Recoverable Mutual Exclusion Under System-Wide Failures

Wojciech Golab*
Department of Electrical and Computer Engineering
University of Waterloo
wgolab@uwaterloo.ca

Danny Hendler†
Department of Computer Science
Ben-Gurion University
hendlerd@cs.bgu.ac.il

ABSTRACT

Recoverable mutual exclusion (RME) is a variation on the classic mutual exclusion (ME) problem that allows processes to crash and recover. The time complexity of RME algorithms is quantified in the same way as for ME, namely by counting remote memory references – expensive memory operations that traverse the processor-to-memory interconnect. Prior work has established that the RMR complexity of the RME problem for n processes is $\Theta(\log n)$ for the class of algorithms that use read/write registers and single-word comparison primitives such as Compare-And-Swap (Golab and Ramaraju 2016), $O(\log n/\log\log n)$ for the class of algorithms that use read/write registers and additional single-word read-modify-primitives such as Fetch-And-Store (Golab and Hendler 2017), and $\Theta(1)$ for the class of algorithms that use read/write registers and specialized double-word read-modify-write primitives (Golab and Hendler 2017). These complexity bounds hold in a model of computation where processes may fail independently, and where a process that fails while accessing the mutex is required to recover eventually. This body of work leaves open two important questions: (i) what is the tight bound on the RMR complexity of RME for the class of algorithms that use read/write registers and commonly supported single-word read-modify-primitives; and (ii) how is the RMR complexity of RME affected by variations in the failure model? This paper answers both questions partially by showing that RME can be solved using $O(1)$ RMRs per passage in the worst case in a model where failures are system-wide (i.e., all processes crash simultaneously), and processes receive additional information from the environment regarding the occurrence of the failure. The upper bound algorithm we present relies crucially on a novel RMR-efficient barrier that processes use to synchronize recovery actions after each failure. The barrier uses read/write registers and single-word Compare-And-Swap only. Additionally, we present a transformation that can add properties such as critical section re-entry and a strong notion of starvation freedom to any RME algorithm while preserving its asymptotic RMR complexity.

*Wojciech Golab is supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada, Discovery Grants Program, and by a Google Faculty Research Award.
†Danny Hendler is supported in part by the Israel Science Foundation (grant 1749/14).

PODC '18, July 23–27, 2018, Egham, United Kingdom

ACM ISBN 978-1-4503-5795-1/18/07…$15.00
https://doi.org/10.1145/3212734.3212755

CCS CONCEPTS

• **Theory of computation** → **Shared memory algorithms**; • **Software and its engineering** → *Mutual exclusion*; • **Computer systems organization** → Reliability;

KEYWORDS

Mutual exclusion; recovery; fault tolerance; concurrency; shared memory; multi-core algorithms; non-volatile main memory; persistent data structures.

1 INTRODUCTION

Non-volatile main memory (NVRAM) will soon become available commercially. This novel storage medium will challenge the traditional coupling of volatile main memory with non-volatile secondary storage in computer architectures, which has long led to a separation of program state into operational data stored using disposable in-memory data structures, and recovery data stored using durable but sequential on-disk structures such as transaction logs. The eventual convergence of primary and secondary storage into a single layer in the memory hierarchy will allow computer systems to simultaneously harness the performance benefits of main memory and the durability of secondary storage, provided that software design adapts accordingly. In particular, system designers must rethink traditional log-based recovery techniques, which restrict parallelism during recovery, and embrace techniques that allow multiple processing cores to recover concurrently using state saved in NVRAM.

One of the fundamental research problems related to constructing fault-tolerant data structures for NVRAM pertains to hardening mutual exclusion locks against crash-recovery failures. Research on this topic was revived recently by Golab and Ramaraju, who formulated and solved the Recoverable Mutual Exclusion (RME) problem [18]. RME is a modern take on the classic mutual exclusion (ME) problem formulated by Dijkstra [11] that allows a process to crash while accessing (i.e., acquiring, releasing, or holding) a lock provided that it recovers eventually and resumes execution, beginning with recovery actions that restore the internal state of the lock. In contrast, the ME problem assumes that a process never fails while accessing a lock, as otherwise a conventional lock becomes vulnerable to starvation.

Recent work on the RME problem [16, 18, 22] precipitated a flurry of upper bounds on time complexity, which in this context is quantified by counting *remote memory references* (RMRs) – expensive memory operations that cause traffic on the interconnect joining processors with memory. In the model with independent process failures, Golab and Ramaraju proved that RME can be solved for N processes using only atomic reads and writes in $O(\log N)$ RMRs,

which matches the tight bound known for conventional ME [5, 32]. However, the tight bound on RMR complexity is not known for the class of algorithms that use commonly supported single-word Read-Modify-Write primitives, some of which are known to be sufficiently powerful to solve ME using only $O(1)$ RMRs. In particular, only two solutions to the RME problem have been published that beat that logarithmic bound: Golab and Hendler presented an $O(\log N / \log \log N)$ RMRs algorithm that uses single-word Fetch-And-Store and Compare-And-Swap, and an $O(1)$ RMRs algorithm that uses specialized double-word Read-Modify-Write operations [16].

This paper advances the state of the art by providing partial answers to two fundamental questions: (i) what is the tight bound on the RMR complexity of RME for the class of algorithms that use read/write registers and commonly supported single-word read-modify-primitives; and (ii) how is the RMR complexity of RME affected by variations in the failure model? Specifically, we show that RME can be solved using only $O(1)$ RMRs in the worst case in a model where failures are system-wide and processes receive additional information from the environment regarding the occurrence of the failure. The algorithms we present rely crucially on a novel RMR-efficient barrier that processes use to synchronize recovery actions after each failure. The barrier uses read/write registers and single-word Compare-And-Swap only. Additionally, we present a transformation that can add properties such as critical section re-entry and a strong notion of starvation freedom to any RME algorithm while preserving its asymptotic RMR complexity.

