Adaptive Randomized Mutual Exclusion in Sub-Logarithmic Expected Time

Danny Hendler
Department of Computer-Science
Ben-Gurion University
Beer-Sheva 84105, Israel
hendlerd@cs.bgu.ac.il

Philipp Woelfel*
Department of Computer Science
University of Calgary
2500 University Dr. NW
Calgary, AB T2N1N4, Canada
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 mutual exclusion algorithm is *adaptive to point contention*, if its RMR complexity is a function of the maximum number of processes concurrently executing their entry, critical, or exit section.

In the best prior art deterministic adaptive mutual exclusion algorithm, presented by Kim and Anderson [22], a process performs $O\big(\min(k, \log N)\big)$ RMRs as it enters and exits its critical section, where k is point contention and N is the number of processes in the system. Kim and Anderson also proved that a deterministic algorithm with $o(k)$ RMR complexity does not exist [21]. However, they describe a *randomized* mutual exclusion algorithm that has $O(\log k)$ expected RMR complexity against an oblivious adversary. All these results apply for algorithms that use only atomic read and write operations.

We present a randomized adaptive mutual exclusion algorithms with $O(\log k/\log\log k)$ expected amortized RMR complexity, even against a strong adversary, for the cache-coherent shared memory read/write model. Using techniques similar to those used in [17], our algorithm can be adapted for the distributed shared memory read/write model. This establishes that sub-logarithmic adaptive mutual exclusion, using reads and writes only, is possible.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

*Supported by NSERC

PODC'10, July 25–28, 2010, Zurich, Switzerland.

General Terms

Algorithms, Theory

Keywords

Adaptive, Mutual Exclusion, Remote Memory References, RMRs, Distributed Algorithms

1. INTRODUCTION

In the *mutual exclusion* problem, a set of processes must coordinate their accesses 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 [12], the mutual exclusion problem is at the core of Distributed Computing and is still the focus of intense research [4, 23]. We say that a mutual exclusion algorithm is *adaptive to point contention* (or simply *adaptive*), if its time complexity is a function of the number of processes concurrently executing their entry, critical or exit section. In this paper, we consider adaptive mutual exclusion algorithms in the asynchronous shared-memory model with atomic read and write registers.

A natural time complexity measure for algorithms in this model is *step complexity*, which counts the number of shared-memory accesses performed by processes. However, this measure is problematic for mutual exclusion algorithms because a process may perform an unbounded number of steps while busy-waiting for another process that blocks the critical section. An alternative complexity measure for mutual exclusion algorithms is the *remote memory references* (RMRs) metric (see for example Anderson and Kim [5]), which only counts 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, as pointed out, for example, by Anderson [7]. Local-spin mutual exclusion under the RMR metric has been an active area of research over the last 15 years or so (see [3, 6, 11, 19, 20, 21, 22] for some examples).

1.1 Related Work

Several deterministic read/write mutual exclusion algorithms that are local-spin for the distributed shared memory (DSM) model and for (at least some variants of) the *cache coherent* (CC) model and are adaptive to point contention are known. (See Section 1.4 for a description of

the DSM and the CC models.) Afek, Stupp, and Touitou presented an algorithm for the CC model in which a process performs $O(k^4)$ RMRs per *passage* (entry and exit to the critical section), where k is maximum point contention [1]. They later presented an $O(k^2)$ RMRs mutual exclusion algorithm [2]. Attiya and Bortnikov [9] improved on that by presenting an $O(k)$ RMRs algorithm for the CC model. None of these three algorithms are local-spin in the *distributed shared memory* (DSM) model.

Kim and Anderson [22] were the first to present an adaptive local-spin algorithm for both, the CC and the DSM model. Their algorithm has $O\big(\min(k, \log N)\big)$ RMR complexity, where N is the number of processes in the system. No adaptive deterministic algorithm with better RMR complexity is known and it was conjectured by Kim and Anderson [22] that this is best possible. Jayanti presented an *abortable* algorithm that has the same RMR complexity in the DSM model and $O(k)$ RMR complexity in the CC model [19].

Kim and Anderson [21] proved that every deterministic N-process mutual exclusion algorithm has an execution of point contention $k = \Theta(\log \log N)$, in which some process incurs $\Omega(k)$ RMRs, thus precluding deterministic adaptive mutual exclusion with $o(k)$ RMRs.

The best (non-adaptive) deterministic local-spin mutual exclusion algorithm is due to Yang and Anderson [24] and has $O(\log N)$ RMR complexity. Anderson and Kim [5] conjectured that this is best possible. This conjecture was proved by Attiya, Hendler, and Woelfel [10], building on a prior $\Omega(\log N)$ lower bound by Fan and Lynch [13] for a related but different cost model.

1.2 Randomized Mutual Exclusion

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 [8]. 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 only after the process has taken a step following that coin flip. A more detailed description of the adversary models follows in Section 1.4.

Kim and Anderson [21] were the first to employ randomization in order to improve the RMR complexity of mutual exclusion algorithms. They presented a simple randomized variant of their (deterministic) read/write adaptive mutual exclusion algorithm, where, if point contention is at most k, each process incurs only an expected number of $O(\min\{\log k, \log N\})$ RMRs per passage. This contrasts their lower bound of $\Omega(k)$ for deterministic algorithms (and $k = \Theta(\log \log k)$). The adversary model is not discussed in [21], but it can be shown that their algorithm achieves the claimed upper bound on the RMR complexity for the oblivious but not the weak or strong adversary.

Hendler and Woelfel [16] recently presented randomized mutual exclusion algorithms for both the CC and DSM read/write models with expected $O(\log N / \log \log N)$ RMR complexity against the strong adversary. Their algorithm uses atomic *compare and swap* (CAS) and read/write registers, but Golab, Hadzilacos, Hendler, and Woelfel [15] (see also [14]) presented a constant-RMRs implementation of CAS objects, using atomic read- and write operations. From these implementations it follows that the randomized algorithm of [16] has expected RMR complexity $O(\log N / \log \log N)$ against a strong adversary.

