Modular Recoverable Mutual Exclusion Under System-Wide Failures

Sahil Dhoked
Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

Wojciech Golab
Department of Electrical and Computer Engineering, University of Waterloo, Canada

Neeraj Mittal
Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

Abstract

Recoverable mutual exclusion (RME) is a fault-tolerant variation of Dijkstra's classic mutual exclusion (ME) problem that allows processes to fail by crashing as long as they recover eventually. A growing body of literature on this topic, starting with the problem formulation by Golab and Ramaraju (PODC'16), examines the cost of solving the RME problem, which is quantified by counting the expensive shared memory operations called remote memory references (RMRs), under a variety of conditions. Published results show that the RMR complexity of RME algorithms, among other factors, depends crucially on the failure model used: individual process versus system-wide. Recent work by Golab and Hendler (PODC'18) also suggests that *explicit* failure detection can be helpful in attaining *constant* RMR solutions to the RME problem in the system-wide failure model. Follow-up work by Jayanti, Jayanti, and Joshi (SPAA'23) shows that such a solution exists even without employing a failure detector, albeit this solution uses a more complex algorithmic approach.

In this work, we dive deeper into the study of RMR-optimal RME algorithms for the system-wide failure model, and present contributions along multiple directions. First, we introduce the notion of *withdrawing* from a lock acquisition rather than resetting the lock. We use this notion to design a withdrawable RME algorithm with optimal $O(1)$ RMR complexity for both cache-coherent (CC) and distributed shared memory (DSM) models in a *modular* way without using an explicit failure detector. In some sense, our technique marries the simplicity of Golab and Hendler's algorithm with Jayanti, Jayanti and Joshi's weaker system model. Second, we present a variation of our algorithm that supports fully dynamic process participation (*i.e.*, both joining and leaving) in the CC model, while maintaining its constant RMR complexity. We show experimentally that our algorithm is substantially faster than Jayanti, Jayanti, and Joshi's algorithm despite having stronger correctness properties. Finally, we establish an impossibility result for fully dynamic RME algorithms with bounded RMR complexity in the DSM model that are adaptive with respect to space, and provide a wait-free withdraw section.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases mutual exclusion, shared memory, persistent memory, fault tolerance, system-wide failure, RMR complexity, dynamic joining, dynamic leaving

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.17

Funding *Wojciech Golab*: Researcher supported in part by an Ontario Early Researcher Award, a Google Faculty Research Award, as well as the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Acknowledgements We thank the anonymous reviewers for their helpful feedback and insightful suggestions.

1 Introduction

One of the most common techniques to mitigate race conditions in a concurrent system is to use *mutual exclusion (ME)*, which establishes a *critical section (CS)* in which a program can access a shared resource without risking interference from other processes. Correct use of mutual exclusion or *mutex locks* ensures that the system always stays in a consistent state, and produces correct outcomes. The ME problem was first defined by Dijkstra more than half a century ago in [16], later formalized by Lamport [35, 36], and then widely studied in scientific literature (*e.g.*, see [4, 41]) under a variety of assumptions regarding both the degree of synchrony and the set of synchronization primitives available for accessing shared memory. The vast majority of research in this area in recent years emphasizes so-called *local spin* algorithms, which incur a bounded number of *remote memory references* (RMRs) – expensive memory operations that trigger communication on the interconnect joining processors with memory – in each attempt to acquire and release the lock. Whether or not a memory operation incurs an RME depends on the underlying shared memory model – cache-coherent (CC) or distributed shared memory (DSM) (*e.g.*, see [38, 12, 20]).

Over decades of research, the study of shared memory algorithms has started to shift away from traditionally strong modelling assumptions, such as a failure-free execution environment where processes agree ahead of time on a set of named shared objects, toward richer models based on faulty shared memories [1, 40], anonymous systems [42, 47], and faulty processes [7, 8, 23, 39]. Alternative models are particularly interesting and important for the ME problem in asynchronous environments since solutions rely intrinsically on blocking synchronization, and failures can have undesirable and even disastrous ripple effects. For example, if one process crashes in the critical section, or even while acquiring or releasing a mutex lock, several other processes who are contending for the same lock can potentially stall. Although such failures are not common, they can occur in the real world due to software bugs, or even hardware failures in large scale platforms where processing elements are interconnected in complex ways.

The *recoverable mutual exclusion* (RME) problem, formulated recently by Golab and Ramaraju [23, 24], is the most recent attempt to marry resilience against process failures with mathematical rigour in the ongoing study of lock-based synchronization. The RME problem involves designing an algorithm that ensures mutual exclusion under the assumption that processes may fail at *any* point during their execution, either independently or simultaneously. Since a slow process cannot be distinguished reliably from a crashed process in an asynchronous environment [11, 18], one expects informally that an RME algorithm must receive some help from the environment in responding to a failure. The primary mechanism for this in Golab and Ramaraju's model is the crucial assumption that a crashed process[1] eventually recovers and cleans up the internal state of the RME lock by attempting to acquire and release it again, which can be regarded as an implicit form of failure detection. Golab and Hendler's work on system-wide failures [22] adds an explicit epoch-based failure detector that helps synchronize a cohort of recovering processes, which simplifies the RME problem to the point where it can be solved for n processes using common synchronization primitives with (*optimal*) $O(1)$ RMR complexity for both CC and DSM models.

The growing body of work on the RME problem focuses primarily on fundamental correctness properties of RME and techniques for implementing these properties, with little attention paid to how RME locks could be used in practice to implement fault-tolerant data structures. Results are especially sparse for RMR-efficient algorithms in the system-wide

[1] More precisely, a process that crashes outside of the lock's non-critical section.

Algorithm 1 Process execution model.

```
while true do
    Non-Critical Section (NCS)
    Recover();                          }
    Enter();                            }
    Critical Section (CS)               }   OR   { Withdraw();
    Exit();                             }
```

failure model, which is arguably the more practical failure model; in a well-designed software system, processes are more likely to fail together due to a power outage than individually due to software bugs. Somewhat surprisingly, only two studies have been published in this space so far, namely work by Golab and Hendler (GH) [22], and Jayanti, Jayanti and Joshi (JJJ) [28, 29]. Both works follow roughly the same algorithmic technique, which starts with a conventional mutex algorithm (base mutex), and applies a transformation to reset the base mutex carefully after a system-wide failure. GH is a black box technique that uses a single instance of the base mutex and requires an explicit failure detector (as explained earlier), whereas JJJ avoids the failure detector but uses a somewhat complex arrangement of three instances of a base mutex (with wait-free exit) and provides a weaker fairness guarantee.

In this work, we advance the state of the art with respect to RME locks for system-wide failures along several directions. First, we introduce a new algorithmic technique called *withdrawing* whereby a process can remove itself from a queue of waiting processes upon failure without resetting the entire queue to its initial state. This technique allows us to combine the simplicity of the GH algorithm with the weaker assumptions of the JJJ model. We demonstrate the power of our approach by constructing a novel RME lock with optimal $O(1)$ RMR complexity for both CC and DSM models using the well-known Mellor-Crummey and Scott's (MCS) mutex lock [38, 17] as the building block. Our algorithm requires only one instance of this base mutex and provides first-come-first-served (FCFS) fairness, like GH, and yet does not rely on a failure detector, similar to JJJ. Furthermore, we make the argument that the ability to withdraw is a useful feature for an RME lock to not only use internally, but also expose to the application, and formulate the *withdrawable recoverable mutual exclusion* problem. In particular, we show in Appendix A that withdrawability alone can be sufficient for an application's recovery goals using the example of a lock-based concurrent linked-list presented in [25]. We also advocate for *modular* algorithmic designs that separate optional features like Golab and Ramaraju's critical section re-entry (CSR) property from the core functions of an RME lock. Second, we present a variation of our RME algorithm for the CC model that supports fully dynamic process participation (*i.e.*, support for both joining and leaving) while maintaining its most desirable features. Our algorithmic approach not only allows for a separation of concerns but, as we show experimentally, also gives us a substantial performance advantage over JJJ, which relies crucially on critical section ownership state to achieve correct recovery. Finally, we establish an impossibility result for fully dynamic RME algorithms that are RMR-efficient in the DSM model, adaptive with respect to space, and provide a wait-free withdraw section.

Roadmap. The rest of the text is organized as follows. We present our system model and formulate the withdrawable RME problem in Section 2. We present an RMR-optimal RME lock with a wait-free withdraw section assuming system-wide failures for both CC and DSM models in Section 3, a transformation to add the CSR property in Section 4, and a fully dynamic variant for the CC model in Section 5. Section 6 describes an impossibility result about designing a fully dynamic RME algorithm with a wait-free withdraw section for

the DSM model under certain conditions. We discuss related work in Section 7. Section 8 summarizes our conclusions and outlines directions for future research. Discussion of a use case for withdrawable locks, detailed experiments, and an equivalence result for withdrawability and abortability are presented in Appendix A, B, and C, respectively.

