Randomized Mutual Exclusion in O(log N / log log N) RMRs

[Extended Abstract]

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

Philipp Woelfel
Department of Computer-Science
University of Calgary
woelfel@cpsc.ucalgary.ca

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. A recent proof [9] established an $\Omega(\log N)$ lower bound on the number of RMRs incurred by processes as they enter and exit the critical section, matching an upper bound by Yang and Anderson [18]. Both these bounds apply for algorithms that only use read and write operations. The lower bound of [9] only holds for deterministic algorithms, however; the question of whether randomized mutual exclusion algorithms, using reads and writes only, can achieve sub-logarithmic expected RMR complexity remained open. This paper answers this question in the affirmative.

We present two strong-adversary [8] randomized local-spin mutual exclusion algorithms. In both algorithms, processes incur $O(\log N/\log\log N)$ expected RMRs per passage in every execution. Our first algorithm has sub-optimal worst-case RMR complexity of $O\big((\log N/\log\log N)^2\big)$. Our second algorithm is a variant of the first that can be combined with a deterministic algorithm, such as [18], to obtain $O(\log N)$ worst-case RMR complexity. The combined algorithm thus achieves sub-logarithmic expected RMR complexity while maintaining optimal worst-case RMR complexity. Our upper bounds apply for both the *cache coherent* (CC) and the *distributed shared memory* (DSM) models.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*

General Terms

Algorithms, Theory

Keywords

Mutual Exclusion, Remote Memory References (RMRs)

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 [11], the mutual exclusion problem is at the core of Distributed Computing and is still the focus of intense research [3, 17]. 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 [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 [7]. Indeed, recent mutual exclusion research investigates the RMR complexity of local-spin algorithms (see [2, 5, 6, 10, 14, 15, 16] for some examples).

Anderson and Kim presented a simple randomized variant [16] of their (deterministic) read/write adaptive mutual exclusion algorithm [5]. Their randomized variant has expected $O(\log k)$ RMR complexity, where k is point contention. Since they presented a lower bound that precludes deterministic algorithms with $O(\log k)$ RMR complexity, this established a separation in terms of RMR complexity between randomized and deterministic adaptive algorithms. With the single exception of [16], prior art local-spin mutual exclusion research dealt exclusively with deterministic algorithms. Yang and Anderson presented the first $O(\log N)$ RMRs read/write mutual exclusion algorithm [18]. Anderson and Kim [4] conjectured that this was best possible. This conjecture was recently proved by Attiya, Hendler and Woelfel [9]. The lower bound of [9] holds only for deterministic algorithms, however, since it assumes that a scheduling adversary knows processes' future steps. The question of whether randomization can help break the logarithmic barrier of [9] thus remained open. In this paper, we provide a positive answer to this question.

Golab et al. [12] presented a constant-RMRs read/write implementation of *compare-and-swap.* Moreover, they proved that any shared-memory algorithm using reads, writes, and conditional operations (such as *compare-and-swap*), can be simulated by a read/write algorithm, with only a constant multiplicative increase in RMR complexity. The algorithms we present use variables that support the *compare-and-swap* and read operations. It follows from [12], that these variables can be implemented from reads and writes only while maintaining asymptotic RMR complexities.[1]

Our Contributions. We establish a separation in terms of RMR complexity between randomized and deterministic read/write mutual exclusion algorithms. Since the lower bound of [9] holds also for algorithms that can use conditional operations in addition to reads and writes, this separation applies for such algorithms also.

We present starvation-free randomized mutual exclusion algorithms for the CC model, in which processes incur $O(\log N/\log\log N)$ expected RMRs per passage in every execution, even against a strong-adversary [8] that can schedule processes according to their execution history. Our first algorithm has sub-optimal worst-case RMR complexity of $O\big((\log N/\log\log N)^2\big)$. Our second algorithm is a variant of the first that can be combined with a deterministic algorithm, such as [18], to obtain $O(\log N)$ worst-case RMR complexity. The combined algorithm thus achieves sub-logarithmic expected RMR complexity while maintaining optimal worst-case RMR complexity. We then describe (in Section 5) the simple modifications that are required for these algorithms to achieve the same properties in the DSM model.

Model. Our model of computation is based on [13]. A concurrent system models an asynchronous shared memory system where N *processes* communicate by executing *operations* on shared *variables*. Each process is a sequential execution path that performs a sequence of *steps*, each of which invokes a single operation on a shared variable.

In the *cache-coherent* (CC) computation 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. A memory access to a remote variable is called a *remote memory reference* (RMR). In the *distributed shared memory* (DSM) computation model, each processor has its own locally-accessible shared-memory segment. A processor can also access variables in remote memory segments; each such access incurs an RMR.

An execution of the entry section and the subsequent exit section by a process is called a *passage*. The *worst-case RMR complexity* of a mutual exclusion algorithm is the supremum number of RMRs performed by a process as it performs a passage, where the supremum is taken over all passages in all of the algorithm's executions. The *expected RMR complexity* of a mutual exclusion algorithm is the supremum, over all the algorithm executions, of the expected number of RMRs performed by a process as it performs a passage.

The *compare-and-swap* operation (abbreviated CAS) is defined as follows: *CAS(v,expected,new)* changes the value of variable 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*. A stronger version of *CAS*, such as that implemented in [12], returns the value of v just before *CAS* is applied, instead of returning *true* or *false*.

[1]In the *CAS* implementation of [12], the *CAS* operation returns the previous object value. Although a read operation was not implemented in [12], it can be easily implemented by calling *CAS* with an expected value outside the object's values-domain; this operation returns the object's current value without changing it.

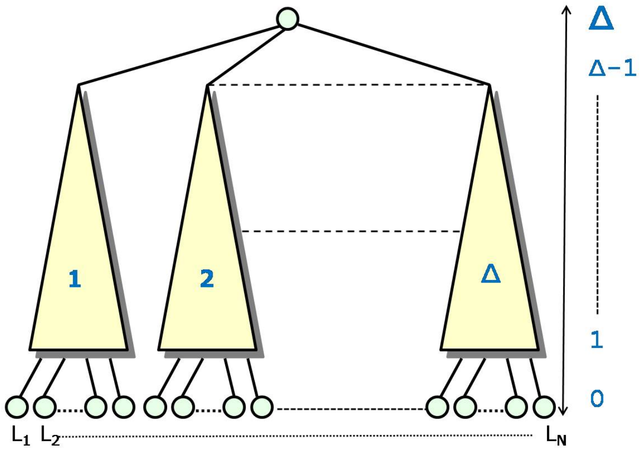

Figure 1: The tree used by the randomized algorithm.

