Distrib. Comput. (2011) 24:3–19
DOI 10.1007/s00446-011-0128-6

Randomized mutual exclusion with sub-logarithmic RMR-complexity

Danny Hendler · Philipp Woelfel

Received: 20 May 2010 / Accepted: 28 November 2010 / Published online: 21 April 2011

Abstract *Mutual exclusion* is a fundamental distributed coordination problem. Shared-memory mutual exclusion research focuses on *local-spin* algorithms and uses the *remote memory references* (RMRs) metric. Attiya, Hendler, and Woelfel (40th STOC, 2008) established an $\Omega(\log N)$ lower bound on the number of RMRs incurred by processes as they enter and exit the critical section, where N is the number of processes in the system. This matches the upper bound of Yang and Anderson (Distrib. Comput. 9(1):51–60, 1995). The upper and lower bounds apply for algorithms that only use read and write operations. The lower bound of Attiya et al., however, only holds for deterministic algorithms. The question of whether randomized mutual exclusion algorithms, using reads and writes only, can achieve sub-logarithmic expected RMR complexity remained open. We answer this question in the affirmative by presenting starvation-free randomized mutual exclusion algorithms for the cache coherent (CC) and the distributed shared memory (DSM) model that have sub-logarithmic expected RMR complexity against the strong adversary. More specifically, each process incurs an expected number of $O(\log N / \log\log N)$ RMRs per passage through the entry and exit sections, while in the worst case the number of RMRs is $O(\log N)$.

P. Woelfel was supported by NSERC.

D. Hendler
Ben-Gurion University, Beersheba, Israel
e-mail: hendlerd@cs.bgu.ac.il

P. Woelfel (✉)
University of Calgary, Calgary, AB, Canada
e-mail: woelfel@ucalgary.ca

Keywords Mutual exclusion · Remote memory references · RMRs · Strong adversary · Randomization

1 Introduction

In the *mutual exclusion* problem, a set of processes must coordinate their access to a *critical section* so that, at any point in time, at most a single process is inside the critical section. Introduced by Dijkstra in 1965 [10], the mutual exclusion problem is at the core of Distributed Computing and is still the focus of intense research [2,18]. In this paper, we consider mutual exclusion in the asynchronous shared-memory model.

A natural way to measure the time complexity of algorithms in this model is to count the number of shared-memory accesses performed by processes. This measure is problematic for mutual exclusion implementations because, in this case, a process may perform an unbounded number of memory accesses while busy-waiting for another process to leave the critical section [1]. Instead, we can measure the time complexity of an algorithm by counting only *remote memory references* (RMRs), i.e., memory accesses that traverse the processor-to-memory interconnect. *Local-spin* algorithms, which perform busy-waiting by repeatedly reading *locally accessible* shared variables, achieve bounded RMR complexity and have practical performance benefits [6]. Indeed, recent mutual exclusion research has almost entirely focused on the RMR complexity of local-spin algorithms (see [4,5,8,9,14–17] for some examples).

Yang and Anderson presented the first $O(\log N)$ RMRs read/write mutual exclusion algorithm [19]. Anderson and

Kim [3] conjectured that this was best possible. This conjecture was recently proved by Attiya, Hendler, and Woelfel [8]. Their lower bound, however, holds only for deterministic algorithms since it assumes that a scheduling adversary knows processes' future steps. The question of whether randomization can help break the logarithmic barrier of [8] thus remained open. In this paper, we provide a positive answer to this question.

In order to reason about randomized distributed algorithms, one has to specify an *adversary* that models how the system reacts to random choices made by processes. Typical models are the *oblivious*, the *weak*, and the *strong* adversary. An *oblivious* adversary has to make all scheduling decisions in advance, before any process has flipped a coin. This model corresponds to a system, where the coin flips made by processes have no influence on the scheduling. A more realistic model is the *strong* adversary, who sees every coin flip as it appears, and can use that knowledge for any future scheduling decisions. There are several variants of *weak* adversaries, but generally, a weak adversary sees the coin flip of a process not before that process has taken a step following that coin flip. Our work focuses on the strong adversary; a more detailed description follows in Section 1.1.

With the single exception of [16], prior art local-spin mutual exclusion research dealt exclusively with deterministic algorithms. Anderson and Kim presented a simple randomized variant [16] of their (deterministic) read/write adaptive mutual exclusion algorithm [4]. In their algorithm, if point contention is at most k, then each process incurs only an expected number of $O(\min\{\log k, \log N\})$ RMRs per passage. This contrasts the author's deterministic lower bound of $\Omega(k)$. The adversary model is not discussed in the paper, but from all we know the upper bound on the RMR complexity only holds for the oblivious adversary. We know of no other algorithms or lower bound results that affect the RMR complexity of mutual exclusion or other problems.

Golab, Hadzilacos, Hendler, and Woelfel [13] (see also [11]) presented a constant-RMRs read/write implementation of *compare-and-swap*. Moreover, they proved that any deterministic shared-memory algorithm using reads, writes, and conditional operations (such as *compare-and-swap*), can be simulated by a deterministic read/write algorithm, with only a constant increase in the RMR complexity. The algorithms we present use variables that support the *compare-and-swap* and read operations. It follows from [13] and [11], that these variables can be implemented from only reads and writes while maintaining asymptotic worst-case RMR complexities. Although *expected* RMR complexity is not necessarily preserved if one replaces atomic operations with linearizable ones (see [12] for a discussion of these issues), any linearizable CAS implementation with $O(1)$ RMR complexity can be used in our algorithm.

1.1 Model

Our computation model is an asynchronous shared memory system where N *processes* communicate by executing *operations* on shared *variables* (or *registers*). Each process is a sequential execution path that performs a sequence of *steps*, each of which invokes a single operation on a shared variable.

The *compare-and-swap* operation (abbreviated CAS) is defined as follows: $v.\mathtt{CAS}(expected, new)$ changes the value of register v to new only if its value just before CAS is applied is $expected$; in this case, the CAS operation returns **true** and we say it is *successful*. Otherwise, CAS does not change the value of v and returns **false**; in this case, we say that the CAS was *unsuccessful*.

We consider a shared memory systems that supports atomic read-, write-, and CAS operations. In addition, a process p can generate random values by executing an operation that assigns one of p's local variables an integer chosen uniformly at random from an arbitrary finite set of integers.

The scheduling, generated by the *adversary*, can depend on the random values generated by the processes. We assume a strong adversary model (see for example [7]). At each point in time t, the adversary decides which process is taking the next step. In order to make this decision, it can take all preceding events into account, including previously generated coin flips. Note that generating a random value x counts as a step for the process that generates x. Hence, the adversary sees the value of x before p can take the next step, and thus it can decide whether p or some other process takes the next step, immediately after x has been generated.

We consider the *cache-coherent* (CC) and the *distributed shared memory* (DSM) models. In the DSM model, each processor has its own segment of shared memory registers, and the shared memory consists of segments of all processors. Any access to the segment of process p by process q incurs a *remote memory reference* (RMR). In the CC model, each processor maintains local copies of shared variables it accesses inside its cache, whose consistency is ensured by a coherence protocol. At any given time a variable is remote to a processor if the corresponding cache does not contain an up-to-date copy of the variable. We assume that a write operation on a variable invalidates all cache-copies, and even the writing processor has no valid cache-copy after that write operation. A memory access to a remote variable is called a *remote memory reference* (RMR). More precisely, any write-operation incurs an RMR, and any read-operation of register r by process p incurs an RMR, if either p has not read r before, or if some process has written r since p's preceding read of r. We also assume that all CAS operations incur RMRs. Note that these assumptions are conservative, in the sense that our upper bounds hold for all reasonable cache-coherent models, while on some models (e.g., with write-back caches) a better RMR complexity may actually be achieved.

1.2 Results

First, in Sects. 2–4 we present and analyze a starvation-free randomized mutual exclusion algorithm for the CC model, in which processes incur at most $O(\log N)$ RMRs and an expected number of $O(\log N / \log\log N)$ RMRs per passage in every execution, even against a strong-adversary that can schedule processes according to their execution history. Our algorithm uses read-, write-, and CAS operations. Replacing CAS operations by the implementations given in [11,13], we obtain an algorithm with the same asymptotic RMR complexity for shared memory systems that support only atomic read- and write-operations. In Section 5, we show how the algorithm can be modified for the DSM model, so that it exhibits the same RMR complexity.

Our results are summarized by the following theorem.

Theorem 1 *There exist randomized starvation-free mutual exclusion algorithms for the CC and the DSM model, where against the the strong adversary each process incurs an expected number of $O(\log N / \log\log N)$ RMRs and at most $O(\log N)$ RMRs per passage through the entry, critical and exit section.*

The result holds vacuously, if we replace atomic CAS objects with linearizable implementations, where each CAS operation requires at most $O(1)$ RMRs, such as the one from [13] and [11] (we discuss this in more detail at the end of Sect. 4.4). This establishes a separation in terms of RMR complexity between deterministic read/write mutual exclusion algorithms and randomized ones (in the strong adversary model). Since the lower bound of $\Omega(\log N)$ RMRs per passage [8] holds also for algorithms that can use conditional operations in addition to reads and writes, this separation applies for such algorithms, too.

2 High-level description of the algorithm

In this section, we provide a high-level overview of our randomized mutual-exclusion algorithm. For the rest of this paper, the number of processes sharing the implementation is N, and the processes have distinct IDs in $\{1, \ldots, N\}$. For the ease of description we assume (w.l.o.g.) that $N = \Delta^{\Delta-1}$ for some positive integer Δ. Note that $\Delta = \Theta(\log N / \log\log N)$.

The arbitration tree. As in previous mutual exclusion algorithms, we use an arbitration tree that processes climb up in order to enter the critical section. But contrary to other algorithms, our arbitration tree in not binary. Instead, it is a complete Δ-ary tree of height Δ and with N leaves. We say that a node is at *level* i if its height is i, where the root has height one. Each inner tree node is represented by a structure Node that includes a lock field accessed by read and CAS operations. Each process p is associated with a unique leaf leaf$[p]$ in the tree.