1.3 Our Contributions

We establish that adaptive mutual exclusion in sublogarithmic expected amortized RMR complexity is possible with atomic read/write registers. More specifically, by extending the techniques of [16], we present an adaptive starvation-free mutual exclusion algorithm that uses atomic read, write, and CAS operations. If point-contention is k, then against a weak adversary each process incurs an expected number of $O(\log k / \log \log k)$ RMRs per passage through the critical section. Against a strong adversary the expected *amortized* RMR complexity is also $O(\log k / \log \log k)$ RMRs. The worst-case RMR-complexity of our algorithm is $O(\min(k \log k, \log N))$. The amortized and worst-case bounds hold even if atomic CAS operations are replaced by the linearizable $O(1)$ RMRs CAS implementation of [15]. This establishes that mutual exclusion can be solved using reads and writes only, with an expected amortized RMR complexity of $O(\log k / \log \log k)$, even against the strong adversary.

1.4 Model

Our model of computation is based on [18]. 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.

The *compare-and-swap* (CAS) operation 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, the operation 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 the one implemented in [15], returns the value of v just before CAS is applied, instead of returning **true** or **false**.

We say that a process is *active*, if it is poised to perform a step in its entry, critical, or exit section. When some some processes are active, an *adversary* selects which of them will be allowed to perform its next step. We assume a model in which processes do not fail stop outside their remainder section. That is, if a process is active, then the adversary must

eventually allow it to perform its next step. A randomized algorithm may make random selections (often called *coin-flips*) for determining what next step to take. A *strong adversary* [8] can let a process p flip a coin and then take the result of that coin flip into account in order to decide whether p or some other process is allowed to take its next step. In a *weak adversary* [8] model, on the other hand, a process can flip a coin and then perform a step (based on the coin-flip outcome) in a single atomic operation.

The rest of this paper is organized as follows. We provide a high-level overview of our CC algorithm in Section 2. We then describe the algorithm in detail in Section 3, proving some complexity claims for the renaming part of the algorithm (see Section 3). For lack of space, we provide only a high-level overview of our algorithm's correctness and analysis proofs in Section 4. Finally, in Section 5 we describe how the CC algorithm is modified to work in the DSM model.

2. THE CACHE-COHERENT RANDOMIZED ALGORITHM: AN OVERVIEW

In this section, we provide a high-level overview of our adaptive randomized mutual exclusion algorithm. We then provide a detailed description of the algorithm in the next section. For presentation simplicity, and without loss of generality, we henceforth assume that $N = (\Delta + 1)!$ holds for some integer $\Delta > 0$.

The key data-structure underlying our algorithm is what we call a factorial tree. A *factorial tree* of height $\alpha \in \mathbf{N}^+$ is a tree of height α, in which every internal node in level i (for $i \in \{0, \ldots, \alpha - 1\}$, where the root's level is 0) has $i + 2$ children. Thus, each level $i \in \{0, \ldots, \alpha\}$ of a factorial tree consists of exactly $(i + 1)!$ nodes. As the following claim establishes, the height of a factorial tree with N leaves is sub-logarithmic in N.

CLAIM 1. *Let $\mathcal{F}$ be a factorial tree of height α with N leaves. Then $\alpha = \Theta(\log N / \log\log N)$.*

PROOF. By Stirling's formula we have

$$N = (\alpha + 1)! = 2^{\Theta(\alpha \cdot \log \alpha)},$$

and so $\alpha = \Theta(\log N / \log\log N)$. □

Our algorithm makes use of two trees with similar structure. We use a factorial tree $\mathcal{R}$ of height Δ and N leaves, called the *renaming tree*, and an *extended factorial tree* $\mathcal{T}$ of the same height, called the *tournament tree*. The sole difference in topology between a factorial tree and an extended factorial tree is that each internal node of the latter has a special child, that is a leaf, called *renaming leaf*. The following claim follows immediately from Claim 1.

CLAIM 2. *Let $\mathcal{T}$ be an extended factorial tree of height at least α with N level-α leaves. Then $\alpha = \Theta(\log N / \log\log N)$.*

The nodes in $\mathcal{R}$ and $\mathcal{T}$ are pairs $v = (\ell, i)$, $i \in \{0, \ldots, (\ell + 1)! - 1\}$, where $\ell \in \{0, \ldots, \Delta\}$ is the *level* of v and i is the *ordinal number* of v. The renaming leaves in $\mathcal{T}$ are also pairs (ℓ, i), where ℓ again indicates the level of that leaf, but the ordinal number i is larger than the ordinal number of all inner nodes on level ℓ.

Processes use the tournament tree $\mathcal{T}$ to synchronize their accesses of the critical section. In order to enter the critical section, a process tries to capture the locks of all the nodes along the path from a leaf to the root. However, if processes were to start their algorithm from a level-Δ leaf of $\mathcal{T}$, the algorithm's RMR complexity would be a function of N rather than a function of the point contention k. In order for the algorithm to be adaptive to point contention, a process p first *captures a node* of the renaming tree in a randomized manner by calling the `getName` procedure described shortly. After capturing a node (ℓ, i) of $\mathcal{R}$, p starts its algorithm in $\mathcal{T}$ from the child that is the renaming leaf of node (ℓ, i) in $\mathcal{T}$. If point contention is linear in N, then p may receive a name in level Δ. In this case, p starts its algorithm in $\mathcal{T}$ from a leaf of $\mathcal{T}$ that is statically assigned to it.

As we prove later, the essential property of the `getName` procedure is that it is guaranteed to return a node (ℓ, i) such that the expected value of ℓ is $O(\log k / \log\log k)$, where k is the point contention during the execution of `getName`. Moreover, we show that ℓ is (deterministically) at most $\min(k, \Delta)$.

After being assigned a starting leaf of $\mathcal{T}$, process p has to compete with other processes trying to enter the critical section by synchronizing with other processes accessing $\mathcal{T}$. Similarly to [16], p may enter its critical section either by capturing all nodes along the path from its starting leaf to $\mathcal{T}$'s root, or, otherwise, it can be randomly *promoted* by some other process q, when q exits its critical section. If p is promoted, then it no longer needs to climb up $\mathcal{T}$. In this case, p busy-waits until it is *signalled* to enter the critical section.