2 System Model

Our model is based on [22, 23, 24]. We consider an asynchronous shared memory system in which processes communicate by performing single-word read, write and read-modify-write (RMW) operations on shared variables, and also have access to private variables. Our algorithms use two RMW primitives: (i) Fetch-And-Store (`FAS`), which retrieves the old value and blindly writes a new value, and (ii) Compare-And-Swap (`CAS`), which conditionally writes a new value if the old value matches a given comparison value, and returns a boolean success indicator.

We assume the *crash-recover* failure model, meaning that a process may fail at any time during its execution by crashing, and may recover by resuming execution from the beginning of its program. Failures are *system-wide* [22], meaning that all processes crash simultaneously, as opposed to the individual failure model considered in [23, 24, 15]. Upon crashing, a process loses its call stack, and its private variables (including the program counter) are reset to their initial values; however shared variables are stored in persistent memory and retain their most recently written value prior to the crash.

The execution model of a process with respect to a recoverable lock is depicted in algorithm 1. A process typically executes the NCS (non-critical section), `Recover`, `Enter`, CS (critical section) and `Exit` sections, in that order, and then returns to the NCS. The internal structure of the lock is cleaned up, if needed, inside the `Recover` section, then the `Enter` section is used to acquire the lock, and the `Exit` section releases it. A process can also execute the NCS and `Withdraw` sections, which bypasses the CS and yet allows the internal structure of the lock to be cleaned up after a failure if needed inside the `Withdraw` section. A system-wide failure returns every process to the NCS by resetting its program counter to the initial value.

The `Withdraw` section is a *new feature* in our model, as compared to [23, 24], that provides the application more flexibility in using the recoverable lock. This becomes especially useful when an RME lock is used as a component to build another more advanced RME lock satisfying additional desirable properties, as illustrated in this work; the specific execution path taken by a process depends on the needs of the program using the lock. A trivial way to implement `Withdraw` section is to simply execute the `Recover`, `Enter` and `Exit` sections in order, with an empty CS (*e.g.*, as on Line 50 in Figure 6 of [21]). However, this naïve implementation is inherently blocking, and can cause deadlock in the application if not used carefully. On the other hand, as we show in this work, it is possible to implement the `Withdraw` section efficiently in a *wait-free* manner (*i.e.,* bounded number of steps).

A system execution is modeled as a sequence of process steps called a *history*. In each step, a process performs some local computation affecting only its private variables, and executes at most one shared memory operation. A process p is said to be *live* in a history H if it leaves the NCS at least once, meaning that H contains at least one step by p. We also consider crash steps that represent system-wide failures and do not belong to any process, but we do not model executions of the NCS and CS as steps. The projection of a history H onto a process p, denoted $H|p$, is the maximal subsequence of H comprising all steps of p as well as crash steps. The concatenation of histories G and H is denoted $G \circ H$. An *epoch* is a contiguous subhistory of a history H between (and excluding) two consecutive crash steps.

A *c-passage* of a process p is a sequence of steps by p from the first step of the `Recover` section to the last step of the `Exit` section, or a crash failure, whichever occurs first. A *w-passage* of a process p is a sequence of steps by p from the first step of the `Withdraw` section to the last step of the `Withdraw` section, or a crash failure, whichever occurs first. A *passage* is either a c-passage or a w-passage. A passage is *failure-free* unless a process crashes before completing the last step of the `Exit` or `Withdraw` section. A *super-passage* of a process p is a maximal non-empty sequence of consecutive passages executed by p, where only the last passage in the sequence can be failure-free. A history H is *fair* if it is finite, or it is infinite and every process that is live in H either takes infinitely many steps or stops taking steps after completing a failure-free passage and returning to the NCS.

Two passages *interfere* if their respective super-passages overlap (*i.e.*, neither ends before the other starts). A passage by a process p is *0-failure-concurrent* (0-FC) if p crashes in the respective super-passage. A passage is *k-failure-concurrent* (k-FC), $k > 0$, if it interferes with some $(k-1)$-FC passage (possibly itself). Clearly, k-FC implies $(k+1)$-FC for all $k \geq 0$. Intuitively, the parameter k in the notion of failure-concurrency measures the maximum "distance" of a given passage from any failure, where two interfering passages are said to be at a "distance" of one from each other.

Unless otherwise stated, we allow *dynamic* participation of processes. A *new* process can join the system at run time and use the lock to execute a CS, referred to as *dynamic joining* [28, 29]. As part of joining, a process may need to allocate some memory – shared as well as private – needed to synchronize with other processes. We also consider the dual problem in which any existing process can leave the system at run time after completing a failure-free passage, referred to as *dynamic leaving*. In this case, the departing process needs to know that it can safely reclaim its memory (unless a separate garbage collection mechanism is being used). We say that an algorithm is *fully dynamic* if it supports dynamic joining as well as dynamic leaving. The original MCS algorithm (without wait-free exit) is fully dynamic. The RME algorithms in [28, 29] only support dynamic joining but not dynamic leaving.

2.1 RME Correctness and Other Properties Reformulated

This section summarizes the correctness properties of RME, reformulated as needed to accommodate `Withdraw` section and w-passage. New additions are typeset in *italics*.

Mutual Exclusion (ME) At most one process is in the CS at any point in any history.

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-passages. If a process p leaves the NCS in some step of H, then p eventually either enters the CS *or begins executing* `Withdraw` *section.*

If a process fails inside the CS, then a shared resource (*e.g.*, a data structure) may be left in an inconsistent state. In such cases, it may be desirable to allow the same process to re-enter CS and "fix" the shared resource, if needed, before any other process can enter the CS (*e.g.*, [24, 21, 22, 27]). This property is referred to as *critical section re-entry (CSR)*. We use a stronger variant of CSR in this work, which is defined as:

Bounded Critical Section Reentry (BCSR) For any history H, if a process p crashes inside the CS, then, until p has reentered the CS *or begun executing* `Withdraw`, any subsequent execution of `Recover` and `Enter` sections by p either completes within a bounded number of p's own steps or ends with p crashing.

In addition to the above qualitative properties, it is also desirable for an RME algorithm to satisfy the following:

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

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

Note that the `Withdraw` section can in some cases serve an application's recovery goals better than Golab and Ramaraju's CSR property [23, 24], which we reformulated earlier. As a specific example, consider the linked list in Chapter 9 of Herlihy and Shavit [25], which we reproduce in Appendix A along with a discussion of recoverability. The `Add` and `Remove` methods both use hand-over-hand locking to traverse the list, and both operations take effect atomically at a single line of code that writes a pointer (Line 163 and Line 178, respectively). Since garbage collection is used to reclaim list nodes, strict linearizability [2, 6] can be achieved easily by simply repairing the locks (via either `Recover`/`Enter`/`Exit` or `Withdraw`). Re-entering the CS in this case is unnecessary since the linked list structure itself cannot be corrupted by a crash failure, not to mention that the application would need to determine specifically which linked list operation was interrupted by a failure to reach the correct CS as each node-level lock protects multiple critical sections.

Many applications require a lock to provide some guarantees about fairness. Intuitively, a fairness property imposes a constraint on when and/or how often a process trying to enter the CS can be overtaken by another process. Definitions of such properties refer to a *doorway*, which is a *bounded* prefix of the `Enter` section, and intuitively determines the order in which processes acquire the lock. We consider the following two notions of fairness, the first of which has novel formulation for the system-wide failure model:

First-Come-First-Served (FCFS) For any two *concurrent* c-passages π and π' belonging to processes p and p', respectively, if p completes its doorway in π before p' starts its doorway in π', then p' does not enter the CS in π' before p enters the CS in π.

k-First-Come-First-Served (k-FCFS), where $k \geq 0$ For any two c-passagees π and π' belonging to processes p and p', respectively, where neither passage is k-FC, if p completes its doorway in π before p' starts its doorway in π', then p' does not enter the CS in π' before p enters the CS in π.

Intuitively, FCFS imposes constraints on all c-passages, whereas k-FCFS imposes constraints only on those c-passages that are sufficiently far away from any failure. Note that FCFS implies 0-FCFS and k-FCFS implies $(k+1)$-FCFS for all $k \geq 0$. Further, FCFS and (B)CSR are mutually incompatible properties. An RME algorithm can satisfy at most one of these properties; but it can simultaneously satisfy k-FCFS, for some k, as well as BCSR.

