Randomized Mutual Exclusion with Constant Amortized RMR Complexity on the DSM

George Giakkoupis
INRIA Rennes – Bretagne Atlantique
Rennes, France
george.giakkoupis@inria.fr

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

***Abstract*—In this paper we settle an open question by determining the remote memory reference (RMR) complexity of randomized mutual exclusion, on the distributed shared memory model (DSM) with atomic registers, in a weak but natural (and stronger than oblivious) adversary model. In particular, we present a mutual exclusion algorithm that has constant expected amortized RMR complexity and is deterministically deadlock free. Prior to this work, no randomized algorithm with $o(\log n/\log\log n)$ RMR complexity was known for the DSM model. Our algorithm is fairly simple, and compares favorably with one by Bender and Gilbert [11] for the CC model, which has expected amortized RMR complexity $O(\log^2 \log n)$ and provides only probabilistic deadlock freedom.**

***Keywords*-Mutual exclusion; RMR complexity; shared memory; oblivious adversary; DSM**

I. Introduction

Mutual exclusion, introduced by Dijkstra [16], is one of the best studied problems in concurrent computing. A *mutual exclusion object* (or *lock*) is a fundamental synchronization primitive that allows processes to coordinate their access to a shared resource, by serializing the execution of a piece of code, called *critical section*. At any point in time, at most one process must be in its critical section; we say that this process *owns the lock*. A process obtains a lock through an *entry section* (or *capture protocol*), and the owner of a lock frees up the lock by executing an *exit section* (or *release protocol*). A textbook by Raynal [33] is devoted to mutual exclusion research up to the mid 80s, and a survey by Anderson, Kim, and Herman [3] covers research between 1986 and 2003.

Early mutual exclusion algorithms did not take into account the gap between high processor speeds and the low speed/bandwidth of the processor–memory interconnect [12]. In *distributed shared memory (DSM)* systems, each shared variable is permanently locally accessible to a single processor and remote to all other processors. In *cache-coherent (CC)* systems, each processor keeps local copies of (remote) shared variables in its cache, and the consistency of copies in different caches is maintained by a *coherence protocol*. Memory accesses that cannot be resolved locally in DSM and CC systems are called *remote memory references (RMRs)*. RMRs are orders of magnitude slower than local memory accesses. Hence, the performance of many algorithms for shared memory multiprocessor systems depends critically on the number of RMRs they incur [7], [31].

The mutual exclusion problem inherently requires processes to *busy-wait* in their entry section, and thus the number of shared memory accesses cannot be bounded. Therefore, the traditional step complexity measure, which counts the number of shared memory accesses, is not useful to determine the performance of mutual exclusion algorithms. *Local-spin* algorithms, which perform busy-waiting by repeatedly reading locally accessible shared variables, can achieve bounded RMR complexity and have practical performance benefits [7]. Recent research has almost entirely used the RMR complexity as a metric for the performance of mutual exclusion algorithms (see, e.g., [5]–[7], [9]–[11], [14], [15], [19], [22], [23], [25]–[29], [32]).

Using strong primitives, such as fetch&increment objects, it is possible to implement mutual exclusion so that every process incurs only a constant number of RMRs per passage through the critical section. A prominent example is the MCS lock [31], which uses an object that allows both compare&swap (CAS) and swap operations. Other examples can be found in standard textbooks, such as [24].

A significant amount of research has focused on determining the RMR complexity of the mutual exclusion problem if only atomic registers are available. Some common synchronization primitives, and in particular CAS and load-linked/store-conditional objects, have linearizable implementations with constant RMR complexity from registers [20], and therefore they cannot help improving the asymptotic worst-case RMR complexity.

Unless mentioned otherwise, the results discussed below hold for the CC and the DSM model with atomic registers, and n is the number of processes.

The deterministic RMR complexity of mutual exclusion is $\Theta(\log n)$ RMRs per passage through the critical section. The upper bound was established by Yang and Anderson's algorithm with $O(\log n)$ worst-case RMR complexity [34]. Further, Anderson and Kim [4] conjectured that this bound is optimal. Following several lower bound proofs of increasing strength [13], [17], [27], Attiya, Hendler, and Woelfel [10]

proved this conjecture true.

More recently, *randomized* techniques have been employed to improve the efficiency of mutual exclusion algorithms. To capture how random decisions made by processes can influence the order in which processes take steps (e.g., because accesses of some shared registers may be slower than others), it is assumed that an *adversary* produces the schedule. Among the most common adversary models are the *strong adaptive adversary*, where scheduling decisions can depend on all past events, including local coin flips, and the *oblivious adversary*, where scheduling decisions are independent of processes' random decisions, i.e., the adversary fixes the schedule in advance. Unfortunately, little can be gained by using randomization in the strong adaptive adversary model: Giakkoupis and Woelfel [19] showed that any mutual exclusion algorithm in this model has expected RMR complexity $\Omega(\log n/\log\log n)$, matching an upper bound by Hendler and Woelfel [23].

