Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity

Daniel Katzan
Tel Aviv University, Israel

Adam Morrison
Tel Aviv University, Israel

Abstract

We present the first recoverable mutual exclusion (RME) algorithm that is simultaneously abortable, adaptive to point contention, and with sublogarithmic RMR complexity. Our algorithm has $O(\min(K, \log_W N))$ RMR passage complexity and $O(F + \min(K, \log_W N))$ RMR super-passage complexity, where K is the number of concurrent processes (point contention), W is the size (in bits) of registers, and F is the number of crashes in a super-passage. Under the standard assumption that $W = \Theta(\log N)$, these bounds translate to worst-case $O(\frac{\log N}{\log \log N})$ passage complexity and $O(F + \frac{\log N}{\log \log N})$ super-passage complexity. Our key building blocks are:

- A D-process abortable RME algorithm, for $D \leq W$, with $O(1)$ passage complexity and $O(1 + F)$ super-passage complexity. We obtain this algorithm by using the Fetch-And-Add (FAA) primitive, unlike prior work on RME that uses Fetch-And-Store (FAS/SWAP).
- A generic transformation that transforms any abortable RME algorithm with passage complexity of $B < W$, into an abortable RME lock with passage complexity of $O(\min(K, B))$.

2012 ACM Subject Classification Theory of computation → Shared memory algorithms

Keywords and phrases Mutual exclusion, recovery, non-volatile memory

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.15

Related Version The full version of the paper is available at `http://arxiv.org/abs/2011.07622`.

Funding Israel Science Foundation (grant No. 2005/17) and Blavatnik ICRC at TAU.

1 Introduction

Mutual exclusion (ME) [10] is a central problem in distributed computing. A mutual exclusion algorithm, or *lock*, ensures that some *critical section* of code is accessed by at most one process at all times. To enter the critical section (CS), a process first executes an *entry section* to *acquire* the lock. After leaving the critical section, the process executes an *exit section* to *release* the lock. The standard complexity measure for ME is *remote memory references* (RMR) complexity [3, 6]. RMR complexity models the property that memory access cost on a shared-memory machine is not uniform. Some accesses are *local* and cheap, while the rest are *remote* and expensive (e.g., processor cache hits and misses, respectively). The RMR complexity measure thus charges a process only for remote accesses. There are various RMR definitions, modeling cache-coherent (CC) and distributed shared-memory (DSM) systems. The complexity of a ME algorithm is usually defined as its *passage complexity*, i.e., the number of RMRs incurred by a process as it goes through an entry and corresponding exit of the critical section.

For decades, the vast majority of mutual exclusion algorithms were designed under the assumption that processes are *reliable*: they do not crash during the mutual exclusion algorithm or critical section. This assumption models the fact that when a machine or program crashes, its memory state is wiped out. However, the recent introduction of non-volatile main memory (NVRAM) technology can render this assumption invalid. With NVRAM, memory state can remain persistent over a program or machine crash. This change creates the *recoverable mutual exclusion* (RME) problem [13], of designing an ME algorithm that can tolerate processes crashing and returning to execute the algorithm. In RME, a *passage* of a process p is defined as the execution fragment from when p enters the lock algorithm and until either p completes the exit section or crashes. If p crashes mid-passage and recovers, it re-enters the lock algorithm, which starts a new passage. Such a sequence of p's passages that ends with a crash-free passage (in which p acquires and releases the lock) is called a *super-passage* of p.

RME constitutes an exciting clean slate for ME research. Over the years, locks with many desired properties (e.g., fairness) were designed and associated complexity trade-offs were explored [27]. These questions are now re-opened for RME, which has spurred a flurry of research [7,9,11–14,18–21]. In this paper, we study such questions. In a nutshell, we introduce an RME algorithm that is *abortable*, *adaptive*, and has *sublogarithmic* RMR complexity. Our lock is the first RME algorithm adaptive to the number of concurrent processes (or *point contention*) and the first *abortable* RME algorithm with sublogarithmic RMR complexity. It is also the first deterministic, worst-case sublogarithmic abortable lock in the DSM model (irrespective of recoverability). Our algorithm also features other desirable properties not present in prior work, as detailed shortly.

Abortable ME & RME. An *abortable* lock [17,25,26] allows a process waiting to acquire the lock to give up and exit the lock algorithm in a finite number of its own steps. Jayanti and Joshi [21] argue that abortability is even more important in the RME setting. The reason is that a crashed process might delay waiting processes for longer periods of time, which increases the motivation for allowing processes to abort their lock acquisition attempt and proceed to perform other useful work.

Mutual exclusion, and therefore abortable ME (AME), incurs a worst-case RMR cost of $\Omega(\log N)$ in an N-process system with standard read, write, and comparison primitives such as Compare-And-Swap (CAS) or LL/SC [6]. This logarithmic bound is achieved for both ME [28] and AME [16], and was recently achieved for a recoverable, abortable lock by Jayanti and Joshi [21]. However, while there exists an AME algorithm with sublogarithmic worst-case RMR complexity (in the CC model) [2], no such abortable algorithm is known for RME. Moreover, Jayanti and Joshi's $O(\log N)$ abortable RME algorithm is suboptimal in a few ways. First, its worst-case RMR complexity is logarithmic only on a *relaxed* CC model, in which a failed CAS on a variable does not cause another process with a cached copy of the variable to incur an RMR on its next access to it, which is not the case on real CC machines. Their algorithm has linear RMR complexity in the realistic, standard CC model. Second, their algorithm is starvation-free only if the number of aborts is finite.

Adaptive ME. A lock is *adaptive* with respect to *point contention* if its RMR complexity depends on K, the number of processes concurrently trying to access the lock, and not only on N, the number of processes in the system. Adaptive locks are desirable because they are often faster when $K \ll N$. There exist locks with worst-case RMR cost of $O(\min(\log N, K))$ for both ME [15] and AME [16], but no adaptive RME algorithm is known (independent of abortability).

1.1 Overview of Our Results

In the following, we denote the number of crashes in a super-passage by F and the size (in bits) of the system's registers by W. We obtain three keys results, which, when combined, yield the first RME algorithm that is simultaneously abortable and adaptive, with worst-case $O(\log_W N)$ passage complexity and $O(F + \log_W N)$ super-passage complexity, in both CC and DSM models. Assuming (as is standard) that $W = \Theta(\log N)$, this translates to worst-case $O(\frac{\log N}{\log \log N})$ passage complexity and $O(F + \frac{\log N}{\log \log N})$ super-passage complexity. In contrast to Jayanti and Joshi's abortable RME algorithm [21], our lock achieves sublogarithmic RMR complexity in the *standard* CC model and is unconditionally starvation-free. Our algorithm's space complexity is a static (pre-allocated) $O(NW \log_W N)$ memory words (which translates to $O(\frac{N \log^2 N}{\log \log N})$ if $W = \Theta(\log N)$). Jayanti and Joshi's algorithm also uses static memory, but it relies on unbounded counters. The other sublogarithmic RME algorithms [9, 11, 18] use dynamic memory allocation, and may consume unbounded space.