2.2 Complexity Measures

In terms of complexity measures, we are concerned in this work with time and space. Time complexity is quantified by counting *remote memory references* (RMRs), which are defined in an architecture-dependent manner. In the *cache-coherent* (CC) model, we conservatively count every shared memory operation as an RMR, except where a process p reads a variable that p has already read earlier, and which has not been updated (*i.e.*, accessed by any means other than a read) since p's most recent read. In the *distributed shared memory* (DSM)

Algorithm 2 MCS algorithm with wait-free exit (adapted from [17, 15]).

```
1  struct QNode {
2      locked: boolean variable, initially FALSE;
3      next: reference to QNode, initially null;
4  };
5  global shared variables
6      Tail: reference to QNode, initially null;
7  per-process shared/persistent variables
8      Pool_p: array [0,1] of QNode, all elements
               initially {FALSE, null};
9      Mine_p: reference to QNode, initially &Pool[0];
10     Curr_p: integer variable ∈ {0,1}, initially 0;
11 private variables
12     pred: reference to QNode;
13 Procedure Enter()
14     Mine_p := &Pool_p[Curr_p];
15     Mine_p.locked := TRUE;
16     Mine_p.next := null;
17     pred := FAS(Tail, Mine_p);
18     if pred = null then return;
19     if CAS(pred.next, null, Mine_p) then
20         await ¬Mine_p.locked;
21 Procedure Exit()
22     if CAS(Tail, Mine_p, null) then return;
23     CAS(Mine_p.next, null, Mine_p);
24     if Mine_p.next ≠ Mine_p then
25         Mine_p.next.locked := FALSE;
26     Curr_p := 1 − Curr_p;
```

model, each shared variable is statically allocated to a memory module that is local to exactly one process and remote to all others. Space complexity is simply the number of memory words used per process.

3 An RMR-Optimal RME Algorithm with Dynamic Joining for CC and DSM Models

In this section, we present an RME algorithm for system-wide failures that satisfies the ME and SF correctness properties, and has $O(1)$ RMR complexity in both CC and DSM models. Later, we describe a separate transformation to add the BCSR property to any RME lock.

3.1 Background: MCS Algorithm with Wait-Free Exit

The MCS algorithm is a queue-based ME algorithm that has optimal $O(1)$ RMR complexity in both the CC and DSM models. It maintains a (single) queue of all outstanding requests for critical section; requests are satisfied in the order in which they are inserted into the queue. Pseudocode for the algorithm is given in algorithm 2, and has been modified from the original version [38] to also satisfy the BE property as described in [17].

A request is represented using a `QNode`, which consists of two fields: (i) a boolean variable, *locked*, to indicate whether the node is currently locked and thus its owner does not have permission to enter its critical section, and (ii) a reference to `QNode`, *next*, to store the address of the successor node.

To enter its critical section, a process first appends its node to the queue (Line 17). If the queue was not empty (implying that it has a predecessor), it tries to create a forward link from its predecessor's node to its own node (Line 19). If successful, it waits for its node to be unlocked by its predecessor (Line 20). If it either did not have a predecessor (Line 18) or failed to create the link (Line 19), then it implies that the process holds the lock; in this case the process simply returns.

When leaving its critical section, a process first attempts to remove its node from the queue (Line 22). The attempt will fail if another process has already appended its node to the queue. In that case, the process notifies its successor that the critical section is now empty by either writing a special value to the next field of its node (if the link has not been created yet) (Line 23) or unlocking its successor's node (if the link has already been created) (Line 25).

In the original MCS algorithm, a process can reuse the same node immediately after completing its exit section for its next CS request. However, in the wait-free exit version, immediate reuse may create a deadlock. Using a pool of two nodes and alternating between them [17] solves this problem (Line 26). Intuitively, once a process is "enabled" to enter its critical section, then it can be inferred indirectly that the node it used for its previous request has served its purpose and no process would access its fields anymore; thus, it can be reclaimed for its next request. Similar techniques have been used to achieve the wait-free exit property in other queue-based locks [12, 43, 37, 26], but our particular RME algorithm is derived from the queue-based ME lock described in [17].

3.2 The Main Idea

We modify the augmented MCS algorithm described in Section 3.1 to obtain an RMR-optimal RME algorithm under the system-wide failure model that satisfies the ME and SF but not the BCSR property. Assume, for now, that processes do not reuse queue nodes. The key idea is that, if a process crashes while executing its passage, it terminates or aborts its interrupted attempt to use the lock (for executing its critical section), and initiates a new attempt to use the lock using a fresh node. A process aborts its attempt by basically executing steps of the exit section and unlocking its successor node, if any. Note that a process aborts its attempt even if it crashed in its critical section. Intuitively, this "clears" the queue of any "old" nodes, which is then "repopulated" using "new" nodes.

Note that, in the (augmented) MCS algorithm used for solving the ME problem, queue nodes are unlocked in a serial manner in the same order in which they were appended to the queue. However, in the RME variant, queue nodes associated with aborted attempts may be unlocked out of order and even concurrently. This out-of-order unlocking of queue nodes associated with aborted attempts does not violate the ME property because *none* of these queue nodes, which were appended to the queue before the failure, can be used by its owner now to execute its critical section.

However, the above approach interferes with the node reuse mechanism used in the augmented MCS algorithm. A slow process p may erroneously unlock a reused node owned by a fast process q because q's node was the successor of p's node in an earlier epoch. To address this problem, we replace the *locked* field in a queue node with a field that stores the address of the predecessor node. A process unlocks its successor node by replacing the contents of this field in its successor's node with a **null** value using a `CAS` instruction, which will succeed only if the (successor) node has not been reused.

3.3 A Formal Description

Pseudocode of the RMR-optimal RME algorithm for system-wide failures is presented in algorithm 3. To avoid repetition, we only describe the differences between algorithm 2 (augmented MCS) and algorithm 3.

As explained earlier, the *locked* field in `QNode` (Line 2) has been replaced with the *pred* field (Line 28); a process stores the address of its predecessor node in *pred* (Line 55). The steps of the `Exit` section in algorithm 2 have been abstracted into a `Cleanup` method in algorithm 3, which is then invoked from the `Exit`, `Recover` and `Withdraw` sections. A process unlocks its successor node by clearing the *pred* field of the (successor) node using a `CAS` instruction (Line 50). At the beginning of the `Cleanup` method, a process also deletes the link from its predecessor node to its own node (Lines 45–46). This step is required to prevent too many "older" predecessors from performing a `CAS` instruction on the *pred* field of its node. Although these `CAS` instructions will fail, they may still cause the process to incur an RMR while spinning on *pred* field in the CC model.

Algorithm 3 An RMR-optimal RME algorithm for tolerating system-wide failures that satisfies the ME and SF properties.

```
27 struct QNode {
28     pred: reference to QNode, initially null;
29     next: reference to QNode, initially null;
30 };
31 global shared variables
32     Tail: reference to QNode, initially null;
33 per-process shared/persistent variables
34     Pool_p: array [0,1] of QNode, all elements
               initially {null, null};
35     Mine_p: reference to QNode, initially &Pool[0];
36     Curr_p: integer variable ∈ {0,1}, initially 0;
37 private variables
38     pred: reference to QNode;
39 Procedure Withdraw()
40     Cleanup();
41 Procedure Recover()
42     Cleanup();
43     Curr_p := 1 − Curr_p;
44 Procedure Cleanup()
45     if Mine_p.pred ≠ null then
46         CAS(Mine_p.pred.next, Mine_p, null);
47     CAS(Tail, Mine_p, null);
48     CAS(Mine_p.next, null, Mine_p);
49     if Mine_p.next ≠ Mine_p then
50         CAS(Mine_p.next.pred, Mine_p, null);
51 Procedure Enter()
52     Mine_p := &Pool_p[Curr_p];
53     Mine_p.pred := null;
54     Mine_p.next := null;
55     pred := FAS(Tail, Mine_p);
56     Mine_p.pred := pred;
57     if Mine_p.pred = null then return;
58     if CAS(Mine_p.pred.next, null, Mine_p) then
59         await Mine_p.pred = null;
60 Procedure Exit()
61     Cleanup();
```

Recall that, in the ME algorithm described in algorithm 2, a process alternates between two nodes. The same idea works in the recoverable version, except that we perform the node switch at the end of the `Recover` section.

We refer to the algorithm described in algorithm 3 as MCS-SW (SW stands for system-wide). The doorway of MCS-SW consists of Lines 52–55. We have:

▶ **Theorem 1.** *MCS-SW satisfies the ME, SF, BE, BR, BW and FCFS properties of the RME problem. It has $O(1)$ RMR complexity in both CC and DSM models, as well as $O(1)$ space complexity per process. Finally, it uses bounded variables and supports dynamic joining.*

4 Adding the BCSR Property

To our knowledge, two RMR-preserving transformations have been proposed to add the BCSR property to an RME lock that only satisfies the ME and SF properties. The first transformation, given by Golab and Ramaraju [23, 24], assumes the individual failure model. The second transformation, given by Golab and Hendler [22], assumes the system-wide failure model. Neither transformation can be applied to the RME lock described in Section 3.

The first transformation works under the assumption that a process cannot detect the failure of another process. This implies that, if a process p has entered its CS during a passage, then no other process can gain entry into its CS even if p fails during its CS until p has started its exit section, possibly in a future passage. While this assumption holds for any RME lock designed for independent failure model without explicit failure detection, it may not hold for an RME lock designed for the system-wide failure model (with or without explicit failure detection). Specifically, the RME lock in Section 3 exploits the property that one's own failure also implies the failure of every other process in the system. So, a process p can gain entry into its CS after another process q fails while executing its CS without requiring q to start its exit section. The second transformation, on the other hand, requires an explicit epoch-based failure detector.

