Word-Size RMR Tradeoffs for Recoverable Mutual Exclusion

David Yu Cheng Chan, George Giakkoupis, Philipp Woelfel

Word-Size RMR Tradeoffs for Recoverable Mutual Exclusion

David Yu Cheng Chan
University of Calgary
Calgary, Canada
david.chan1@ucalgary.ca

George Giakkoupis
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
george.giakkoupis@inria.fr

Philipp Woelfel
University of Calgary
Calgary, Canada
woelfel@ucalgary.ca

ABSTRACT

We present tradeoffs between RMR complexity and memory word size for recoverable mutual exclusion (RME) algorithms using arbitrary synchronization primitives. Assuming that each memory location stores w bits, we show that n-process mutual exclusion has an RMR complexity of at least $\Omega\left(\min\left\{\log_w n, \log n/\log\log n\right\}\right)$ on the DSM and the CC model. For $w = (\log n)^{\Omega(1)}$, our lower bound asymptotically matches an upper bound by Katzan and Morrison [19], whose RME mutual exclusion algorithm employs w-bit fetch-and-add operations. Our lower bound is the first one that does not restrict the type of atomic operations that can be executed on a memory location.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**; • **Software and its engineering → Mutual exclusion**.

KEYWORDS

mutual exclusion, recoverable mutual exclusion, critical section, RME, concurrency, shared memory, fault tolerance

ACM Reference Format:
David Yu Cheng Chan, George Giakkoupis, and Philipp Woelfel. 2023. Word-Size RMR Tradeoffs for Recoverable Mutual Exclusion. In *ACM Symposium on Principles of Distributed Computing (PODC '23), June 19–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3583668.3594597

1 INTRODUCTION

Mutual exclusion [8] is arguably the most important tool for synchronization in concurrent algorithms. A mutual exclusion algorithm is used to serialize access to a piece of code, called the *critical section*. To get access to the critical section, a process needs to complete a protocol called *entry section*. Once it is finished with the critical section, it indicates so by executing the *exit section*. The *mutual exclusion* property guarantees that no two processes are in the critical section at the same time. In addition, it is generally required that these algorithms are *deadlock-free*, meaning that as long as all participating processes keep taking steps, not all of them can be stuck in the entry or exit section.

PODC '23, June 19–23, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0121-4/23/06…$15.00
https://doi.org/10.1145/3583668.3594597

The conventional model assumes a completely fault-free system, in which processes cannot crash. Motivated by recent advances in non-volatile memory technology, Golab and Ramaraju [12] defined the *recoverable mutual exclusion* (RME) problem. Here, process crashes are allowed, but shared memory is persisted. Any individual process can crash at any point in time, upon which its entire local state is reset, including any local variables, while the state of the shared memory is unaffected. After crashing, a process starts a recovery procedure and then resumes its mutual exclusion protocol. Since its introduction in 2015, the RME problem has been studied thoroughly [4, 5, 7, 10–13, 15–17, 19, 22].

The number of steps a process needs to execute in the entry section of a mutual exclusion algorithm cannot be bounded, because processes may have to busy-wait until the critical section becomes free. To analytically predict the performance of (conventional and recoverable) mutual exclusion algorithms, it is standard to analyze the maximum number of *remote memory references* (RMRs) that can be incurred during the entry and exit protocols. RMRs capture shared memory operations that require expensive communication through the processor-memory interconnect, or between processors. In the *distributed shared memory* (DSM) model, memory is partitioned into segments, each belonging to a different process. Whenever a process accesses shared memory outside its own segment, an RMR is incurred. In the *cache-coherent* (CC) model, each shared memory access incurs an RMR, except for a read operation by a process holding a valid cache copy of the accessed memory location.

In general, the RMR complexity of shared memory problems is very sensitive to the types of atomic operations supported by the system. Often, the minimal assumption is that atomic read and write operations are supported. More powerful operations fetch (and return) the value of a memory location, but at the same time modify the value stored at that location. For example, fetch-and-increment increments it, fetch-and-add adds an integer value (given as a parameter) to it, fetch-and-store writes a new value, and compare-and-swap conditionally writes a value, provided that the current value matches another given parameter. All these operations are supported by many of today's hardware architectures.

Much is known about the RMR complexity of conventional mutual exclusion for n processes. For systems that support only atomic read and write operations, there are mutual exclusion algorithms with RMR complexity $O(\log n)$ [23], which is optimal [2]. These bounds remain the same, even if the system supports compare-and-swap, which can be implemented from registers with constant RMR complexity [9]. By employing fetch-and-store or fetch-and-increment, the RMR complexity can be reduced to $O(1)$ [6, 20, 21].

The dependency of the RMR complexity of mutual exclusion on the word size w (measured in bits) of memory locations has not been studied explicitly, because existing algorithms and lower

bounds are independent of w, as long as $w = \Omega(\log n)$. It is common to assume $w = \Omega(\log n)$, so that a process can at least store its own ID using only a constant number of memory words.

For recoverable mutual exclusion on systems supporting atomic fetch-and-store operations, the best known algorithms have an RMR complexity of $\Theta(\log n/\log\log n)$ [10, 15]. (Here, the RMR complexity is the maximum number of RMRs a process can incur in any *passage*, which begins with the process's entry section or recover protocol, and ends when the process crashes or finishes its subsequent exit section.) Recently, Chan and Woelfel [5] showed that this is optimal for algorithms using only read, fetch-and-store, fetch-and-increment, and compare-and-swap operations. For the upper bounds, a word size w of $O(\log n)$ bits is sufficient, but the matching lower bound is true for any word size.

Better RMR complexity has been achieved for RME by using artificially defined operations that can atomically change two memory locations, and where the value written to one location depends on the value of the other location [10, 13]. Such operations do not exist in real systems, and have only been defined to study what it takes to improve the RMR complexity of RME. Throughout this paper we will make the standard assumption that each operation can only affect a single memory location.

Katzan and Morrison [19] observed that using fetch-and-add operations, which are supported by common hardware, the RMR complexity can be reduced to $O(1)$, assuming a word size of $n^{\Omega(1)}$ bits. More generally, their algorithm has an RMR complexity of $O(\log_w n)$ for w-bit fetch-and-add objects. One can argue that it is unrealistic to assume that the size of memory locations is polynomial in the number n of processors.

These results lead to two fundamental questions: First, are there atomic shared memory operations that allow us to solve RME with an RMR complexity of $o(\log n/\log\log n)$, under the common assumption that the word size is logarithmic, or at least poly-logarithmic in n? Second, is it possible to improve upon the trade-off between word-size and RMR complexity that the algorithm by Katzan and Morrison exhibits? In this paper, we provide negative answers to both questions:

Theorem 1. *Any deadlock-free n-process RME algorithm tolerating individual process crashes, where all shared memory locations store values from a domain of size 2^w (but support arbitrary atomic operations), has RMR complexity $\Omega\left(\min\left(\log_w n, \frac{\log n}{\log\log n}\right)\right)$ in the CC and the DSM model.*

The lower bound is asymptotically tight for $w \geq (\log n)^{\varepsilon}$, for any $\varepsilon > 0$, as it matches the upper bound of [19]. It shows that it is not possible to improve the algorithms with $O(\log n/\log\log n)$ RMR complexity [10, 15] on systems whose memory locations can store only poly-logarithmic many bits. As far as we know, all known RME algorithms implicitly assume $w = \Omega(\log n)$, and RMR upper bounds for smaller word sizes have not been studied.

This is the first RMR lower bound that does not rely on restricting the types of shared memory operations an algorithm can use (but instead restricts the word size). It is also the first trade-off between word size and RMR complexity. In fact, as far as we know, no such lower bounds or trade-offs have previously been observed for *any* shared memory problem.

1.1 Technical Contribution

Our proof builds on ideas of Chan and Woelfel [5], which in turn uses the framework of Anderson and Kim [1], but requires some deeper combinatorial insights and has to deal with several technical difficulties in a novel way. In this section we sketch the core ideas of [1, 5], and explain how our new approach is different.

Anderson and Kim's lower bound of $\Omega(\log n/\log\log n)$ applies to conventional (non-recoverable) *one-shot* mutual exclusion, i.e., each process tries to enter and exit the critical section once, and then stops taking steps. This lower bound assumes that each process can only perform read and write operations.

The idea is to construct a sequence of executions, $E_0, E_1, \ldots,$ where E_i comprises i *rounds*. With each execution E_i we associate a set of *active* processes that perform exactly one RMR in each of the rounds $1, \ldots, i$ and at the end of E_i are poised to perform another RMR. Moreover, in each round some processes may be forced to run to completion upon which they are *finished*, and others are removed from the execution (i.e., we determine a new execution, where the removed processes do not take any steps). The remaining processes continue to be active. The main invariant is that any active process in round i can be removed from E_i without affecting any of the non-removed processes participating in E_i, and without making any non-removed process "visible" on a register. For example, if in round 1 processes p_1, p_2, and p_3 write to a register R in this order, then p_3 cannot be active (and thus must be finished) at the end of the round, as removing p_3 would make p_2 visible on R. On the other hand, p_1 and p_2 can both be active, as neither has seen (or discovered) the other. If in some round a process q_1 writes a register and then immediately after that a process q_2 reads it, then it is not possible that q_1 and q_2 both remain active, because q_2 has discovered q_1, and so removing q_1 from the execution would affect q_2.

