# Recoverable Mutual Exclusion in Sub-logarithmic Time

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

Danny Hendler†
Department of Computer Science
Ben-Gurion University, Israel
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 on the RME problem established an upper bound of $O(\log N)$ RMRs in an asynchronous shared memory model with $N$ processes that communicate using atomic read and write operations, prompting the question whether sub-logarithmic RMR complexity is attainable using common read-modify-write primitives. We answer this question positively in the cache-coherent model by presenting an RME algorithm that incurs $O(\log N / \log \log N)$ RMRs and uses read, write, Fetch-And-Store, and Compare-And-Swap instructions. We also present an $O(1)$ RMRs algorithm that relies on double-word Compare-And-Swap and a double-word variation of Fetch-And-Store. Both algorithms are inspired by Mellor-Crummey and Scott's queue lock.

## 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

The traditional coupling of volatile main memory with non-volatile secondary storage in computer architectures has led to a separation of program state into operational data stored using in-memory data structures, and recovery data stored using sequential on-disk structures such as transaction logs. In the event of a system-wide failure, such as a power outage, the in-memory data structures are lost and must be reconstructed entirely from the recovery data, making the software system temporarily unavailable. As a result, the design of in-memory data structures emphasizes disposable constructs optimized for parallelism, in contrast to the structures that hold recovery data, which cannot be discarded upon a failure and which benefit less from parallelism since their performance is limited by the secondary storage.

Recent developments in non-volatile main memory (NVRAM) media foreshadow the eventual convergence of primary and secondary storage into a single layer in the memory hierarchy that combines the performance benefits of conventional main memory with the durability of secondary storage. Traditional log-based recovery techniques can be applied correctly in such systems but fail to take full advantage of the parallelism enabled by allowing processing cores to access recovery data directly using memory operations rather than slow block transfers from secondary storage. As a result, harnessing the performance benefits of NVRAM-based platforms requires a careful rethinking of recovery mechanisms.

This paper focuses specifically on recoverable lock-based data structures, in which synchronization is achieved using mutual exclusion [10]. The abstract problem of implementing locks that tolerate crash-recovery failures was formalized recently by Golab and Ramaraju as Recoverable Mutual Exclusion (RME) [16]. In short, RME is a generalization of the standard mutual exclusion problem (ME) whereby a process that crashes while accessing (i.e., acquiring, releasing, or holding) a lock is resurrected eventually and allowed to execute recovery actions. This abstraction captures both individual process failures and system-wide failures, such as due to power outages. In contrast, the ME problem assumes that a process never fails while accessing a lock, as otherwise a conventional lock becomes vulnerable to starvation.

RME is, informally speaking, a harder problem than conventional ME because the solution must deal simultaneously with both concurrency and failures. However, no formal proof of this point has been published, and indeed little is known about the RME problem as compared to ME, which has been studied thoroughly [2, 30]. Recent work on ME established tight 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. For the class of algorithms that use only reads, writes and comparison primitives (e.g., CAS), the tight bound for acquiring and releasing an $N$-process mutex lock once is $\Theta(\log N)$ RMRs in the worst case [4, 32]. The same bound applies for RME [16]. For the class of algorithms that use additional primitives such as Fetch-And-Store, ME is solvable

*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).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*PODC '17, July 25-27, 2017, Washington, DC, USA*
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4992-5/17/07…$15.00
http://dx.doi.org/10.1145/3087801.3087819

in only $O(1)$ RMRs but initial attempts to establish the same upper bound for RME were unsuccessful, with all known solutions incurring $\Omega(\log N)$ RMRs in the worst case [16]. This points to the difficulty of using read-modify-write primitives in a crash recovery model, where failures can annihilate the response of a memory operation (e.g., the value fetched by a Fetch-And-Store) before it can be written to shared memory. Thus, a recovering process is unable to determine this response directly using its private state, or even decide whether the read-modify-write primitive was executed at all.

The technical contribution of this paper is a pair of upper bounds on the RMR complexity of the RME problem that improve upon prior results. First, we prove an $O(1)$ bound in both the cache-coherent and distributed shared memory models using double-word Compare-And-Swap and a double-word variant of Fetch-And-Store (Section 3). This matches the RMR complexity of solving ME using reads, writes, and common single-word read-modify-primitives. Second, we establish an $O(\log N/\log\log N)$ bound in the cache-coherent model using common single-word primitives, and thus prove a time complexity separation versus the class of algorithms that use only single-word reads, writes and comparison-primitives (Sections 4–5). These are the first sub-logarithmic upper bounds on worst-case RMR complexity for the RME problem, ignoring trivial solutions that assume queue operations are supported directly in hardware.

## 2 MODEL

Our model follows [16]. 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 protects a shared resource, by following the execution path illustrated in Figure 1. In the *non-critical section* (NCS), processes do not access the lock or the shared resource. In the *recovery protocol*, denoted **Recover**, a process cleans up the internal structure of the lock, if needed, following a crash failure. In the *entry protocol*, denoted **Enter**, a process attempts to acquire the lock and waits for the lock to become available. The entry protocol is sometimes modeled as a bounded section of code called the *doorway*, which determines the order in which processes access the shared resource, followed by the *waiting room*, where processes wait for their turn. Access to the shared resource is confined to the *critical section* (CS). In the *exit protocol*, denoted **Exit**, a process releases the lock and makes the shared resource available to others.

**loop forever**
- Non-Critical Section (NCS)
- **Recover**
- **Enter** { Doorway / Waiting Room }
- Critical Section (CS)
- **Exit**

**Figure 1: Execution path of a process participating in RME.**

Correctness properties for mutex algorithms are expressed in reference to *histories* that record the actions of processes as they execute the pseudo-code in Figure 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 bounded local computation such as arithmetic operations, accessing one or more private variables,[1] and advancing the program counter. A crash step denotes a process failure that resets the private variables of a process to their initial values, including the program counter, which is reset to the non-critical section (top of the loop in Figure 1). The next step a process takes after a crash step 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 occurs first. 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. A *super-passage* is a maximal non-empty sequence of consecutive passages executed by the same process where (only) the last passage in the sequence is failure-free.