Note that the transformation for the CC model is relatively straightforward and involves spinning on a global variable. However, the one for the DSM model is non-trivial because spinning on a global variable may incur unbounded RMRs in the worst case. In this section,

we present a new transformation that can be applied to any RME lock for system-wide failures without using an explicit failure detector, and incurs only $O(1)$ RMRs in both CC and DSM models. Additionally, it preserves all properties of MCS-SW except for fairness which now weakens to 0-FCFS, a direct consequence of CSR.

4.1 The Main Idea

The RMR-optimal RME algorithm described in Section 3 does not satisfy the BCSR property. This is because, when processes append new nodes to the queue after aborting their interrupted attempts, these new nodes may be appended to the queue in a different order. As a result, a process that crashes while executing its critical section may need to wait for one or more processes to complete their critical sections before it can reenter its own because nodes of other processes *now precede* its own node in the queue.

To add the BCSR property, we maintain a global variable that stores the identifier of the process currently in CS. A process writes its identifier to the variable when it gains entry into the CS and resets it upon leaving the CS. If a process crashes during its CS, then, upon starting a new c-passage, it can return from `Recover` and `Enter` methods of the target lock immediately if the variable contains its own identifier. We refer to this path to enter the CS as *legacy* path. Otherwise, it first acquires the base lock and then possibly waits for the process with legacy admission into the CS to leave, if applicable. We refer to this path to enter the CS as *regular* path.

Jayanti, Jayanti and Joshi define a *capturable* object that can be used to synchronize access to the CS between processes taking the two paths, while incurring only $O(1)$ RMRs in both CC and DSM models [29]. However, their capturable object uses an *unbounded* sequence number as well as a *read-modify-write* instruction. We use a similar, but simpler, approach that avoids the two limitations. Our approach exploits the fact that any legacy admission can only happen at the beginning of an epoch (period between two consecutive system-wide failures), after which all admissions are regular.

The main idea is that a process that takes the regular path, say p, uses its own memory location (basically a boolean variable) to spin and stores the address of its spin location in another global variable before spinning (provided that the CS is already occupied). The process leaving the CS is responsible for signalling p by writing to the location provided by p.

4.2 A Formal Description

Pseudocode of the transformation to attain the BCSR property is presented in algorithm 4. We refer to the transformation as CSR-SW.

The algorithm uses the following shared variables: (i) base RME lock, BLock, that satisfies the ME and SF properties (*e.g.*, the lock in algorithm 3), (ii) integer, CSOwner, to store the identifier of the process that currently owns the CS, (iii) location, CSWait, to store the address of the boolean variable on which a process will spin, (iv) per process boolean variable, Locked, for spinning until signalled, and (v) per process variable, *skipRE*, to track whether a process skipped acquiring the base RME lock in this current passage.

In the `Recover` method, a process first checks if it already owns the CS (Line 71). If yes, it makes a note of it (Line 72) and completes the super-passage of the base lock by invoking its `Withdraw` method (Line 73) (legacy path). Otherwise, it starts a new passage of the base lock by invoking its `Recover` method (Line 76) (regular path).

Algorithm 4 An RMR-optimal transformation to achieve the BCSR property.

```
62 global shared variables
63   BLock: instance of recoverable (base) lock;
64   CSOwner: process identifier, initially ⊥;
65   CSWait: reference to a boolean variable,
           initially null;
66 per-process shared/persistent variables
67   Locked_p: boolean variable, initially null;
68 private variables
69   skipRE: boolean variable;
70 Procedure Recover()
71   if CSOwner = p then
72     skipRE := TRUE;
73     BLock.Withdraw();
74   else
75     skipRE := FALSE;
76     BLock.Recover();
77 Procedure Enter()
78   if CSOwner = p then return;
79   BLock.Enter();
80   Locked_p := TRUE;
81   CSWait := &Locked_p;
82   if CSOwner = ⊥ then Locked_p := FALSE;
83   await not(Locked_p);
84   CSOwner := p;
85 Procedure Exit()
86   CSOwner := ⊥;
87   if CSWait ≠ null then *CSWait := FALSE;
88   if ¬skipRE then BLock.Exit();
89 Procedure Withdraw()
90   if CSOwner = p then
91     CSOwner := ⊥;
92     if CSWait ≠ null then *CSWait := FALSE;
93   BLock.Withdraw();
```

In the `Enter` method, if a process already owns the CS, it returns immediately (Line 78) (legacy path). Otherwise, it acquires the base lock (Line 79). It next initializes its spin location (Line 80), announces the address of its spin location (Line 81) and busy-waits if the CS is already occupied (Lines 82 and 83). Finally, it claims the ownership of the CS (Line 84) (regular path).

In the `Exit` method, a process releases the ownership of the CS (Line 86) and signals the waiting process if any (Line 87). If it took the regular path, it completes the super-passage of the base lock by invoking its `Exit` method (Line 88).

In the `Withdraw` method, if a process currently owns the CS, it releases its ownership (Line 91) and signals the waiting process if any (Line 92). It then invokes the `Withdraw` method of the base lock (Line 93).

The doorway of the target lock consists of Line 78 along with the doorway of the base lock if acquired. We have:

▶ **Theorem 2.** *CSR-SW preserves the ME, SF, BR, BE, BW and k-FCFS for $k \geq 0$ properties of the base lock, and adds the BCSR property to the target lock. It uses bounded variables and supports dynamic joining. Finally, it preserves the RMR and space complexities of the base lock.*

Intuitively, the ME property holds for the following reasons. First, regular admissions to the CS are serialized using the base lock. Second, there is at most one legacy admission to the CS during an epoch. Third, a regular admission can only occur if the process with legacy admission has vacated the CS.

Intuitively, the SF property holds for the following reasons. First, the base lock satisfies the SF property. Second, only a process that takes the regular path, after acquiring the base lock, needs to busy-wait for the CS to become empty due to legacy admission. If such a process, say p, finds CSOwner to have a non-bottom value implying that another process, say q, is currently in the CS, then p would have already written the address of its spin location to CSWait. Clearly, upon leaving CS, q is guaranteed to find the address of p's spin location and signal p to quit its busy-wait loop.

▶ **Corollary 3.** *The target lock obtained by applying CSR-SW to MCS-SW satisfies the ME, SF, BCSR, BE, BR, BW and 0-FCFS properties. It uses bounded variables and supports dynamic joining. Finally, it has $O(1)$ RMR complexity in both CC and DSM models, and uses $O(1)$ space per process.*

5 A Fully Dynamic RMR-Optimal RME Algorithm for the CC Model

The RME algorithms in [29] as well as those presented in Sections 3 and 4 support dynamic joining. While a process can leave the system if it has no super-passage in progress, it cannot reclaim its memory since other processes may still dereference one of its locations (*e.g.*, the *next* field of its queue node). A separate garbage collection mechanism is required to identify nodes whose memory can be safely reclaimed. We describe an RMR-optimal RME algorithm for the CC model that allows a process to not only join the system at any time but also leave at any time and *safely deallocate its memory when leaving*. The algorithm, however, uses an unbounded sequence number.

Our RME algorithm is derived from Lee's ME lock [37, 29], which is specifically designed for the CC model, and uses the idea described in Section 3. Like the MCS lock, Lee's lock is also queue-based and requests are satisfied in the order in which they are appended to the queue. However, the queue is implicit and a queue node does not store *next* pointers. A process instead spins on a memory location in its predecessor's node until it is signalled by the predecessor. Further, unlike in the MCS lock, a process does not attempt to remove its node from the queue.

Algorithm 5 describes the pseudocode of the RME algorithm with support for dynamic leaving, based on Lee's ME lock. We refer to the algorithm as LeeDL-SW. Note that the original Lee's lock consists of Lines 118–121 for acquiring the lock and Line 112 for releasing the lock. To acquire the lock, a process switches its queue node (Line 118), initializes it (Line 119), appends it to the queue (Line 120) and waits until signalled by its predecessor (Line 121). To release the lock, a process simply signals its successor (Line 112).

We now describe adding recoverability, as well as the *optional* critical section re-entry and dynamic leaving properties to Lee's lock, while preserving its RMR optimality. For convenience, the code lines used to achieve critical section re-entry and dynamic leaving properties are tagged with one and two "◀" symbols, respectively, at the end of each line.

Adding recoverability. We define a `Cleanup` method in which a process attempts to remove its node from the queue (Lines 110 and 111) and then signal its successor in case it has one (Line 112). The `Cleanup` method is invoked from the `Recover`, `Exit` and `Withdraw` methods.

Adding critical section re-entry. We use a shared variable to keep track of the process currently executing its CS (CSOwner). Once a process has been signalled by its predecessor (*i.e.*, it has acquired Lee's lock), the process has to wait until the CS has become empty (Line 122) and then claims the ownership of the CS (Line 123). A process releases the ownership of the CS as part of the `Cleanup` method (Line 109). If a process fails during its CS, it can immediately return from the `Recover` and `Enter` methods if it already owns the CS (Lines 114 and 117).