2 MODEL

We consider N asynchronous unreliable processes, $p_1, p_2, \ldots, p_N$, that communicate by accessing variables in shared memory. These processes compete for an exclusive lock (recoverable mutex [18]), which protects a shared resource, by executing an infinite loop whose body comprises the following sections: the *non-critical section* (NCS), where processes do not access the lock or the shared resource; the *recovery protocol* (**Recover**), where a process repairs up the internal structure of the lock following a crash failure; the *entry protocol* (**Enter**), where a process waits for access to the shared resource; the *critical section* (CS), where a process accesses the shared resource; and finally the *exit protocol* (**Exit**), where a process makes the shared resource available to others.

Correctness properties for recoverable mutex algorithms are expressed in reference to *histories* that record the actions of processes. Formally, a history H is a sequence of *steps* that come in two varieties. An *ordinary step* is a shared memory operation combined with bounded local computation (e.g., arithmetic, accessing private variables, advancing the program counter). A *crash step* is a system-wide failure that resets the private variables of all processes to their initial values, and transitions the program counters to the NCS. The next step a process takes after a crash step is either another crash step, or 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 occurs first. A passage is called *failure-free* if it does not end with a crash step. 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 history H is *fair* if it is finite, or if it is infinite and every process that participates either executes infinitely many steps or stops taking steps after completing a failure-free passage.

The fairness property implies that a process always recovers eventually following a crash in the middle of a passage. This aspect of the model makes it possible to maintain liveness properties in the presence of process failures without relying on accurate failure detection, and is inherited from Golab and Ramaraju [18]. It can be viewed as a generalization of the traditional assumption that a process participating in mutual exclusion never halts permanently outside of the NCS.

The fundamental correctness properties of RME algorithms include the following [18]:

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

Starvation Freedom (SF): For any infinite fair history H, if a process p_i leaves the NCS in some step of H then eventually p_i itself enters the CS, or else there are infinitely many crash steps in H.

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.

Bounded Recovery (BR): There exists an integer $b > 0$ such that for any history H and for any process p_i, any execution of the recovery protocol by p_i incurs at most b steps.

Bounded Exit (BE): There exists an integer $b > 0$ such that for any history H and for any process p_i, any execution of the exit protocol by p_i incurs at most b steps.

The time complexity of RME algorithms is defined as the number of *remote memory references* (RMRs) executed by a process per passage. An RMR is any memory operation that traverses the processor-to-memory interconnect, and is defined in an architecture-specific manner. In this paper we consider both the cache-coherent (CC) and distributed shared memory (DSM) models [3]. 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, which is determined statically at initialization.

The algorithms presented in this paper assume that processes have knowledge of an *epoch number* parameter, which is a positive integer that increases after each system-wide failure. For example, the epoch number can be the value of a counter that is incremented by the operating system after every failure, or the time of the last system boot.[1] This parameter augments the model introduced in [18], and is given as input to the pseudo-code procedures **Recover**, **Enter**, and **Exit**. All passages executed between successive failures use the same epoch number, whereas passages made after a particular failure use a higher epoch number than passages made before this failure. Thus, epoch numbers increase monotonically, but need not reflect the exact number of failures that have occurred since the beginning of the execution. A practical algorithm that records

[1] The counter-based method of computing epoch numbers does not guarantee monotonicity in cases where failures occur in rapid succession as there may be insufficient time between such failures to save the updated value of the counter to NVRAM. This limitation does not break the algorithms presented in this paper.

epoch numbers in registers of finite size can only tolerate a finite (but very large) number of failures.

3 BARRIER

This section presents the implementation of an RMR-efficient synchronization barrier, which is used heavily in the transformations presented later on in this paper. A barrier is accessed by executing a *barrier procedure* whose specification is captured in Definition 3.1 below. Two variations of this barrier are presented, and one is used as a building block of the other. Both variations are based on the premise that some distinguished process called the *barrier leader* (*leader* for short) must open the barrier before other processes, called non-leaders, are able to progress beyond the barrier. The barrier leader is fixed for the duration of a given epoch. The barrier can be reused in different epochs with different leaders.

Definition 3.1. A barrier procedure satisfies the following properties for any given epoch e:

(i) no call to the barrier procedure in epoch e terminates before the leader has begun its call;
(ii) in a fair history, the leader's call in epoch e eventually terminates (or a failure occurs); and
(iii) in a fair history, if the leader's call in epoch e terminates then every other call in epoch e eventually terminates (or a failure occurs).

3.1 Barrier Variation 1: Known Leader

In the first variation, the barrier procedure is called **BarrierSub**, and is invoked with two arguments: the epoch number *epoch*, and the process ID of the leader, which is decided using some external mechanism. The leader opens the barrier at line 3 by writing a global shared variable, and then uses a distributed signaling mechanism to wake up non-leader processes that may be waiting on the barrier. The signaling mechanism borrows ideas from [17], except that it uses Compare-And-Swap for handshaking among two processes as opposed to implementing leader election from reads and writes. This barrier variation is designed for and needed in the DSM model only, where it achieves $O(1)$ RMR complexity.

After writing R, the leader executes the helper procedure **BSub-Leader** in which it synchronizes with non-leader processes that may be accessing the barrier. This helper procedure begins with a handshaking mechanism at lines 8–13, which uses the array $C[lid][1..N]$ of CAS objects to settle a potential race condition between the leader and non-leaders executing line 10 and line 18. If the leader is the first to swap *epoch* into $C[lid][j]$ (i.e., wins the handshake) then p_j will progress through the barrier without blocking, otherwise p_j will wait for a signal. The for loop at lines 8–13 generates a list of non-leaders who win the handshake (i.e., with whom the leader loses the handshake) and stores their IDs in consecutive elements of $L[lid][1..N]$ at line 11. The position of a process p_j in this array is recorded in $I[lid][j]$ at line 12. Finally, the leader signals the first process in this list at line 16, and sets off a chain reaction by which the kth process in the list signals process number $k+1$.

