Randomized Abortable Mutual Exclusion with Constant Amortized RMR Complexity on the CC Model

George Giakkoupis, Philipp Woelfel

Randomized Abortable Mutual Exclusion with Constant Amortized RMR Complexity on the CC Model

George Giakkoupis*
INRIA, Rennes
France
george.giakkoupis@inria.fr

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

ABSTRACT

We present an abortable mutual exclusion algorithm for the cache-coherent (CC) model with atomic registers and CAS objects. The algorithm has constant expected amortized RMR complexity in the oblivious adversary model and is deterministically deadlock-free. This is the first abortable mutual exclusion algorithm that achieves $o(\log n/\log\log n)$ RMR complexity.

KEYWORDS

????

1 INTRODUCTION

Mutual exclusion, introduced by Dijkstra [17], is one of the best studied problems in concurrent computing. A mutual exclusion object (or *lock*) allows processes to coordinate their access to a shared resource by serializing the execution of a piece of code, called the *critical section*. Each process obtains a lock through an *entry section*, but at any time at most one process can own the lock. A process that owns the lock executes the critical section, and to release the lock it executes an *exit section*. Raynal devoted a textbook [34] to mutual exclusion research up to the mid 80s, and a survey by Anderson, Kim, and Herman [5] covers the research between 1986 and 2003.

Early mutual exclusion algorithms did not take into account the gap between high processor speeds and the low speed and bandwidth of the processor-memory interconnect [13]. In *distributed shared memory (DSM)* systems, each shared variable is permanently locally accessible to a single processor and remote to all other processors. In *cache-coherent (CC)* systems, each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a *coherence protocol*. Memory accesses that cannot be resolved locally and have to traverse the processor-to-memory interconnect are called *remote memory references* (RMRs). RMRs are orders of magnitude slower than local memory accesses. Hence, the performance of many algorithms for shared memory multiprocessor systems depends critically on the number of RMRs they incur [6, 32]. Recent research has almost entirely used the RMR complexity as a metric for the performance of mutual exclusion algorithms [3, 4, 6, 9, 12, 14, 15, 19, 20, 23, 24, 26–31, 33]. Tradeoffs between RMRs and fence operations have also been studied more recently [8, 10, 11].

However, even RMR-efficient mutual exclusion locks do not meet a critical demand of many systems [35]. Specifically, the locks employed in database and real-time systems must support a "timeout" capability that allows a process waiting too long for the lock, to abort its attempt. In database systems, such as Oracle's Parallel Server and IBM's DB2, the ability of a thread to abort lock attempts serves the dual purpose of recovering from a transaction deadlock and tolerating preemption of the thread that holds the lock [35]. In real time systems, the abort capability can be used to avoid overshooting a deadline. Locks that allow a process to abort its attempt to enter the critical section, and return to the remainder section within a finite number of their own steps, are called *abortable* locks.

Using strong primitives, such as fetch&increment objects, it is possible to implement mutual exclusion so that every process incurs only a constant number of RMRs per passage through the critical section (for an overview of algorithms using strong primitives see [25]). But those primitives are often not available in common architectures. In systems that provide only atomic registers and compare&swap (CAS) objects, the RMR complexity of the mutual exclusion problem is higher. (In fact, CAS and load-linked/store-conditional objects can be simulated from registers with $O(1)$ RMR overhead per operation [21].)

Randomization can be used to improve the efficiency of mutual exclusion algorithms. For the analysis of randomized algorithms, the order in which processes take steps is determined by an *adversary*. Among the most common adversary models are the *strong adaptive adversary*, where scheduling decisions can depend on all past events including local coin flips, and the *oblivious adversary*, where scheduling decisions are independent of random decisions made by processes (i.e., the adversary fixes the schedule in advance).

Unless mentioned otherwise, the results discussed below hold for the CC and the DSM models with atomic registers and CAS objects, and n is the number of processes. The deterministic RMR complexity of mutual exclusion is $\Theta(\log n)$ RMRs per passage through the critical section [9, 36]. Jayanti showed that this RMR complexity can even be achieved for abortable mutual exclusion [26]. Randomization can only slightly improve RMR complexity in the strong

*Dr. Trovato insisted his name be first.
†The secretary disavows any knowledge of this author's actions.

PODC'17, July 25-27, 2017, Washington, DC, USA.

ACM ISBN 978-1-4503-4992-5/17/07…$15.00.
https://doi.org/10.1145/3087801.3087837

adaptive adversary model. Giakkoupis and Woelfel [19] showed that any mutual exclusion algorithm in this model has expected RMR complexity $\Omega(\log n/\log\log n)$, matching an upper bound by Hendler and Woelfel [24].

Recently, researchers have increasingly focused on devising efficient randomized algorithms for the weaker, oblivious adversary model, e.g., for test-and-set [2, 18] or consensus [7]. Bender and Gilbert devised a randomized mutual exclusion algorithm that achieves $O(\log^2 \log n)$ expected amortized RMR complexity against the oblivious adversary in the CC model [12]. The algorithm is Monte Carlo in the sense that it provides only probabilistic progress guarantees. A deterministically deadlock-free algorithm with constant expected amortized RMR complexity in the DSM model was presented by Giakkoupis and Woelfel [20]. But this algorithm is not RMR efficient for the CC model.