Result #1: W-process abortable RME with $O(1)$ passage and $O(1 + F)$ super-passage complexity (§ 3). Our key building block is a D-process algorithm, for $D \leq W$. It has constant RMR cost for a passage, regardless of if the process arrives after a crash. The novelty of our algorithm is that it uses the Fetch-And-Add (FAA) primitive to beat the $\Omega(\log D)$ passage complexity lower-bound. In contrast, the building blocks in prior RME work with worst-case sublogarithmic RMR complexity use the Fetch-And-Store (FAS, or SWAP) primitive and assume no bound on D, even though they are ultimately used by only a bounded number of processes in the final algorithm. By departing from FAS and exploiting the process usage bound, we overcome difficulties that made the prior algorithms' building blocks [11, 18] have only $O(D)$ RMR passage complexity.

These prior algorithms use a FAS-based queue-based lock as a building block. They start with an $O(1)$ RMR queue-based ME algorithm [8, 23], in which a process trying to acquire the lock uses FAS to append a node to the queue tail, and then spins on that node waiting for its turn to enter the critical section. Unfortunately, if the process crashes after the FAS, before writing its result to memory, then when it recovers and returns to the algorithm, it does not know whether it has added itself to the queue and/or who is its predecessor (previously obtained from the FAS response). To overcome this problem, a recovering process reconstructs the queue state into some valid state, *which incurs a linear number of RMRs*. The recovery procedure is blocking (not wait-free), and multiple processes cannot recover concurrently. Overall, these prior building blocks have $O(D)$ passage complexity and $O(1 + FD)$ super-passage complexity. In contrast, our D-process abortable RME algorithm has $O(1)$ passage complexity and $O(1 + F)$ super-passage complexity, has wait-free recovery, and allows multiple processes to recover concurrently. While other $O(1)$ RME algorithms exist, they either assume a weaker crash model [12], rely on non-standard primitives that are not available on real machines [11, 19], or obtain only amortized, not worst-case, $O(1)$ RMR complexity [7].

Result #2: Tournament tree with wait-free exit (§ 4). In both ours and prior work [11,18], the main lock is obtained by constructing a *tournament tree* from the D-process locks. The tree has N leaves, one for each process. Each internal node is a D-process lock, so the tree has height $O(\log_D N)$. To acquire the main lock, a process competes to acquire each lock on the path from its leaf to the root, until it wins at the root and enters the critical section. Our algorithm differs from prior tournament trees in a couple of simple ways, but which have important impact.

First: In our tree, a process that recovers from a crash returns directly to the node in which it crashed. This allows us to leverage our node lock's $O(1+F)$ super-passage complexity to obtain $O(H + F)$ super-passage complexity for the tree, where H is the tree's height. By taking $D = W = \Theta(\log N)$, our overall lock has $O(F + \frac{\log N}{\log \log N})$ super-passage complexity and $O(\frac{\log N}{\log \log N})$ passage complexity. In contrast, prior trees perform recovery by having a process restart its ascent from the leaf. In fact, in these algorithms, there is no asymptotic benefit from returning directly to the node where the crash occurred. The reason is that

Table 1 Comparison of RME algorithms. (SP: super-passage, WF: wait-free, $\mathbf{F}^*$: total number of crashes in the system, FASAS: Fetch-And-Swap-And-Swap.) All algorithms satisfy starvation-freedom, wait-free critical-section re-entry, and wait-free exit (defined in § 2).

Algorithm	**Passage Complexity**	**Super-Passage Complexity**	**Primitives Used**	**Space Complexity**	**Additional Properties**
Golab & Ramaraju [14, Section 4.2] with MCS [23] as base lock	$O(1)$ (no concurrent crashes)	$O(1)$ (no concurrent crashes)	CAS, FAS	$O(N \log N)$	
	$O(\log N)$ (concurrent crashes)	$O(\log N)$ (concurrent crashes)			
	$O(N)$ (if crashes)	$O(FN)$ (if crashes)			
Jayanti & Joshi [20]	$O(\log N)$	$O(\log N + F)$	CAS	$O(N \log N)$	FCFS, SP WF Exit
Jayanti, Jayanti, & Joshi [18]	$O(\frac{\log N}{\log \log N})$	$O((1+F)(\frac{\log N}{\log \log N}))$	FAS	Unbounded	
Jayanti, Jayanti, & Joshi [19]	$O(1)$	$O(1)$ in the DSM model $O(F)$ in the CC model	FASAS	$O(N)$	SP WF Exit
Chan & Woelfel [7]	$O(1)$ *amortized*	same as passage complexity	CAS, FAA	Unbounded	SP WF Exit
Dhoked & Mittal [9]	$O(min(\sqrt{F^*}, \frac{\log N}{\log \log N})$	same as passage complexity	CAS, FAS	Unbounded	**Crash-adaptive**
Jayanti & Joshi [21]	$O(\log N)$	$O(\log N + F)$	CAS	$O(N \log N)$	**Abortable**, SP WF Exit, FCFS
This work	$\boldsymbol{O(\min(K, \frac{\log N}{\log \log N}))}$	$\boldsymbol{O(\min(K, \frac{\log N}{\log \log N}) + F)}$	**FAA, CAS**	$\boldsymbol{O(\frac{N \log^2 N}{\log \log N})}$	**Abortable, adaptive, SP WF Exit**

node lock recovery in these trees has $O(D)$ complexity, so to obtain overall sublogarithmic complexity, they take $D = \frac{\log N}{\log \log N}$, which means that node crash recovery costs the same as climbing to the node. Consequently, their overall super-passage complexity is multiplicative in F, $O((1+F)\frac{\log N}{\log \log N})$, instead of additive as in our tree.

Second: Our tree's exit section is wait-free (assuming finitely many crashes). In contrast, in the prior trees, a process that crashes during its exit section might subsequently block. The reason is a subtle issue related to composition of RME locks. The model in these works [11,18] is that a process p that crashes in its exit section must complete a crash-free passage upon recovery (i.e., re-enter the critical section and exit it again). Thus, p must re-ascend to the root after recovering. Each node lock satisfies a *bounded CS re-entry* property, which allows p to re-enter the node's CS (i.e., ascend) without blocking – provided that p crashed inside the node's CS. However, this property does not apply if p released the node lock (i.e., descended) before crashing. For such a node, p simply attempts to re-acquire the node lock. Consequently, p might block during its recovery, even though logically *it is only trying to release the overall lock.* We address this problem by carefully modeling the interface of an RME algorithm in a way that facilitates composition, which enables a recovering process to avoid re-acquiring node locks it had already released. Our overall algorithm thereby satisfies a new *super-passage wait-free exit* property.

Result #3: Generic RME adaptivity transformation (§ 5). We present a generic transformation that transforms any abortable RME algorithm with passage complexity of $B < W$ into an abortable RME lock with passage complexity of $O(\min(K, B))$, where K is the number of processes executing the algorithm concurrently with the process going through the super-passage, i.e., the *point contention.* Applying this transformation to our tournament tree lock yields the final algorithm.