A non-leader process p_i calls **BSub-NonLeader** if $R < epoch$ at line 1, and executes the handshake at line 18. If p_i swaps *epoch* into $C[lid][i]$ before the leader (i.e., p_i wins the handshake) then p_i waits for a signal at line 17, and then signals the next process in the list created by the leader at lines 21–24. Otherwise, if the leader already swapped *epoch* into $C[lid][i]$ then p_i returns immediately since the barrier is open and p_i is not required to participate in the distributed signaling mechanism.

Theorem 3.2 asserts the correctness properties of the barrier. The proof is omitted due to lack of space.

Theorem 3.2. **BarrierSub** *satisfies its specification (Definition 3.1).*

3.2 Barrier variation 2: Unknown Leader

The second variation of the barrier is designed for both the CC and DSM models, where it also achieves $O(1)$ RMR complexity. The barrier procedure is called **Barrier**, and is invoked with two arguments: the epoch number *epoch*, and a Boolean *isLeader* indicating whether the caller is the barrier leader. Thus, each process knows whether it is the leader, but non-leader processes do not know the ID of the leader. The barrier is opened by the leader by writing the current epoch number to a shared variable R. The variable R persists across failures but does not need to be reset since the epoch number grows monotonically. Synchronization with non-leader processes is straightforward in the CC model, and is contained in procedure **BarrierCC**. The algorithm for the DSM model is contained in procedure **BarrierDSM**, and is much more involved. This procedure uses **BarrierSub** as a subroutine, which we refer to as the *secondary barrier*.

The implementations of **Barrier**, **BarrierCC**, and **BarrierDSM** are presented in Figure 2. In the CC model, synchronization is achieved easily using a global spin variable R. The leader increases the value of R to the current epoch number at line 30, and the other processes wait for this state change at line 32. The execution path in the DSM model is more involved since a global spin variable cannot be used in an RMR-efficient manner. Synchronization is instead achieved using a combination of two mechanisms. First, the leader increases the value of the shared variable R to the current epoch number at line 48, which opens the barrier for any other process that reads R at line 41 after R is updated. A secondary mechanism is needed to handle the situation where the other process arrives at the barrier before the leader, and does not know who the leader is. This is dealt with by electing a secondary leader using the CAS object C at line 49 and line 54. Once the barrier is opened by the barrier leader, the secondary leader is unblocked at line 52, and all processes meet at the secondary barrier **BarrierSub** at line 58.

The DSM execution path begins with resetting the CAS object C at lines 42–45. The algorithm cannot rely on the leader from the previous epoch to reset C at the end of **BarrierDSM** because a crash failure can prevent the leader from progressing that far.[2] Resetting C is difficult because one has to distinguish between the case when C holds the ID of the leader from a prior epoch (which must be reset), versus the case when C holds the ID of the current leader (which must not be reset). The algorithm makes the distinction by appending a binary tag to the process ID, which is why C holds an ordered pair of the from $\langle ID, tag \rangle$. The tag is toggled when process p_i reaches line 46, and is recorded implicitly

[2]The algorithm can be simplified under the assumption that a crashed process eventually recovers and reaches the barrier again, but in this section we present a more general solution that does not rely on this assumption.

Shared variables:

- R: read/write register, **init** 0
- $C[1..N][1..N]$: **array** of readable **CAS** objects, **init** 0, elements $C[i][1..N]$ local to process p_i
- $I[1..N][1..N]$: **array** of read/write registers, **init** 0, elements $I[i][1..N]$ local to process p_i
- $L[1..N][1..N]$: **array** of read/write registers, **init** 0, elements $L[i][1..N]$ local to process p_i
- $S[1..N]$: **array** of read/write registers, **init** 0, element i local to process p_i

Private variables:

- j, k, tmp: integer, uninitialized

Procedure BarrierSub $(epoch, lid)$ for process p_i

```
   // try fast path
1  if R = epoch then return
   // take slow path
2  if lid = i then
3      R := epoch
4      BSub-Leader (epoch)
5  else
6      BSub-NonLeader (epoch, lid)
```

Procedure BSub-Leader $(epoch)$ for process p_i

```
7  k := 1
8  for j in 1..N do
9      tmp := C[i][j]
10     if CAS(&C[i][j], tmp, epoch) = epoch then
11         L[i][k] := j
12         I[i][j] := k
13         k := k + 1
14 if k > 1 then
15     tmp := L[i][1]
16     S[tmp] := epoch
```

Procedure BSub-NonLeader $(epoch, lid)$ for process p_i

```
17 tmp := C[lid][j]
18 if CAS(&C[lid][j], tmp, epoch) < epoch then
19     await S[i] = epoch
20     k := I[lid][i]
21     if k < N then
22         tmp := L[lid][k + 1]
23         if tmp ≠ 0 then
24             S[tmp] := epoch
```

Figure 1: RMR-efficient barrier with known leader.

by storing a pair of epoch numbers in the array $E[i][0..1]$. The two array elements hold distinct values (except in the initial state), and the index of the higher value indicates the state of the tag after p_i's last call to **SetTag**. The value of the tag a process p_i will hold in the current epoch after calling **SetTag** is determined by calling **GetTag** at line 44. This procedure reads $E[i][0..1]$ at lines 33–34, and searches for an array element that holds the current epoch number. If such an element exists, its index is the value of the tag for p_i in the current epoch (line 36 and line 38). If not, then the value of p_i's tag in the current epoch is different than in the last epoch where p_i toggled its tag, and is computed at line 40.

To reset C, process p_i first checks whether C holds the ID of some process p_j and its tag (as opposed to $\perp$ and a tag) at lines 42–43. If so, and the tag stored in C does not match the tag computed for p_j for the current epoch (line 44), then p_i attempts to reset C using CAS at line 45. The tag prevents the ABA problem in case some other process resets C and p_j then wins the secondary leader election at line 49 or line 54 in the current epoch after p_i reads C at line 42 and before p_i executes the CAS at line 45.

