Noname manuscript No.
(will be inserted by the editor)

Recoverable Mutual Exclusion

Wojciech Golab · Aditya Ramaraju

Received: September 2016 / Revision 1: September 2018 / Revision 2: September 2019 / Accepted: October 2019

Abstract Mutex locks have traditionally been the most common mechanism for protecting shared data structures in concurrent programs. However, the robustness of such locks against process failures has not been studied thoroughly. The vast majority of mutex algorithms are designed around the assumption that processes are reliable, meaning that a process may not fail while executing the lock acquisition and release code, or while inside the critical section. If such a failure does occur, then the liveness properties of a conventional mutex lock may cease to hold until the application or operating system intervenes by cleaning up the internal structure of the lock. For example, a process that is attempting to acquire an otherwise starvation-free mutex may be blocked forever waiting for a failed process to release the critical section. Adding to the difficulty, if the failed process recovers and attempts to acquire the same mutex again without appropriate cleanup, then the mutex may become corrupted to the point where it loses safety, notably the mutual exclusion property. We address this challenge by formalizing the problem of recoverable mutual exclusion, and proposing several solutions that vary both in their assumptions regarding hardware support for synchronization, and in their efficiency. Compared to known solutions, our algorithms are more robust as they do not restrict where or when a process may crash, and provide stricter guarantees in terms of efficiency, which we define in terms of remote memory references.

Keywords mutual exclusion · fault tolerance · recovery · concurrency · synchronization · shared memory · non-volatile main memory · multi-core algorithms · durable data structures

This research is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, Discovery Grants Program; the Ontario Early Researcher Awards Program; and the Google Faculty Research Awards Program.

W. Golab
Department of Electrical and Computer Engineering
University of Waterloo, Canada
E-mail: wgolab@uwaterloo.ca

A. Ramaraju
Department of Electrical and Computer Engineering
University of Waterloo, Canada
E-mail: a2ramaraju@uwaterloo.ca

1 Introduction

Concurrent programs running on multi-core architectures empower essential applications and services today, ranging from mundane desktop computing to massive back-end infrastructures that drive web search, e-commerce, and online social networks. Mutex locks have traditionally been the most popular mechanism for protecting shared data structures in concurrent programs owing to their simplicity: any sequential data structure can be made thread-safe by protecting operations on the data structure using a critical section. Mutual exclusion is therefore one of the oldest problems addressed in the literature on shared memory synchronization, starting with Dijkstra's seminal paper [12]. Over decades of research, locks have evolved from simple two-process algorithms to scalable mechanisms that provide elaborate correctness properties, such as fairness and local spinning [4]. One technical aspect of mutual exclusion, however, has remained nearly constant: a typical mutex lock cannot survive the failure of even one process while it is accessing the lock.

Failures in concurrent programs are an increasingly important reality, and may arise in several ways. For example, individual processes or threads of control in a

Sec.	Algorithm	Failure-Free RMR Complexity	Worst-Case RMR Complexity	Synchronization Primitives Used
3.1	two out of N processes	$O(1)$	$O(1)$	reads and writes
3.3	N-process arbitration tree	$O(\log N)$	$O(\log N)$	reads and writes
4.1	N-process base mutex (conventional) $\longrightarrow$ N-process target mutex (recoverable)	same as base mutex	grows with the number of failures (unbounded)	same as base mutex + reads and writes
4.2	target mutex from Sec. 4.1 $\longrightarrow$ N-process recoverable mutex with bounded RMR complexity	same as base mutex in Sec. 4.1	$O(N)$ when base mutex from Sec. 4.1 incurs $O(N)$ RMRs and uses $O(N)$ space	same as base mutex from Sec. 4.1 + reads, writes, and Compare-And-Swap

Fig. 1 Summary of recoverable mutual exclusion algorithms with respect to remote memory reference (RMR) complexity and required synchronization primitives.

program may terminate abruptly due to software bugs or deadlock resolution mechanisms. Similarly, processes may be affected by a loss of electrical power or faulty interconnect, especially in large-scale multiprocessor architectures that incorporate multiple system boards connected by a complex network of electronic and optical components. The straightforward approach to dealing with such failures is to shut down the entire program and restart it, which destroys all in-memory data structures in the program's address space, forcing recovery from slow secondary storage. In comparison, recovery by repairing in-memory data structures directly, as opposed to rebuilding them, is potentially much less disruptive as it bypasses secondary storage altogether. This style of recovery is especially well suited to forthcoming hardware architectures that incorporate non-volatile main memory (NVRAM) [16,41,42,44], which allows in-memory data structures to survive power failures. NVRAM collapses main memory and secondary storage into a single layer in the memory hierarchy, and makes it possible to use data structure repair to deal with both system-wide failures arising from power loss and partial failures arising from software bugs or hardware faults. In contrast, conventional systems with dynamic random-access memory (DRAM) must fall back on secondary storage to recover from power loss even if they implement data structure repair to deal more efficiently with partial failures, which increases software complexity.

Restoring a data structure protected by a mutex lock following a failure entails solving two technical problems: (1) completing or rolling back any actions in the critical section that may have been interrupted by the failure; and (2) repairing the internal state of the lock, if needed, to preserve safety and liveness. The first problem is arguably much easier because the critical section entails sequential reading and writing, and can be solved using database systems techniques [22]. The second problem involves more intricate lock acquisition and release code that deals with race conditions. If a process crashes while executing this code, then the liveness properties of a conventional mutex lock may cease to hold until the lock is repaired. For example, a process that is attempting to acquire an otherwise starvation-free mutex lock may be blocked forever, waiting for a crashed process to release the critical section. Adding to the difficulty, if the failed process recovers and attempts to acquire the same mutex again without appropriate cleanup, then the mutex may be corrupted to the point where it loses safety, particularly the mutual exclusion property.

Prior work on recoverable locking overcomes the inherent technical difficulty of the problem by introducing simplifying assumptions into the model of computation. Lamport and Taubenfeld define models where a process failure affects the values of shared variables in well-defined ways, for example by resetting any single-writer shared registers owned by the failed process [34,48]. Bohannon, Lieuwen, and Silberschatz, as well as Bohannon, Lieuwen, Silberschatz, Sudarshan, and Gava, focus instead on algorithms with multi-writer variables, and assume that the operating system provides failure detection [8,9]. In their approach, the internal state of the lock is repaired carefully by a reliable centralized recovery process that executes concurrently with the lock acquisition and release protocols. Our approach similarly allows multi-writer variables, but deals with failure detection and recovery in a more abstract way by assuming that a failed process is revived eventually and given an opportunity to participate in recovery. Specifically, recovery actions are integrated into the lock acquisition code, where they are exposed to both concurrency and failures.

The contributions in this paper begin with a formal definition of recoverable mutual exclusion with respect to our crash-recovery failure model. We then present a collection of recoverable mutex algorithms, whose properties are summarized in Figure 1. We start with a two-process solution in Section 3.1 that uses atomic read and write operations only, then progress to an

N-process solution in Section 3.3 also using reads and writes, and finally present a pair of algorithms in Sections 4.1–4.2 that can transform any conventional mutex ("base mutex") into a recoverable mutex ("target mutex") while preserving efficiency in the absence of failures. The latter transformation can be used to add recoverability to efficient queue-based locks (e.g., MCS [39]).

2 Model

We consider N asynchronous unreliable processes, labeled $p_1, p_2, \ldots, p_N$, that communicate by accessing variables in shared memory. These processes compete for an exclusive lock (mutex), which can be used to protect a shared resource, by following the execution path illustrated in Algorithm 1. At initialization, as well as immediately after crashing, a process is in the remainder section (RS), where it does not access the lock. Upon leaving the RS, a process always executes the *recovery section*, denoted by the procedure **Recover**(). This is where the process cleans up the internal structure of the lock, if needed, following a crash failure. Next, a process attempts to acquire the lock in the *entry section*, denoted by the procedure **Enter**(). The entry section is sometimes modeled in two parts: a bounded[1] section of code called the *doorway* that determines the order in which processes acquire the lock, followed by a *waiting room*, where processes wait for their turn.[2] Upon completing the entry section, a process has exclusive accesses to the *critical section* (CS). A process subsequently releases the lock by executing the *exit section*, denoted by the procedure **Exit**(), and finally transitions back to the RS.

Our model intentionally defines a sequential flow of control in which processes always execute code in the order prescribed by Algorithm 1: RS, **Recover**(), **Enter**(), CS, and **Exit**(). This convention follows naturally from the assumption that **Recover**(), **Enter**() and **Exit**() are procedures that are called by an application and punctuated by application-specific code (e.g., the RS and CS), rather than sections of a contiguous algorithm.[3] Branches from the code of one procedure to the code of a different procedure are not permitted because in practice one procedure must return control to the application (i.e., pop a stack frame) before the application invokes the next procedure in the sequence. For example, a process that has no recovery actions to perform can branch from the beginning of **Recover**() to the end, skipping the body of the recovery section, but a process that is recovering from a failure in **Exit**() cannot branch from **Recover**() directly to **Exit**() and skip the application-specific CS code.

The boundary between the recovery and entry sections in Algorithm 1 is determined by the position of the doorway within an algorithm. That is, the recovery section ends prior to the first step of the doorway, which defines the start of the entry section. The separation of code into explicit recovery and entry sections makes it possible to define the doorway as a prefix of the entry section, as in conventional mutual exclusion. On the other hand, the recovery section has no formal significance in algorithms that lack doorways entirely.

```
loop forever
    Remainder Section (RS)
    Recover()
    Enter() { Doorway
            { Waiting Room
    Critical Section (CS)
    Exit()
```

Algorithm 1: Execution path of a process participating in recoverable mutual exclusion. A crash failure reverts a process back to the RS.

Correctness properties for mutex algorithms are expressed in reference to *histories* that record the actions of processes as they execute the pseudo-code of Algorithm 1. Formally, a history H is a sequence of *steps* that come in two varieties: *ordinary steps* and *crash steps*. An ordinary step is a shared memory operation combined with local computation such as arithmetic operations, accessing one or more private variables, and advancing the program counter. A crash step denotes a process failure that resets the private variables of a process to their initial values. Each process has a *program counter*, which is a private variable that identifies the next ordinary step a process is poised to take. The program counter points to the RS initially, and is updated in each step. We say that a process is *at line X* of an algorithm if its program counter identifies an ordinary step corresponding to line X of the algorithm's pseudo-code. One line of pseudo-code may entail multiple steps.

[1] The term *bounded* in reference to a piece of code means that there exists a function f of the number of processes N such that the code performs at most $f(N)$ shared memory operations in all executions of the algorithm instantiated for N processes.

[2] As explained later on in the model near the discussion of First-Come-First-Served fairness, we assume that the doorway is well-defined and bounded only in a subset of execution histories that are relevant to our weaker notion of FCFS.

[3] In a practical implementation, the code of **Recover**() and **Enter**() can be packaged in a single procedure for simplicity.

The next step a process takes after a crash is either another crash step, or the first step of **Recover**(). A process is said to *recover* following a crash step by executing the first step of **Recover**(). A *passage* is a sequence of steps taken by a process from when it begins **Recover**() to when it completes **Exit**(), or crashes, whichever event occurs first. We say that a process is *in cleanup* if it is executing **Recover**() following a passage that ended with a crash step. A passage is called *failure-free* if it does not end with a crash step, which includes all incomplete passages. A *super-passage* is a maximal non-empty collection of consecutive passages executed by the same process where (only) the last passage in the collection is failure-free. A process is *executing a super-passage* if it is either outside the RS or in the RS following a crash failure. A process may execute the CS at most once per passage, and possibly multiple times in one super-passage.

A process that completes the CS and then crashes in **Exit**() is required in our model to continue taking steps until it completes a failure-free passage, which entails one or more additional and redundant executions of the CS. Although this behavior is somewhat inefficient, it simplifies the model conceptually by reducing the number of possible execution paths. Jayanti and Joshi [29] propose an alternative model of recovery that permits additional transitions among the different sections of a recoverable mutex, and opens the door to recovery without redundant executions of the CS in some scenarios. Their model allows a process that crashes in **Exit**() to transition from **Recover**() directly to the CS, to **Exit**(), or even to the RS. Informally speaking, such transitions might occur if the crash occurs at the beginning of, in the middle of, and at the end of **Exit**(), respectively.

A passage through a recoverable mutex algorithm may be influenced directly or indirectly by process failures. We formalize the notion of potential influence by reasoning precisely about interleaving among passages. We say that one passage *interferes* with another if their super-passages are concurrent, meaning that neither super-passage ends before the other begins. For any integer $k \geq 0$, we call a passage *k-failure-concurrent* if and only if:

- $k = 0$ and the passage ends with a crash or begins in cleanup (i.e., the previous passage of the same process ended with a crash); or
- $k > 0$ and the passage interferes with any passage (possibly itself) that is $(k-1)$-failure-concurrent.[4]

[4] The term *cleanup-concurrent* defined in the conference version of this paper [20] is analogous to 1-failure-concurrent in this model.

It follows easily that if a passage is k-failure-concurrent then it is also k'-failure-concurrent for any $k' > k$. In general, smaller values of k are preferred when the desired behavior of one process (e.g., a complexity bound) is conditioned on the absence of a k-failure-concurrent passage. This is because decreasing k tends to reduce the number of such k-failure-concurrent passages.

The set of histories generated by possible executions of a mutex algorithm is prefix-closed. For any finite history H, we denote the length of H (i.e., number of steps in H) by $|H|$. A history H is *fair* if it is finite, or if it is infinite and every process either executes zero steps, or halts in the RS after completing a failure-free passage, or executes infinitely many steps. We assume for simplicity of analysis that the critical section is bounded.

The correctness properties of our recoverable mutex algorithms are derived from Lamport's formalism [36], which considers only permanent crashes ("unannounced death") and Byzantine failures ("malfunctioning"). Differences between our definitions of correctness properties and Lamport's are highlighted in *italics* for clarity. Note that mutual exclusion, deadlock-freedom, and terminating exit are the only properties considered essential for correctness; the others are desirable but optional [4].

Mutual Exclusion (ME): For any finite history H, at most one process is in the CS at the end of H.

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

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

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

Wait-Free Exit (WFE): For any history H, any execution of **Exit**() by a process p_i completes in a bounded number of p_i's steps.

The DF and SF properties guarantee progress only in fair histories with finitely many crash steps, which allows for histories where every process takes infinitely many steps and yet no process enters the CS. Such histories must be included in the model because no algorithm can guarantee progress irrespective of the frequency of failures. Jayanti, Jayanti, and Joshi define a more stringent SF property that prevents starvation in executions that contain infinitely many failures pro-

vided that each process crashes only a finite number of times in each super-passage [30].

The SF and WFE properties are compatible with the algorithmic techniques introduced in this paper, and are either guaranteed or preserved by all our algorithms (see Theorems 1,3–6 for Algorithms 2–6, respectively).

Next, we introduce new properties to constrain the complexity of the recovery section:

Wait-Free Recovery (WFR): For any history H, any execution of **Recover**() by a process p_i completes in a bounded number of p_i's steps.

k-Bounded Recovery (k-BR): For any history H, any execution of **Recover**() by a process p_i completes in a bounded number of p_i's steps unless p_i's passage is k-failure-concurrent.[5]

It is essential for a recoverable mutual exclusion algorithm to satisfy WFR or k-BR for some $k \geq 0$, which ensures that the recovery section is bounded in the absence of failures. It follows easily that WFR is strictly stronger than 0-BR, and that for any $k \geq 0$ and $k' > k$, k-BR is strictly stronger than k'-BR. In any algorithm that satisfies WFR, the recovery code can be shifted to the entry section and joined with the doorway, although doing so weakens the FCFS and k-FCFS properties defined shortly by lengthening the doorway.

We also propose a pair of properties that simplify recovery from a crash failure inside the critical section, and make it possible to nest recoverable mutex locks:

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

Bounded Critical Section Re-entry (BCSR): For any history H and for any process p_i, if p_i crashes inside the CS then p_i incurs a bounded number of steps in each subsequent execution of the recovery and entry sections until it re-enters the CS.

Lemma 1 *If a recoverable mutex algorithm satisfies BCSR then it also satisfies CSR.*

Proof Suppose that process p_i crashes inside the CS in step c of some history H of the mutex algorithm. Suppose for contradiction that H has a prefix G that includes c, and at the end of which some other process p_j has entered the CS, and yet p_i has not yet re-entered the CS after c in G. By the BCSR property, if p_i continues to takes steps after G then it re-enters the CS in a bounded number of its own steps by completing the recovery and entry sections. Let G' denote the extension of G where p_i has done so. Then G' is a finite history of the mutex algorithm at the end of which both p_i and p_j are in the CS simultaneously. This contradicts the mutual exclusion property of the algorithm, which is required for correctness. □

[5] The Bounded Recovery property defined in the conference version of this paper [20] is analogous to 1-BR in this model.

Finally, we present definitions related to first-come-first-served (FCFS) fairness. The standard definition of FCFS is given first for completeness:

First-Come-First-Served (FCFS): For any history H, suppose that process p_i completes the doorway in its ℓ_i-th passage before process p_j begins the doorway in its ℓ_j-th passage. Then p_j does not enter the CS in its ℓ_j-th passage before p_i enters the CS in its ℓ_i-th passage.

The above FCFS property is incompatible with the CSR property in the sense that a process may bypass the doorway and waiting room entirely while recovering from a failure in the CS, and so we introduce an alternative fairness property called k-FCFS.

k-First-Come-First-Served (k-FCFS): For any history H, suppose that process p_i begins its ℓ_i-th passage and p_j begins its ℓ_j-th passage. Suppose further that neither passage is k-failure-concurrent, and that process p_i completes the doorway in its ℓ_i-th passage before process p_j begins the doorway in its ℓ_j-th passage in H. Then p_j does not enter the CS in its ℓ_j-th passage before p_i enters the CS in its ℓ_i-th passage.

Intuitively, k-FCFS is a weakening of FCFS that ignores k-failure-concurrent passages, where processes may enter the CS out of order with respect to their execution of the doorway. In fact, the doorway can be undefined or unbounded in such passages since it is not relevant. (We adopt this approach in the transformations presented in Section 4.) It follows easily that FCFS is strictly stronger than 0-FCFS, and that for any $k \geq 0$ and $k' > k$, k-FCFS is strictly stronger than k'-FCFS.