2. THE RANDOMIZED ALGORITHM

In this section, we describe our first randomized mutual-exclusion algorithm. In the following description, we let N denote the number of processes sharing the implementation. For presentation simplicity, we assume w.l.o.g. that $N = \Delta^{\Delta}$ for some positive integer Δ. Note that $\Delta = \Theta(\log N/\log\log N)$.

The data structure underlying the algorithm consists of a complete Δ-ary tree of height Δ and with leaves $L_1, \ldots, L_N$ (see Figure 1). We say that a node is in *level i* if its height is i (leaves have height 0 and the root has height Δ). Each inner tree node is represented by a structure that includes a *lock* field accessed by read and *CAS* operations. Each process p is associated with the leaf L_p, and the parent of a non-root node v is denoted *parent(v)*. We let r denote the root node and define $parent(r) = \perp$.

The pseudo-code of the randomized algorithm is shown in Figure 2. We first describe the key ideas on which our algorithm is based and then provide a detailed description of the pseudo-code of Figure 2.

A process p can enter the critical section in one of two ways. It can either succeed in *capturing* all the locks on the path from its leaf to the root or, otherwise, it can be *promoted* by some other process q when q exits the critical section. If p is promoted, then it no longer needs to climb up the tree. In this case, p busy-waits until it is *signalled* to enter the critical section.

Our algorithm employs two types of promotion mechanisms that differ in the manner in which the process to be promoted is selected. A *promote* array of size Δ is associated with every internal tree node.

Figure 2: Randomized Mutual Exclusion Algorithm for Process $p \in \{0, \ldots, N-1\}$

```
1  define Node: struct {lock: int init ⊥, promote: array
   [0..$\Delta - 1$] of int init ⊥, nextToPromote: int init 0}
2  shared: $\mathcal{T}$: Complete $\Delta$-ary tree with leaves
   $L_1, \ldots, L_N$, promQ: Queue, spin: array [1..N] of
   boolean init false
3  local: n: Node, ch, next, j, promLevel, q: int
4  loop
5    | Non Critical Section
6    | promLevel=$\Delta + 1$
7    | n=$L_p$
8    | for level=1 to $\Delta$ do
9    |   | Let ch be the integer such that n is the
     |   | (ch+1)'th child of parent(n)
10   |   | n=parent(n)
11   |   | CAS(n.promote[ch],⊥,p)
12   |   | loop
13   |   |   | if CAS(n.lock,⊥,p) then
14   |   |   |   | if CAS(n.promote[ch],p,⊥) then
15   |   |   |   |   | continue for
16   |   |   |   | else
17   |   |   |   |   | level=level+1
18   |   |   |   | end
19   |   |   | end
20   |   |   | await ((n.promote[ch] = ⊥) ∨ n.lock = ⊥)
21   |   |   | if n.promote[ch] = ⊥ then
22   |   |   |   | promLevel=level
23   |   |   |   | await (spin[p]=true )
24   |   |   |   | spin[p]=false
25   |   |   |   | goto CS
26   |   |   | end
27   |   | end loop
28   | end
29   | CS: Critical Section
30   | n=$L_p$
31   | for level=1 to min(promLevel-1,$\Delta - 1$) do
32   |   | n=parent(n)
33   |   | PromAndRel(n,p)
34   | end
35   | if EMPTY(promQ) then
36   |   | q=r.lock
37   |   | PromAndRel(n,q)
38   | else
39   |   | spin[promQ.Deq()]=true
40   | end
41 end loop
```

Figure 3: Procedure PromAndRel Performed by Process p

```
   Input: Node n, process-ID curOwner
1  local: $j_1$, $j_2$, proc int
2  $j_1$ = random(0, $\Delta$-1)
3  $j_2$ = n.nextToPromote
4  foreach $j \in \{j_1, j_2\}$ do
5    | if (proc=n.promote[j]) $\notin$ {p, ⊥} then
6    |   | if CAS(n.promote[j], proc, ⊥) then
7    |   |   | promQ.Enq(proc)
8    |   | end
9    | end
10 end
11 n.nextToPromote = (n.nextToPromote + 1) mod $\Delta$
12 CAS(n.lock, curOwner, ⊥)
```

When a process p climbs to node n, it *registers* at n by writing its ID to the entry of the *promote* array corresponding to the sub-tree from which p ascends to n.

When a process q exits the critical section, it performs both *randomized promotion* and *deterministic promotion* at each node whose lock it captured in its entry section. (if q itself was promoted in its entry section, then q did not capture the locks of all nodes on the path from its leaf to the root.) To perform randomized promotion at node n, q selects randomly and uniformly an entry of the *promote* array associated with n; if some process is registered at that entry, then q promotes p. If process p is registered at n when some process performs randomized promotion at n we say that p *participates in a lottery.* Clearly the probability that p is eventually promoted increases with the number of lotteries in which it participate. Moreover, we show that whenever p busy waits at a node, the number of lotteries it participates in is proportional to the number of RMRs it incurs. This is the basis to our proof (see Section 3) that randomized promotion ensures the sub-logarithmic expected RMR complexity of our algorithm.

Sub-logarithmic expected RMR complexity does not preclude starvation, however. In order to bound the *worst-case* RMR complexity and avoid starvation, our algorithm employs *deterministic promotion* in addition to randomized promotion. With deterministic promotion, a process performing the exit section inspects an entry of node n's *promote* array pointed at by a *nextToPromote* index field associated with n; if some process is registered at that entry, it is promoted. The *nextToPromote* index is incremented in a round robin manner every time deterministic promotion is attempted. As we show, this ensures that a process waiting at some node n will either capture n's lock, or will be promoted, after n's lock is released at most Δ times. This is the basis to our proof in Section 3 that deterministic promotion ensures our bound on worst-case RMR complexity. We now provide a more detailed description of the algorithm.