None of the randomized algorithms above are abortable or can easily be made so. A much more complicated randomized abortable mutual exclusion algorithm was devised by Pareek and Woelfel [33], and has expected RMR complexity $O(\log n/\log\log n)$ for an adversary that is slightly weaker than the strong adaptive adversary.

In this paper we present an abortable mutual exclusion algorithm with $O(1)$ expected amortized RMR complexity on the CC model with atomic registers and CAS objects, against a natural adversary model that is stronger than the oblivious adversary. The algorithm is deterministic deadlock-free. It is the first mutual exclusion algorithm that achieves optimal amortized RMR complexity in this model, and also the first one that combines abortability with an RMR complexity of $o(\log n/\log\log n)$.

Note that due to the constant-RMR implementations of linearizable CAS objects from registers [21], all deterministic mutual exclusion algorithms as well as some of the randomized ones, need only registers. But these simulations do not in general preserve the randomized RMR complexity of algorithms [16, 22]. The algorithm by Bender and Gilbert for the oblivious adversary model [12] needs atomic[1] CAS objects, and it is not clear whether they can be replaced by linearizable constant RMR implementations. On the other hand, the algorithm with constant amortized expected RMR complexity for the DSM model [20] uses only registers. However, for abortable mutual exclusion, it is significantly more difficult to use only registers. In particular, the CAS implementation from registers given in [21] is not wait-free, so an abort-mechanism would have to be made available for `CAS()` operations in order to use such implementations. We designed our algorithm in such a way, that atomic CAS objects can be replaced by linearizable, wait-free ones; the amortized RMR complexity of the mutual exclusion algorithm then becomes asymptotically the same as that of operations on the CAS objects. However, abortability of our algorithm is only preserved if operations on the CAS objects are abortable, but no such implementations with $O(1)$ RMR complexity are known.

Model and Problem Statement. We consider the standard cache-coherent (CC) model, where a set $\mathcal{P} = \{1, \ldots, n\}$ of n processes communicate by executing atomic operations on shared registers and CAS objects. A *compare-and-swap (CAS)* object C supports the operations C.`read()`, which returns the value of C, and C.`CAS(`*old*, *new*`)`, which writes *new* into C if $C = old$, and otherwise does not change C. A `CAS()` operation returns a Boolean value indicating whether it successfully changed the value. Processes are equipped with local caches for read operations on registers, and a cache protocol ensures coherency. A *remote memory reference* (*RMR*) is a shared memory access of a register that cannot be resolved locally (i.e., a cache miss). Each write of a register, and all operations on CAS objects are considered cache misses, and thus each of them incurs an RMR (even though some systems also provide write caches). A read by process p on a register R only incurs an RMR if p never read R before, or if some process wrote to it since p read it last.

A (non-abortable) mutual exclusion algorithm supports the methods `lock()` and `release()`. Once the `lock()` method terminates, the process is in the critical section. At the end of the critical section, the process calls `release()`, and it enters the remainder section when that call terminates. The safety property *mutual exclusion* requires that no two processes are in the critical section at the same time. In an *abortable* mutual exclusion algorithm, a process may receive a signal to abort at any point during its `lock()` method call. When that happens, the `lock()` method call must terminate within a finite number of the calling process' steps, and upon termination the process enters either the critical section or the remainder section. (Which one it is can be indicated by the return value of the `lock()` method, but for readability reasons we use **goto** statements in our code.) The standard progress condition for mutual exclusion is *deadlock-freedom*. This means that as long as there is a pending `lock()` method call, one such call will terminate, provided all processes that have pending `lock()` or `release()` method calls continue to take steps. For an abortable mutual exclusion algorithm it is also required that the `release()` method is wait-free, i.e., any process can finish it within a finite number of its own steps.

Processes can flip (private) coins to make random decisions. We consider an adversary that is stronger than the oblivious adversary; in particular correlations between whether past steps were RMRs and the future schedule are allowed. The precise adversary will be described in the analysis section. It is obtained by restricting a strong adaptive adversary, which determines the process to take the next step (or to receive the abort signal) by taking into account the entire past execution. The restriction is such that the adversary is not allowed to know the results of some past coin flips, as well as the decisions that a process based on those coin flips. However, the adversary knows when a process is poised to execute an operation on a CAS object, as well what operation the process is about to perform on the CAS object. Since the actual adversary definition is intricate, we state the main result for the oblivious adversary.

Theorem 1.1. *There is a randomized abortable mutual exclusion algorithm for the CC model with atomic registers and CAS objects, which has constant expected amortized RMR complexity against the oblivious adversary.*

Our algorithm uses unbounded sequence numbers for tagging. For example, when a process p attempts to capture the lock, it generates a unique tag by incrementing a sequence number, and augments this tag when it writes information to registers or CAS objects. This tag can be used by other processes, for example to identify that two writes by p to objects have been made during p's same attempt to capture the lock. This naive approach for tagging

[1] With atomic we mean that each operation is executed instantaneously.

requires unbounded objects. There are techniques for bounding tags (see e.g., [1]) without increasing step complexity.

In Section 2 we describe the main ideas of our algorithm, and then give a detailed line by line description. The complexity analysis and correctness proofs are involved, and deferred to the appendix.