Process failures may influence the time complexity of a passage through a recoverable mutex algorithm. We formalize the notion of potential influence by reasoning precisely about interleaving among passages. We say that two passages *interfere* if their super-passages are concurrent, meaning that neither super-passage ends before the other begins. We call a passage *cleanup-concurrent* if it begins in cleanup, or if the passage interferes with some passage that begins in cleanup.

The set of histories generated by a mutex algorithm is prefix-closed. Given a history $H$ and a process $p_i$, we say that $p_i$ is *active in* $H$ if it takes at least one step in $H$. A history $H$ is *fair* if it is finite, or if it is infinite and every active process either executes infinitely many steps or stops taking steps after completing a failure-free passage. We assume that the critical section is bounded.

The fundamental correctness properties of RME algorithms are defined as follows:

**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 $H$.

**Terminating Exit (TE):** For any infinite fair history $H$, if a process $p_i$ begins executing the exit protocol then it eventually returns to the NCS or crashes.

**Bounded Recovery (BR):** For any infinite 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 cleanup-concurrent.

**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 its own crash failure.

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

[1] In this paper a "private" variable is accessed by only one process and stored in a volatile CPU register. Variables with local (e.g., function) scope are private.

processor-to-memory interconnect, and its precise definition depends on the shared memory hardware architecture. In this paper we consider both the cache-coherent (CC) and distributed shared memory (DSM) models [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, which is determined statically at initialization. A mutex algorithm is called *local spin* if its worst-case RMR complexity per passage is bounded.

The use of volatile caches in the CC model complicates access to non-volatile main memory. Shared memory operations must be combined with explicit cache line flushes or write-backs, for example using Intel's clflush or clwb instructions, to ensure that their effects are persistent. The interleaving of such instructions with shared memory operations is outside the scope of this paper.

## 3 AN $O(1)$ RMRS ALGORITHM

The RME problem can be solved easily in $O(1)$ RMRs per passage in the CC and DSM models if sufficiently powerful primitives are available. In particular, if a queue of process IDs can be implemented with fundamental operations incurring $O(1)$ RMRs then a queue-based lock can be constructed as described in [14]. However, such a queue is itself difficult to implement, and hardware transactional memory [20] does not solve the problem fully since memory transactions are not guaranteed to complete in a bounded number of RMRs under contention. The RME algorithm presented in this section relies instead on double-word atomic read-modify-write instructions. The design is inspired by Mellor-Crummey and Scott's queue lock (MCS lock) [27], whose pseudo-code is included for completeness in Figure 2. [2]

The MCS lock maintains a linked list of *QNode* structures that determines the order of entry into the CS. Each QNode contains a link *next* to the successor, as well as a spin variable *locked* that is used to transfer ownership of the CS from a predecessor to a successor. The QNode of process $p_i$ is denoted by $q_i$. In total, two memory operations are required to append a QNode in the entry protocol: (a) The tail of the list is updated using an atomic Fetch-And-Store (FAS) operation at line 2, which returns a reference to the predecessor's QNode or $\perp$ if the list was empty. This reference is stored in a private variable *prev*. (b) The predecessor's *next* pointer is updated, if required, at line 5.

The exit protocol similarly uses multiple memory operations to delete a QNode from the linked list: (c) If no successor is identified by reading $q_i.next$ at line 7, then a Compare-And-Swap (CAS) operation is executed at line 8 to swing the tail pointer $T$ from the caller's QNode back to $\perp$. A successful CAS removes the QNode explicitly from the linked list. (d) A failed CAS implies the arrival of a successor whose QNode is determined by busy-waiting on the predecessor's *next* pointer at line 10. (e) Finally, the successor is notified by resetting its *locked* variable at line 11, which marks the successor's QNode as the new head node of the linked list while preserving the *next* pointer from the (former) predecessor's QNode.

[2] The notation & in the pseudo-code means "address of", as in C/C++.

**Define** QNode: struct { *next*: **ref to** QNode, *locked*: **Boolean** }

**Shared variables:**

- $T$: **ref to** QNode, **init** $\perp$
- $q_i$: **ref to** QNode structure **init** $\langle \perp, \textbf{false} \rangle$ and local to process $p_i$ in DSM model

**Private variables:**

- *prev*: **ref to** QNode, uninitialized

**Procedure Enter** ( )

```
1  q_i.next := ⊥
   // insert QNode into linked list
2  prev := FAS(&T, q_i)
   // check for predecessor
3  if prev ≠ ⊥ then
       // link with predecessor
4      q_i.locked := true
5      prev.next := q_i
       // wait for predecessor to release lock
6      await q_i.locked ≠ true
```

**Procedure Exit** ( )

```
   // check for successor
7  if q_i.next = ⊥ then
       // no known successor, try to unlink from T
8      if CAS(&T, q_i, ⊥) = true then
           // no successor, lock idle
9          return
       // wait for successor to link with p_i
10     await q_i.next ≠ ⊥
   // transfer lock ownership to successor
11 q_i.next.locked := false
```

**Figure 2: Mellor-Crummey and Scott's queue lock for process $p_i$.**

Figure 3 illustrates a possible configuration of the QNode structures in an execution of the MCS algorithm where processes arrive in the following order: $p_1, p_2, p_3, p_4$. Process $p_1$ has already returned to the NCS with $q_1.locked$ = **false**, and $q_1.next = q_2$ implies that $p_1$ passed the CS to $p_2$ at line 11 rather than by unlinking itself at line 8. In other words, $p_2$ appended its QNode at line 2 before $p_1$ executed the CAS instruction in **Exit**. Accordingly, process $p_2$ is currently in the CS with $q_2.locked$ = **false** and $prev = q_1$. Process $p_3$ is the successor of $p_2$ with $prev = q_2$. Process $p_4$ is the successor of $p_3$, and has not yet linked with $p_3$ at line 5 of **Enter**, which is why $q_3.next = \perp$. Similarly, $q_4.next = \perp$ since $p_4$ does not yet have a successor. The *prev* pointers are stored in private variables and do not appear in the QNode structures.