Summary of contributions and related work. Table 1 compares our final algorithm to prior RME work. Dhoked and Mittal [9] use a definition of "adaptivity" that requires RMR cost to depend on the total number of crashes; we refer to this property as *crash-adaptivity.* Crash-adaptivity is thus orthogonal to the traditional notion of adaptivity [5]. Chan & Woelfel's algorithm [7] uses FAA, but it is used to assign processes with tickets, which is different from our technique (§ 3). Their algorithm has only an amortized RMR passage complexity bound and its worst-case RMR cost is unbounded.

2 Model and Preliminaries

Model. We consider a system in which N deterministic, asynchronous, and unreliable processes communicate over a shared memory. The shared memory, M, is an array of $\Theta(W)$-bit words. (Henceforth, we refer to the shared memory simply as "memory"; process-private variables are not part of the shared memory.) The system supports the standard read, write, CAS, and FAA operations. $CAS(a, o, n)$ atomically changes $M[a]$ from o to n if $M[a] = o$ and returns true; otherwise, it returns false without changing $M[a]$. $FAA(a, x)$ atomically adds x to $M[a]$ and returns $M[a]$'s original content.

A *configuration* consists of the state of the memory and of all processes, where the state of process p consists of its internal program counter and (non-shared) variables. Given a configuration σ, an *execution fragment* is a (possibly infinite) sequence of *steps*, each of which moves the system from one configuration to another, starting from σ. In a *normal step*, some process p invokes an operation on a memory word and receives the operation's response. In a *crash* step, the state of some process p resets to its initial state (but the memory state remains unchanged). An *execution* is an execution fragment starting from the system's initial configuration.

Notation. Given an execution fragment α, if β is a subsequence of α, we write $\beta \subseteq \alpha$. If e is a step taken in α, we write $e \in \alpha$. If e is the t-th step in an execution E, we say that e *is at time* t. We use $[t, t']$ to denote the subsequence of E whose first and last steps are at times t and t' in E, respectively.

RMR complexity. The RMR complexity measure breaks the memory accesses by a process p into *local* and *remote* references, and charges p only for remote references. We consider two types of RMR models. In the DSM model, each memory word is local to one process and remote to all others, and process p performs an RMR if it accesses a memory word remote to it. In the CC model, the processes are thought of as having coherent caches, with RMRs occurring when a process accesses an uncached memory word. Formally: (1) every write, CAS, or FAA is an RMR, and (2) a read by p of word x is an RMR if it is the first time p accesses x or if after p's prior access to x, another process performed a write, CAS, or FAA on x.

Recoverable mutual exclusion (RME). Our RME model draws from the models of Golab and Ramaraju [14] and Jayanti and Joshi [20]. In the spirit of [14], we model the RME algorithm as an object exporting methods invoked by a client process. In the spirit of [20], we require recovery to re-execute the section in which the crash occurred, rather than restart the entire passage. An *RME algorithm* (or *lock*) provides the methods *Recover*, *Try*, and *Exit*. (In the code, we show the methods taking an argument specifying the calling process' id.) If process p invokes *Try* and it returns TRUE, then p has *acquired* the lock and *enters* the critical section (CS). Subsequently, p *exits* the CS by invoking *Exit*. If *Exit* completes, we say that p has *released* the lock. The *Recover* method guides p's execution after a crash, which resets p to its initial state. We assume p's initial state is to invoke *Recover*, which returns $r \in \{TRY, CS, EXIT\}$. If $r = TRY$, p invokes *Try*. If $r = CS$, p enters the CS. If $r = EXIT$, p invokes *Exit*.

A *super-passage* of p begins with p completing *Recover* and invoking *Try*, either for the first time, or for the first time after p's prior super-passage ended. The super-passage ends when p completes *Exit*. A *passage* of p begins with p starting a super-passage, or when p invokes *Recover* following a crash step. The passage ends at the earliest of p completing *Exit* or crashing. We refer to an L-passage (or L-super-passage) to denote the lock L that a passage (or super-passage) applies to; similarly, we refer to a step taken in lock L's code as an L-step. We omit L when the context is clear. These definitions facilitate composition of RME locks. For instance, suppose that process p is releasing locks in a tournament tree and crashes after releasing some node lock L. When p recovers, it can invoke $L.Recover$, which will return TRY, and thereby learn that it has released L and can descend from it – without the *Recover* invocation counting as starting a new L-super-passage.

Well-formed executions formalize the above described process behavior:

▶ **Definition 1.** *An execution is* well-formed *if the following hold for every lock L and process p:*

1. Recover invocation*: p's first L-step after a crash step is to invoke L.Recover.*
2. Try invocation*: p invokes L.Try only if p is starting a new L-super-passage, or if p's prior crash step was during L.Try.*
3. CS invocation*: p enters the CS of L only if p receives $TRUE$ from L.Try in its current L-passage, or if p's prior crash step was during the CS.*
4. Exit invocation*: p invokes L.Exit only if p is in the CS of L, or if p's prior crash step was during L.Exit.*

Henceforth, we consider only well-formed execution. We also consider only *well-behaved* RME algorithms, in which *Recover* correctly identifies where a process crashes:

▶ **Definition 2.** *An RME algorithm is* well-behaved *if the following hold, for every process p and every* well-formed *execution:*

1. *p's first complete invocation of* Recover*, and p's first complete invocation of* Recover *following a complete passage of* Exit*, returns TRY.*
2. *p's first complete invocation of* Recover *following a crash during* Try *return TRY.*
3. *p's first complete invocation of* Recover *following a crash during the CS returns CS.*
4. *p's first complete invocation of* Recover *following a crash during* Exit *returns $EXIT$.*
5. *A complete invocation of* Recover *by p during the CS returns CS.*

Note: We consider p to be in the *Try* or *Exit* section from the time it executes the first memory operation of that section and until it either crashes or executes the last memory operation of that section. Thus, p is considered to be in the CS after it executes its final *Try* memory operation.

Fairness. We make a standard fairness assumption on executions: once p starts a super-passage, it does not stop taking steps until the super-passage ends.

Abortable RME. At any point during its super-passage, process p can non-deterministically choose to abort its attempt, which we model by p receiving an external *abort* signal that remains visible to p throughout the super-passage (i.e., including after crashes) and resets once p finishes the super-passages. Abortable RME extends the definition of a super-passage as follows. If p is signalled to abort and its execution of *Try* returns FALSE, then p has aborted and the super-passage ends. (It is not mandatory for *Try* to return FALSE, because an abort may be signalled just as p acquires the lock.)

***D*-ported locks.** We model locks that may be used by at most D processes concurrently as follows. In a *D-ported lock*, each process invokes the methods with a *port* argument, $1 \le k \le D$, which acts as an identifier. We augment the definition of a well-formed execution to include the following conditions:

5. *Constant port usage*: For every process p and L-super-passage of p, p does not change its port for L throughout the super-passage.
6. *No concurrent super-passages*: For any L-super-passages sp_i and sp_j of processes $p_i \neq p_j$, if sp_i and sp_j are concurrent, then p_i's port for L in sp_i is different than p_j's port for L in sp_j. (Two super-passages are not *concurrent* if one ends before the other begins.)