2 ALGORITHM

Main Ideas. The pseudocode for our algorithm can be found on Pages 5–6. It is based on the convention that when a process receives an abort signal while executing `lock()`, then it continues executing its program until it reaches an `await()` statement, or its `lock()` method terminates. If the execution reaches an `await()` statement first, then the process immediately calls `abort()`.

We first describe the high level idea of the algorithm, assuming no aborts. Later we discuss how we deal with aborts. Some of the variables in the pseudocode are indexed by α. We will explain the purpose of this index later, and for now we will omit α from the corresponding variable names.

In order to avoid busy waiting on CAS objects, we use registers that change their value to signal waiting processes that CAS objects have been updated. For example after a process changed the value of the CAS object S from $(\cdot, \mathit{locked})$ to $(\cdot, \mathit{unlocked})$ in line 8, it writes to S' in line 9 to signal that it has changed S. Processes waiting for S to become $(\cdot, \mathit{unlocked})$, busy wait by reading S' instead of S (in line 23 and line 38). Similarly, registers $B'[p]$ and $A'[p]$ are used for signaling, and to avoid busy waiting on CAS objects $B[p][0]$ and $A[p]$, respectively.

The critical section is protected by `CAS` object S. When some process is critical (meaning it is in the critical section), the value of S is (i, locked) (where i is a sequence number), and otherwise it is $(i, \mathit{unlocked})$. A naive way to implement that would be as follows: To capture the lock, a process p waits until the value of S is $(\cdot, \mathit{unlocked})$, and then tries to change it to $(\cdot, \mathit{locked})$ with a `CAS()` operation (similar to lines 23–25). But then for each successful `CAS()` operation on S we may have $n-1$ failed ones, yielding a forbiddingly large RMR complexity.

The idea is now that before processes start waiting for S to become $(\cdot, \mathit{unlocked})$, they get a chance to contact each other and join forces (i.e., share work). To do that, each process p uses a *backpack*, represented by CAS object $B[p][1]$. Initially, the process sets the CAS object to an integer (which is a unique sequence number), meaning the backpack is *open*. We then use a randomized mechanism that gives other processes a chance to discover p, i.e., see that p is participating and has an open backpack. A process b that discovers p tries to write its own ID and a sequence number into $B[p][1]$ using a `CAS()` operation (line 16). If that succeeds, b can simply start waiting (in the repeat-until loop in lines 17–20), and let p do the work. After each attempt to enter the critical section via "winning" S, process p checks its backpack $B[p][1]$, and adds the process it found there, if any, to its local set *bpack* (see method `closeBackpacks()`). Once process p is critical, it adds all processes collected in *bpack* during its earlier attempts to win S, to a multiset Q (line 3). All accesses to Q will be protected by the critical section, so a sequential data structure (such as a linked list) can be used to implement Q. Then p removes a process q from Q (if Q is not empty) and tries to *promote* q, meaning that it calls a method `promote()` which is used to signal q that it is q's turn to become critical. The promotion mechanism requires some handshaking between p and q, because q may have decided to abort, and then p and q need to agree whether q enters the remainder or the critical section. If the promotion fails (i.e., q enters the remainder section), then p repeats this procedure by removing another process from Q, provided $Q \neq \emptyset$ (lines 4–6).

For this approach to work we need to deal with two issues: First, we need a way for processes to discover each other. Second, once a process b discovers a process p, p may finish its `lock()` and `release()` methods before b can join p's backpack. (Process p has to *close* its backpack before entering the remainder section, by changing the value of $B[p][1]$ to *closed*, so that b does not join the backpack and then deadlocks, waiting to be signaled by p.) Thus, b may waste an RMR by its `CAS()` operation in a failed attempt to join p's backpack. We have to make sure that this does not happen too often. In particular, at least a constant fraction of such backpack join attempts need to succeed in expectation. We now show how to deal with those issues.

To allow processes to discover each other, we use a register Z. Before each attempt to lock S, a process b first flips a random coin, and based on the outcome either it writes to Z a pair consisting of its ID and a sequence number, or it reads Z. In the latter case, if p finds the ID and sequence number of a different process, it tries to join that process' backpack as described earlier (lines 13–16). The idea is that if two processes consecutively access Z, and the first one writes to Z while the second one reads Z (which happens with probability 1/4), then the second one will discover the first one, and may later be able to enter the first one's backpack.

We now show how to deal with the second issue, which is to ensure that a process b, which discovered a process p, gets a chance of entering p's backpack before p closes it. Recall that eventually, processes that do not manage to enter a backpack, have to wait for S to become unlocked and try to lock S themselves (lines 23–25). Suppose S is locked, and let p^* be the process that locked S. While p^* is critical, it will try and select a random process r^* (and some other processes) among all of those that participated in the discovery phase (how this is done will be explained shortly). Process p^* will then allow r^* to go through the critical section before S gets unlocked, by adding it to the multiset Q (as described earlier). This has the effect that all processes waiting for S to become unlocked keep waiting and will not close their backpacks until the randomly chosen process r^* has taken sufficiently many steps. As a result, a constant fraction of all processes will make equally many steps as r^* (because the adversary does not know who r^* is), and that will suffice for each of them to join the backpack of their discovered process, if any.