The lower bound then proceeds by constructing $E_0, E_1, \ldots, E_\ell$ iteratively for some $\ell \in \Omega(\log n/\log\log n)$, such that E_ℓ has still at least 2 active processes (which must have performed ℓ RMRs).

To construct E_i from E_{i-1}, one distinguishes between low and high contention rounds. A low contention round occurs if at the beginning of E_i the majority of processes are poised to access a "low contention" register, which means fewer than k processes are about to access it, where k is a poly-logarithmic threshold. Otherwise, the round is called a high contention round.

We will focus on the high contention round, which is the one where most of our new ideas are applied. For the purpose of simplicity, assume that on each register there are exactly k processes poised to perform their next shared memory step, which is a write that will incur an RMR. Then we can schedule the writes in arbitrary order and after that let the last writer, p, run to completion, removing any active processes that p may otherwise discover. (Recall that by the invariants of this construction, we can always remove arbitrary active processes without affecting any other processes.) It is easy to see that this preserves the desired properties, because p's write overwrites and thus"hides" all earlier writes to that register.

Moreover, as p can perform only $o(\log n)$ RMRs (which is larger than the lower bound we are trying to prove), not too many processes need to be removed. As a result, at least a $1/(\log n)^{O(1)}$ fraction of the active processes at the end of round E_{i-1} are still

active in round E_i, which is sufficient to obtain the desired lower bound.

The RME lower bound of Chan and Woelfel [5] allows processes to perform, for example, fetch-and-store (FAS) operations, but it does not apply to conventional, non-recoverable mutual exclusion. The reason is exactly the high contention round: If many processes perform a FAS on the same register, then each of them will discover the previous one, and not more than one process can remain active. (In fact, standard constant RMR queue lock algorithms are based on that idea, e.g., [6, 20, 21].) In the recoverable model, however, carefully chosen process crashes can rescue the lower bound idea. Assume that at least $k+1 = (\log n)^{\Theta(1)}$ processes are poised to perform FAS operations on the same register. Take an arbitrary group of exactly $k+1$ of those processes, $\alpha, \beta_1, \ldots, \beta_k$. For a process β_j from that group consider the following execution: First, let β_j perform its FAS. Then let α perform its FAS (in which it overwrites β_j's FAS but also learns about β_j), let α crash (so that it forgets about β_j), and finally let α recover and run to completion, which may require removing from the execution $o(\log n)$ other processes that would otherwise be discovered. Unless β_j is one of the processes we have to remove, β_j's FAS is now hidden by α's FAS, much in the same way as before earlier writes on a register were hidden by the last write. (The reason why we chose multiple β-processes for the group is that some of them have to be removed because they get discovered by α-processes, possibly from other groups. The size of the group guarantees that sufficiently many groups have at least one β-process that does not get discovered, and can remain active.)

This technique fails again for certain powerful operations, such as fetch-and-add (FAA) with large word-size. This is exactly what Katzan and Morrison [19] exploit: In their algorithm each process p performs FAA(2^p), effectively setting the p-th bit of the register and learning the set of all processes that performed the operation before p. This makes it impossible to hide any processes.

Intuitively, however, if the memory word size, w, is small enough, then not all processes can leave information in a register. This is what our lower bound exploits. The difficulty is to make no assumption on what kind of information processes can leave behind.

To that end we prove a complex combinatorial lemma (the Process-Hiding Lemma). For simplicity consider a word-size $w = \Theta(\log n)$. Consider a group of $k = (\log n)^{\Theta(1)}$ processes, such that at the beginning of round i all processes in the group are poised to access the same register, and each such access will incur an RMR. As before, we consider a high contention round i when constructing execution E_i from E_{i-1}. A simplified version of the lemma states that we can find two (not necessarily distinct) sets, A and B, of processes in that group, as well as a special process $z \in B$, such that if either exactly each process in A takes a single step (in some predefined order) or exactly each process in B takes a single step, the register value resulting from those steps is the same. In addition, if we let all processes in $A \cup B \setminus \{z\}$ crash, recover, and run to completion, then z will not be discovered. This way, we obtain two possible executions for round i: In the first one, all processes in B (including z) take a single (RMR) step, then all processes in $A \cup B \setminus \{z\}$ crash, recover and run to completion. In the second one, all processes in A take a single step instead of those in B, and the rest is the same as before. Because the register value is the same at the end of each of the two executions, no process except for z can distinguish the two. Thus, if we want to keep z an active process, we use the first execution. If at a later point during our construction (of additional rounds) we realize that we need to remove z from the execution (because otherwise it would get discovered), we can switch to the second one, and no other process is affected by this.

In fact, the statement (and proof) of the Process-Hiding Lemma is more complex than described above, as it has to handle multiple groups simultaneously, each containing k processes that are poised to perform their next step on the same register. Moreover, even though the overall outline of the entire lower bound proof is similar to [5], the deeper details are quite different and technically much more difficult.

1.2 Other Related Work

Researchers have considered several additional desirable properties of RME algorithms, such as starvation-freedom, first-come-first-serve, or abortability. Below, we focus on the RMR complexity of solutions to the basic RME problem, ignoring any extended properties.

The first RME algorithm by Golab and Ramaraju [12] has an RMR complexity of $O(n)$. This was then improved by Jayanti and Joshi to $O(\log n)$ [16], and finally to $O(\log n/\log\log n)$ by Golab and Hendler [10] for the CC model, and Jayanti, Jayanti, and Joshi [15] for the DSM model. All these algorithms use fetch-and-store, or compare-and-swap, or both. Using fetch-and-increment, Chan and Woelfel obtained constant *amortized* passage complexity [4]. Dhoked and Mittal [7] presented a *crash-adaptive* algorithm, which has improved performance if the total number of process crashes, f, is limited: It has an RMR complexity of $O(\min\{\sqrt{f}, \log n/\log\log n\})$. Katzan and Morrison's algorithm [19] adapts to the number, k, of processes participating concurrently; assuming w-bit fetch-and-add, where $w = \Theta(\log n)$, it has an RMR complexity of $O(\min\{k, \log n/\log\log n\})$.

For the weaker system-wide crash model, where all processes must crash simultaneously, Golab and Hendler [11] solved the RME algorithm with constant RMR complexity. However, the authors assume non-standard system support, guaranteeing that an epoch counter is incremented with each system crash. A recent algorithm by Jayanti, Jayanti, and Joshi [14] also achieves constant RMR complexity without making such assumptions.

2 PRELIMINARIES

We assume the standard asynchronous shared memory model. It comprises n processes with unique IDs and any number of *base objects* (shared memory locations). Processes communicate by performing arbitrary atomic operations (called *steps*) on the base objects. Each base object stores w bits.

We assume (w.l.o.g.) that processes execute *one-time mutual exclusion*: First, a process executes an *entry protocol*. Once completed, the process is in the *critical section*, where it may perform other operations on memory locations that are not used by the mutual exclusion algorithm. Then it executes an *exit protocol*. After that, it takes no more steps. A process that has not yet started the entry protocol or has finished the exit protocol, is in the *remainder section*.

We assume that at any point, when a process is about to perform a step of its entry or exit protocol, it may instead be forced (by the system) to perform a *crash step*. We say the process *crashes* when it performs that step. When that happens, all its local variables are reset to their initial values, and the process immediately begins executing a *recover protocol*. After the recover protocol, it essentially resumes its mutual exclusion protocol; what exactly this means is determined by the RME properties defined below. Generally, RME algorithms also tolerate crash steps in the critical section (see the critical-section reentry property described below), but our lower bound applies regardless.

A process begins a *passage*, when it performs the first shared memory step of its entry or recover protocol, and ends with the next crash step, or when it enters the remainder section (after finishing the exit protocol). A *super-passage* of a process begins with the start of its entry protocol, and ends when the process finishes its subsequent exit protocol. Thus, a super-passage may comprise multiple passages, separated by crash steps, and in the absence of crashes, passages and super-passages are the same.

The algorithm must satisfy *mutual exclusion*, which means that no two processes can be in the critical section at the same time. It must also satisfy *deadlock-freedom*, which requires that some process must eventually enter the remainder section, provided that not all processes are in the remainder section, all processes that are not in the remainder section keep taking steps, and the number of crash steps is finite.

For our lower bound, it is sufficient if each process can crash at most once. The definition of recoverable mutual exclusion [12] also requires another property, called *critical section re-entry*, but our lower bound is independent of that.