Problem statement. Design a well-behaved abortable RME algorithm with the following properties.

1. **Mutual exclusion**: At most one process is in the CS at any time t.
2. **Deadlock-freedom**: If a process p starts a super-passage sp at time t, and does not abort sp, and if every process that enters the CS eventually leaves it, then there is some time $t' > t$ and some process q such the q enters the CS in time t', *or else there are infinitely many crash steps.*
3. **Bounded abort**: If a process p has abort signalled while executing *Try*, and executes sufficiently many steps without crashing, then p complete its execution of *Try*.

The following properties are also desirable, and all but FCFS are satisfied by our algorithm:

4. **Starvation-freedom**: If the total number of crashes in the execution is finite and process p executes infinitely many steps and every process that enters the CS eventually leaves it, then p enters the CS in each super-passage in which it does not receive an abort signal.
5. **CS re-entry**: If process p crashes while in the CS, then no other process enters the CS from the time p crashes to the time when p next enters the CS.
6. **Wait-free CS re-entry**: If process p crashes in the CS, and executes sufficiently many steps without crashing, then p enters the CS.
7. **Wait-free exit**: If process p is executing *Exit*, and executes sufficiently many steps without crashing, then p completes its execution of *Exit*.
8. **Super-passage wait-free exit**: If process p is executing *Exit*, then p completes an execution of *Exit* after a finite number of its own steps, *or else p crashes infinitely many times.* (Notice that p may crash and return to re-execute *Exit*.)
9. **First-Come-First-Served (FCFS)**: If there exists a bounded section of code in the start of the entry section, referred to as the *doorway* such that, if process p_i finishes the doorway in its super-passage sp_i for the first time before some process p_j begins its doorway for the first time in its super-passage sp_j, and p_i does not abort sp_i, then p_j does not enter the CS in sp_j before p_i enters the CS in sp_i.

Super-passage wait-free exit is a novel property introduced in this work. It guarantees that a process completes *Exit* in a finite number of its own steps, as long as it only crashes finitely many times. Wait-free exit does not imply super-passage wait-free exit since it does not apply if the process crashes during *Exit*. Clearly, starvation-freedom implies deadlock-freedom, wait-free CS re-entry implies CS re-entry, and super-passage wait-free exit implies wait-free exit.

Lock complexity. The *passage complexity* (respectively, *super-passage complexity*) of a lock is the maximum number of RMRs that a process can incur while executing a passage (respectively, super-passage). We denote by F the maximum number of times a process crashes in an execution.

3 W-Port Abortable RME Algorithm

Here, we present our D-process abortable RME algorithm, for $D \leq W$, which has $O(1)$ passage RMR complexity and $O(1+F)$ super-passage complexity. The algorithm is similar in structure to Jayanti and Joshi's abortable RME algorithm [21], in that it is built around a recoverable auxiliary object that tracks the processes waiting to acquire the lock. This object's RMR complexity determines the algorithm's complexity. Non-abortable RME locks implement such an object with a FAS-based linked list [11, 18]. Such a list has $O(1+FD)$ super-passage complexity – i.e., a crash-free passage incurs $O(1)$ RMRs – but it is hard to make abortable. Jayanti and Joshi instead use a recoverable *min-array* [15]. This object supports aborting, but its passage complexity is logarithmic, even in the absence of crashes.

Our key idea is to represent the "waiting room" object with a FAA-based W-bit mask (a single word), where a process p arriving/leaving is indicated by flipping a bit associated with p's port. The key ideas are that (1) if p crashes and recovers, it can learn its state in $O(1)$ RMRs simply by reading the bit mask and (2) the algorithm carefully avoids relying on any FAA's return value. Our design thus obtains the best of both worlds: the object can be updated with $O(1)$ RMRs as well as supports efficient aborting (with a single bit flip). The trade-off we make in this design choice is that we only guarantee starvation-freedom, but not FCFS. Unlike a min-array, the bit mask cannot track the order of arriving processes, as bit setting operations commute. We do, however, track the order in which processes acquire the lock, and thereby guarantee starvation-freedom.

Our algorithm guarantees starvation-freedom unconditionally, even if there are infinitely many aborts. This turns out to be a subtle issue to handle correctly (§ 3.2), and the Jayanti and Joshi algorithm is prone to executions in which a process that does not abort starves as a result of other processes aborting infinitely often (we show an example in § 3.2).

Since we assume W-bit memory words, we are careful not to use unbounded, monotonically increasing counters, which the Jayanti and Joshi lock does use. Our algorithm's RMR bounds are in both the DSM and CC models, whereas the Jayanti and Joshi lock has linear RMR complexity on the standard CC model.

3.1 Algorithm Walk-Through

Figure 1 presents the pseudo code of the algorithm. We assume participating processes uses distinct ports in the range $0, \dots, W-1$, so we refer to processes and ports interchangeably. For simplicity, we present the algorithm assuming dynamic memory allocation with safe reclamation [24]. In this environment, a process can *allocate* and *retire* objects, and it is guaranteed that an allocation does not return a previously-retired object if some process still has a reference to that object. We show how to satisfy this assumption (with $O(D^2)$ static, pre-allocated memory) in the full version [22, Appendix A].

Each process p has a status word, $STATUS[p]$, and a pointer to a boolean spin variable, $GO[p]$. (In the DSM model, a process allocates its spin variables from local memory, so that it can spin on them with $O(1)$ RMR cost.) The lock's state consists of a W-bit word, $ACTIVE$, and a $\Theta(W)$-bit word, $LOCK\_STATUS$. The $LOCK\_STATUS$ word holds a tuple $(taken, owner, owner\_go)$, where $taken$ is a bit indicating if the lock is acquired by some process. If $taken$ is set, $owner$ is the id (port) of the lock's owner and $owner\_go$ points to the owner's spin variable.

The $STATUS$ word of each process p, initialized to TRY, indicates in which section the process is currently at. This information is used by *Recover* to steer p to the right method when it arrives. The $STATUS$ word changes when completing *Try* and entering the CS, when aborting during *Try*, when exiting the CS and executing *Exit*, and when *Exit* completes. Note that the *Exit* method may be called as a subroutine during the *Try* section's abort flow. In this case, its operations are considered part of the *Try* section (i.e., the subroutine call is to avoid putting a copy of *Exit*'s code in *Try*). To distinguish these subroutine calls from when a process invokes *Exit* to exit the CS, we add an *abort* argument to *Exit*, which is *FALSE* if and only if *Exit* is invoked to exit the CS (i.e., not as a subroutine).