To find a random process r^* we borrow a technique from [20]. We use a shared array $R[1 \ldots \ell]$, where $\ell = \lceil \log n \rceil$, and each array entry is initially $(0, 0)$. At the beginning of an attempt to capture the lock, a process writes a pair consisting of its ID and sequence number to $R[\lambda]$, where $\lambda \in \{1, \ldots, \ell\}$ is chosen according to a truncated geometric distribution, satisfying $\lambda = i$ with probability $\Theta(1/2^i)$ (lines 10–11). After locking S, process p^* scans this array from left to right until it finds the first index i such that $R[i] = (0, 0)$. Then it adds each process found to Q, and erases all registers that it

read on $R[]$. Let K be the set of processes that write to $R[]$ before the point t when p^* begins scanning $R[]$, and let $\kappa = \lceil \log |K| \rceil$. Then with constant probability all registers $R[0], \ldots, R[\kappa]$ were written to, and exactly one process in K wrote to $R[\kappa]$. Given that this happens, the process r^* that wrote to $R[\kappa]$ is (roughly) uniformly distributed over K. As a result, in expectation $\Omega(|K|)$ processes will manage to join backpacks of other processes while S is locked. Note that this is just a simplified description of the core insight; the intricate analysis considers a possibly larger set of writes, depending on the execution.

However, processes that write to $R[]$ after p^* finished scanning the array, and before S gets unlocked again, may not have a chance of joining a backpack. To deal with this issue we use instead two arrays, $R_0[]$ and $R_1[]$, and processes choose one of the two arrays to write to at random. Moreover, which of the two arrays is being scanned when a process wins S, alternates with each such win of S. More precisely, the sequence number stored with S gets incremented every time S gets locked, and when it is i, array $R_{i \bmod 2}$ is scanned. As a result, all processes that write to $R[(i+1) \bmod 2]$ while $S \in \{(i, \mathit{locked}), (i, \mathit{unlocked})\}$ get a chance of joining some other process' backpack. Since a process chooses at random which of the two arrays $R_0[]$ and $R_1[]$ to write to, each process has a $1/2$ probability of choosing the right "side". Then in expectation a constant fraction of the processes that write to $R_\alpha[]$, $\alpha \in \{0, 1\}$, will be able to join a backpack, and then later get added to multiset Q by the backpack owner.

Processes that neither manage to join any backpack nor lock S, will simply restart the procedure. Before a process p does that, it has to check whether its backpack $B[p][1]$ is occupied. If so, it has to move that backpacked process into a local variable *bpack*, in order to make room in its backpack for the next attempt (see method `closeBackpacks()`).

Next we describe how our algorithm accommodates process aborts. Suppose that a process p gets an abort signal while it is executing its `lock()` method. Then p calls method `abort()` as soon as it is busy waiting in one of the `await()` statements. In the simplest case, p has no process in its backpack set *bpack*. Then p can go to the remainder section, but as explained above, handshaking is necessary to synchronize with a potential process in the critical section that is trying to promote p. The handshaking mechanism for an aborting process is implemented in method `giveUp()`.

As a result of the above handshaking mechanism, p may either enter the remainder section, or get promoted in the critical section without any further waiting. In the former case, and if p has some processes in its backpack set *bpack* when it aborts, it has to make sure that those processes do not deadlock, because they are waiting under the assumption that they will eventually be added to the multiset Q and then get promoted. Since we can only allow non-concurrent access to Q, an aborting process cannot add the elements of its backpack to Q. Therefore (see the while-loop in `abort()`) p removes an arbitrary process r from *bpack*, and tries to hand over to it the set of remaining processes in *bpack*, by writing that set to CAS object $B[r][0]$, by using a `CAS()` operation. It then signals r, using variable $B'[r]$, that $B[r][0]$ has changed. If that `CAS()` succeeds, process r will find out (because while in a backpack, it also spins on $B'[r]$ in line 18), and it will learn that it has been removed from p's backpack. When that happens, r will continue trying to capture the lock itself. But while waiting, r may have also aborted. If it did so, it would have closed $B[r][0]$, and p would not have managed to hand over its backpack (i.e., the corresponding `CAS()` would have failed). Therefore, p has to repeat the procedure of removing processes from its backpack and trying to hand over the backpack to them, until this succeeds, or until its backpack is empty.

One issue to be careful about is to avoid cyclic backpack relationships. Let $b \leftarrow p$ denote that process b is in process p's backpack (at a fixed point in time). If the binary relation $\leftarrow$ is cyclic at any point, then processes deadlock, because each process that is part of the cycle will not make progress until its carrier (the process whose backpack it is in) passes through the critical section. This affects some implementation decisions, as discussed below.

Class CCLock

shared:

// $\ell := \lceil \log n \rceil$

Register $Z_s, R_s[1 \dots \ell+1]$, for $s \in \{0,1\}$; **init** $(0,0)$ // Stores values in $\mathcal{P}_0 \times \mathbb{N}_0$

Compare-and-Swap S; **init** $(0, unlocked)$ // Stores a pair in $\mathbb{N}_0 \times \{locked, unlocked\}$

Compare-and-Swap $A[1 \dots n]$; **init** $(0, done)$ // Stores pairs in $\mathbb{N}_0 \times \{want, critical, done\}$

Compare-and-Swap $B[1 \dots n][0 \dots 1]$; **init** *closed* // Stores values in $\mathbb{N} \cup \{closed\} \cup 2^{\mathcal{P} \times \mathbb{N}}$