The Entry Section. The entry section consists of lines **6-28**. In iteration h of the *external entry loop* (lines **8-28**), a process tries to capture the lock of the level-h node on the path from its leaf to the root. Each node n has a *promote* array of size Δ, where entry *promote*[j] is used by a process ascending to n from its j'th child. Process p first ascends one level up the path from L_p to the root (line **10**). It then registers at n by writing its ID to the entry of *n.promote* (line **11**) corresponding to the sub-tree from which it ascended to n.[2] As we've already mentioned, once p is registered at n, it may be promoted by processes performing their exit section. Process p then executes the *internal entry loop* of lines **12-27** until it either captures n's lock or it is promoted. In each iteration of the internal entry loop, p first tries to capture n's lock by performing a *CAS* operation (line **13**). If the *CAS* succeeds, p tries to *un-register at n* by swapping out its ID (line **14**). If it succeeds to un-register, then p can climb up the tree and exits the current iteration of the external loop (line **15**). Otherwise, p has been promoted at n. It increments *p.level* (line **17**), so that it now stores the number of the lowest level whose lock was not captured by p.

If the *CAS* operation of either line **13** or line **14** fails,

[2]This is done by a *CAS* operation, since the *CAS* object presented in [12] does not support a write operation.

p busy-waits in line **20** until it either identifies that it was promoted (this occurs immediately if p previously failed at line **14**) or it reads $\perp$ from n's lock. If p was promoted (line **21**), then it records the lowest level whose lock it did not capture (line **22**) so that it knows which locks to release in its exit section. It then busy-waits until it is signalled (line **23**) and, once it is, resets its spin variable and proceeds to the critical section (lines **24**-**25**). If the condition of line **21** does not hold, then it must be that p read $\perp$ from $n.lock$ in line **20**. In this case, p proceeds to the next iteration of the internal loop.

The Exit Section. In the for-loop of lines **31**-**34**, process p ascends the path from its leaf up until the highest non-root node whose lock it captured in its entry section. At each node n along this path, p calls the *PromAndRel* procedure, which we describe shorty, in order to promote processes that may be waiting at n and to release n.

Process p releases every node that it captured in the course of the preceding entry section, with the exception of the root's lock, which is released only if the promotion queue is empty (lines **35**-**37**). If the queue is non-empty, the first process ID in it is dequeued and that process is signalled to enter the critical section (line **39**).

The *PromAndRel* procedure receives two parameters: a node n and the ID of its current owner *curOwner*. *PromAndRel* can promote up to two processes. In lines **2** and **3**, it sets two indexes – j_1 and j_2 – to the *n.promote* array; it then tries to promote up to two processes whose IDs are stored in the corresponding entries (lines **5**-**7**). Index j_1 is chosen uniformly at random from $\{0, \ldots, \Delta - 1\}$ and is used to perform randomized promotion. Index j_2 is used to perform deterministic promotion. It is set to the value of node n's *nextToPromote* field. The *nextToPromote* field is incremented in a round-robin manner (line **11**). To promote process *proc*, p first tries to swap-out *proc*'s ID from the corresponding entry of the *promote* array (line **6**). If it succeeds, p enqueues *proc* to a queue of promoted processes (line **7**). Finally, *PromAndRel* releases node n's lock (line **12**).

3. RANDOMIZED ALGORITHM CORRECTNESS AND COMPLEXITY

We use the following notation in the proofs that follow. We say that *process p captures the lock of node n* whenever it applies a successful CAS operation on *n.lock* in line **13** of the entry section. We say that *process p releases the lock of node n* whenever it sets the value of *n.lock* to $\perp$ in Line **12** of *PromAndRel*. We say that *process p is promoted* whenever some process enqueues p to *promQ* in line **7** of *PromAndRel*. We say that *process p is signalled* whenever some process sets the spin flag of process p in line **39** of the exit section. We say that process p is *registered at n* if p's ID is written at an entry of *n.promote*. We say that an RMR is a *lock-change RMR* if it is caused by a read of *n.lock* in line **20** of the entry section.

3.1 Mutual Exclusion

We now provide that our randomized algorithm satisfies mutual exclusion. The intuition underlying these proofs is simple: a process p leaves the exit section either after executing line **37** or after executing line **39**. In the latter case, the root node remains captured (not necessarily by p) and the first process in the promotion queue is signalled and will eventually enter the critical section. In the former case, since the promotion queue is empty, processes can enter the critical section only after capturing the root's lock in line **13** of the entry section; the use of the CAS operation guarantees that at most a single process succeeds in doing that.

We use the following notation in the proofs that follow. We say that *process i captures the lock of node n* whenever it applies a successful CAS operation on *n.lock* in line **13**. We say that *process i releases the lock of node n* whenever it sets the value of *n.lock* to $\perp$ in line **12** of the *PromAndRel* procedure. We say that *process p is promoted* whenever some process enqueues p to *promQ* in line **7** of *PromAndRel*. We say that *process p is signalled* whenever some process sets the spin flag of process p in line **39**. We say that process p is *enabled to execute line i*, and write $p@i$, if i is the next line that p will execute when scheduled. We write $p@[i - j]$ to indicate that process p is enabled to execute some line in the range $[i - j]$. We let r denote the root node of the tree represented by the *nodes* array. We let *SigProcs(E)* denote the set of processes that have been signalled in an execution E but have not yet entered the critical section after being signalled. We simply write *SigProcs* when E is understood.

Lemma 1 *The following invariants hold after any execution E of the algorithm.*

$$
\begin{aligned}
(r.lock = \perp) \quad &\Longrightarrow \quad |\{p | p@[\textbf{29} - \textbf{39}]\}| = 0. && (1)\\
|\{p | p@[\textbf{29} - \textbf{39}]\}| \quad &\leq \quad 1. && (2)\\
|SigProcs(E)| \quad &\leq \quad 1. && (3)\\
|SigProcs(E)| = 1 \quad &\Longrightarrow \quad |\{p | p@[\textbf{29} - \textbf{39}]\}| = 0\\
&\qquad\quad \wedge r.lock \neq \perp. && (4)
\end{aligned}
$$

We claim that the only lines that may violate Invariants 1-4 are lines **15**, **25**, line **12** of *PromAndRel* (when called from line **37**), and **39**. A process enters the critical section either after line **25** is executed or after line line **15** is executed when *level*=Δ holds. It follows that the execution of either one of these two lines may violate Invariants 1, 2 and 4. When *PromAndRel* is called from line **37**, line **12** of *PromANdRel* changes *r.lock* to $\perp$ hence Invariants 1 and 4 may be violated. Finally, line **39** increments $|SigProcs|$, hence invariants 3 and 4 may be violated.