In the normal (crash- and abort-free) flow, a passage of process p proceeds as follows. First, p allocates its spin variable, if it does not currently exist (lines 11–16). Then p flips its bit in the $ACTIVE$ word, but only if p's bit is not already set (lines 17–18). This check avoids corrupting $ACTIVE$ when p recovers from a crash. Next, p executes a *Promote* procedure, which tries to pick some waiting process (possibly p) and make it the owner of the lock, if the lock is currently unowned (line 19). Finally, p begins spinning on its spin variable, waiting for an indication that it has become the lock owner (lines 20–25). Upon exiting the CS, p clears its bit in $ACTIVE$ (again, only if the bit is currently set, to handle crash recovery) (lines 39–40). Then p executes *Promote* (line 41). Performing this call will have no effect, since p is still holding the lock, which may appear strange, but is required in order to support the abort flow, as explained shortly. Then, if p is indeed the lock owner (another check useful only in the abort flow), it releases the lock by clearing the *taken* bit in $LOCK\_STATUS$ (lines 42–45). Note that p leaves the *owner* and *owner_go* fields intact, for reasons described shortly. Finally, p executes *Promote* again, to hand the lock off to some waiting process (line 46). It then retires its spin variable, clears its GO pointer, and updates its $STATUS$ to TRY, thereby completing *Exit* and thus its current passage and super-passage (line 47–50).

```
1   ACTIVE: int // initially 0
2   STATUS: array of W status words // initially all TRY
3   GO: array of W pointers to booleans // initially all ⊥
4   LOCK_STATUS: struct {bool, port_id, bool*}
5   // initially (0, 0, ⊥)

7   void Try(int k) {
8       if STATUS[k] = ABORT:
9           Exit(k, TRUE)
10          return FALSE
11      if GO[k] = ⊥:
12          if got abort signal:
13              STATUS[k] := ABORT
14              Exit(k, TRUE)
15              return FALSE
16          GO[k] := new Bool()
17      if k-th bit in ACTIVE is 0:
18          FAA(ACTIVE, 2^k)
19      Promote(⊥)
20      while *GO[k] = FALSE:
21          if got abort signal:
22              STATUS[k] := ABORT
23              Exit(k, TRUE)
24              return FALSE
25
26      STATUS[k] := CS
27      return TRUE
28  }
29  status Recover(int k) {
30      if STATUS[k] = EXIT:
31          return EXIT
32      if STATUS[k] = CS:
33          return CS
34      return TRY
35  }
```

```
36  void Exit(int k, bool abort) {
37      if abort = FALSE:
38          STATUS[k] = EXIT
39      if k-th bit in ACTIVE is 1:
40          FAA(ACTIVE, -2^k)
41      Promote(k)
42      (taken, owner, owner_go) := LOCK_STATUS
43      if taken = 1 and owner = k:
44          CAS(LOCK_STATUS, (1, owner, owner_go),
45                  (0, owner, owner_go))
46      Promote(⊥)
47      if GO[k] ≠⊥
48          Retire(GO[k])
49          GO[k] := ⊥
50      STATUS[k] := TRY
51  }
52  void Promote(int j) {
53      (taken, owner, owner_go) := LOCK_STATUS
54      if taken = 0:
55          active := ACTIVE
56          if active ≠ 0:
57              j := next(owner, active)
58          if j ≠⊥:
59              CAS(LOCK_STATUS, (0, owner, owner_go),
60                      (1, j, GO[j]))
61      (taken, owner, owner_go) := LOCK_STATUS
62      if taken = 1:
63          *owner_go := TRUE
64  }
```

Figure 1 W-port abortable RME algorithm.

If p receives the abort signal while spinning in Try, it sets its $STATUS$ to $ABORT$, executes the *Exit* method as a subroutine, and returns $FALSE$ (label 12–15). If p crashes during the execution of *Exit*, *Recover* will steer it to Try once it recovers, at which point it will again execute the *Exit* method and return $FALSE$. In the abort flow, the call to *Exit* does not modify p's $STATUS$ (the if is not taken, lines 37–38).

The main goal of $Promote(j)$ is to promote some waiting process to be the lock owner, if the lock is currently unowned. *Promote* tries to promote one of the waiting processes (as specified by ACTIVE). If there is no such process, then *Promote* tries to promote process j if $j \neq \perp$, and does not promote any process otherwise (lines 53–60). A secondary goal of *Promote* is that it signals the (current or newly promoted) owner by writing to its spin variable (lines 61–63). Picking a process to promote from among the waiting processes is done in a manner that guarantees starvation-freedom. To this end, *Promote* picks the next id whose bit is set in $ACTIVE$, when ids are scanned starting from the previous owner's id (which, as described above, is written in $LOCK\_STATUS$) and moving up (modulo W). (In the code, this is specified as $next(owner, active)$.) Having picked a process q to promote, *Promote* tries to update $LOCK\_STATUS$ to $(1, q, GO[q])$ using a single CAS. Finally, before completing, *Promote* checks again if the lock is owned by some process r (possibly $r \neq q$), and if so, signals r by writing $TRUE$ to r's spin variable.

The reason for executing *Promote* in *Exit* before releasing the lock, and not only afterwards, is to handle a scenario in which the lock owner q has released the lock and $next(q, ACTIVE) = p$, so any process r (possibly, but not necessarily, q) executing *Promote* tries to hand the lock to p. If now p is signalled to abort, and did not also execute *Promote* before departing, deadlock would occur. By having p call $Promote(p)$, we guarantee that either (1) some process (possibly p) promotes p, so p's *Exit* call releases the lock before completing the abort; or (2) some process r (possibly, but not necessarily p), which does not observe p in $ACTIVE$, updates $LOCK\_STATUS$ from $(0, q, G)$ to $(1, q', G')$. In the latter case, our memory management assumption implies that $LOCK\_STATUS$ will not recycle to contain $(0, q, G)$ before every processes that has read $(0, q, G)$ from $LOCK\_STATUS$ executes its CAS. All such CASs, who are about to change $(0, q, G)$ to $(1, p, GO[p])$ thus fail, so the lock does not get handed to p and no deadlock occurs after it completes its abort.

3.2 Discussion: Guaranteeing Starvation-Freedom In the Presence of Infinitely Many Aborts

As discussed in § 3.1, a key idea in our algorithm is to invoke *Promote* even before releasing the lock, to handle the case in which the lock is about to be handed to an aborting process. While simple, this is a subtle idea, because a different (more straightforward) approach to dealing with this issue can lead to starvation. We explain the issue by describing and analyzing a starvation problem in Jayanti and Joshi's abortable RME algorithm [21]. The structure of our algorithm and of Jayanti and Joshi's algorithm is similar, if one thinks of our $ACTIVE$ word and their min-array as abortable objects which (1) maintain the set of waiting processes and (2) have some notion of the "next in line" waiting process, which becomes the lock owner. (Jayanti and Joshi refer to this object as a *registry*.) We describe the problem in the Jayanti and Joshi lock by contrasting its behavior with our algorithm's.