Note that unlike in deterministic algorithms, in randomized ones linearizable implementations of objects can in general not replace atomic objects, without affecting the probability distribution over possible outcomes [21]. (Linearizability is defined in Section II; an object is atomic[1] if each operation takes effect instantaneously, once invoked. In particular, multiple operations on the same atomic object do not overlap in an execution.) The known constant-RMR CAS implementations [20], however, preserve those probability distributions against a strong adaptive adversary. Therefore, the tight $\Theta(\log n/\log\log n)$ expected RMR bound for mutual exclusion for the strong adaptive adversary holds even if CAS or load-linked/store-conditional objects are available, in addition to registers. Hence, it is not possible to achieve $o(\log n/\log\log n)$ RMR complexity without using stronger, less common synchronization primitives.

The strong adaptive adversary constitutes a very pessimistic system assumption, as it assumes that the system reacts in the most undesirable way to random decisions made by processes. Recently, researchers have increasingly focused on finding efficient randomized algorithms for the weaker, oblivious adversary model, e.g., for test-and-set [2], [18] or consensus [8].

Bender and Gilbert [11] have devised a randomized mutual exclusion algorithm (which will henceforth be called BG algorithm) that achieves $O(\log^2\log n)$ expected amortized RMR complexity against the oblivious adversary, in a CC model that provides atomic registers and CAS objects. However, unlike existing randomized algorithms for the strong adaptive adversary, the BG algorithm guarantees deadlock freedom only with high probability per passage through the critical section (rather than with certainty). The BG algorithm uses CAS objects as mentioned above, and it remains unknown whether a similar efficient implementation from registers exist.[2]

For the DSM model, no mutual exclusion algorithm with randomized $o(\log n/\log\log n)$ RMR complexity against an oblivious adversary was known, until now.

Our Contribution

We present a mutual exclusion algorithm for DSM systems that is optimal w.r.t. several parameters. In particular, it

- has constant expected amortized RMR complexity in the oblivious adversary model,
- is deterministically deadlock free (and can be transformed into starvation-free using standard techniques),
- can be implemented from atomic $O(\log n)$-bit registers only.

In fact, we use an adversary that is stronger than the oblivious one, and seems realistic for the DSM model. The adversary can make scheduling decisions based on limited information about the operations each process has incurred in the past and the operation it will incur in its next step. While the adversary cannot know the exact register location on which such an operation has or will be performed, it can take into account the type of this operation (read or write), and whether it is a local or remote reference.

In our presentation of the DSM algorithm, we use a single CAS object, whose only purpose is to allow processes to repeatedly elect a leader, i.e., solve name consensus. Our complexity analysis makes no assumption that the CAS object is atomic (it assumes linearizability, but even weaker consistency conditions suffice), and therefore known implementation of CAS objects from registers [20] can be used without sacrificing the asymptotic RMR complexity of our mutual exclusion algorithm.

Finally, our algorithm is fairly simple. This is in contrast to the BG algorithm, which relies on a stack of other implemented objects, such as max-registers and approximate counters with various properties.

II. Model

We consider the standard distributed shared memory (DSM) model, where a set $\{0,\ldots,n-1\}$ of n processes communicate by executing read and write operations on shared atomic registers. The set of registers is partitioned into n memory segments, one for each process. A read or write step on a register R by process p incurs a remote memory reference (RMR) if and only if R is not in p's memory segment. In this paper we assume that some registers are remote to all processes—it is not hard to see that

[1]Sometimes in the literature the term "atomic" is used to denote "linearizable" (see, e.g., Lynch's textbook [30]).

[2]A statement in [11], saying that it is safe to replace the CAS objects with the implementation provided in [20], for the reason that this implementation is strongly linearizable, is incorrect. Strong linearizability [21] only preserves the power of the strong adaptive adversary, but there are examples showing that it does not in general preserve the power of the oblivious adversary.

this assumption can be made w.l.o.g. in mutual exclusion algorithms.