As we prove, the expected number of RMRs incurred by a process at level ℓ of $\mathcal{T}$ is $O(\log \ell)$. In order to bound the *worst-case* RMR complexity of our algorithm, we introduce a mechanism that allows a process to count the number of RMRs it incurs at each level ℓ. If p incurs more than $\log \ell$ RMRs at some node n in level ℓ of $\mathcal{T}$ without making progress, then p starts to compete on a deterministic starvation-free $(\ell + 3)$-process mutex object associated with n. (Observe that each level-ℓ node of $\mathcal{T}$ has $\ell + 3$ children.) As we prove, this mechanism allows us to bound the *worst-case* RMR complexity of our algorithm by $O(\min(k \log k, \log N)$, which is slightly worse than the $O(\min(k, \log N))$ complexity of the deterministic algorithm of [24].

3. A DETAILED DESCRIPTION OF THE ALGORITHM

We start by describing our randomized renaming algorithm and proving a few claims about its properties. We then describe the main mutual exclusion algorithm.

3.1 Randomized Renaming

Our renaming algorithm employs a factorial tree $\mathcal{R}$ of height Δ and N nodes, called the renaming tree. From Claim 1, $\Delta = \Theta(\log N / \log\log N)$. Each node of $\mathcal{R}$ stores a value in $\{\perp\} \cup \{0, \ldots, N - 1\}$ and supports the `CAS` operation. We say that a node of $\mathcal{R}$ is *captured by process p*, if its value equals p. We say that the node is *free* if its value is $\perp$. We assume that each leaf of $\mathcal{R}$ is permanently captured by the process to which it is statically assigned.

The pseudo-code of the renaming procedures is shown in Algorithm 1. The `getName` procedure does not receive any input and returns a name $n = (\ell, i)$; when it returns n, process p has captured that node n in $\mathcal{R}$. The `getName`

Algorithm 1: Renaming procedures for process $p \in \{0, \ldots, N-1\}$

```
1  shared R: Factorial tree with leaves L_0, ..., L_{N-1}
2  local i, level, n: int
3  Procedure getName {
     Output: (ℓ, i), for ℓ ∈ {0, ..., Δ} and
             i ∈ {0, ..., (ℓ+1)! − 1}
4  for ℓ = 0, ..., Δ − 1 do
5  |  i ← random value in {0, ..., (ℓ+1)! − 1}
6  |  if n[ℓ][i].CAS(⊥, p) then
7  |  |  return (ℓ, i)
8  |  end
9  end
10 return (Δ, p)
11 }
12 Procedure releaseName {
     Input: (ℓ, i), for ℓ ∈ {0, ..., Δ} and
            i ∈ {0, ..., (ℓ+1)! − 1}
13 if ℓ < Δ then
14 |  n[l][i].CAS(p, ⊥)
15 end
16 }
```

procedure operates as follows. Starting at level 0 (which consists only of the root node) and descending down $\mathcal{R}$ until an internal node is captured, (lines **4**–**9**), p randomly and uniformly picks a node at level ℓ (line **5**) and tries to capture it by atomically swapping its value from $\perp$ to p (line **6**). If an internal node of $\mathcal{R}$ was captured, then `getName` returns its name in line **7**. Otherwise, a leaf of $\mathcal{R}$ that is statically assigned to p is returned in line **10**.

Our implementation of `getName` uses the `CAS` operation. In what follows, we assume that `CAS` operations are atomic. However, the worst-case and amortized bounds against the strong adversary remain (asymptotically) the same if we replace atomic `CAS` operations by linarizable $O(1)$ RMRs CAS-implementations, such as the ones presented in [15].

The `releaseName` procedure receives as its input the name of the node currently captured by p and releases it by atomically swapping its value from p to $\perp$ (line **14**). (This is not required if the corresponding `getName` returned p's statically allocated leaf.)

We next establish a few complexity bounds on our randomized renaming algorithm.

LEMMA 3. *Assume that point contention during an execution of* `getName` *is at most k, then the expected level number returned by any* `getName` *call under a weak adversary model is $O(\log k / \log\log k)$.*

PROOF. From Claim 1 and the assumptions, there exists some $\beta = \beta(k) = O(\log k / \log\log k)$ such that $(\beta+1)! \geq 2k$. Hence, at least half of the nodes of any level $\ell \geq \beta$ of $\mathcal{R}$ are free at any point in time during the execution of the `getName` procedure. Since a weak adversary cannot foresee a coin toss outcome before scheduling the next step of a process, the probability that a process caputres a node when it executes line **6** for $\ell \geq \beta$ is at least $1/2$. □

Lemma 3 does not hold for a strong adversary model: it can be shown that a strong adversary can always cause a *specific* `getName` operation to return level number k in an execution with point contention k, if it is allowed to extend the execution long enough. This is because a strong adversary can halt a process after it randomly selects a node in line **5** and allow it to perform line **6** only after this node is captured by some other process. In fact, a similar argument works against the randomized algorithm of [21].

However, as we now show, the expected *amortized* level numbers returned by `getName` operations in any execution is $O(\log k / \log\log k)$ even under the strong adversary model.

LEMMA 4. *Let E be an execution in which processes repeatedly (and alternately) call the* `getName` *and* `releaseName` *procedures. Assume that point contention during E is at most k. Then the expected amortized level numbers returned by calls of* `getName` *during E under a strong adversary model is $O(\log k / \log\log k)$.*

PROOF. We assume w.l.o.g. that all invocations of `getName` during E respond during E (letting such calls terminate cannot decrease the average level number).

Consider the m-th call of `getName` by process p. We say process p *becomes interested* in node $n = (g, j)$ at time t, if at that point in time its local variable ℓ has value g, it executes line **5**, and the value chosen at random is j. The process *ceases* to be interested in node n, when it either executes an unsuccessful `CAS` operation in line **6** (thus failing to capture node n), or, in case that the `CAS` operation is successful, when it releases the name associated with n by executing the `CAS` operation in line **14** for that node $n = (g, j)$.

Let $Y_{p,m,g} \in \{0, 1\}$ be an indicator random variable, where $Y_{p,m,g} = 1$ iff

1. during its m-th `getName` call process p becomes interested in some node $n = (g, j)$, where $0 \leq j < (g+1)!$, and
2. at that point in time some process $q \neq p$ is already interested in the same node n.