The non-atomic collection of atomic memory operations used in the entrance and exit protocols makes it difficult to recover the MCS lock following a crash failure as a process $p_i$ cannot determine

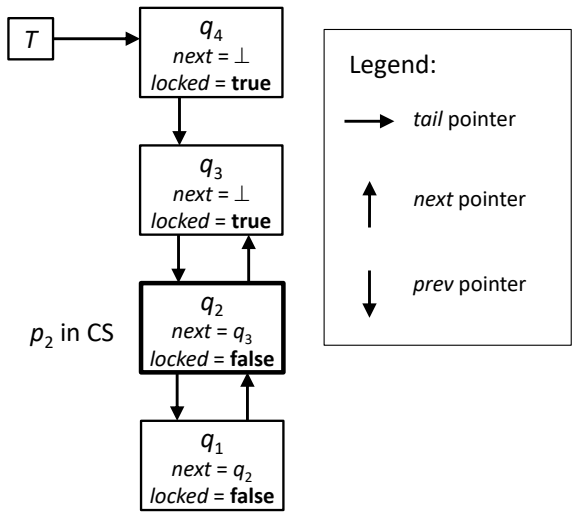


**Figure 3: Example configuration of QNode structures in an execution of the MCS algorithm for processes $p_1, p_2, p_3, p_4$.**

only from its private state whether a failure occurred immediately before the FAS at line 2, in which case $p_i$ must attempt to append its QNode again using FAS, versus immediately after, in which case the QNode is already in the linked list and $p_i$ must not append it again. The exit protocol suffers from a similar problem around the CAS instruction at line 8, whose response is needed to determine whether $p_i$'s QNode is still in the linked list. A scan of the linked list from the tail to find $p_i$'s QNode is not helpful since it requires following the predecessor pointers, which are never written to main memory. For example, in Figure 3 the QNodes $q_3$ and $q_4$ become disconnected following a failure of process $p_4$, making a scan of the linked list from the tail infeasible. Even if a *prev* field was added to the QNode as a shared variable, a failure between the FAS instruction and the write operation on this new field could lead to breaks in the linked list, making a complete traversal infeasible even in the absence of concurrent updates to the list.

Aside from the problematic FAS instruction, the MCS algorithm contains several write operations that cannot be repeated safely after a failure: one in the entry protocol at line 5, where predecessor's *next* pointer is written, and the other in the exit protocol at line 11, where the successor's *locked* variable is set to **false** after the successor's QNode is determined at line 10. Failures around such write operations also complicate recovery, although they are arguably easier to deal with compared to the FAS instruction since there is no danger of a lost return value.

We derive a recoverable mutex from MCS by leveraging double-word atomic read-modify-write instructions, whose greater power makes it possible to determine their outcome (or lack thereof) following a crash failure. First, the FAS in the entry protocol is replaced with a Fetch-And-Store-And-Store (FASAS). FASAS takes two memory locations, $M1$ and $M2$, as arguments, as well as a value $V$, and has the following effect: it copies the value at $M1$ to $M2$, and then stores $V$ at $M1$. Thus, FASAS is the atomic composition of FAS and a write of the response to memory. Second, the CAS in the exit

**Define** QNode: struct { *prev*: **ref to** QNode, *next*: **ref to** QNode, *linked*: **Boolean** }

**Shared variables:**

- $T$: **ref to** QNode, **init** ⊥
- $q_i$: **ref to** QNode structure of process $p_i$ **init** $\langle q_i, \perp, \textbf{false}\rangle$ and local to process $p_i$ in DSM model

**Private variables:**

- *prev*: **ref to** QNode or ⊥, uninitialized

**Procedure Recover** ( )

```
// empty, recovery actions integrated into Enter
```

**Procedure Enter** ( )

```
12 prev := q_i.prev
   // check whether previous passage was failure-free
13 if prev = q_i then
14     q_i.next := ⊥
15     q_i.linked := false
       // insert QNode into linked list
16     FASAS(&T, &(q_i.prev), q_i)
17     prev := q_i.prev
   // check for predecessor
18 if prev ≠ ⊥ then
       // link with predecessor
19     DCAS(&(prev.next), &(q_i.linked), ⊥, false, q_i, true)
       // wait for predecessor to release lock
20     await q_i.prev = ⊥
```

**Procedure Exit** ( )

```
   // try to unlink from T
21 if DCAS(&T, &(q_i.prev), q_i, ⊥, ⊥, q_i) = false then
       // wait for successor to link with p_i
22     await q_i.next ≠ ⊥
       // transfer lock ownership to successor
23     FASAS(&(q_i.next.prev), &(q_i.prev), ⊥)
```

**Figure 4: $O(1)$-RMRs recoverable mutex for process $p_i$.**

protocol is replaced by a double-word Compare-And-Swap (DCAS), which swaps values into two memory locations provided that their prior values both match the comparison arguments.[3] These instructions, when used correctly, resolve the technicalities surrounding recovery around line 2 or line 8 of MCS. They can also replace the writes at line 5 and line 11 to ensure that the *next* and *locked* fields are updated at most once irrespective of failures.

Our MCS-based RME algorithm is presented in Figure 4. Like MCS, our recoverable lock incurs $O(1)$ RMRs per passage in the worst case in both the CC and DSM models, and is *context-free*, meaning that neither the number of processes nor their identities

[3]In the cache-coherent model, the FASAS and DCAS instructions must write the updated values of both memory locations atomically to non-volatile memory.