Sequential Multiset Q; **init** $\emptyset$ // Stores pairs in $\mathcal{P} \times \mathbb{N}$

Register S'; **init** 0 // Stores values in $\mathbb{N}_0$

Register $A'[1 \dots n]$; **init** $(0,0)$ // Stores values in $\mathcal{P}_0 \times \mathbb{N}_0$

Register $B'[1 \dots n], Y[1 \dots n], X[1 \dots n][1 \dots \infty]$; **init** False // Stores Boolean values

// All other variables are local and their scope spans the entire class

// *abort_sig* is a local Boolean; it becomes True when an abort signal is received

Method lock()

```
1  abort_sig := False; bpack := ∅
2  while True do
3      (i, ·) := S; await(S' ≥ i or abort_sig)
4      if abort_sig then abort()
5      (c, ·) := A[id()]; c := c + 1
6      A[id()].CAS((c − 1, done), (c, want))
7      B'[id()] := False
8      foreach i ∈ {0, 1} do B[id()][i].CAS(closed, c)
9      Choose α ∈ {0, 1} uniformly at random
10     Choose λ ∈ {1, . . . , ℓ} s.t. Pr(λ = j) = 2^{−j} if 1 ≤ j < ℓ, and Pr(λ = ℓ) = 2^{−ℓ+1}
11     R_α[λ] := (id(), c)
12     Choose β ∈ {0, 1} uniformly at random
13     if β = 1 then Z_α := (id(), c)
14     else
15         (carrier, carrier_seq) := Z_α
16         if B[carrier][1].CAS(carrier_seq, {(id(), c)}) then
17             repeat
18                 await(A'[id()] = (·, c) or B'[id()] or abort_sig)
19                 B'[id()] := False
20             until A'[id()] = (·, c) or B[id()][0] ≠ c or abort_sig
21         end
22     end
23     (i, ·) := S; await(S' ≥ i or A'[id()] = (·, c) or abort_sig)
24     if S' ≥ i then
25         if S.CAS((i, unlocked), (i + 1, locked)) then
26             A[id()].CAS((c, want), (c, critical))
27             X[id()][c] := True
28             i := i + 1
29             for j = 2, . . . , ℓ + 1 do
30                 (r, d) := R_{i mod 2}[j]
31                 if (r, d) ≠ (0, 0) then R_{i mod 2}[j] := (0, 0)
32                 if (r, d) ≠ (0, 0) and r ≠ id() then
33                     Q := Q ∪ {(r, d)}
34                 end
35                 else goto critical section
36             end
37         end
38         await(S' ≥ i + 1 or A'[id()] = (·, c) or abort_sig)
39     end
40     if giveUp() then closeBackpacks()
41     else goto critical section
42 end
```

Method `closeBackpacks()`

```
1 foreach i ∈ {0, 1} do
2     if B[id()][i].CAS(c,closed) = False then
3         tmp := B[id()][i]
4         B[id()][i].CAS(tmp, closed); bpack := bpack ∪ tmp
5     end
6 end
```

Method `giveUp()`

```
1 X[id()][c] := True
2 (proc, seq) := A′[id()]
3 if seq = c then
4     Y[proc] := True
5     A[id()].CAS((c, want), (c, critical))
6 end
7 return A[id()].CAS((c, want), (c, done))
```

Method `promote(`r, d`)`

```
1 Y[id()] := False
2 A′[r] := (id(), d)
3 if X[r][d] = False then return True
4 if A[r].CAS((d, want), (d, critical)) then return True
5 return Y[id()]
```

Method `release()`

```
1 A[id()].CAS((c, critical), (c, done))
2 closeBackpacks()
3 Q := Q ∪ bpack; bpack := ∅
4 while Q ≠ ∅ do
5     Remove an arbitrary pair (r, d) from Q
6     if promote(r, d) then goto remainder section
7 end
8 (i, ·) := S; S.CAS((i, locked),(i, unlocked))
9 S′ := i
10 goto remainder section
```

Method `abort()`

```
1 while bpack ≠ ∅ do
2     Remove an arbitrary pair (r, d) from bpack
3     if B[r][0].CAS(d, bpack) then
4         bpack := ∅
5         B′[r] := True
6     end
7 end
8 goto remainder section
```

Implementation. We first describe the helper method `closeBackpacks()`. As explained above, during its attempt to enter the critical section a process' backpack may be joined by other processes. The shared CAS objects $B[p][j]$, $j \in \{0,1\}$, represent backpacks which are empty if their value is p's current sequence number c, and closed (i.e., cannot be joined by other processes) if their value is *closed*. Otherwise, each of those objects may store a set of pairs of process IDs and their sequence numbers. The purpose of method `closeBackpacks()` is to transfer the contents of those sets to p's local variable *bpack* and to close the shared backpacks. To do that, a process simply first tries to change the value of each CAS object $B[p][j]$ from c to *closed* using a `CAS` operation in line 2. If that `CAS` succeeds, then the backpack was empty. Otherwise, the backpack contained a set of pairs of process IDs and sequence numbers. Process p reads that set in line 3, and then performs another `CAS` on $B[p][j]$ to close the backpack, using the value just read as the first argument. This guarantees that the `CAS` succeeds. Finally, p adds the set it found to its set stored in the local variable *bpack*.