To analytically predict the performance of (recoverable and conventional) mutual exclusion algorithms, two models have been studied. In the *cache-coherent* (CC) model, each process is equipped with a cache, and all processes are connected via a bus to the shared memory. Whenever a process performs a read operation it stores a copy of the read value in its cache. Any non-read operation (by any process) of that memory location invalidates the cache copy. We say a process's operation incurs a *remote memory reference* (RMR), if it is a non-read operation, or it is a read operation for a memory location of which the process has no valid cache copy. In the *distributed shared memory* (DSM) model, the shared memory is partitioned into segments, one for each process. An operation by a process p on a shared memory location incurs an RMR if and only if that memory location is not in p's memory segment. The RMR complexity of a mutual exclusion algorithm is the maximum number of RMRs a process may incur in a passage.

3 THE RME LOWER BOUND PROOF

Consider an arbitrary algorithm that solves the RME problem with $o(w)$ RMR complexity. The goal of this section is to show that this RME algorithm has $\Omega(\min(\log_w n, \log n/\log\log n))$ RMR complexity. Since $\log_w n = \log n/\log\log n$ for $w = \log n$, it suffices to assume that $w \geq \log n$ and then simply show that this RME algorithm has $\Omega(\log_w n)$ RMR complexity. (Clearly, smaller values of w can only increase the RMR complexity.)

Towards that end, we consider a system that uses this RME algorithm in the following manner (all these assumptions can be made w.l.o.g.):

(A1) The number of processes is sufficiently large such that every passage of every execution incurs no more than w RMRs. (Because of asymptotic notation in the assumptions above, this may not be true for small n.)
(A2) In the critical section, each process performs exactly one step, which incurs an RMR.
(A3) Each process crashes at most once.

Then it suffices to construct an execution of this system in which some process never crashes and never enters the critical section, yet incurs $\Omega(\log_w n)$ RMRs.

Let $\mathcal{P} = \{1, \ldots, n\}$ be the set of processes in this system, $\mathcal{R}'$ the set of shared objects in the system, and $\mathcal{R} \subseteq \mathcal{R}'$ the set of shared objects used by the RME algorithm. A *configuration* C consists of a state for each process $p \in \mathcal{P}$ and each shared object $R \in \mathcal{R}'$. Let C_0 denote the initial configuration, i.e., the configuration that consists of the initial state of each process and object in this system. A *schedule* is a sequence over $\{p, \hat{p} : p \in \mathcal{P}\}$, where p denotes a non-crash step by process p, and $\hat{p}$ denotes a crash-step by p.

An *execution* is a sequence of *events*, where each event corresponds to a step by some process and contains the following information: the process that is executing the step, the shared memory operation that process is executing, the shared object on which the shared memory operation is executed, and whether the shared memory operation incurs an RMR.

3.1 Proof Strategy

Similar to [1, 5], our proof is constructed in rounds. In each round, every process that is still trying to enter the critical section, takes a number of steps, and incurs at least one RMR. The construction method differs depending on the amount of contention, i.e., whether most processes are poised to access shared objects that many other processes are poised to access. Intuitively, the low contention scenario is simply about removing processes such that no pair of processes can communicate via the same shared object, so its proof is essentially unchanged from earlier results. On the other hand, the high contention scenario requires various techniques to minimize the amount of useful information communicated by processes even when they are accessing the same object. Consequently, our proof significantly differs in the high contention scenario, because unlike these earlier results, our proof needs to deal with arbitrary shared memory operations.

The proof is based on a simple observation: if multiple processes are 'actively' attempting to enter the critical section, then they cannot safely enter the critical section before discovering one another, lest they violate mutual exclusion. Thus, throughout the proof several closely related schedules are constructed in a manner that maximizes both the number of these *active* processes and the number of RMRs they incur without discovering one another.

More formally, let $\sigma_{round}[0..\infty][0..2^n - 1]$ be an initially empty table of schedules with an unbounded number of rows and 2^n columns. Roughly speaking, for every non-negative integer i, the i-th row of the table will contain only schedules in which the active processes have incurred at least i RMRs. For every integer $s \in$

$\{0, 1, \ldots, 2^n - 1\}$, the s-th column is associated with the unique set $S \subseteq \mathcal{P}$ of processes such that $s = \sum_{p \in S} 2^{p-1}$. Then the s-th column will contain only schedules in which only the processes in S can begin super-passages.

Filling the first row of the table is simple: in the empty schedule, every active process has incurred 0 RMRs, and the set of processes that have begun super-passages is $\emptyset$, a subset of every possible set of processes. Thus, every cell of $\sigma_{round}[0][0..2^n - 1]$ is set to contain the empty schedule.

The proof then proceeds in rounds, where in each round $i \geq 1$, some cells of the i-th row are filled with schedules derived by appending more steps to the schedules in the $(i-1)$-th row. Since the only desired schedules are those in which the active processes do not discover one another, many of the cells in each row will be left with the value $\perp$, indicating that no schedule matching the required criteria was found. Thus, as the proof iterates through the rows of the table, the number of cells in each row that are filled with schedules decreases.

As such, the goal of each round is to limit this decrease, such that $\Omega(\log_w n)$ rounds complete before the number of schedules becomes too small to continue. After this point, every schedule in the final round would have active processes that incur $\Omega(\log_w n)$ RMRs without entering the critical section (or crashing).

To facilitate that, a number of invariants are maintained on every row of schedules constructed. Roughly speaking, these invariants are:

(1) The s-th column contains only schedules in which only the processes in the associated set S can begin super-passages.
(2) There is exactly one *maximal* schedule which has the maximal number of active processes, and all other schedules are 'sub'-schedules that correspond to every proper subset of the active processes in the maximal schedule. This invariant ensures that if the maximal schedule cannot be extended without allowing some active processes to discover one another, then a sub-schedule can be extended and made into the new maximal schedule for the next round.
(3) The state of every process is the same in every schedule it is part of. This invariant ensures that the active processes have not discovered one another, since they have the same state in a schedule where there are no other active processes.
(4) The set of processes that have completed their super-passage is the same over the entire row of schedules. This invariant avoids potentially complicated scenarios where a process that completes its super-passage in one schedule is still active in another.
(5) For each shared object, its value in each schedule depends only on whether the schedule contains the process that last accessed it in the maximal schedule. This invariant ensures that shared object values are sufficiently similar across different schedules that it becomes difficult for the active processes to later distinguish between different schedules.
(6) In every schedule, each process crashes at most once, and every process that has not completed its super-passage has never crashed. The invariant limits the number of crashes so that the resulting lower bound holds even when each process crashes at most once.
(7) In every schedule, every process that has not completed its super-passage has not yet entered the critical section. The invariant makes the proof significantly simpler, since it prevents interactions between the active processes and the inactive processes that have already entered the critical section but not yet completed their super-passage.
(8) In the DSM model, the shared objects that are owned by active processes have not been accessed by any other active process. This invariant also simplifies the proof, since it prevents non-RMR-incurring steps from allowing an active process to discover another active process, and thus allows the proof to focus on the RMR-incurring steps.
(9) In the CC model, for each process p, the set of shared objects that p has valid cache copies of is identical over all schedules that contain p. This invariant ensures that in the CC model, for each process p, the number of RMRs incurred by p is the same in every schedule it is part of.
(10) In the i-th row, every active process in every schedule has incurred at least i RMRs.

It is easy to see that these invariants hold for row 0. Furthermore, for every non-negative integer i, let n_i be the number of active processes in the maximal schedule of row i. Then the second invariant asserts that row i has 2^{n_i} schedules. Moreover, to show that $\Omega(\log_w n)$ rounds can be completed, it suffices to show that for every integer $i \geq 1$, $n_i > n_{i-1}/w^{O(1)}$.

Each round of the proof is divided into two phases: a setup phase in which non-RMR-incurring steps are appended to the schedules until every active process in every schedule is poised to incur an RMR, and a contention phase, in which RMR-incurring steps are appended in specific orders that limit the fraction of active processes discovered.

In the setup phase, multiple non-RMR-incurring step(s) are appended for each active process until they are poised to incur an RMR. By the above invariants, the non-RMR-incurring steps appended for each process are the same in every schedule that contains the process. This is because each process begins with the same state in every schedule that contains the process, and then:

- In the DSM model, its non-RMR-incurring steps only access its own shared objects, which have never been accessed by any other active process, and thus these steps intuitively provide no new information that would cause the process to change its next steps.
- In the CC model, its non-RMR-incurring steps would be reads on shared objects that it has valid cache copies of in every schedule that contains it. Then, since the process already has valid cache copies of these shared objects, they intuitively provide no new information that would cause the process to change its next steps.

In the contention phase, the construction method differs depending on whether at least half of all active processes (in the maximal schedule) are poised to access a shared object that at least k processes are poised to access, where $k = w^d$ for some sufficiently large constant d.