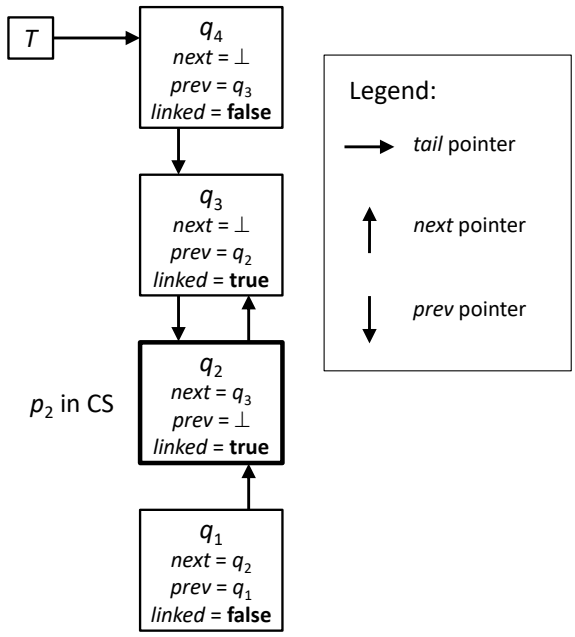


**Figure 5: Example configuration of QNode structures in an execution of the recoverable mutex for processes $p_1, p_2, p_3, p_4$.**

need to be known in advance. The doorway comprises the bounded code at lines 12–16 of the entry protocol, and the recovery protocol is empty since all recovery actions are folded into the entry and exit protocols. In the absence of failures, process $p_i$ reads the initial value $q_i$ from $q_i.prev$ at line 12 and proceeds to reinitialize $q_i$ at lines 14–15. The FASAS at line 16 atomically swings the tail pointer $T$ to $p_i$'s QNode $q_i$ and saves the previous value to $q_i.prev$, which establishes a link from $q_i$ to the predecessor's QNode (if any). Next, $p_i$ inspects $q_i.prev$ at line 17 to determine whether it is the head of the queue. A value of $\perp$ indicates that $q_i$ is the head element, in which case $p_i$ proceeds directly to the CS. Any other value indicates that $q_i$ is not the head and that $p_i$ must create a link from the predecessor's QNode to $q_i$ and then wait. To create such a link, $p_i$ updates the predecessor's *next* field at line 19 using DCAS, which also sets $q_i.linked$ = **true**. At line 20, $p_i$ waits for the predecessor to release the CS by overwriting $q_i.prev$ with $\perp$. The exit protocol then attempts to atomically swing the tail pointer from $q_i$ back to $\perp$, and reset $q_i.prev$ back to $q_i$ using a DCAS at line 21. If the DCAS is unsuccessful, $p_i$ waits at line 22 for the successor to write $q_i.next$, then executes FASAS at line 23, resetting the successor's *prev* pointer to $\perp$ and its own *prev* pointer to $q_i$. Thus, the successor is unblocked atomically with the deletion of $q_i$ from the linked list.

Figure 5 illustrates a possible configuration of the QNode structures in an execution of the recoverable mutex where processes arrive in the following order: $p_1, p_2, p_3, p_4$. Process $p_1$ has already returned to the NCS with $q_1.prev = q_1$, and $q_1.next = q_2$ implies that $p_1$ passed the CS to $p_2$ at line 23 rather than by unlinking itself at line 21. In other words, $p_2$ appended itself at line 16 before $p_1$ executed the DCAS instruction in **Exit**. Accordingly, process $p_2$ is currently in the CS with $q_2.prev = \perp$. Process $p_3$ is the successor of $p_2$ with $q_3.prev = q_2$ and $q_3.linked$ = **true**, reflecting the fact that $q_2.next = q_3$. Process $p_4$ has not yet linked with $p_3$ at line 19 of **Enter**, which is why $q_3.next = \perp$ and $q_4.linked$ = **false**. Similarly, $q_4.next = \perp$ since $p_4$ does not yet have a successor. The *prev* pointers are stored in the QNode structures alongside the *next* pointers to facilitate recovery.

The execution path on recovery from a crash failure is based on the following invariant: $q_i.prev$ points to $q_i$ if $q_i$ is not in the linked list, points to the predecessor's QNode if $q_i$ is in the linked list but not the head element, and equals $\perp$ if $q_i$ is the head node. Thus, the doorway is executed only if $q_i.prev = q_i$ (line 13), and the remainder of the entry protocol is executed only if $q_i.prev \neq \perp$ (line 18). Recovery from a failure in the critical section with $q_i.prev = \perp$ bypasses the entry protocol entirely after line 12, and so $p_i$ is able to re-enter the CS in a bounded number of its own steps. The use of DCAS at line 19 ensures that the *next* pointer is written at most once per super-passage.

In the context of Figure 5, failures are dealt with as follows. If $p_1$ fails and then takes another step then it begins a new passage on recovery. If $p_2$ fails then it bypasses **Enter** via lines 13 and 18 because $q_2.prev = \perp$, and so $p_2$ re-enters the CS. If $p_3$ fails then it repeats the DCAS instruction at line 19 of **Enter** assuming that $q_3.prev = q_2$ continues to hold, and this DCAS fails because $q_3.linked$ = **true**, forcing $p_3$ to wait for the CS at line 20. Alternatively, if $p_2$ leaves the CS concurrently with $p_3$'s recovery then it is possible that $p_3$ executes **Enter** with $q_3.prev = \perp$ as a result of $p_2$ executing line 23, in which case $p_3$ proceeds to the CS. If $p_4$ fails then it proceeds to the DCAS instruction at line 19 of **Enter** assuming that $q_4.prev = q_3$ continues to hold, and this DCAS succeeds because $q_3.next = \perp$ and $q_4.linked$ = **false**. Alternatively, if $p_3$ enters and then leaves the CS concurrently with $p_4$'s recovery then it is possible that $p_4$ executes **Enter** with $q_4.prev = \perp$, in which case $p_4$ proceeds to the CS.

Theorem 3.1. *The algorithm satisfies ME, SF, BR, CSR, and TE. Furthermore, its worst-case RMR complexity per passage is $O(1)$ in the CC and DSM models.*

The ME and SF properties were verified successfully for $N = 3$ using a PlusCal [24] translation of the algorithm and the TLC model checker [25].