Aside from fundamental safety and liveness properties, we are interested in the efficiency of mutex algorithms, particularly with respect to the number of *remote memory references* (RMRs) executed by a process per passage. The precise definition of RMRs depends on the shared memory hardware architecture, and in this paper we consider both the cache-coherent (CC) and distributed shared memory (DSM) models [4], which are illustrated in Figure 2. In the CC model, any memory location can be made locally accessible to a process by creating a copy of it in the corresponding processor's cache. A distributed coherence protocol ensures that when a memory location is overwritten by one processor, cached copies held by other processors are either invalidated or updated. In contrast, the DSM model lacks (coherent) caches but allows each processor to locally

access a portion of the address space corresponding to its attached memory module.[6]

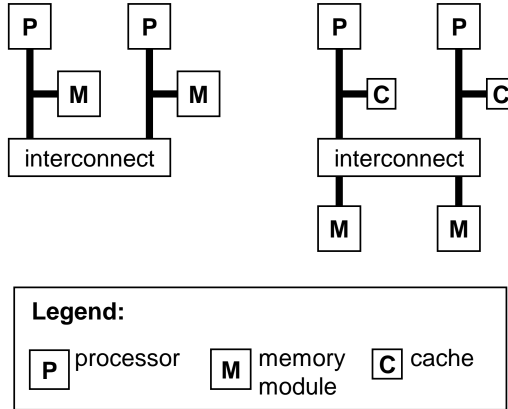

Fig. 2 Abstract models of shared memory architectures—DSM (left) and CC (right).

Loosely speaking, an RMR in the CC or DSM model is any memory operation that traverses the interconnect shown in Figure 2. In the CC model, we (conservatively) count each shared memory operation as an RMR with the exception of an *in-cache* read, which occurs when a process p_i reads a variable v that it has already read in an earlier step, following which step no process has accessed v except by a read operation. In the DSM model, each shared variable is local to exactly one process (assuming a one-to-one mapping between processes and processors), which is determined statically at initialization. A mutex algorithm is called *local spin* if its worst-case RMR complexity per passage is bounded.

3 Solutions Using Atomic Reads and Writes

Although modern multiprocessors support a variety of powerful read-modify-write primitives, algorithms that use only atomic reads and writes can be made recoverable more easily if they rely partially or wholly on single-writer shared variables. This is because updates to such variables are idempotent. Therefore, we first present two such algorithms, and then use them in Section 4 as building blocks of more advanced solutions that attain better RMR complexity in the absence of failures by leveraging read-modify-write operations. Notably, the N-process algorithm described shortly in Section 3.3 is used to protect the recovery section of the algorithms presented later on in Section 4. We also present a transformation in Section 3.2 that adds the BCSR property to any recoverable mutex.

[6] Despite the prevalence of cache-coherent architectures, the DSM model remains important in practice because of its inherent scalability. Intel's Single-chip Cloud Computer, for example, sacrifices cache-coherence "to simplify the design, reduce power consumption and to encourage the exploration of datacenter distributed memory software models" [26].

3.1 Algorithm for Two-of-N-Processes

Shared variables:
- T: long process ID or $\perp$, initially $\perp$
- $C[...]$: associative array indexed by left and right, each element a tuple of the form $\langle$long process ID or $\perp$, integer$\rangle$, each element initially $\langle \perp, 0\rangle$
- $P[1..N]$: array of integer spin variables, element $P[i]$ local to process p_i in the DSM model, initially all elements zero

Definitions:
- $\mathsf{other}(side) = \begin{cases} \mathsf{left} & \text{if } side = \mathsf{right} \\ \mathsf{right} & \text{if } side = \mathsf{left} \end{cases}$

Procedure Recover(*side*) for process p_i

```
1  if C[side] = ⟨i, 1⟩ then
2      ⟨rival, ...⟩ := C[other(side)]
3      if rival ≠ ⊥ then
4          P[rival] := 2
5  else if C[side] = ⟨⊥, 2⟩ then
6      execute lines 19–22 of Exit()
```

Procedure Enter(*side*) for process p_i

```
7  C[side] := ⟨i, 1⟩
8  T := i
9  P[i] := 0
10 ⟨rival, ...⟩ := C[other(side)]
11 if rival ≠ ⊥ then
12     if T = i then
13         if P[rival] = 0 then
14             P[rival] := 1
15         await P[i] ≥ 1
16         if T = i then
17             await P[i] = 2
```

Procedure Exit(*side*) for process p_i

```
18 C[side] := ⟨⊥, 2⟩
19 rival := T
20 if rival ∉ {i, ⊥} then
21     P[rival] := 2
22 C[side] := ⟨⊥, 0⟩
```

Algorithm 2: Recoverable extension of Yang and Anderson's two-process mutex.

Our first algorithm is obtained by transforming Yang and Anderson's two-process local-spin mutex [49]. Like Yang and Anderson's algorithm, ours is designed for participation by two out of N processes at a time, meaning that at most two processes at a time execute a super-passage through the mutex. The execution paths of the two processes are distinguished by a special argument $side \in \{\mathsf{left}, \mathsf{right}\}$ of the procedures **Recover**(), **Enter**(), and **Exit**(), which is used internally as the index of an associative array (e.g., see line 1). The rules regarding the values of the argument are two-fold:

1. A process must use the same value of *side* consistently in each super-passage; and
2. If two processes are executing super-passages concurrently then they must use distinct values of *side*.

We will use the notation $side_i$ in reference to the value of the argument *side* for process p_i that is executing a super-passage at the end of a given finite history. The shorthand notation $\mathsf{other}(side)$ defined for convenience in Algorithm 2 denotes the "opposite side" to the one being used by a given process.

The recoverable two-process mutex is presented as Algorithm 2 for process p_i. There are three shared variables: T is used to break symmetry at line 8; C is an associative array used to track the progress of a process at lines 7, 18, and 22; and P is an array of spin variables used for busy-waiting at lines 15 and 17. The entry and exit sections generally follow the structure of Yang and Anderson's algorithm, but there are important differences: (i) the elements of array C, which hold the IDs of processes competing for the CS in the original algorithm, are augmented with an integer tag that indicates progress through a passage; and (ii) an additional write to C is added at line 22 of **Exit**() so that a recovering process can detect whether it crashed in the exit section. In the body of **Enter**(), starting at line 7, a process proceeds directly to the CS unless it encounters a rival at line 11, in which case the busy-wait loop at line 15 ensures that both processes have progressed past line 8, and the second busy-wait loop at line 17 gives priority to the process that executed line 8 the earliest. **Exit**() checks for a rival at line 19, and hands over the critical section at line 21, if required.

The recovery section at lines 1–6 is a new addition, and is executed whenever a process transitions out of the RS. If process p_i is not in cleanup then $C[side(i)] = \langle \bot, 0 \rangle$ holds while it executes **Recover**(), and so the conditions tested at lines 1 and 5 are both false. In that case, p_i proceeds directly to **Enter**(), which ensures bounded recovery. Otherwise, p_i tries to determine where it failed. Lines 1–4 handle a crash inside **Enter**() or the CS, in which case p_i determines at line 2 whether it has a rival p_j. If so, then p_i assigns $P[j] := 2$ at line 4 to ensure that the rival is not stuck at line 15 or 17, and then repeats the entry section. Lines 5–6 handle a crash inside **Exit**(), in which case p_i repeats the body of the exit section and then proceeds to the entry section.

The remainder of this section presents the analysis of Algorithm 2. Theorem 1 asserts the main correctness properties: ME, SF, 0-BR, and $O(1)$ worst-case RMR complexity in the CC and DSM models.

Lemma 2 *Algorithm 2 satisfies mutual exclusion.*

Proof Suppose for contradiction that distinct processes p_i and p_j are in the CS simultaneously at the end of some finite history H. Consider the order in which p_i and p_j execute line 8, where they write their ID into T in their final passages prior to entering the CS. Without loss of generality, suppose that p_j executes line 8 first (the other case is symmetric). Then p_j has already completed line 7 by the time p_i executes line 8. This implies that p_i later reads $\langle j, 1 \rangle$ from $C[\mathsf{other}(side)]$ at line 10. Therefore, p_i subsequently executes lines 12 and 16, where it reads i from T, and hence p_i proceeds to line 17, where it waits for $P[i] = 2$. Since we assume that p_i eventually enters the CS, p_j must write $P[i] := 2$ at some point after p_i resets $P[i] := 0$ at line 9 and before p_i completes line 17. This is a contradiction because p_j completes line 8 before p_i, then completes **Enter**() and remains in the critical section up to the end of H. In particular, p_j does not execute line 4 of **Recover**() or line 21 of **Exit**() after p_i completes line 9, and these are the only two lines where p_j may write $P[i] := 2$. □

Lemma 3 *Let H be a finite history of Algorithm 2. If process p_i is executing a super-passage at the end of H and $C[\mathsf{other}(side_i)] = \langle j, ... \rangle$ then $i \neq j$.*

Proof A process p_i writes its own ID into $C[side_i]$ at line 7, and $side_i$ does not change until p_i completes a super-passage, which entails executing the exit section and erasing i from $C[side]$ at line 18. Since elements of C are initialized to $\langle \bot, 0 \rangle$, this implies that only $C[side_i]$ may contain i while p_i is executing a super-passage. As a result, if $C[\mathsf{other}(side_i)]$ contains the ID $j \neq \bot$ then $j \neq i$. □

Lemma 4 *Let H be a history of Algorithm 2 and let p_i be any process. Let S be any contiguous subhistory of H in which p_i does not execute line 9. Then there is no step in S that decreases the value of $P[i]$.*

Proof In general $P[i]$ only takes on the values 0, 1 and 2. Furthermore, only process p_i may assign $P[i] := 0$, which occurs at line 9. Therefore, if p_i does not execute

line 9 in S then $P[i]$ can only decrease if it transitions from 2 down to 1 by the action of some rival process p_j. This may only happen at line 14, and only after p_j reads $P[i] = 0$ earlier at line 13. Since we assume that $P[i] = 2$ immediately before p_j assigns $P[i] := 1$, it follows that another process p_k assigns $P[i] := 2$ after p_j's read at line 13 and before p_j's write at line 14. This can only occur at line 21 of **Exit**() after p_k reads $T = i$ at line 19. Since the mutex is accessed by at most two processes at a time and since we assume that p_j is at line 14 at this point, either p_j or p_k is the last process to write T, and so p_i must be one of these two processes since $T = i$. It follows from line 20 that $k \neq i$, which implies that $j = i$. Thus, p_j reads its own ID from $C[\mathsf{other}(side_j)]$ at line 10, which contradicts Lemma 3. □

Lemma 5 *Algorithm 2 satisfies starvation-freedom.*

Proof Let H be a fair history with finitely many crash steps, and suppose for contradiction that some process p_i is in the recovery or entry section after some finite prefix of H, and does not subsequently enter the CS. Then in its final passage, p_i becomes stuck at line 15 or at line 17, where it busy-waits on $P[i]$. Without loss of generality, assume that p_i is the first such process to write T at line 8.

Since p_i reaches line 15 or line 17 in its last passage in H, p_i reads the ID of a rival process p_j from $C[\mathsf{other}(side_i)]$ at line 10, where $i \neq j$ by Lemma 3. Then $C[\mathsf{other}(side_i)] = C[side_j] = \langle j, 1\rangle$, and so p_j is either executing a passage or is in the RS following a crash failure. Let s be the first step in the corresponding super-passage of p_j. Then $C[side_j] = \langle \bot, 0\rangle$ holds immediately after s, either by initialization or by the most recent execution of line 22 prior to s by a process that accessed the mutex with the same *side* as p_j's. Subsequently, $side_j$ does not change and either $C[side_j] = \langle \bot, 0\rangle$ or $C[side_j] = \langle j, 1\rangle$ holds until p_j reaches line 18. Since H is fair and contains finitely many crash steps, this implies that p_j eventually executes line 8 where it writes $T := j$. Let w_j be the last write of T at line 8 by p_j after s and before p_j completes a failure-free passage, and let w_i be the last write of T at line 8 by p_i, which occurs in p_i's final passage where it becomes stuck.

Next, we show that p_j does not crash after w_j and before reaching line 18, which assigns $C[side_j] := \langle \bot, 2\rangle$ and erases p_j's ID from $C[side_j]$. Suppose otherwise, and note that on recovery p_j assumes the same *side* as before the failure. Then $C[side_j] = \langle j, 1\rangle$ holds on recovery by p_j's earlier execution of line 7, and continues to hold no matter how many times p_j crashes, until p_j reaches line 18. Since H is fair and contains finitely many crash steps, p_j eventually executes lines 7–8. In other words, p_j writes T again before completing a failure-free passage, which contradicts the definition of w_j. Thus, p_j does not crash before reaching line 18.

Case A: w_j occurs before w_i. Then by our careful choice of p_i as the first process to execute line 8 and then get stuck, p_j either crashes or enters the CS after w_j, but does not become stuck in this passage. As explained earlier, p_j does not crash after w_j and before reaching line 18, and so p_j completes **Enter**() without crashing. From this point onward, p_j proceeds to line 18 of **Exit**(). Thus, p_j eventually assigns $C[side] := \langle \bot, 2\rangle$ at line 18. Subsequently, p_j executes lines 19–22 of **Exit**(), either directly or by crashing, recovering, and executing line 6 of **Recover**(). Now recall from the definition of steps s and w_j that p_i discovers p_j at line 10 in its final passage, and so it follows that p_i executes line 10 before p_j reaches line 18 of **Exit**(), which erases p_j's ID from $C[side_j]$. As a result, w_i occurs before p_j reads T in **Exit**(), which ensures that p_j discovers p_i at line 19. At this point p_i has already executed line 9 for the last time, and so when p_j eventually assigns $P[i] := 2$ at line 21, it follows from Lemma 4 that the value of $P[i]$ does not decrease in the remainder of H. This contradicts p_i busy-waiting forever for $P[i] \geq 1$ at line 15 or $P[i] = 2$ at line 17.

Case B: w_j occurs after w_i. As explained earlier, p_j does not crash after w_j and before reaching line 18. Thus, p_j either reaches line 18 or becomes stuck in **Enter**() without crashing, in contrast to Case A where we know it does not become stuck by our careful choice of p_i. Now consider p_j's execution of **Enter**() following step w_j at line 8.

Subcase B1: p_i is stuck at line 15 waiting for $P[i] \geq 1$. In this case p_i must read $T = i$ at line 12 before w_j, as otherwise w_j overwrites T with $j \neq i$. As a result, p_i assigns $P[i] := 0$ at line 9 also before w_j, and this is the last such write in H. Next, p_j performs w_j at line 8 and reads $T = j$ at line 12 since w_j occurs after w_i. Then, either p_j reads $P[i] \neq 0$ at line 13 or it reads $P[i] = 0$ and then assigns $P[i] := 1$ at line 14. In both cases $P[i] \geq 1$ holds, and the value of $P[i]$ does not decrease in the remainder of H by Lemma 4. This contradicts the hypothesis of Subcase B1.

Subcase B2: p_i is stuck at line 17 waiting for $P[i] = 2$. In this case p_i reads $P[i] \geq 1$ earlier at line 15 after assigning $P[i] := 0$ at line 9, which is the last such write in H. Thus, another process p_k, $k \neq i$, overwrites $P[i]$ with a positive value between p_i's final executions of lines 9 and 15. Let w_k be p_k's write to T at line 8 in this passage. It follows that w_k precedes w_i, as otherwise p_i would not observe $T = i$ at line 16 after completing line 15 and before reaching line 17 because w_i is the last

step in H where p_i assigns $T := i$. Thus, w_k precedes w_j as well under the hypothesis of Case B, and so p_k is either p_j or some process that returns to the RS before p_j begins its passage leading to w_j. Since p_k's write to $P[i]$ occurs after p_i discovers p_j at line 10 in its final passage, it follows in fact that $k = j$ and that w_k is part of p_j's super-passage that starts at step s.

Thus, we have shown that p_j writes T at line 8 at least twice in the super-passage starting at step s: first in w_k, which occurs before w_i, and then in w_j, which occurs after w_i. Since both writes occur in the same super-passage, this implies that p_j crashes outside the RS with $C[side_j] = \langle j, 1\rangle$ or $C[side_j] = \langle \perp, 2\rangle$, and returns to **Enter**() after recovering. Next, consider p_j's write of $P[i]$ in the passage containing step w_k, where we argued that p_j assigns a positive value to $P[i]$. This write must occur at line 14, as otherwise it assigns $P[i] := 2$ and this value does not decrease in the remainder of H by Lemma 4. Moreover, this write occurs between p_i's final executions of lines 9 and 15, hence after p_i's write w_i to T at line 8. Therefore, p_j's passage containing p_j's second write w_j begins after w_i, and also after p_i writes its ID to $C[side_i]$ at line 7. Now consider p_i's execution of **Recover**() in the latter passage, in which $C[side_j] = \langle j, 1\rangle$ or $C[side_j] = \langle \perp, 2\rangle$ holds initially. If p_j recovers with $C[side_j] = \langle j, 1\rangle$ then it reads p_i's ID from $C[side_i]$ at line 2, and then assigns $P[i] := 2$ at line 4. Otherwise, if p_j recovers with $C[side_j] = \langle \perp, 2\rangle$ then p_j executes line 5, which in turn executes lines 19–22 of **Exit**(). Here process p_j reads the ID of p_i from T, and then assigns $P[i] := 2$ at line 19. In both cases, p_j's write to $P[i]$ occurs after p_i's last execution of line 8, and so it follows from Lemma 4 that the value of $P[i]$ does not decrease in the remainder of H, which contradicts the hypothesis of Subcase B2. □

Lemma 6 *Algorithm 2 satisfies wait-free recovery and wait-free exit.*

Proof Since the recovery and exit sections do not contain any loops, it follows that in any history, any execution of **Recover**() and **Exit**() by a process p_i completes in a bounded number of p_i's steps. □

Lemma 7 *Algorithm 2 has worst-case RMR complexity $O(1)$ per passage in the CC and DSM models.*

Proof It suffices to prove that a process incurs $O(1)$ RMRs at lines 15 and 17 because there are no other loops. The lemma follows easily in the DSM model since process p_i only spins on $P[i]$, which is defined at initialization as local to p_i.

In the CC model the spin loops must be analyzed in greater detail to account for possible cache invalidations. The spin loops occur at lines 15 and 17, where p_i waits for $P[i] \geq 1$ and $P[i] = 2$, respectively. A write operation on $P[i]$ by another process p_j can only occur at lines 4, 14, or 21. If such a write occurs at line 14 or line 21 then it ensures that $P[i] = 2$, which holds until p_i's next execution of line 9 by Lemma 4. As a result, p_i's spin loop ends after one RMR to read $P[i] = 2$ into its local cache. On the other hand, if a write of $P[i]$ by another process p_j occurs at line 14 then it ensures that $P[i] \geq 1$, which holds until p_i's next execution of line 9 by Lemma 4. As a result, $P[i]$ is not written at line 14 again until p_i executes line 9 due to the condition $P[i] = 0$ tested at line 13 and because the mutex is accessed by at most two processes at a time. Thus, line 14 contributes at most one RMR to any execution of a spin loop by p_i.