Observe that all invariants hold vacuously before execution starts and that no line other than the aforementioned can violate any of the invariants. We proceed by proving that none of the above lines can violate the invariants.

Claim 1 *The execution of line* **15** *does not violate the invariants.*

Proof. Let s be the step in which process p enters the critical section by executing line **15**, when *level*=Δ, and assume the invariants hold before s occurs. Process p can execute line **15** only after applying a successful *CAS* operation in line **13**; let s' denote this step. Since *level*=Δ when s' occurs, it follows that $r.lock = \perp$ just before s' and $r.lock = p$ immediately after s'. From the induction hypothesis applied to Invariant 1, this implies in turn that, right after s' occurs, there are no processes in the critical or exit sections. Also, from the induction hypothesis applied to Invariants 3-4, *SigProcs* is empty right after s'. It follows that during the time interval starting with s' and ending just before s, there is no process in the entry or exit sections. This implies that, right after s, p is the single process in the entry or exit sections and *SigProcs* is empty, hence all invariants hold after s. $\square$

Claim 2 *The execution of line* **25** *does not violate the invariants.*

PROOF. Let s be the step in which process p enters the critical section by executing line **25** and assume the invariants hold before s occurs. Since p can execute line **25** only after the condition of line **23** is satisfied, p was signalled by some process q. From the induction hypothesis applied to Invariants 3 and 4, when p is signalled it is the only member of *SigProcs*, $r.lock \neq \perp$ holds, and there are no processes in the critical or exit sections. It follows that no process is in the critical or exit sections in the time period starting when p is signalled by q and ending just before s occurs. Hence, right after s, p is the only process in the critical section and $\big((r.lock \neq \perp) \wedge |SigProcs(E)| = 0\big)$ holds. Therefore all invariants hold after s. $\square$

Claim 3 *The execution of line* **12** *of the* PromAndRel *procedure does not violate the invariants.*

PROOF. Clearly, line **12** does not violate the invariants when *PromAndRel* is called from line **33** of the algorithm, so we only need to consider calls from line **37** of the algorithm. Let s be the step in which process p frees the root's lock in line **12** and assume the invariants hold before s occurs. Observe that $p@[\mathbf{29}-\mathbf{39}]$ holds just before s occurs. From the induction hypothesis applied to Invariants 2-4, p is the only process in the critical or exit sections and *SigProcs* is empty right before s occurs. Hence, right after s, there are no processes in the entry or exit sections, *SigProcs* is empty and $r.lock{=}\perp$, implying that all invariants still hold. $\square$

Claim 4 *The execution of line* **39** *does not violate the invariants.*

PROOF. Let s be the step in which process p signals process q in line **39** and assume the invariants hold before s occurs. Observe that the test of line **5** of *PromAndRel* guarantees that $p \neq q$ holds and that $p@[\mathbf{29}{-}\mathbf{39}]$ holds just before s occurs. From the induction hypothesis applied to Invariants 1, 3 and 4, p is the only process in the critical or exit sections and $\big((r.lock \neq \perp) \wedge (|SigProcs| = 0)\big)$ immediately before s occurs. Hence, right after s, there are no processes in the entry or exit sections and $\big((r.lock \neq \perp) \wedge (|SigProcs| = 1)\big)$ holds, implying that all invariants still hold. $\square$

Mutual exclusion follows immediately from Claims 1-4.

Lemma 2 *The randomized algorithm satisfies mutual exclusion. Moreover, there is always at most a single process in the critical or exit sections.*

3.2 Starvation-Freedom and Worst-Case RMR Complexity

Claim 5 *A process incurs $O(\log N/\log\log N)$ RMRs in the exit section (lines 30-39).*

PROOF. In the exit section, a process performs a constant number of RMRs in lines **35-39** and less than $\Delta{=}O(\log N/\log\log N)$ iterations of the loop of lines **31-34**, in each of which it performs a constant number of RMRs. $\square$

Claim 6 *Node locks cannot be recaptured before they are released. Moreover, the lock of any non-root node can only be released by the process that captured it.*

PROOF. Since node locks can only be captured by a CAS operation (line **13** of the entry section) that succeeds only if the lock's current value is $\perp$, once a lock is captured by some process p, it cannot be re-captured before it is released. Non-root locks are released by *PromAndRel* when it is called in line **33** of the exit section, which proves the second part of the claim. $\square$

Claim 7 *Assume that process p starts executing an iteration of the external entry loop at time t_0, trying to capture node n's lock. Assume also, that by time $t_1 > t_0$, p incurs z lock-change RMRs and is still executing the same iteration. Then n's lock was released at least $\lfloor z/2 \rfloor$ times during time interval $[t_0, t_1]$.*

PROOF. In the CC model, whenever p incurs a lock-change RMR, the value of *n.lock* has been modified by another process since last read by p. From Claim 6, a lock cannot be re-captured before it is released. It follows that n's lock is released at least once between any two lock-change RMRs incurred by p. $\square$

Claim 8 *Let p be a process performing iteration j of the external entry loop, for some $j \in \{1, \ldots, \Delta\}$, trying to capture node n's lock. Let t be the time when p starts executing the internal loop in iteration j. Then after n's lock is released at most Δ times after t (if it ever is), p has either already terminated iteration j, or otherwise will do so within a constant number of RMRs.*

PROOF. Let m be the node from which p ascends to node n and let *ch* denote the ordinal number of m among n's children. Process p writes its ID to $n.promote[ch]$ (in line **11**) before it starts executing the internal entry loop. Since m is not the root-node, Claim 6 guarantees that no other process will capture m's lock before p releases it. It follows that $n.promote[ch]{=}p$ holds as long as p is not promoted and does not capture n's lock.