Intuitively, starvation-freedom should follow from property (2) of the "waiting room" object, because every process executing *Promote* will eventually agree on the process p to promote, which would then become the lock owner. For this to be true, however, aborts need to be handled very carefully. Phrased in our terminology, in the Jayanti and Joshi algorithm, a process p that receives an abort signal starts executing *Exit*, where it removes itself from the “waiting room” object. Subsequently, if $LOCK\_STATUS = (0, o, os)$, p tries (using a single CAS) to update $LOCK\_STATUS$ from $(0, o, os)$ to $(0, p, GO[p])$. In other words, p tries to make it look as if it had acquired the lock and immediately released it. The motivation for this step is to fail any *Promote* that is about to make p the lock owner, which if not handled, would result in deadlock.

This approach has the unfortunate side-effect of failing concurrent *Promote*s even if they are not about to make p the lock owner. This can lead to an execution in which aborting processes prevent the lock from being acquired, as described next.

Process p_1 arrives and enters the critical section. Process p_2, p_3, p_4 arrive and enter the waiting room. Now p_1 leaves the CS and executes *Exit*, which (in Jayanti and Joshi's algorithm) has a single *Promote* call, after releasing the lock. Suppose the “waiting room” object indicates that p_2 should be the next lock owner. Now, p_1 stops in its *Promote* call, just before CASing $LOCK\_STATUS$ from $(0, p_1, *)$ to $(1, p_2, *)$. Next, p_3 aborts, executes the *Exit* code and successfully changes $LOCK\_STATUS$ to $(0, p_3, *)$.

As a result, p_1's CAS in *Promote* fails. p_1 completes its *Exit* section and then returns to the Try section, executes *Promote*, and stops just before CASing $LOCK\_STATUS$ from $(0, p_3, *)$ to $(1, p_2, *)$. Now, p_3 proceeds to the *Promote* call in *Exit*, stopping just before CASing $LOCK\_STATUS$ from $(0, p_3, *)$ to $(1, p_2, *)$. We have reached a state in which p_4 is waiting, $LOCK\_STATUS$ is $(0, p_3, *)$, p_1 is in its Try *Promote* and p_3 is in its *Exit* promote, both about to CAS $LOCK\_STATUS$ from $(0, p_3, *)$ to $(1, p_2, *)$.

We continue as follows. Now p_4 receives the abort signal, proceeds to execute *Exit*, and successfully changes $LOCK\_STATUS$ from $(0, p_3, *)$ to $(0, p_4, *)$. Consequently, the CAS of both p_1 and p_3 fails, so p_1 enters the waiting room, whereas p_3 departs the algorithm, returns, and stops in the Try *Promote* before CASing $LOCK\_STATUS$ from $(0, p_4, *)$ to $(1, p_2, *)$. As for p_4, it enters the Exit *Promote* and stops before CASing $LOCK\_STATUS$ from $(0, p_4, *)$ to $(1, p_2, *)$. We have reached a similar situation as in the previous paragraph, and can therefore keep repeating this scenario indefinitely. Throughout, p_2 keeps taking steps in the waiting room, but will never enter the CS.

3.3 Proofs of RME Properties

We refer to our W-port abortable RME algorithm as Algorithm M. In the the full version [22], we prove the following theorem:

▶ **Theorem 3.** *If every execution of Algorithm M is well-formed, then Algorithm M satisfies mutual exclusion, bounded abort, starvation-freedom, CS re-entry, wait-free CS re-entry, wait-free exit, and super-passage wait-free exit. The passage complexity of Algorithm M in both the CC and DSM models is $O(1)$ and the super-passage complexity is $O(1 + F)$. (Assuming, for the DSM model, that process memory allocations return local memory.) The space complexity of the algorithm is $O(D^2)$.*

Here, we omit the proof, due to space constraints, and point out of some of its high-level aspects. Whenever a process p starts a super-passage in our algorithm, it allocates a fresh spin variable. To avoid unbounded space consumption, the memory used for spin variables eventually has to be recycled, i.e., an allocation by process p can return a variable it previously used. Our proofs assume that this recycling is done safely, namely, that an allocation of a new spin variable does not return an object that is currently being referenced by some process. (We show how to satisfy this assumption using $O(D^2)$ static pre-allocated memory words in the full version [22, Appendix A].)

The above safe memory management assumption implies two properties that we use throughout the proofs. First, that if a process p is about to CAS $LOCK\_STATUS$ in *Promote*, and $LOCK\_STATUS$ has changed between p last reading it and executing the CAS, then the CAS will fail. This holds because $LOCK\_STATUS$ necessarily contains a different *owner_go* value. Second, that if p sets the spin variable of q to $TRUE$ and q has already started a new super-passage, then q will never read that $TRUE$ value. This holds because q allocates a different spin variable for its new super-passage.

4 Tournament Tree

A *tournament tree* lock, referred to as the *main* lock, is constructed by statically arranging multiple D-port RME algorithms, referred to as *node* locks, in a D-ary tree with N leaves (we assume $D \leq W$). Each leaf is uniquely associated with a process. To acquire the main lock, a process competes to acquire each lock on the path from its leaf to the root, until it wins at the root and enters the main lock's CS. To release the main lock, the process descends from the root to its leaf, releasing each node lock on the path. In this section, we present our tournament tree algorithm.

Our algorithm has two distinguishing features: (1) that its super-passage RMR complexity is additive in F, the number of crashes, and not multiplicative; and (2) that it satisfies *super-passage wait-free exit* (SP-WF-Exit), i.e., a process releasing the main lock is guaranteed to complete some execution of *Exit* after a finite number of its own steps (including crashes).

Our algorithm's super-passage RMR complexity is $O(FR+B\log_D N)$, where R and B are the recovery cost and passage complexity of the node lock, respectively. In comparison, prior trees have super-passage complexity of $O(F(R+B\log_D N))$. Obtaining our bound is simple: a process just needs to write its location in the tree to NVRAM, so that upon crash recovery, it can resume from there instead of starting to walk up or down the tree from scratch. We suspect that this simple optimization was not performed in prior tournament trees because their node lock has $R = \log_D N = O(\frac{\log N}{\log \log N})$ and $B = O(1)$, so directly returning to the node at which the crash occurred does not asymptotically improve complexity. With our W-port RME algorithm, however, $R = B = O(1)$, so being additive in F is asymptotically better, and would not be obtained using prior tournament trees.

The problem of obtaining SP-WF-Exit highlights the difficulty of composing recoverable locks. The issue is that a process in the main lock is composing critical sections of the node locks, which creates the problem of how recovery of the main and node locks interact. In the model of prior work [11, 14], a process crashing in the main lock's exit section attempts to re-acquire the main lock upon recovering. As a result, the process might now block in some node lock's entry section, which violates SF-WF-Exit for the main lock. We address this problem by carefully modeling RME algorithms in a way that facilitates composition (§ 2). Instead of assuming how a process participates in the algorithm (i.e., cycling through entry, CS, exit), we model the RME algorithm as an object whose *Recover* procedure informs the process where it crashed in the super-passage. This approach allows client algorithms, composing the lock, to decide how to proceed. Our model allows a process returning to lock x after crashing in the main lock to realize that it had completed an x-super-passage *and not start a new one*. Consequently, our tournament tree avoids the problems described above and satisfies SP-WF-Exit.