Process, p_i incurs at most two RMRs at line 15: one to read $P[i]$ into its local cache, if $P[i]$ is not already cached after line 15, and at most one due to a write at line 4, 14, or 21. Next, p_i incurs at most three RMRs at line 17: one to read $P[i]$ into its local cache, if $P[i]$ is not already cached after line 15, at most one due to line 14, and one more when p_i first reads $P[i] = 2$ due to a write at line 4 or 21. □

Theorem 1 *Algorithm 2 satisfies ME, SF, WFR, and WFE. Furthermore, its worst-case RMR complexity per passage is $O(1)$ in the CC and DSM models.*

Proof Properties ME, SF, WFR, and WFE follow immediately from Lemmas 2, 5, and 6. The RMR complexity bound is established in Lemma 7. □

3.2 Adding Bounded Critical Section Re-entry

A recoverable mutual exclusion algorithm can be augmented easily with the BCSR property using only $O(N)$ additional shared variables and $O(1)$ additional RMRs per passage in the CC and DSM models. Algorithm 3 presents a transformation that achieves this goal by tracking CS ownership using an array $C[1..N]$ of bits. A process p_i assigns $C[i] := 1$ shortly before entering the CS at line 27, and resets this variable at line 28 shortly after clearing the CS. If p_i crashes in the CS, which implies $C[i] = 1$, then it bypasses the body of the recovery and entry sections in each subsequent passage until it re-enters the CS and reaches the exit section. We apply this transformation to the two-process algorithm from Section 3.1 before using it as a building block of an N-process algorithm in Section 3.3.

The analysis of Algorithm 3 is straightforward with the exception of one detail: the execution path following a failure in the CS omits the recovery and entry sections

```
Shared variables:
  – mtxB: base mutex, recoverable
  – C[1..N]: array of integer, all elements initially zero

Procedure Recover() for process p_i
23 if C[i] = 0 then
24 |  mtxB.Recover()

Procedure Enter() for process p_i
25 if C[i] = 0 then
26 |  mtxB.Enter()
27 |  C[i] := 1

Procedure Exit() for process p_i
28 C[i] := 0
29 mtxB.Exit()
```

Algorithm 3: Transformation from recoverable base mutex to recoverable BCSR target mutex.

of $mtxB$, which goes against the sequential flow of control defined in our model (see Algorithm 1). We first justify this misuse of the base mutex in Lemma 8.

Lemma 8 *For any history H of Algorithm 3, let H' be the projection of H onto steps internal to $mtxB$, as well as all crash steps except where a process p_i fails while $C[i] = 1$ holds. Then H' is a history of the algorithm that implements $mtxB$.*

Proof A process p_i executes the base recovery, entry, and exit sections in H' in the correct order for each passage of the target algorithm, as required, except when $C[i] = 1$ holds upon leaving the RS. The latter situation arises when a process p_i crashes after completing line 26 and before completing line 28. In that case p_i's last access to $mtxB$ is a call to **Enter**() at line 26, and its next access is either a call to **Exit**() if it reaches line 29 and does not crash after resetting $C[i]$ at line 28, or a call to **Recover**() if it completes line 28 and crashes before calling **Exit**() at line 29. In the former case, the correct calling sequence with respect to Algorithm 1 is preserved since all crash steps by p_i occurring while $C[i] = 1$ are omitted from H'. In the latter case, the crash step is recorded in H' and p_i begins a new passage through $mtxB$, as required. □

Theorem 2 *Algorithm 3 satisfies BCSR.*

Proof Suppose that a process p_i crashes in the CS of the target mutex. Then $C[i] = 1$ holds at the time of failure due to line 27, and so p_i completes the target recovery and entry sections in a bounded number of its own steps in each subsequent passage until it re-enters the target CS and then completes line 28. □

Theorem 3 *Algorithm 3 preserves the following correctness properties of $mtxB$: ME, DF, SF, WFR, k-BR, TE, WFE, k-FCFS, and asymptotic worst-case RMR complexity per passage in the CC and DSM models.*

Proof Let H be a history of Algorithm 3, and let H' be the projection defined in the statement of Lemma 8. We make the following observations: (i) H' is a history of $mtxB$ by Lemma 8; (ii) if H is fair then H' is also fair; (iii) if a passage in H' is k-failure-concurrent then all passages in the corresponding super-passage through the target mutex in H are also k-failure-concurrent because H contains all the crash steps of H'. The correctness properties of $mtxB$ are therefore preserved by the following arguments.
ME: The target CS is protected by $mtxB$.
DF and SF: The target algorithm has no loops, and so a process may be stuck on the way to the CS only inside the recovery or entry section of $mtxB$.
WFR and k-BR: These properties follow from the simple structure of the recovery section.
TE and WFE: These properties follow from the simple structure of the exit section. The target algorithm has no loops, and so a process may be stuck in **Exit**() only if it is stuck inside $mtxB$.**Exit**().
k-FCFS: This property follows from the structure of the entry section assuming that the doorway of the target algorithm is defined as the code from the beginning of **Enter**() until the last step of the doorway of $mtxB$ executed at line 26. Note that although the BCSR mechanism may disrupt the order of entry into the CS prescribed by $mtxB$, passages in which a process p_i bypasses the body of **Enter**() with $C[i] = 1$ are 0-failure-concurrent, and hence irrelevant to the definition of k-FCFS for all $k \geq 0$.
RMR complexity: Each passage through the target algorithm entails at most one execution of the base recovery, entry, and exit sections, as well as a constant number of additional steps. The transformation therefore introduces $O(1)$ additional RMRs per passage in the worst case in the CC and DSM models. □

3.3 An Algorithm for N Processes

Our N-process solution is modeled after Yang and Anderson's N-process algorithm [49], which is based on the arbitration tree of Kessels [33]. The algorithm is structured as a binary tree of height $O(\log N)$ where each node is a two-process mutex implemented using the algorithm described in Section 3.1. Each process is mapped statically to a leaf node in the tree as follows: process p_i enters at leaf node number $\lceil i/2 \rceil$ counting

from 1. Furthermore, $side = \mathsf{left}$ at leaf level if i is odd and $side = \mathsf{right}$ if i is even.

Shared variables:
A complete binary tree containing at least $\lceil N/2 \rceil$ and fewer than N leaf nodes, numbered starting at 1, and where each node is an instance of Algorithm 2 from Section 3.1 augmented with BCSR using the transformation from Section 3.2.

Procedure Recover() for process p_i

```
30 (empty)
```

Procedure Enter() for process p_i

```
31 node := leaf node ⌈i/2⌉
32 if i is odd then
33 |  side := left
34 else
35 |  side := right
36 while node is not the root do
37 |  node.Recover(side)
38 |  node.Enter(side)
39 |  if node is a left child then
40 |  |  side := left
41 |  else
42 |  |  side := right
43 |  node := parent of node
```

Procedure Exit() for process p_i

```
44 node := ⊥
45 repeat
46 |  if node = ⊥ then
47 |  |  node := root node
48 |  else
49 |  |  node := child of node on the path to leaf
   |  |  node ⌈i/2⌉
50 |  if node is a leaf and i is odd, or if leaf node ⌈i/2⌉
   |  is in the left subtree of node then
51 |  |  side := left
52 |  else
53 |  |  side := right
54 |  node.Exit(side)
55 until node is a leaf
```

Algorithm 4: Recoverable extension of Yang and Anderson's N-process mutex.

The recoverable N-process mutex is presented as Algorithm 4 for process p_i. Detailed pseudo-code is analogous to the non-recoverable N-process algorithm in [49] except that the two-process instances are implemented using our recoverable two-process mutex. The recovery section is empty since no additional recovery actions are required beyond those performed internally by the two-process mutex instances. The entry section entails executing the recovery and entry section (back to back) of each two-process mutex on the path from the designated leaf node of a process to the root, in that order. The exit section releases the two-process mutex instances in the opposite order, namely from root to leaf.

The remainder of this section presents the analysis of Algorithm 4. Theorem 4 asserts the main correctness properties: ME, SF, WFR, WFE, and $O(\log N)$ worst-case RMR complexity in the CC and DSM models.

As the first step towards proving the correctness properties of the N-process mutex, we first establish that the two-process instances satisfy their safety properties when used inside Algorithm 4 (see Lemma 9). The safety properties of Algorithm 4 will then follow easily, and liveness (SF) will be established by a separate proof in Lemma 13.

Lemma 9 *For any finite history H of Algorithm 4, and for any two-process mutex instance M_2 in the arbitration tree, let H' be the projection of H onto steps executed inside M_2 as well as all crash steps. Then H' is a history of Algorithm 2 augmented with BCSR using Algorithm 3, and hence M_2 satisfies its safety properties (ME, WFR, WFE, and RMR complexity) in H'.*

Proof The safety properties of M_2 in H' follow from Theorem 1 provided that H' is indeed a possible history of Algorithm 2. To that end, we must show that M_2 is accessed correctly by processes, particularly that the following properties hold with respect to super-passages through M_2:

1. at most two processes at a time are executing a super-passage; and
2. if two processes at a time are executing a super-passage then they ascended to the tree node corresponding to M_2 from distinct subtrees; and
3. the *side* parameter used by a process is fixed in any given super-passage; and
4. if two processes are executing super-passages concurrently then their *side* parameters are distinct.

The proof proceeds by induction on the height h of M_2 above leaf level in Algorithm 4, which generates H. In the base case, $h = 0$, at most two processes are mapped statically to M_2, with one process (having the smaller ID) always accessing with $side = \mathsf{left}$ and the other always with $side = \mathsf{right}$. Thus, clauses 1–4 follow immediately. Next, suppose for induction that the lemma holds for all two-process instances up to and including some height $h \geq 0$, and consider an instance M_2 at height $h + 1$, if one exists.
Clause 1 and 2: Each process ascending to M_2 from one of the subtrees occupies the critical section of a mutex at height h at the root of the subtree, and so it follows

from the induction hypothesis, Theorem 1, and Theorem 3 that this critical section is executed in mutual exclusion. Thus, at most two processes at a time (one per subtree of M_2's tree node) may be outside the RS of M_2. This observation and the BCSR property of the mutex at the root of the subtree (see Theorem 2) further ensure that at most two processes at a time (one per subtree) are executing a super-passage through M_2, as required.

Clause 3: Next, consider the *side* of a process p_i executing a super-passage through M_2. Since the path of nodes traversed by p_i through the arbitration tree is determined uniquely by i, it follows that p_i always accesses M_2 with the same *side*: left if it ascends from the left subtree (see lines 40 and 51), and right if it ascends from the right subtree (see lines 42 and 53) of the corresponding tree node.

Clause 4: If processes p_i and p_j execute super-passages through M_2 concurrently then they ascend from distinct subtrees by clause 2. In particular, if $i < j$, then p_i and p_j ascend to M_2 from the left and right subtrees, respectively. As explained in the proof of clause 3, this implies that $side_i = \mathsf{left}$ and $side_j = \mathsf{right}$. Thus, $side_i \neq side_j$. ◻

Lemma 10 *Algorithm 4 satisfies mutual exclusion.*

Proof A process enters the CS of Algorithm 4 only while it is in the CS of the two-process instance at the root node of the arbitration tree. This two-process instance satisfies mutual exclusion by Lemma 9. ◻

Lemma 11 *Algorithm 4 satisfies wait-free recovery and wait-free exit.*

Proof The recovery section of Algorithm 4 is empty, and the exit section performs $O(\log N)$ iterations of a loop that invokes the wait-free exit protocol of Algorithm 2 (see Theorem 1) with $O(1)$ additional steps introduced by the BCSR transformation described in Algorithm 3 (see Theorem 3). ◻

Lemma 12 *Algorithm 4 satisfies bounded critical section re-entry.*

Proof Suppose that process p_i crashes inside the CS in some history H. We must show that p_i incurs a bounded number of steps in each subsequent execution of the recovery and entry sections until it re-enters the CS. The number of steps p_i incurs in **Recover**() is always zero, since Algorithm 4 has a trivial recovery section. The number of steps p_i incurs in each execution of **Enter**() is bounded until p_i re-enters the CS because in that case it is re-entering the CS of every two-process mutex on the bounded path from its designated leaf to the root (even if p_i crashes repeatedly), and each such execution of **Recover**() and **Enter**() of the two-process mutex incurs a bounded number of p_i's steps by the BCSR property (see Theorem 2). Thus, p_i incurs a bounded number of steps in each subsequent execution of **Recover**() and **Enter**() of Algorithm 4 until it re-enters the CS, as required. ◻

Lemma 13 *Algorithm 4 satisfies starvation-freedom.*

Proof Let H be an infinite fair history of Algorithm 4 and suppose that H contains finitely many crash steps. Let p_i be a process that leaves the RS in H. We must show that p_i subsequently enters the CS. Suppose for contradiction that p_i does not, which implies that p_i never completes the super-passage through Algorithm 4 in which we suppose it leaves the RS. Then p_i takes infinitely many steps in H, and in particular it becomes stuck in some two-process mutex instance M_2 in the arbitration tree, as otherwise it would eventually ascend to the CS of Algorithm 4. Let H' be the projection of H onto steps taken inside M_2. Then H' is a history of Algorithm 2 augmented with BCSR using Algorithm 3 by Lemma 9, and it contains finitely many crash steps because it is a subsequence of H. Since p_i is stuck forever in M_2 in H, it also follows that H' is infinite as p_i is equally stuck in H'. To complete the proof, we will show that H' is fair, in which case p_i's lack of progress in H' contradicts the SF property of M_2 (see Theorem 1 and Theorem 3).

Consider any process p_j and suppose that p_j begins a super-passage through M_2 by leaving the RS of M_2 in H'. Since H' is infinite and contains finitely many crash steps, we must show that p_j continues to take steps in H' until it completes this super-passage through M_2 or becomes stuck forever. This certainly holds if $j = i$ since p_i becomes stuck in M_2, so consider $j \neq i$. Anytime p_j begins a passage through M_2, it either completes this passage, becomes stuck, or crashes. Thus, we must show that if p_j crashes then it eventually leaves the RS of M_2 again, possibly after a finite number of additional crash steps, and resumes its ongoing super-passage. The corresponding crash step s in H causes p_j to transition from the recovery, entry or exit section to the RS with respect to Algorithm 4, and also from the CS to the RS with respect to every two-process mutex p_j has acquired during its traversal of the arbitration tree. On recovery, p_j leaves the RS of Algorithm 4, and until it re-enters the CS of Algorithm 4, p_i is able to complete **Recover**() and **Enter**() in a bounded number of steps by the BCSR property of the two-process mutex at each tree node (see Theorem 2) on the bounded path from p_i's leaf to the root. Since H contains finitely many crash steps, this implies that eventually p_j ascends back

to M_2 without crashing, and hence leaves the RS of M_2 in H', as required. □

Lemma 14 *Algorithm 4 has worst-case RMR complexity $O(\log N)$ per passage in the CC and DSM models.*

Proof The lemma follows directly from Lemma 9, the constant worst-case RMR complexity per passage of Algorithm 2 (see Theorem 1), the worst-case RMR complexity preservation property of Algorithm 3 (see Theorem 3), and the logarithmic height of the arbitration tree. □

Theorem 4 *Algorithm 4 satisfies ME, WFR, WFE, BCSR, and SF. Furthermore, its worst-case RMR complexity per passage is $O(\log N)$ in the CC and DSM models.*

Proof Properties ME, WFR, WFE, BCSR, and SF follow immediately from Lemmas 10, 11, 12, and 13. The RMR complexity bound is established in Lemma 14. □

3.4 Discussion of Space Complexity

Burns and Lynch [10] proved a lower bound of $\Omega(N)$ on the space complexity (i.e., the requirement of at least N shared variables) of deadlock-free mutual exclusion algorithms that use only reads and writes. Our N-process algorithm from Section 3.3 as well as Yang and Anderson's algorithm [49], on which ours is based, exceed this bound because they require one spin variable per process per level of the arbitration tree, for a total of $\Theta(N \log N)$ shared variables. The space complexity of Yang and Anderson's algorithm can be reduced to $\Theta(N)$ using a transformation proposed by Kim and Anderson [32]. It is not clear whether the same transformation works correctly for our recoverable algorithm because of the additional execution paths by which a recovering process may overwrite its rival's spin variable, namely at line 4 or at line 21 by way of line 6. In the CC model, our two-process algorithm can be simplified similarly to Yang and Anderson's so that only two spin variables are required per instance, which reduces the space complexity of the N-process solution to $\Theta(N)$.

4 General Solutions

In this section we develop recoverable mutual exclusion algorithms that improve on the N-process solution from Section 3 in terms of RMRs in the absence of failures. Such algorithms necessarily require synchronization primitives other than atomic reads, writes, and Compare-And-Swap (CAS), as otherwise they are subject to the lower bound of $\Omega(\log N)$ for worst-case RMR complexity per passage [6]. In asynchronous failure-free models, the best known solutions in terms of RMRs are queue locks [4] such as Mellor-Crummey and Scott's algorithm (MCS) [39], which has $O(1)$ RMR complexity in the CC and DSM models, and uses both CAS and Fetch-And-Store (FAS) in addition to atomic reads and writes.

The high-level idea underlying our constructions in this section is to augment an ordinary mutex with a recovery procedure that restores the internal structure of the lock, such as the process queue in MCS, in the event of a failure. To our knowledge, the recoverable MCS lock of Bohannon, Lieuwen, and Silberschatz [8] was the first to follow this approach. Their solution makes two assumptions: crash failures are permanent and detectable by the operating system, and the recovery section is executed in a single dedicated process that is itself reliable. That is, the operating system invokes the recovery code automatically when a failure occurs, and there is no need to synchronize multiple processes recovering in parallel. Their algorithm achieves $O(1)$ RMR complexity per passage in the CC and DSM models, but only when the passage does not overlap with the execution of the recovery process, and does not require any synchronization primitives beyond those used in MCS.

Our initial attempt to generalize the approach of Bohannon, Lieuwen, and Silberschatz to crash-recovery failures [45] led to an algorithm that has $O(1)$ RMR complexity in the CC and DSM models in a failure-free passage and $\Theta(N)$ RMRs in the worst case, but requires a powerful Fetch-And-Store-And-Store (FASAS) primitive that is not supported by modern multiprocessors except through transactional memory. Note that [8] and [45] both require $\Omega(N)$ RMRs in the worst case to execute the recovery section alone (ignoring the cost of interacting with the operating system to detect failures in [8]) due to some form of interaction with every other process. In [8], this entails checking if the other processes are attempting to acquire the lock, and in [45], it entails traversing the queue structure to test whether the recovering process is already in it. In comparison, the composition of our constructions in Sections 4.1 and 4.2 transforms the MCS lock (among other algorithms) into a recoverable mutex that incurs $O(1)$ RMRs in the DSM and CC models in a passage that is not 2-failure-concurrent, $O(N)$ RMRs in the DSM and CC models per passage in the worst case, and uses only widely-supported synchronization primitives.

4.1 Transformation for Recoverability

Our first construction, presented in Algorithm 5, is a general transformation of ordinary mutex algorithms (base algorithms) into recoverable ones (target algorithms), and therefore operates in a fundamentally different way from the recoverable MCS locks discussed earlier. Specifically, our construction is agnostic with regard to the internal structure of the base algorithm in that we deal with failures by resetting the base algorithm to its initial state, rather than by attempting delicate repairs. We assume that the base mutex algorithm provides a procedure **Reset**() for this purpose that can be executed by any process as long as no other process is accessing the base mutex concurrently. For example, this method can loop over the objects internal to the base mutex and overwrite each with its initial value.