Every process that releases n's lock tries to deterministically promote a process registered at n in entry *n.nextToPromote* of the *promote* array (lines **3-6** of *PromAndRel*). Observe that, from Lemma 2, the execution of the exit section is sequential. Since *n.nextToPromote* is incremented modulo Δ every time n's lock is released (line **11** of *PromAndRel*) and is never modified elsewhere, some process will write $\perp$ to $n.promote[ch]$ in line **6** of *PromAndRel* by the time n's lock is released at most Δ times after t, unless p is no longer registered at n by then. If p has already finished executing the internal entry loop by time t, the claim clearly holds. Otherwise, p proceeds to execute lines **22**–**25** of the entry section and will terminate iteration j within a constant number of RMRs. $\square$

Claim 9 *A process incurs $O(\log N/\log\log N)$ RMRs as it performs a single iteration of the external entry loop.*

PROOF. Let n be the node p tries to capture in iteration j of the external entry loop. From Claims 7 and 8, p incurs a total of $O(\log N/\log\log N)$ lock-change RMRs during iteration j. Moreover, since in each iteration of the internal entry loop, except for possibly the last, p incurs such an

RMR, it follows that the number of internal loop iteration performed by p in iteration j is also $O(\log N/\log\log N)$. Finally observe that p can incur at most a single RMR in line **20** on account of reading *n.promote*$[ch]$, since, once it does, p proceeds to wait for a signal (in line **23**) and then performs at most a constant number of additional RMRs before it finishes executing the external loop iteration. The claim follows. $\square$

Claim 10 *A process incurs $O\big((\log N/\log\log N)^2\big)$ RMRs in the entry section.*

PROOF. Immediate from Claim 9 and the fact that p executes at most $\Delta = O(\log N/\log\log N)$ external entry loop iterations. $\square$

From Claims 5 and 10, we get the following lower bound on the algorithm's worst-case RMR complexity.

Lemma 3 *The algorithm has $O\big((\log N/\log\log N)^2\big)$ worst-case RMR complexity.*

Claim 11 *Let p be a process executing iteration j of the external entry loop, trying to capture the lock of node n and let ch denote the entry of n.promote corresponding to p. Then, when p exits iteration j, n.promote$[ch] \neq p$ holds.*

PROOF. Process p can exit iteration j only from lines **15** or **25**. From Claim 6, no other process can write its ID to *n.promote*$[ch]$ before p exits iteration j. Process p can execute line **15** only after it succeeds in setting *n.promote[ch]=*$\perp$ in line **14**. Also, p can execute line **25** only after the condition of line **21** is satisfied, implying that *n.promote[ch]=*$\perp$ holds. Finally observe that no process other than p ever writes value p to an entry of the *promote* array. $\square$

Claim 12 *If a process p is promoted while performing iteration j of the external entry loop on some node n, then p eventually busy-waits at line* **23** *in the course of performing that iteration.*

PROOF. Let q be the process that promoted p (in line **7** of *PromAndRel*) and let ch be the entry of *n.promote* corresponding to p. Before executing line **7**, q must perform a successful *CAS* that swaps the value of *n.promote*$[ch]$ from p to $\perp$ in line **6** of *PromAndRel*. From Claim 11, this implies that p is still in iteration j when that *CAS* occurs. It follows that when p next executes line **20**, *n.promote*$[ch]$=$\perp$ holds and p proceeds to execute lines **21-23**. $\square$

Lemma 4 *The algorithm is starvation-free.*

PROOF. Observe that Lemma 3 does not guarantee starvation-freedom, since, theoretically, a process may be stuck forever in a busy-waiting loop without incurring RMRs and without making progress. We prove that this does not happen. A process performs busy-waiting loops only in lines **20** and **23**. The proof proceeds (and concludes) by considering each of these cases separately.

Claim 13 *A process cannot be stuck forever in the busy-waiting loop of line 23.*

PROOF. Clearly from the code, process p executes line **23** only if the condition of line **21** was satisfied, implying that p was promoted by some process q in line **6**of *PromAndRel.* Immediately after that, q enqueues p to the promotion queue in line **7** of *PromAndRel.* Let t be the time when this occurs, let k, for some $1 \leq k \leq N-1$, be the position of p in the promotion queue and let $q_1, \ldots, q_{k-1}$ denote the processes that precede p in the queue at time t.

From Lemma 5, the operations on the promotion queue (in line **7** of *PromAndRel* and in line **39** of the exit section) are performed sequentially and queue semantics are maintained. It follows from the test of line **35** that, starting from time t, the root's lock remains captured until the promotion queue becomes empty. Also, it follows from Claim 12, that every signalled process eventually enters the critical section. Thus, processes $q_1, \ldots, q_{k-1}$ enter and exit the critical section one after the other and eventually q_{k-1} signals p in line **39**. $\square$

Claim 14 *A process cannot be stuck forever in the busy-waiting loop of line 20.*

PROOF. Suppose, by way of contradiction, that the algorithm has an infinite execution E in which a set of processes $\mathcal{P}$ are stuck forever in line **20**. Let $p \in \mathcal{P}$ be a process that is stuck in line **20** on a node n with maximum level h and assume *n.lock=*q when p busy-waits in line **20**. Since process q succeeded in capturing node n's lock, and from our maximality assumption, q does not get stuck in line **20** when climbing up from n. From Claim 13, q does not get stuck in line **23** either. It follows that q eventually enters and then exits the critical section.

Thus, q eventually releases *n.lock.* Let t be the time when this occurs and consider the first time after t when p reads *n.lock* in line **20**. If *n.lock=*$\perp$, p will immediately exit the busy-waiting loop. Assume, then, that p reads the ID of another process. In this case, p incurs an RMR. However, from Claim 9, this can only happen $O(\log N/\log\log N)$ times. This is a contradiction. $\square$

3.3 Expected RMR Complexity

Claim 15 *Assume process p starts executing iteration j of the external entry loop, for some $j \in \{1, \ldots, \Delta\}$, at time t_0, trying to capture the lock of node n; let $t_1 > t_0$ and let $I = [t_0, t_1]$. Also, let $z(I)$ denote the number of RMRs incurred by p in I, and let $s(I)$ denote the number of lock-change RMRs incurred by p during I. Then there exist constants c_1, c_2 such that $z(I) \leq c_1 \cdot s(I) + c_2$ holds.*

PROOF. Observe that the number of non-lock-change RMRs incurred in each iteration of the internal entry loop is constant. This is because at most a single RMR will be incurred by p on account of reading *n.promote*$[ch]$ in line **20**; after such an RMR is incurred, p executes lines **22-25** and terminates iteration j within a constant number of RMRs. If, in the same internal loop iteration, p incurs an RMR in line **13** but does not incur a lock-change RMR, then p will incur a lock-change RMR in the next internal loop iteration (if there is one). Since the number of RMRs incurred in iteration j outside the internal entry loop is constant, the claim follows. $\square$

Lemma 5 *The expected number of RMRs performed in the course of the algorithm's entry and exit sections is $O(\log N/\log\log N)$.*

Proof. From Claim 5, the number of RMRs performed in the course of the exit section is $O(\log N/\log\log N)$. It thus suffices to prove the claim for the entry section.