We now discuss method `lock()`. In line 1, process p sets its set *bpack* to $\emptyset$, because it has no process backpacked. It then begins attempts of entering the critical section in an infinite while-loop, until it either succeeds or receives the abort signal and calls `abort()`. With each attempt, it increments a sequence number stored in the first component of the pair in $A[p]$ (line 5), and then it changes the second component of that pair from *done* to *want* to indicate that it is making an attempt to capture the lock (line 6). In line 7, process p makes sure that $B'[p]$ = `False`. This allows other processes to signal p that $B[p][0]$ has changed by writing `True` to $B'[p]$. In line 8 process p opens its shared backpacks $B[p][0]$ and $B[p][1]$ by changing them from *closed* to c, where c is p's current sequence number. Then, in lines 9–15 the process makes all its random decisions: It first chooses a "side" $\alpha \in \{0,1\}$ and writes to $R_\alpha[\lambda]$, where λ is a truncated geometric distribution. Then it chooses $\beta \in \{0,1\}$ at random, and if $\beta = 1$ process p writes to Z_α, otherwise p reads a pair $(carrier, carrier\_seq)$ from Z_α. In the latter case, p tries to join the backpack of process *carrier* by executing a `CAS()` operation on $B[carrier][1]$ in line 16. If that is successful, process p repeats the loop in lines 17–20 until it gets notified that it has been promoted, or that some aborting process has given p its backpack by writing it to $B[p][0]$. The signal for promotion is received on $A'[p]$; specifically, a process trying to promote p writes a pair consisting of its own ID and p's sequence number to $A'[p]$. Hence, p breaks out of the repeat-until loop (and any `await()` statement) if $A'[p] = (\cdot, c)$. Similarly, a process b that aborts, and wants to hand p its backpack, changes the value of $B[p][0]$ to b's backpack contents, and then signals p by writing `True` to $B'[p]$. This allows p to break out of its `await()` statement in line 18. After that p resets $B'[p]$, and then checks in line 20 if $B[p][0]$ has indeed received a backpack. If not, it starts another iteration of the repeat-until loop.

The reason why the second check of $B[p][0]$ is necessary, is rather subtle: An aborting process b may successfully write its backpack contents to $B[p][0]$ (see line 3), but then fall asleep just before being able to write `True` to $B'[p]$ (in line 5). Now p may abort and call `lock()` again. Then p's execution may again reach the `await()` statement in line 18, and then b wakes up and signals p that it transferred b's backpack to $B[p][0]$, even though $B[p][0]$ is still empty. As a result, p breaks out of the `await()` statement. But we cannot allow p to continue, because it is still in a backpack, and starting another attempt to win the lock might lead to a situation where the backpack relation $\leftarrow$ is cyclic. (E.g., the carrier of p might get an opportunity to join p's backpack.)

Now suppose p does not manage to join a backpack, or it finished the repeat-until loop. Then, in line 23 it first reads S, and then waits until S' is at least as large as the sequence number, i, stored in S or until $A'[p] = (\cdot, c)$. If $A'[p] = (\cdot, c)$, then p has been promoted, so it proceeds directly to call `giveUp()` in line 40. The handshaking mechanism in that method will ensure that the `giveUp()` call returns `False`, so p enters the critical section.

Now suppose that p breaks out of the `await()` statement in line 23 because $S' \geq i$. Then it is ensured that at some point after reading S, the value of S was $(i, unlocked)$ (because i is only written to S' after a process changes S from $(i, locked)$ to $(i, unlocked)$, in lines 8–9) Therefore, p now makes an attempt to lock S in line 25.

First suppose that this attempt fails. Then in line 38 p waits until S' has become larger than i (or until $A[p] = (c, critical)$). Once S' is larger than i, it is ensured that S has been unlocked again. The fact that p waits twice, once in line 23 and once in line 38 ensures that p does not finish line 38 before S has been at least once unlocked, then locked, and unlocked again. As a result p will wait long enough for a process that may have discovered p on Z_α to get a fair chance to join p's backpack.

Then, in line 40 process p calls `giveUp()`. That call returns `False` if p got promoted, and `True` otherwise. In the latter case, p calls `closeBackpacks()` to move processes from its shared backpack variables $B[p][j]$, $j \in \{0,1\}$, to its local *bpack* set, and then starts another iteration of the while-loop.

We now consider the case that p manages to lock S by changing its value from $(i, unlocked)$ to $(i+1, locked)$ in line 25. Then in lines 26–28, p changes its status $A[p]$ to $(c, critical)$, writes its current sequence number c to $X[p]$, and increments i. Updating $A[p]$ indicates that p is now poised to enter the critical section, and updating $X[p]$ indicates that p cannot be promoted anymore. The increment of i ensures that i reflects the value of the first component of S.

In the for-loop in lines 29–36, p scans through $R_{i \bmod 2}[]$ as described earlier, until it finds the first entry with value (0,0). Each pair $(r,d) \neq (0,0)$ process p finds in an array entry $R[j]$ gets added to Q, and $R[j]$ is reset to $(0,0)$ afterwards. Finally, process p enters the critical section (line 35).