Resetting the base mutex is a disruptive action that requires careful synchronization between processes executing the recovery section (i.e., processes in cleanup) and other processes that may be accessing the base algorithm. We simplify this task by protecting the core of the recovery section using an auxiliary recoverable mutex, namely the one described in Section 3.3, which incurs additional RMRs for a process in cleanup but does not affect the RMR complexity in the absence of failures. In the critical section of the auxiliary mutex, the process in cleanup first "breaks" the base mutex in a manner that allows other processes to leave the base entry and exit sections in a bounded number of their own steps. It then waits for every other process to either arrive at a gating mechanism that prevents further access to the base mutex, or to crash and recover, in both cases raising a signal by writing a spin variable. Finally, the process in cleanup resets the base mutex, opens the gate, and releases the auxiliary mutex.

Algorithm 5 uses an array $C[1..N]$ of integer variables to record the progress of each process in a super-passage, similarly to the way Algorithm 2 embeds integer tags into elements of its own array C. In the absence of failures, a process executing the target mutex bypasses the recovery section at line 56, executes the body of the target entry section where it acquires the base mutex at line 79, completes the critical section, and finally releases the base mutex in the target exit section at line 82. If a failure occurs, the execution path is steered carefully using the gating mechanism, which is implemented using an array $Gate[1..N]$ of spin variables. The gate is controlled in the core of the recovery section at lines 63–70, which are protected by $mtxA$, the recoverable auxiliary mutex. A process in cleanup first closes the gate at line 63, and then reopens it at line 69. The base mutex is reinitialized at line 68, which is only executed while the gate is closed.

One of the technical challenges in implementing the recovery section of the target algorithm is to suspend access to the base mutex, denoted $mtxB$, without sacrificing liveness. Consider for example the problematic scenario where process p_1 waits inside $mtxB$.**Enter**() for process p_2 to release $mtxB$, then p_2 crashes, closes the gate while in cleanup, and waits for p_1 to release $mtxB$ so that it can be reset safely. To prevent deadlock, executions of $mtxB$.**Enter**() at line 79 and base exit section at line 82 are modified as follows: process p_i repeatedly checks $Gate[i]$ while accessing $mtxB$, and returns immediately to the target entry or exit section if it observes $Gate[i] \neq \bot$, meaning that the gate is closed. In this context, "repeatedly checks" means that p_i reads $Gate[i]$ after executing each step of the base mutex code (or at least after each step of a busy-wait loop). Thus, when the gate is closed by a process in cleanup, p_i is able to leave the base mutex code in a bounded number of its own steps, allowing the recovery section to make progress.

The recovery section only deals with crash failures that occurred outside the RS and CS, excluding crashes at lines 72–76 of the entry section. These scenarios are detected at line 56, including cases where recovery itself was interrupted by a crash. After assigning $C[i] := 4$ at line 57, process p_i reads $Gate[i]$ at line 58 to determine whether the gate is already closed, either by p_i itself prior to failure or by another process in cleanup. If it is closed, then $Gate[i]$ holds the ID of the process in cleanup, and p_i signals this process at line 60. This step is required to prevent deadlock, as explained earlier, if p_i crashed while holding the base mutex. Next, p_i acquires the auxiliary mutex at lines 61–62, closes the gate at line 63, waits at lines 64–67 for other processes to drain out of the base mutex code, resets $mtxB$ at line 68, opens the gate at line 69, and finally releases the auxiliary mutex at line 70.

In the target entry section, process p_i first assigns $C[i] := 1$ at line 72, then checks whether the gate is closed at lines 73–74, and waits at line 76 until the gate is opened, if required. Prior to busy-waiting, p_i signals the process in cleanup at line 75 to prevent deadlock. When the gate is released, p_i updates its status by assigning $C[i] := 2$ at line 77 and then checks the gate again at line 78. If the gate is once again closed, p_i must restart the entry section to ensure that it does not access $mtxB$ at line 79 concurrently with a process in cleanup. Note that checking the gate earlier at line 73 does not suffice for this purpose because at that point p_i has a different status (i.e., $C[i] = 1$ then vs. $C[i] = 2$ now), which is inspected in the recovery section

Shared variables:
- $mtxA$: auxiliary mutex implemented using Algorithm 4 from Section 3.3
- $mtxB$: base mutex
- $C[1..N]$: array of integer, element $C[i]$ local to process p_i in the DSM model, initially zero
- $P[1..N][1..N]$: array of Boolean, elements $P[i][1..N]$ local to process p_i in the DSM model, initially false
- $Gate[1..N]$: array of proc. ID or $\perp$, element $Gate[i]$ local to process p_i in DSM model, initially $\perp$

Procedure Recover() for process p_i

```
56 if C[i] ∉ {0,1} then
57     C[i] := 4
       // signal process in cleanup
58     incleanup := Gate[i]
59     if incleanup ≠ ⊥ then
60         P[incleanup][i] := true
       // clean up the base mutex
61     mtxA.Recover()
62     mtxA.Enter()
       // close the gate
63     for z ∈ 1..N do Gate[z] := i
       // wait for processes to clear base mutex
64     for z ∈ 1..N do
65         P[i][z] := false
66         if z ≠ i ∧ C[z] ∈ {2,3} then
67             await P[i][z] = true
68     mtxB.Reset()
       // reopen the gate
69     for z ∈ 1..N do Gate[z] := ⊥
70     mtxA.Exit()
71     C[i] := 0
```

Procedure Enter() for process p_i

```
72 C[i] := 1
   // wait at gate if needed
73 incleanup := Gate[i]
74 if incleanup ≠ ⊥ then
       // signal process in cleanup
75     P[incleanup][i] := true
       // wait for gate to reopen
76     await Gate[i] = ⊥
77 C[i] := 2
78 if Gate[i] ≠ ⊥ then goto line 72
79 execute steps of mtxB.Enter() interleaved with
   reads of Gate[i] until done or Gate[i] ≠ ⊥
80 if Gate[i] ≠ ⊥ then goto line 72
```

Procedure Exit() for process p_i

```
81 C[i] := 3
82 execute steps of mtxB.Exit() interleaved with reads
   of Gate[i] until done or Gate[i] ≠ ⊥
83 C[i] := 0
   // signal process in cleanup
84 incleanup = Gate[i]
85 if incleanup ≠ ⊥ then
86     P[incleanup][i] := true
```

Algorithm 5: Transformation from N-process base mutex to N-process recoverable target mutex.

at line 66. On the other hand, if the gate is still open at line 78, then p_i executes $mtxB$.**Enter**() at line 79 until it either acquires the base critical section or detects that another process has closed the gate. The latter detection is accomplished by reading $Gate[i]$ after each step of $mtxB$.**Enter**() at line 79. Upon executing $mtxB$.**Enter**() to completion, process p_i checks the gate yet again at line 80 and restarts the entry section of the target mutex if required, which prevents unsafe progress into the target critical section. As we show later on in Lemma 18, executing $mtxB$.**Enter**() repeatedly in the same passage through the target mutex is safe because in that scenario $mtxB$ is reset between two consecutive calls to $mtxB$.**Enter**(). Once p_i enters and leaves the CS of the target mutex, it executes $mtxB$.**Exit**() at line 82 until it either succeeds or detects that another process has closed the gate. The gate is checked after each step of $mtxB$.**Exit**(), as in the earlier execution of $mtxB$.**Enter**(). Process p_i then checks the gate yet again at lines 84–85, and if the gate is closed, p_i signals the process in cleanup at line 86.

The remainder of this section presents the analysis of Algorithm 5. Theorem 5 asserts the main correctness properties, which include preservation of ME and SF with respect to the base algorithm, as well as preservation of RMR complexity in the absence of failures.

One of the difficulties in analyzing Algorithm 5 is that while the gate is closed, $mtxB$ no longer ensures the mutual exclusion property of the target algorithm because a process executing the base entry or exit section does not follow the usual execution path prescribed by the base algorithm. Specifically, a process may break out of the base entry or exit section when it detects that the gate is closed. To facilitate discussion of $mtxB$ in the proofs of correctness, we will first establish several technical lemmas.

Lemma 15 *For any history H of Algorithm 5, and for any processes p_i and p_j, if p_i is at lines 64–69 then $Gate[j] = i$ holds continuously until p_i itself assigns $Gate[j] := \perp$ at line 69, and $Gate[j] = \perp$ holds continuously after p_i does so.*

Proof Process p_i assigns $Gate[z] := i$ at line 63 prior to reaching lines 64–69, and this value can only be overwritten by another execution of line 63 or by an execution of line 69. Such a step can only be applied by p_i itself because lines 63–69 are protected by $mtxA$, which provides BCSR by Theorem 4. □

Lemma 16 *For any history H of Algorithm 5, and for any processes p_i and p_j, if p_i is at line 68, or at line 69 with $Gate[j] \neq \perp$, then p_j is not at lines 79–82.*

Proof Suppose that at the end of H, p_i is at line 68, or at line 69 with $Gate[j] \neq \perp$, and consider process p_j. If

$i = j$ then this follows from the assumption on p_i, so consider $j \neq i$. Observe that earlier in its execution of **Recover**(), p_i assigns $P[i][j] := \mathsf{false}$ at line 65. Since only p_i writes false to this array element and only at line 65, it follows from p_i's completion of lines 64–67 that either p_i reads $C[j] \notin \{2, 3\}$ at line 66, or else some process assigns $P[i][j] := \mathsf{true}$ after p_i completes line 65. The latter assignment can only happen by the action of p_j, namely at line 60, 75, or 86.

Case A: p_i reads $C[j] \notin \{2, 3\}$ at line 66. Furthermore, $Gate[j] = i$ holds continuously from when p_i reads $C[j]$ at line 66 until p_i assigns $Gate[j] := \bot$ at line 69 by Lemma 15. Now consider p_j's steps after p_i reads $C[j] \notin \{2, 3\}$, at which point p_j is not at lines 78-83. No matter what execution path p_j follows, it cannot reach lines 79–82 without first executing line 78. However, p_j cannot progress beyond this line until p_i assigns $Gate[j] := \bot$ at line 69, as otherwise $Gate[j] = i$ holds continuously, causing p_j to branch back to line 72 from line 78.

Case B: p_j executes line 60 after p_i completes line 65. If p_j continues to take steps then it either crashes infinitely often or eventually attempts to execute lines 57–71 of **Recover**() because $C[j] = 4$ holds continuously by line 57 until p_j reaches line 71. If p_j attempts to execute lines 57–71 then it waits for $mtxA$ at line 62 until p_i releases it at line 70. Thus, p_j does not execute lines 79–82 until p_i leaves lines 68–69 entirely.

Case C: p_j executes line 75 or line 86 after p_i completes line 65. No matter what execution path p_j follows, it cannot reach lines 79–82 without first executing line 78 after executing line 75 or line 86. It follows from Lemma 15 that $Gate[j] = i$ holds continuously after p_i completes line 65 until it opens the gate for p_j at line 69, and so p_j cannot progress beyond line 78 until p_i is at line 69 with $Gate[j] \neq \bot$. □

Lemma 17 *For any history H of Algorithm 5, and for any processes p_i and p_j, if p_i crashes at line 68 in some step c of H, then p_j is not at lines 79–82 between c and the next step where p_i returns to line 68 on recovery.*

Proof First, note that $Gate[1..N] = i$ holds immediately before step c by Lemma 15, hence immediately after step c as well. Furthermore, $Gate[1..N] = i$ continues to hold until p_i reaches line 69 on recovery from the crash in step c because all write operations on $Gate$ are protected by $mtxA$, which p_i holds from line 63 to line 70, and because $mtxA$ provides BCSR (Theorem 4). Thus, $Gate[1..N] = i$ holds continuously from immediately before c until p_i reaches line 68 on recovery. Now let p_j be any process and note that p_j is outside of lines 79–82 immediately after c by Lemma 16. As a result, p_j can only reach lines 79–82 after c by completing line 78 first and then proceeding to access $mtxB$ at 79. However, this does not happen because we showed that $Gate[j] = i$ holds continuously until p_i returns to line 68 on recovery. Thus, p_j remains outside of lines 79–82 in this part of the history H, as required. □

Lemma 18 *For any history H of Algorithm 5, let G be any maximal contiguous subhistory of H such that no process is at line 68 immediately before any step of G. Then processes access $mtxB$ in G according to the following rules:*

1. *at the beginning of G, each process is outside of lines 79–82, and in the RS of $mtxB$; and*
2. *if a process p_j breaks out of the base entry or exit section because $Gate[j] \neq \bot$, then p_j does not execute any additional steps of the base entry or exit section of $mtxB$ later on in G; and*
3. *each process executes calls to $mtxB$.***Enter**() *and $mtxB$.***Exit**() *in G in an alternating sequence starting with a call to $mtxB$.***Enter**().

Proof Clause 1: It suffices to show that each process is outside of lines 79–82 at the beginning of G, as this implies that every process is in the RS or $mtxB$. If G begins at the start of H then each process is in the RS of both the target mutex and $mtxB$, hence outside of lines 79–82, as required. Otherwise, G begins immediately after some process p_i completes line 68, or crashes while at line 68. Let s be the step in H by which this happens. Then p_i is at line 68 immediately before s, and so no process p_j is at lines 79–82 by Lemma 16. The same holds immediately after s whether p_i crashes or not, hence at the beginning of G, as required.

Clause 2: Suppose that a process p_j breaks out of executing $mtxB$.**Enter**() at line 79, or $mtxB$.**Exit**() at line 82, because $Gate[j] \neq \bot$. Then some process p_i assigns $Gate[j] := i$ at line 63 earlier in H, and $Gate[j] = i$ holds by Lemma 15 until p_i resets it back to $\bot$ at line 69. However, in that case p_i does not reset $Gate[j]$ in G because it has to complete line 68 between lines 63 and 69. Thus, $Gate[j] = i$ holds continuously in G after p_j breaks out of $mtxB$.**Enter**() or $mtxB$.**Exit**().

If p_j breaks out of $mtxB$.**Enter**() at line 79, then it either branches to line 72 from line 80, or crashes. It both cases, it does not execute $mtxB$.**Enter**() or $mtxB$.**Exit**() again in G due to line 78 because we showed earlier that $Gate[j] = i$ holds until the end of G.

If p_j breaks out of $mtxB$.**Exit**() at line 82, then it does not execute $mtxB$.**Enter**() or $mtxB$.**Exit**() again in the same passage. In subsequent passages, it cannot reach $mtxB$.**Enter**() or $mtxB$.**Exit**() again in G due to line 78 because we showed earlier that $Gate[j] = i$ holds until the end of G.

Clause 3: If no process fails outside the RS of $mtxB$ then clause 3 follows from clauses 1-2 and from the order of lines 79 and 82, except possibly if a process p_j completes $mtxB$.**Enter**() at lines 79 and then branches back to line 72 from line 80 after reading $Gate[j] \neq \bot$. Then some process p_i assigns $Gate[j] := i$ at line 63 earlier in H, and $Gate[j] = i$ holds by Lemma 15 until p_i resets it back to $\bot$ at line 69. As in the analysis of clause 2, this does not happen in G because it requires p_i to complete line 68, and so $Gate[j] = i$ holds continuously in G after p_j reads $Gate[j] \neq \bot$. Then p_j does not access $mtxB$ again in G due to line 78. □

Lemma 19 *For any history H of Algorithm 5, let G be any maximal contiguous subhistory of H such that no process is at line 68 immediately before any step of G. Then the projection of G onto steps of mtxB is a history of the base mutex algorithm, and mtxB satisfies its safety properties (ME, WFE, FCFS) before and after each step of G.*

Proof Case 1: $G = H$. It follows that $mtxB$ is in its initial state at the beginning of G by the initialization of Algorithm 5. Then Lemma 18 implies that G is a history of the base mutex algorithm in which this algorithm satisfies its safety properties.
Case 2: G begins immediately after a process p_i that is in cleanup completes line 68. It follows from Lemma 16 that no process p_j is at lines 79–82 while p_i is at line 68, and so no other process executes steps of the procedures $mtxB$.**Enter**() or $mtxB$.**Exit**() while p_i is at line 68. This implies that the **Reset**() operation p_i executes at line 68 correctly resets $mtxB$ to its initial state immediately before the first step of G, and that $mtxB$ remains in this state at the start of G. Then Lemma 18 implies that G is a history of the base mutex algorithm in which this algorithm satisfies its safety properties, as in Case 1.
Case 3: G begins immediately after a process p_i takes a crash step c while at line 68. Then it follows from Lemma 17 that every process p_j remains outside of lines 79–82 from c, which happens before the beginning of G, until p_i reaches line 68 again, which happens after the last step of G. Furthermore, inside **Recover**(), $mtxB$ can only be accessed at line 68, which can only occur outside of G by construction of G. Thus, G is empty since no process at all accesses $mtxB$ in G. It follows trivially that G is a history of the base mutex algorithm in which this algorithm satisfies its safety properties. □

Lemma 20 *Algorithm 5 satisfies mutual exclusion.*

Proof First, note that to enter the target CS, a process p_ℓ must execute lines 79–80, where $mtxB$ is acquired at line 79 unless $Gate[l]$ becomes closed (i.e., $Gate[l] \neq \bot$). It follows from Lemma 16 that if $Gate[l]$ becomes closed while p_ℓ is executing $mtxB$.**Enter**() then the gate cannot be opened at line 69 while p_ℓ is at lines 79–80. Thus, if p_ℓ completes these lines then it does so with $Gate[l] = \bot$, as otherwise it would branch back to line 72 from line 80. Completion of lines 79–80 therefore implies execution of $mtxB$.**Enter**() at line 79 to completion. In other words, a process p_ℓ that is in the target CS is simultaneously in the CS of $mtxB$.

Now suppose for contradiction that distinct processes p_j and p_k are in the CS of the target algorithm simultaneously at the end of some finite history H. Without loss of generality, assume that $|H|$ is minimal, and observe that by Lemma 16 there can be no process p_i at line 68 at the end of H, or immediately before the last step of H, in which either p_j or p_k enters the CS. Therefore, H has a non-empty suffix G such that no process is at line 68 immediately before any step of G. Let G be maximal so that Lemma 19 applies to the chosen H and G. Then Lemma 19 implies that $mtxB$ provides mutual exclusion after each step of G. This is a contradiction because at the end of G, distinct processes p_j and p_k are not only in the CS of the target algorithm but also in the CS of the $mtxB$, as shown in the first paragraph of this proof. □

Lemma 21 *For any fair history H of Algorithm 5 containing finitely many failures, if a process becomes stuck forever in some passage then this occurs in a single execution of $mtxB$.**Enter**() or $mtxB$.**Exit**().*

Proof Let H be an infinite fair history with finitely many failures. Suppose for contradiction that some process p_i is stuck forever in a passage but not inside a single execution of $mtxB$.**Enter**() or $mtxB$.**Exit**(). Then one of the following cases applies to p_i in its final super-passage in H.
Case 1: p_i is stuck waiting for some process p_j at line 67 of **Recover**(). Since $C[j] \in \{2, 3\}$ during p_i's most recent execution of line 66 for j, p_j has reached line 77 at least once in some step of H. Since H is fair, p_j continues to take steps from this execution of line 77 and onward until it crashes or completes is super-passage. Next, we will show that p_j eventually assigns $P[i][j] :=$ `true` in this super-passage, which contradicts p_i being stuck at line 67.