A *schedule* is a sequence of process IDs, and yields an execution in which processes take shared memory steps in the order determined by the sequence. Processes can flip (private) coins to make random decisions. We consider an adversary that schedules processes in an adaptive way, but with limited information. When scheduling the next process to take a step, the adversary has available the following information about each past step of any process, and about the step each process is poised to execute: the type of that step, i.e., whether it is a read or write, and whether the step constitutes a remote or local reference, i.e., whether or not the affected register is in the executing process' memory segment. The exact location of the register to which a read or write operation is applied, or what value is being read or written, is not revealed to the adversary, even after the process has executed that operation. In addition, we assume that the adversary learns whenever a `lock()` or `release()` call responds. (We assume that each process calls `release()` immediately after termination of its `lock()` method call, so the adversary knows when a process is poised to call `release()`. Thus, it can delay those `release()` calls arbitrarily.)

A *compare&swap (CAS)* object C supports the operations C.`read()`, which returns the value of C, and C.`CAS(`old, new`)`, which writes new into C if $C = old$, and otherwise does not change C. In either case, it returns the value that C had at the point immediately before the operation was applied. It is known that CAS objects can be implemented from atomic registers such that each `CAS()` operation incurs $O(1)$ RMRs [20]. This implementation is *linearizable*: In any execution on an implemented CAS object, every `CAS()` operation can be associated with a *linearization* point between the invocation and response of that `CAS()`, such that ordering all `CAS()` operations by their linearization points yields a sequential execution that matches the specification of CAS.

The mutual exclusion problem can be specified in terms of a *lock* object, which supports operations `lock()` and `release()`. Each processes must alternate `lock()` and `release()` calls, starting with `lock()`. We say a process is in the *entry section* if its `lock()` method is pending; it is in the *critical section* if it has completed a `lock()` call but since then not called `release()`; and it is in the *exit section* while its `release()` method is pending. A process that is not in any of the entry, critical, or exit sections is in the *remainder section*.

A lock object provides the *safety* property of *mutual exclusion*, which states that no two processes can be in the critical section at the same time. Several *progress* conditions have been considered for mutual exclusion algorithms. The weakest standard condition is *deadlock freedom*, which guarantees system progress: as long as all processes that are not in the remainder section take sufficiently many steps, some process will enter the critical section.

III. The Algorithm

A. Main Ideas and High Level Description

We start by describing the core ideas of the algorithm. The complete pseudocode is given in Fig. 1, but here we will make some simplifications in order to not distract from the main insights. In particular, our code uses some sequence numbers, which we omit from this description. Also, some of the variables in the pseudocode are indexed by α. We will explain the purpose of this index later, and for now we will omit α from the corresponding variable names.

To decide which process enters the critical section first, we use a simple leader election protocol. The functionality of that is provided by a CAS object, denoted S in our pseudocode, which is initially $\perp$. Each process p executes S.`CAS(`$\perp, p$`)`, and if it succeeds (i.e., the operation returns $\perp$) it becomes the leader, otherwise it loses.

A basic idea (albeit one that does not work without some additional twists) is the following: The losers of the leader election try to "notify" the leader, and if successful, the leader coordinates their passage through the critical section. For this mechanism, we use the notion of a *backpack*. Intuitively, processes try to join the leader's backpack while it is "open". At some point, the leader "closes" its backpack, and then arranges that all processes in the backpack go through the critical section, one by one.

More precisely, the leader w has n distinct registers in its local memory segment, namely, $B[w][0..n-1]$. If process p wants to join w's backpack (after losing the leader election), it writes its ID into $B[w][p]$. A flag, stored in register $A[w]$ in our pseudocode, indicates whether w has already closed the backpack. Hence, after writing its ID to $B[w][p]$, process p checks that flag, and if the backpack is still open, it is guaranteed to be found: After closing the backpack, the leader will scan the array $B[w][\cdot]$, and for every process it finds, it starts a handshaking procedure, coordinating the processes found to the critical section. (We say the leader *promotes* into the critical section each process it finds.) Hence, if process p joins the backpack while the backpack is still open, p can busy-wait on some variable in its local memory segment (in our case $B[p][w]$) which the leader will use to notify p, when it is p's turn to get promoted. Otherwise, if p finds that the backpack is closed when it checks the flag in $A[w]$, then p "gives up", and does not wait for the leader to promote it. Finally, once the leader w has coordinated all processes from its backpack into the critical section, it resets the CAS object S by executing S.`CAS(`$w, \perp$`)`, so that subsequently other processes can become leaders.

While one can easily design a correct algorithm based on the above technique, this technique by itself does not achieve the desired RMR complexity, even against an oblivious

adversary: The adversary could first schedule one process until it becomes the leader and has closed its backpack. After that, and before the leader resets the CAS object S, the adversary schedules all remaining processes to participate in the leader election. These processes will fail to join the backpack, and thus their RMRs are "wasted".