Further, for fixed m and p, let

$$Y_{p,m} = \sum_{0 \leq g < \Delta} Y_{p,m,g}.$$

Define

$$Y = \sum_{p,m} Y_{p,m},$$

where the sum is taken over all processes p and m from one to the number of `getName` calls invoked by p. Now, let $X_{p,m}$ be the total number of times process p unsuccessfully executes the `CAS` operation in line **6** during its m-th `getName` call in E. Moreover, let

$$X = \sum_{p,m} X_{p,m}.$$

Then it suffices to prove that $E[X] = O(M \cdot \log k / \log\log k)$, where M is the total number of `getName` calls invoked during E. This is immediate from the following two statements:

(a) $X \leq Y$, and

(b) for each process p and each m, $E[Y_{p,m}] = O(\log k / \log\log k)$.

Proof of (a): Whenever the `CAS` operation of some process p on some node $n = (g, j)$ in line **6** fails, this is due

to some other process q occupying node n at that point in time. We say that q *fails* process p at level g, when this happens. Suppose that during its m_0-th `getName` call, some process p_0 fails b other processes $p_1, \ldots, p_b$ at level g, while p_z, $1 \le z \le b$, is performing its m_z-th `getName` call. Then there is one node $n = (g, j)$, and for each $0 \le z \le b$ a point t_z during p_z's m_z-th `getName` call at which p_z becomes interested in n. Let t'_z be the point in time when p_z ceases to be interested in that node. I.e., $[t_z, t'_z]$ is the time interval during which p_z is interested in node n. Let π be a permutation on $\{0, \ldots, b\}$ such that $t_{\pi(0)} \le \cdots \le t_{\pi(b)}$.

We claim that $Y_{p_{\pi(z)}, m_{\pi(z)}, g} = 1$ for $1 \le z \le b$. Suppose there is $1 \le z \le b$ such that $Y_{p_{\pi(z)}, m_{\pi(z)}, g} = 0$, i.e., when $p_{\pi(z)}$ becomes interested in n, no other process is interested in that node. Note that p_0 releases node n only after $t_{\pi(z)}$: If $\pi(z) = 0$ this is obvious, and otherwise, if p_0 released n before t_π, then it would not fail $p_{\pi(z)}$. Hence, $t_{\pi(z)} < t'_0$. But then $t_{\pi(z)} < t_0$, by the assumption that at time $t_{\pi(z)}$ no process (and in particular not p_0) is interested in node n. Now consider $p_{\pi(z-1)}$. By the choice of π, $t_{\pi(z-1)} < t_{\pi(z)}$ and by the assumption that at time $t_{\pi(z)}$ no other process is interested in n, $t'_{\pi(z-1)} < t_{\pi(z)}$. But then we can conclude that $t'_{\pi(z-1)} < t_0$. Hence, $p_{\pi(z-1)}$ stops being interested in n before p_0 starts being interested, which means that p_0 cannot fail $p_{\pi(z-1)}$—a contradiction.

We have shown $Y_{p_{\pi(z)}, m_{\pi(z)}, g} = 1$ for $1 \le z \le b$, and so

$$\sum_{0 \le z \le b} Y_{p_z, m_z, g} \ge b.$$

Now recall that if a node q gets failed on level g during its m-th `getName`, then there is only one process failing q at that point. Thus, for any level g,

$$\sum_{p,m} Y_{p,m,n}$$

is an upper bound on the number of times a process gets failed on level g. Clearly then Y is an upper bound on the total number of times a process gets failed, which is X. Claim (a) follows immediately.

Proof of (b): By Stirling's formula, there is a constant a and a positive integer $\beta = \beta(k) \le a \cdot \log k / \log\log k$, such that $\beta! \ge k$. Consider some level g. At any point in time, there can be at most k processes that are interested in a node on level g. Hence, the probability that a process p becomes interested in a node on level g, in which some other process is already interested, is at most $(k-1)/((g+1)!)$. Therefore, for any process p and its m-th `getName`-method call during E,

$$\text{Prob}(Y_{p,m,g} = 1) \le \frac{k}{(g+1)!}.$$

Obviously, $(\beta + c)! \ge k \cdot 2^c$, and so

$$\begin{aligned} E[Y_{p,m}] &= \sum_{0 \le g \le \Delta} \text{Prob}(Y_{p,m,g} = 1) \\ &= \sum_{0 \le g < \beta} \text{Prob}(Y_{p,m,g} = 1) + \sum_{\beta \le g \le \Delta} \text{Prob}(Y_{p,m,g} = 1) \\ &\le \beta + \sum_{c \ge 0} \text{Prob}(Y_{p,m,\beta+c} = 1) \\ &\le \beta + \sum_{c \ge 0} \frac{k}{(\beta + c + 1)!} \;\le\; \beta + \sum_{c \ge 0} \frac{k}{k \cdot 2^{c+1}} \\ &\le \beta + 1 = O(\log k / \log\log k). \end{aligned}$$

□

We next provide a deterministic bound on the level numbers returned by `getName` and on the number of steps it performs as a function of point contention.

LEMMA 5. *If the point-contention during an execution of* `getName` *is at most k, then within $O(\ell)$ steps that call returns a node (ℓ, i) (for some i), where $\ell < min(k, \Delta)$, and a subsequent* `releaseName` *call releases that name in $O(1)$ steps.*

PROOF. Clearly from the algorithm, the highest level number returned by `getName` is Δ and `releaseName` performs a constant number of steps. It remains to show that $\ell < k$ holds.

Consider an execution of the `getName` procedure by some process p that returns a name in level ℓ. We let $N_{p,j}$, $j \in \{0, \ldots, \ell\}$, denote the j-th node of $\mathcal{R}$ on which p applies the `CAS` operation of line **6**. Clearly from the algorithm and assumptions, the `CAS` operation applied by p on node $N_{p,\ell}$ succeeds, while the `CAS` operations it applies to $N_{p,0}, \ldots, N_{p,\ell-1}$ (assuming $\ell > 0$) fail. Let s_p be the time when p starts executing `getName`. Also, let $t_{p,j}$ denote the time when p applies its `CAS` operation to $N_{p,j}$, for $0 \le j < \ell$. We prove Claim 6 below by induction on j, and the lemma then follows immediately from parts (I3) and (I5) of that claim for $j = \ell - 1$. □

CLAIM 6. *Suppose p's* `CAS` *operation at time $t = t_{p,j}$ fails. Then there is a point in time t', and a set Q of processes, such that the following hold.*

(I1) $s_p \le t' \le t$,

(I2) $p \notin Q$,

(I3) all processes in Q are active at time t',

(I4) each process $q \in Q$ owns a node N_{p,ℓ_q}, for $\ell_q \le j$, at some point in time t_q, where $t_q \in [s_p, t]$, and

(I5) $|Q| = j + 1$.