First, observe that by Lemma 15, $Gate[j] = i$ holds continuously from p_i's most recent execution of line 66 for j prior to it becoming stuck at line 67. This has two implications. First, p_j cannot become stuck forever in $mtxB$.**Enter**() at line 79 or $mtxB$.**Exit**() at line 82. As a result, p_j eventually either crashes or overwrites $C[j]$ with a new value by branching to line 72 from line 78

or line 80 of **Enter**() ($C[j]$ transitions from 2 to 1), by completing line 77 of **Exit**() ($C[j]$ transitions from 2 to 3), or by completing line 83 of **Exit**() ($C[j]$ transitions from 3 to 0). Second, if p_j crashes or overwrites $C[j]$ at line 72, 77, or 83, then $Gate[j] = i$ holds from immediately prior to this step and onward.

Now, consider the steps of p_j after p_i's most recent execution of line 66 for j. If p_j continues to take steps without crashing, then it eventually executes line 72, 77, or 83. In this case all execution paths lead eventually to p_j assigning $P[i][j] := \mathsf{true}$ at line 75 of **Enter**() or at line 86 of **Exit**(). This is because $Gate[j] = i$ holds continuously, which activates the branches at line 78 and line 80 of **Enter**(), and because p_j cannot be stuck in $mtxB$.**Enter**() or $mtxB$.**Exit**() while $Gate[j] = i$.

Otherwise, suppose that p_j crashes after p_i's most recent execution of line 66 for j before reaching a step that assigns $P[i][j] := \mathsf{true}$. Then $Gate[j] = i$ holds immediately after this crash step and onward in H. Each time p_j attempts another passage in the same super-passage, it either follows a path leading to line 60 of **Recover**() in a bounded number of p_j's own steps, or a path that bypasses the body of **Recover**() at line 56 and leads to line 75 of **Enter**(), also in a bounded number of p_j's own steps. Since H is fair and contains finitely many failures, it follows that p_j eventually executes line 60 or line 75 where it assigns $P[i][j] := \mathsf{true}$, as required.

Case 2: p_i is stuck in $mtxA$ at lines 61–62 or line 70 of **Recover**(). We will show that the projection of H onto steps of $mtxA$ is fair, which follows from two points. First, the code between $mtxA$.**Recover**()/**Enter**() at lines 61–62 and $mtxA$.**Exit**() at line 70 is bounded, except for the busy-wait loop at line 67, which terminates eventually by our analysis in Case 1. Second, if some process p_k crashes while outside the RS of $mtxA$, then this occurs with $C[k] = 4$ (see line 57) and so p_k can reach $mtxA$.**Recover**() at line 61 in a bounded number of its own steps on recovery. Thus, if p_j leaves the RS of $mtxA$ in H then it continues to take steps of $mtxA$ until it either completes its passage through $mtxA$ or crashes. In other words, H is fair with respect to $mtxA$. Since H contains finitely many failures, the hypothesis of Case 2 contradicts Theorem 4, which states that $mtxA$ provides wait-free recovery, starvation freedom, and wait-free exit.

Case 3: p_i is stuck waiting for the gate to reopen at line 76 of **Enter**(). Since H is fair and since Cases 1–2 prove that no process may be stuck forever in **Recover**(), it follows that every process p_j that closes $Gate[i]$ at line 63 reopens it eventually at line 69 in the absence of failures. If such a process fails after closing the gate and before reopening it, then $C[j] = 4$ holds by line 57, and so on recovery p_j is confined to lines 57–71 by the branch at line 56 until it completes **Recover**(), even if it fails repeatedly. Since H contains finitely many crash steps, it follows that p_j eventually assigns $Gate[i] := \bot$ at line 69. Thus, $Gate[i] = \bot$ holds continuously in some suffix of H, which contradicts p_i waiting forever at line 76.

Case 4: p_i repeatedly branches back to line 72 from line 78 or from line 80. This implies that $Gate[i] \neq \bot$ holds infinitely often in H, and so either some process p_k closes the gate at line 63 and never reopens it at line 69, or there are infinitely many complete executions of line 63–69. If p_k crashes after assigning $Gate[i] := k$ at line 63 and before assigning $Gate[i] := \bot$ at line 69, then it does so with $C[i] = 4$ by line 57, and so it must complete the body of **Recover**() before progressing to **Enter**() due to line 56, even if it crashes again. Since H is fair and contains finitely many failures, this ensures that p_k eventually completes lines 63–69 because Cases 1 and 2 show that p_k cannot become stuck in **Recover**(). This rules out the possibility that p_k closes the gate for p_i and never reopens it. Since H is fair and contains finitely many failures, it also follows that in some suffix of H, each new execution of **Recover**() by a process p_k begins with $C[k] = 0$ and returns immediately after line 56. This rules out the possibility of infinitely many complete executions of line 63–69 in H. Thus, we reach a contradiction. □

Lemma 22 *Let H be any infinite fair history of Algorithm 5 with finitely many failures. Then H has a suffix S where $mtxB$ begins in its initial state, and satisfies its safety (ME, WFE, FCFS) liveness properties (DF, SF, TE).*

Proof Let H be an infinite fair history of Algorithm 5 with finitely many failures. Consider the maximal suffix G of H after the last crash failure, and let $G = H$ if H is failure-free. Since G is failure-free and since Lemma 21 implies that no process may be stuck forever in **Recover**(), H has a non-empty suffix S in which no process is at lines 56–71 of **Recover**(). In particular, no process is at line 68 immediately before or after any step of S. Assume without loss of generality that S is maximal with respect to the latter property so that Lemma 19 applies, meaning that $mtxB$ satisfies its safety properties (ME, WFE, FCFS) before and after each step of S. Furthermore, note that S is fair with respect to $mtxB$ because Lemma 21 rules out the possibility that a process is stuck forever outside of $mtxB$. Lemma 18 and fairness of S with respect to $mtxB$ imply that $mtxB$ satisfies its liveness properties in S (DF, SF, TE). □

Lemma 23 *Algorithm 5 satisfies deadlock-freedom (respectively starvation-freedom) if $mtxB$ satisfies the same property.*

Proof Let H be an infinite fair history of Algorithm 5 with finitely many failures. Suppose that $mtxB$ satisfies deadlock-freedom (respectively starvation-freedom). Suppose for contradiction that some process p_i is in the recovery or entry section after some finite prefix of H, and no process subsequently enters (respectively p_i does not subsequently enter) the CS of Algorithm 5. This implies that no process subsequently enters (respectively p_i does not subsequently enter) the CS of $mtxB$. It follows from Lemma 21 that p_i may only become stuck in a single execution of $mtxB$.**Enter**() at line 79 or $mtxB$.**Exit**() at line 82, and so p_i is outside the RS of $mtxB$ while it is stuck. In particular, p_i must be stuck in $mtxB$.**Enter**() since we assume that it became stuck before entering the CS of $mtxB$. Thus, p_i is stuck in $mtxB$.**Enter**(), and for each step of p_i in this final execution of $mtxB$.**Enter**(), no process subsequently enters (respectively p_i does not subsequently enter) the CS of $mtxB$.

Now consider the suffix of S of H whose existence is guaranteed by Lemma 22. It follows that $mtxB$ satisfies its deadlock-freedom (respectively starvation freedom) property in S, which contradicts the conclusion of the first paragraph of this proof. □

Lemma 24 *Algorithm 5 satisfies terminating exit (respectively wait-free exit) if $mtxB$ satisfies the same property.*

Proof Terminating exit: Suppose that $mtxB$ satisfies terminating exit. Let H be an infinite fair history of Algorithm 5 with finitely many failures. Suppose for contradiction that some process p_i becomes stuck forever in **Exit**(). This implies that p_i is stuck in $mtxB$.**Exit**() at line 82 since the rest of the target exit section is wait-free. Now consider the suffix of S of H whose existence is guaranteed by Lemma 22. It follows that $mtxB$ satisfies its terminating exit property in S, which contradicts the observation that p_i is stuck forever in $mtxB$.**Exit**().
Wait-free exit: Suppose that $mtxB$ satisfies wait-free exit. Then there exists a function $f(N)$ of the maximum number of processes that bounds the number of steps incurred in any call to $mtxB$.**Exit**(). Let H be any finite history of Algorithm 5 and consider an execution of $mtxB$.**Exit**() by some process p_i, which can only occur at line 82 of **Exit**(). It follows from Lemma 16 that no process is at line 68 while p_i is at line 82. Now let G be the maximal contiguous subhistory of H where the latter property holds and p_i performs its call to $mtxB$.**Exit**() under consideration. Then $mtxB$ satisfies its safety properties before and after each step of G by Lemma 19, including wait-free exit in at most $f(N)$ steps. Since **Exit**() comprises $mtxB$.**Exit**() and $O(1)$ additional steps, it follows that the number of steps p_i takes in **Exit**() is also bounded by a function $f'(N) = f(N) + O(1)$. Thus, the target mutex satisfies wait-free exit. □

Lemma 25 *Algorithm 5 satisfies 0-bounded recovery.*

Proof Let H be any history of Algorithm 5 and consider an execution of **Recover**() by a process p_i. Suppose that p_i is not in cleanup while it executes **Recover**(), meaning that it is either executing its first passage in H, or the next passage following a failure-free passage. In either case, $C[i] = 0$ holds at the beginning of **Recover**(), either by initialization or by p_i's most recent execution of line 83 in **Exit**(). As a result, the condition at line 56 is false and so p_i bypasses the remainder of the recovery section. Thus, p_i completes **Recover**() in a bounded number of its own steps, as required. □

Lemma 26 *Suppose that $mtxB$ in Algorithm 5 has $O(f(N))$ worst-case RMR complexity per passage in the CC and DSM models for some function $f(N)$, and uses V shared variables internally. Then for any history H of Algorithm 5 and any process p_i, the number of RMRs p_i incurs in the CC and DSM models in one passage is as follows:*

- **Recover**()*: $O(N + V)$ if p_i is in cleanup after recovering with $C[i] \notin \{0, 1\}$ at line 56, and $O(1)$ otherwise.*
- **Enter**()*: $O(f(N) \times (1 + F))$ where F denotes an upper bound on the number of passages that begin in cleanup after recovering with $C[i] \notin \{0, 1\}$ at line 56 and are concurrent with p_i's passage.*[7]
- **Exit**()*: $O(f(N))$.*

Proof In **Recover**(), if p_i is in cleanup after recovering with $C[i] \notin \{0, 1\}$ at line 56, then the condition at line 56 is true. Then p_i executes lines 57–71, where it performs $O(N)$ RMRs accessing spin variables and $O(V)$ RMRs resetting $mtxB$ at line 68. Note that in the CC model, each execution of the busy-wait loop at line 67 by p_i takes at most two RMRs, namely one to read $P[i][j] = \mathsf{false}$ and one more to read $P[i][j] = \mathsf{true}$, as only p_i assigns $P[i][j] := \mathsf{false}$ and p_i has already done so at line 65 prior to entering the busy-wait loop. Otherwise, if p_i is not in cleanup, or is in cleanup but after recovering with $C[i] \in \{0, 1\}$ at line 56, then the condition at line 56 is false. In that case p_i completes

[7] The RMR complexity of **Enter**() is unbounded if F does not exist for a given history H.

the recovery section in $O(1)$ steps and hence in $O(1)$ RMRs.

In **Enter**(), process p_i executes lines 72–80 zero or more times, as well as $O(1)$ additional steps. In each iteration of lines 72–80, it completes the busy-wait loop at line 76 at most once, then executes $mtxB$.**Enter**() at line 79 at most once, and executes $O(1)$ other steps. First, we will bound the number of RMRs incurred in $mtxB$.**Enter**(). Note that while p_i is at lines 79–82, no other process can be at line 68 by Lemma 16. Therefore, p_i's steps inside $mtxB$ during the passage under consideration are contained in some non-empty contiguous subhistory G of H such that no process is at line 68 immediately before any step of G. Assume that G is maximal so that Lemma 19 applies. Then it follows that $mtxB$ satisfies its safety properties in G, including worst-case RMR complexity, and so p_i incurs $O(f(N))$ RMRs accessing $mtxB$.**Enter**() at each execution of line 79. It is possible that p_i executes line 79 multiple times due to lines 78 and 80, which requires that $Gate[i]$ is first closed and then reopened so that p_i can progress past lines 74–76. The number of additional executions of line 79 is therefore bounded by the number of concurrent passages that begin in cleanup with $C[i] \notin \{0, 1\}$, in other words F. Thus, p_i incurs $O(f(N) + f(N) \times F)$ RMRs executing line 79. Next, consider the busy-wait loop at line 76. In the DSM model, the loop incurs zero RMRs since $Gate[i]$ is declared local to p_i. In the CC model, the loop incurs at most one RMR every time some process in cleanup writes its ID to $Gate[i]$ at line 63, as well as one more RMR when p_i finally reads $Gate[i] = \bot$. Thus, the total number of RMRs p_i incurs at line 76 across all repetitions of lines 72–80 in one passage is $O(F)$ in the CC model. The RMR complexity of **Enter**() is therefore $O(f(N) \times (1 + F))$, as required.

In **Exit**(), process p_i executes $mtxB$.**Exit**() and $O(1)$ other steps. As in the analysis of **Enter**(), it follows that $mtxB$ meets its RMR complexity guarantee, and hence the cost of $mtxB$.**Exit**() is $O(f(N))$ RMRs. Thus, the target exit section incurs $O(f(N))$ RMRs in total, as required. □

Theorem 5 *Algorithm 5 satisfies ME and 0-BR, and also preserves DF, SF, TE, and WFE. Furthermore, supposing that $mtxB$ has $O(f(N))$ worst-case RMR complexity per passage in the CC and DSM models for some function $f(N)$, and uses V shared variables internally, the number of RMRs a process p_i incurs in one passage through Algorithm 5 is as follows: $O(N + V + f(N) \times (1 + F))$ RMRs if p_i's passage is 1-failure-concurrent, where F denotes an upper bound on the number of passages that begin in cleanup and are concurrent with p_i's, and $O(f(N))$ RMRs otherwise.*

Proof Properties ME and 0-BR follow directly from Lemmas 20 and 25 respectively. Preservation of DF, SF, TE, and WFE follows from Lemmas 23 and 24, respectively. The worst-case RMR complexity in the CC and DSM models follows from Lemma 26. □

We complete the analysis by discussing FCFS fairness in Algorithm 5. On first impression, it may appear that if $mtxB$ provides FCFS then Algorithm 5 does as well, provided that we define the doorway of Algorithm 5 in the natural way: from the first statement of **Enter**() at line 72 to the end of $mtxB$'s doorway at line 79. However, on closer inspection we discover two subtle technicalities. First, the proposed doorway of Algorithm 5 is not always bounded. Second, the end of the doorway is not specified clearly because the target entry section may execute $mtxB$.**Enter**() at line 79 zero times in one passage, such as when a process is recovering from a crash in the CS, or multiple times in one passage, such as when the gate is closed and reopened repeatedly by one or more processes in cleanup. We circumvent both issues by focusing on passages that are not 1-failure-concurrent, and proving 1-FCFS.

Lemma 27 *If the base mutex used by Algorithm 5 satisfies First-Come-First-Served (FCFS) fairness then Algorithm 5 satisfies 1-FCFS.*

Proof Let H be a history of Algorithm 5, and suppose that process p_i enters the CS in its ℓ_i-th passage and p_j enters the CS in its ℓ_j-th passage in H. Suppose that neither passage is 1-failure-concurrent, which implies that the condition at line 56 of **Recover**() holds and that line 79 of **Enter**() is executed exactly once. Suppose for contradiction that p_i completes the doorway in its ℓ_i-th passage before p_j begins the doorway in its ℓ_j-th passage, and yet p_j enters the CS in its ℓ_j-th passage before p_i enters the CS in its ℓ_i-th passage. Consider the contiguous subhistory G of H starting immediately after p_i begins the base doorway in its ℓ_i-th passage and ending in the step where p_i enters the CS in the same passage. Then by Lemma 16, no process is at line 68 immediately before or after any step in G since p_i is at lines 79–82 in G. Now extend G to a maximal contiguous subhistory G' of H such that no process is at line 68 immediately before any step of G'. Then all processes are outside the RS of $mtxB$ at the beginning of G' by Lemma 18, and $mtxB$ satisfies its safety properties before and after each step of G' by Lemma 19. In particular, $mtxB$ provides ME and FCFS. Now consider the order of entry into the base CS versus the order of execution of the base doorway in G' in the passages under consideration. By definition of H, p_i completes the base doorway in its ℓ_i-th passage

through Algorithm 5 before p_j begins the base doorway in its ℓ_j-th passage through Algorithm 5. On the other hand, p_j enters the target CS before p_i enters the target CS, and so p_j enters the base CS before p_i enters the base CS by the ME property of $mtxB$ and since a process that is in the target CS is simultaneously in the base CS. This contradicts the FCFS property of $mtxB$. □

4.2 Transformation for Bounding RMR Complexity

The construction presented earlier in Section 4.1, when instantiated with a local-spin base mutex, has bounded RMR complexity in failure-free histories. However, the number of RMRs a process executes per passage in the worst case may be arbitrarily large as it depends on the number of crash failures in the history. In this section, we describe an additional transformation that bounds RMR complexity in the worst case, and in the absence of failures matches (up to a constant factor) the RMR complexity of the base mutex internal to Algorithm 5. Our technique is inspired by two ideas: Scott and Scherer's *abortable mutual exclusion* [47], in which a process that is waiting in the entry section may give up its interest in the critical section; and Lamport's *fast path* mechanism [37], which improves the efficiency of a mutex algorithm in the absence of contention.

The new transformation, illustrated conceptually in Figure 3, uses the construction from Section 4.1 (Algorithm 5) as its base mutex. By default, processes use the base mutex, denoted $mtxC$, to protect the target critical section. However, executions of $mtxC$.**Enter**() must be terminated early if another process in cleanup is detected as otherwise the RMR complexity of the target entry section would be unbounded. This is accomplished by breaking out of $mtxC$.**Enter**(), similarly to how a process gives up interest in the critical section in an abortable mutual exclusion algorithm by executing an abort protocol. Such a process is diverted away from the default path and into a *bounded path* that guarantees bounded RMR complexity. The bounded path is slower than the default path in the absence of contention, and so it is engaged only when needed.