We present detailed pseudo code and prove all of the algorithm's properties. Due to space limits, omitted proofs appear in the full version [22, Appendix C].

```
 1  STATUS: array of N status words // initially all TRY.
 2  CURR_NODE: array of N nodes
 3  // initially CURR_NODE[i] is the i-th leaf.

 5 void Try(int pid) {
 6     if STATUS[pid] = ABORT:
 7         Exit(k, TRUE)
 8         return FALSE
 9     node = CURR_NODE[pid]
10     while STATUS[pid] ≠ CS or node ≠ root:
11         if node is the j-th child of node.parent,
12         then set k to j
13         if node.Recover(k) = TRY:
14             node.Try(k)
15         if received abort signal:
16             STATUS[pid] := ABORT
17             Exit(pid, TRUE)
18             return FALSE
19         if node = root:
20             break
21         node := node.parent
22         CURR_NODE[pid] := node
23     STATUS[pid] := CS
24     return TRUE
25 }
```

```
27 void Exit(int pid, bool aborting) {
28     if aborting = FALSE:
29         STATUS[pid] = EXIT
30     node := CURR_NODE[pid]
31     while TRUE:
32         if node is the j-th child of node,
33         then set k to be j
34         if node.Recover(k) ≠ TRY:
35             node.Exit(k, FALSE)
36         if node = LEAF:
37             break
38         node := node.child(k)
39         CURR_NODE[pid] := node
40     STATUS[pid] := TRY
41 }
42 status Recover(int pid) {
43     if STATUS[pid] = EXIT:
44         return EXIT
45     if STATUS[pid] = CS:
46         return CS
47     return TRY
48 }
```

Figure 2 The Tournament Tree.

4.1 Algorithm Walk-Through

Figure 2 shows the pseudo code of the algorithm. Each node has immutable *parent* and *child* pointers (as mentioned before, the tree structure is static). The *parent* of root is $\perp$, as are all *child* pointers of a leaf node. Each process is statically assigned to a leaf based on its id (*pid*). Each node contains a D-port abortable RME lock.

Similarly to our W-port algorithm, each process p has a status word, $STATUS[p]$, which is used by the main lock's *Recover* procedure. Each process has a *current_node* pointer.

In Try, a process walks the path from its leaf to the root, acquiring each node lock along the way (lines 10–22). In each such lock, it uses a statically assigned port, corresponding to the number of the child from which it climbed into the node. After successfully acquiring the lock at node x, process p writes x to *current_node*$[p]$ (line 22). This allows p to return to x if it crashes, instead of having to start from scratch and climb the entire path again. The *Exit* flow is symmetric, with p releasing each lock along the path back to the leaf, and updating *current_node*$[p]$ after each lock release (lines 31–39). In both entry and exit flows, p always execute node lock's *Recover* procedure before entering that lock's Try or Exit section. This allows p to behave correctly after crash recovery: on its way up (respectively, down) it will not execute *Enter* (respectively, *Exit*) on the same node lock twice (lines 13–14, respectively lines 34–35).

To support aborts, process p checks the abort signal after acquiring each node lock (lines 15-18). If an abort was signalled, p starts executing the main lock's exit code to descend from the current node back to its leaf, releasing the node locks it holds along the way. (Similarly to the W-port algorithm, an aborting process execute *Exit* as a subroutine; it does not formally enter the main lock's exit section). The algorithm correctly supports aborts because if an abort is signalled while p is in some node lock's Try execution, it is guaranteed to complete in a finite number of its own steps. Subsequently, it will execute the main lock's abort handling code in a constant number of its own steps.

5 Adaptive Transformation

We now present our generic adaptivity transformation, which transforms any abortable RME algorithm L whose RMR complexity depends only on N into an abortable RME algorithm whose RMR complexity also depends on the *point contention* [1, 4], K, which is the number of processes executing the algorithm concurrently with the process going through the super-passage. We show how to transform an abortable RME algorithm with passage complexity $B < W$, super-passage complexity B^*, and space complexity S, into an abortable RME algorithm with passage complexity $O(\min(K, B))$, super-passage complexity $O(K + F)$ if $K < B$ or $O(B^* + F)$ otherwise, and space complexity $O(S + N + B^2)$.

The transformation is essentially a fast-path/slow-path construction, where the fast path is our W-port abortable RME algorithm and the slow path is the original lock L. A process p attempts to capture port $k = 0, \ldots, W - 1$ so it can use it in the fast path lock. Each such capture attempt is performed with CAS, and hence incurs an RMR. The idea is that if p fails to capture a port, then another process q succeeds. Therefore, if p fails to capture any port, the point contention is $> W$. In this case, p gives up and enters the slow path. The fast path and slow paths are synchronized with a 2-port abortable RME lock, again implemented with our lock (§ 3).

We present detailed pseudo code and prove all of the algorithm's properties. Due to space limits, omitted proofs appear in the full version [22, Appendix D].

```
void Try(int pid) {
    if STATUS[pid] = ABORT:
        Exit(pid, TRUE)
        return FALSE
    k := CURR_K[pid]
    while k < B:
        if K_OWNERS[k] = pid
            or CAS(K_OWNERS[k], $\bot$, pid):
            PATH[pid] := FAST
            if fast_path.Recover(k) = TRY
                if fast_path.Try(k) = FALSE:
                    STATUS[pid] := ABORT
                    Exit(pid, TRUE)
                    return FALSE
            break loop
        k := k + 1
        CURR_K[pid] := k
    if PATH[pid] $\neq$ FAST:
        PATH[pid] := SLOW
        if slow_path.Recover(pid) = TRY:
            if slow_path.Try(pid) = FALSE:
                STATUS[pid] := ABORT
                Exit(pid, TRUE)
                return FALSE
    if PATH[pid] = FAST:
        SIDE[pid] := RIGHT
    else // PATH[pid] = SLOW
        SIDE[pid] := LEFT
    if 2_rme.Recover(SIDE[pid]) = TRY:
        if 2_rme.Try(SIDE[pid]) = FALSE:
            STATUS[pid] := ABORT
            Exit(pid, TRUE)
            return FALSE
    STATUS[pid] := CS
}
```

```
STATUS: array of $N$ status words // initially all TRY
SIDE: array of $N$ SIDE words // initially all $\bot$
K_OWNERS: array of $B$ pids // initially all $\bot$
CURR_K: array of $N$ integers // initially all 0

void Exit(int pid, bool aborting) {
    if aborting = FALSE:
        STATUS[pid] := EXIT
    if SIDE[pid] $\neq \bot$ and
        2_rme.Recover(SIDE[pid]) $\neq$ TRY
            2_rme.exit(SIDE[pid], FALSE)
    SIDE[pid] := $\bot$

    if PATH[pid] = FAST:
        k := CURR_K[pid]
        if K_OWNERS[k] = pid and
            fast_path.Recover(k) $\neq$ TRY:
                fast_path.Exit(k, FALSE)
        K_OWNERS[k] := $\bot$
    else if PATH[pid] = SLOW:
        if slow_path.Recover(p) $\neq$ TRY:
            slow_path.Exit(p, FALSE)
    PATH[pid] := $\bot$
    CURR_K[pid] := 0
    STATUS[pid] := TRY
}

status Recover(int pid) {
    if STATUS[pid] = EXIT:
        return EXIT
    if STATUS[pid] = CS:
        return CS
    return TRY
}
```

Figure 3 Adaptive Transformation.