In a low contention scenario, less than half of all active processes (in the maximal schedule) are poised to access a shared object that at least k processes are poised to access. Thus on average, each

shared object has relatively few processes poised to access it. In this case, a graph is constructed with nodes representing the active processes, and edges that intuitively indicate processes that could discover one another: either because they are poised to access the same shared object, or they are poised to access a shared object that is owned or was previously accessed by another active process. Since the contention is relatively low, the resulting graph is relatively sparse, and thus contains a relatively large independent set. Then any schedule that contains any process outside this independent set is discarded, so that the remaining schedules only contain active processes that would not discover one another with their next step. These remaining schedules then have a single step appended for each active process, and then are used to fill the next row of $\sigma_{round}[0..\infty][0..2^n-1]$. It is straightforward to show that the above invariants still hold for this new row of schedules. Furthermore, due to the relative largeness of the independent set, it is also straightforward to show that $n_i > n_{i-1}/w^{O(1)}$ for every row $i \geq 1$ constructed in a low contention scenario.

In a high contention scenario, at least half of all active processes (in the maximal schedule) are poised to access a shared object that at least k processes are poised to access. Thus, on average, each shared object has relatively many processes poised to access it. In this scenario, it is often inevitable that some active processes are discovered by the others, and these active processes must then be inactivated by allowing them to enter the critical section and then complete their super-passage. To further complicate matters, each such process could discover $O(w)$ other active processes before completing its super-passage, and these discovered processes must then be removed (schedules that contain such processes are discarded).

What makes matters even more complex is that we consider shared objects that support arbitrary operations. Such operations could potentially allow every active process in a single operation to discover all other active processes that have previously accessed the shared object! As such, this is the point where our proof significantly diverges from the earlier results of [1, 5].

First, we divide the active processes into groups of $\Theta(k)$ processes, such that within each group, all processes are poised to access the same shared object (we remove any active processes that cannot be placed into such groups, the number of which is at most a constant fraction of the active processes). We also remove any active processes that either own a shared object that some group is poised to access, or were the last to access a shared object that some group is poised to access. Since this is the high contention scenario, the number of shared objects that the groups are poised to access, and hence the number of active processes that need to be removed, is just a fraction of the total. Then there are two cases: either the majority of groups contain at least one process that is poised to perform a read operation, or not.

The first case is simple: since read operations do not change the state of a shared object, it is impossible to discover processes from the read operations they have performed. Thus we remove any schedule that contains a process that is poised to perform a non-read operation. The remaining schedules then have a single step appended for each remaining active process, and then are used to fill the next row of $\sigma_{round}[0..\infty][0..2^n-1]$. It is straightforward to show that the above invariants still hold for this new row of schedules. Furthermore, since $k = w^d$ and the majority of groups contain at least one process that is poised to perform a read operation, it is also straightforward to show that $n_i > n_{i-1}/w^{O(1)}$ for every row $i \geq 1$ constructed in a read high contention scenario.

In the second case, we first remove every group that contains a process that is poised to perform a read operation, i.e., we remove any schedule that contains a process from these groups. So all remaining groups contain only active processes that are poised to perform non-read operations. Then let $X_1, X_2, \ldots, X_m$ be these remaining groups of active processes, and for each group X_j, let R_j be the shared object that group X_j is poised to access. We then consider certain subsets A_j of X_j, and associate with each an execution in which each process from A_j takes exactly one step (in a specific fixed order). Thus, for each $A_j \subseteq X_j$ we obtain a unique state of R_j at the end of the associated execution. Let y_j be the state that results from the largest number of subsets of X_j (with ties broken arbitrarily).

At this point, we make another important observation: if there are multiple methods to change a shared object to some state y, any operation that finds the state to be y cannot determine which method was used.

With this in mind, our goal is the following:

- First, identify for every group sets A_j and V_j, where $A_j \subseteq V_j \subseteq X_j$, such that when each process in A_j takes exactly one step (in some predetermined order) the state of R_j is changed to y_j.
  The processes in these sets V_j are called the alpha processes, and every schedule that does not contain all of the alpha processes is being discarded. The alpha processes will be crashed to make them forget any information they learned, and then they will be allowed to run until they complete their super-passages. Any other active processes that they discover along the way are removed (schedules that contain such processes are discarded). As a result, these alpha processes are the processes that can potentially be discovered by other processes in future rounds.
- Then, for at least a constant fraction of the groups, identify sets $B_j \subseteq V_j$ as well as a process $z_j \in X_j \setminus V_j$, such that when the processes in B_j take steps in some order, the state of R_j is also changed to y_j. These processes in $B_j \cup \{z_j\}$ are called the beta processes.
  Note that A_j and B_j are not necessarily disjoint, or distinct.

This way, we now obtain two possible (sub-)executions: In one, first all processes in A_j take a single step, in the other first all processes in $B_j \cup \{z_j\}$ take a single step. In both executions, after the steps by the processes in A_j or B_j, all processes in V_j crash and then run to completion. The resulting two configurations are indistinguishable to all remaining active processes other than z_j. In other words, the undiscovered active process z_j can take an RMR-incurring step that is "hidden" by the steps of the alpha processes in its group.

Achieving this goal (of finding suitable sets A_j, B_j, and V_j as well as processes z_j), is not trivial. Each alpha process can incur $o(w)$ RMRs in its super-passage, allowing it to discover $o(w)$ active processes. Furthermore, the alpha processes can communicate with one another to change their behaviors, so the set of processes that

are discovered by the alpha processes can differ wildly even if only a single alpha process is added or removed, making it difficult to ensure that any chosen process z_j is not discovered.

A new combinatorial lemma, the *Process-Hiding Lemma*, is a key technical contribution of this proof. Essentially, the Process-Hiding Lemma asserts that as long as there are at least $216w^3$ processes within each group, there must exist sets of alpha and beta processes that satisfy the above requirements. (Recall that each group contains $\Theta(k)$ processes, where $k = w^d$, so the lemma is applicable with a sufficiently large constant d.) Then, since $k = w^d$ and a constant fraction of the groups of $\Theta(k)$ processes yield an undiscovered beta process for the new maximal schedule, we can also prove that $n_i > n_{i-1}/w^{O(1)}$ for every row $i \geq 1$ constructed in a high contention scenario.

Thus, regardless of whether each round $i \geq 1$ has a low contention phase or a high contention phase, $n_i > n_{i-1}/w^{O(1)}$. By the second invariant, the number of schedules in each row i is 2^{n_i}. So $\Omega(\log_w n)$ rounds complete before the number of schedules becomes too small to continue. After that the schedules in the final round have active processes that incur $\Omega(\log_w n)$ RMRs without entering the critical section (or crashing). Consequently, the algorithm has $\Omega(\log_w n)$ RMR complexity.

We state and prove the Process-Hiding Lemma in Section 3.2. In Section 3.3, we formally state the invariants that the array of schedules satisfy. Then in Section 3.4, we show how to derive the RMR lower bound from those invariants. The proof of the invariants is deferred to the appendix.

3.2 The Process-Hiding Lemma

In this subsection, we prove the following key statement.

Lemma 2 (Process-Hiding Lemma). *Let X and Y be sets, and let (X_i), $i \in \{1, \ldots, m\}$, be a partition of X. Let $\ell \geq 0$ be an integer and $\delta \geq 1$ a real number, and suppose that $|Y| \leq 2^\ell$ and $|X_i| \geq 108\delta\ell^2$ for all $i \in \{1, \ldots, m\}$. For each $y \in Y$, let $f_y \colon 2^X \to Y$ be a function, and let $y_0 \in Y$.*

There exist sequences (y_i), (A_i), and (V_i), $i \in \{1, \ldots, m\}$, such that for each $i \in \{1, \ldots, m\}$,

$$y_i \in Y \text{ and } \emptyset \subseteq A_i \subseteq V_i \subseteq X_i,$$

and

- *$f_{y_{i-1}}(A_i) = y_i$ for all $i \in \{1, \ldots, m\}$;*
- *for each set $D \subseteq X$ of size $|D| \leq \delta \cdot |\bigcup_{1 \leq i \leq m} V_i|$, there exists a subset of indices $I_D \subseteq \{1, \ldots, m\}$ of size $|I_D| \geq m/2$, and sequences (z_i) and (B_i), $i \in I_D$, such that for each $i \in I_D$,*

$$z_i \in X_i \setminus (V_i \cup D) \text{ and } B_i \subseteq V_i,$$

and $f_{y_{i-1}}(B_i \cup \{z_i\}) = y_i$.