To ensure mutual exclusion, both the bounded path and the default path are protected by an auxiliary recoverable mutex implemented using the algorithm from Section 3.3. In the absence of failures, the auxiliary mutex is accessed only along the default path where it is already protected by the base mutex, and so contention is minimal. Therefore, augmenting the auxiliary mutex with a suitable fast path ensures that the target mutex has the same RMR complexity asymptotically as the base mutex in the absence of failures.

Breaking out of $mtxC$.**Enter**() is a delicate operation as it can lead to a penalty in terms of RMRs when the process executes $mtxC$.**Recover**() during its next passage through the target algorithm. We avoid this by leveraging a feature of Algorithm 5 stated earlier in Lemma 26: the number of RMRs a process p_i incurs while executing $mtxC$.**Recover**() in a passage that begins with $C[i] = 1$ (i.e., p_i is in cleanup after a crash early on in $mtxC$.**Enter**()) is the same as in a passage that begins with $C[i] = 0$ (i.e., p_i is in cleanup after a crash late in $mtxC$.**Exit**() or p_i is not in cleanup at all). Thus, the RMR penalty due to execution of $mtxC$.**Recover**() can be avoided entirely if p_i breaks out of $mtxC$.**Enter**() before line 77, where it assigns $C[i] := 2$. An explicit abort protocol is not required to give up interest in the target CS, and therefore our analysis does not refer to abortability.

Adding a fast path to the recoverable auxiliary mutex is more challenging as the natural approach of generalizing a known fast path implementation to crash-recovery failures leads to some undesirable technicalities. Lamport's use of the fast path [37], for example, leads to unbounded RMR complexity. Yang and Anderson's adaptation of Lamport's mechanism to local-spin algorithms [49] bounds the RMR complexity but only at $\Theta(N)$ in the worst case. Finally, Anderson and Kim's mechanism [2] bounds the RMR complexity at $\Theta(\log N)$ in the worst case, but introduces identifiers that grow without bound. We avoid these issues altogether by using Compare-And-Swap (CAS) to open and close the fast path instead of relying on atomic reads and writes only. This is a reasonable design choice because the two transformations presented in Section 4 are intended to be used with RMR-efficient base algorithms that already rely on atomic read-modify-write primitives. Note that the RMR-efficient simulation of CAS from reads and writes of Golab, Hendler, Hadzilacos, and Woelfel is not applicable in this context as it assumes reliable processes [17].

The pseudo-code for the target algorithm is presented in Algorithm 6. The construction uses three mutexes as building blocks: $mtxC$ is the base mutex obtained by instantiating the construction from Section 4.1; $mtxA_N$ is the N-process auxiliary mutex that protects the critical section in the default and bounded paths, as explained earlier; and $mtxA_2$ is an additional two-process auxiliary mutex used to synchronize $mtxA_N$ with the fast path. Both $mtxA_N$ and $mtxA_2$ are implemented using the algorithms presented earlier in Section 3. The forthcoming analysis also refers in some places to the base mutex internal to Algorithm 5 (i.e., $mtxB$), which affects the RMR complexity of $mtxC$ but is not referenced explicitly in Algorithm 6.

Shared variables:
- $mtxA_2$: two-process recoverable mutex implemented using Algorithm 2 from Section 3.1, and augmented with BCSR using Algorithm 3 from Section 3.2
- $mtxA_N$: N-process recoverable mutex implemented using Algorithms 4 from Section 3.3, respectively, and augmented with BCSR using Algorithm 3 from Section 3.2
- $mtxC$: recoverable base mutex implemented using Algorithm 5 from Section 4.1, and augmented with BCSR using Algorithm 3 from Section 3.2
- F: holds process ID or $\bot$, initially $\bot$

Private variables:
- $Bounded[1..N]$: array of Boolean, initially all elements false
- $Breakout[1..N]$: array of Boolean, element i private to process p_i, initially all elements false

Procedure Recover() for process p_i

```
87  mtxC.Recover()
```

Procedure Enter() for process p_i

```
88  if Breakout[i] = false then
89      execute steps of mtxC.Enter(), break out
        immediately after the second completion of
        lines 72–76 of Algorithm 5
90      if broke out of mtxC.Enter() at line 89 then
91          Breakout[i] := true
92      else
93          Breakout[i] := false
    // try to capture the fast path using CAS
94  if Bounded[i] = false ∧ CAS(F, ⊥, i) ∈ {⊥, i} then
95      mtxA2.Recover(left)
96      mtxA2.Enter(left)
97  else
98      Bounded[i] := true
99      mtxAN.Recover()
100     mtxAN.Enter()
101     mtxA2.Recover(right)
102     mtxA2.Enter(right)
```

Procedure Exit() for process p_i

```
103 if F = i then
104     mtxA2.Exit(left)
        // release the fast path
105     F := ⊥
106 else
107     mtxA2.Exit(right)
108     mtxAN.Exit()
109     Bounded[i] := false
110 if Breakout[i] = false then
111     mtxC.Exit()
112 else
113     Breakout[i] := false
```

Algorithm 6: Transformation for bounding worst-case RMR complexity in the algorithm from Section 4.1.

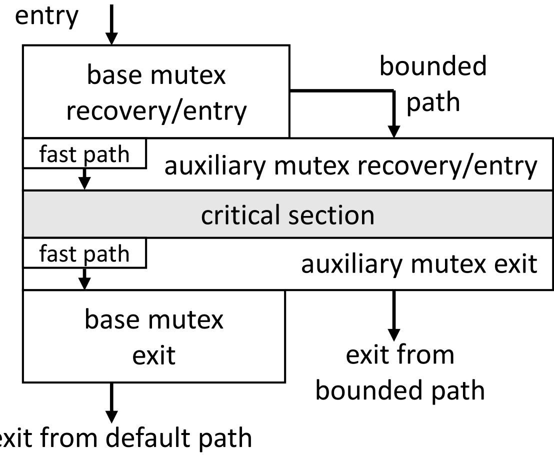

Fig. 3 Conceptual structure of the target algorithm with default (left) and bounded (right) paths.

The *default path* in Algorithm 6 refers to the pseudocode statements executed in a failure-free history. In the absence of failures, a process p_i begins by executing $mtxC$.**Recover**() at line 87 of **Recover**(), then executes $mtxC$.**Enter**() to completion at line 89 of **Enter**(), and then attempts to steer around the auxiliary mutex $mtxA_N$ by executing a Compare-And-Swap operation (denoted CAS) at line 94.[8] The code after an execution of line 94 in **Enter**() where the CAS succeeds (or $F = i$ holds already prior to the CAS) and $Bounded[i]$ = false holds, and before the write operation at line 105 in **Exit**(), is called the *fast path* of Algorithm 6.

Since the base mutex protects lines 94–110 in the absence of failures, there is no contention on the fast path and so p_i successfully swaps its ID into F. Next, p_i acquires $mtxA_2$ at lines 95–96, completes the target critical section, releases $mtxA_2$ at line 104, and releases the fast path by overwriting F with $\bot$ at line 105. Finally, p_i releases $mtxC$ at line 111. Thus, the fast path entails acquiring $mtxC$, F, and the auxiliary mutex $mtxA_2$, and then releasing all three components in the opposite order.

In the presence of failures by other processes, p_i may break out of $mtxC$.**Enter**() at line 89. This event is recorded in $Breakout[i]$ at line 91 so that p_i remembers not to execute $mtxC$.**Exit**() later on at line 111, as that would be unsafe given the incomplete execution of $mtxC$.**Enter**(). (On the other hand, it is safe for p_i to execute $mtxC$.**Recover**() in its next passage through the target mutex, similarly to the case when p_i crashes in $mtxC$.**Enter**() instead of breaking out volun-

[8] The "∧" operator at line 94 should be interpreted like && in C++, meaning that the right operand is evaluated only if the left operand is true.

tarily.) Moreover, p_i may skip $mtxC$.**Enter**() entirely (see line 88), which is necessary to avoid deadlock in the scenario where p_i breaks out and then crashes after acquiring $mtxA_2$, while another process cleans up and acquires $mtxC$. The rest of the target entry and exit sections is executed similarly to the failure-free case, except that $mtxC$ no longer protects attempts to acquire the fast path, leading to the possibility that p_i fails to capture the fast path at line 94 due to contention. In that case, p_i acquires $mtxA_N$ at lines 99–100, acquires $mtxA_2$ at lines 101–102, completes the target critical section, releases $mtxA_2$ at line 107, and finally releases $mtxA_N$ at line 108. The code after an execution of line 94 in **Enter**() where the CAS fails or $Bounded[i]$ = true holds, and before the write operation at line 109 in **Exit**(), is called the *bounded path* of Algorithm 6. The variable $Bounded[i]$ is used at lines 98 and 109 to record p_i's presence in the bounded path for recovery.

If p_i itself crashes, then on recovery it executes **Recover**() and **Enter**() up to line 94, as described earlier. If p_i failed inside the bounded path (i.e., $Bounded[i]$ = true), then by the test at line 94 it proceeds to line 98 where it re-enters the bounded path. In this case, p_i cannot enter the fast path at line 94 as otherwise it will acquire $mtxA_2$ with $side$ = left at lines 95–96 before it has a chance to recover its prior passage through $mtxA_2$ from lines 101–102 with $side$ = right, breaking the assumptions stated in Section 3 for accessing the two-process mutex. Otherwise, if p_i failed outside the bounded path (i.e., $Bounded[i]$ = false), then it attempts to capture the fast path, and falls back on the bounded path only if necessary. If p_i was already in the fast path when it crashed, then $F = i$ holds and the response of the CAS at line 94 indicates that it is safe for p_i to re-enter the fast path. After completing the target critical section, p_i releases the auxiliary mutexes and completes the fast path in the target exit section at lines 103–109.

The RMR complexity of the target algorithm in the absence of failures is dominated by $mtxC$ since the auxiliary two-process mutex used in the fast path ($mtxA_2$) requires only $O(1)$ RMRs. Failures affect RMRs in two ways: (i) the complexity of the base recovery section may increase, which affects line 87; and (ii) breakouts at line 89 raise pressure on the fast path and deflect processes into the bounded path, which incurs an additional $\Theta(\log N)$ RMRs per passage. Note that (i) applies only while some process is in cleanup with respect to both $mtxC$ and the target algorithm. In contrast, contention on the fast path in scenario (ii) lingers on until the last process drains out of the bounded path, which could occur long after $mtxC$ has been cleaned up. As a result, (ii) may affect the RMR complexity of any passage that is 1-failure-concurrent or interferes with a 1-failure-concurrent passage (i.e., any 2-failure-concurrent passage).

The remainder of this section presents the analysis of Algorithm 6. Theorem 6 asserts the main correctness properties of the construction. We begin the analysis by establishing in Lemmas 28–29 that the building blocks of the target algorithm are accessed correctly.

Lemma 28 *For any history H of Algorithm 6, at most one process at a time is in the fast path.*

Proof The fast path is defined as the code after a successful CAS operation at line 94 in **Enter**() and before the write operation at line 105 in **Exit**(). A successful CAS by p_i at line 94 is a transition of F from $\perp$ to i, and a write operation at line 105 is a transition of F from i to $\perp$. Thus, if p_i is in the fast path then $F = i$ holds continuously, which implies that at most once process at a time can be in the fast path. □

Lemma 29 *For any history H of Algorithm 6, the following hold:*

1. *the projection H_N of H onto crash steps and steps executed inside $mtxA_N$ is a history of Algorithm 4; and*
2. *the projection H_2 of H onto crash steps and steps executed inside $mtxA_2$ is a history of Algorithm 2; and*
3. *the projection H_C of H onto crash steps and steps executed inside $mtxC$, with an additional crash step by process p_i inserted each time process p_i either bypasses $mtxC$.***Enter**() *at line 88 or breaks out of $mtxC$.***Enter**() *at line 89, is a history of Algorithm 5.*

Proof For each clause, we must show that processes follow the execution path described in Algorithm 1 with respect to the given recoverable mutex in the target algorithm. Furthermore, in clause 1 we must show that the two-process mutex is accessed by at most two processes at a time, and according to the rules stated in Section 3.1 regarding the special argument $side$ passed to the procedures **Recover**(), **Enter**() and **Exit**().
Clause 1: Only the bounded path of Algorithm 6 accesses $mtxA_N$. Executions of $mtxA_N$.**Recover**() and $mtxA_N$.**Enter**() at lines 99–100 are matched by a call to $mtxA_N$.**Exit**() at line 108. Furthermore, if a process p_i crashes outside the RS of $mtxA_N$, then its next access to this mutex in H (if such an access occurs) is a call to $mtxA_N$.**Recover**() at line 99, as required. Thus, $mtxA_N$ is accessed correctly in H, and so H_N is a history of Algorithm 4, as required.
Clause 2: There are several points in Algorithm 6 where $mtxA_2$ is accessed. Executions of $mtxA_2$.**Recover**()

and $mtxA_2$.**Enter**() at lines 95–96 and lines 101–102 are matched by calls to $mtxA_2$.**Exit**() at lines 104 and 107, respectively. Furthermore, $mtxA_2$ is only accessed by two processes at a time: one using $side$ = left in the fast path at lines 95–96 and 104, where operations on F at lines 94–105 guarantee mutual exclusion (see Lemma 28), and one using $side$ = right in the bounded path at lines 101–102 and 107, where $mtxA_N$ guarantees mutual exclusion because it is accessed correctly at lines 99–108 (see clause 1). Also note that the ID of a process p_i executing the "left" side in the fast path does not change, even in the event of failure inside the fast path, until this process completes its super-passage through $mtxA_2$. This is because no other process is able to enter the fast path until p_i recovers with $Bounded[i]$ = false, re-enters the fast path at line 94 when the CAS operation returns p_i's ID, and finally releases the fast path at line 105. The analogous statement holds for the process executing the "right" side in the bounded path because lines 101–102 and 107 are protected by $mtxA_N$, which satisfies its safety properties because it is accessed correctly by clause 1, and which we assume is augmented with BCSR using Algorithm 3 from Section 3.2. Thus, $mtxA_2$ is accessed correctly in H, and so H_2 is a history of Algorithm 2, as required.

Clause 3: There are four places where $mtxC$ is accessed in Algorithm 6: lines 87, 89, and 111. In the absence of failures, every call to $mtxC$.**Recover**() at line 87 is matched by a call to $mtxC$.**Enter**() at 89 and a call to $mtxC$.**Exit**() at 111, as required. If a process p_i breaks out of line 89, then it assigns $Breakout[i]$ = true at line 91, and later bypasses the exit section of $mtxC$ at lines 110–111. Process p_i further bypasses the entry section of $mtxC$ at line 88 upon recovering from a crash with $Breakout[i]$ = true, and continues to bypass the entry and exit sections until it completes a failure-free passage by writing $Breakout[i]$:= false at line 113. In that case, the projection defined in clause 3 inserts a crash step for process p_i when it bypasses or breaks out of $mtxC$.**Enter**(), which completes a passage through $mtxC$ and makes it safe for p_i to call $mtxC$.**Recover**() again in its next passage through the target algorithm. The crash step is introduced for analysis only, and preserves the execution pattern shown in Algorithm 1 with respect to $mtxC$ even though p_i does not actually crash in the code of the target mutex. Thus, H_C is a history of Algorithm 5, as required. □

Lemma 30 *Algorithm 6 satisfies mutual exclusion.*

Proof The critical section of the target algorithm is protected by $mtxA_2$, which ensures mutual exclusion by Lemma 29 and Theorem 1. □

Lemma 31 *For any fair history H of Algorithm 6 containing finitely many failures, if a process becomes stuck forever in some passage then this occurs in a single execution of $mtxB$.**Enter**() or $mtxB$.**Exit**() in Algorithm 5, which implements $mtxC$ in Algorithm 6.*

Proof Let H be an infinite fair history of Algorithm 6 with finitely many crash steps. Suppose that a process p_i becomes stuck forever in some passage in H. Since Algorithm 6 does not contain any loops, it suffices to show that p_i does not become stuck inside $mtxA_2$ or $mtxA_N$, or inside $mtxC$ except while accessing $mtxB$ at line 79 or line 82 of Algorithm 5. We proceed by a case analysis.

Case 1: no process becomes stuck in $mtxA_2$. Let H_2 be the projection of H onto crash steps and steps of $mtxA_2$, as defined in Lemma 29. In the absence of failures, a process p_i leaves the RS of $mtxA_2$ at lines 95–96 or 101–95, where it executes $mtxA_2$**Recover**() and $mtxA_2$.**Enter**(), then proceeds directly to the CS of the target algorithm, which terminates eventually since H is fair, and finally executes $mtxA_2$.**Exit**() at line 104 or 107. It remains to show that if p_i crashes along this path in H then it eventually recovers and reaches $mtxA_2$.**Recover**() again in H_2. Proving this point establishes that H_2 is fair, and completes Case 1 since no process may be stuck in $mtxA_2$ given that this algorithm provides starvation-freedom and wait-free exit (see Theorem 1).

Subcase 1a: p_i crashes in the fast path outside the RS of $mtxA_2$. Then $F = i$ and $Bounded[i]$ = false hold at the point of failure by p_i's prior execution of line 94. Therefore, p_i is restricted to taking the following actions on recovery until it reaches $mtxA_2$.**Recover**() at line 95, no matter how many times it crashes again: (i) p_i calls $mtxC$.**Recover**() at line 87 and some (possibly empty) prefix of $mtxC$.**Enter**() at line 89; and (ii) p_i reaches line 94 and re-enters the fast path because $F = i$ and $Bounded[i]$ = false continue to hold, causing the CAS operation to return i. Note that in scenario (i), p_i's steps in the recovery and entry sections of $mtxC$ are bounded. If p_i's crash occurred in the CS of $mtxC$ with $Breakout[i]$ = false, this is because p_i can re-enter the CS of $mtxC$ in a bounded number of its own steps by the BCSR property of $mtxC$, which holds by the application of Algorithm 3 and by Lemma 29. Otherwise, p_i's crash occurred after breaking out of $mtxC$ with $Breakout[i]$ = true in Algorithm 6 and $C[i] = 1$ in Algorithm 5, and so p_i subsequently skips the body of the recovery section of $mtxC$ (see line 56 of Algorithm 5), and then skips the entry section of $mtxC$ (see line 88 of Algorithm 6). Thus, p_i has an unobstructed path to $mtxA_2$.**Recover**() in scenarios (i) and (ii). Since H contains finitely many crash steps, H_2 does as well, and

so eventually p_i reaches $mtxA_2$.**Recover**() at line 95, as required.

Subcase 1b: p_i crashes in the bounded path outside the RS of $mtxA_2$. Then $Bounded[i]$ = true holds at the point of failure by p_i's prior execution of line 94 and line 98. Therefore, p_i is restricted to taking the following actions on recovery until it reaches $mtxA_2$.**Recover**() at line 101, no matter how many times it crashes again: (i) p_i calls $mtxC$.**Recover**() at line 87 and some prefix of $mtxC$.**Enter**() at line 89 in a bounded number of its own steps, as in Subcase 1a; (ii) p_i reaches line 94 and fails to enter the fast path again because $Bounded[i]$ = true continues to hold; and (iii) p_i re-enters the CS of $mtxA_N$ at lines 99–100 in a bounded number of its own steps since $mtxA_N$ provides BCSR by the application of Algorithm 3 and by Lemma 29. Since H contains finitely many crash steps, H_2 does as well, and so eventually p_i reaches $mtxA_2$.**Recover**() at line 101, as required.