A process p tries to enter the critical section by climbing up the path from leaf leaf$[p]$ to the root, and capturing the lock of each node on the way. In order to capture the lock of node v, a process p has to successfully change the value of v.lock from $\perp$ to its ID, using a CAS operation. If that CAS operation fails, p starts to spin (busy-wait) on v.lock, waiting for the lock to become free again. Once v.lock $= \perp$, the process performs another CAS operation trying to capture the lock, and so on. If p manages to capture all locks on the path from leaf$[p]$ to the root, it can safely enter the critical section (clearly only one process can own the root-lock). In the exit section, p releases all these locks again. Note that if there were no contention, then a process would incur only one RMR to capture a lock and thus the total number of RMRs per passage would be $O(\Delta) = O(\log N / \log\log N)$.

Randomized promotion. While one attempt to capture the lock at node v may cost only one RMR, every time the process p attempts to capture the lock, some other process may succeed before p. We therefore add a randomized mechanism that gives p a chance of $1/\Delta$ to enter the critical section immediately, when it fails to capture the lock C times, where C is a sufficiently large constant. Before trying to capture the lock at node v, process p writes its ID in a unique position in an array apply$[0 \ldots \Delta - 1]$ (we say p *applies for promotion*). That unique position is determined by the rank of the child of v from which p ascended to v. Each time some process q releases the lock v.lock, it first randomly chooses an index $i \in \{0, \ldots, \Delta - 1\}$ and checks the array position apply$[i]$. If it finds a process ID there, say p, then it tries to *promote* that process (that requires some handshaking with process p and may fail). But if that promotion is successful, q adds p to a *promotion* queue promQ (access to the queue is guaranteed to be mutually exclusive, so a sequential queue implementation suffices). On the other hand, the promoted process, p, can start spinning on some local variable, waiting to become the head of the promotion queue. Just before the process q that promoted p finishes its exit section, it notifies the head of the promotion queue, p', that p' is now allowed to enter the critical section. In addition, it *hands over* the root lock to p', so that the process entering the critical section always owns that lock. When p' is about to finish its exit section, it will notify the next process in the promotion queue (if there is one), and so on. This way, all processes in the promotion queue will be allowed to enter the critical section, and once a process has been promoted, it performs only a constant number of RMRs until it enters the critical section.

As already mentioned, for roughly every C attempts to capture the lock of a node, $C = O(1)$, a process has one chance of being promoted, and the probability that it

succeeds being promoted is $1/\Delta$. Hence, the number of attempts to capture node locks is approximately geometrically distributed and the expected number of such attempts is $O(\Delta)$. Together with the fact that a process needs to capture only Δ locks in order to enter the critical section (because Δ is the height of the tree), this leads to an expected RMR complexity of $O(\Delta)$.

Deterministic promotion and starvation freedom. It is possible that each time process p has a chance of being promoted, the promoting process makes the "wrong" random choice, thus it does not promote p. On the other hand, p might repeatedly fail to capture the lock $v.\mathsf{lock}$, because some other process (possibly always the same one) manages to capture it before p. Therefore, the algorithm described so far is not starvation free in a strong (worst-case) sense. More precisely, for any T, the probability that a process has not entered the critical section after every process has taken T steps is larger than 0 (although it converges to 0 with increasing T). For the same reason we cannot bound the worst-case RMR complexity of the algorithm just described.

In order to obtain a bound on the worst-case RMR complexity (which would also yield starvation freedom), processes could switch to a different worst-case efficient algorithm once they have performed too many RMRs. But this causes two potential problems: First of all it would require that a process can at any point estimate the number of RMRs it has incurred. But while, for example, process p busy-waits for the value of $v.\mathsf{lock}$ to switch from q back to $\perp$, some other process q may release the lock (change its value to $\perp$), and then re-capture it again before p even notices that the lock had been released. Then p's next read of $v.\mathsf{lock}$ incurs an RMR, but p is unaware of that. Moreover, even if p could keep track of RMRs, it would still have to be able to terminate its attempt to enter the critical section in the randomized algorithm, in order to be able to switch to the deterministic one.

We fix these problems using a deterministic promotion mechanism: With each node v in the arbitration tree, we maintain a sequential modulo-Δ counter $v.\mathsf{token}$ (which will only be increased in mutual exclusion). Each time a process q releases a lock, it tries to find a process registered in the array entry $\mathsf{apply}[j]$ in order to promote it, in two ways: It chooses j at random as described above. But then it does the same for $j = v.\mathsf{token}$. After that it increments $v.\mathsf{token}$ (modulo Δ). This way, while process p is busy-waiting for $v.\mathsf{lock}$ to be released, if Δ promotions on that node happen, p will not be left out and thus is guaranteed to be promoted. Moreover, by monitoring $v.\mathsf{token}$ while also spinning on $v.\mathsf{lock}$, p can roughly keep track of the number of RMRs it incurs on $v.\mathsf{lock}$, because in any time interval in which $v.\mathsf{lock}$ changes often enough (but only a constant number of times), some process must capture and release this lock, and thus increase $v.\mathsf{token}$.

Bounding the worst-case RMR complexity. With the technique from above, a process might in the worst-case still incur $\Omega(\Delta)$ RMRs on each level of the arbitration tree and thus $\Omega(\Delta^2) = \Omega\left((\log N/\log\log N)^2\right)$ RMRs in total. This is worse than the $O(\log N)$ RMR complexity of deterministic mutual exclusion algorithms such as the ones by Yang and Anderson [19] and Anderson and Kim [3]. In order to bound the total number of RMRs a process incurs on each level in the worst-case, we allow it to change its tactic to win a lock of a node, if it has already made too many attempts. We add a deterministic starvation-free Δ-process mutual exclusion object $v.\mathsf{MX}$ to each node v, that can be used by threads with IDs in $\{0,\ldots,\Delta-1\}$. A process can call $\texttt{GetLock}_i\texttt{()}$, using thread ID $i \in \{0,\ldots,\Delta-1\}$, in order to enter the critical section and it can exit the critical section using $\texttt{RelLock}_i\texttt{()}$. (Both methods have worst-case RMR complexity $O(\log\Delta)$ using e.g., the mutual exclusion implementation in [3].) An additional method `LockOwner()` returns the ID of the thread that is currently in the critical section (if any).

The idea is now the following: A process p that is trying to win the lock of a node v keeps track of the number of attempts it has made to capture that lock. If this number exceeds $\lceil\log\Delta\rceil$, then p calls $v.\mathsf{MX}.\texttt{GetLock}_i\texttt{()}$, where i is the rank of the unique child from which p ascended to v. Once it has received the lock, it makes only two more attempts to capture $v.\mathsf{lock}$. On the other hand, a process that is releasing the lock $v.\mathsf{lock}$ tries to promote one additional process: It uses a method-call $v.\mathsf{MX}.\texttt{LockOwner()}$ to determine whether some process is in the critical section of $v.\mathsf{MX}$, and then tries to promote that process in the same way as it promotes other processes randomly and deterministically as described above. This way we can guarantee that when p has finished its entry section in $v.\mathsf{MX}$, it will either manage to capture $v.\mathsf{lock}$ within the next two attempts, or it will be promoted.

With this strategy, the worst-case number of RMRs needed to capture the lock of a node or to be promoted while trying to capture that lock is bounded by $O(\log\Delta)$. Since a process needs to capture only Δ nodes in order to enter the critical section, we obtain a worst-case RMR-complexity of $O(\Delta\log\Delta) = O(\log N)$.

3 Implementation

We now describe the implementation of our algorithm in detail. See Fig. 1 for the main mutual exclusion algorithm, and Figs. 2 and 3 for the entry and exit sections, respectively.

Fig. 1 The randomized mutual exclusion algorithm

Algorithm 1: RandomMutEx$_p$

```
1  define Node: struct { ;
2       lock: int init ⊥,;
3       MX: Starvation-free Δ-process mutual exclusion object;
4       apply: array [0 ... Δ − 1] of int init ⊥,;
5       token: int init 0,;
6  } ;
7  shared: ;
8       root: Node ;                                  /* root of the arbitration tree */
9       leaf: array [0 ... N − 1] of type Node ;      /* leaf[i] is the i-th leaf in */
   ;                                                      /* the arbitration tree */
10      promQ: Queue init ∅;
11      notified: array [0 ... N − 1] of type boolean init false ;
12 local: ;
13      v: Node;
14      i, j, j′, q, tok, ctr: int ;
15 Entry_p();
16 // Critical Section;
17 Exit_p();
```

Fig. 2 The entry section

Function Entry$_p$

```
18 notified[p] := false;
19 v := leaf[p];
20 repeat
21   Let i be the integer such that v is the (i + 1)-th child of parent(v);
22   v :=parent(v);
23   v.apply[i].CAS(⊥, p);
24   ctr := 0;
25   repeat
26     ctr := ctr + 1;
27     if ctr > ⌈log Δ⌉ then
28       if v.apply[i].CAS(p, ⊥) then
29         v.MX.GetLock_i();
30         v.apply[i].CAS(⊥, p);
31         await (v.lock = ⊥ ∨ v.apply[i] ≠ p);
32       end
33     end
34     if ¬ v.lock.CAS(⊥, p) then
35       tok := v.token;
36       await (v.token ≠ tok ∨ v.apply[i] ≠ p ∨ v.lock = ⊥) ;
37     end
38     if v.MX.LockOwner() = i then v.MX.RelLock_i();
39   until v.apply[i] ≠ p ∨ v.lock = p;
40   if ¬ v.apply[i].CAS(p, ⊥) then
41     await (notified[p] = true) ;
42   end
43 until notified[p] ∨ v = root;
```

Shared data. The algorithm uses a structure of type Node to represent the nodes of the arbitration tree (see lines 1–6). Each Node object has a CAS-able field lock that processes use to lock the node. The field MX is a Δ-process mutual exclusion object that provides the functions `GetLock`$_i$`()`, `RelLock`$_i$`()`, and `LockOwner()`. The methods `GetLock`$_i$`()` and `RelLock`$_i$`()` can be accessed by a process using thread ID $i \in \{0, \ldots, \Delta - 1\}$ in order to enter and exit the critical section. In addition, we add a method `LockOwner()` that returns the ID of the process in the critical section, if there is one, or $\perp$, if there is no such process. It is obvious how to implement these methods using a mutual exclusion algorithm, as for example the one proposed by Yang and Anderson [19] or by Anderson and Kim [3] such that `GetLock`$_i$`()` and `RelLock`$_i$`()` have worst-case RMR-complexity $O(\log \Delta)$, and `LockOwner()` has constant RMR-complexity. In addition, the Node object has as fields an array apply of Δ CAS-able registers that processes use to "apply for promotion", and an integer token that is used as a pointer in that array.