Assume process p starts executing the entry section at time t_0, let $t_1 > t_0$ and let $I = [t_0, t_1]$. Also, let $s(I)$ denote the number of lock-change RMRs incurred by p in I and let $z(I)$ denote the total number of RMRs performed by p in I. Applying Claim 15 for all the iterations of the external entry loop and summing up, we get:

$$z(I) = O\big(s(I) + \log N/\log\log N\big). \tag{5}$$

It thus suffices to prove that $E\big[s(I)\big] = O(\log N/\log\log N)$ holds.

We say that process p *participates in a lottery*, whenever some process in its exit section tries to randomly promote (in lines **2** and **4**-**7** of *PromAndRel*) a process on some node n while p is registered at n. Let $r(I)$ denote the number of times during I in which p participates in a lottery. Since a process tries to perform randomized promotion on every node whose lock it releases, we get from Claim 7:

$$r(I) = \Omega\big(s(I)\big). \tag{6}$$

We let G_x denote a geometrically distributed (with probability x) random variable. Whenever p participates in a lottery, it has probability $1/\Delta$ of being promoted regardless of the behavior of an adversarial scheduler. Moreover, once p is promoted, it will not participate in additional lotteries. We thus have:

$$E\big[r(I)\big] \le E\big[G_{1/\Delta}\big] = \frac{1}{1/\Delta} = \Delta = O(\log N/\log\log N). \tag{7}$$

Where the inequality above follows from the fact that, from Lemma 3 and Claim 7, a process participates in at most $O\big((\log N/\log\log N)^2\big)$ lotteries. The lemma now follows from Equations 5 - 7.

$\square$

From Lemmas 2-5 we get:

Theorem 1 *The randomized algorithm is a correct starvation-free mutual exclusion algorithm and has $O(\log N/\log\log N)$ expected RMR complexity and $O\big((\log N/\log\log N)^2\big)$ worst-case RMR complexity.*

4. THE COMBINED ALGORITHM

The randomized algorithm presented in Section 2 (henceforth the *non-quitting algorithm*) has $O\big((\log N/\log\log N)^2\big)$ worst-case RMR complexity, which is sub-optimal. In this section, we present a variant of that algorithm, henceforth called the *quitting algorithm*, that can be combined with a deterministic algorithm such as [18]. The resulting *Combined Randomized Deterministic* (henceforth CRD) algorithm achieves optimal worst-case RMR complexity of $O(\log N)$ while maintaining expected RMR complexity of $O(\log N/\log N\log N)$.

The entry code and exit code of the quitting algorithm are encapsulated within procedures *REnter* and *RExit*, respectively. The pseudo-code of these procedures is presented in Figures 4 and 5. The key difference between the quitting and non-quitting algorithms is the following. Whereas a process p is either promoted or succeeds in capturing the root node's lock in the entry code of the non-quitting algorithm, a third option exists in the entry code of the quitting algorithm: p *quits* executing *REnter* if it incurs $\Theta(\log N)$ RMRs. In this case, as we explain soon, p has to execute the entry code of a deterministic algorithm before it can enter the CS. We now describe the quitting algorithm in more detail. We then describe the CRD algorithm.

REnter. Similarly to the non-quitting algorithm, a process p, executing the entry section of the quitting algorithm, may enter the critical section after capturing the locks of all nodes on its path to the root (in line **20**, when *level*=Δ) or after it is signalled (in line **39**). Unlike the non-quitting algorithm, p may also quit *REnter* (by returning *FALSE* in line **32**) when the value of the *p.RMRsNum* local variable exceeds $\log N$ (line **29**).

The internal entry loop of the non-quitting algorithm was changed so that a process can maintain an asymptotically-accurate estimate of the number of RMRs it incurs. This estimate is stored in the *RMRsNum* local variable. To facilitate that, the structure of the *lock* field was changed and now consists of a pair of values <*i,nextToPromote*>, where i is either a process ID or $\perp$, and *nextToPromote* has the same function as in the non-quitting algorithm. As we prove in the full paper, since the *nextToPromote* field is incremented (modulo Δ) each time a process releases node n's lock, an argumentation similar to that used in the proof of Claim 8 establishes that the inequality of line **11** holds every time the lock's value changed since last read by p.

The structure of the internal loop was also changed in order to eliminate the *await* of line **20** of the non-quitting algorithm. This is because a process may incur $\Theta(\log N/\log\log N)$ RMRs while executing line **20** of that algorithm.

When the value of *p.RMRsNum* exceeds $\log N$, p tries to un-register at n by swapping out its ID (line **30**). If it succeeds, then p records the level in which it quits (line **31**) so that later, when it executes the exit section of the quitting randomized algorithm (*RExit*), it can release all the locks it captured on its path up to that level. Process p then returns *FALSE* to indicate that it failed to enter the CS; this will then trigger a call to the entry section of the deterministic algorithm. If the *CAS* of line **30** fails, then p was already promoted at n. It will then return from *REnter* successfully in line **39**.

RExit. Since process p may have returned from *REnter* before capturing all locks on its path either because it was promoted or because it quitted *REnter*, the loop of lines **2**-**5** ascends only up until the highest node on p's path to the root that was captured by p in *REnter* (see lines *REnter*:**31**, *REnter*:**36** and *RExit*:**2**).

Similarly to the non-quitting algorithm, randomized and deterministic promotions are always applied to a node before its lock is released. Both promotions and lock release are performed by the *CRDPromAndRel* procedure, whose pseudo-code appears in Figure 6.

Unlike the non-quitting exit section, and as mentioned before, *CRDPromAndRel* releasees a lock by writing to it the pair of values <$\perp$,j_2> (line *CRDPromAndRel*:**12**), where j_2 is the updated *nextToPromote* index.