Case 2: no process becomes stuck in $mtxA_N$. The analysis is similar to $mtxA_2$ in Case 1. Let H_N be the projection of H onto crash steps and steps of $mtxA_N$, as defined in Lemma 29. In the absence of failures, process p_i leaves the RS of $mtxA_N$ at lines 99–100, where it calls $mtxA_N$.**Recover**() and $mtxA_N$.**Enter**(), then begins a passage through $mtxA_2$ at lines 101–107, which terminates eventually by Case 1 and since the CS of the target algorithm terminates eventually, and finally calls $mtxA_N$.**Exit**() at line 108. It remains to show that if p_i crashes along this path in H then it eventually reaches $mtxA_N$.**Recover**() at line 99 again in H_N. Proving this point establishes that H_N is fair, and completes Case 2 since no process may be stuck in $mtxA_N$ given that this algorithm provides starvation-freedom and wait-free exit.

If p_i fails outside the RS of $mtxA_N$ then it follows that $Bounded[i]$ = true holds at the point of failure by p_i's prior execution of line 98. Therefore, p_i is restricted to taking the following actions on recovery until it reaches $mtxA_N$.**Recover**() at line 99: (i) p_i calls $mtxC$.**Recover**() at line 87 and some prefix of $mtxC$.**Enter**() at line 89 in a bounded number of its own steps, as in Subcase 1a and Subcase 1b; and (ii) p_i reaches line 94 and fails to acquire the fast path because $Bounded[i]$ = true continues to hold. Since H contains finitely many crash steps, H_N does as well, and so eventually p_i reaches $mtxA_N$.**Recover**() at line 99, as required.

Case 3: no process becomes stuck in $mtxC$ except while accessing $mtxB$ at line 79 or line 82 of Algorithm 5. Let H_C be the projection of H onto steps of $mtxC$, as defined in Lemma 29. In the absence of failures, process p_i leaves the RS of $mtxC$ at line 87, where it calls $mtxC$.**Recover**(), and then follows one of two execution paths. If p_i executes a call to $mtxC$.**Enter**() completely at line 89 then it assigns $Breakout[i]$:= false at line 93. Next, p_i executes lines 94–109, which terminate eventually by Cases 1 and 2 and since CS of the target algorithm terminates eventually. Finally, p_i executes lines 110–111 where it calls $mtxC$.**Exit**(). Otherwise, if p_i breaks out of $mtxC$.**Enter**() at line 89, then it assigns $Breakout[i]$:= true at line 91. In this case p_i does not access $mtxC$ again in the same passage through the target mutex because it executes lines 94–109 and then bypasses $mtxC$.**Exit**() at lines 110–111. Accordingly, H_C records a crash step for p_i in this scenario, allowing p_i to continue in $mtxC$.**Recover**() at line 87. Similarly, if p_i bypasses $mtxC$.**Enter**() at lines 88–89, then this occurs with $Breakout[i]$ = true, and so p_i also bypasses $mtxC$.**Exit**() at lines 110–111 in the same passage. As in the case when p_i breaks out of $mtxC$.**Enter**(), H_C records a crash step for p_i when it bypasses $mtxC$.**Enter**(). Finally, if p_i fails after breaking out of, or bypassing, $mtxC$.**Enter**(), then it reaches $mtxC$.**Recover**() at line 87, as required, the next time it takes a non-crash step.

So far, the analysis of Case 3 has shown that H_C is fair except possibly in cases where a process halts after completing a failure-free passage through the target mutex in H and where the corresponding passage through $mtxC$ is incomplete and not failure-free, hence requiring another call to $mtxC$.**Recover**(). This is a consequence of introducing a crash step in H_C each time a process breaks out of, or bypasses, $mtxC$.**Enter**() in Algorithm 6, which also means that the finite number of failures in H does not immediately imply that H_C contains finitely many failures. Thus, we cannot complete the analysis by applying Lemma 21 directly to H_C, even though it is a history of the algorithm that implements $mtxC$ by Lemma 29. However, the conclusion of Lemma 21 applied to H_C, namely that p_i must be stuck in $mtxB$.**Enter**() or $mtxB$.**Exit**(), still holds. We arrive at this observation by a re-examination of the proof of Lemma 21, and how it is affected by the additional crash steps inserted in the construction of H_C from H.

Recall the case analysis in the proof of Lemma 21, and consider how the additional crash steps in H_C affect the progress of a process p_i in a passage of $mtxC$. In Case 1, p_i cannot become stuck waiting at line 67 of $mtxC$.**Recover**() for a process p_z that executes one of the additional crash steps, because the crash occurs after p_z reads $Gate[z] = \perp$ at line 73 or line 76, which causes p_z to break out of $mtxC$. This holds even when p_z bypasses $mtxC$.**Enter**() with $Breakout[z]$ = true because it must first break out of $mtxC$ and then assign

$Breakout[z]$:= `true` at line 91. Reading $Gate[z] = \perp$ indicates that the process holding the auxiliary mutex in $mtxC$.**Recover**() has reached line 69 already after closing the gate earlier at line 63, and hence completed the busy-wait loop line 67. In Case 2, p_i cannot become stuck waiting inside $mtxA$ for such a process p_z because p_z crashes in the RS of this auxiliary mutex. (Also, Case 1 rules out the possibility of a process becoming stuck in the CS of the auxiliary mutex.) In Cases 3-4, p_i cannot become stuck in $mtxC$.**Enter**() waiting for such a p_z to open the gate for p_i because p_z does not have a pending call to $mtxC$.**Recover**() when it crashes, and so p_z did not close the gate in the first place. Thus, p_i can only be stuck in $mtxC$ if it is stuck specifically in $mtxB$.**Enter**() or $mtxB$.**Exit**() in Algorithm 5, as required to conclude Case 3 in this proof. □

Lemma 32 *Let H be an infinite fair history of Algorithm 6 with finitely many crash steps. Let H_C be the projection of H defined in Lemma 29. Then H_C has a suffix S satisfying the following properties:*

1. *the projection of S onto steps of mtxB in Algorithm 5 (i.e., the base mutex internal to mtxC) is a fair history of the mutex algorithm that implements mtxB; and*
2. *for every process p_i that is stuck in a passage of Algorithm 6 in H, p_i is also stuck in a passage of mtxB in S.*

Proof Let H be an infinite fair history of Algorithm 6 with finitely many crash steps. Let H_C be the projection of H defined in Lemma 29. Then H_C is a history of Algorithm 5 by Lemma 29, though not necessarily one that is also fair and contains finitely many crash steps. It follows that lines 58–71 of $mtxC$.**Recover**() are executed finitely many times in H_C because line 58 is reached in the code of $mtxC$ only when a process p_i begins $mtxC$.**Recover**() with $C[i] \notin \{0, 1\}$, which occurs only when p_i crashes in H, and never when p_i breaks out of, or bypasses, $mtxC$ in Algorithm 6. (The breakout is from a position in $mtxC$.**Enter**() where $C[i] = 1$ holds, and continues to hold if $mtxC$.**Enter**() is bypassed.) Therefore, there is a well-defined last execution of line 68 of Algorithm 5 in H and H_C. Now let S be the suffix of H_C beginning immediately after this last execution of line 68. We will show that S has the properties claimed in the lemma.

<u>Clause 1.</u> Let S_B be the projection of S onto steps of $mtxB$. Then S_B is a history of $mtxB$ by Lemma 19. Furthermore, since a process may only become stuck in H_C inside $mtxB$.**Enter**() or $mtxB$.**Exit**() by Lemma 31, it follows that S_B is fair, as required.

<u>Clause 2.</u> Suppose that p_i is stuck in a passage of Algorithm 6 in H. Since H is fair, p_i is stuck in a single execution of $mtxB$.**Enter**() or $mtxB$.**Exit**() in Algorithm 5 by Lemma 31. Since S records a suffix of the steps of H taken inside $mtxB$, this implies that p_i is also stuck in a passage of $mtxB$ in S. □

Lemma 33 *Algorithm 6 satisfies deadlock-freedom (respectively starvation-freedom) if the base mutex internal to mtxC (i.e., mtxB in Algorithm 5) satisfies the same property.*

Proof Let H be an infinite fair history of Algorithm 6 with finitely many crash steps. Suppose that $mtxB$ in Algorithm 5 satisfies deadlock-freedom (respectively starvation-freedom). Suppose also for contradiction that some process p_i is in the recovery or entry section after some finite prefix of H, and no process subsequently enters (respectively p_i does not subsequently enter) the CS. Let H_C be the projection of H defined in Lemma 29, and let S be a suffix of H_C whose existence is guaranteed by Lemma 32. Let S_B be the projection of S onto steps of $mtxB$. It follows from Lemma 31 that S_B is a fair history of the mutex algorithm that implements $mtxB$, and that any process that is stuck in H is also stuck in S_B. It also follows from the structure of Algorithm 5 and Algorithm 6 that since p_i is stuck before the CS in H (i.e., stuck in **Recover**() or **Enter**()) then the same holds in S_B (i.e., stuck in **Enter**()). Moreover, if a process executes the CS of $mtxB$ then it executes the CS of Algorithm 6 as well, or else crashes, because it cannot be stuck in $mtxA_2$ or $mtxA_N$.

We assumed earlier that some process p_i is in the recovery or entry section of Algorithm 6 in H, and no process subsequently enters the CS (respectively p_i does not subsequently enter the CS). Thus, p_i is stuck before the CS of Algorithm 6 in H, and there are no additional executions of the CS (respectively p_i does not subsequently enter the CS) after p_i leaves the RS. This implies that p_i is stuck in $mtxB$.**Enter**() in S_B, and either there are no additional executions of the CS of $mtxB$ (respectively p_i has no additional executions of the CS of $mtxB$), or else one or more processes (different from p_i) enter the CS of $mtxB$ and then crash before entering the CS of Algorithm 6. In the latter case, processes can only enter the CS of $mtxB$ finitely many times as each such entry is associated with a distinct crash, and we assume that H contains finitely many failures. Thus, eventually p_i remains stuck in $mtxB$.**Enter**(), and for each additional step p_i takes, there is no subsequent execution of the CS of $mtxB$ by any process. Since S_B is a fair history of $mtxB$, this contradicts the deadlock-freedom (respectively starvation-freedom) property of $mtxB$. □

Lemma 34 *Algorithm 6 satisfies 0-bounded recovery.*

Proof Let H be any history of Algorithm 6 and consider an execution of **Recover**() by a process p_i, which corresponds to one execution of $mtxC$.**Recover**() at line 87. Let H_C be the projection of H defined in Lemma 29. Suppose that p_i is not in cleanup with respect to Algorithm 6 while it executes **Recover**(), meaning that it is either executing its first passage in H, or the next passage following a failure-free passage. Then p_i is either not in cleanup with respect to $mtxC$ in the corresponding passage in H_C, or in cleanup after breaking out of, or bypassing, $mtxC$.**Enter**() in Algorithm 6. In the former case, p_i is able to complete $mtxC$.**Recover**() in a bounded number of its owns steps by the 0-BR property of Algorithm 5, which holds by Lemma 29 and Theorem 5. In the latter case, p_i executes $mtxC$.**Recover**() with $C[i] = 1$ in Algorithm 5, and so it also completes $mtxC$.**Recover**() in a bounded number of its owns steps due to line 56 of Algorithm 5. Thus, in all cases p_i completes **Recover**() of Algorithm 6 in a bounded number of its own steps, as required. □

Lemma 35 *Algorithm 6 satisfies terminating exit (respectively wait-free exit) if the base mutex internal to mtxC (i.e., mtxB in Algorithm 5) satisfies the same property.*

Proof The auxiliary mutexes $mtxA_2$ and $mtxA_N$ both satisfy wait-free exit by Lemma 29, Theorem 1 and Theorem 4. Therefore, busy-waiting in **Exit**() of Algorithm 6 is limited to $mtxC$.**Exit**() at line 111.
Terminating exit: Suppose that $mtxB$ satisfies terminating exit. Let H be an infinite fair history of Algorithm 6 with finitely many failures. Suppose for contradiction that some process p_i becomes stuck forever in **Exit**() of Algorithm 6. Then Lemma 31 implies that p_i is stuck in a single execution of $mtxB$.**Exit**() in Algorithm 5, which implements $mtxC$ in Algorithm 6. Let H_C be the projection of H defined in Lemma 29. It follows from Lemma 32 that H_C has a suffix S whose projection S_B onto steps of $mtxB$ in Algorithm 5 (i.e., the base mutex internal to $mtxC$) is a fair history of the mutex algorithm that implements $mtxB$. Since p_i is stuck in $mtxB$.**Exit**(), it is also stuck in S_B, which contradicts the terminating exit property of $mtxB$.
Wait-free exit: Suppose that $mtxB$ satisfies wait-free exit. Then $mtxC$ satisfies wait-free exit by Lemma 24, and so there exists a function $f(N)$ of the maximum number of processes that bounds the number of steps incurred in any call to $mtxC$.**Exit**(). Let H be any finite history of Algorithm 6 and consider an execution of $mtxC$.**Exit**() by some process p_i at line 82. It follows from Lemma 29 and the wait-free exit property of $mtxC$ that this call completes in at most $f(N)$ steps. Since the other code in **Exit**() of Algorithm 6 is wait-free, this implies that Algorithm 6 satisfies wait-free exit. □

Lemma 36 *For any history H of Algorithm 6, for any process p_i, and for any passage through Algorithm 6 by p_i in H:*

1. *if p_i begins this passage with $C[i] \notin \{0, 1\}$ in the state of mtxC (Algorithm 5) then p_i's passage is 0-failure-concurrent;*
2. *if p_i breaks out of $mtxC$.**Enter**() at line 89 of Algorithm 6 then p_i's passage is 1-failure-concurrent; and*
3. *if p_i bypasses $mtxC$.**Enter**() at line 88 with $Breakout[i]$ = true then p_i's passage is 0-failure-concurrent.*

Proof Consider any H and p_i, and any passage by p_i through Algorithm 6. Let H_C be the projection of H defined in Lemma 29.
Part 1: Suppose that p_i's corresponding passage in H_C begins with $C[i] \notin \{0, 1\}$. Then p_i began its passage with $C[i] \in \{2, 3, 4\}$, and so p_i's most recent crash step in H_C corresponds to a failure at line 87 of Algorithm 6 inside $mtxC$.**Recover**(), or at line 89 inside $mtxC$.**Enter**() but before breaking out, or at line 111 inside $mtxC$.**Exit**(). Note that a crash in H_C that represents p_i breaking out of $mtxC$.**Enter**() at line 89 (see construction of H_C in Lemma 29) always occurs with $C[i] = 1$ in Algorithm 5, and so it does not apply here. Similarly, a crash in H_C that represents p_i bypassing $mtxC$.**Enter**() always occurs with $C[i] \in \{0, 1\}$ in Algorithm 5 because it follows the completion of $mtxC$.**Recover**() at line 87 of Algorithm 6, and so this case also does not apply. Thus, p_i is in cleanup both in its passage through Algorithm 5 in H_C and in the corresponding passage through Algorithm 6 in H, as required.
Part 2: Suppose that p_i breaks out of $mtxC$.**Enter**() at line 89. Then p_i completes lines 72–76 of Algorithm 5 twice, which implies that it read $Gate[i] \neq \perp$ at line 79. At the time when p_i read this value, some process p_z had closed the gate at line 63 and had not yet reopened it at line 69. In this scenario, p_z began $mtxC$.**Recover**() with $C[z] \notin \{0, 1\}$, and so p_z's passage through Algorithm 6 in H is 0-failure-concurrent by part 1. Moreover, the latter passage interferes with p_i's, hence p_i's passage in H is 1-failure-concurrent, as required.
Part 3: Suppose that process p_i bypasses $mtxC$.**Enter**() at line 88. Then p_i's passage through Algorithms 6 begins with $Breakout[i]$ = true, which implies that p_i begins in cleanup because a failure-free passage through Algorithm 6 always ends with $Breakout[i]$ = false due to lines 110–113. In other words, p_i's passage is 0-failure-concurrent, as required. □

Lemma 37 *Suppose that the base mutex internal to $mtxC$ (i.e., $mtxB$ in Algorithm 5) has $O(f(N))$ worst-case RMR complexity per passage in the CC and DSM models, and uses V shared variables internally. Let H be a history of Algorithm 6. For any process p_i, and for any passage of p_i in H, the number of RMRs p_i incurs in the CC and DSM models while executing the corresponding steps of $mtxC$ is as follows:*

- **Recover**()*: $O(N+V)$ if p_i is in cleanup, and $O(1)$ otherwise.*
- **Enter**() *and* **Exit**()*: $O(f(N))$.*

Proof Consider a passage by p_i in H. Let H_C be the projection of H defined in Lemma 29, and consider p_i's corresponding passage through $mtxC$.

For $mtxC$.**Recover**(), the worst-case RMR complexity is $O(N+V)$ by Lemma 26, as required. On the other hand, if p_i's passage through Algorithm 6 is not 0-failure-concurrent then it follows from Lemma 36 that p_i begins the corresponding passage in H_C with $C[i] \in \{0, 1\}$. In this case Lemma 26 dictates that $mtxC$.**Recover**() incurs only $O(1)$ RMRs, as required.

For $mtxC$.**Enter**(), Lemma 26 states that p_i incurs $O(f(N) \times (1+F))$ RMRs where F denotes an upper bound on the number of passages that begin in cleanup after recovering with $C[i] \notin \{0, 1\}$ in Algorithm 5 and are concurrent with p_i's passage. The $(1+F)$ factor in this formula represents the repetition of line 79 in Algorithm 5 due to branches at lines 78 and 80, which are executed $O(1)$ times in H due to the breakout mechanism at line 89 of Algorithm 6. As a result, the actual number of RMRs incurred in $mtxC$.**Enter**() at line 79 is $O(f(N))$ in all passages through Algorithm 6, similarly to the case when $F = 0$ in Lemma 26.

Finally, the RMR bound for $mtxC$.**Exit**() follows directly from Lemma 26. ☐

Lemma 38 *Suppose that the base mutex internal to $mtxC$ (i.e., $mtxB$ in Algorithm 5) has $O(f(N))$ worst-case RMR complexity per passage in the CC and DSM models, and uses V shared variables internally. Then for any history H of Algorithm 6 and any process p_i, the number of RMRs p_i incurs in the CC and DSM models in one passage is as follows:*

- **Recover**()*: $O(N+V)$ if p_i is in cleanup, and $O(1)$ otherwise.*
- **Enter**() *and* **Exit**()*: $O(f(N))$ if p_i's passage is not 2-failure-concurrent, otherwise $O(f(N)+\log N)$.*

Proof The **Recover**() procedure of Algorithm 6 comprises a call to $mtxC$.**Recover**() and no other steps. Thus, the required RMR bound follows directly from Lemma 37.