The arbitration tree is given by a statically allocated set of shared Node objects. The root is stored in the shared Node object root, and the N leaves are stored in an array leaf$[0 \ldots N-1]$. For a Node v, The function `parent(`v`)` returns the parent of v, or $\perp$ if $v =$ root.

Processes have sequential access to a queue promQ to keep track of promoted processes. Since the queue implementation is not concurrent, the queue operations promQ.`Enq()`

Fig. 3 The exit section

```
Function Exit_p
44 foreach node v on the path from leaf[p] to the root, where v.lock = p do
45 |  tok := v.token;
46 |  i := v.MX.LockOwner();
47 |  Pick j' uniformly at random from {0, ..., Δ − 1};
48 |  for j ∈ {j', tok, i} − {⊥} do
49 |  |  q := v.apply[j];
50 |  |  if q ≠ ⊥ ∧ v.apply[j].CAS(q, ⊥) then
51 |  |  |  promQ.Enq(q);
52 |  |  end
53 |  end
54 |  v.token := (tok + 1) mod Δ;
55 |  if v ≠ root then v.lock.CAS(p, ⊥);
56 end
57 if promQ = ∅ then
58 |  root.lock.CAS(p, ⊥);
59 else
60 |  q := promQ.Deq();
61 |  root.lock.CAS(p, q);
62 |  notified[q] := true
63 end
```

and promQ.Deq() can easily be implemented with $O(1)$ step-complexity. Finally, a shared array notified of CAS-able registers allows us to implement a notification mechanism for promoted processes.

The entry section. The entry section of process p is described by the function $\texttt{Entry}_p\texttt{()}$ (lines 18–43). Process p begins in line 18 by resetting the array entry notified[p], which can then later be used by some other process to notify p of its promotion. It then determines the leaf $v = \mathsf{leaf}[p]$, from where it starts its ascent to the root (line 19).

In the ℓ-th iteration of the *outer entry loop* (lines 20–43), process p tries to capture the lock of the $(\ell+1)$-th vertex on the path from leaf leaf[p] to the root. Process p first determines the rank i of its current node v among all the children of the parent of v, and then ascends to that parent (lines 21 and 22). It then *applies for promotion* at node v by writing its ID to the entry of v.apply[i] (line 23), where i is the rank of the child (among all children of v) from which p ascended to v. (Although, here it would be fine to update v.apply[i] with a write operation, we use a CAS operation instead in order to allow the algorithm to work even with non-writable CAS objects.) Process p then executes the *inner entry loop*, lines 25–39, until it either captures the lock of node v, or gets promoted. The counter ctr (initialized to 0 in line 24) is used to keep track of the number of iterations of the inner entry loop that were performed. In each iteration of the inner entry loop, p first increases that counter and then checks whether it has executed too many (i.e., $\lceil \log \Delta \rceil$) iterations of the inner entry loop (lines 26 and 27).

First assume that this is not the case. Then p proceeds to line 34, where it tries to capture v's lock by switching the value of v.lock from $\perp$ to p, using a CAS operation. If the CAS operation fails, p reads the current value of v.token into tok, and then starts to busy-wait until one of the values of v.token or v.apply[i] changes, or until the lock v.lock gets released (lines 35–36). The value of v.apply[i] can only change if some other process, say q, *promotes* p by resetting the value of apply[i] back to $\perp$; in this case p can finish its inner entry loop. If v.lock gets released, then p should start a new iteration of the inner entry loop in which it can try to capture the lock again. A change of v.token also indicates that p should start another iteration: The reason is that this way the value of p's local variable ctr will reflect (asymptotically) the number of RMRs p has incurred. Once one of these values changes, or if p managed to capture the lock of node v, p will proceed with line 39, where the inner entry loop ends. (The if-statement in line 38 will evaluate to **false**.) If p owns v.lock at this point, or if p has been promoted, it can terminate its inner entry loop; otherwise it repeats it.

Now consider the case that $ctr > \lceil \log \Delta \rceil$ when p executes line 27. In this case, p has not succeeded in capturing the lock of node v sufficiently many times. We say p becomes *desperate*. Process p proceeds to line 28 and tries to *withdraw its application for promotion*, by swapping the value of apply[i] back to $\perp$ using a CAS operation. This operation may fail, if some other process has promoted p in the meantime. In that case, p can continue its inner entry loop as usual (but it will not attempt another iteration). Otherwise, p proceeds to line 29, where it participates in a Δ-process mutual exclusion algorithm on object v.MX, using thread ID i. Once p is in the critical section of that object, it can re-apply for promotion (line 30). (It is necessary to withdraw the application in the first place in order to avoid situations, in which a process in the critical section of v.MX and another processes that gets promoted while in the entry section of v.MX block each other.) Then, in line 31, process p waits until either v.lock is free or until it has been promoted. Once

one of these two events occur, p continues to try capturing the lock in line 34, as described previously. The main idea is that if some process q releases the lock v.lock while p is in the critical section of v.MX, then q will promote p. Hence, if p is desperate and fails to capture the lock in line 34, then it will be guaranteed that by the time p has finished waiting in line 36, it will have been promoted.

Finally, we discuss the remaining part of the outer entry loop, lines 40–43. If p's execution reaches this part of the code, then it either has been promoted or it has captured the lock of node v. Process p tries to withdraw its application for promotion in line 40, and, if that succeeds, it knows that it hasn't been promoted. If $v = \mathsf{root}$, then p has captured the root-lock and it can finish the outer entry loop in line 43. Otherwise, it starts its next iteration of that loop, continuing its ascent in the arbitration tree. If the CAS operation in line 40 fails, however, then p has been promoted. It will be guaranteed that p has been added to promQ. The process preceding p in the queue (or, in case p is enqueued to the front, the process enqueuing p), will notify p in the course of performing its exit section by changing notified[p] to **true**. Therefore, in line 41 p waits until the register notified[p] is set. Once this happens, p can enter the critical section.

The exit section. A process p in the exit section fulfills several tasks:

1. Process p releases all the node-locks it has captured.
2. Before p releases the lock of a node v, it tries to promote (randomly and deterministically) some of the processes that may have incurred RMRs while trying to capture v.lock.
3. If there is a process in the promotion queue, promQ, then p notifies the frontmost process, q, to enter the critical section. After that, p hands over the root-lock to q to ensure that this is always owned by the process in the critical section.

In the for-loop (lines 44–56), process p considers each node on the path from leaf[p] to the root, for which it holds a lock. For each such node v, in lines 45–46 p reads the current value of v.token and the thread ID of the process in the critical section of v.MX, if there is any (otherwise v.MX.`LockOwner()` returns $\perp$). It then samples a value uniformly at random from $\{0, \ldots, \Delta - 1\}$ (line 47). In the for-loop of lines 48–53, p then tries to promote the process that has applied for promotion by writing its ID to apply[j], where j takes the value of the token, the value of the process in the critical section of v.MX, and the randomly chosen value.

Such a promotion attempt is executed by first reading the ID q of the promotion applicant from apply[j] (line 49). If that value is not $\perp$, p tries to swap the value back to $\perp$ using the CAS operation in line 50. If that fails, the applicant q has already withdrawn its application (possibly it won the lock of the node v in the meantime) and so p's attempt to promote the process fails. Otherwise, p executes line 51 and enqueues process q into the promotion queue promQ. The code of the entry section ensures that when q tries to withdraw its application for promotion the next time, q will notice that it has been promoted.

After the (successful or unsuccessful) promotion attempts, p increases the value of token modulo Δ (line 54). Finally, in line 55, it releases the lock of node v, unless v is the root of the arbitration tree.

It cannot release the root-lock, yet, because it has to make sure that processes in the promotion queue are allowed to enter the critical section, first. Therefore, p first checks whether the promotion queue is empty, and only if that's the case, it releases the root-lock (lines 57–58). If the queue is not empty, then p removes the first process q from promQ (line 60), and *hands over* the root-lock to q (line 61). Then, in line 62, it notifies q so that q can enter the critical section. Note that the mechanism of handing over the root-lock ensures that a process entering the critical section is always the owner of that lock, and mutual exclusion follows from this property.

4 Analysis and correctness

In this section, we formally prove correctness (safety and starvation freedom) of our algorithm for the CC model, and analyze its RMR complexity. We start by defining some notation and terminology.

The repeat-until loop starting at line 20 and ending at line 43 is called *outer entry loop*, and the one starting at line 25 and ending at line 39 is called *inner entry loop*.

We say process p is *registered* at node u, whenever it executes the function `Entry`$_p$`()` and its local variable v has value u. The process owns the lock of node u, when u.lock $= p$. It can *capture* the lock only by successfully executing the CAS operation in line 34 while registered at node u. It can cease to own the lock either by *releasing* it or by *handing it over* to some other process. The process can release it by executing the CAS operation in line 55 while the variable v has value u, or in line 58 (if $u = \mathsf{root}$). It can hand it over to some other process q, by executing the CAS operation in line 61.