5.1 Algorithm Walk-Through

Figure 3 presents the transformed algorithm's pseudo code. The transformed algorithm uses three auxiliary abortable RME locks: a *slow_path* lock, which is an N-process base lock being transformed into an adaptive lock, and $fast\_path$ as well as 2_rme locks, both of which are instances of our D-port abortable RME (§ 3). The $fast\_path$ instance uses $D = B$ and the 2_rme instance uses $D = 2$.

The algorithm maintains K_OWNERS, an array of B words (initially all $\bot$) through which processes in the entry section try to capture ports to use in the fast-path lock (lines 5–17). Each process maintains a $CURR\_K$ variable to store the next port the process attempts to capture, or its captured port (once it captures one). To capture a port, process p scans K_OWNERS, using CAS at each slot k in an attempt to capture port k. If p captures port k, it enters the fast-path lock using that port. Overall, if p reaches slot k in K_OWNERS, then k other processes have captured ports $0, \ldots, k-1$. If p reaches the end of K_OWNERS and fails to capture a port, it enters the slow-path lock (lines 18–24). Regardless of which lock p ultimately enters, it invokes that lock's *Recover* method first, to correctly handle the case in which p is recovering from a crash.

We use the 2-RME lock to ensure mutual exclusion between the owners of the fast-path and the slow-path. Once p acquires its lock, it enters the 2-RME lock from the right (respectively, left) if it is on the fast-path (respectively, slow-path). In the 2-RME lock, p takes on a unique right/left id, corresponding to its direction of entry. Once p acquires the 2-RME lock, it enters the CS (lines 29–33).

In the exit section, p releases the 2-RME lock (lines 44–46) and then the fast-path or slow-path lock, as appropriate (lines 49–57). After releasing the fast-path lock, p releases its port (line 54) . These steps are done carefully to avoid having p return to the fast-path lock after crashing with the same port that is now being used by another process.

To handle aborts, if p receives a FALSE return value from some *Enter* execution, it executes the transformed lock's exit code (which, as a byproduct, releases p's port if it has one). Subsequently, p completes the abort.

6 Putting It All Together & Conclusion

Let T be the RME algorithm obtained by instantiating our tournament tree (§ 4) with our W-port abortable RME algorithm (§ 3). Then T's RMR passage complexity is $O(\log_W N) < W$, super-passage complexity is $O(\log_W N + F)$ and space complexity is $O(NW \log_W N)$. We can therefore apply the transformation of § 5 to T, obtaining our main result:

▶ **Theorem 4.** *There exists an abortable RME with $O(\min(K, \log_W N))$ RMR passage complexity, $O(F + \min(K, \log_W N))$ RMR super-passage complexity, and $O(NW \log_W N)$ space complexity where K is the point contention, W is the memory word size, N is the number of processes, and F is the number of crashes in a super-passage.*

Many questions about ME properties in the context of RME remain open, and we are far from understanding how the demand for recoverability affects the possibility of obtaining other desirable properties and their cost. Can the sublogarithmic RMR bounds be improved using only primitives supported in hardware, such as FAS and FAA? It is known that a weaker crash model facilitate better bounds [12], but is relaxing the crash model necessary? What, if any, is the connection between RME and abortable mutual exclusion? Both problems involve a similar concept, of a process "disappearing" from the algorithm, and for both problems, the best known RMR bounds (assuming standard primitives) are $O(\frac{\log N}{\log \log N})$. Can a formal connection between these problems be established?

References

1 Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-Lived Renaming Made Adaptive. In *PODC*, 1999.

2 A. Alon and A. Morrison. Deterministic Abortable Mutual Exclusion with Sublogarithmic Adaptive RMR Complexity. In *PODC*, 2018.

3 J.H. Anderson and Y.J. Kim. An Improved Lower Bound for the Time Complexity of Mutual Exclusion. *Distributed Computing*, 15(4), 2002.

4 H. Attiya. Adapting to Point Contention with Long-Lived Safe Agreement. In *SIROCCO*, 2006.

5 H. Attiya and A. Fouren. Algorithms Adapting to Point Contention. *JACM*, 50(4), 2003.

6 H. Attiya, D. Hendler, and P. Woelfel. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *STOC*, 2008.

7 D.Y.C. Chan and P. Woelfel. Recoverable Mutual Exclusion with Constant Amortized RMR Complexity from Standard Primitives. In *PODC*, 2020.

8 T.S. Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap, 1993.

9 S. Dhoked and N. Mittal. An Adaptive Approach to Recoverable Mutual Exclusion. In *PODC*, 2020.

10 E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *CACM*, 8(9), 1965.

11 W. Golab and D. Hendler. Recoverable Mutual Exclusion in Sub-Logarithmic Time. In *PODC*, 2017.

12 W. Golab and D. Hendler. Recoverable Mutual Exclusion Under System-Wide Failures. In *PODC*, 2018.

13 W. Golab and A. Ramaraju. Recoverable Mutual Exclusion: [Extended Abstract]. In *PODC*, 2016.

14 W. Golab and A. Ramaraju. Recoverable Mutual Exclusion. *Distributed Computing*, 32(6), 2019.

15 P. Jayanti. F-Arrays: Implementation and Applications. In *PODC*, 2002.

16 P. Jayanti. Adaptive and Efficient Abortable Mutual Exclusion. In *PODC*, 2003.

17 P. Jayanti and S. Jayanti. Constant Amortized RMR Abortable Mutex for CC and DSM. In *PODC*, 2019.

18 P. Jayanti, S. Jayanti, and A. Joshi. A Recoverable Mutex Algorithm with Sub-Logarithmic RMR on Both CC and DSM. In *PODC*, 2019.

19 P. Jayanti, S.V. Jayanti, and A. Joshi. Optimal Recoverable Mutual Exclusion Using only FASAS. In *NETYS*, 2018.

20 P. Jayanti and A. Joshi. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *DISC*, 2017.

21 P. Jayanti and A. Joshi. Recoverable Mutual Exclusion with Abortability. In *NETYS*, 2019.

22 Daniel Katzan and Adam Morrison. Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. *CoRR*, 2020. `arXiv:2011.07622`.

23 J.M. Mellor-Crummey and M.L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *TOCS*, 9(1), 1991.

24 M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *TPDS*, 15(6), 2004.

25 M.L. Scott. Non-blocking Timeout in Scalable Queue-based Spin Locks. In *PODC*, 2002.

26 M.L. Scott and W.N. Scherer. Scalable Queue-based Spin Locks with Timeout. In *PPoPP*, 2001.

27 Gadi Taubenfeld. *Synchronization algorithms and concurrent programming.* Pearson / Prentice Hall, 2006.

28 J.H. Yang and J.H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1), 1995.