Next, consider **Enter**() and **Exit**(). The calls to $mtxC$.**Enter**() and $mtxC$.**Exit**() incur $O(f(N))$ RMRs by Lemma 37. Furthermore, it follows from Lemma 29 as well as Theorems 1 and 4 that a process incurs $O(\log N)$ RMRs while accessing $mtxA_2$ and $mtxA_N$ at lines 95–96, 99–100, 101–102, 104, 107 and 108. All other statements in the Algorithm 6 incur $O(1)$ RMRs. Thus, the target entry and exit sections incur $O(f(N)+\log N)$ RMRs in the CC and DSM models in the worst case.

Now suppose that process p_i executes a passage that is not 2-failure-concurrent with respect to the target algorithm. Further to the above analysis, we must show that the RMR complexity of **Enter**() and **Exit**() is reduced from $O(f(N)+\log N)$ to $O(f(N))$. In other words, we must show that $mtxA_N$ is bypassed entirely in such a passage, since this is the component responsible for the logarithmic term in the RMR complexity bound established earlier for arbitrary passages.

To prove that p_i acquires the fast path, first note that since p_i's passage is not 2-failure-concurrent in this part of the analysis, p_i does not bypass, or break out of, $mtxC$.**Enter**() by Lemma 36 as otherwise its passage would be 0-failure-concurrent, or 1-failure-concurrent, respectively. Thus, p_i executes $mtxC$.**Enter**() at line 89 to completion, and proceeds to line 94 after acquiring the CS of $mtxC$, with $Breakout[i] = \mathsf{false}$. Next, consider p_i's attempt to acquire the fast path at line 94. Since p_i's passage is not 2-failure-concurrent, it follows that p_i did not begin in cleanup, and so it reaches line 94 with $Bounded[i] = \mathsf{false}$. This holds either by initialization or by the execution of line 109 after line 98 in p_i's previous passage, which is failure-free since p_i is not in cleanup in the passage under consideration. It remains to show that p_i enters the fast path, in which case it bypasses $mtxA_N$, as required.

Suppose for contradiction that p_i fails to enter the fast path, which means that $F \notin \{\perp, i\}$ holds immediately before the `CAS` instruction at line 94. This implies that some other process p_j has acquired the fast path and has not yet released it by the time p_i reaches line 94. Since p_i is in the CS of $mtxC$ at this point, it follows from the mutual exclusion property of $mtxC$ and from Lemma 29 that p_j is not in the CS of $mtxC$ while in the fast path. As a result, p_j either bypassed or broke out of $mtxC$.**Enter**() in its passage. By Lemma 36, p_j's passage is either 0-failure-concurrent or 1-failure-concurrent, hence 1-failure-concurrent in both cases. Since p_j's passage interferes with p_i's passage, this implies that p_i's passage is 2-cleanup-concurrent, which contradicts our earlier supposition. Thus, we have shown that p_i acquires the fast path successfully, as required to establish that p_i incurs $O(f(N))$ RMRs in a passage

through Algorithm 6 that is not 2-failure-concurrent. □

Theorem 6 *Algorithm 6 satisfies ME and 0-BR, and also preserves the DF, SF, TE, and WFE properties of the base mutex internal to mtxC (i.e., mtxB in Algorithm 5). Furthermore, supposing that the base mutex internal to mtxC has $O(f(N))$ worst-case RMR complexity per passage in the CC and DSM models for some function $f(N)$, and uses V shared variables internally, the number of RMRs a process p_i incurs in one passage through Algorithm 6 is as follows: $O(N + V + f(N))$ if p_i's passage is 2-failure-concurrent, and $O(f(N))$ otherwise.*

Proof Properties ME and 0-BR follow directly from Lemmas 30 and 34, respectively. Preservation of DF, SF, TE and WFE follows from Lemmas 33 and 35. The RMR complexity in the CC and DSM models follows from Lemma 38. □

We complete the analysis by discussing FCFS fairness in Algorithm 6. The target doorway is defined to be the same as the base doorway, which is executed at line 89 of Algorithm 6, assuming that a process completes this line without breaking out of the base entry section or bypassing it. It follows easily that the doorway is executed to completion in any passage through Algorithm 6 that is not 2-failure-concurrent, as this is the case when the fast path is taken (see RMR bounds in Lemma 38). With that observation in mind, we now establish 2-FCFS in Lemma 39.

Lemma 39 *If the base mutex used by Algorithm 5 satisfies First-Come-First-Served (FCFS) fairness then Algorithm 6 satisfies 1-FCFS.*

Proof If the base mutex of Algorithm 5 satisfies FCFS then it follows from Lemma 27 that Algorithm 5 satisfies 1-FCFS. Thus, $mtxC$ in Algorithm 6 satisfies 1-FCFS. Let H be a history of Algorithm 6, and suppose that process p_i enters the CS in its ℓ_i-th passage and p_j enters the CS in its ℓ_j-th passage in H. Suppose that neither passage is 1-failure-concurrent. This implies that both processes acquire $mtxC$ at line 89 prior to entering the CS, as otherwise their passages would be 0-failure-concurrent or 1-failure-concurrent by Lemma 36, hence 1-failure-concurrent in either case. Thus, both processes enter the CS of $mtxC$ in addition to the CS of Algorithm 6. Suppose for contradiction that p_i completes the target doorway in its ℓ_i-th passage before p_j begins the target doorway in its ℓ_j-th passage, and yet p_j enters the target CS in its ℓ_j-th passage before p_i enters the target CS in its ℓ_i-th passage. Now consider p_i's corresponding ℓ_i-th passage and p_j's corresponding ℓ_j-th passage through $mtxC$, noting that one passage through the target mutex maps to exactly one passage through the base mutex in this case. Then the order of execution of the doorway of $mtxC$ is analogous in the corresponding passages through $mtxC$: p_i completes the doorway of $mtxC$ in its ℓ_i-th passage before p_j begins the doorway of $mtxC$ in its ℓ_j-th passage. Similarly, p_j enters the CS of $mtxC$ in its ℓ_j-th passage before p_i enters the CS of $mtxC$ in its ℓ_i-th passage because the base and target algorithms both provide ME by Theorems 5 and 6, and because the target CS is nested inside the base CS in any passage that is not 1-failure-concurrent. This contradicts the 1-FCFS property of Algorithm 5. □

4.3 Discussion of Space Complexity

The space complexity of the Algorithm 5 from Section 4.1 is $O(g(N) + N^2)$, where $O(g(N))$ denotes the space complexity of the base mutex. The N^2 term is due to the two-dimensional array of spin variables, and can be reduced to $O(N)$ in the CC model. Algorithm 6 in Section 4.2 adds $O(N \log N)$ space for the auxiliary N-process mutex, which can be reduced to $O(N)$ in the CC model, as discussed in Section 3.4.

5 Related Work

Literature on mutual exclusion begins with Dijkstra's seminal paper [12], although the first known solution to the problem is credited to Dekker, who proposed a two-process algorithm that uses one-bit read/write registers. Lamport advanced the state of the art by formalizing the correctness properties of mutual exclusion [35, 36], and also introduced the famous Bakery algorithm as an example of first-come-first-served (FCFS) fairness [34]. Whereas the Bakery uses only reads and writes, and orders processes using numerical tickets, more scalable FCFS algorithms implement various queue structures using read-modify-write primitives [5,38,21,39]. For a detailed treatment of progress in mutual exclusion research up to 2003, the reader is referred to [4, 46].

Local spin mutual exclusion algorithms, which guarantee bounded RMR complexity per passage by busy-waiting only on locally accessible shared variables, have been studied intensively due to their performance benefits [5]. For the class of algorithms that use reads, writes and comparison primitives, the tight bound on RMRs per passage in the worst case is $\Theta(\log N)$. Yang and Anderson [49] proved the upper bound in the CC and DSM models using an arbitration tree modeled after Kessels

[33]. Attiya, Hendler, and Woelfel [6] later proved the matching lower bound, building on a series of earlier results [3,11,14,17]. In comparison, queue-based locks achieve $O(1)$ RMR complexity but require additional primitives, such as atomic Fetch-And-Store or Fetch-And-Add.

Recovery from failures is featured prominently in the seminal work of Lamport [36], who formalized two types of faulty process behavior: "unannounced death", similar to a crash in our model but permanent, and "malfunctioning", whereby the private state and communication variables of a process assume arbitrary values. Lamport's Bakery algorithm [34] tolerates the first type of failure provided that a faulty process returns to the remainder section and its communication variables (i.e., the single-writer shared registers written by it) are reset eventually to zero—an assumption that is false in our model.

Taubenfeld's treatment of fault-tolerant mutual exclusion focuses on a crash-recovery model where process failures affect the values of shared variables in well-defined ways [48]. This model is defined around single-writer registers, and assumes that only those variables that a process "owns" (i.e., has write access to) may be affected by its failure. In one variation, the program counter is reset to the remainder section on failure, and any variables a process owns are reset to default values. In another variation, the program counter and variables owned by a process may adopt arbitrary values, possibly leading to temporary violations of safety and liveness. Solutions in the latter category are based upon Dijkstra's self-stabilization paradigm [13]. In comparison, a crash failure in our model resets the program counter deterministically and does not affect shared variables at all. Furthermore, we consider multi-writer registers in Section 3, as well as read-modify-write primitives in Section 4.

Bohannon, Lieuwen, Silberschatz, Sudarshan, and Gava [9] proposed a technique for determining the ownership of a Test-And-Set lock following a permanent crash failure, which is the most difficult aspect of recovery in this case given that the mutex algorithm itself is quite simple. In a follow-up paper, Bohannon, Lieuwen, and Silberschatz [8] added recoverability to Mellor-Crummey and Scott's queue-based mutex algorithm [39] by designing an intricate mechanism to repair the queue structure after a process fails while enqueuing or dequeuing itself. Both papers augment the mutex lock with additional shared variables to detect the intent of a process to enter the CS, and perform corrective actions inside a dedicated recovery process that is able to detect failures by querying the operating system. This recovery process is itself assumed to be reliable.

In terms of RMR complexity, the recoverable MCS lock in [8] does not bound the number of RMRs per passage in the CC or DSM models. This is because when the recovery process is executing corrective actions, a process in the entry section or exit section waits at specific points for such actions to finish by spinning on a global variable. The RMR complexity of this busy-wait loop is unbounded in the DSM model because all processes share the same spin variable. The RMR complexity is also unbounded in the CC model unless the number of failures is bounded, as otherwise the recovery section may invoke corrective actions arbitrarily many times in parallel with one execution of the entry or exit section, each time causing an RMR. In comparison, our model allows both failures and concurrency in the recovery section, and three of our algorithms guarantee bounded RMR complexity per passage irrespective of the number of failures.

Michael and Kim proposed another fault-tolerant mutex lock in which a process that is waiting to acquire a lock can "usurp" the lock if it determines that the previous lock holder has crashed permanently [40]. This technique is not applicable in our model since we assume that a crashed process eventually recovers and attempts to clean up the lock.

Research on mutual exclusion, and more generally on concurrent objects, has focused mostly on models where the memory is reliable. In this body of work, a limited form or resilience against unreliable processes follows immediately from liveness guarantees in an asynchronous environment, where a slow process cannot be distinguished from one that has crashed permanently. For example, Herlihy introduced *wait-free* objects [24], which guarantee the progress of each correct process individually. In comparison, only a handful of papers consider computation using unreliable memory, focusing on minor corruptions such as bit flips. Afek, Greenberg, Merritt, and Taubenfeld [1] considered the consensus problem in this general context, Moscibroda and Oshman [43] focused on mutual exclusion, and Jayanti, Chandra, and Toueg [28] proposed implementations of shared objects from unreliable base objects. In contrast to these techniques, which break if the number of corruptions exceeds a specified bound, Hoepman, Papatriantafilou, and Tsigas [25], as well as Johnen and Higham [31], proposed self-stabilizing shared objects that can tolerate any number of memory failures but may lose their safety properties temporarily after a failure.

Several new developments on the topic of recoverable mutual exclusion have occurred since the publica-

tion of the conference version of this manuscript [20]. Jayanti and Joshi recently proposed an RME algorithm that uses reads, writes, and single-word Compare-And-Swap operations, and incurs $O(\log N)$ RMRs per passage in the CC and DSM models [29]. The new breakthrough in this work is wait-free recovery code, achieved using an f-array – a shared object type that generalizes multi-writer snapshots by supporting the computation of a user-specified function f (e.g., min or max) over the array elements [27]. The logarithmic RMR complexity bound in our work and [29] was shown to be sub-optimal in the CC model in follow-up work by Golab and Hendler [18,19]. Jayanti, Jayanti and Joshi enhanced and extended this result by proposing an algorithm that attains sub-logarithmic RMR complexity in both the CC and DSM models [30].

The best known RMR complexity bounds for the recoverable mutual exclusion problem under the independent crash failure assumption are $O(1)$ in the CC and DSM models for algorithms that use atomic reads, writes, and double-word read-modify-write primitives, and $O(\log N/\log\log N)$ in the CC and DSM models for algorithms that use only atomic reads, writes, and commonly supported single-word read-modify-write primitives [18,30]. Under the assumption of system-wide failures and with additional information provided to processes by the environment, $O(1)$ RMR complexity is achievable in the CC and DSM models using only atomic reads, writes, and commonly supported single-word read-modify-write primitives [19].

6 Conclusion

In this paper, we formalized the recoverable mutual exclusion problem and presented four solutions, three of which guarantee bounded RMR complexity per passage in the CC and DSM models irrespective of failures (see Figure 1). Our work leaves open several research questions: What is the tight RMR complexity bound for RME algorithms that use only commonly supported single-word synchronization primitives primitives? Can randomized solutions beat deterministic ones in terms of RMRs, as has been shown for ordinary mutual exclusion [6,7,15,23]? Is it possible to solve the problem for N processes using bounded RMRs in the DSM model and only $O(N)$ shared variables, thus improving on the space complexity of our N-process algorithms? (Simple optimizations reduce the space complexity of our solutions to $O(N)$ in the CC model.) How would one design a recoverable reader-writer lock?

Acknowledgements Sincere thanks to Peter Buhr, Patrick Lam, and the anonymous referees of PODC'16 and Distributed Computing for detailed feedback and helpful suggestions on earlier drafts of this work. We are grateful also to Vassos Hadzilacos, Danny Hendler, Prasad Jayanti, Gadi Taubenfeld, and Sam Toueg for stimulating technical discussions.

References

1. Y. Afek, D. S. Greenberg, M. Merritt, and G. Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6): 1231–1274, 1995.
2. J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, 2001.
3. J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, 2002.
4. J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.
5. T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.
6. H. Attiya, D. Hendler, and P. 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.
7. M. A. Bender and S. Gilbert. Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work. In *Proc. of the 52nd Symposium on Foundations of Computer Science (FOCS)*, pages 728–737, 2011.
8. P. Bohannon, D. F. Lieuwen, and A. Silberschatz. Recovering scalable spin locks. In *Proc. of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 314–322, 1996.
9. P. Bohannon, D. F. Lieuwen, A. Silberschatz, S. Sudarshan, and J. Gava. Recoverable user-level mutual exclusion. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 293–301, 1995.
10. J. E. Burns and N. A. Lynch, Bounds on shared memory for mutual exclusion. *Inf. Comput.*, 107(2):171–184, 1993.
11. R. Cypher. The communication requirements of mutual exclusion. In *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 147–156, 1995.
12. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
13. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643-644, 1974.
14. R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *Proc. of the 25th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, 2006.
15. G. Giakkoupis and P. Woelfel. Randomized mutual exclusion with constant amortized RMR complexity on the DSM. In *Proc. of the 55th Symposium on Foundations of Computer Science (FOCS)*, pages 504–513, 2014.
16. P. B. Gibbons, How emerging memory technologies will have you rethinking algorithm design, In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, page 303, 2016.

17. W. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012.
18. W. Golab and D. Hendler. Recoverable mutual exclusion in sub-logarithmic time. In *Proc. of the 36th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 211–220, 2017.
19. W. Golab and D. Hendler. Recoverable mutual exclusion under system-wide failures. In *Proc. of the 37th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 17–26, 2018.
20. W. Golab, A. Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.
21. G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.
22. J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1993.
23. D. Hendler and P. Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distributed Computing*, 24(1):3–19, 2011.
24. M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
25. J.-H. Hoepman, M. Papatriantafilou, and P. Tsigas. Self-stabilization of wait-free shared memory objects. In *Proc. of the 9th International Workshop on Distributed Algorithms (WDAG)*, pages 273–287, 1995.
26. Intel Corporation. Single-chip cloud computer. `http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-overview-paper.pdf`.
27. P. Jayanti. F-arrays: Implementation and Applications. In *Proc. of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279, 2002.
28. P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3): 451–500, 1998.
29. P. Jayanti and A. Joshi. Recoverable FCFS mutual exclusion with wait-free recovery. In *Proc. of the 31st International Symposium on Distributed Computing (DISC)*, pages 30:1–30:15, 2017.
30. P. Jayanti, S. Jayanti, and A. Joshi. A Recoverable Mutex Algorithm with Sub-logarithmic RMR on Both CC and DSM. In *Proc. of the 38th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 177–186, 2019.
31. C. Johnen and L. Higham. Fault-tolerant implementations of regular registers by safe registers with applications to networks. In *Proc. of 10th International Conference of Distributed Computing and Networking (ICDCN)*, pages 337–348, 2009.
32. Y.-J. Kim and J. H. Anderson, A space- and time-efficient local-spin spin lock. *Inf. Process. Lett.*, 84(1):47–55, 2002.
33. J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.
34. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
35. L. Lamport. The mutual exclusion problem: part I – a theory of interprocess communication. *Journal of the ACM*, 33(2):313–326, 1986.
36. L. Lamport. The mutual exclusion problem: part II – statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.
37. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.
38. P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the 8th International Parallel Processing Symposium (IPPS)*, pages 165–171, 1994.
39. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
40. M. Michael and Y. Kim. Fault tolerant mutual exclusion locks for shared memory systems. US Patent 7,493,618, 2009.
41. S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Trans. Parallel Distrib. Syst.*, 27(5):1537–1550, 2016.
42. J. C. Mogul, E. Argollo, M. A. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.
43. T. Moscibroda and R. 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.
44. D. Narayanan and O. Hodson. Whole-system persistence. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 401–410, 2012.
45. A. Ramaraju. RGLock: Recoverable mutual exclusion for non-volatile main memory systems. Master's thesis, University of Waterloo, 2015. `https://uwspace.uwaterloo.ca/handle/10012/9473`.
46. M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.
47. M. Scott and W. Scherer. Scalable queue-based spin locks with timeout. In *Proc. of the 8th ACM SIGPLAN symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 44–52, 2001.
48. G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice Hall, 2006.
49. J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.