A process p can *apply for promotion at a node u* by setting a value u.apply[i] to p. It can *withdraw* that application by resetting the value u.apply[i] to $\perp$. We say that process p gets *promoted*, when some other process $r \neq p$ changes the value of u.apply[i] from p to $\perp$ for some node u and some integer i (this can only happen when r sucessfully executes the CAS operation in line 50). We say that a process r *notifies*

p of its promotion, when r changes the value of notified[p] to **true** (this can only happen when r executes line 62). A process p is *desperate* whenever its local variable ctr has a value larger than $\lceil \log \Delta \rceil$.

4.1 Structural properties

We first prove some properties that describe the state of the shared and local variables of a process in certain situations. The following two claims state that in each iteration of the inner entry loop, a process ascends one level in the tree, and describe the application mechanism that processes follow.

Claim 1 Suppose process p has executed line 22 k times during one call of $\texttt{Entry}_p()$ that has not yet terminated. Then its local variable v is the $(k+1)$-th node on the path from leaf[p] to the root.

Proof In the function $\texttt{Entry}_p()$, the value of v is only modified in line 19 and line 22. Thus, the claim follows immediately from the structure of the outer entry loop. □

Claim 2 When u.apply[j] $= q \neq \perp$ for some integer j and some node u, then

(a) q is the last process that changed the value of u.apply[j],
(b) q is registered at node u,
(c) the next operation q will execute is in lines 24–28 or 31–40.

Proof Part (a) follows immediately from lines 23 and 30, the only places in which u.apply[j] can be changed to a non-$\perp$ value. For (c) note that when p finished executing line 28 or line 40, by the semantics of the CAS operation, u.apply[j] $\neq q$. Now (b) follows from the fact that once q has executed the CAS operation in 23 or 30 successfully, it is registered at node u and it can only cease to be registered at node u by finishing its entry section or by starting another iteration of the outer entry loop. According to (c), by that time u.apply[j] must have already changed to a non-q value. □

In the following claim, we establish some straightforward properties of the node locks.

Claim 3 (a) The owner of a node lock changes only if the node lock is captured, released, or handed over, and only node locks that have no owner can be captured.
(b) A process p can only cease to own a lock by executing a CAS operation in its exit section.
(c) During an execution of $\texttt{Exit}_p()$, p ceases to own all non-root locks it owns on the path from parent(leaf[p]) to the root.
(d) Whenever process p executes line 43, either notified[p] = **true**, or p owns v.lock.
(e) If process p is registered at node v while starting an iteration of the outer entry loop (i.e., while executing line 20), then it owns exactly the locks of all vertices on the path from parent(leaf[p]) to v.

Proof (a): It is obvious from the four CAS operations in lines 34, 55, 58, and 61, which are the only operations of the algorithm that affect the value of v.lock for any node v.
(b): Note that in order to cease to own a node-lock of vertex u a $\texttt{CAS}(p, \cdot)$-method call must be executed for the object u.lock. This can only happen in p's exit-section.
(c): For each non-root node v on the path from leaf[p] to the root for which p owns the lock, p executes line 55, which releases the lock. The root-lock is either released in line 58 or handed over to some other process in line 61.
(d): Assume notified[p] = **false** and v.lock $\neq p$ when p executes line 43. At the point when process p finished its inner entry loop either v.apply[i] $\neq p$ or it had captured the lock of v. In the latter case, by Claim 3(b) p does not release the lock until its call of $\texttt{Entry}_p()$ terminates, so v.lock $= p$ when p executes line 43.
So assume v.apply[i] $\neq p$ when process p finished the inner entry loop. By Claim 2(a) no other process can change the value of v.apply[i] to p, so v.apply[i] $\neq p$ when p executed the CAS operation in line 40. Therefore, that CAS operation failed, and thus p did not finish line 41 until notified[p] = **true**. But notified[p] cannot be reset to **false** during the entire algorithm, which contradicts the assumption.
(e): Clearly, p can only capture a lock of a node if it is registered at that node, and by Claim 1, p can only have captured the locks on the path from parent(leaf[p]) to v. By part (c), during a call of $\texttt{Exit}_p()$, the process ceases to own each lock it previously owned on that path. Therefore, it suffices to show that the claim is true for the first call of $\texttt{Entry}_p()$ by process p.

We prove that p has captured all the locks on the path, when it starts a new iteration of the outer entry loop. If $v =$ leaf[p], then p cannot have captured any locks, yet, so the claim is true. Now suppose p starts a new iteration at a point in time when $v \neq$ leaf[p]. Consider the point in time at which it executed line 43 previously. Since p started a new iteration of the outer entry loop, $v \neq$ root and notified[p] = **false** at that point. Thus, by part (d), p owned v.lock, and by part (b) it cannot have ceased to own it since. □

4.2 Mutual exclusion and correct access of objects

We now prove the safety property of our algorithm. Moreover, the part of the exit section in which the shared objects u.token and promQ are accessed can only be executed by one

process at a time. This guarantees that these objects are only accessed sequentially. Finally, we show that access to the mutual exclusion objects $u.\mathsf{MX}$ is correct.

The following statement shows that no two processes can act on the same node u while their local variables i have the same value. This is important to ensure that there is no concurrent access of $u.\mathsf{apply}[j]$, or the mutual exclusion object $u.\mathsf{MX}$ by processes using the same thread ID j.

Claim 4 Suppose two distinct processes p and q are registered at the same node u. Then their local variables i have different values.

Proof For the purpose of a contradiction assume p and q are both registered at u and that both their local variables i have value j. Let u' be the j-th child of u. If u' is a leaf, then by Claim 1, $u' = \mathsf{leaf}[p] = \mathsf{leaf}[q]$, which contradicts $p \neq q$. If u' is an inner node, then according to Claim 3(e), both processes own the lock of u'—*a* contradiction. □

The next claim establishes the safety of our algorithm.

Claim 5 At any point in time at which process p has finished its $\texttt{Entry}_p\texttt{()}$-function, and not yet executed either line 58 or line 61, it owns the root-lock.

Proof Clearly, the claim is true at the beginning of any execution. By Claim 3(b), p can only cease to own the root-lock by executing either line 58 or line 61. Hence, it suffices to show that each process owns the root-lock at the point of the response of its $\texttt{Entry}_p\texttt{()}$ call.

For the purpose of a contradiction, suppose t is the first time at which the $\texttt{Entry}_p\texttt{()}$ call of some process p terminates, but p does not own the root-lock at time t. By the semantics of line 43, if $\mathsf{notified}[p] = \textbf{false}$ at time t, then $v = \mathsf{root}$ and thus by Claim 3(d) p owns the root-lock. Hence, assume $\mathsf{notified}[p] = \textbf{true}$ at time t. Then some other process q must have set $\mathsf{notified}[p]$ to **true** at some time $t' < t$ by executing line 62. Hence, q previously executed the CAS operation in line 61 in which it handed over the root-lock to p (by the definition of t, q owned the root-lock at that point in time.) By Claim 3(b), p cannot cease to own the lock until time t. □

Clearly, only one process can own the root-lock at any point in time. Therefore, from Claim 5 we immediately obtain the following.

Corollary 1 *Algorithm 1 satisfies mutual exclusion.*

This guarantees that access to the queue promQ occurs only sequentially, and that $v.\mathsf{token}$ is in fact incremented in line 54 (modulo Δ). We summarize this in the following corollary. In addition, we establish that access to the mutual exclusion object $u.\mathsf{MX}$ is "correct".

Corollary 2 (a) *When a process writes $v.\mathsf{token}$ in line 54, it increments that register by one, modulo Δ.*

(b) *A process only accesses the queue promQ in mutual exclusion, and when it owns the root-lock.*

(c) *For each node u and each integer $j \in \{0, \ldots, \Delta - 1\}$, at any point in time there is at most one process that has called $u.\mathsf{MX}.\texttt{GetLock}_j\texttt{()}$ and not finished $u.\mathsf{MX}.\texttt{RelLock}_j\texttt{()}$.*

(d) *Calls of $u.\mathsf{MX}.\texttt{GetLock}_j\texttt{()}$ and $u.\mathsf{MX}.\texttt{RelLock}_j\texttt{()}$ occur in order.*

Proof Part (a) follows immediately from the structure of the for-loop in $\texttt{Exit}_p\texttt{()}$ together with Claim 3(b) and the fact that $u.\mathsf{token}$ can only be changed in line 54 before a process has released $u.\mathsf{lock}$. Part (b) follows immediately from Claim 5 and the structure of $\texttt{Exit}_p\texttt{()}$. For part (d) note that every process that executes $u.\mathsf{MX}.\texttt{GetLock}_j\texttt{()}$ later executes line 38, where the if-statement evaluates positive, and the process thus calls $u.\mathsf{MX}.\texttt{RelLock}_j\texttt{()}$. Part (c) follows immediately from Claim 4 and part (d). □

4.3 Worst-case RMR complexity

In this section we bound the number of RMRs process p incurs while executing $\texttt{Entry}_p\texttt{()}$ and $\texttt{Exit}_p\texttt{()}$. For the exit-section an $O(\Delta)$ RMRs upper bound is obvious from the for-loop. For the entry-section, we first establish that processes incur only a constant number of RMRs while executing one of the **await**-statements. From this we conclude that in each iteration of the inner entry loop (except when p is desperate), p incurs only $O(1)$ RMRs. On the other hand, after at most $\lceil \log \Delta \rceil$ iterations, p becomes desperate. We show that once desperate, p does not start another iteration of the inner entry loop. Moreover, in the iteration where p is desperate it incurs only $O(\log \Delta)$ RMRs. From all this we can conclude that while trying to win the lock of a node u, process p incurs at most $O(\log \Delta) = O(\log N)$ RMRs until it succeeds or gets promoted. Once promoted, p incurs only $O(1)$ RMRs until it enters the critical section. Hence, the worst-case is that p has to win all locks on the path from its leaf to the root. This costs $O(\Delta \cdot \log \Delta)$ RMRs, which is $O(\log N)$.