To understand the power of the Process-Hiding Lemma, first consider the case $m = 1$. Suppose that, at the beginning of a high contention round, X_1 is the set of processes poised to perform a step on the same memory location R, and y_0 is the value of that memory location. Then for a set $A_1 \subseteq X_1$, $f_{y_0}(A_1)$ is the value of R, if all processes in A_1 take one step (in some predefined order). Given sets $A_1 \subseteq V_1 \subseteq X_1$, D is the set of processes that are discovered by the processes in V_1, if all of them take a crash step and then run to completion. We use δ to indicate the number of shared memory locations a single process can access while running to completion, and thus the number of processes it can discover. The lemma states that there exists a set $B_1 \subseteq V_1$ and a process $z_1 \in X_1 \setminus (V_1 \cup D)$ (which is not being discovered by the processes in V_1), such that if all processes in $B_1 \cup \{z_1\}$ take a step (instead of the processes in A_1), the resulting value of register R is also y_1. Hence, no matter if the processes in A_1 take a step, or the processes in $B_1 \cup \{z_1\}$, and then all processes in V_1 crash and run to completion, the resulting value of R is the same. Thus, the two configurations are indistinguishable to all remaining processes other than z_1.

We use $m > 1$ to deal with the case where very many processes are poised to perform an operation on register R. In that case, we split the processes into groups X_i, $i = 1, \ldots, m$, each of which is of size roughly w^d for some constant d. For each group X_i we construct sets of processes $V_i \subseteq X_i$, $A_i \subseteq V_i$, $B_i \subseteq V_i$, and $z_i \in X_i \setminus V_i$ as before. Now, D is the set of processes discovered, if all processes in $V_1 \cup \cdots \cup V_m$ crash and then run to completion. Moreover, y_i is the value the register has after all processes in $A_1, A_2, \ldots, A_i$ have taken steps. The lemma states that for a constant fraction of indices $j \in \{1, \ldots, m\}$ the same value y_j is obtained, if the processes in A_j are replaced with the processes in $B_j \cup \{z_j\}$.

Note that all variable names used are local to this section. We will use the following operations.

Definition 3. *For any set X, and for any $A \subseteq X$ and $B \subseteq 2^X$, we define*

$$\sigma_A(B) = \{S \colon S \in B,\ A \subseteq S\}, \quad \pi_A(B) = \{S \setminus A \colon S \in \sigma_A(B)\}.$$

If X is a singleton set $X = \{x\}$, we will write σ_x and π_x to denote σ_X and π_X, respectively.

To visualize these definition note that if for any two distinct sets $S, S' \in \sigma_A(B)$ their intersection is $S \cap S' = A$, then $\sigma_A(B)$ is a sunflower system, with core A and petals the elements of $\pi_A(B)$ (see, e.g., [18] for an introduction to sunflower systems). Our analysis, however, does not involve sunflowers or results about them.

Recall that in a k-partite hypergraph $H = (X_1, \ldots, X_k, E)$, the vertices are partitioned into k disjoint sets $X_1, \ldots, X_k$, and each hyperedge $e \in E$ contains precisely one vertex from each set. The next basic lemma states that if X_1 has size at most $s \cdot (1 + \epsilon)$, then: either there are two (not necessarily distinct) vertices $z_1, z_2 \in X_1$ such that the set $\pi_{z_1}(E) \cup \pi_{z_2}(E)$ has size at least $|E|/s$; or there are at least $s \cdot (1 + \epsilon)(1 - 2\epsilon)$ distinct vertices $z \in X_1$ such that their sets $\pi_z(E)$ intersect.

Lemma 4. *Let $H = (X_1, \ldots, X_k, E)$ be a k-partite hypergraph such that $|X_1| \leq s \cdot (1 + \epsilon)$, where s is a positive integer and $0 \leq \epsilon < 1/2$. There exists a set $Z \subseteq X_1$ such that*

(a) $|Z| \leq 2$ and $|\bigcup_{z \in Z} \pi_z(E)| \geq |E|/s$; or
(b) $|Z| \geq s \cdot (1 + \epsilon)(1 - 2\epsilon)$ and $\bigcap_{z \in Z} \pi_z(E) \neq \emptyset$.

Proof. We assume that (a) is not true for any $Z \subseteq X_1$, and we will show that (b) holds for some $Z \subseteq X_1$. Let $\ell = |X_1|$, and suppose that $X_1 = \{x_1, \ldots, x_\ell\}$. For $1 \leq i \leq \ell$, let $p_i = \pi_{x_i}(E)$, and suppose that $|p_1| \geq |p_2| \geq \cdots \geq |p_\ell|$. We can bound $|p_1|$ from below and above as follows. We observe that $\sum_{1 \leq i \leq \ell} |p_i| = |E|$. And since $|p_1| = \max_{1 \leq i \leq \ell} |p_i| \geq (1/\ell) \sum_{1 \leq i \leq \ell} |p_i|$, we have

$$|p_1| \geq |E|/\ell \geq |E|/[s(1 + \epsilon)],$$

since $\ell = |X_1| \le s(1+\epsilon)$. Also

$$|p_1| < |E|/s,$$

because of our assumption that (a) is not true for any $Z \subseteq X_1$. Let

$$\lambda = \max\{i \colon |p_1| + |p_i| \ge |E|/s\}.$$

We now compute a lower bound on $\sum_{1\le i\le\lambda} \frac{|p_1\cap p_i|}{|p_1|}$. We have

$$\begin{aligned}
\sum_{1\le i\le\lambda} \frac{|p_1\cap p_i|}{|p_1|} &= \sum_{1\le i\le\lambda} \frac{|p_1|+|p_i|-|p_1\cup p_i|}{|p_1|} \\
&= \sum_{1\le i\le\lambda} \left(1 - \frac{|p_1\cup p_i|-|p_i|}{|p_1|}\right) \\
&\ge \sum_{1\le i\le\lambda} \left(1 - \frac{|E|/s-|p_i|}{|E|/\lceil s(1+\epsilon)\rceil}\right) \\
&= \sum_{1\le i\le\lambda} \frac{|E|-|E|(1+\epsilon)+|p_i|s(1+\epsilon)}{|E|} \\
&= -\lambda\epsilon + \frac{s(1+\epsilon)}{|E|}\cdot\sum_{1\le i\le\lambda}|p_i|,
\end{aligned}$$

where for the inequality in the third line we used that $|p_1\cup p_i| < |E|/s$, because of our assumption that (a) does not hold for any $Z\subseteq X_1$, and also that $|p_1| \ge |E|/\lceil s(1+\epsilon)\rceil$. By the definition of λ, we have $|p_1|+|p_i| < |E|/s$ for $\lambda < i \le \ell$. Then

$$\sum_{\lambda<i\le\ell}|p_i| \le \ell(|E|/s-|p_1|) \le \ell(|E|/s-|E|/\ell) = (\ell/s-1)|E| \le \epsilon|E|,$$

since $\ell = |X_1| \le s(1+\epsilon)$. Thus, $\sum_{1\le i\le\lambda}|p_i| \ge (1-\epsilon)|E|$. Substituting this and $\ell \le s(1+\epsilon)$ into the previous equation, gives

$$\sum_{1\le i\le\lambda}\frac{|p_1\cap p_i|}{|p_1|} \ge -s(1+\epsilon)\epsilon + s(1+\epsilon)(1-\epsilon) = s(1+\epsilon)(1-2\epsilon).$$

We can now use an elementary expectation argument to complete the proof. Choose some $e' \in p_1$ uniformly at random, and let R_i, for $i\in\{1,\dots,\lambda\}$, be the indicator random variable of the event $e'\in p_i$. Then $\mathbf{E}[R_i] = \Pr[e'\in p_i] = \frac{|p_1\cap p_i|}{|p_1|}$, and

$$\mathbf{E}\left[\sum_{1\le i\le\lambda}R_i\right] = \sum_{1\le i\le\lambda}\frac{|p_1\cap p_i|}{|p_1|} \ge s(1+\epsilon)(1-2\epsilon).$$

Since $\sum_{1\le i\le\lambda}R_i$ is the number of sets $p_1,\dots,p_\lambda$ that contain e', the above inequality implies that there is some $e^*\in p_1$ that is contained in at least $s(1+\epsilon)(1-2\epsilon)$ sets. This implies that (b) holds for some $Z\subseteq\{x_1,\dots,x_\lambda\}\subseteq X_1$. □

The next lemma is obtained by iteratively applying Lemma 4. It states that if $|X_i| \le s\cdot(1+\epsilon)$ holds for all X_i (not just for X_1), and the total number of hyperedges is $|E| \ge s^k$, then there is a subset of hyperedges and an index d such that: for every $i\ne d$ at most two vertices from X_i are contained in those hyperedges; and at least $s\cdot(1+\epsilon)(1-2\epsilon)$ vertices from X_d are contained in the hyperedges.

Lemma 5. *Let $H=(X_1,\dots,X_k,E)$ be a k-partite hypergraph such that $|X_i| \le s\cdot(1+\epsilon)$ for all $1\le i\le k$, and $|E|\ge s^k$, where s is a positive integer and $0\le\epsilon<1/2$. There exist a set $\{e_1,e_2,\dots\}$ of hyperedges and an index $d\in\{1,\dots,k\}$ such that the set $U=\bigcup_r e_r$ satisfies*

(a) $|U\cap X_i|\le 2$ for all $i\ne d$; and

(b) $|U\cap X_d| \ge s\cdot(1+\epsilon)(1-2\epsilon)$.