In the `release()` method, process p first changes its status from $(c, critical)$ to $(c, done)$ in line 1. After that, it calls `closeBackpacks()` in line 2, where it moves all processes found in its shared backpack variables $B[p][j]$, $j \in \{0,1\}$, to the local set *bpack*. Then, in line 3, p adds the elements of that set to Q. In each iteration of the while-loop in lines 4–6, process p removes a pair (r,d) from Q, and then tries to call promote r by calling `promote(`r,d`)`. If that call returns `True`, then p successfully promoted r, and can enter the remainder section. Otherwise, it keeps attempting to promote processes until one promotion succeeds or $Q = \emptyset$. If Q becomes empty, then there are no more processes that can be promoted into the critical section. In this case, p unlocks S in line 8, and signals in line 9 that it has done so by writing to S'. Finally, p enters the remainder section.

We now describe methods `giveUp()` and `promote(`r, d`)`. A process calls `giveUp()` when it wants to terminate its current attempt to win the lock, either at the end of the while-loop of `lock()` (line 40), or when it is trying to abort (line 1). A process calls `promote(`r, d`)` to promote process r into the critical section, after it found the pair (r, d) in Q. Both methods are wait-free and guarantee that if a `promote(`r, d`)` call returns `True`, then and only then r's call of `giveUp()` while r has a sequence number of d returns `False`.

In method `giveUp()`, process p first writes its current sequence number to $X[p]$, and then reads $A'[p]$ into a pair $(proc, seq)$ (lines 1–2). If seq matches p's current sequence number, then process $proc$ is trying to promote p. In that case, in lines 4–5 process p writes to $Y[proc]$ to indicate that it has received the promotion signal, and tries to changes $A[p]$ from $(c, want)$ to $(c, critical)$. In either case, in line 7, p executes $A[p]$.`CAS(`$(c, want), (c, done)$`)`, and its `giveUp()` call returns the return value of that `CAS`. If the `CAS` fails, then either p's `CAS` in the previous line succeeded, or process $proc$ successfully promoted p by changing $A[p]$ (in line 4). Otherwise, p will not get promoted, and thus successfully gave up on its attempt to enter the critical section.

In method `promote(`r, d`)` the calling process, p, first writes `False` to $Y[p]$, and then signals r by writing (p, d) to $A'[r]$ (lines 1–2). In line 3, process p checks if $X[r] < d$; if yes, then r has not called `giveUp()` yet, so p knows that when r calls `giveUp()`, r will receive the signal on $A'[r]$ in time, and get promoted. Otherwise, p tries to change $A[r]$ from $(d, want)$ to $(d, critical)$ using a `CAS`. If that `CAS` succeeds, then again r is promoted in time (and r will find out when its `CAS` in line 7 fails). Because of that `CAS`, the check of $X[r]$ in line 3 is not necessary for correctness. But our analysis requires that the adversary does not learn the identity of r before process r has proceeded far enough in its current attempt of getting the lock (i.e., it must have at least tried to enter a backpack in line 16, if it chose $\beta = 0$ in line 12). Due to our assumption that a `CAS` reveals information to the adversary, we cannot execute this `CAS` too early; the check in line 3 ensures that r has already called `giveUp()` when the `CAS` in line 4 gets performed. If the return value of that `CAS` is `True`, then p successfully promoted r. Otherwise, p needs to check $Y[p]$ to determine if the promotion was successful. If $Y[p] =$ `True`, then r wrote to line 4 after p reset Y in line 1. Then and only then r got promoted due to its own successful `CAS` in line 5.

We finally describe the `abort()` method that process p calls after it receives the abort signal during its `lock()` method, and p's execution has reached an `await()` statement. First, p calls `giveUp()` in line 1, and the return value indicates if p already got promoted. If yes, p enters the critical section. Otherwise, p moves the contents of $B[p][j]$, $j \in \{0, 1\}$, into $bpack$ by calling `closeBackpacks()` in line 1. Then, in the while-loop in lines 1–5 process p tries to hand its backpack $bpack$ over to one of the processes in that backpack. To do so, it removes a pair (r, d) from $bpack$ and tries to change $B[r][0]$ to $bpack$ using a `CAS`. It keeps repeating this, as long the `CAS` fails and as long as $bpack \neq \emptyset$. (If such a `CAS` fails, then this means that r has already aborted, and thus is not in p's backpack anymore.) If $bpack$ becomes empty, then p can enter the critical section, because no process is waiting in p's backpack anymore. Otherwise, in one of the while-loop iterations p successfully hands its backpack to some process r. It can then reset $bpack$ to $\emptyset$, and signal r by writing `True` to $B'[r]$. This enables r to break out of the repeat-until loop (lines 17–20) in `lock()`, because r has been removed from p's backpack, so it will not get promoted.

REFERENCES

[1] Zahra Aghazadeh and Philipp Woelfel. 2016. Upper Bounds for Boundless Tagging with Bounded Objects. In *Proc. of 30th DISC*. 442–457. https://doi.org/10.1007/978-3-662-53426-7_32

[2] Dan Alistarh and James Aspnes. 2011. Sub-Logarithmic Test-And-Set Against a Weak Adversary. In *Proc. of 25th DISC*. 97–109.

[3] James Anderson and Yong-Jik Kim. 2000. Adaptive Mutual Exclusion with Local Spinning. In *Proc. of 14th DISC*. 29–43.

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

[5] J. Anderson, Y.-J. Kim, and T. Herman. 2003. Shared-memory Mutual Exclusion: Major Research Trends Since 1986. *Distr. Comp.* 16 (2003), 75–110.