PROOF. Consider the base case, $j = 0$. Since p's `CAS` operation on $N_{p,0}$ (which is $\mathcal{R}$'s root node) fails at time t, it must be that some process $q \ne p$ owns $N_{p,0}$ at time t. Set $t' = t$ and $Q = \{q\}$. Clearly, (I1)-(I5) are satisfied.

Now let $j > 0$. Suppose p's `CAS` operation fails at time $t = t_{p,j}$ on node $N_{p,j}$, for some $j > 0$. For $0 \le m \le j$, let q_m be the process that owns $N_{p,m}$ at time $t_{p,m}$, that is, at the point in time when p's `CAS` operation on $N_{p,m}$ fails. Also, let s_{q_m} be the time at which q_m started the execution of the

getName instance during which it owned $N_{p,m}$ at time $t_{p,m}$. The following two possibilities exist.

Case 1: $s_{q_j} \leq s_p$. From the induction hypothesis applied to p, there exists a process set Q' of size j' and a point in time t' such that (I1)-(I5) hold for $j' = j - 1$. Specifically, (I4) is true for nodes $N_{p,0}, \ldots, N_{p,j-1}$. We claim that $Q = Q' \cup \{q_j\}$ and t' satisfy (I1)-(I5). (I1), (I2), and (I5) follow from the induction hypothesis and because a process owns at most a single node at any point of time. (I4) holds since q_j owns $N_{p,j}$ at time $t_{p,j} \in [s_p, t]$. Finally, (I3) holds because $s_{q_j} \leq s_p \leq t' \leq t_{p,j}$ and since q_j is active all throughout $[s_{q_j}, t_{p,j}]$.

Case 2: $s_{q_j} > s_p$. Since the CAS operations of q_j failed on nodes $N_{p,0}, \ldots, N_{p,j-1}$, we can apply the induction hypothesis to q_j and these nodes. Therefore there exists a processes group Q' of size $j-1$ and a point in time t' such that (I1)-(I5) hold for $j' = j-1$ w.r.t. q_j and nodes $N_{p,0}, \ldots, N_{p,j-1}$. We claim that $Q' \cup \{q_j\}$ and t' now satisfy (I1)-(I5) w.r.t. p. (I5) follows immediately from the induction hypothesis. (I1) holds because $s_p < s_{q_j} \leq t' < t$. (I2) holds since p does not own any node during interval $[s_p, t']$. (I3) follows from the induction hypothesis and since q_j is active at time t'. Finally, (I4) holds by the induction hypothesis and since q_j owns $N_{p,j}$ at time t. □

3.2 The Main Algorithm

Similarly to many mutual exclusion algorithms, our main algorithm uses an arbitration tree. Each process starts at a leaf of the arbitration tree and moves up the tree to the root, locking nodes on its path. Once a process locks the root, it can enter the critical section. In [17], a tree $\mathcal{T}_\Delta$ with branching factor $\Delta = \log N / \log\log N$ is used, and all the leaves are on level $\Theta(\log N / \log\log N)$. The key difference between the main part of our algorithm and that of [17] is in the topology of the arbitration tree we use. Our algorithm uses an *extended factorial tree* $\mathcal{T}$. We remind the reader that an extended factorial tree has the same structure as a (regular) factorial tree, except that every internal node $n = (\ell, i)$ has an additional leaf child, called a *renaming leaf*. Thus, every internal node in level ℓ of an extended factorial tree has $\ell + 3$ children (one more than a regular factorial tree).

The reasons our algorithm uses an extended factorial tree rather than the tree $\mathcal{T}_\Delta$ used in [17] are twofold. First, since the nodes on each level of $\mathcal{T}_\Delta$ have branching factor $\log N / \log\log N$, the expected RMR complexity of a process trying to ascend from a node of $\mathcal{T}_\Delta$ is a function of N regardless of the node's level; in contrast, the branching factor of a factorial tree node is proportional to its level. Second, since each internal node of an extended factorial tree has a renaming leaf, a process may start ascending the arbitration tree from a level proportional to the point contention it incurred during the renaming algorithm rather than from the bottommost level.

The structure of $\mathcal{T}$'s internal nodes is shown in Algorithm 2, lines **17**–**21**. Each level-ℓ node contains a lock variable *lock* that can be manipulated by CAS operations, a deterministic starvation-free $(\ell+3)$-process mutual exclusion object M, an integer *M_winner* storing the ID of the last process that won M on this node (initialized to 0), an array *apply*, and an index *token* into that array. The mutex-object M supports the method calls $\texttt{Entry}_i()$ and $\texttt{Exit}_i()$. As long as a process has called $\texttt{Entry}_i()$ but not yet finished $\texttt{Exit}_i()$, no other process is allowed to call $\texttt{Entry}_j()$ or $\texttt{Exit}_j()$ for $j = i$.

Algorithm 2: AdaptiveRandomMutEx$_p$

```
17 define Node⟨ℓ⟩: struct {
18       lock: int init ⊥,
19       M: Mutex object for processes with names in
         {0, ..., ℓ + 3}
20       apply: array [0 ... ℓ + 2] of int init ⊥
21       token, M_winner: int init 0 }
22 shared
23       promQ: Sequential queue init ∅
24       promoted: array [0 ... N − 1] of boolean
25                                     init false
26 local
27       v, startName: Node;   (root is the root-node)
28       i, j, j', tok, ℓ: int
29 Entry_p()
30 // Critical Section
31 Exit_p()
```

Processes share a "promotion" queue *promQ*, which can only be accessed sequentially, and a "promotion array" *promoted*. We explain the purpose of these data structures later. We assume that the following helper functions are available (but we do not provide their pseudo-code, since their implementation is obvious):

1. `Node getLeaf(Node` (ℓ, i)`)`: returns the leaf in $\mathcal{T}$ associated with name (ℓ, i). If (ℓ, i) is an internal node then its renaming leaf is returned. Otherwise, (ℓ, i) is a leaf itself, and is returned by `getLeaf`.
2. **int** `getLevel(Node` v`)`: returns the level of node v.
3. `Node getParent(Node` v`)`: returns the parent of node v.