## 4 A $K$-PORT EXTENDED MCS LOCK

In this section, we present a *k-port extended MCS lock object* for the cache-coherent (CC) model. As the name implies, this is an extension of the MCS lock, enabling concurrent access by up to $k$ processes out of the $N$ processes in the system. Implementation correctness relies on the *sequential port access* assumption: there are no two concurrent passages on any port. The salient feature of the extended MCS lock is that a process incurs $O(1)$ RMRs per passage unless recovering from a failure that occurred outside the non-critical section (NCS) and critical section (CS), and $O(k)$ RMRs in the worst case. This property is necessary for the algorithm to function correctly as a building block of our $O(\log N / \log \log N)$ RMRs recoverable mutex, which is presented later on in Section 5. In comparison, Golab and Ramaraju's transformation [16, Section 4.2]

applied to MCS achieves the same worst-case complexity but guarantees $O(1)$ RMRs only when a passage is not *cleanup-concurrent*, which is insufficient for our purposes as it allows the failure of one process to inflate the RMR complexity for another process.

The extended MCS lock uses a modified QNode structure (see Figure 6). The fields *locked* and *next* are defined as in MCS (see Section 3). The field *prev* identifies the predecessor, and may hold a special value FAIL when the owner of the QNode is recovering from a failure. The field *nextStep* is new, and holds a pseudo-code line number that identifies the next step to be performed by the process in the entry or exit protocol. It may also assume a special value *inCS*, signifying that the process finished the **Enter** procedure and has not begun executing **Exit**. The *nextStep* field is updated in these parts of the pseudo-code in order to announce the intention of taking the step. Updates of this field are not shown in the pseudo-code for presentation simplicity, and occur in every execution of **Enter** and **Exit**, except when a portion of **Exit** is executed at line 54 or line 58 of **Recover**. Nevertheless, upon recovery, there remains uncertainly regarding whether the fault occurred before or after the indicated step. The other new field is *fin*, which specifies whether the passage that used this QNode record completed the exit protocol successfully (see line 39). We assume that a QNode is never reused following the completion of a failure-free passage, which makes it possible to read *fin* = **true** in a QNode whose process has returned to the NCS or even started a new passage. In addition to QNode structures, the extended MCS lock uses a *tail* variable analogous to *T* in MCS (Figure 2), which points to the QNode of the last process in the waiting queue, or **null** if the queue is empty. Finally, an auxiliary recoverable lock, denoted *rLock*, is used to protect the most sensitive part of the recovery protocol (lines 74–96 in Figure 8). This lock is implemented using a $k$-process instance of the $O(\log N)$ RMRs RME algorithm by Golab and Ramaraju [16, Section 3.2].

The **Enter** procedure (lines 24–31 in Figure 6) receives two parameters: a port number *pNum* and a reference *myNode* to a QNode record, which is allocated newly for each passage except in some recovery situations. The executing process $p$ first initializes the *myNode* record and then stores a reference to it in the *pNum*'th entry of the *nodes* array (lines 24–25). At line 26, $p$ links itself to the queue by swapping *myNode* into *tail*. It then stores *tail*'s previous value to *myNode.prev* at line 27. Process $p$ then checks at line 28 whether or not it has a predecessor. If not, it proceeds immediately to the CS. Otherwise, it prepares for waiting (line 29), links to its predecessor by writing a reference to *myNode* in the *next* field of its predecessor record (line 30), and then waits to be signalled (line 31). Upon being signalled, $p$ proceeds to the CS.

The **Exit** procedure (lines 32–39) receives the port number as its single parameter and uses it to extract the port's QNode record from the *nodes* array (line 32). Then, process $p$ checks whether the *next* field of its record was set by a successor process (line 33). If unset, $p$ attempts to CAS *tail* back to **null** in line 34, which will succeed only if $p$ has no successor. If the CAS is successful, $p$ returns to the NCS after setting the *fin* flag of *myNode* (line 39), which signifies that the passage that used this record has terminated. If the CAS fails, then there is a successor, so $p$ awaits for it to link to *myNode* (line 35) and, once it does, signals it (line 36) and returns to the NCS. Finally, if the *next* field is found to have been set at line 33, $p$ signals its successor at line 38 and returns to the NCS.

The **Recover** procedure (lines 40–65) receives the port number as its single parameter. If this is the first passage on this port or the previous passage was completed (line 41), then there is no need for recovery. In this case, the procedure simply returns and the process continues to execute the entry protocol (line 42). Otherwise, recovery proceeds according to the step a process was about to perform upon failure. If the failure occurred at a point when the FAS at line 26 may have been performed but the *prev* field of the port's record was not updated (line 43), then $p$ checks *prev* and, if it observes a $\perp$ value, determines whether a reference to its QNode from the last passage is visible to anyone by calling the helper function **IsLinkedTo** (line 44), which we describe shortly. If the QNode is not visible, then $p$ repeats the entry protocol using the same QNode (line 45). Otherwise, if *prev* does not point to a QNode (line 46) then $p$ announces that it is about to repair the queue structure by writing *prev* = FAIL (line 47) and then performs repairs by calling helper procedure **RecoverSwap** (line 48), which we describe shortly. Finally, if *prev* does point to a QNode then there is no need to repair the queue, but a failure may have occurred inside **RecoverSwap** while $p$ was holding *rLock*. In this case $p$ makes one passage through *rLock* to recover it (line 50). Execution then continues in the middle of the entry protocol (line 51).