[6] T. Anderson. 1990. The Performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 1 (1990), 6–16.

[7] James Aspnes. 2012. Faster randomized consensus with an oblivious adversary. In *Proc. of 31st PODC*. 1–8.

[8] Hagit Attiya, Danny Hendler, and Smadar Levy. 2013. An $O(1)$-barriers optimal RMRs mutual exclusion algorithm. In *Proc. of 31st PODC*. 220–229.

[9] Hagit Attiya, Danny Hendler, and Philipp Woelfel. 2008. Tight RMR Lower Bounds for Mutual Exclusion and Other Problems. In *Proc. of 40th ACM STOC*. 217–226.

[10] Hagit Attiya, Danny Hendler, and Philipp Woelfel. 2015. Trading Fences with RMRs and Separating Memory Models. In *Proc. of 34th PODC*. 173–182. https://doi.org/10.1145/2767386.2767427

[11] Ohad Ben-Baruch and Danny Hendler. 2015. The Price of being Adaptive. In *Proc. of 34th PODC*. 183–192. https://doi.org/10.1145/2767386.2767428

[12] Michael Bender and Seth Gilbert. 2011. Mutual Exclusion with $O(\log^2 \log n)$ Amortized Work. In *Proc. of 52nd FOCS*. 728–737.

[13] D. Culler, J.P. Singh, and A. Gupta. 1998. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann.

[14] Robert Danek and Wojciech Golab. 2010. Closing the Complexity Gap between FCFS Mutual Exclusion and Mutual Exclusion. *Distributed Computing* 23, 2 (2010), 87–111.

[15] Robert Danek and Hyonho Lee. 2008. Brief Announcement: Local-Spin Algorithms for Abortable Mutual Exclusion and Related Problems. In *Proc. of 22nd DISC*. 512–513.

[16] Oksana Denysyuk and Philipp Woelfel. 2016. Are Shared Objects Composable under an Oblivious Adversary?. In *Proc. of 35th PODC*. 335–344.

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

[18] George Giakkoupis and Philipp Woelfel. 2012. On the time and space complexity of randomized test-and-set. In *Proc. of 31st PODC*. 19–28.

[19] George Giakkoupis and Philipp Woelfel. 2012. A tight RMR lower bound for randomized mutual exclusion. In *Proc. of 44th ACM STOC*. 983–1002.

[20] George Giakkoupis and Philipp Woelfel. 2014. Randomized Mutual Exclusion with Constant Amortized RMR Complexity on the DSM. In *Proc. of 55th FOCS*. 504–513. https://doi.org/10.1109/FOCS.2014.60

[21] Wojciech Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. 2012. RMR-efficient implementations of comparison primitives using read and write operations. *Distr. Comp.* 25, 2 (2012), 109–162.

[22] Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable Implementations Do Not Suffice for Randomized Distributed Computation. In *Proc. of 43rd ACM STOC*. 373–382.

[23] Danny Hendler and Philipp Woelfel. 2010. Adaptive Randomized Mutual Exclusion in Sub-Logarithmic Expected Time. In *Proc. of 29th PODC*. 141–150.

[24] Danny Hendler and Philipp Woelfel. 2011. Randomized Mutual Exclusion with Sub-Logarithmic RMR-Complexity. *Distr. Comp.* 24, 1 (2011), 3–19.

[25] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufman.

[26] Prasad Jayanti. 2003. Adaptive and efficient abortable mutual exclusion. In *Proc. of 22nd PODC*. 295–304. https://doi.org/10.1145/872035.872079

[27] Prasad Jayanti, Srdjan Petrovic, and Neha Narula. 2005. Read/Write Based Fast-Path Transformation for FCFS Mutual Exclusion. In *Proc. of 31st SOFSEM*. 209–218.

[28] Y.-J. Kim and J. Anderson. 2001. A Time Complexity Bound for Adaptive Mutual Exclusion. In *Proc. of 15th DISC*. 1–15.

[29] Yong-Jik Kim and James Anderson. 2006. Nonatomic Mutual Exclusion with Local Spinning. *Distr. Comp.* 19, 1 (2006), 19–61.

[30] Hyonho Lee. 2005. Transformations of Mutual Exclusion Algorithms from the Cache-Coherent Model to the Distributed Shared Memory Model. In *Proc. of 25th ICDCS*. 261–270.

[31] Hyonho Lee. 2010. Fast Local-Spin Abortable Mutual Exclusion with Bounded Space. In *Proc. of 14th OPODIS*. 364–379.
[32] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.* 9, 1 (1991), 21–65.
[33] Abhijeet Pareek and Philipp Woelfel. 2012. RMR-Efficient Randomized Abortable Mutual Exclusion. In *Proc. of 26th DISC*. 267–281.
[34] Michel Raynal. 1986. *Algorithms for Mutual Exclusion.* MIT Press.
[35] Michael L. Scott. 2002. Non-blocking timeout in scalable queue-based spin locks. In *Proc. of 21st PODC*. 31–40. https://doi.org/10.1145/571825.571830
[36] Jae-Heon Yang and James Anderson. 1995. A Fast, Scalable Mutual Exclusion Algorithm. *Distr. Comp.* 9, 1 (1995), 51–60.