Adding dynamic leaving. A process can leave the system only if it has no super-passage in progress. Moreover, before leaving the system, the process must invoke the `SafeToReclaim` method. It then either waits for the method to complete or for a system-wide failure to occur. Once either of these two events has occurred, it can safely reclaim all its memory.

We use a sequence number to keep track of the *rough* number of times the CS has been occupied (CSBusy). A process increments this sequence number after obtaining the ownership of the CS (Line 124). Note that the count need not be exact due to failures. The purpose of the `SafeToReclaim` method is to ensure that the caller does not have a successor

Algorithm 5 An RMR-optimal RME lock for the CC model that supports dynamic joining *as well as dynamic leaving*. Code in red color is only required for the CSR property.

```
94  struct QNode{
95      locked: boolean variable, initially FALSE;
96  };
97  global shared variables
98      TAIL: reference to QNode, initially null;
99      CSOWNER: process identifier, initially ⊥;
100     CSBUSY: integer variable, initially 0;
101 per-process shared/persistent variables
102     POOL_p: array [0,1] of QNode, all elements
                initially {FALSE};
103     FACE_p: integer variable, initially 0;
104     LEAVING_p: boolean variable, initially
                   FALSE ;
105 private variables
106     pred, tail: reference to QNode;
107     busy: integer variable;
108 Procedure Cleanup()
109     if CSOWNER = p then CSOWNER := ⊥;
110     if TAIL = &POOL_p[FACE_p] then
111         CAS(TAIL, &POOL_p[FACE_p], null)
112     POOL_p[FACE_p].locked := FALSE;
113 Procedure Recover()
114     if CSOWNER = p then return;
115     Cleanup();
116 Procedure Enter()
117     if CSOWNER = p then return;
118     FACE_p := 1 − FACE_p;
119     POOL_p[FACE_p].locked := TRUE;
120     pred := FAS(TAIL, &POOL_p[FACE_p]);
121     if pred ≠ null then await not(pred.locked) ;
122     await CSOWNER = ⊥;
123     CSOWNER := p;
124     CSBUSY := CSBUSY + 1;
125 Procedure Exit()
126     Cleanup();
127 Procedure Withdraw()
128     Cleanup();
129 Procedure SafeToReclaim()
130     if not(LEAVING) then
131         LEAVING := TRUE;
132         tail := TAIL;
133         if tail = null then return;
134         busy := CSBUSY;
135         await (TAIL ≠ tail) or
                   (CSBUSY > busy);
136         if TAIL = null then return;
137         await (CSBUSY > busy);
```

that is still executing its `Enter` method and thus may *dereference* one of its queue nodes. Clearly, the condition holds if the queue is empty (*i.e.*, the tail pointer is **null**). Note that, when a process invokes the `SafeToReclaim` method, the tail pointer cannot point to any of its two queue nodes since the process would have completed a failure-free instance of the `Cleanup` method with respect to each of its nodes. Thus, if a process finds that the queue is not empty when leaving, it can infer that one or more nodes have been appended to the queue after its most recent append operation. The issue is that some of these nodes may have been appended to the queue before the last failure, and thus will be eventually abandoned by their owners.

In the `SafeToReclaim` method, the process first reads the tail pointer and returns if the queue is empty (Lines 132 and 133). Otherwise, it reads the sequence number mentioned earlier (Line 134). It then waits until either the tail pointer or the sequence number has advanced (Line 135). It next re-reads the tail pointer and returns if the queue is now empty (Line 136). If not, it implies that a *new* node has been added to the queue since the last system-wide failure. Thus, either the sequence number will eventually advance (Line 137) or the system will crash (Lines 130 and 131). The process can safely reclaim its memory in either case.

It is possible that a process may incur multiple RMRs while spinning on the tail pointer. This may happen when the same node is appended to the queue again or when a failed `CAS` instruction is executed on the tail pointer. In the first case, we can show that the sequence number has also advanced. In the second case, we can show that failed `CAS` instructions can generate at most three RMRs for a spinning process. This is because a process signals its successor only after trying to remove its node from the queue.

▶ **Theorem 4.** *LeeDL-SW satisfies the ME, SF, BCSR, BE, BR, BW and 0-FCFS properties. It supports dynamic joining as well as dynamic leaving. Finally, it has $O(1)$ RMR complexity in the CC model, and uses $O(1)$ space per process.*

We implemented the RME algorithm from this section and compared it against the CC-specific version of Jayanti, Jayanti and Joshi's (JJJ) RME algorithm (Section 3 of [29]). With BCSR property, our algorithm provided 50-75% better throughput than JJJ at low levels of parallelism, and around 15% better throughput at higher levels of parallelism. Without BCSR property, our algorithm was roughly 2-3× faster than JJJ. Details of our experiments are provided in Appendix B.

6 On Achieving Dynamic Joining and Leaving, Wait-Free Withdraw, Adaptive Space, and Bounded RMR Complexity in the DSM Model

In this section, we present an impossibility result for the DSM model (without explicit failure detection), and for a particular class of RME algorithms characterized according to the following combination of properties:

1. *Bounded RMR complexity per passage.* A process can only busy-wait by spinning on a variable that is local to it in the DSM model, and can only access a bounded number of remote memory locations per passage.
2. *Adaptive space complexity with external memory allocation.* The algorithm's state comprises $O(1)$ memory words shared by all processes, called the *global state*, and $O(1)$ additional memory words per process called *process-local state*. The process-local state for process p is allocated by process p externally (*i.e.*, outside of the RME algorithm) and is local to process p in the DSM model.
3. *Memory safety.* Externally allocated memory can be accessed by the RME algorithm only after it has been allocated and before it has been freed (*i.e.*, *use-after-free* is not permitted).
4. *Dynamic joining and leaving.* Any process can become active in an execution history after allocating its process-local state. Any active process can leave after a failure-free passage, meaning that it can free its process-local state in a bounded number of its own steps and then halt in the NCS.
5. *Wait-free withdraw:* Any process can complete any invocation of the `Withdraw` method in a bounded number of its own steps.

We now present the main result of this section:

▶ **Theorem 5.** *No RME algorithm can satisfy properties 1–5 simultaneously.*

Proof. Suppose for contradiction that some RME algorithm $\mathcal{A}$ does satisfy each of the properties 1–5. Since the algorithm's global state comprises only $O(1)$ memory words by property 2, it is easy to find two processes p and q such that all the global state is remote to q. We first construct a finite failure-free execution history H_0 involving these two processes and having the structure $H_0 = G_0^p \circ G_0^q$ where p begins a c-passage and enters the CS in G_0^p, and q begins a c-passage in G_0^q and enters a busy-wait loop since it cannot enter the CS concurrently with p. It follows that q must be spinning on its local memory at the end of G_0^q since $\mathcal{A}$ has bounded RMR complexity (property 1) and since every variable in the global state is remote to q by our choice of q. Next, observe that since H_0 exists, the execution $H_1 = G_0^p \circ f \circ G_0^q$ is also possible where f is a crash step (system-wide failure). This is because q cannot distinguish between the prefixes G_0^p and $G_0^p \circ f$ without an explicit failure

detector. We extend H_1 to the history $H_2 = G_0^p \circ f \circ G_0^q \circ G_1^p$ where p calls the `Withdraw` method in G_1^p and q takes no additional steps. It follows that p must eventually access q's local memory in some step s of G_1^p as otherwise q is stuck in a busy-wait loop forever if we extend H_2 by interleaving steps of p and q in a fair order. Next, we transform G_1^p to G_2^p by truncating this suffix at the step immediately before p accesses q's memory in step s, and observe that the history $H_3 = G_0^p \circ G_0^q \circ f \circ G_2^p$ is also possible, once again because there is no explicit failure detector. Process q cannot distinguish between the prefixes G_0^p and $G_0^p \circ f$, as explained earlier, and p cannot distinguish between $G_0^p \circ f \circ G_0^q$ and $G_0^p \circ G_0^q \circ f$ since only q takes steps in G_0^q. Finally, we transform H_3 to $H_4 = G_0^p \circ G_0^q \circ f \circ G_2^p \circ G_1^q$ where q executes the `Withdraw` method in G_1^q, frees its local memory, and leaves the execution (property 4) while p takes no additional steps. Such a history is possible since the `Withdraw` method is bounded (property 5). Extending H_4 by p's step s, which it is poised to execute at the end of fragment G_2^p, leads to a contradiction of memory safety (property 3) as p accesses q's local memory after q has already freed it. ◀

7 Related Work

Golab and Ramaraju's formulation of the RME problem [23, 24] is a theoretical take on the practical problem of making mutual exclusion locks robust against crash failures, which can be traced back to several earlier works [7, 8, 39, 46]. A pervasive pattern in this area of shared memory research is that fault-tolerant locks rely in various ways on support from the execution environment, for example where a centralized recovery process is invoked after a crash to clean up the internal state of the lock, where an explicit failure detector allows a waiting process to usurp a critical section held by a crashed process, or where shared variables are reset automatically to specific values during a failure. The RME problem formulation avoids such specific assumptions, and instead considers that a crashed process recovers and resumes execution eventually, unless it failed in the NCS.