Figure 4: Procedure REnter for Process p

```
   Output: FALSE if quits, TRUE otherwise.
1  local: RMRsNum: int init 0,
          owner, nextToPromote: int
3         highestLevel=$\Delta + 1$
4  n=$L_p$
5  for level=1 to $\Delta$ do
6      Let ch be the integer such that n is the (ch+1)'th
       child of parent(n)
7      n=parent(n)
8      CAS(n.promote[ch],$\perp$,p)
9      <owner,nextToPromote>=n.lock
10     loop
11         if n.lock $\neq$ <owner,nextToPromote> then
12             RMRsNum=RMRsNum+1
13             <owner,nextToPromote>=n.lock
14         end
15         if owner=$\perp$ then
16             if CAS(n.lock,
               <$\perp$, nextToPromote>,
               <p, nextToPromote>)
17             then
18                 if CAS(n.promote[ch],p,$\perp$) then
19                     if (level=$\Delta$) then
20                         return TRUE
21                     else
22                         continue for
23                     end
24                 else
25                     level=level+1
26                 end
27             end
28         end
29         if RMRsNum > $\log N$ then
30             if CAS(n.promote[ch],p,$\perp$) then
31                 highestLevel=level
32                 return FALSE
33             end
34         end
35         if (n.promote[ch] = $\perp$) then
36             highestLevel=level
37             await spin[p]=true
38             spin[p]=false
39             return TRUE
40         end
41     end loop
42 end
```

Figure 5: Procedure RExit for Process p

```
1  n=$L_p$
2  for level=1 to min(highestLevel-1, $\Delta - 1$) do
3      n=parent(n)
4      CRDPromAndRel(n)
5  end
6  if quitted then return
7  if EMPTY(promQ) then
8      CRDPromAndRel(r)
9  else
10     spin[promQ.Deq()]=true
11 end
```

Figure 6: Procedure CRDPromAndRel Performed by Process p

```
   Input: Node n
1  local: $j_1$, $j_2$, prev, tmp, proc int
2  $j_1$ = random(0, $\Delta$-1)
3  prev=<tmp,$j_2$>=n.lock
4  foreach $j \in \{j_1, j_2\}$ do
5      if (proc=n.promote[j]) $\notin$ {p, $\perp$} then
6          if CAS(n.promote[j], proc, $\perp$) then
7              promQ.Enq(proc)
8          end
9      end
10 end
11 $j_2 = j_2 + 1 \bmod \Delta$
12 CAS(n.lock, prev, <$\perp$,$j_2$>)
```

Figure 7: The CRD Algorithm for Process p

```
1  define Node: struct {lock: int init <$\perp$, 0>, promote:
   array [0..$\Delta - 1$] of int init $\perp$}
2  shared: $\mathcal{T}$: Complete $\Delta$-ary tree with leaves
   $L_1, \ldots, L_N$, promQ: Queue,
   spin: array [1..N] of boolean init false
3  local: n: Node, ch, next: int, highestLevel: int, quitted:
   int
4  loop
5      Non Critical Section
6      quitted=REnter()
7      if quitted then
8          DEnter()
9          TwoPMutexEnter(RIGHT)
10     else
11         TwoPMutexEnter(LEFT)
12     end
13     CS: Critical Section
14     if ¬quitted then
15         TwoPMutexExit(LEFT)
16     else
17         TwoPMutexExit(RIGHT)
18         DExit()
19     end
20     RExit()
21 end loop
```

Finally, if p quitted *REnter* (as indicated by the *quitted* flag set in line *CRD*:**6**), then it must not release the root node's lock, nor should it signal a process (line *RExit*:**6**).

CRD Algorithm. The main code of the CRD algorithm is shown in Figure 7. In addition to the quitting randomized algorithm, it uses two deterministic mutual-exclusion implementations. An $O(\log N)$ RMRs N-process implementation (such as [18]), whose entry and exit sections are encapsulated within the *DEntry* and *DExit* procedures, respectively; and an $O(1)$ RMRs 2-process deterministic implementation (such as a 2-process instance of [18]), whose entry and exit sections are encapsulated within the *TwoPMutexEnter* and *TwoPMutexExit* procedures, respectively. Figure 8 depicts the structure of the entry and exit sections of the CRD algorithm.

One may think that combining our randomized algorithm with a deterministic algorithm should not be hard: each process can alternate between executing steps of the two al-

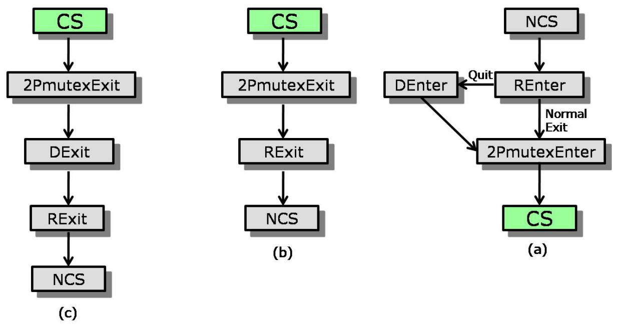

Figure 8: The structure of the CRD algorithm: (a) entry section, (b) "normal" exit section, (c) exit section in case the process quitted in its respective entry section.

gorithms; once it captures the lock of one of them, it can abort from the other. This approach does not seem to work. To abort from the randomized algorithm, a process p must release all the locks that it captured, thus possibly causing processes that are waiting at these nodes to incur RMRs; however, as p is not holding the root's lock, it cannot promote other processes, and so sub-logarithmic expected RMR complexity cannot be ensured.

Our solution to this problem is that processes *quit* the randomized algorithm rather than aborting: a process that quits the randomized algorithm enters the CS through the deterministic algorithm. Only when it exits the CS does it unlock the locks it captured before quitting. We now describe the CRD algorithm in more detail.

The entry section of the CRD algorithm consists of lines **6-11**. To enter the critical section, a process first executes the entry section of the quitting algorithm (line **6**). Processes that do not quit *REnter* then call the entry section of the 2-process algorithm (line **11**), playing the role of the left-hand process in that algorithm.

Processes that do quit *REnter* need to capture the lock of the N-process deterministic algorithm (line **8**). Once they do, they call the entry section of the 2-process algorithm (line **9**), playing the role of the right-hand process in that algorithm. As we prove in the final paper, the 2-process mutual-exclusion algorithm is always used by at most a single "left-hand" process (a process that did not quit *REnter*) and at most a single "right-hand" process (a process that did quit *REnter* and holds the lock of the deterministic algorithm).

The exit section of the CRD algorithm consists of lines **14-20**, where a process simply performs in reverse order the exit sections of the algorithms whose entry sections it executed in lines **6-11**. A process that did not quit *REnter*, first executes the exit section of the 2-process algorithm (playing the role of the left-hand process, in line **15**) and then executes *RExit* (line **20**). A process that did quit *REnter*, executes the exit section of the 2-process algorithm (playing the role of the right-hand process, in line **17**), followed by the exit sections of the deterministic (line **18**) and randomized (line **20**) algorithms.