Proof. We recursively construct sequences (Z_i), $i\in\{1,\dots,d\}$, and (H_i), $i\in\{0,\dots,d\}$, for some $d\in\{1,\dots,k\}$, such that $Z_i\subseteq X_i$ and $H_i = (X_{i+1},\dots,X_k,E_i)$ is a $(k-i)$-partite hypergraph. The index d is also determined by the recursive construction.

We initialize the recursive construction by setting $H_0 = H$. For d we just know that $1\le d\le k$ initially. For each $i\ge 1$, if $i\le d$ then we determine Z_i and H_i from H_{i-1} as follows. We have two cases.

- Case $i<k$: We let Z_i be a subset of X_i such that:
  (i) $|Z_i|\le 2$ and $|\bigcup_{z\in Z_i}\pi_z(E_{i-1})| \ge |E_{i-1}|/s$, or
  (ii) $|Z_i| \ge s\cdot(1+\epsilon)(1-2\epsilon)$ and $\bigcap_{z\in Z_i}\pi_z(E_{i-1})\ne\emptyset$.
  From Lemma 4, such a set Z_i exists. Then,
  - If (i) holds, we let $E_i = \bigcup_{z\in Z_i}\pi_z(E_{i-1})$, thus it holds $|E_i|\ge|E_{i-1}|/s$. We also establish that $d>i$.
  - If (ii) holds, we let E_i be a singleton set $E_i=\{e^*\}$ such that $e^*\in\bigcap_{z\in Z_i}\pi_z(E_{i-1})$. We also set $d=i$, and the recursive construction is completed.
- Case $i=k$: We let $Z_i = \bigcup_{e\in E_{i-1}} e\subseteq X_k$, thus $|Z_i| = |E_{i-1}|$. Also we let $E_i=\{\emptyset\}$ and set $d=k$.

It is immediate from the above construction that for $1\le i<d$, $|E_i|\ge|E_{i-1}|/s$. And since $|E_0| = |E| \ge s^k$, it follows that for $0\le i<d$,

$$|E_i| \ge s^{k-i}.$$

Also, for $1\le i<d$,

$$1\le|Z_i|\le 2,$$

where the left inequality holds because $|E_i|\ge s^{k-i}>0$. For $|Z_d|$, we have that $|Z_d|\ge s\cdot(1+\epsilon)(1-2\epsilon)$ if $d<k$. And if $d=k$ then $|Z_d| = |E_{k-1}| \ge s^{k-(k-1)} = s$. Thus, in all cases

$$|Z_d| \ge s\cdot(1+\epsilon)(1-2\epsilon).$$

Finally, we note that the set E_d is a singleton set, $E_d=\{e^*\}$, where $e^*=\emptyset$ if $d=k$.

Consider now the sequence $E = F_0\supseteq F_1\supseteq\cdots\supseteq F_d\supseteq F$, where $F_i = \bigcup_{z\in Z_i}\sigma_z(F_{i-1})$ for $1\le i\le d$, and $F=\sigma_{e^*}(F_d)$.[1] It is immediate that F contains all hyperedges $e=\{x_1,x_2,\dots,x_d\}\cup e^*\in E$ such that $x_i\in Z_i$ for $1\le i\le d$. We will show that F is the desired set $\{e_1,e_2,\dots\}$ of hyperedges, i.e., we will show that (a) and (b) hold for $U=\bigcup_{e\in F}e$.

For $1\le i\le d$, let $U_i = \bigcup_{e\in F_i}e$. Since $F_1\supseteq F_2\supseteq\cdots\supseteq F_d\supseteq F$, it follows $U_1\supseteq U_2\supseteq\cdots\supseteq U_d\supseteq U$. For $1\le i<d$, we have that $U_i\cap X_i\subseteq Z_i$, by the definition of F_i, thus $U\cap X_i\subseteq U_i\cap X_i\subseteq Z_i$, and $|U\cap X_i|\le|Z_i|\le 2$. For $d<i\le k$, we have $U\cap X_i\subseteq e^*\cap X_i$ thus $|U\cap X_i|\le 1$. We have thus proved (a).

Let $x_d\in Z_d$. Recall that $E_d=\{e^*\}$, where $e^*\in\bigcap_{z\in Z_d}\pi_z(E_{d-1})$ if $d<k$ and $e^*=\emptyset$ otherwise. It follows that $\{x_d\}\cup e^*\in E_{d-1}$. Similarly, by iteratively applying the definition of $E_i=\bigcup_{z\in Z_i}\pi_z(E_{i-1})$, for $i=d-1,d-2,\dots,1$, we obtain that there is a sequence of vertices $x_{d-1},x_{d-2},\dots,x_1$, where $x_i\in Z_i$ for $1\le i<d$, such that

$$\{x_1\}\cup\cdots\cup\{x_{d-1}\}\cup\{x_d\}\cup e^*\in E_0 = E.$$

Therefore, $\{x_1,\dots,x_d\}\cup e^*\in F$. Since we can repeat the argument for any $x_d\in Z_d$ to obtain a hyperedge in F, it follows

[1] The definition of sequence (F_i), $i\in\{0,\dots,d-1\}$, is the same as that of (E_i), $i\in\{0,\dots,d-1\}$, except that operator σ is used instead of π.

$Z_d \subseteq \bigcup_{e\in F} e = U$. Since also $Z_d \subseteq X_d$ and $|Z_d| \geq s\cdot(1+\epsilon)(1-2\epsilon)$, we conclude that

$$|U \cap X_d| \geq |Z_d| \geq s\cdot(1+\epsilon)(1-2\epsilon).$$

This proves (b). □

We now use Lemma 5 to prove our main result.

Proof of Lemma 2. We recursively construct sequences (y_i), (H_i), (F_i), and (d_i), $i \in \{1,\ldots,m\}$, such that $y_i \in Y$, H_i is a hypergraph, F_i is a subset of hyperedges of H_i, and d_i is an integer.

Recall that $y_0 \in Y$ is given in the lemma's statement. For each $i \in \{1,\ldots,m\}$, given y_{i-1}, we define y_i, H_i, F_i, and d_i as follows:

Let $k = 4\ell$. Let $X_{i,1},\ldots,X_{i,k}$ be mutually disjoint subsets of X_i, such that for $1 \leq j \leq k$,

$$|X_{i,j}| = \lfloor 27\delta\ell \rfloor.$$

We can define such disjoint sets because $|X_i| \geq 108\delta\ell^2 = k\cdot 27\delta\ell$. Let $(X_{i,1},\ldots,X_{i,k},E_i)$ be a complete k-partite hypergraph, and for each $y \in Y$, let

$$E_{i,y} = \{e \in E_i : f_{y_{i-1}}(e) = y\}.$$

We let y_i be an element $y \in Y$ that maximizes $|E_{i,y}|$, i.e., $|E_{i,y_i}| \geq |E_{i,y}|$ for all $y \in Y$. We then define H_i to be the k-partite hypergraph $H_i = (X_{i,1},\ldots,X_{i,k},E_{i,y_i})$. We observe that

$$|E_{i,y_i}| \geq |E_i|/|Y| \geq \lfloor 27\delta\ell \rfloor^k / 2^\ell \geq (\lfloor 27\delta\ell \rfloor/1.2)^k,$$

where for the last inequality we used that $\ell = k/4$ and $2^{1/4} < 1.2$. Next, we apply Lemma 5 to hypergraph H_i, for $s = \lfloor 27\delta\ell \rfloor/1.2$ and $\epsilon = 0.2$. We let F_i be the set of hyperedges and d_i the index predicted by the lemma. Then $\emptyset \neq F_i \subseteq E_{i,y_i}$, $d_i \in \{1,\ldots,k\}$, and the set $U_i = \bigcup_{e\in F_i} e$ satisfies[2]

(i) $|U_i \cap X_{i,j}| \leq 2$ for all $j \neq d_i$, and
(ii) $|U_i \cap X_{i,d_i}| \geq s\cdot(1+\epsilon)(1-2\epsilon) = 0.6\lfloor 27\delta\ell \rfloor$.

We define the sequences (A_i) and (V_i), $i \in \{1,\ldots,m\}$, using the objects constructed above, as follows. For each $i \in \{1,\ldots,m\}$, we let A_i be a hyperedge from the set F_i, and let

$$V_i = (U_i \setminus X_{i,d_i}) \cup A_i.$$

Clearly, $\emptyset \subsetneq A_i \subseteq V_i \subseteq X_i$. Also, since $A_i \in F_i \subseteq E_{i,y_i}$, we have that $f_{y_{i-1}}(A_i) = y_i$ holds, as desired.

