees that a process may only be *overtaken* $\mathcal{O}(n)$ times. If the algorithm were to be modified such that there are n strides per attempt, then the number of times a process can be overtaken reduces to $\mathcal{O}(1)$ while the RMR overhead increases to $\mathcal{O}(n)$. Thus, there exists a potential trade-off between the "degree" of fairness (FRF) and the RMR complexity. The number of strides per attempt can be tailored to achieve this trade-off.

9.3 Future Work

This work can be further extended as follows:

1. **Implementation**: This work lays out various techniques of building RME algorithms and proves their correctness. A real-world implementation of under different scenarios

would aid in providing further insights into the performance comparison and usage expectation of the RME algorithms.

2. **Withdraw option**: The system model was modified to incorporate an option for a process to `Withdraw` a request after a system-wide failure. This option can further be extended for RME under the independent failure model which frames the problem of the lower bound for RME with the `Withdraw` option as an open problem.

3. **Weakly RME**: The filter lock of the framework can be further modified to be more responsive (logarithmic in the number of failures) and space-efficient (reuse nodes across filter locks on different levels). One potential way of achieving this is splitting the set of slow processes uniformly into two sets, thereby creating a hyper-adaptive framework.

4. **RME variants**: Extensive research has been performed into generalized variants of the ME problem (Mellor-Crummey and Scott, 1991b; Jayanti et al., 2005; Brandenburg and Anderson, 2010, 2011; Gokhale et al., 2021; Hadzilacos, 2001; Joung, 2000; Jayanti et al., 2003). The RME problem can also be modified to these generalized variants. Efficient recoverable algorithms can be developed for important variants of the ME problem including recoverable reader-writer mutual exclusion (RRWME) and recoverable group mutual exclusion (RGME).

5. **Memory reclamation**: The memory reclamation algorithm consumes $\mathcal{O}(n^2)$ space. This space consumption is higher than the space consumption of the MCS lock ($\mathcal{O}(n)$). An open problem in this area is establishment of a trade-off between RMR and space complexity for RME algorithms (Bansal et al., 2012). Another open problem in the space of memory reclamation is the development of *non-blocking* fault-tolerant memory reclamation for other recoverable data structures.

REFERENCES

AMD (2019, September). *AMD64 Architecture Programmer's Manual Volume 3: General Purpose and System Instructions*. AMD.

Anderson, J. H. and Y.-J. Kim (2002, December). An Improved Lower Bound for the Time Complexity of Mutual Exclusion. *Distributed Computing (DC) 15*(4), 221–253.

Arcangeli, A., M. Cao, P. E. McKenney, and D. Sarma (2003). Using Read-Copy-Update Techniques for System V IPC in the Linux 2.5 Kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pp. 297–309.

Attiya, H., D. Hendler, and P. Woelfel (2008, May). Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, New York, NY, USA, pp. 217–226. ACM.

Bansal, N., V. Bhatt, P. Jayanti, and R. Kondapally (2012). Tight time-space tradeoff for mutual exclusion. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, New York, NY, USA, pp. 971–982. Association for Computing Machinery.

Bohannon, P., D. Lieuwen, and A. Silberschatz (1996). Recovering Scalable Spin Locks. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Washington, DC, USA, pp. 314–322. IEEE Computer Society.

Bohannon, P., D. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava (1995). Recoverable User-level Mutual Exclusion. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Washington, DC, USA, pp. 293–301. IEEE Computer Society.

Brandenburg, B. B. and J. H. Anderson (2010, sep). Spin-based reader-writer synchronization for multiprocessor real-time systems. *Real-Time Syst. 46*(1), 25–87.

Brandenburg, B. B. and J. H. Anderson (2011). Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, EMSOFT '11, New York, NY, USA, pp. 69–78. Association for Computing Machinery.

Brown, T. A. (2015, July). Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, Donostia-San Sebastián, Spain, pp. 261–270.

Chan, D. and P. Woelfel (2021). A Tight Lower Bound for the RMR Complexity of Recoverable Mutual Exclusion. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*. Association for Computing Machinery (ACM).

Chan, D. Y. C. and P. Woelfel (2020, August). Recoverable Mutual Exclusion with Constant Amortized RMR Complexity from Standard Primitives. In *Proceedings of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA. Association for Computing Machinery (ACM).

Dhoked, S. and N. Mittal (2020). An Adaptive Approach to Recoverable Mutual Exclusion. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA. ACM.