3.2.1 The Entry Section

The entry section of our algorithm is implemented by the Entry function. A process p first calls getName and then uses the method getLeaf to determine the leaf in $\mathcal{T}$ that corresponds to the name it received. Process p then iterates in the *outer entry loop* of lines **34**–**58** until it either succeeds in capturing all the locks on the path from its starting leaf to $\mathcal{T}$'s root, or it is promoted. We now describe this loop in more detail.

At the beginning of each outer entry loop iteration, process p first climbs to the next internal node v on the path from its starting leaf to the root (line **36**). The local variable i is set to the ordinal number of the child from which p ascends to v (in line **35**). Next, process p *applies for promotion* by changing the value of $v.apply[i]$ from $\perp$ to its ID. As we soon describe, once p has applied for promotion, it may be *promoted* by processes performing their exit section. If p gets promoted, then it need not ascend further and will enter the critical section within a constant number of RMRs. Once process p has applied for promotion, it initializes its local variable *ctr*, which is used to count the number of iterations it makes in the following repeat-until loop. In this loop (lines **39**–**54**), called the *inner entry loop*, p iterates until it either captures v's lock or is promoted. We proceed to describe the inner entry loop in more detail.

```
Function Entry_p
32 startName := getName()
33 v := startLeaf :=getLeaf(startName)
34 repeat
35   Let i be the integer such that v is the (i + 1)-th
     child of its parent
36   v :=getParent(v); ℓ :=getLevel(v)
37   v.apply[i].CAS(⊥, p)
38   ctr := 0
39   repeat
40     ctr := ctr + 1
41     if ctr > ⌈log ℓ⌉ then
42       if v.apply[i].CAS(p, ⊥) then
43         v.M.Entry_i()
44         v.apply[i].CAS(⊥, p)
45         v.M_winner := i
46         await v.lock = ⊥ ∨ v.apply[i] ≠ p
47       end
48     end
49     if ¬ v.lock.CAS(⊥, p) then
50       tok := v.token
51       await
         v.token ≠ tok ∨ v.apply[i] ≠ p ∨ v.lock = ⊥
52     end
53     if v.M_winner = i then v.M_winner := ⊥;
       v.M.Exit_i()
54   until v.apply[i] ≠ p ∨ v.lock = p
55   if ¬ v.apply[i].CAS(p, ⊥) then
56     await promoted[p] = true
57   end
58 until promoted[p] ∨ v = root
```

We use the following mechanism to bound the worst-case RMR complexity of our algorithm: If a process p performs too many iterations of the inner entry loop, and thus incurs too many RMRs, it becomes *desperate*. When this happens, p reverts to performing the deterministic mutex algorithm of object $v.M$. Object $v.M$ represents an optimal (non-adaptive) deterministic local-spin mutual exclusion algorithm for both the CC and DSM models, such as the Yang and Anderson algorithm [24], shared by $(\ell + 3)$-processes.

The key idea is that once p is in the critical section of $v.M$, it has exclusive access to register $v.MWinner$, and thus can set it to the value of i. This will guarantee that p gets promoted quickly. We first describe p's behavior in each iteration of the internal loop before it exceeds this number of RMRs.

In line **49**, p tries to capture v's lock. If that fails, p reads the value of the token register $v.token$ in line **50**. Next, p proceeds to busy-wait in line **51** until it identifies that one of the following 3 conditions is satisfied: either v's lock is free again, or p was promoted, or v's lock was released and re-captured at least once since p last read $v.token$. If at least one of these conditions is satisfied, then p terminates the busy-waiting of line **51** and proceeds to evaluate the condition of line **54**. Since p's `CAS` operation in line **49** failed, the first condition of line **53** will be false; however, it might be that p has been promoted, in which case p exits the internal loop, fails the `CAS` of line **55** (hence "realizing" it was promoted) and awaits in line **56** to be *signalled* when it may enter the critical section (we explain how a process is signalled when we describe the exit code). Once signalled, p may proceed to enter the critical section, since the first condition of line **58** is satisfied.

If p's CAS at line **49** succeeds, then p proceeds to line **54**, where the first condition is satisfied. It then exits the inner entry loop, succeeds in *withdrawing its application for promotion* at line **55**, and then either proceeds to enter the critical section (if v is the root) or proceeds to the next external loop iteration.

We conclude the description of the entry section by explaining what p does when it becomes desperate, i.e., when the value of ctr exceeds $\log \ell$ (that is, the condition of line **41** is satisfied). In this case, p first tries to withdraw its application for promotion at node v (line **42**). If that fails, process p has already been promoted and will proceed as described above. If it succeeds to withdraw its application, it proceeds to perform the entry code of $v.M$ (recall that $v.M$ is a deterministic $(\ell + 3)$-process mutex object), playing in this algorithm the role of the process with ID i. Once it enters $v.M$'s critical section (thus incurring additional $\Theta(\log \ell)$ RMRs), p writes the ordinal number i to $v.M\_winner$ and then applies for promotion again. As we will describe later, when a process releases a lock (in the course of its exit section), it promotes the winner of $v.M$, if there is one. Process p then busy waits in line **46** until it is either promoted or until v's lock is free again. In either case, p proceeds to perform line **49**. As we prove later, p will either succeed in capturing the lock or, otherwise, will discover in line **51** that it was promoted. In either case, p proceeds to perform the exit section of $v.M$ in line **53**. It will next either climb to the next level of $\mathcal{T}$ or wait to be signalled at line **56**.

3.2.2 The Exit Section

The exit section of our algorithm is implemented by the `Exit` function. In the foreach-loop of lines **59**–**72**, process p ascends up the path from its starting leaf to the root and considers every node v on this path whose lock it captured. For each such node v, p randomly and uniformly selects an index j' into v's *apply* array in line **63**. Process p then tries to promote up to 3 processes in lines **64**–**69**.