Theorem 3.3 asserts the correctness properties of the barrier. The proof is omitted due to lack of space.

Theorem 3.3. **Barrier** *satisfies its specification (Definition 3.1) and has worst-case $O(1)$ RMR complexity in the CC and DSM models.*

4 TRANSFORMATIONS

This section presents several RMR-efficient transformations for mutual exclusion (ME) and recoverable mutual exclusion (RME) algorithms. Applying these to the MCS lock [27], we obtain an $O(1)$ RMRs RME algorithm that uses read/write registers as well as single-word Fetch-And-Store and Compare-And-Swap.

4.1 Transformation 1: Conventional ME to RME

The first transformation, presented in Figure 3, converts a conventional mutex (base mutex) to a recoverable mutex (target mutex) by resetting the base mutex after each failure. The main algorithmic idea is similar to [18], except that the overhead of resetting the base mutex is reduced greatly by taking advantage of the stronger failure model. Internally, the transformation uses a conventional base mutex B, a CAS object C, and the RMR-efficient barrier described earlier in Section 3. The target entry and exit protocols simply invoke their counterparts on the base mutex B. The recovery protocol uses the shared variable C to determine who will be responsible for resetting B, and the barrier to ensure that no process accesses B unsafely while it is being reset.

In steady-state failure-free operation, C holds the current epoch number, and so the body of the recovery protocol is bypassed due

Shared variables:

- R: read/write register, **init** 0
- C: readable/writable **CAS** object, **init** $\langle \perp, 0 \rangle$
- $E[1..N][0..1]$: **array** of read/write register, **init** 0, element $E[i][0..1]$ local to process p_i
- $S[1..N]$: **array** of **Boolean** spin variables, element $S[i]$ local to process p_i, **init** 0

Private variables:

- $secldr, tag, ltag, e0, e1$: integer, uninitialized

Procedure Barrier ($epoch$, $isLeader$) for process p_i

```
25 if CC model then
26 |  BarrierCC (epoch, isLeader)
27 else if DSM model then
28 |  BarrierDSM (epoch, isLeader)
```

Procedure BarrierCC ($epoch$, $isLeader$) for process p_i

```
29 if isLeader then
30 |  R := epoch
31 else
32 |  await R = epoch
```

Procedure GetTag ($epoch$, i) for process p_i

```
33 e0 := E[i][0]
34 e1 := E[i][1]
35 if e0 = epoch then
36 |  return 0
37 else if e1 = epoch then
38 |  return 1
39 else
40 |  if e0 > e1 then return 1 else return 0
```

Procedure BarrierDSM ($epoch$, $isLeader$) for process p_i

```
   // try fast path
41 if R := epoch then return
   // slow path, reset CAS object first
42 ⟨secldr, ltag⟩ := C
43 if secldr ≠ ⊥ then
44 |  if ltag ≠ GetTag(epoch, secldr) then
45 |  |  CAS(&C, ⟨secldr, ltag⟩, ⟨⊥, 0⟩)
   // elect secondary leader
46 tag := SetTag(epoch)
47 if isLeader then
   |  // open the barrier
48 |  R := epoch
49 |  ⟨secldr, ltag⟩ := CAS(&C, ⟨⊥, 0⟩, ⟨i, tag⟩)
50 |  if secldr = ⊥ then
51 |  |  secldr := i
   |  // signal secondary leader
52 |  S[secldr] := epoch
53 else
54 |  ⟨secldr, ltag⟩ := CAS(&C, ⟨⊥, 0⟩, ⟨i, tag⟩)
55 |  if secldr = ⊥ then
56 |  |  secldr := i
57 |  |  await S[i] = epoch
   // wait for secondary leader
58 BarrierSub(epoch, secldr)
```

Procedure SetTag ($epoch$) for process p_i

```
59 tag := GetTag(epoch, i)
60 E[i][tag] := epoch
61 return tag
```

Figure 2: RMR-efficient barrier with unknown leader.

to the conditional statements at line 63 and line 71. Following the first failure, C holds a value ≥ 0 and less than the current epoch number. Under this condition, processes proceed to line 64 to elect a leader, and the latter process updates the value of C to a negative value ($-epoch$), indicating that recovery is in progress. The leader then resets B at line 66, and updates C to a positive value ($epoch$) at line 67, indicating that recovery is complete. The leader then enters the barrier at line 68. A process that lost the leader election at line 64 enters the barrier as a non-leader at line 70.

Some processes may begin executing the recovery protocol after the leader has already been chosen in the current epoch, and before the leader reaches line 67. Such processes read a negative value ($-epoch$) from C at line 62, and then enter the barrier as a non-leader at line 72. If the recovery protocol in the previous epoch was interrupted by a failure, then a process may also read a negative value $> -epoch$ and < 0 from C at line 62, in which case it proceeds to line 64, similarly to the case when C holds a value ≥ 0 and $< epoch$. The leader election is then repeated for the current epoch.

Theorem 4.1 asserts the correctness properties of the transformation presented in Figure 3.

Theorem 4.1. *The target mutex implemented using the transformation presented in Figure 3 satisfies mutual exclusion, ensures starvation freedom if B ensures starvation freedom in failure-free histories, ensures bounded exit if B ensures bounded exit, and has RMR complexity $O(f(B))$ where $f(B)$ denotes the sum of the RMR complexity of B and the RMR cost of resetting B at line 66.*

The statement of Theorem 4.1 intentionally omits the bounded recovery property because the recovery protocol presented in Figure 3 may perform busy-waiting inside the barrier operations. In particular, a process that enters the barrier as a non-leader may busy-wait at lines 70 or 72. However, the recovery protocol is bounded

Procedure Recover (*epoch*) for process p_i

```
62 cur := C
63 if −epoch < cur < epoch then
64     ret := CAS(&C, cur, −epoch)
65     if ret = cur then
66         reset B to its initial state
67         C := epoch
68         Barrier (epoch, true)
69     else
70         Barrier (epoch, false)
71 else if cur = −epoch then
72     Barrier (epoch, false)
```

Shared variables:

- B: conventional mutex (base mutex)
- C: readable/writable **CAS** object, **init** 0

Private variables:

- cur, ret: integers, uninitialized

Procedure Enter (*epoch*) for process p_i

```
73 B.Enter( )
```

Procedure Exit (*epoch*) for process p_i

```
74 B.Exit( )
```

Figure 3: Transformation of conventional mutex to recoverable mutex.

in passages that begin in a state where C holds the value of the current epoch. This includes all passages occurring in failure-free histories, provided that the initial epoch number matches the initial value of the shared variable C. It also includes passages that begin after some process has completed lines 66–67 after the most recent failure.