Dhoked, S. and N. Mittal (2021). Adaptive and fair transformation for recoverable mutual exclusion. *CoRR abs/2110.08308*.

Dijkstra, E. W. (1965). Solution of a Problem in Concurrent Programming Control. *Communications of the ACM 8*(9), 569.

Dvir, R. and G. Taubenfeld (2017, October). Mutual Exclusion Algorithms with Constant RMR Complexity and Wait-Free Exit Code. In J. Aspnes, A. Bessani, P. Felber, and J. Leitão (Eds.), *Proceedings of the 21st International Conference on Principles of Distributed Systems (OPODIS)*, Volume 95, Dagstuhl, Germany, pp. 17:1–17:16. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Fraser, K. (2004, February). *Practical Lock-Freedom*. Ph. D. thesis, University of Cambridge.

Gokhale, S., S. Dhoked, and N. Mittal (2021). On group mutual exclusion for dynamic systems. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, New York, NY, USA, pp. 446–447. Association for Computing Machinery.

Golab, W. and D. Hendler (2017). Recoverable Mutual Exclusion in Sub-Logarithmic Time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, pp. 211–220. Association for Computing Machinery (ACM).

Golab, W. and D. Hendler (2018). Recoverable Mutual Exclusion Under System-Wide Failures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, pp. 17–26. Association for Computing Machinery (ACM).

Golab, W., D. Hendler, and P. Woelfel (2006). An $\mathcal{O}(1)$ RMRs Leader Election Algorithm. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, pp. 238—-247. Association for Computing Machinery (ACM).

Golab, W. and A. Ramaraju (2016). Recoverable Mutual Exclusion: [Extended Abstract]. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, pp. 65–74. ACM.

Golab, W. and A. Ramaraju (2019, November). Recoverable Mutual Exclusion. *DC 32*(6), 535–564.

Hadzilacos, V. (2001). A note on group mutual exclusion. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, New York, NY, USA, pp. 100–106. Association for Computing Machinery.

Hart, T. E., P. E. McKenney, A. D. Brown, and J. Walpole (2007). Performance of Memory Reclamation for Lockless Synchronization. *Journal of Parallel and Distributed Computing (JPDC) 67*(12), 1270–1285.

Intel (2016, September). *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M*. Intel.

Jayanti, P., S. Jayanti, and A. Joshi (2019). A Recoverable Mutex Algorithm with Sublogarithmic RMR on Both CC and DSM. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, New York, NY, USA, pp. 177–186. Association for Computing Machinery (ACM).

Jayanti, P. and A. Joshi (2017, October). Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In A. W. Richa (Ed.), *Proceedings of the 31st Symposium on Distributed Computing (DISC)*, Volume 91, Dagstuhl, Germany, pp. 30:1–30:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Jayanti, P. and A. Joshi (2019). Recoverable mutual exclusion with abortability. In *Proceedings of the International Conference on Networked Systems (NETSYS)*, pp. 217–232. Springer International Publishing.

Jayanti, P., S. Petrovic, and N. Narula (2005). Read/write based fast-path transformation for fcfs mutual exclusion. In P. Vojtáš, M. Bieliková, B. Charron-Bost, and O. Sýkora (Eds.), *SOFSEM 2005: Theory and Practice of Computer Science*, Berlin, Heidelberg, pp. 209–218. Springer Berlin Heidelberg.

Jayanti, P., S. Petrovic, and K. Tan (2003). Fair group mutual exclusion. In *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*, PODC '03, New York, NY, USA, pp. 275–284. Association for Computing Machinery.

Joung, Y.-J. (2000). Asynchronous group mutual exclusion. *Distributed computing 13*(4), 189–206.

Katzan, D. and A. Morrison (2021). Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. In Q. Bramas, R. Oshman, and P. Romano (Eds.), *Proceedings of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, Volume 184 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, pp. 15:1–15:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

McKenney, P. E. and J. D. Slingwine (1998). Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 509–518.

Mellor-Crummey, J. M. and M. L. Scott (1991a, February). Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems 9*(1), 21–65.

Mellor-Crummey, J. M. and M. L. Scott (1991b). Scalable reader-writer synchronization for shared-memory multiprocessors. *ACM SIGPLAN Notices 26*(7), 106–113.

Michael, M. M. (2004). Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 15*(6), 491–504.

Narayanan, D. and O. Hodson (2012, March). Whole-System Persistence. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, pp. 401–410. ACM.

Oukid, I. and L. Lersch (2018). On the diversity of memory and storage technologies. In *Datenbank-Spektrum: Vol. 18, No. 2*, Berlin Heidelberg, pp. 121–127. Springer Nature.