For the remainder of this section we fix an arbitrary process p. The following claim is helpful to show that p incurs only $O(1)$ RMRs while busy-waiting due to one of the **await**-statements in its entry-section.

Claim 6 Let $k \in \mathbb{N}$, u an arbitrary node and T a time interval. If one of $u.\mathsf{lock}$ or $u.\mathsf{token}$ is modified at least three times during T, then during T some process executes at least once lines 45–55, while its local variable v has value u.

Proof First assume that $u.\mathsf{lock}$ is modified at least three times. By Claim 3(a), during T some process q becomes the owner

of u.lock and ceases to be the owner of u.lock. Since q has to own the root-lock when it starts executing $\texttt{Exit}_q\,()$ (Claim 5), and it can only cease to own the lock during a CAS operation in $\texttt{Exit}_q\,()$, see Claim 3 (b), q starts executing $\texttt{Exit}_q\,()$ at some point in time during T and while it is the owner of u.lock. Thus, in order to release u.lock, process q has to execute at least one complete iteration from line 45 to line 55 of the for-loop while its local variable v has value u.

Now assume that u.token is modified at least three times during T. This implies that line 54 is executed at least three times by three (not necessarily distinct) processes p_1, p_2, p_3, while their local variables v have value u. Since at most one process' execution can be between line 44 and line 55 at a time (Claims 3 (b) and 5), it follows that p_2 executes a complete iteration of the for-loop for $v = u$ during T. □

Claim 7 While busy-waiting (in line 31, line 36, or line 41), process p incurs at most a constant number of RMRs.

Proof First consider the case that p is busy-waiting in line 41. Process p is the only one that can change notified[p] to **false**. Hence, when reading notified[p] in line 41 causes an RMR, then some other process must have changed notified[p] to **true**, and notified[p] cannot be changed again until p is done busy-waiting.

Now consider the case that p is busy-waiting in line 31 or in line 36. Let u be the node at which p is registered while busy-waiting and let k be the value of p's local variable i. Each RMR by p while p is busy-waiting is due to some other process modifying either the value of u.lock, u.apply[k], or u.token. If some other process changes u.apply[k], then by Claim 2 (a) u.apply[k] $\neq p$ until p reads u.apply[k] again and thus stops busy-waiting. Therefore we only have to count RMRs of p on u.lock and u.token.

For the purpose of a contradiction assume that p incurs at least 13 such RMRs while busy-waiting and before u.apply[k] changes for the first time. Let T be the time interval that starts when p begins busy-waiting and that ends when p incurs the 13th RMR (on u.lock or u.token).

First, assume that p is busy-waiting in line 31. Then p has previously executed line 29. Since it used thread ID k when calling u.MX.$\texttt{GetLock}_k\,()$ in line 29, the lock u.MX is "owned" by k. By the assumption that during T process p incurs 13 RMRs on u.lock, trivially u.lock must be modified at least 13 times. Then by Claim 6, during T some other process, say q, executes at least lines 45–55 while its local variable v has value u. During that iteration of the for-loop, q receives the value k when it calls u.MX.$\texttt{LockOwner}()$ in line 46, and then, in line 50, resets u.apply[k] from p to $\perp$. Since all this happens during T, this contradicts the assumption that u.apply[k] remains unchanged during T.

Now assume that p is busy-waiting in line 36. If p incurs a total of 13 RMRs on u.lock and u.token, then it incurs either at least 12 RMRs on u.lock or at least two RMRs on u.token. First, suppose p incurs at least two RMRs on u.token. Note that u.token can only change if some process executes line 54. Let tok be the value p read in line 35. If one of p's reads of u.token that causes an RMR returns a value different from tok, then p is done busy waiting. So suppose that each of these read operations return the same value. Since u.token changes between the read operations (otherwise they wouldn't incur RMRs), by the semantics of line 54, u.token takes each value in $\{0, \ldots, \Delta - 1\}$ at least twice during T. Therefore, during T, at least once the value of u.token changes from $(k-1) \bmod \Delta$ to k and later to $(k+1) \bmod \Delta$. Thus, during T some process p' receives the value $tok = k$ when it reads u.token in line 45 and later executes line 54. Therefore, during T process p' executes line 50 while its local variable v has value u and $j = tok = k$. By the assumption that p is busy-waiting in line 36 and that u.apply[k] does not change during T, u.apply[k] $= p$ when this happens. Hence, p' changes the value of that register to $\perp$, contradicting the assumption that u.apply[k] does not change during T.

Finally, suppose during T there are 12 RMRs incurred by p due to changes of u.lock. Partition T into four disjoint time intervals $T_1, \ldots, T_4$, such that during T_ℓ, $1 \leq \ell \leq 4$, exactly three RMRs occur on u.lock. Consider a time interval T_ℓ, $\ell \in \{1, 3\}$. By Claim 6, some process executes line 54 during T_ℓ and thus changes u.token. When p reads u.token the next time, that read operation incurs an RMR. Clearly, p reads u.token at least once during each time interval $T_{\ell'}$, $1 \leq \ell' \leq 4$. Hence, either in T_ℓ or in $T_{\ell+1}$ process p's read of u.token incurs an RMR. Therefore, during $T_1 \cup \cdots \cup T_4 = T$, there are at least two RMRs incurred by changes to u.token. As argued above this contradicts the assumptions. □

In the following claim we show that once p becomes desperate it finishes its entry section without starting another iteration of the inner entry loop.

Claim 8 If p is desperate at the point in time when it finishes an iteration of the inner entry loop while registered at node u, then it has captured u.lock or it has been promoted.

Proof Suppose p becomes desperate (in line 26) and after that finishes the inner entry loop without capturing u.lock and without being promoted. Then p's CAS operation in line 28 succeeds, and p's execution will eventually reach line 31. When p stops busy-waiting in line 31, u.lock $= \perp$. Let t_1 be that point in time.

By the assumptions, the CAS operation p executes in line 34 at some time t_2 fails and p's execution will eventually reach line 36. Let t_3 be the point in time when p finishes busy-waiting there. Let k be the value of p's local variable i during $[t_1, t_3]$. Note that u.MX is "owned" by k throughout $[t_1, t_3]$, because p called u.MX.$\texttt{GetLock}_k\,()$ when it executed line 29 before t_1.

Some process p' must have captured u.lock at some point in time $t \in (t_1, t_2]$ as otherwise p would have captured it itself when executing line 34 (recall that no process owned the lock at time t_1). When p finishes busy-waiting at time t_3, p' must have released the lock or changed the value of u.token. Before that and after t, process p' executes line 50 for $v = u$ and $j = k$. Thus, p' promotes p at some point in time in $[t_1, t_3]$—a contradiction. □

We can now bound the number of RMRs a process incurs while executing an iteration of the inner entry loop.

Claim 9 Consider an arbitrary iteration of the inner entry loop executed by process p. If p is not desperate, then it performs $O(1)$ RMRs during the iteration, otherwise it performs $O(\log \Delta)$ RMRs.

Proof In every line of the algorithm, except for lines 29 and 38, p can incur at most $O(1)$ RMRs. (For the **await**-operations, this follows from Claim 7.) Hence, when p is not desperate during the iteration, it performs only $O(1)$ RMRs, because it has to execute only $O(1)$ such lines of code.

If p is desperate, it also calls v.MX.Entry$_i$() and v.MX.Exit$_i$(), besides executing $O(1)$ other lines of code. The RMR-complexity of each of these two method calls is $O(\log \Delta)$, so the claim follows. □

From the previous claims we get immediately that the total number of RMRs incurred by a process is roughly proportional to the total number of iterations of the inner entry loops it executed.

Claim 10 If during its Entry$_p$() method-call, process p executes in total k iterations of the inner entry loop, then it incurs $O(k)$ RMRs.

Proof During one complete iteration of the outer entry loop, process p executes at least one iteration of the inner entry loop, and outside that at most $O(1)$ RMRs. Therefore, it suffices to restrict ourselves to one iteration of the outer entry loop, and to show that if during that p executes ℓ iterations of the inner entry loop, then it incurs at most $O(\ell)$ RMRs.

If $\ell \leq \lceil \log \Delta \rceil$, then p is not desperate during the ℓ iterations and thus by Claim 9 incurs at most $O(\ell)$ RMRs in total. If $\ell = \lceil \log \Delta \rceil + 1$, then in the ℓ-th iteration p becomes desperate. In the first $\ell - 1$ iterations it incurs a total of $O(\log \Delta)$ RMRs and in the ℓ-th iterations it incurs $O(\log \Delta)$ RMRs (by Claim 9). By Claim 8, p will not start another iteration once it becomes desperate, and therefore, $\ell \leq \lceil \log \Delta \rceil + 1$. Therefore, in this case the total number of RMRs during the iteration of the outer entry loop is $O(\log \Delta) = O(\ell)$. □

Claim 11 Each process p performs at most $O(\Delta)$ steps during Exit$_p$().

Proof The claim follows immediately from the fact that the height of the tree is Δ. □

Claims 8, 10, and 11 immediately yield the main result of this section.

Lemma 1 *Each process p incurs at most $O(\Delta \cdot \log \Delta)$ RMRs during its execution of* Entry$_p$() *and* Exit$_p$().