If the failure occurred when the process was about to perform the CAS of line 34 (line 52), then the following cases exist: 1) if *tail* still stores a reference to the *myNode* record (line 53), then the failure occurred *before* the CAS executed and so recovery is done by executing lines 34–39 of **Exit** (line 54), which repeats the CAS; 2) If *tail* equals **null**, then it must be that the CAS was performed successfully, and so execution simply continues with the entry protocol (lines 55–56); 3) otherwise, other processes linked to the queue either before or after the failure and the only way to tell how to proceed is to check whether *myNode* has a successor by calling the **IsLinkedTo** function (line 57). If *myNode* has a successor, then recovery is done by executing lines 35–39 of the **Exit** procedure (line 58), otherwise it must be that the CAS was successfully executed and the process proceeds to the entry section (line 59). If the failure occurred in the **Exit** procedure not when about to perform the CAS of line 34 (line 62), then recovery is done by completing the execution of the **Exit** procedure and then continuing to the **Enter** procedure (line 63). If a failure other than a swap failure occurred in the **Enter** procedure, recovery is done by resuming **Enter** starting from *nextStep* (line 65).

The **IsLinkedTo** function, presented in Figure 7 (lines 66–72), checks whether the QNode record associated with port *pNum* is referenced. It does so by reading the current value of *tail* (line 67) and the current values of the records used by all other ports (lines 68–69). For any such record, the function waits for the *prev* field to be updated to a non-$\perp$ value (line 71). If either *tail* or any of these records holds a reference to *pNum*'s record, **IsLinkedTo** returns **true**, otherwise it returns **false**.

Procedure **RecoverSwap**, presented in Figure 8 (lines 73–96), is invoked by processes that incur *swap failures*, that is, they fail just before or after they perform the FAS of line 26. It is challenging to recover from such failures, because failing processes must be able

**Define** QNode: struct { *locked*: **Boolean**, *prev,next*: **ref to** QNode, *nextStep*: **int**, fin: **Boolean** }

**Shared variables:**

- *nodes*: **array** [0..*k*-1] **of ref to** QNode **init null**
- *tail*: **ref to** QNode **init null**
- *rLock*: *k*-process recoverable lock [16, Sec. 3.2]

**Private variables:**

- *myNode*, *prev*: **ref to** QNode **init null**

**Procedure Enter** (**int** *pNum*, **ref to** QNode *myNode*)

```
24 myNode.next := null, myNode.prev := ⊥, myNode.fin := false
25 nodes[pNum] := myNode
   // insert QNode into linked list
26 prev := FAS (&tail, myNode)
27 myNode.prev := prev
   // check for predecessor
28 if prev ≠ null then
       // link with predecessor
29     myNode.locked := true
30     prev.next := myNode
       // wait for predecessor to release lock
31     await myNode.locked := false
```

**Procedure Exit** (**int** *pNum*)

```
32 myNode := nodes[pNum]
   // check for successor
33 if myNode.next = null then
       // no known successor, try to unlink from tail
34     if CAS (&tail, myNode, null) = false then
           // wait for successor to link with p_i
35         await myNode.next ≠ null
           // transfer lock ownership to successor
36         myNode.next.locked := false
37 else
       // transfer lock ownership to successor
38     myNode.next.locked := false
   // record successful completion of exit protocol
39 myNode.fin := true
```

**Procedure Recover** (**int** *pNum*)

```
40 myNode := nodes[pNum]
41 if myNode = null ∨ myNode.fin then
       // easy case: no failure or failure in NCS
42     continue to Enter(pNum, new QNode)
43 else if myNode.nextStep ∈ {26, 27} then
       // swap failure
44     if myNode.prev = ⊥ ∧ ¬IsLinkedTo(pNum) then
           // not in the queue, nothing to recover
45         continue to Enter(pNum, myNode)
46     else if myNode.prev ∈ {⊥, FAIL} then
           // in the queue, must repair queue structure
47         myNode.prev := FAIL
48         call RecoverSwap(pNum)
49     else
           // recover the auxiliary lock in case a
              failure occurred inside RecoverSwap
50         rLock.Recover(), rLock.Enter(), rLock.Exit()
       // recovery complete, proceed to waiting room
51     continue to Enter(pNum, myNode) at line 28
52 else if myNode.nextStep = 34 then
       // failure in exit protocol
53     if tail = myNode then
           // still at the tail, attempt CAS again
54         execute lines 34–39 of Exit(pNum)
55     else if tail = null then
           // no longer in the queue
56         continue to Enter(pNum, new QNode)
57     else if IsLinkedTo(pNum) then
           // in the queue but not at the tail, wait
              for successor and transfer lock ownership
58         execute lines 35–39 of Exit(pNum)
       // recovery complete
59     continue to Enter(pNum, new QNode)
60 else if myNode.nextStep = inCS then
       // critical section re-entry
61     skip Enter and continue to critical section
62 else if myNode.nextStep ∈ Exit then
       // benign failure in exit protocol
63     complete Exit from line myNode.nextStep and continue to
       Enter(pNum, new QNode)
64 else if myNode.nextStep ∈ Enter then
       // benign failure in entry protocol
65     continue to Enter(pNum, myNode) at line myNode.nextStep
```

**Figure 6: Entry, exit and recovery protocol of $k$-Port Extended MCS Algorithm for process $p_i$, entering via port $pNum \in 0..k\text{-}1$.**

to distinguish between a scenario where their swap operation failed without changing *tail* (we call this an *invisible swap failure*) and another scenario where the failure occurred after the swap took place, but before the *prev* field was updated in line 27 (we call this a *visible swap failure*). When the swap failure is invisible, the process must retry its swap, whereas in case of a visible swap failure it

**Private variables:**

- *lTail*: **ref to** QNode
- *lnodes* **array** [0..*k*-1] **of ref to** QNode **init null**

**Function IsLinkedTo (int *pNum*)**

```
66 myNode := nodes[pNum]
   // save tail pointer
67 lTail := tail
   // inspect the latest QNode of every other process
68 for i := 0 to k-1, i ≠ pNum do
69     lnodes[i] := nodes[i]
       // determine prev pointer if QNode exists
70     if lnodes[i] ≠ null then
71         await lnodes[i].prev ≠ ⊥
   // decide whether my QNode is referenced by the
      tail pointer or by another QNode's prev pointer
72 return myNode ∈ {lTail} ⋃
   {lnodes[j].prev | 0 ≤ j < k ∧ lnodes[j] ≠ null}
```

**Figure 7: Helper function IsLinkedTo of the Extended MCS Algorithm.**

may need to find an appropriate predecessor and link to it. The difficulty is compounded further by the fact that multiple processes may be recovering simultaneously and that new passages may start on other ports.