The remainder of this section is dedicated to the proof of Theorem 4.1. The analysis begins with a proof that the base mutex B is accessed correctly.

Lemma 4.2. *No process executes B.**Enter**() or B.**Exit**() while another process is resetting B at line 66. Furthermore, B is reset at most once in each epoch, and no process executes B.**Enter**() or B.**Exit**() in a given epoch e before B is reset in epoch e at line 66.*

Proof. Consider an arbitrary epoch e. Suppose that processes continue to take steps, not necessarily in a fair order, in epoch e. Since C can only be assigned a value of $-e$ (line 64) or $+e$ (line 67) in epoch e, and since C is initialized to a value less than any epoch number, it follows that $-e < C < e$ holds at the beginning of epoch e. Assuming that processes continue to take steps and there is no failure, the next two transitions in the value of C are $C = -e$ at line 64 and $C = e$ at line 67. Furthermore, at most one process p_i succeeds in swapping $-e$ into C at line 64, and only this process reaches lines 66–67 in epoch e. Due to the conditional statements at lines 63 and 71, no process can complete the recovery protocol until p_i reaches line 67, and after p_i has done so, both conditions are false. As a result, at most one process p_i resets B at line 66 in epoch e, no process completes the recovery protocol until after p_i reaches line 67 after resetting B, and no process executes lines 66–67 again after p_i has done so. These statements imply the lemma since B is only reset at line 66. □

Lemma 4.3. *The target mutex algorithm ensures mutual exclusion.*

Proof. The base mutex B maintains its correctness properties because it is accessed correctly by Lemma 4.2 and the structure of the transformation (lines 73 and 74). Therefore, the critical section of the target algorithm is protected by B. □

Lemma 4.4. *If B ensures starvation freedom in failure-free histories, then the target mutex algorithm satisfies starvation freedom.*

Proof. Consider any fair execution history H of the target mutex. Suppose for contradiction that some process p_i leaves the NCS in H and never enters the CS despite a finite number of failures in H. Let e be the epoch number corresponding to the suffix of H after the last failure. Since p_i has an incomplete super-passage after this last failure, and since H is fair, it follows that p_i takes infinitely many steps in epoch e but does not enter the CS. This implies that p_i is stuck forever in epoch e. Then p_i must be stuck in a barrier or in some access to the base mutex B since the rest of the algorithm is wait-free.

Case 1: p_i is stuck in a barrier at line 68, line 70, or line 72. The barrier leader never gets stuck by the correctness properties of the barrier (Theorem 3.3), and so p_i must be stuck at line 70 or line 72 where it accesses the barrier as a non-leader. If p_i is stuck at line 70 then it read $-e < C < e$ at line 62, and then failed to swap $-e$ into C at line 64, which implies that some other process p_j won the CAS operation. Then p_j is the barrier leader who eventually completes line 68 and opens the barrier. In that case the fact that p_i is stuck forever contradicts the correctness of the barrier (Theorem 3.3). On the other hand, if p_i is stuck at line 72 then it read a value $\leq -e$ or $> e$ from C at line 62. Values $< -e$ and $> e$ are not possible in epoch e, and so p_i must have read $C = -e$. Once again, this implies that some process p_j won the CAS at line 64, and eventually opens the barrier at line 68, so the assumption that p_i is stuck contradicts the correctness of the barrier.

Case 2: p_i is stuck in some access to B at line 66, line 73, or line 74. Since line 66 is wait-free sequential code, p_i must be stuck at line 73 or line 74. It follows from Lemma 4.2 that B is first reset exactly once at line 66 before p_i can reach line 73 or line 74, and so the steps executed by all processes at line 73 and line 74 in epoch e constitute some failure-free execution history H of the base mutex B. Furthermore, H is fair and failure-free as well. This follows from several observations: the execution over the target mutex is fair, processes access B in the correct order (i.e., entry first, then exit,

in an alternating sequence), and Case 1 rules out the possibility of a process getting stuck outside of line 73 or line 74. As a result, the fact that p_i is stuck forever in H executing the entry or exit protocol of B contradicts the starvation freedom of B. □

Lemma 4.5. *Letting $f(B)$ denote the sum of the RMR complexity of the base mutex and the RMR cost of resetting B at line 66, the target mutex has worst-case RMR complexity $O(f(B))$.*

Proof. The RMR complexity of the target mutex in one passage is the sum of several components. First, each process accesses the barrier at most once per passage, and this incurs $O(1)$ RMRs by Theorem 3.3. Second, each process accesses B at most three times per passage – to reset it at line 66, to acquire it at line 73, and to release it at line 74 – and each such access costs $O(f(B))$ RMRs. (Recall that B is accessed correctly by Lemma 4.2 and the structure of the transformation, namely the order of lines 73 and 74.) Finally, each process incurs $O(1)$ additional RMRs per passage executing other lines of code. □

Proof of Theorem 4.1. The mutual exclusion property follows from Lemma 4.3. Preservation of starvation freedom follows from Lemma 4.4. Preservation of bounded exit follows from the simple structure of the exit protocol (line 74). The RMR complexity property follows from Lemma 4.5. □

In addition to the properties stated in Theorem 4.1, the transformation has a feature not provided by prior RME algorithms: it guarantees a weak form of starvation freedom even when processes are not required to recover following a failure. This new liveness property is stated formally in Definitions 4.6–4.7.