For $1 \leq i \leq m$, we can express V_i as

$$V_i = (U_i \setminus X_{i,d_i}) \cup (A_i \cap X_{i,d_i}),$$

because $A_i \setminus X_{i,d_i} \subseteq U_i \setminus X_{i,d_i}$. Then

$$|V_i| = \sum_{j\neq d_i} |U_i \cap X_{i,j}| + |A_i \cap X_{i,d_i}| \leq 2(k-1)+1 = 8\ell - 1,$$

where the inequality holds because of (i). Also,

$$|U_i \setminus V_i| = |U_i \cap X_{i,d_i}| - |A_i \cap X_{i,d_i}| \geq 0.6\lfloor 27\delta\ell \rfloor - 1 \geq 16\delta\ell - 2,$$

where the first inequality holds because of (ii).

Now, let $D \subseteq X$ such that $|D| \leq \delta\cdot|\bigcup_{1\leq i\leq m} V_i|$. Then

$$|D| \leq \delta\cdot\sum_{1\leq i\leq m} |V_i| \leq \delta m(8\ell-1) \leq m(8\delta\ell - 1),$$

since $\delta \geq 1$. Let

$$I_D = \big\{i \in \{1,\ldots,m\} : (U_i \setminus V_i) \setminus D \neq \emptyset\big\}.$$

To lower-bound the size of I_D we observe that the sets $U_i \setminus V_i$, $1 \leq i \leq m$, are mutually disjoint (because the sets X_i, $1 \leq i \leq m$, are mutually disjoint), and we use inequality $|U_i \setminus V_i| \geq 16\delta\ell - 2$ shown above, to obtain that the number of distinct sets $U_i \setminus V_i$ that are subsets of D is at most

$$\frac{|D|}{16\delta\ell - 2} \leq \frac{m(8\delta\ell-1)}{16\delta\ell-2} = \frac{m}{2}.$$

It follows that $|I_D| \geq m - m/2 \geq m/2$, as desired.

Finally we define the sequences (z_i) and (B_i), $i \in I_D$, as follows. For each $i \in I_D$, let

$$z_i \in (U_i \setminus V_i) \setminus D.$$

Such a z_i exists because $(U_i \setminus V_i) \setminus D \neq \emptyset$, since $i \in I_D$. Also, since $V_i = (U_i \setminus X_{i,d_i}) \cup A_i \supseteq U_i \setminus X_{i,d_i}$, we have

$$(U_i \setminus V_i) \setminus D \subseteq (X_{i,d_i} \setminus V_i) \setminus D = X_{i,d_i} \setminus (V_i \cup D).$$

Thus $z_i \in X_{i,d_i} \setminus (V_i \cup D) \subseteq X_i \setminus (V_i \cup D)$, as desired. Next, for each $i \in I_D$, let $e_i \in F_i$ be hyperedge such that

$$e_i \cap X_{i,d_i} = z_i.$$

Clearly, such a hyperedge e_i exists since $z_i \in U_i \cap X_{i,d_i}$. We let

$$B_i = e_i \setminus \{z_i\}.$$

Then $B_i \subseteq U_i \setminus X_{i,d_i} \subseteq V_i$, as desired. Finally, since $e_i \in F_i \subseteq E_{i,y_i}$, we have $f_{y_{i-1}}(B_i \cup \{z_i\}) = f_{y_{i-1}}(e_i) = y_i$. This concludes the proof of Lemma 2. □

3.3 Invariants

For every schedule σ, configuration C, and shared object R, we define the following:

- $P(\sigma)$: the set of all processes that have steps in σ.
- $E(C,\sigma)$: the execution determined by σ starting in configuration C.
- $val_R(C,\sigma)$: the value (state) of R at the end of $E(C,\sigma)$.
- $state_p(C,\sigma)$: the state of p at the end of $E(C,\sigma)$.
- $last_R(C,\sigma)$: the process that last performed an operation on R at the end of $E(C,\sigma)$; or $\perp$ if no process has ever performed an operation on R.
- $F(C,\sigma)$: the set of processes that have finished their superpassage at the end of $E(C,\sigma)$.

We also define $E(\sigma) = E(C_0,\sigma)$, $val_R(\sigma) = val_R(C_0,\sigma)$, $state_p(\sigma) = state_p(C_0,\sigma)$, $last_R(\sigma) = last_R(C_0,\sigma)$, and $F(\sigma) = F(C_0,\sigma)$. Then let d be a sufficiently large constant and $k = w^d$. Finally, given any array $A[0..2^n-1]$ and any set $S \subseteq \mathcal{P}$, we use $A[S]$ to denote $A[\sum_{p\in S} 2^{p-1}]$.

We now formally define the invariants that each array of schedules should satisfy, and detail the construction of these arrays of schedules. Let i be a non-negative integer, and $A[0..2^n-1]$ be an array such that each array entry contains either a schedule or $\perp$. Then we say that $A[0..2^n-1]$ is *i-compliant* if it satisfies the following invariants:

(I1) For every set $S \subseteq \mathcal{P}$, if $A[S] \neq \perp$, then $P(A[S]) \subseteq S$. (Note that this implies $F(A[S]) \subseteq S$.)
(I2) There is a unique set $S_{max} \subseteq \mathcal{P}$ such that for every set $S \subseteq \mathcal{P}$, $A[S] \neq \perp$ if and only if $F(A[S_{max}]) \subseteq S \subseteq S_{max}$.

[2] $F_i \neq \emptyset$ holds because of (ii).

(I3) For every process $p \in S_{max}$ and every set $S \subseteq \mathcal{P}$ that contains p, if $A[S] \neq \perp$, then $state_p(A[S]) = state_p(A[S_{max}])$.

(I4) $F(A[S]) = F(A[S_{max}])$ for every set $S \subseteq \mathcal{P}$ with $A[S] \neq \perp$. (Note that this invariant immediately follows from Invariants (I1), (I2), and (I3).)

(I5) For every shared object $R \in \mathcal{R}$, there is a value y_R such that for every set $S \subseteq \mathcal{P}$, if $A[S] \neq \perp$, then:

$$val_R(A[S]) = \begin{cases} val_R(A[S_{max}]) & \text{if } last_R(A[S_{max}]) \in S \\ y_R & \text{otherwise} \end{cases}$$

Note that it is possible that $y_R = val_R(A[S_{max}])$. Furthermore, if $last_R(A[S_{max}]) = \perp \notin S$, then $val_R(A[S]) = y_R$ for every set $S \subseteq \mathcal{P}$ with $A[S] \neq \perp$.

(I6) For every set $S \subseteq \mathcal{P}$ with $A[S] \neq \perp$, during $E(A[S])$, each process crashes at most once, and each process that is not in $F(A[S])$ never crashes.

(I7) For every set $S \subseteq \mathcal{P}$ with $A[S] \neq \perp$, each process that is not in $F(A[S])$ does not enter the critical section during $E(A[S])$.

(I8) In the DSM model, for every process $p \in S_{max} \setminus F(A[S_{max}])$, every shared object $R \in \mathcal{R}$ owned by p, and every set $S \subseteq \mathcal{P}$ with $A[S] \neq \perp$, R can only be accessed by p during $E(A[S])$. (Or equivalently, in the DSM model, for every set $S \subseteq \mathcal{P}$ such that $A[S] \neq \perp$, during $E(A[S])$, each shared object $R \in \mathcal{R}$ can only be accessed by its owner if the owner of R is in $S_{max} \setminus F(A[S_{max}])$.)

(I9) In the CC model, for every process $p \in S_{max} \setminus F(A[S_{max}])$, there is a set $\mathcal{R}_p$ of shared objects such that for every set $S \subseteq \mathcal{P}$ that contains p, if $A[S] \neq \perp$, then the set of shared objects that p has valid cache copies of at the end of $E(A[S])$ is exactly $\mathcal{R}_p$. (Or equivalently, for every set $S \subseteq \mathcal{P}$ such that $A[S] \neq \perp$, and every process $p \in S \cap (S_{max} \setminus F(A[S_{max}]))$, the set of shared objects that p has valid cache copies of at the end of $E(A[S])$ is exactly the same as at the end of $E(A[S_{max}])$.)

(I10) For every set $S \subseteq \mathcal{P}$ and every process $p \in S \setminus F(A[S])$, if $A[S] \neq \perp$, then p incurs at least i RMRs during $E(A[S])$.

Let i be a non-negative integer, and $A[0..2^n - 1]$ be an array that is i-compliant. Then we denote by $S_{max}(A[0..2^n - 1])$ the unique set of Invariant (I2).

Let $\sigma_{round}[0..\infty, 0..2^n - 1]$ be a table with all entries initially containing $\perp$. Our goal is to fill in the table such that for every non-negative integer i, either $\sigma_{round}[i, 0..2^n - 1]$ is i-compliant, or $i \in \Omega(\log_w n)$.

The proof of the above invariants can be found in the full version of the paper [3].

3.4 Completing the Proof of Theorem 1

For every integer $i \geq 0$, if $\sigma_{round}[i, 0..2^n - 1]$ is i-compliant, then let $S^i_{max} = S_{max}(\sigma_{round}[i, 0..2^n - 1])$, and let $n_i = |S^i_{max} \setminus F(\sigma_{round}[i, S^i_{max}])|$. We show the following lemma in [3].