Processes whose application for promotion is pending can get promoted. In particular, if $v.apply[z] = q$ throughout this promotion procedure for some process q, then q will be promoted if 1) $z = j'$ (we call this *randomized promotion*); 2) $z = tok$ (we call this *deterministic promotion*); and 3) $z = v.M\_winner$ (this is the latest process to have won the deterministic mutex algorithm of $v.M$). For each of these entries, p checks whether there is a process q that has applied for promotion at that entry and, if so, tries to promote it by atomically swapping the corresponding entry in the *apply* array to $\perp$ (lines **65**–**66**). If p succeeds in swapping q's value, then it enqueues q to the promotion queue. As we prove, this guarantees that q will eventually enter the critical section after performing an additional constant number of RMRs. In line **70**, p increments v's *token* (modulo v's branching factor), thus guaranteeing that every process registered in v will either climb up or be promoted after v's lock is released at most $\ell + 3$ times.

Next, process p performs lines **73**–**79** and either releases the root-lock or hands it over to some other process. If the promotion queue *promQ* is empty, then p must release

Function Exit$_p$

```
59 foreach node v on the path from getLeaf(startLeaf)
   to the root, where v.lock = p do
60 |  ℓ :=getLevel(v)
61 |  tok := v.token
62 |  i := v.M_winner
63 |  Pick j' uniformly at random from {0, ..., ℓ}
64 |  for j ∈ {j', tok, i} do
65 |  |  q := v.apply[j]
66 |  |  if q ≠ ⊥ ∧ v.apply[j].CAS(q, ⊥) then
67 |  |  |  promQ.Enq(q)
68 |  |  end
69 |  end
70 |  v.token := (tok + 1) mod (ℓ + 3)
71 |  if v ≠ root then v.lock.CAS(p, ⊥)
72 end
73 if promQ = ∅ then
74 |  root.lock.CAS(p, ⊥)
75 else
76 |  q := promQ.Deq()
77 |  root.lock.CAS(p, q)
78 |  promoted[q] := true
79 end
80 releaseName(startName)
```

the lock for guaranteeing deadlock-freedom; it does that at line **74**. If the promotion queue is non-empty, however, p does not release the root's lock. Instead, p dequeues the first process q in the promotion queue (line **76**), hands the root-lock over to q (line **77**) and *signals* q that it may now proceed to enter the critical section (line **78**). Finally, p releases the name it captured in its entry section.

4. CORRECTNESS AND ANALYSIS

The proofs of correctness and liveness of our algorithm, i.e., mutual exclusion and starvation-freedom, are similar to the ones in the journal version [17] of [10], currently in preparation. The key difference between the proofs for our algorithm and those provided in [17] stems from the different structures of the arbitration trees underlying the two algorithms. Here we present a high-level overview of the proof arguments.

LEMMA 7. *The algorithm* `AdaptiveRandomMutEx` *satisfies mutual exclusion and starvation freedom.*

PROOF SKETCH. A process p can only leave the inner entry loop (line **54**) if it gets promoted (and thus $v.apply[i] \neq p$) or has captured the lock of the node it is currently registered in (and thus $v.lock = p$). Therefore, a process can only leave the outer entry loop (line **58**), once it has been promoted or captured the root-lock. We argue that even if it has been promoted it still has to capture the root-lock in order to leave the outer entry loop and enter the critical section. Since at any point in time only one process can own the root-lock, this guarantees mutual exclusion.

In order to be promoted, a process p has to first apply for promotion, by writing its ID in $u.apply[j]$ for some node u and some index j. Process p gets promoted, if some other process, p', changes that value back to $\perp$. If that happens, p will detect it when executing line **55**, and will then wait to be signalled in line **56**. However, another process can only signal p (in line **78**), after handing over the root-lock to p (line **77**).

As for starvation freedom, we argue that processes cannot busy-wait indefinitely in one of the **await**-operations of `Entry`$_p$`()` (line **46**, line **51**, or line **56**). Process r starts busy-waiting in line **56** only after it was promoted by some other process r' that executed line **66**. But then r' will add r to the promotion queue, and will later remove a process from the promotion queue and will signal it (line **78**). This guarantees that the first process in the promotion queue, say r'', will be removed from $promQ$ and be signalled. Thus r'' will stop waiting in line **56** and will enter the critical section. (It is easily seen that r'' does not busy-wait in any of the other **await**-operations.) On its way through the exit-section, r'' will then signal the next process in the promotion queue, and will also remove it from the queue. This continues, until all processes (including r) have been removed from the queue and were signalled. □

LEMMA 8. *During an execution of* `AdaptiveRandomMutEx`*, process p incurs at most $O(L \cdot \log L)$ RMRs, where L is the level of the node returned by p's* `getName` *operation.*

PROOF SKETCH. During its `getName` and corresponding `releaseName` calls, p incurs at most $O(L)$ RMRs (Lemma 5). During one iteration of the the inner entry loop, a process incurs only a constant number of RMRs, unless p becomes desperate (that happens when $ctr > \lceil \log \ell \rceil$). On the other hand, when p becomes desperate, it is guaranteed that it will either win the lock of its current node v, or it will be promoted. Moreover, in an iteration when p is desperate, p incurs $O(\log \ell)$ RMRs due to the operations on the $(\ell + 3)$-process mutual exclusion object M. But this happens only when p has already incurred $O(\log \ell)$ RMRs in the previous $\lceil \log \ell \rceil$ iterations. We conclude that, for each node on level ℓ visited by p, p incurs $O(\log \ell)$ RMRs. Since $\ell \leq L$, and since p visits at most $L + 1$ nodes, the worst-case RMR-bound follows. □

LEMMA 9. *During an execution of* `AdaptiveRandomMutEx`*, process p incurs an expected number of $O(L)$ RMRs, where L is the level of the node returned by p's* `getName` *operation.*

PROOF SKETCH. The main idea is that if p performs a constant number of iterations of the inner entry loop for some node v, then it has a probability of $\Omega(1/\ell)$ of being promoted (where ℓ is the level of v). Since the level number of each node that the process visits is bounded by L, the expected total number of iterations of the inner entry loop, until the process gets promoted is $O(L)$. The statement then follows from the fact that in each iteration p incurs only $O(1)$ RMRs, unless p is desperate, and that the total number of RMRs p incurs in an iteration while it is desperate is not larger than the total number of RMRs it incurs in preceding iterations. □

In order to analyze the expected RMR complexity, we use the following obvious statement.