Proof For Exit$_p$() the statement follows immediately from Claim 11. Now consider Entry$_p$(). By Claim 8, p executes at most $\lceil \log \Delta \rceil + 1$ iterations of the inner entry loop during one iteration of the outer entry loop. There are Δ nodes on the path from leaf$[p]$ to the root, and so p executes at most Δ iterations of the outer entry loop. Hence, the total number of iterations of the inner entry loop is $\Delta \cdot (\lceil \log \Delta \rceil + 1)$, and therefore the result follows from Claim 10. □

4.4 Expected RMR complexity

We now bound the expected number of RMRs a process p incurs. First, we will see that once p gets promoted it will not execute any other iteration of the inner entry loop, and thus not asymptotically increase the RMR complexity.

Claim 12 Once process p gets promoted it will not execute another iteration of the inner entry loop during its current Entry$_p$() method-call. Instead, its execution will reach line 41 within a constant number of steps.

Proof Let r be the process that promotes p (by executing line 50), i.e., r successfully executes u.apply$[j]$.CAS$(p, \perp)$ for some node u and some index j. Immediately before that CAS operation, u.apply$[j] = p$, and thus by Claim 2 (b),(c) process p is registered at node u and its next operation is in lines 24–28 or 31–40.

If p's next operation is in line 24–28, then p's CAS operation in line 28 will fail. Thus, p's execution will reach line 34 within a constant number of steps, and without executing line 30. Therefore, assume that p's next operation is in one of lines 31–40. Then p cannot set u.apply$[j]$ back to p in the current iteration of the outer entry loop, as the only place where this could happen is line 30. Hence, p will not wait (or continue to wait) in one of the **await**-operations, and it will also exit the inner entry loop. Eventually, after a constant number of steps, p will execute line 40, and the CAS operation therein fails. Therefore, p's execution reaches line 41 within a constant number of steps, and without executing another iteration of the inner entry loop. □

Now consider a situation in which a process applies for promotion (i.e., u.apply$[k] = p$ for some node u and some index k) while it is not desperate or about to become desperate (i.e., its local variable ctr has value less than $\lceil \log \Delta \rceil$). In the claim below we show that if at that time some other

process chooses k as its random value j' in line 47, then p will get promoted before it can execute another complete iteration of the inner entry loop.

We say that process p *participates in a promotion lottery* at time t, if there is a process q and a node u, such that at time t,

1. process q executes line 47,
2. q's local variable v has value u, and
3. $u.\mathsf{apply}[k] = p$ for some $0 \leq k < \Delta$ (i.e., p is registered at node u and p's application for promotion is pending).

We say that p *wins* a promotion lottery at time t, if at that point some process q executes line 47 and chooses the value j', such that the value of $u.\mathsf{apply}[j']$ is p.

Claim 13 Let t be some point in time at which p is not desperate and wins a promotion lottery. Then p will either get promoted or become desperate before it can execute a complete iteration of the inner entry loop.

Proof For the purpose of a contradiction assume that in a time interval T, starting at time t, p executes a complete iteration of the inner entry loop without being promoted or becoming desperate.

Since p has to participate in a promotion lottery, in order to win it, at time t some process q executes line 47, choosing the random value $j' = k$, while its local variable v has value u. Therefore, q owns the lock $u.\mathsf{lock}$ at time t. Moreover, p can only win the promotion lottery if $u.\mathsf{apply}[k] = p$ at time t. This requires that p is registered at node u (Claim 2 (b)).

Now let $t' \geq t$ be the point in time when q executes line 50 for $j = k$. Then obviously q owns $u.\mathsf{lock}$ throughout $[t, t']$. We consider the cases, $t' \in T$ and $t' \notin T$.

First assume $t' \in T$. By the assumption that p doesn't get promoted during $T \supseteq [t, t']$, p must withdraw its application in the time interval $[t, t')$. Since p is not desperate at any point in time during T, it can only withdraw its application in line 40 (and not in line 28). Moreover, this must happen while p is still registered at node u. But then at some point before t', p executes line 39 while registered at node u, and at that point it must own the lock of node u in order to be able to leave the inner entry loop. But then p will not cease to own the lock throughout T (Claim 3 (b)), which contradicts that q owns it throughout $[t, t']$.

Now consider the case $t' \notin T$, i.e., t' occurs after T. Then q owns $u.\mathsf{lock}$ throughout T. By the assumption that p executes a complete iteration of the inner entry loop during T, at some point during that interval it executes line 34, the CAS therein fails (because q owns $u.\mathsf{lock}$), and then at point $t^* \in T$, p's execution reaches line 36. Since p doesn't get promoted during T, it can only stop busy-waiting there, when $u.\mathsf{token}$ changes or when q releases $u.\mathsf{lock}$. Neither of that happens before t', and thus not during T, either. Hence, p cannot complete its iteration of the inner entry loop. □

As a consequence of this claim, whenever process p participates in a promotion lottery, it has a $1/\Delta$ chance of getting promoted and thus finishing the entry section within a constant number of RMRs. It is important that the point of the promotion lottery is when j' is being chosen at random. The claim says that if a promotion lottery happens, and if the outcome is such that p wins the lottery, then even the strong adversary cannot prevent p from being promoted.

In the following we show that if the adversary lets p execute six consecutive iterations of the inner entry loop, then it cannot avoid that p participates in at least one promotion lottery. Hence, the adversary must either ensure that p wins each node lock early, or it risks that p wins a promotion lottery and consequently gets promoted.

Claim 14 Let T be a time interval throughout which p is registered at node u and p is not desperate. If during T, p executes at least six complete iterations of the inner entry loop, then there is a point $t \in T$ such that p participates in a promotion lottery at time t.

Proof For the purpose of contradiction assume that p does not participate in a promotion lottery during T. Consider a time interval $T' \subseteq T$ during which p completes five iterations of the inner entry loop, and after which it starts a sixth iteration. Then during T', p does not capture $u.\mathsf{lock}$, and by Claim 12, p does not get promoted. Hence, $u.\mathsf{apply}[k] = p$ throughout T' (recall that p is not desperate).

During T', p finishes line 36 at least 5 times. But p can only finish that line, when $v.\mathsf{lock}$ changed to $\perp$ after p executed line 34, or when $v.\mathsf{token}$ changed after p executed line 35. Therefore, during T' at least one of $u.\mathsf{lock}$ and $u.\mathsf{token}$ changes 3 times or more. By Claim 6, during T' some other process q executes lines 45–55, while its local variable v has value u. Therefore, there is a point in time $t \in T' \subseteq T$, at which process q executes line 47, and at which q's local variable v has value u. Since $u.\mathsf{apply}[k] = p$, p participates in a promotion lottery at time t. □

Lemma 2 *The expected number of RMRs incurred by process p during a call of* $\mathtt{Entry}_p()$ *and* $\mathtt{Exit}_p()$ *is* $O(\Delta)$.

Proof Let α be the path from $\mathsf{leaf}[p]$ to the root of the arbitration tree. By Claim 11, p incurs at most $O(\Delta)$ RMRs during $\mathtt{Exit}_p()$. We prove that p executes an expected number of $O(\Delta)$ iterations of the inner entry loop during its call of $\mathtt{Entry}_p()$. The lemma then follows from Claim 10.

Let X_u be the number of iterations of the inner entry loop that p executes while it is registered at node $u \in \alpha$. Only in the last of the X_u iterations p can become desperate or get promoted (by Claim 12). Thus, by Claim 14, p participates in at least $\lfloor (X_u - 2)/6 \rfloor$ promotion lotteries, while registered

at node u and while its local variable ctr has a value less than $\lfloor \log \Delta \rfloor$. Hence, in total, p participates in at least[1]

$$Y := \sum_{u \in \alpha} \lfloor (X_u - 2)/6 \rfloor \geq \sum_{u \in \alpha} (X_u/6 - 2)$$
$$= -2\Delta + \frac{1}{6} \cdot \sum_{u \in \alpha} X_u \quad (1)$$

promotion lotteries, each one happening in a different iteration of p's inner entry loop and while p's local variable ctr satisfies $ctr \leq \lfloor \log \Delta \rfloor - 1$. Then p does not win the first $Y - 2$ out these Y lotteries: Once it wins a promotion lottery while $ctr \leq \lfloor \log \Delta \rfloor - 1$, by Claim 13 it gets promoted no later than during its following iteration of the inner entry loop (note that due to the ctr value, it is not desperate during that iteration), and by Claim 12 it does not execute any more iterations after that.

The probability that p wins a promotion lottery in which it participates is $1/\Delta$, and the events are independent for each promotion lottery. Thus, $\text{Prob}(Y > k) \leq (1 - 1/\Delta)^k$ and so Y has expectation

$$E[Y] = \sum_{k \geq 0} \text{Prob}(Y > k) \leq \sum_{k \geq 0} (1 - 1/\Delta)^k = \Delta. \quad (2)$$

We conclude from (1) and (2) that the total number of iterations of the inner entry loops performed by p has expectation

$$E\left[\sum_{u \in \alpha} X_u\right] \leq E\left[6 \cdot Y + 12 \cdot \Delta\right] \leq 18 \cdot \Delta.$$

□

Remark 1 As observed by Golab, Higham, and Woelfel [12], replacing atomic objects with linearizable implementations can increase the power of the strong adversary and thus might increase the expected complexity of a randomized algorithm. For example, if a non-atomic but linearizable operation o overlaps with a coin flip made by another process, then the adversary might schedule other events in such a way that the linearization point of o occurs either before the coin flip or after it, depending on the outcome of the coin flip. In general, under certain circumstances the adversary can use the result of a coin flip to determine the linearization order of implemented operations that overlap with that coin flip, but still linearize these operations before the coin flip. On the other hand, if these operations are atomic, then their order can only depend on the coin flip outcome if they happen after the coin flip.

However, in the case of our algorithm, we can safely replace atomic CAS operations with linearizable ones: The proof of Lemma 2 states that in any scheduling in which p executes at least $c \cdot \Delta + d(n)$ RMRs (for an appropriate constant c and some function d), then p has to participate in at least $\Omega(d(n))$ promotion lotteries. This argument relies on the statement that during the time interval T defined in Claim 14, process p participates in at least one promotion lottery.