Both classic ME algorithms and more recent RME algorithms are evaluated primarily with respect to remote memory reference (RMR) complexity in the cache-coherent (CC) and distributed shared memory (DSM) multiprocessor architectures, space complexity, as well as the set of correctness properties (*e.g.*, starvation freedom and fairness) achieved. In terms of worst-case RMR bounds for the individual process failure model, Golab and Ramaraju [23] established that RME can be solved for n processes using reads and writes with $O(\log n)$ RMRs per passage, which matches the lower bound of Attiya, *et al.* [5]. The latter bound applies to both ME and RME, and can be generalized for ME to comparison primitives using the RMR-efficient construction of Golab, *et al.* [20]. For algorithms that use other read-modify-write primitives, such as Fetch-And-Store or Fetch-And-Increment, a sub-logarithmic (*i.e.*, $O(\log n / \log\log n)$) upper bound was established jointly by Golab and Hendler [21] as well as Jayanti, Jayanti, and Joshi [27], and proven to be tight by Chan and Woelfel [10]. Katzan and Morrison [34] also proposed an $O(\log_w n)$ RMRs solution using w-bit Fetch-And-Add, which matches [21, 27] when $w \in \Theta(\log n)$ and reduces to $O(1)$ in the extreme case when $w \in \Theta(n)$; it was recently shown to be tight by Chan *et al.* in [9].

Most RME algorithms that tolerate individual process failures work correctly and achieve the same RMR complexity in the system-wide failure model. Golab and Hendler [22] solved the problem directly for system-wide failures using a failure detector and commonly supported primitives, and showed that the RMR complexity can be reduced to $O(1)$ using $O(\log n)$-bit Fetch-And-Store or Fetch-And-Increment primitives. Jayanti, Jayanti, and Joshi [33, 29] were the first to present RMR-optimal RME algorithms for system-wide failures that do not use an explicit failure detector. Their approach is presented as a way to transform a traditional

ME lock into an RME lock (under certain conditions) by maintaining three copies of the ME lock and using an intricate synchronization mechanism to guide requesting processes to an uncorrupted copy while concurrently resetting a possibly corrupted copy.

Some (ME) locks support the *abort* feature, which allows a process to abandon – within a bounded number of its own steps – its attempt to acquire the lock [19, 3, 30, 31]. This is useful in situations when a process may only wish to wait for a fixed amount of time to acquire the lock, and, if unable to do so, would prefer to cancel the attempt and perform some other task before reattempting to acquire the lock. The notion of abortability has been extended to RME locks as well in [32, 34]. We show in Appendix C that abortable RME and withdrawable RME are *equivalent* problems under *individual* failures and certain assumptions by giving RMR-preserving transformations that convert one type of lock into the other. Surprisingly, the two problems may not be same under *system-wide* failures. In particular, as we show in this work, it is possible to design an RMR-optimal RME lock that supports a wait-free withdraw section *assuming system-wide failures*. Note that any abortable RME lock under system-wide failures will also implement an abortable ME lock in a failure-free environment. However, the best known abortable ME lock has $\Omega(\log n / \log\log n)$ RMR complexity in the worst-case [3]. Intuitively, the reason for this gap is that a process may receive the abort signal *at any time* during its passage, whereas a process can execute the withdraw section only at the *beginning* of its passage *after a failure* to simplify recovery.

A comprehensive discussion of the RME problem and its solutions can be found in [14].

8 Conclusion and Future Work

In this work, we have presented a *modular* way to design an RMR-optimal RME lock for both CC and DSM models under system-wide failures without relying on an explicit failure detector. Our approach is flexible in the sense that an application can pick and choose the properties the RME lock should satisfy depending on its needs (*e.g.*, CSR, FRF[2], *etc.*). Further, we have proposed the notion of *withdrawing* from a lock acquisition as opposed to resetting the lock *after a failure*. The latter is more complex and requires greater synchronization among processes. Moreover, withdrawable RME locks make it easier in some scenarios to write fault-tolerant application programs for persistent memory.

In the future, we plan to conduct more comprehensive experiments to compare the performance of different RME lock alternatives to better understand how their features (*e.g.*, correctness properties, reliance on a failure detector) impact performance. We also plan to design a fully dynamic RMR-optimal RME algorithm for the DSM model, and investigate whether the blocking synchronization used in our dynamic algorithm for the CC model is inherently necessary.

References

1 Yehuda Afek, David S. Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM (JACM)*, 42(6):1231–1274, 1995.

2 Marcos K. Aguilera and Svend Frølund. Strict linearizability and the power of aborting. Technical Report HPL-2003-241, Hewlett-Packard Labs, 2003.

[2] FRF refers to *failure-robust fairness*, a type of fairness property defined in [22] for the system-wide failure model, which constraints the number of times a given process can be overtaken by other processes as regard to their super-passages. It is possible to provide a general transformation that adds the FRF property to any RME lock similar to that for the BCSR property [13].

3 Adam Alon and Adam Morrison. Deterministic abortable mutual exclusion with sublogarithmic adaptive RMR complexity. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 27–36, 2018.

4 James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing (DC)*, 16(2-3):75–110, 2003.

5 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proc. of the 40th ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008.

6 Ryan Berryhill, Wojciech Golab, and Mahesh Tripunitara. Robust shared objects for non-volatile main memory. In *Proc. of the 19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 20:1–20:17, 2016.

7 Philip Bohannon, Daniel Lieuwen, and Avi Silberschatz. Recovering scalable spin locks. In *Proc. of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 314–322, 1996.

8 Philip Bohannon, Daniel Lieuwen, Avi Silberschatz, S. Sudarshan, and Jacques Gava. Recoverable user-level mutual exclusion. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 293–301, 1995.

9 David Yu Cheng Chan, George Giakkoupis, and Philipp Woelfel. Word-size rmr trade-offs for recoverable mutual exclusion. In *Proc. of the 43rd ACM Symposium on Principles of Distributed Computing (PODC)*, 2023.

10 David Yu Cheng Chan and Philipp Woelfel. A tight lower bound for the RMR complexity of recoverable mutual exclusion. In *Proc. of the 40th ACM Symposium on Principles of Distributed Computing (PODC)*, 2021.

11 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

12 Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report 93-02-02, Department of Computer Science, University of Washington, 1993.

13 Sahil Dhoked. *Synchronization and Fault Tolerance Techniques in Concurrent Shared Memory Systems*. PhD thesis, The University of Texas at Dallas, 2022.

14 Sahil Dhoked, Wojciech Golab, and Neeraj Mittal. *Recoverable Mutual Exclusion*. Springer Nature, 2023.

15 Sahil Dhoked and Neeraj Mittal. An adaptive approach to recoverable mutual exclusion. In *Proc. of the 39th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–10, New York, NY, USA, 2020. doi:10.1145/3382734.3405739.

16 Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM (CACM)*, 8(9):569, 1965.

17 Rotem Dvir and Gadi Taubenfeld. Mutual exclusion algorithms with constant RMR complexity and wait-free exit code. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *Proc. of the International Conference on Principles of Distributed Systems (OPODIS)*, volume 95, pages 17:1–17:16, Dagstuhl, Germany, October 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

18 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32:374–382, 1985.

19 George Giakkoupis and Philipp Woelfel. Randomized abortable mutual exclusion with constant amortized RMR complexity on the CC model. In *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 221–229, 2017.

20 Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing (DC)*, 25(2):109–162, 2012.

21 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–220, 2017.

22 Wojciech Golab and Danny Hendler. Recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 17–26, 2018.

23 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

24 Wojciech Golab and Aditya Ramaraju. Recoverable mutual exclusion. *Distributed Computing (DC)*, 32(6):535–564, 2019.

25 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufman, 2012.

26 Prasad Jayanti, Siddhartha Jayanti, and Sucharita Jayanti. Towards an ideal queue lock. In *Proc. of the 21st International Conference on Distributed Computing and Networking (ICDCN)*, January 2020.

27 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. A recoverable mutex algorithm with sub-logarithmic RMR on both CC and DSM. In *Proc. of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 177–186, 2019.

28 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant rmr recoverable mutex under system-wide crashes, 2023. `arXiv:2302.00748`.

29 Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. Constant RMR system-wide failure resilient durable locks with dynamic joining. In *Proc. of the 35th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 227–237, 2023.

30 Prasad Jayanti and Siddhartha V. Jayanti. Constant amortized RMR complexity deterministic abortable mutual exclusion algorithm for CC and DSM models. In *Proc. of the 38th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 167–176, 2019.

31 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In Mohamed Faouzi Atig and Alexander A. Schwarzmann, editors, *Proc. of the International Conference on Networked Systems (NetSys)*, pages 217–232, 2019.

32 Prasad Jayanti and Anup Joshi. Recoverable mutual exclusion with abortability. In *Proc. of 7th International Conference on Networked Systems (NETYS)*, pages 217–232, 2019.

33 Anup Joshi. *Recoverable Mutual Exclusion Algorithms for Crash-Restart Shared-Memory Systems.* PhD thesis, Dartmouth College, May 2020.