CLAIM 10. *Let X and Y be two discrete random variables, such that $E[X]$ exists and $E[X|Y = y] \leq f(y)$ for some function $f : \mathbb{R} \to \mathbb{R}$. Then*

$$E[X] \leq E[f(Y)].$$

PROOF. Since $E[X]$ exists, the following sums are uniquely determined:

$$\begin{aligned}
E[X] &= \sum_x x \cdot \mathrm{Prob}(X = x) \\
&= \sum_x x \cdot \left(\sum_y \mathrm{Prob}(X = x | Y = y) \cdot \mathrm{Prob}(Y = y) \right) \\
&= \sum_y \mathrm{Prob}(Y = y) \left(\sum_x x \cdot \mathrm{Prob}(X = x | Y = y) \right) \\
&= \sum_y \mathrm{Prob}(Y = y) E[X | Y = y] \\
&\leq \sum_y \mathrm{Prob}(Y = y) f(y) \\
&= E[f(Y)].
\end{aligned}$$

□

THEOREM 11. *Algorithm* `AdaptiveRandomMutEx` *is a starvation-free adaptive randomized mutual exclusion algorithm that has* $O(\log k / \log\log k)$ *expected RMR complexity against the weak adversary,* $O(\log k / \log\log k)$ *expected amortized RMR complexity against the strong adversary, and* $O\big(\min(k \log k, \log N)\big)$ *worst-case RMR complexity, where* k *is the maximum point-contention.*

PROOF. Consider a call of `AdaptiveRandomMutEx` by process p and let X be the number of RMRs p incurs. Let L be the level of the leaf corresponding to the name returned by the `getName`-call in p's entry-section. By Lemma 5 and Lemma 9, $E[X|L = y] = O(y)$ for all y. By Lemma 3 for the weak adversary, $E[L] \leq \log k / \log\log k$. Hence, applying Claim 10, we obtain $E[X] = O(\log k / \log\log k)$.

The proof of the amortized RMR complexity bound for the strong adversary is analogous. The worst-case RMR bound follows immediately from Lemma 5 and Lemma 8. □

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. This is the same technique as the one used in the unpublished manuscript [17]—we present it here for completeness reasons.

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 using primitives called wait-signal objects (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 the appropriate methods at most once, and the methods take no arguments and return no values. The semantics, progress condition, and RMR complexity guaranteed by WS objects are as follows:

(S) If $r \neq p$ calls W_p, then that method call does not terminate before p calls W_p.`Signal()`.

(P) `Signal()` is wait-free. Moreover, in any execution where process p calls W_p.`Signal()`, all calls to W_p.`Wait()` terminate.

(R) Both `Wait()` an `Signal()` incur $O(1)$ RMRs.

For an implementation of WS objects we refer to [17]. We use these objects in order to modify the CC algorithm presented in Section 2 so that the desired expected and worst-case RMR complexity is achieved in the DSM model.

In the algorithm for the CC model, a process p can busy-wait only in lines 46, 51, and 56. In our analysis for the CC model, the worst-case and expected RMR complexity bounds are only based on the total number of iterations of the inner entry loop. Moreover, if we use a mutual exclusion algorithm such as that of [24] to implement the mutual exclusion object M, then the RMR complexity of the operations on that object are asymptotically the same in the CC and DSM models. It follows that the aforementioned **await** operations are the only operations that can cause the algorithm to incur higher asymptotic RMR complexity in the DSM model than in the CC model.

Dealing with line **56** is straightforward: for all p, we simply store entry *promoted*[p] in p's local memory segment. This way, line **56** incurs no RMRs in the DSM model.

We now explain how we modify lines 46 and 51 for the DSM model. 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 tries to swap a pointer *ptr* pointing to that object into $v.lock$. Suppose it succeeds and so now owns 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 $v.lock$ in order to notify all processes that the lock has been released.

Observe that, in order to hand the root lock over from process p to process q, process p has to create a new WS object W_q and swap a pointer to that object into $root.lock$.

Mutual exclusion for the DSM algorithm follows from the same arguments as for the CC model. The analysis of the RMR-complexity is also essentially the same. Starvation freedom follows immediately from guarantees (P) and (S) of WS objects and by using the same arguments provided for the CC model algorithm.

6. REFERENCES

[1] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive collect with applications. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 262–272, 1999.

[2] Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

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

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

[5] J. H. Anderson and Y.-J. Kim. Fast and scalable mutual exclusion. In *Distributed Computing, 13th International Symposium*, pages 180–194, 1999.

[6] J. H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2002.

[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*, 16:165–175, 2003.

[9] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. *Distributed Computing*, 15(3):177–189, 2002.

[10] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*, pages 217–226, 2008.

[11] R. Danek and W. M. Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. In *Distributed Computing, 22nd International Symposium (DISC)*, pages 93–108, 2008.

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

[13] R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 275–284, 2006.

[14] W. Golab. *Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations*. PhD thesis, University of Toronto, 2010.

[15] 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 *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 3–12, 2007.

[16] D. Hendler and P. Woelfel. Randomized mutual exclusion in $O(\log N / \log \log N)$ RMRs. In *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 26–35, 2009.

[17] D. Hendler and P. Woelfel. Randomized mutual exclusion in $O(\log N / \log \log N)$ RMRs. *Journal version of [16], in preparation*, 2010.

[18] 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.

[19] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing (PODC 2003)*, pages 295–304, 2003.

[20] P. Jayanti, S. Petrovic, and N. Narula. Read/write based fast-path transformation for FCFS mutual exclusion. In *31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, pages 209–218, 2005.

[21] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Distributed Computing, 15th International Conference (DISC)*, pages 1–15, 2001.

[22] Y.-J. Kim and J. H. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, 19(3):197–236, 2007.

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

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