Consider the point in time t, when some process q whose local variable v has value u makes a random choice in line 47. By definition, p participates in a promotion lottery at time t if and only if $u.\mathsf{apply}[k] = p$ at that point for some $0 \leq k < \Delta$. Thus, an adversary can only gain an advantage if it manages to decide *after* the coin flip has happened (and depending on the outcome of that coin flip), whether $u.\mathsf{apply}[k] = p$ at the point of the coin flip. The only way to potentially achieve this is by letting an implemented CAS operation, which affects whether $u.\mathsf{apply}[k] = p$, overlap with point t.

But recall that $u.\mathsf{apply}[k] = p$ throughout the time interval T' considered in the proof of Claim 14. In fact, throughout T' no process executes a CAS operation that could potentially change $u.\mathsf{apply}[k]$ to a value different from p. Therefore, even if CAS operations are implemented and not atomic, p will participate in any promotion lottery that happens at node u during the interval T'. As a consequence, nothing changes in the analysis, if atomic CAS operations are replaced by linearizable implementations.

4.5 Starvation freedom

It remains to prove that the algorithm is starvation free. We first prove that no process can starve in the promotion queue.

Claim 15 If r is a process in promQ, then r does not starve.

Proof Suppose some process r' executes line 54. Let r_k be the k-th process in promQ, at that point. We prove the following by induction on k:

(a) r_k will be removed from promQ, and
(b) r_k will finish its entry-section.

The claim then follows: When some process q adds $r = r_k$ to promQ, then q later executes line 54 (at a point when r is still in promQ), and so by (b) r_k finishes its entry-section. By Claim 11, no process starves in the exit-section. Thus, it remains to show (a) and (b).

If $k = 1$, then process r' will remove r_k from promQ when it executes line 60 and then signal r_k by writing **true** to $\mathsf{notified}[r_k]$ (line 62). From the semantics of line 50, r_k has been promoted before it was enqueued in promQ. Thus, by Claim 12, r_k's execution will reach line 41 within a constant number of its steps. Since r will be signaled by r', it will eventually stop spinning in line 41 and thus finish its entry-section.

[1] Note that some of the terms in the sum may become negative, if $X_u < 2$. This is not a problem, because Y is only a lower bound on the number of promotion lotteries.

Now suppose $k > 0$. Note that r_1 is the process in front of, the queue when process r' executes line 54. By the induction hypothesis, process r_1 will be removed from promQ and later enter its exit section. Thus, its execution will reach line 54, but at that point in time, r_k has moved up to the $(k-1)$-th position in the queue (because r_1 was removed). The claim now follows from the induction hypothesis. □

Lemma 3 *The algorithm* `RandomMutEx` *is starvation free.*

Proof By Claim 11, `Exit`$_p$`()` is even wait-free, so it suffices to prove that no process starves while executing `Entry`$_p$`()`. By Claim 8 and the semantics of the inner and outer entry loop, each process performs only a bounded number of iterations of the inner and outer entry loops. Therefore, it suffices to prove that no process spins indefinitely in one of the **await**-operations, if all other processes get to take enough steps. (Note that this also implies that no process remains indefinitely in the critical section of v.MX for a node v, and so starvation-freedom follows from starvation-freedom of that object).

First consider a process p that spins in line 41. Clearly, p must previously have been promoted, by some, process p'. The only way p' can promote p is by successfully executing the CAS operation in line 50 for $q = p$. But that means that p' then adds p to promQ at line 51, and so later, when p' executes line 53, $p \in$ promQ. By Claim 15, p does not starve.

Now assume that some process spins forever at line 31 or line 36. We say that the process *starves at node* u, if it is registered at node u while spinning indefinitely. Suppose process p starves at some node u, but no process starves at a node $u' \neq u$ that is on the path from u to the root (clearly, if some process starves, then u and p exist). Each time p reads u.lock, and u.lock has changed since p's previous read, p incurs an RMR. Due to the upper bound on the worst-case RMR complexity of the algorithm (Lemma 1), it follows that u.lock will eventually stop changing. By the assumption that p starves at node u, u.lock $= q \neq \perp$ when it stops changing (because otherwise p would stop spinning when it reads u.lock the next time). But if q does not release the lock of u, then it must be starving itself, or else it would eventually start its exit-section and by Claim 3(c), it would release u.lock. From Claim 1 we conclude that q starves at a node $u' \neq u$ that is on the path from u to the root—a contradiction. □

From Lemmas 1, 2, and 3, and $\Delta = O(\log N / \log\log N)$ we immediately obtain the following result.

Corollary 3 *Algorithm* `RandomMutEx` *is a starvation-free mutual exclusion algorithm, where one run of the algorithm on the CC model incurs at most* $O(\log N)$ *RMRs and an expected number of* $O(\log N / \log\log N)$ *RMRs.*

5 Modifications for the DSM model

We now describe how to modify our algorithm for the DSM model, so that it achieves the same RMR complexity as the algorithm for the CC model. In the DSM model, processes that try to capture a node-lock cannot busy-wait on the lock without incurring an unbounded number of RMRs. Instead, we must ensure that they spin on local variables. We therefore need a mechanism that allows a process releasing a node-lock to notify all processes waiting for that lock to be released.

Such a mechanism can be implemented with primitives which we call wait-signal objects (short: WS objects). A WS object W_p is *owned* by a unique process p, as indicated by the subscript. It provides two methods, `Wait()`, which can only be called by a process $r \neq p$, and `Signal()`, which can only be called by process p. Each process is allowed to call one of these methods at most once, and the methods take no arguments or return values. The semantics, progress condition, and RMR complexity are as follows:

(S) If r calls `Wait()`, then that method call does not terminate before p has called `Signal()`.
(P) `Signal()` is wait-free, and in any execution where process p calls `Signal()`, all `Wait()` calls terminate as long as all processes get to take sufficiently many steps.
(R) Each function call `Wait()` or `Signal()` incurs at most $O(1)$ RMRs.

5.1 Using wait-signal objects to accommodate the DSM algorithm

We postpone a description of an implementation of WS objects, and first show how the algorithm presented in Section 3 can be modified to achieve the desired expected and worst-case RMR complexity in the DSM model. In the algorithm for the CC model, presented in Figs. 1–3, a process p can busy-wait only in lines 31, 36, and 41. In our analysis for the CC-model, the worst- and expected RMR complexity is only based on the total number of iterations of the inner entry loop, see Claim 10. Moreover, if we use for example Yang and Anderson's [19] mutual exclusion algorithm to implement the object u.MX, then the RMR complexity of the operations on that object are asymptotically the same in the CC and the DSM model. Hence, these **await**()-operations are the only operations that can cause the algorithm to incur a higher asymptotic RMR cost in the DSM model than in the CC model.

We can allocate each register notified[p] to p's local memory segment. This way, line 41 incurs no RMRs in the DSM model.

We now show how to deal with lines 31 and 36. When process p tries to lock node v, then instead of trying to swap its own ID into v.lock, it creates a new WS object W_p, and

Fig. 4 Implementation of the wait-signal object

Class WS_p

```
1  shared: ;
2      spin: array int init 0   /* spin[q] is stored in q's mem. segment          */
3      sleep: array int init 0  /* sleep[0...N-1] is in p's mem. segment          */
4  local: ;
5      S, S': Set init ∅;
6      r: int init 0;
7  Functions:;
8      Wait_q() for q ≠ p;
9      Signal_p();
```

Function Wait_q

```
10 spin[q] := χ_∅;
11 if sleep[q].CAS(0,1) then
12 |   await (spin[q] ≠ χ_∅);
13 |   Let S be the set encoded by the
   |   characteristics vector in spin[q].;
14 |   S' := S − {q};
15 |   if ∃r ∈ S then
16 |   |   spin[r] := χ_S';
17 |   end
18 end
```

Function Signal_p

```
19 S := ∅;
20 for r ∈ {0,...,N−1} do
21 |   if ¬sleep[r].CAS(0,1) then
22 |   |   S := S ∪ {r};
23 |   end
24 end
25 if ∃r ∈ S then
26 |   spin[r] := χ_S;
27 end
```

tries to swap a pointer ptr pointing to that object into v.lock. Suppose it succeeds, so it owns the lock v.lock. If process r later on fails to capture v.lock, because the lock is still owned by process p, then r obtains the pointer ptr to W_p. Thus, r can call W_p.`Wait()` in order to wait until the lock has been released. On the other hand, after releasing the lock of node v, process p simply calls W_p.`Signal()` in order to notify all processes that the lock has been released.

The implementations of the entry- and exit-sections for the DSM model can be found in Figs. 5 and 6 at the end of this section. Note that in this implementation we assume that a garbage collector takes care of freeing up the memory for unreferenced WS objects.[2]

One additional noteworthy detail is that in order to hand the root-lock over from process p to process q, now p has to create a new WS object W_q and swap a pointer to that object into root.lock (see line 79).

Mutual exclusion follows with the same arguments as for the CC model. Also the analysis of the RMR-complexity is essentially the same. Starvation freedom follows immediately from (P) and (S) and using the same arguments provided for the CC model algorithm.

5.2 Implementation of wait-signal objects

A WS object W_p owned by process p stores an array sleep$[0 \ldots N-1]$ of CAS objects that store values in $\{0, 1\}$. Initially, sleep$[i] = 0$ for all $0 \leq i < N$. Moreover, we use a shared array spin$[0 \ldots N-1]$, where register spin$[i]$ is located in process i's memory. Each array entry stores a characteristics vector χ_S of a set $S \subseteq \{0, \ldots, N-1\}$ (i.e., $\chi_S = (x_0, \ldots, x_{N-1}) \in \{0, 1\}^N$ such that $x_i = 1$ if and only if $i \in S$).