Ramaraju, A. (2015). RGLock: Recoverable Mutual Exclusion for Non-Volatile Main Memory Systems. Master's thesis, Electrical and Computer Engineering Department, University of Waterloo.

Wen, H., J. Izraelevitz, W. Cai, A. H. Beadle, and M. L. Scott (2018). Interval-Based Memory Reclamation. In *Proceedings of the 23rd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, PPoPP '18, New York, NY, USA, pp. 1—-13. Association for Computing Machinery (ACM).

Yang, J.-H. and J. H. Anderson (1995, March). A Fast, Scalable Mutual Mxclusion Algorithm. *Distributed Computing (DC) 9*(1), 51–60.

BIOGRAPHICAL SKETCH

Sahil Dhoked began his journey with a BTech degree in Computer Science and Engineering from **IIT Indore**, India (2009-2013). At **IIT Indore**, he developed many skills required in the software industry. Next, to put these skills to use and gain some experience in the industry, he joined **Microsoft**, India as a software engineer (2013-2016). This industry experience later helped him when he started his master's degree in Computer Science from **UT Dallas** in 2016.

As a master's student (MS in CS), he studied various courses, including but not limited to *Algorithms and Data structures*, *Artificial Intelligence*, *Game Theory*, *etc.* One such course was the course on *Multicore Programming*, which laid the foundation for his PhD Thesis. In 2017, he joined the PhD program at **UT Dallas** to work under the guidance of Dr. Neeraj Mittal.

As a PhD candidate, Sahil developed a passion for multi-core systems. His primary focus was to gain an expertise in distributed systems and concurrency. Specifically, his area of research was based on "Mutual Exclusion". During his stint as a PhD candidate, he published multiple papers. One noteworthy publication is the one in PODC 2020, for which he also received the best student paper award.

After completing his PhD, Sahil wishes to utilize his research experience in the industry. Starting June 2022, he is joining **Meta** as a research scientist where he plans to fulfil his research goals

CURRICULUM VITAE

Sahil Dhoked

Contact Information

LinkedIn : https://www.linkedin.com/in/sahildhoked/
Github : https://github.com/dhokedsahil

Education

UT DALLAS | **PHD** | MULTICORE & DISTRIBUTED COMPUTING 2017-2022
UT DALLAS | MS IN CS | INTELLIGENT SYSTEMS 2016-2022
IIT INDORE | B. TECH | COMPUTER SCIENCE AND ENGINEERING 2009-2013

Experience

SOFTWARE ENGINEERING INTERN | **Microsoft** MAY 2012 – JULY 2012
SOFTWARE ENGINEER | **Microsoft** JULY 2013 – JAN 2016
LECTURER | UT DALLAS FALL 2019
RESEARCH ASSISTANT | UT DALLAS FALL 2017 - FALL 2021
TEACHING ASSISTANT | UT DALLAS FALL 2016 - FALL 2021

Rewards and Certifications

Best student paper award at PODC 2020

"Delight the Customer" Award for Optimizing SQL Server query performance at Microsoft

Demonstrated phone app in MGX (Microsoft Global Exchange) in 2015

Certification for Exam 70-461: Querying Microsoft SQL Server 2012/2014

Cleared qualifier exams for Algorithms & Data structures, Artificial Intelligence, and Database design

Publications

Dhoked, S. and N. Mittal (2020) An Adaptive Approach to Recoverable Mutual Exclusion. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*. https://doi.org/10.1145/3382734.3405739

Dhoked, S. and N. Mittal (2021) Memory Reclamation for Recoverable Mutual Exclusion. In *CoRR abs/2103.01538*. https://arxiv.org/abs/2103.01538

Dhoked, S. and N. Mittal (2021) Adaptive and Fair Transformation for Recoverable Mutual Exclusion In *CoRR abs/2110.08308*. https://arxiv.org/abs/2110.08308

Gokhale, S., S. Dhoked, and N. Mittal (2021) On Group Mutual Exclusion for Dynamic Systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3437801.3441608

Skills and Abilities

Programming

- C, C++, C#, Java, Python
- Android, Windows Phone
- Sql Server, MySQL, ASP.NET, MVC, HTML

Tools

- LATEX, Git, C++ STL, pthreads, CUDA, cmd, bash, vim, tmux

Extra-Curricular

Represented IIT Indore chess and football teams in the 56th, 57th and 58th Inter-IIT Sports Meet (2010-2012)