Figure 9 depicts a configuration that may arise from multiple swap failures. In this scenario, the processes using $q_1$-$q_6$ underwent a swap failure: processes using $q_1$,$q_2$,$q_4$,$q_5$ underwent an invisible swap failure and those using $q_3$,$q_6$ underwent a visible swap failure. As a result, the queue is split into several *fragments* of QNode records, each ordered according to the reverse of the *prev* references (which equals the order determined by the eventual values of the *next* references): $(s_0, s_1, r_1)$, $(q_3, s_2, r_2)$, and $(q_6, s_3, r_3)$. By invoking **IsLinkedTo**, the processes using $q_1$,$q_2$,$q_4$,$q_5$ are able to determine that their records are not linked to (i.e., do not belong to any fragment) and so they must retry the swap. Each of the processes using $q_3$,$q_6$, on the other hand, will be able to determine in turn that its swap was performed and then, by invoking **RecoverSwap**, to compute which fragments exist, and to reconnect the queue structure. We differentiate between several types of fragments. The fragment containing records reachable from *tail* is the *tail fragment*. At most a single fragment may be *non-failed*, that is, containing only records associated with passages that already executed lines **26**–**27**. All other fragments (if any) are *middle fragments*.

We proceed to describe the **RecoverSwap** procedure, which receives the port number as its parameter (line **73**). It starts by capturing the *rLock* recoverable lock (line **74**), which simplifies recovery by ensuring mutual exclusion among processes attempting to reconnect the queue structure. Next, in the loop of lines **76**–**89**, for each port, the *prev* references are traced back starting from the port's record (if any) until a **null** or a FAIL reference value is encountered. The references starting from *tail* are also traced back, in iteration *pNum* of the loop. If the *prev* field is yet to be updated,

**Private variables:**

- *myNode*, *cur*, *prv*: **ref to** QNode
- *R*: binary relation, initially empty

**Procedure RecoverSwap (int *pNum*)**

```
73 myNode := nodes[pNum]
   // acquire auxiliary lock
74 rLock.Recover(), rLock.Enter()
75 R := ∅
   // inspect the tail pointer and the latest QNode
      of every other processes
76 for i := 0 to k-1 do
77     if i ≠ pNum then
           // inspect the next QNode
78         cur := nodes[i]
79     else
           // inspect the tail pointer
80         cur := tail
81         R := R ∪ (tail, TAIL)
       // trace prev pointers from QNode
82     while cur ≠ null do
83         await cur.prev ≠ ⊥
84         prv := cur.prev
85         if prv ∉ {null, FAIL} ∧ ¬prv.fin then
86             R := R ∪ (prv, cur)
87             cur := prv
88         else
89             break
90 derive maximal fragments from R:
   (n_{1,1} → ··· → n_{1,l_1}), ..., (n_{j,1} → ... , n_{j,l_j})
91 let f ∈ {1, ..., j} be myNode's fragment number
92 if f > 1 then
       // link my QNode to previous fragment
93     myNode.prev := n_{f-1,l_{f-1}}
94 else
       // my QNode is at the head of the queue
95     myNode.prev := null
96 rLock.Exit()
```

**Figure 8: Helper procedure RecoverSwap of the Extended MCS Algorithm.**

*p* awaits for this to occur at line **83**. Tracing stops if the *prev* field is **null**, equals FAIL, or is a reference to a record whose passage was completed (line **85**). All these predecessor-successor pairs are added to a set *R* (line **86**). All pairs added to *R* consist of two record references, with the single exception of the record referenced from the *tail* variable, which is designated as a special TAIL relation (line **81**). The reasons for this are twofold. First, this facilitates identifying the tail fragment. Second, whereas non-tail fragments always consist of at least two records, the tail fragment may consist of the single record referenced by *tail*. Based on *R*, process *p* may

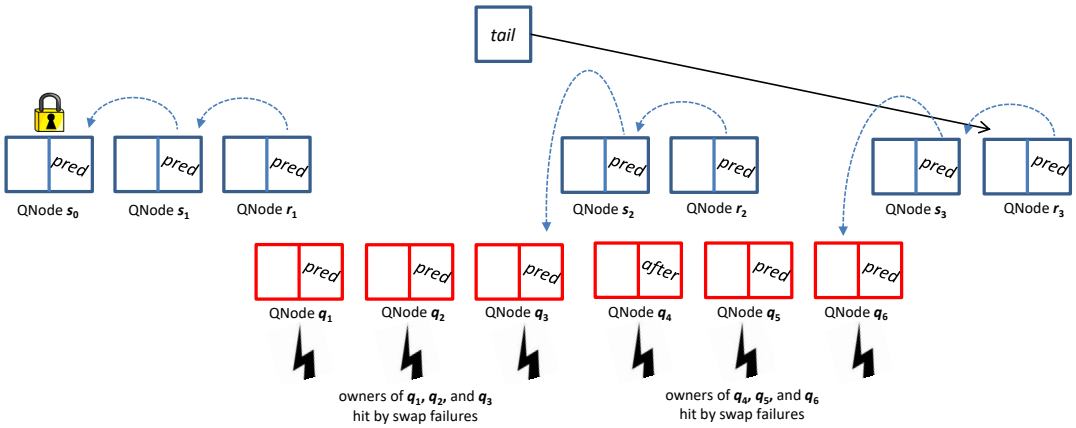


**Figure 9: Queue fragments after multiple swap failures. Dashed arrows represent *prev* references. QNode records $q_6, s_3, r_3$ constitute a tail-fragment, $q_3, s_2, r_2$ constitute a middle-fragment, and $s_0, s_1, r_1$ constitute a non-failed fragment.**

compute maximal queue-fragments and determine their types – tail-fragment, middle-fragment, or non-failed fragment (line 90). Fragments $1, \ldots, j$ are indexed such that if there is a non-failed fragment then it is indexed as 1, and the tail fragment is indexed as last (fragment $j$). The computation of maximal fragments is relatively straightforward. Due to lack of space, its pseudo-code is omitted. We note that new records may be associated with some of the ports throughout the execution of **RecoverSwap** but this does not compromise the correctness of the RME algorithm and, moreover, only $O(k)$ distinct records are accessed throughout a single execution of **RecoverSwap**.