For the rest of this description, all method calls `Wait()` and `Signal()` are for an object W_p owned by process p. The idea is that in a call to `Wait()` by process q, q "notifies" p that it is waiting by swapping the value of sleep$[q]$ from 0 to 1 using a CAS-operation. If that succeeds, q can start to wait for spin$[q]$ to change. On the other hand, when p calls `Signal()`, it also tries to swap all values sleep$[q]$ from 0 to 1. This way, it determines a set S of processes that successfully swapped their sleep-array entries. Each process not in S that still tries to swap its sleep-array entry from 0 to 1 will fail to do so, and thus know that the `Signal()` call has already been invoked. Once p has collected the set S of processes that are now waiting, it has to notify all processes $r \in S$ by changing their spin$[r]$-values. Since this may cause too many RMRs if S is large, p only notifies one arbitrary process $r \in S$, by writing χ_S into spin$[r]$, and then r notifies another process in S and so on.

The code of an implementation can be found in Fig. 4. Here is a detailed description of this implementation.

Wait. In order to wait on the WS object, process q first resets spin$[q]$ to the value $\chi_\emptyset$ (line 10). In line 11 it then tries to swap the value of sleep$[q]$ from 0 to 1. If that fails, process p has already called `Signal()`, and so q is done. Otherwise, q starts spinning on spin$[q]$ until that value changes to χ_S for some set $S \neq \emptyset$ (line 12). The set $S' = S - \{q\}$, determined in lines 13–14 is a set of processes that still need

[2] For systems where garbage collection is not supported, it is not difficult to extend our algorithm so that each process that creates a WaitSignal object also takes care of freeing the object's memory once it is not referenced anymore. However, as we are mainly concerned here with the time and not space the complexity of randomized mutual exclusion, a detailed description of such a mechanism is beyond the scope of this paper.

Fig. 5 The entry section for the DSM model

```
Function EntryDSM_p
28 notified[p] := false;
29 v := leaf[p];
30 repeat
31     Let i be the integer such that v is the (i + 1)-th child of parent(v);
32     v :=parent(v);
33     v.apply[i].CAS(⊥, p);
34     ctr := 0;
35     repeat
36         ctr := ctr + 1;
37         if ctr > ⌈log Δ⌉ then
38             if v.apply[i].CAS(p, ⊥) then
39                 v.MX.GetLock_i();
40                 v.apply[i].CAS(⊥, p);
41                 ptr := v.lock;
42                 if ptr ≠ ⊥ then ptr→Wait();
43             end
44         end
45         Create a new WS object (owned by p) and let ptr point to it.;
46         if ¬ v.lock.CAS(⊥, ptr) then
47             ptr := v.lock;
48             if ptr ≠ ⊥ then ptr→Wait();
49         end
50         if v.MX.LockOwner() = i then v.MX.RelLock_i();
51     until v.apply[i] ≠ p ∨ v.lock = p;
52     if ¬ v.apply[i].CAS(p, ⊥) then
53         await (notified[p] = true) ;
54     end
55 until notified[p] ∨ v = root;
```

Fig. 6 The exit section for the DSM model

```
Function ExitDSM_p
56 foreach node v on the path from leaf[p] to the root, where v.lock stores a pointer to a
   WS object owned by p do
57     tok := v.token;
58     i := v.MX.LockOwner();
59     Pick j′ uniformly at random from {0, . . . , Δ − 1};
60     for j ∈ {j′, tok, i} − {⊥} do
61         q := v.apply[j];
62         if q ≠ ⊥ ∧ v.apply[j].CAS(q, ⊥) then
63             promQ.Enq(q);
64         end
65     end
66     v.token := (tok + 1) mod Δ;
67     ptr := v.lock;
68     if v ≠ root then
69         v.lock.CAS(ptr, ⊥);
70         ptr→Signal();
71     end
72 end
73 if promQ = ∅ then
74     root.lock.CAS(ptr, ⊥);
75     ptr→Signal();
76 else
77     q := promQ.Deq();
78     Create a new WS object owned by q and let ptr′ point to it.;
79     root.lock.CAS(ptr, ptr′);
80     notified[q] := true
81 end
```

to be "signaled". Hence, if $S' \neq \emptyset, q$ picks an arbitrary process $r \in S'$, and signals it by writing $\chi_{S'}$ to spin$[r]$ (lines 15–16).

Signal. In a call to Signal$_p$, process p tries to swap the value of each array entry sleep$[r]$ from 0 to 1, and if that succeeds adds r to an initially empty set S (lines 19–24). Finally, it picks an arbitrary process $r \in S'$, and signals it by writing χ_S to spin$[r]$ (lines 25–26).

Lemma 4 *The implementation of WS objects in Fig. 4 satisfies properties (S), (P), and (R).*

Proof We first prove property (S). Suppose process q calls `Wait()`. If the `CAS`-operation in line 11 fails, then p has obviously already called `Signal()`. Hence, suppose that `CAS`-operation succeeds. Then due to the semantics of line 12, q can only finish its `Wait()` call, if some other process writes to $\mathsf{spin}[q]$. By a simple inductive argument, the first process q' to write to some register $\mathsf{spin}[i]$, $0 \leq i < N$, is p. Hence, q can only finish its method call, after p has called `Signal()`.

Next we prove property (P). Wait-freeness of `Signal()` follows immediately from the code. Now consider an execution E in which p calls `Signal()` and in which each process gets to take sufficiently many of steps. Since `Signal()` is wait-free, it finishes during E. Clearly, if process r's CAS-operation in line 11 succeeds then and only then process p's CAS-operation in line 21 on $\mathsf{sleep}[r]$ fails. Hence, once p finishes line 24, its local variable S represents exactly the set of processes who call `Wait()` during E and whose CAS-operation in line 11 succeeds. From the semantics of the if-statement in that line, all other processes that call `Wait()` during E finish that function call. Hence, consider the processes in S. Again from the semantics of the if-statement in line 11, each process in S executes line 12. Since exactly the processes in S busy-wait in line 11 and since p will eventually change $\mathsf{spin}[r]$ to χ_S for some $r \in S$, it is obvious that all processes in S will eventually stop busy-waiting.

Finally, we prove property (R). Method `Wait()` has RMR-complexity $O(1)$ since a calling process q only spins on the register $\mathsf{spin}[q]$, which is local to q, and everything else requires only a constant number of steps. For `Signal()` it suffices to note that all memory registers in the for-loop are local to p. □

Conclusion

We presented the first randomized mutual exclusion algorithm that achieves a better expected RMR complexity than what can be achieved in the worst case. While the speed-up is only a modest $O(\log \log n)$ factor, this result shows that randomization can help in principle, even against the strong adversary. It is an important open question, whether this upper bound can be further improved. However, it seems unlikely that an improvement can be achieved by a standard arbitration tree based approach: For such algorithms, it seems, the expected RMR cost is at least roughly the maximum of the branching factor and the depth of the tree, and that maximum is at least $\Omega(\log n / \log \log n)$. We therefore conjecture that at least for the strong adversary an expected RMR cost of $O(\log n / \log \log n)$ is optimal. However, in order to find more efficient mutual exclusion algorithms, one can also consider weaker adversary models, as e.g., the oblivious adversary which cannot base its scheduling on the outcome of coin flips.

References

1. Alur, R., Taubenfeld, G.: Results about fast mutual exclusion. In: Proceedings of the 13th IEEE Real-Time Systems Symposium, pp. 12–22 (1992)
2. Anderson, J., Kim, Y.-J., Herman, T.: Shared-memory mutual exclusion: major research trends since 1986. Distrib. Comput. **16**, 75–110 (2003)
3. Anderson, J.H., Kim, Y.-J.: Fast and scalable mutual exclusion. In: Proceedings of the 13th International Symposium on Distributed Computing (DISC), pp. 180–194 (1999)
4. Anderson, J.H., Kim, Y.-J.: Adaptive mutual exclusion with local spinning. In: Proceedings of the 14th International Symposium on Distributed Computing (DISC), pp. 29–43 (2000)
5. Anderson, J.H., Kim, Y.-J.: An improved lower bound for the time complexity of mutual exclusion. Distrib. Comput. **15**, 221–253 (2002)
6. Anderson, T.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. **1**, 6–16 (1990)
7. Aspnes, J.: Randomized protocols for asynchronous consensus. Distrib. Comput. **16**, 165–175 (2003)
8. Attiya, H., Hendler, D., Woelfel, P.: Tight RMR lower bounds for mutual exclusion and other problems. In: Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC), pp. 217–226 (2008)
9. Danek, R., Golab, W.M.: Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. Distrib. Comput. **23**(2), 87–111 (2010)
10. Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**, 569 (1965)
11. Golab, W.: Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations. PhD thesis, University of Toronto (2010)
12. Golab, W., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC) (2011) (to appear)
13. Golab, W.M., Hadzilacos, V., Hendler, D., Woelfel, P.: Constant-RMR implementations of cas and other synchronization primitives using read and write operations. In: Proceedings of the 26th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 3–12 (2007)
14. Jayanti, P.: Adaptive and efficient abortable mutual exclusion. In: Proceedings of the 22nd SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), pp. 295–304 (2003)
15. Jayanti, P., Petrovic, S., Narula, N.: Read/write based fast-path transformation for FCFS mutual exclusion. In: Proceedings of the 31st Conference on Current Trends in Theory and Practice of Informatics (SOFSEM), pp. 209–218 (2005)
16. Kim, Y.-J., Anderson, J.: A time complexity bound for adaptive mutual exclusion. In: Proceedings of the 15th International Symposium on Distributed Computing (DISC), pp. 1–15 (2001)
17. Kim, Y.-J., Anderson, J.H.: Nonatomic mutual exclusion with local spinning. Distrib. Comput. **19**(1), 19–61 (2006)
18. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming. Prentice Hall, Upper Saddle River (2006)
19. Yang, J.-H., Anderson, J.H.: A fast, scalable mutual exclusion algorithm. Distrib. Comput. **9**(1), 51–60 (1995)