34 Daniel Katzan and Adam Morrison. Recoverable, abortable, and adaptive mutual exclusion with sublogarithmic RMR complexity. In *Proc. of the 24th International Conference on Principles of Distributed Systems (OPODIS)*, pages 15:1–15:16, 2021.

35 Leslie Lamport. The mutual exclusion problem: part I – a theory of interprocess communication. *Journal of the ACM (JACM)*, 33(2):313–326, 1986.

36 Leslie Lamport. The mutual exclusion problem: part II – statement and solutions. *Journal of the ACM (JACM)*, 33(2):327–348, 1986.

37 Hyonho Lee. Local-spin mutual exclusion algorithms on the DSM model using fetch-&-store objects. Master's thesis, University of Toronto, 2003.

38 John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

39 Maged M. Michael and Yong-Jik Kim. Fault tolerant mutual exclusion locks for shared memory systems, 2009. US Patent 7,493,618.

40 Thomas Moscibroda and Rotem Oshman. Resilience of mutual exclusion algorithms to transient memory faults. In *Proc. of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 69–78, 2011.

41 Michel Raynal. *Algorithms for Mutual Exclusion.* MIT, 1986.

42 Michel Raynal and Gadi Taubenfeld. Mutual exclusion in fully anonymous shared memory systems. *Information Processing Letters*, 158:105938, 2020.

43 I. Rhee. Optimizing a FIFO, scalable spin lock using consistent memory. In *Proc. of the 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 106–114, December 1996.

44 Andy Rudoff. Re: cascade lake doesn't support clwb? [discussion post]. https://groups.google.com/g/pmem/c/DRdYIc70RHc/m/rtoP681rAAAJ, 2021. Google Groups.

45 Andy Rudoff and the Intel PMDK Team. Persistent memory development kit, 2020. [last accessed 2/11/2021]. URL: https://pmem.io/pmdk/.

46 Gadi Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice Hall, 2006.

47 Gadi Taubenfeld. Coordination without prior agreement. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proc. of the 36th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–334, 2017.

A A Recoverable Lock-Based Concurrent Linked List

We begin with a non-recoverable linked list algorithm adapted from Chapter 9 of Herlihy and Shavit's book [25], and shown below in algorithm 6. The list uses two sentinel nodes – head and tail – which ensures that a traversal can always acquire locks on consecutive nodes. The items stored in these sentinel nodes are minimum and maximum values from the domain of values stored in the list, and cannot be added or removed once the list is initialized.

Algorithm 6 A concurrent linked list based on fine-grained locking.

```
138 struct LLNode {
139     item: value stored in this node;
140     next: reference to next linked list node;
141     M: mutex lock;
142 };
143 global shared variables
144     head: reference to head sentinel node of
              linked list;
145 private variables
146     pred, curr, newNode: reference to LLNode;
147 boolean Procedure Add(item)
148     pred := head;
149     pred.M.Enter();
150     curr := pred.next;
151     curr.M.Enter();
152     while curr.item < item do
153         pred.M.Exit();
154         pred := curr;
155         curr := curr.next;
156         curr.M.Enter();
157     if curr.item = item then
158         curr.M.Exit();
159         pred.M.Exit();
160         return FALSE;
161     newNode := new LLNode;
162     newNode.next := curr;
163     pred.next := newNode;
164     curr.M.Exit();
165     pred.M.Exit();
166     return TRUE;
167 boolean Procedure Remove(item)
168     pred := head;
169     pred.M.Enter();
170     curr := pred.next;
171     curr.M.Enter();
172     while curr.item < item do
173         pred.M.Exit();
174         pred := curr;
175         curr := curr.next;
176         curr.M.Enter();
177     if curr.item = item then
178         pred.next := curr.next;
179         curr.M.Exit();
180         pred.M.Exit();
181         return TRUE;
182     curr.M.Exit();
183     pred.M.Exit();
184     return FALSE;
```

A recoverable (*i.e.*, strictly linearizable [2]) version of the above algorithm is obtained by allocating the node structure and program variables (including *curr* and *pred*) in persistent memory, replacing the node-level ME locks with RME locks, and adding a recovery procedure that cleans up any locks that may have been corrupted by a crash. Program variables that were private before the transformation now need appropriate initial values since they may be accessed by the recovery procedure before they can be explicitly initialized. A value of **null** is appropriate for the linked list since the variables are all pointers to `LLNode`.

Algorithm 7 Recovery protocol based on withdrawable RME locks.

```
185 Procedure Recover()
186   if pred ≠ null then pred.M.Withdraw();
187   if curr ≠ null and curr ≠ pred then curr.M.Withdraw();
```

Recovery Using Withdrawable RME Locks. We begin by explaining how recovery can be achieved using the withdrawable RME locks introduced in this work. The recovery procedure executed by a process after a failure (or at the start of every execution) is presented in algorithm 7. The idea is to clean up any locks a process may have held at the time of a crash failure by executing the `Withdraw` section, and allow a garbage collector to deal with memory leaks. No further action is required since the fundamental structure of the linked list cannot be corrupted by a system-wide failure. The relevant locks are identified by the *curr* and *pred* variables, which usually point to the last two linked list nodes accessed, and sometimes to a single node (*e.g.*, if a crash occurs immediately after Line 174). The entire recovery protocol is wait-free as long as the locks provide *wait-free* `Withdraw` sections, which means that it cannot possibly introduce deadlock.

Recovery Using RME Locks in the Style of Jayanti, Jayanti, and Joshi. For comparison, we consider an alternative design of the recovery procedure based on Jayanti, Jayanti, and Joshi's RME lock [29], which lacks a `Withdraw` section and uses a slightly different interface in which the `Recover` section directs a process to either return to the NCS or proceed to the CS via its return value. As explained in Section 7, the JJJ algorithm can be used to simulate a `Withdraw` section, but that would go against the intent of their reformulation of the RME problem, which is to resume execution of the CS if that is where the failure occurred. Applying the latter principle to the linked list, we immediately run into difficulties. To begin with, the recovery procedure must consider different cases depending on the return values of the `Recover` section: IN_REM *vs.* IN_CS. With two locks to recover, there are four principal cases to analyze:

1. *pred*.*M*.`Recover()` returns IN_REM and *curr*.*M*.`Recover()` returns IN_REM (*e.g.*, crash at line 148): nothing to do
2. *pred*.*M*.`Recover()` returns IN_REM and *curr*.*M*.`Recover()` returns IN_CS (*e.g.*, crash at line 153): resume traversal
3. *pred*.*M*.`Recover()` returns IN_CS and *curr*.*M*.`Recover()` returns IN_REM (*e.g.*, crash at line 155): resume traversal
4. *pred*.*M*.`Recover()` returns IN_CS and *curr*.*M*.`Recover()` returns IN_CS (*e.g.*, crash at line 161): resume traversal

Upon closer inspection of the above four cases, we note additional complications. First, the traversal may need to be resumed either inside the `Add` procedure or inside the `Remove` procedure, and so the algorithm must be augmented with additional state to keep track of which procedure the process was executing; this hurts performance and effectively increases the number of cases during recovery from four to seven as the traversal may need to resume at slightly different points in cases 2–4 (case 1 applies equally well to both `Add` and `Remove`). Moreover, in case 4 alone we must consider separately the subcase when $pred = curr$ and the subcase when $pred \neq curr$, which increases the total number of cases from seven to nine. Without fleshing out the recovery protocol in full detail, we conclude that the solution would be substantially more complex than our solution in algorithm 7.

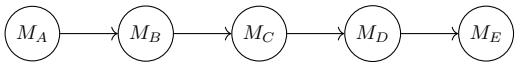


Figure 1 Illustration of deadlock scenario with coarse-grained idempotent actions.

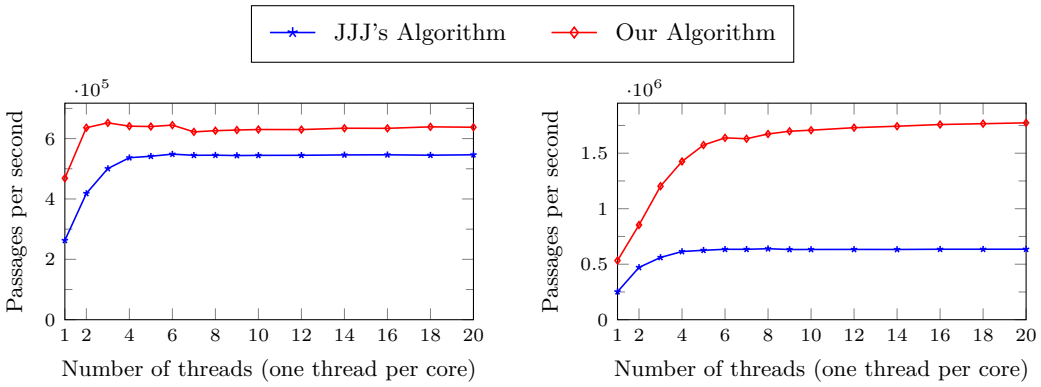
Recovery Using RME Locks in the Style of Golab and Ramaraju. Finally, we consider recovery using plain RME locks, as defined by Golab and Ramaraju [23, 24]. Such locks lack a `Withdraw` section, and are based on the classic concept of idempotent actions: a process recovers by simply repeating its entire passage. A course-grained interpretation of this paradigm has each process repeating the entire linked list operation it was performing at the time of failure, which easily leads to deadlock. In the example structure illustrated below in Figure 1, process p could crash in the CS of lock M_D, at the same time process q could crash in the CS of another lock M_B closer to the head of the list, then p could block while entering M_B during recovery because it is effectively still held by q, and q could subsequently block while entering the lock M_A in the predecessor node, which is still held by p due to hand-over-hand locking.