*Definition 4.6 (**Weak Fairness**).* A history H is *weakly fair* if it is finite, or if it is infinite and every process that participates either executes infinitely many steps, or stops taking steps after completing a failure-free passage, or stops taking steps after a crash step.

*Definition 4.7 (**Weak Starvation Freedom**).* For any infinite weakly fair history H, if a process p_i leaves the NCS in some step of H then eventually p_i itself enters the CS, or p_i stops taking steps after a crash step, or else there are infinitely many crash steps in H.

Theorem 4.8. *The target mutex implemented using the transformation presented in Figure 3 ensures weak starvation freedom if B ensures starvation freedom in failure-free histories.*

Proof. The theorem follows by the same proof as for Lemma 4.4. □

4.2 Transformation 2: RME to CSR RME

The second transformation adds the CSR property to an existing RME algorithm (base mutex). The pseudo-code of the transformation is presented in Figure 4 in black-colored font. The figure contains also pseudo-code in gray-colored font for Transformation 3 (described later) which should be disregarded for now. Critical section ownership is tracked using a pair of shared variables: *inCSpid* stores the ID of the process currently in the CS, or the negation of this ID if that process is re-entering the CS following a failure; *inCSepoch* stores the epoch number of the process who last entered the CS.

A process p_i accessing the target mutex first executes the recovery protocol of the base mutex at line 75. Next, it determines whether some other process is currently in (or dangerously near) the CS. If p_i reads its own ID, or its negation, from *inCSpid* at line 76, then this indicates that p_i must re-enter the CS before any other process enters the CS. In that case, p_i proceeds to execute the base entry protocol at line 88. After completing this line, p_i ensures that the current epoch number is recorded in *inCSepoch* at line 89. This additional information helps other processes distinguish between the case when a process failed in the CS in a prior epoch and must be allowed to re-enter the CS, versus the case when this process entered the CS in the current epoch and does not require any special treatment. Next, p_i writes the negation of its ID to *inCSpid* at line 91, indicating that it is about to re-enter the CS. In the exit protocol, p_i first checks at line 100 whether it had re-entered the CS after an earlier failure, which is the case when $inCSpid = -i$ (see line 91). If so, then p_i opens the $BR1$ barrier at line 102. This barrier is used to barricade other processes inside the recovery protocol (see line 80) in order to allow p_i to re-enter the CS unimpeded. Finally, p_i resets *inCSpid* to $\perp$ at line 102 and completes the base exit protocol at line 105.

A process p_i that does not read its own ID or its negation from *inCSpid* at line 76 of the recovery protocol must check whether another process p_j may be recovering from a failure in the CS, in which case p_i must wait for p_j instead of competing for the base mutex. This case is identified at lines 78–79, which check whether $inCSpid \neq \perp$ and $inCSepoch \neq epoch$. The latter condition implies that p_j must re-enter the CS, and must be given priority over p_i. In that case p_i waits for p_j at a barrier at line 80. Even if $inCSepoch = epoch$, it is possible that p_j is also re-entering the CS and has progressed beyond line 89, however in that case p_j has already cleared the base entry protocol at line 88 and so it is safe for p_i to race against p_j without waiting on the barrier. If p_i does wait, then p_j eventually opens the barrier at line 102 of the exit protocol, at which point p_i is free to compete for the base mutex with other processes. In the entry protocol, p_i then executes the base entry protocol at line 88, ensures that the *inCSepoch* variable stores the current epoch number at line 89, and then writes its ID to *inCSpid* at line 93. In the exit protocol, p_i resets *inCSpid* at line 104 and then completes the base exit protocol at line 105.

Theorem 4.9 asserts the correctness properties of the transformation. The proof is omitted due to lack of space.

Theorem 4.9. *The target mutex implemented using the transformation presented in Figure 4 satisfies mutual exclusion, ensures CSR, ensures SF if B ensures SF, and has the same RMR complexity asymptotically as B.*

The statement of Theorem 4.9 intentionally omits the bounded exit property because the transformation presented in Figure 4 does not preserve BE. This is due to the barrier operation at line 102 of the exit protocol, which may internally perform busy-waiting. However, if the barrier is implemented using the algorithm described in Section 3 then the transformation does ensure BE in two special cases: (1) in the CC model, where the barrier algorithm for the leader is wait-free; and (2) in the DSM model, in a passage where a

Shared variables:

- B: base mutex, recoverable
- $BR1$,$BR2$: Barrier object
- $inCSpid$: read/write register, **init** $\perp$
- $inCSepoch$: read/write register, **init** 0
- $h[1..N]$: **array** of **Boolean**, **init false**
- $hInd$: read/write register, **init** 1
- $hEpoch$: read/write register, **init** 0

Procedure Recover ($epoch$) for process p_i

```
75 B.Recover(epoch)
76 if inCSpid = i ∨ inCSpid = −i then
       // CS re-entry after failure
77     proceed to entry protocol
78 else if inCSpid ≠ ⊥ then
79     if inCSepoch ≠ epoch then
80         BR1.Barrier (epoch, false)
81 if hEpoch ≠ epoch ∧ h[|hInd|] = true ∧ inCSpid ∉
       {hInd, −hInd} then
82     if |hInd| = i then
           // Privileged process recovery
83         hInd := −i
84         proceed to entry protocol
85     else
86         BR2.Barrier (epoch, false)
```

Procedure Enter ($epoch$) for process p_i

```
87 h[i] := true
88 B.Enter(epoch)
89 inCSepoch := epoch
90 if inCSpid = i ∨ inCSpid = −i then
       // CS re-entry after failure
91     inCSpid := −i
92 else
93     inCSpid := i
94 h[i] := false
95 if hEpoch ≠ epoch ∧ ¬(inCSpid < 0 ∧ |inCSpid| ≠
       |hInd| ∧ h[|hInd|] = true) then
96     hEpoch := epoch
97     if hInd < 0 then
98         BR2.Barrier (epoch, true)
99     hInd = (|hInd| mod N) + 1
```

Procedure Exit ($epoch$) for process p_i

```
100 if inCSpid = −i then
        // CS re-entry after failure
101     inCSpid := ⊥
102     BR1.Barrier (epoch, true)
103 else
104     inCSpid := ⊥
105 B.Exit(epoch)
```

Figure 4: Transformation of recoverable mutex to CSR and FRF recoverable mutex.

process does not re-enter the CS following a failure, which includes all failure-free executions.