Lemma 6. *For every integer $i \geq 1$, if $\sigma_{round}[i, 0..2^n - 1]$ is i-compliant, then $n_i \geq n_{i-1}/(64w^{d+1}) - 2$.*

Since $S_{max}(\sigma_{round}[0, 0..2^n - 1]) = \mathcal{P}$ and $\sigma_{round}[0, \mathcal{P}]$ is the empty schedule, $F(\sigma_{round}[0, \mathcal{P}]) = \emptyset$ and $n_0 = n$. By Lemma 6, for every positive integer i, if $\sigma_{round}[i, 0..2^n - 1]$ is i-compliant, then $n_i \geq n_{i-1}/O(w^{d+1})$.

Thus, if $\mathcal{I}$ is the largest positive integer such that $\sigma_{round}[\mathcal{I}, 0..2^n - 1]$ is $\mathcal{I}$-compliant, then $\mathcal{I}$ is in $\Omega(\log_w n)$. So $\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}]$ contains a schedule such that:

- Since we reach the $\mathcal{I}$-th iteration, $\sigma_{round}[\mathcal{I} - 1, 0..2^n - 1]$ has at least $2^{(k^3)}$ non-$\perp$ entries, i.e., $n_{\mathcal{I}-1} \geq k^3 = w^{3d}$. So $n_{\mathcal{I}} \geq w^{3d}/(64w^{d+1}) - 2$, which since d is sufficiently large constant, $n_{\mathcal{I}} \geq w^d$. Thus $|S^{\mathcal{I}}_{max} \setminus F(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])| \geq w^d$.
- For every process $p \in S^{\mathcal{I}}_{max} \setminus F(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$, p incurs at least $\mathcal{I}$ RMRs during $E(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$ (Invariant (I10)).
- For every process $p \in S^{\mathcal{I}}_{max} \setminus F(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$, p never crashes during $E(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$ (Invariant (I6)).
- For every process $p \in S^{\mathcal{I}}_{max} \setminus F(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$, p never enters the critical section during $E(\sigma_{round}[\mathcal{I}, S^{\mathcal{I}}_{max}])$ (Invariant (I7)).

Thus we have proven Theorem 1.

4 CONCLUSION

In this paper we proved tight lower bounds for recoverable mutual exclusion, showing a tradeoff between the word-size, w, of atomic shared base objects and the worst-case RMR complexity. These are the first such lower bounds that are independent of the type of atomic shared memory operations the system provides. The lower bounds are tight for $w = (\log n)^{\Omega(1)}$, matching an upper bound of Katzan and Morrison [19]. For $w = (\log n)^{o(1)}$ it is not known if our RMR complexity lower bound of $\Omega(\log n/\log\log n)$ is tight—we are not aware of any RME algorithm that works with such small shared memory words.

Our lower bound is for the maximum number of RMRs a process incurs during a passage. It is most likely not possible to extend this result to amortized RMR complexity (which measures the average number of RMRs per passage in a worst-case execution). This is due to an RME algorithm with $O(1)$ amortized RMR complexity by Chan and Woelfel [4]. While this algorithm uses unbounded fetch-and-increment objects, it seems quite possible that these can be replaced with bounded ones.

Jayanti, Jayanti, and Joshi [14] presented an RME algorithm with constant RMR complexity in the *system-wide* failure model, where all processes can only crash at the same time. (See also [11] for an algorithm that requires some specialized OS support.) While this algorithm uses an unbounded fetch-and-increment object, it seems that it may be possible to bound it. This would mean that our lower bound, which inherently relies on individual process crashes, cannot be generalized to the system-wide failure model.

It may also be interesting to explore if randomization can help circumvent our impossibility result.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. Support is gratefully acknowledged from the Agence Nationale de la Recherche (ANR) under project ByBloS (ANR-20-CE25-0002), the Natural Science and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN/2019-04852, and the Canada Research Chairs program.

REFERENCES

[1] James H. Anderson and Yong-Jik Kim. 2002. An Improved Lower Bound for the Time Complexity of Mutual Exclusion. *Distributed Computing* 15 (2002), 221–253.

[2] Hagit Attiya, Danny Hendler, and Philipp Woelfel. 2008. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*. 217–226.

[3] David Yu Cheng Chan, George Giakkoupis, and Philipp Woelfel. 2023. Word-Size RMR Trade-offs for Recoverable Mutual Exclusion. (May 2023). https://inria.hal.science/hal-04098408 (full version of this paper).

[4] David Yu Cheng Chan and Philipp Woelfel. 2020. Recoverable Mutual Exclusion with Constant Amortized RMR Complexity from Standard Primitives. In *Proceedings of the 2020 ACM Symposium on Principles of Distributed Computing (PODC)*. 181–190. https://doi.org/10.1145/3382734.3405736

[5] David Yu Cheng Chan and Philipp Woelfel. 2021. Tight Lower Bound for the RMR Complexity of Recoverable Mutual Exclusion. In *Proceedings of the 40th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 533–543. https://doi.org/10.1145/3465084.3467938

[6] Travis Craig. 1993. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical Report TR-93-02-02, Department of Computer Science, University of Washington.

[7] Sahil Dhoked and Neeraj Mittal. 2020. An Adaptive Approach to Recoverable Mutual Exclusion. In *Proceedings of the 2020 ACM Symposium on Principles of Distributed Computing (PODC)*. 1–10. https://doi.org/10.1145/3382734.3405739

[8] E. W. Dijkstra. 1965. Solution of a Problem in Concurrent Programming Control. *Commun. ACM* 8 (1965), 569.

[9] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. 2012. RMR-Efficient Implementations of Comparison Primitives Using Read and Write Operations. *Distributed Computing* 25, 2 (2012), 109–162. https://doi.org/10.1007/s00446-011-0150-8

[10] Wojciech Golab and Danny Hendler. 2017. Recoverable Mutual Exclusion in Sub-logarithmic Time. In *Proceedings of the 2017 ACM Symposium on Principles of Distributed Computing (PODC)*. 211–220. https://doi.org/10.1145/3087801.3087819

[11] Wojciech Golab and Danny Hendler. 2018. Recoverable Mutual Exclusion Under System-Wide Failures. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing (PODC)*. 17–26. https://doi.org/10.1145/3212734.3212755

[12] Wojciech Golab and Aditya Ramaraju. 2019. Recoverable mutual exclusion. *Distributed Computing* 32, 6 (2019), 535–564. https://doi.org/10.1007/s00446-019-00364-0

[13] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. 2018. Optimal Recoverable Mutual Exclusion Using only FASAS. In *Networked Systems (NETYS) - 6th International Conference*. 191–206. https://doi.org/10.1007/978-3-030-05529-5_13

[14] Prasad Jayanti, Siddhartha Jayanti, and Anup Joshi. 2023. Constant RMR Recoverable Mutex under System-wide Crashes. arXiv:2302.00748 https://doi.org/10.48550/arXiv.2302.00748

[15] Prasad Jayanti, Siddhartha V. Jayanti, and Anup Joshi. 2019. A Recoverable Mutex Algorithm with Sub-logarithmic RMR on Both CC and DSM. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*. 177–186. https://doi.org/10.1145/3293611.3331634

[16] Prasad Jayanti and Anup Joshi. 2017. Recoverable FCFS Mutual Exclusion with Wait-Free Recovery. In *Proceedings of the 31st International Symposium on Distributed Computing (DISC)*. 30:1–30:15. https://doi.org/10.4230/LIPIcs.DISC.2017.30

[17] Prasad Jayanti and Anup Joshi. 2022. Recoverable Mutual Exclusion with Abortability. *Computing* 104, 10 (2022), 2225–2252. https://doi.org/10.1007/s00607-022-01105-1

[18] Stasys Jukna. 2011. *Extremal combinatorics: With applications in computer science*. Springer. https://doi.org/10.1007/978-3-642-17364-6

[19] Daniel Katzan and Adam Morrison. 2020. Recoverable, Abortable, and Adaptive Mutual Exclusion with Sublogarithmic RMR Complexity. In *Proceedings of 24th International Conference On Principles Of DIstributed Systems (OPODIS)*. 15:1–15:16. https://doi.org/10.4230/LIPIcs.OPODIS.2020.15

[20] Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proc. of 8th International Symposium on Parallel Processing*. 165–171. https://doi.org/10.1109/IPPS.1994.288305

[21] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 1 (1991), 21–65.

[22] Aditya Ramaraju. 2015. *RGLock: Recoverable mutual exclusion for non-volatile main memorysystems*. Master's thesis. University of Waterloo. https://uwspace.uwaterloo.ca/handle/10012/9473

[23] Jae-Heon Yang and James H. Anderson. 1995. A Fast, Scalable Mutual Exclusion Algorithm. *Distributed Computing* 9, 1 (1995), 51–60.