A more fine-grained interpretation of idempotent actions has each process recovering only the one or two locks it was accessing at the time of failure, similarly to algorithm 7 but with a sequence of calls to `Recover`, `Enter`, and `Exit` in place of a single call to `Withdraw`. However, once again we run into the possibility of deadlock. If a process p tries to recover *pred* first and then *curr*, as in algorithm 7, then consider what happens if the system-wide failure occurred immediately after p released *pred*.M at line 153: another process q could race ahead and traverse the linked list after the crash, acquire the same lock *pred*.M, then block on *curr*.M where p effectively still holds the CS, and cause p to block forever while trying to enter *pred*.M. We achieve a better outcome in this scenario if p tries to recover *curr* first and then *pred*, however even this strategy can cause deadlock if processes access multiple linked list structures and recover them in conflicting orders. This holds even if the RME locks provide wait-free `Recover` and `Exit` sections since a process can block in `Enter`. To summarize, although we can envision correct recovery protocols for linked data structures that use plain RME locks in the style of Golab and Ramaraju, such protocols are much harder to design than algorithm 7 as actions must be ordered carefully to avoid deadlocks.

B Experimental Evaluation

This section presents an empirical comparison of RME algorithms designed specifically for system-wide failures that do not use an explicit failure detector. The hardware platform is a 20-core 2nd generation Intel Xeon processor with Optane Persistent Memory, which supports the CC model. We compare Jayanti, Jayanti and Joshi's RME algorithm for the CC model (Section 3 of [29]), hereby referred to as JJJ, with a simplified version of our RME algorithm from Section 5 that lacks dynamic leaving. Both RME algorithms are based on Lee's ME algorithm [37]. Both algorithms were implemented in C++ using the Intel Persistent Memory Development Kit [45] on Linux. Shared variables were implemented using the `std::atomic` template class, and memory operations were applied with sequential consistency for simplicity. The `libpmemobj` library was used for memory allocation to ensure that shared variables are mapped to persistent memory, and for persistent pointers to deal with address space relocation across failures.

The cache system on our hardware platform is not part of the persistent domain and hence volatile. As a result, if the most recent value of a variable that is meant to be persistent has not been flushed to the persistent memory before a failure, then it will be lost. To



(a) System throughput with CSR property.

(b) System throughput without CSR property.

Figure 2 Scalability comparison of RME algorithms on one processor in failure-free execution.

run correctly on our hardware platform, recoverable algorithms designed for the CC model must be annotated by judicious placement of *persistence* instructions in the source code. The PMDK provides the `pmem_persist` function for this purpose, which internally applies a cache line write-back and store fence. Naïvely persisting variables after every shared memory operation is sufficient to ensure correctness with respect to fundamental correctness properties that refer to the program's state (*e.g.*, ME, SF, CSR), but does not preserve the RMR complexity of busy-wait loops. This is because our hardware platform implements the cache line write-back instruction in a simplistic way that always invalidates the cache line [44]. The naïve approach is easily optimized for algorithms that use simple busy-wait loops, namely ones that spin on a single variable until it reaches a specific value, by persisting the spin variable only after the last iteration of the loop.[3] Even this final persistence instruction can be skipped in the entry section of Lee's ME algorithm since the value of the spin variable is not relevant for recovery when this algorithm is used as a building block of an RME algorithm. It is also safe to omit the persistence instruction when a process reads a single-writer shared variable owned by that process, such as an element of the FACE array in Lee's algorithm. We apply these optimizations to both our and JJJ's algorithm.

Figure 2 presents the scalability of the two algorithms on our hardware platform. Each point plotted is the average of five repetitions, each lasting 5 s. The error bars represent the sample standard deviation, and are imperceptibly small in most cases. The throughput numbers (total number of passages completed per second) presented include the overhead of persistence instructions and the higher latency of persistent memory over DRAM. Figure 2a presents data for variations of the two RME algorithms that satisfy the CSR property, while Figure 2b considers variations of the same algorithms that do not satisfy the CSR property. The non-CSR version of our algorithm is obtained by deleting lines of pseudocode that track CS ownership (specifically, code ending with one "◀"). JJJ's algorithm, in contrast, relies on CS ownership state to achieve mutual exclusion among recovering processes, and so the same transformation is not applicable. Note that Line 14 in Figure 3 of [29] is safe to remove and Line 26 in Figure 3 of [29] must be adjusted.

[3] This also applies for the two spin loops executed in parallel in the entry section of JJJ's algorithm.

Algorithm 8 Equivalence between an RME lock that supports a wait-free `Withdraw` method and an abortable RME lock that supports a wait-free `Exit` method under individual failures.

a : Abortability to Withdrawability.

```
188 global shared variables
189   M: instance of abortable RME lock;
190 per-process shared/persistent variables
191   Abort_p: abort flag;

192 Procedure Withdraw()
193   Abort_p := TRUE;
194   if M.Recover() then
195     if M.Enter() then
196       M.Exit();
197   Abort_p := FALSE;

198 Procedure Recover()
199   if Abort_p then
200     Withdraw();
201   M.Recover();

202 Procedure Enter()
203   M.Enter();

204 Procedure Exit()
205   M.Exit();
```

b : Withdrawability to Abortability.

```
206 global shared variables
207   M: instance of withdrawable RME lock;
208 per-process shared/persistent variables
209   Abort_p: abort flag;

210 boolean Procedure Recover()
211   M.Recover() || await Abort_p;
212   if Abort_p then
213     M.Withdraw();
214     return FALSE;
215   else return TRUE;

216 boolean Procedure Enter()
217   M.Enter() || await Abort_p;
218   if Abort_p then
219     M.Withdraw();
220     return FALSE;
221   else return TRUE;

222 Procedure Exit()
223   M.Withdraw();
```

As the graphs show, with the CSR property, our algorithm has around 75% and 50% higher throughput than JJJ's algorithm for one and two threads, respectively. The gap stabilizes to around 15% at higher levels of parallelism. Without the CSR property, the gap between the two algorithm is much higher; specifically, our algorithm is 2-3 times faster than JJJ's algorithm. Further, while the performance of JJJ's algorithm is immune to whether or not the CSR property holds, our algorithm sees a significant speedup by a factor of 2-3 when the CSR property is not required.

C Abortability and Withdrawability

To support the notion of abortability in our execution model, we modify the `Recover` and `Enter` methods to return a boolean value. A return value of true indicates that the method was executed to completion, whereas a return value of false indicates the attempt to acquire the lock was abandoned and the method terminated prematurely. An application indicates its desire to abort by raising a boolean flag. Note that an attempt to acquire the lock can only be abandoned if the flag is raised. A process typically executes a passage by invoking the `Recover`, `Enter` and `Exit` methods in order. However, if either `Recover` or `Enter` returns false (which would only happen if the abort flag was raised), then the passage is considered to be complete and subsequent methods are not invoked.

In this section, we show the following equivalence between the two types of RME locks under the *individual* failure model. An RME lock that supports a wait-free `Withdraw` method can be transformed into an abortable RME lock that supports a wait-free `Exit` method. Conversely, an abortable RME lock that supports a wait-free `Exit` method can be transformed into an RME lock that supports a wait-free `Withdraw` method. Further, both transformations only incur $O(1)$ additional RMRs and thus are *RMR preserving*.

Abortability to withdrawability. The transformation is given in algorithm 8a. In the `Withdraw` method of the target lock, a process attempts to execute the `Recover`, `Enter` and `Exit` methods of the base abortable lock in sequence with the abort flag raised (by the algorithm). If either `Recover` or `Enter` returns false (indicating that the attempt to acquire the lock was abandoned), the method immediately returns without invoking the remaining methods. Otherwise, the process completes the super-passage of the base abortable lock by executing the `Exit` method. Clearly, all three methods are wait-free in the presence of the abort signal.

Withdrawability to abortability. The transformation is given in algorithm 8b. The main idea is that, in the `Recover` (respectively, `Enter`) method of the target lock, the process invokes the `Recover` (respectively, `Enter`) method of the base withdrawable lock and concurrently monitors the abort flag. If the abort flag is raised (by the environment), the process *simulates* a failure by prematurely terminating the method, and executes the `Withdraw` method instead to complete the super-passage of the base lock.