If *myNode* is not part of the first fragment, then $p$ "glues" its fragment to the predecessor fragment, which is either a middle fragment or the non-failed fragment (if there is one). Process $p$ does this by updating its *prev* field to link to the last record of the predecessor fragment (line 92), and then releases *rLock* (line 96). Otherwise, *myNode is* part of the first fragment, implying that there is no non-failed fragment (which may happen if the first visible swap failure occurred when the queue was empty). In this case, $p$ updates its *prev* reference to **null** (line 95), effectively making its fragment the new single non-failed fragment, and then releases *rLock* at line 96. In both cases, after **RecoverSwap** returns, execution resumes from line 28 of the **Enter** procedure (line 51).

The correctness properties of the algorithm are captured in Theorem 4.1. The RMR complexity bound does not apply in the DSM model due to the busy-wait loops at line 71 and line 83.

Theorem 4.1. *The $k$-port extended MCS lock satisfies ME, SF, TE, BR, and CSR. The RMR complexity in the CC model is $O(1)$ for a passage that does not begin in cleanup, and $O(k)$ for any other passage.*

## 5 $O(\text{LOG}\, N/\text{LOG LOG}\, N)$ RMRS ALGORITHM

Let $k = \lceil \log N / \log \log N \rceil$. We obtain an $O(k)$ RMRs upper bound in the CC model by using the $k$-port extended MCS lock from Section 4 in an arbitration tree of degree $k$. The height of such a tree with $N$ leaves is $\Theta(k)$. Each node in the tree contains its own instance of the $k$-port lock. To enter the CS, a process $p$ must complete the recovery and entry protocol of every $k$-port lock instance on the path from its designated leaf node to the root node. When climbing to a node from its $i$'th child, $p$ uses the $i$'th port. Similarly, to exit the CS, a process must complete the exit protocol of every $k$-port lock instance on this path in the reverse order. All recovery actions are performed in the recovery protocol of the $k$-port algorithm, and for that reason the arbitration tree does not require a separate recovery protocol. The following theorem follows from Theorem 4.1, the fact that both the degree and the height of the tree used by the algorithm is $O(\log N / \log \log N)$, as well as the observation that each extended MCS lock instance is accessed correctly by at most $k$ processes at a time.

Theorem 5.1. *The arbitration tree algorithm satisfies ME, SF, TE, BR, and CSR. Its RMR complexity in the CC model is $O(\log N / \log \log N)$.*

## 6 RELATED WORK

Mutual exclusion (ME) is one of the oldest topics in distributed computing [2, 30]. Recent work on this topic 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 contributes to scalability [3, 9]. For the class of $N$-process algorithms that use reads, writes and comparison primitives (defined in [2]), 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 [21]. Attiya, Hendler, and Woelfel [4] later proved the matching lower bound, building on a series of earlier results [1, 8, 11, 15]. Sub-logarithmic RMR complexity is attainable on expectation using randomization [5, 12, 13, 19], and $O(1)$ RMR complexity is attainable deterministically if additional atomic primitives are available, such as Fetch-And-Store or Fetch-And-Add [3, 17, 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 [23, 31]. Lamport's Bakery algorithm [22] tolerates crash failures provided that any registers owned by the failed process are reset to zero [22]. 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 [6, 7]. 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 lock similarly relies on failure detection but does not require a dedicated recovery process [28]. Instead, a process that is waiting to acquire a lock may "usurp" the lock if it determines that the previous lock holder has crashed permanently.

Recoverable mutual exclusion (RME), which is the problem solved in this paper, was formalized recently by Golab and Ramaraju [16]. RME does not rely on a dedicated recovery process or on failure detection, but assumes that processes recover eventually from crashes outside of the non-critical section. 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 the Fetch-And-Store-And-Store (FASAS) instruction described in Section 3 [29].

The FASAS and DCAS instructions used by our algorithm in Section 3 are not supported by mainstream multiprocessors, and we are not aware of any RMR-efficient software implementations. Non-blocking DCAS implementations from commonly supported primitives have been studied previously in the conventional asynchronous model with crash failures (e.g., [18]).

## 7 CONCLUSION

In this paper, we presented RMR-efficient solutions to the recoverable mutual exclusion (RME) problem, and answered positively one of the questions posed recently by Golab and Ramaraju [16]: "Can this problem be solved using $o(\log N)$ RMRs per passage in the worst case using commonly supported primitives?" Our upper bound of $O(\log N/\log\log N)$ RMRs holds in the CC model, and establishes a complexity separation for RME between the class of algorithms that use reads, writes, and single-word comparison primitives versus those that additionally use unconditional single-word primitives such as Fetch-And-Store. This observation complements the known RMR complexity separation for conventional mutual exclusion (ME). Our results raise two follow-up questions: First, what is the tight bound for the worst-case RMR complexity of solving RME using commonly supported single-word primitives? Second, is this tight bound the same in the CC and DSM models, or is one model inherently more powerful than the other for solving RME?

## REFERENCES


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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] W. Golab. Deconstructing queue-based mutual exclusion Hewlett-Packard Labs Technical Report HPL-2012-100.

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

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

[18] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proc. of the 16th International Conference on Distributed Computing (DISC)*, pages 265–279, 2002.

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

[20] M. Herlihy and E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 289–300, 1993.

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

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

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

[24] L. Lamport. The PlusCal algorithm language. In *Proc. of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 36–60, 2009.

[25] L. Lamport. Specifying systems: the TLA+ language and tools for hardware and software engineers. Addison-Wesley, 2002.

[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.