Similarly to BE, the transformation does not preserve bounded recovery due to the barrier operation at line **80**. BR is guaranteed only in passages where the recovery of B at line **75** completes in a bounded number of steps, and where the value of $inCSepoch$ matches the current epoch number, which enables barrier bypass at line **79**.

4.3 Transformation 3: CSR RME to CSR and Failures-Robust Fair RME

Transformations 1 and 2 do not guarantee fairness in executions containing infinitely many failures in the sense that some processes may enter the CS infinitely often while other processes starve. This is because Transformation 1 resets the base mutex to its initial state after each failure, and Transformation 2 only preserves the starvation freedom (SF) property defined in Section 2, which does not guarantee progress to every process when failures occur infinitely often. Transformation 3, which we present next, prevents this problem by ensuring that a waiting process can be overtaken only a finite number of times even in these executions. We call this new property *Failures-Robust Fairness* (FRF):

*Definition 4.10 (**Failures-Robust Fairness (FRF)**).* For any fair history H containing infinitely many super-passages, if a process p_i leaves the NCS in some step of H then p_i eventually itself enters the CS.

The idea underlying the transformation is a recovery-time helping mechanism. This helping mechanism ensures that once a process p_i starts the base entry section, after at most n epochs in which there are new entries to the CS, p_i is guaranteed to be the next process to enter the CS. The pseudo-code of Transformation 3 is presented in Figure 4 in gray-colored font.[3] The transformation ensures that if the base mutex is starvation-free, then the target mutex satisfies both the CSR and the FRF properties.

The transformation uses the following shared variables. The h array stores a flag per every process, indicating whether or not the process requires recovery-time helping. The $hInd$ variable is an index to the h array, storing the entry number of the *privileged process*, that is, the next process that should be helped (if its flag is set). A process sets its flag in the h array immediately before invoking the base entry code (line **87**). The flag is reset at line **94**,

[3] We note that a transformation adding the FRF property can be applied directly to a recoverable mutex obtained using Transformation 1, resulting in a recovable mutex that satisfies FRF but not CSR.

before entering the CS but after variable *inCSpid* is set (at line **91** or line **93**), thus guaranteeing that p would be the next to enter the CS even in case of a failure immediately after it is reset. The *hInd* index is initialized to 1 and is incremented (modulo N) at line **99** upon a new epoch (line **95**). The *BR*2 barrier (see Section 3) is used for guaranteeing the uninterrupted CS entry of the privileged process upon recovery, if its flag is set.

When a process p_i performs a steady-state failure-free passage, lines **81–86** in the recovery protocol are skipped (because $hEpoch = epoch$) and p_i proceeds to the entry code. In the entry code, p_i sets its h flag at line **87** before invoking the base entry code and resets it at line **94** when it is logically in the CS. Lines **95–99** are also skipped in steady-state failure-free passages. No additional logic is required in the exit section to support the recovery-time helping mechanism.

Upon recovery from a new failure, we consider first the simpler case in which $inCSpid = \perp$ holds. In this case, p_i skips lines **77–80**, and proceeds to evaluate the condition at line **81**. If this is a new epoch and the privileged process requires help, then the condition is true since $inCSpid = \perp$, so the condition of line **82** is evaluated. If $|hInd| = i$, then p_i is the privileged process and it requires help. In this case, p_i sets $hInd = -i$ in order to remember that it needs to execute special actions in the entry section and proceeds into it (lines **83–84**). The other possibility is that p_i is not the privileged process. In this case, p_i waits for the privileged process at line **86**, permitting the latter's unobstructed entry to the CS.

If p_i reaches the entry section from line **84**, then it executes the base entry protocol. After completing the base entry section, p_i sets $inCSpid = i$ and then resets its h flag (lines **93–94**); it no longer requires help, since it is already logically in the CS and the re-entry mechanism ensures it would return to it if the system fails before it exits. Since this is a new epoch and $inCSpid = hInd$ now holds, the condition at line **95** is satisfied and so p_i updates the epoch number, increments *hInd* modulo N and opens the *BR*2 barrier (lines **96–99**), allowing those processes that may wait at line **86** to proceed to the entry section.

The other possible case is that $|inCSpid| = j$ holds upon recovery, for some process ID j. If there is no privileged process, then the CSR code ensures that p_j re-enters the CS. Otherwise there are two subcases to consider. If there is a privileged process $p_i \neq p_j$ then, as we prove, Transformation 3 ensures that the next two entries to the CS are by p_j followed by p_i. If $i = j$, then the algorithm ensures that p_i enters the CS via the CSR code and the helping code is not executed. In all cases, lines **95–99** guarantee that once the privileged process enters the CS (or does not require help in the first place), *hInd* is incremented modulo N, thus ensuring the FRF property.

Theorem 4.11 asserts the correctness properties of Transformation 3. The proof is omitted due to lack of space.

Theorem 4.11. *The target mutex generated by Transformation 3 maintains the CSR property. If the base mutex satisfies SF, then the target mutex obtained from it by Transformation 3 satisfies the FRF property. The target mutex has the same RMR complexity asymptotically as the base mutex.*

The transformation presented in Figure 4 preserves the bounded exit property in the same special cases as the recoverable-to-CSR version, which omits the additional lines of pseudo-code typeset in gray-colored font. These cases are discussed at the end of Section 4.2. Similarly, the full transformation does not preserve bounded recovery due to the barrier operations at lines **80** and **86**. This is true even in some passages where *inCSepoch* matches the current epoch number because of the additional synchronization performed at line **86** for the FRF property.