Correctness. Correctness proofs for the CRD algorithm appear in the full paper. Proving starvation-freedom and expected RMR complexity for the CRD algorithm is quite similar to proving these properties for the non-quitting randomized algorithm (see Section 3). The mutual exclusion proof for the CRD algorithm relies also on the properties of the deterministic N-process and 2-process algorithms and the manner in which they are combined.

The worst-case complexity proof for the CRD algorithm relies on the fact that the number of RMRs incurred by a process in the entry section of the quitting randomized algorithm is $O(\log N)$ as long as the value of the $RMRsNum$ variable does not exceed $\log N$. The proof of the following theorem appears in the full paper.

Theorem 2 *The CRD algorithm is a correct starvation-free mutual exclusion algorithm and has $O(\log N/\log\log N)$ expected RMR complexity and $O(\log N)$ worst-case RMR complexity.*

5. MODIFICATIONS REQUIRED FOR THE DSM MODEL

We now describe how our CC algorithms can be modified for the DSM model. In this model, processes that try to capture a node-lock cannot busy-wait on the lock without incurring an unbounded number of RMRs. Instead, they must 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 by using a synchronization mechanism we call a *wait-signal object* (WS object). WS objects for both the CC and the DSM models were introduced in [12].[3] WS objects support operations *wait* and *signal*. Consider a WS object W. Each process p can call $W.wait(q)$ once for each process $q \neq p$ during an execution, and it can call $W.signal$ once. The semantics and complexity of WS objects are as follows:

1. The call $W.wait(q)$ by process $p \neq q$ does not terminate before process q calls the $W.signal$ operation.
2. Each $W.signal$ function call terminates within a constant number of steps, and once process q has finished a $W.signal$ call, p's function call $W.wait(q)$ terminates within a constant number of steps.
3. Each function call $W.wait(q)$ or $W.signal$ incurs a constant number of RMRs.

We now describe how the algorithm presented in Section 2 is modified for the DSM model so that the expected and worst-case RMR complexities are maintained. Similar modifications can be made to the combined algorithm described in Section 4.

In the algorithm given in Figure 1, a process p can busy-wait only in line **20** or line **23**. 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.

If each variable $spin[p]$ is allocated in p's local memory segment, then Line **23** incurs no RMRs in the DSM model. In order to deal with line **20**, we associate with each node n a WS object $n.W$. The **await**-operation in line **20** is replaced by a call of $n.W.wait(q)$, where q is the process

[3]Note that the WS objects described in [12] are one-time but they can easily be made long-lived.

ID returned from the CAS-operation in line **13**.[4] When p releases a lock, it has to notify waiting processes by calling $n.W.signal$. Thus, this operation is added after the CAS-operation in line **12** of the *PromAndRel* procedure.

With root-locks we have to be a bit more careful, though: since processes now wait for a signal from a specific process, we have to make sure that the process that calls *signal* actually owns the root-lock. We ensure this by *handing over* the root-lock to a promoted process, after performing the promotion in line **39**. More precisely, the **else**-part of line **39**, is replaced by the following 3 operations:

```
$proc = promQ.deq();$
$CAS(r.lock, p, proc);$
$spin[proc] = \mathbf{true}.$
```

Since the root-lock handing over may cause processes waiting on the root to incur RMRs, both randomized and deterministic promotions are also performed at the root node. The last operation of the exit-section is then to signal all processes that are possibly waiting for the root-lock to be released by calling $r.W.signal$.

These modifications have the following consequences. Consider a process p that fails to capture node n's lock because process q has the lock. Before p makes another attempt to capture the lock, it calls $n.W.wait(q)$ and therefore incurs a constant number of RMRs until q calls $n.W.signal$. However, q calls this function only after promoting a process chosen at random from the processes registered at n. Therefore, if p makes k attempts to capture node n's lock, then it necessarily participates in $\Omega(k)$ lotteries. Moreover, every time node n's lock is being released, the process owning that lock calls $n.W.signal$ (this now holds also for the root-lock, as the "handing over" mechanism ensures that the process executing the exit-section has the root-lock), thus ensuring that every $n.W.wait$-call terminates.

6. CONCLUSIONS

In this work we establish a separation in terms of RMR complexity, for both the CC and DSM models, between randomized and deterministic mutual exclusion algorithms that can use read, write, and conditional operations (such as compare-and-swap). We present two starvation-free randomized algorithms, in which processes incur $O(\log N/\log\log N)$ expected RMRs per passage. We also present the CRD algorithm, which combines our second randomized algorithm with an optimal-time deterministic mutual exclusion algorithm (such as [18]) and achieves optimal worst-case logarithmic RMR complexity while maintaining $O(\log N/\log\log N)$ expected RMR complexity.

We do not know whether our upper bound is tight. Moreover, we are also not aware of any non-trivial lower bound on the expected RMR complexity of randomized mutual exclusion. Resolving this gap remains an interesting open question.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their many helpful comments.

[4]We assume here that the DSM algorithm uses the stronger version of CAS presented in [12], that returns the previous object value rather than a success or failure indication.

8. REFERENCES

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proc. of the 13th IEEE Real-Time Systems Symposium*, pages 12–21, December 1992.

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

[3] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

[4] J. H. Anderson and Y.-J. Kim. Fast and scalable mutual exclusion. In *DISC'09*, pages 180–194, 1999.

[5] J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *DISC'00: Proceedings of the 14th International Conference on Distributed Computing*, pages 29–43, London, UK, 2000. Springer-Verlag.

[6] J. H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *PODC'02*, pages 3–12, New York, NY, USA, 2002. ACM.

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

[8] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[9] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *STOC '08: Proc. of the 40th annual ACM symposium on Theory of computing*, pages XX–YY, 2008.

[10] R. Danek and W. M. Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. In *DISC'08*, pages 93–108, 2008.

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

[12] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In *PODC'07*, pages 3–12, 2007.

[13] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[14] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC'03*, pages 295–304, New York, NY, USA, 2003. ACM Press.

[15] P. Jayanti, S. Petrovic, and N. Narula. Read/write based fast-path transformation for FCFS mutual exclusion. In *SOFSEM*, pages 209–218, 2005.

[16] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *DISC'01*, pages 1–15, London, UK, 2001. Springer-Verlag.

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

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