5 RELATED WORK

Mutual exclusion (ME) is one of the oldest topics in distributed computing [3, 11, 30, 31]. Recent research in this space has focused on *local spin* algorithms, which guarantee bounded RMR complexity per passage by busy-waiting only on locally accessible shared variables. Local spinning bounds the communication overhead associated with synchronization and supports multi-core scalability [4, 10]. For the class of N-process algorithms that use reads, writes and comparison primitives (defined in [3]), the tight bound on worst-case RMRs is $\Theta(\log N)$. Yang and Anderson [32] proved the upper bound in the CC and DSM models using an arbitration tree modeled after Kessels [23]. Attiya, Hendler, and Woelfel [5] later proved the matching lower bound, building on a series of earlier results [2, 9, 12, 15]. Sub-logarithmic RMR complexity is attainable on expectation using randomization [6, 13, 14, 20], and $O(1)$ RMR complexity is attainable deterministically if additional atomic primitives are available, such as Fetch-And-Store or Fetch-And-Add [4, 19, 26, 27].

Solutions to the ME problem generally assume reliable processes since synchronization is achieved by busy-waiting, although variations of the problem are solvable in some failure models. Lamport and Taubenfeld define several such models for algorithms based on single-writer registers, assuming that failures affect the values of such shared variables in specific ways [25, 31]. Lamport's Bakery [24] tolerates crash failures provided that any registers owned by the failed process are reset to zero [24]. Bohannon et al. instead consider a failure model in which process crashes are permanent and do not affect the values of shared variables, which complicates recovery [7, 8]. Their solutions rely on operating system support for failure detection and assume for simplicity that recovery is performed in a dedicated process that is itself reliable. Michael and Kim's fault-tolerant mutex similarly relies on failure detection but does not require a dedicated recovery process [28]. Instead, a process that is waiting for a lock may "usurp" the lock if it determines that the previous lock holder has crashed permanently.

Recoverable mutual exclusion (RME) was formalized by Golab and Ramaraju [18]. Two of Golab and Ramaraju's RME solutions are local-spin in both the CC and DSM models, and both incur $\Omega(\log N)$ RMRs per passage in the worst case. Ramaraju's earlier RME algorithm incurs $\Theta(N)$ RMRs in the worst case and relies on a specialized Fetch-And-Store-And-Store (FASAS) instruction, which is equivalent to the atomic composition of a Fetch-And-Store followed by a write that saves the fetched value in NVRAM [29]. The best upper bound proved by Golab and Ramaraju is $O(\log N)$ RMRs in the CC and DSM models, and requires atomic reads and writes only. Golab and Hendler broke the logarithmic barrier by presenting an RME algorithm that uses single-word Fetch-And-Store and Compare-And-Swap and incurs $O(\log N / \log \log N)$ RMRs per passage in the CC model, as well as an algorithm that uses specialized double-word Read-Modify-Write operations (FASAS

and CAS) and incurs $O(1)$ RMRs per passage in the CC and DSM models [16].

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. The new breakthrough in this work is the use of wait-free recovery code, which is 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 [21]. Operations on the f-array are naturally idempotent, which allows the algorithm to avoid busy-waiting in the recovery protocol.

Published solutions to the RME problem [16, 18, 22] assume independent failures, and work correctly if failures are system-wide, but do not benefit from the stricter failure model in terms of RMR complexity. It is not known whether RME is inherently less costly in the model with system-wide failures because the tight RMR complexity bound for RME has not yet been established for the model with independent failures.

The epoch numbers used in the algorithms presented in this paper are inspired by prior work on fault-tolerant computation. The failure detector $\diamond S_e$ proposed by Aguilera, Chen and Toueg associates an epoch number with each process deemed to be currently up [1]. The completeness property of this failure detector requires that the epoch number of an unstable process eventually increases monotonically as long as the process is included in the failure detector's output. Epoch numbers in this paper are used somewhat similarly for detecting system-wide failures, but our model does not use a formal failure detector abstraction in which every process has access to its own failure detector module. We assume instead that processes have common knowledge of the current epoch number, which is obtained by querying the operating system during process initialization.

6 CONCLUSION

We considered an RME model where failures are system-wide and processes receive additional information from the environment regarding the occurrence of the failure. We proved that, in this model, an $O(1)$ RMRs conventional mutex can be transformed to an $O(1)$ RMRs recoverable mutex using only commonly supported single-word synchronization primitives. In future work, we intend to investigate the gap in RMR complexity between our upper bounds for the model with system-wide failures and prior results in the model with individual failures [16, 18, 22].

REFERENCES

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, 2000.

[2] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, 2002.

[3] 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.

[4] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proc. of the 24th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 33–48, 2013.

[11] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[12] 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.

[13] G. Giakkoupis and P. Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proc. of the 44th Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012.

[14] 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.

[15] 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.

[16] 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.

[17] W. Golab, D. Hendler, and P. Woelfel. An O(1) RMRs leader election algorithm. *SIAM J. Comput.*, 39(7): 2726–2760, 2010.

[18] W. Golab, and A. Ramaraju. Recoverable mutual exclusion. In *Proc. of the 35th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 65–74, 2016.

[19] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[20] D. Hendler and P. Woelfel. Randomized mutual exclusion with sub-logarithmic RMR-complexity. *Distributed Computing*, 24(1):3–19, 2011.

[21] P. Jayanti. F-arrays: Implementation and Applications. In *Proc. of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 270–279.

[22] 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.

[23] J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

[24] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[25] L. Lamport. The mutual exclusion problem: part II – statement and solutions. *Journal of the ACM*, 33(2):327–348, 1986.

[26] 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.

[27] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

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

[29] 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.

[30] M. Raynal. *Algorithms for Mutual Exclusion.* MIT Press, 1986.

[31] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming.* Prentice Hall, 2006.

[32] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.