To motivate our second core idea, which deals with this issue, assume for a moment that processes have access to an oracle. After process w becomes the leader, the oracle provides it with exactly one ID, of a process q^* chosen uniformly at random from the set M of processes that are already in the entry section or will enter the entry section before w closes its backpack. Given this information, the leader can busy-wait on $B[w][q^*]$ (which is in w's memory segment) until q^* appears there and thus has joined the backpack. Only after that, does w close its backpack and promotes all the processes it finds in $B[w][\cdot]$ into the critical section. If the random choice of $q^* \in M$ is independent of the adversary's scheduling decision, then we expect that roughly half of the processes in M join w's backpack (i.e, write to $B[w][\cdot]$) before q^* does. Hence, half of the processes in the entry section get promoted (in expectation), and thus for every constant number of RMRs, one process enters the critical section.

We now describe a randomized mechanism that provides a functionality that can replace the oracle above. We use a shared array $R[1..\ell]$, where $\ell = \lfloor \log n \rfloor + 1$, and each array entry is initially $\perp$ (in our pseudocode we use again sequence numbers, so the actual initial value is different). Before participating in the leader election, each process writes its ID to an array entry $R[\lambda]$, where $\lambda \in \{1, \ldots, \ell\}$ is chosen at random in such a way that $\lambda = i$ with probability $\Theta(1/2^i)$. The leader scans this array from left to right until it finds the first index i such that $R[i] = \perp$. Then (slightly simplifying matters) it uses the process ID q^* found in $R[i-1]$ in the same way as the oracle response above, i.e., it waits for q^* to join its backpack. The crucial insight now is the following: Suppose M is the set of processes that write to R before the point t when q^* starts to scan R, and let $m = \lceil \log |M| \rceil$. Then with constant probability the following "good" event happened: all registers $R[0], \ldots, R[m]$ were written by processes in M, and exactly one process in M wrote to $R[m]$.

Given this event, the process that wrote to $R[m]$ is uniformly distributed over M. Hence, by waiting for that process q^* to join its backpack, w ensures that it does not close its backpack before $\Omega(|M|)$ processes have also joined its backpack, in expectation. To deal with some technical issues arising from processes writing to R at different times, and to simplify the analysis, the leader actually waits for *every* process it finds on R, not only the "topmost" one. Clearly, waiting longer cannot make matters worse.

The mechanism above still does not guarantee that all processes have a chance of getting promoted, but only those that write to R during a specific time interval. In particular, a process that writes to R after w scanned that array may be scheduled in such a way that it has no chance of getting promoted. To describe the solution to this problem we use the notion of "good" intervals. Recall that the CAS object S gets reset to $\perp$ whenever a leader finishes its exit section, before it gets captured by the next leader. A good interval starts whenever the CAS object gets reset, and ends just before the next leader starts scanning R. As argued above, our technique guarantees that if M is the set of processes that write to R during a good interval, then in expectation $\Omega(|M|)$ processes get promoted by the one that becomes the leader in that interval.

We employ the following simple trick: We use two copies of essentially the entire data structure (i.e., of almost all shared objects). In the pseudocode, we add a subscript α to each shared object, where the value of $\alpha \in \{0, 1\}$ indicates the copy of the object. (We say "side α" to refer to copy α of the data structure.) At the beginning of its entry section, a process chooses a side $\alpha \in \{0, 1\}$ uniformly at random. Then, it proceeds as described above, but uses side α of the data structure. Since there are also two CAS objects, S_0 and S_1, we may now have two competing leaders. To synchronize between them, we use an additional 2-process lock object, L (in fact, for technical reasons, we need that L be a 4-process lock). As soon as a process becomes the leader of side α (i.e., it captures S_α), it tries to capture L; and when it has captures L, it releases it only after it resets S_α. As a consequence, every point in time belongs to either a good interval on side 0, or a good interval on side 1. Hence, since processes choose α at random, whenever they write to R_α they have (at least) a 1/2 probability of writing during a good interval on side α.

There are several small technical difficulties to overcome in order to make these ideas work. Many of the difficulties stem from the fact that information on R may be outdated, i.e., it is left behind by processes that did not manage to join a backpack. To deal with this and other issues we use sequence numbers that processes increment frequently, and attach to the information they write to registers. Techniques to recycle sequence numbers are known [1], [20] and well understood. It is not difficult to bound sequence numbers so that our algorithm needs only $O(\log n)$-bit numbers, but doing so makes the implementation more complicated and distracts from the core-ideas.

B. Implementation

The pseudocode of our algorithm is given in Fig. 1.

We use array $A[0..n-1]$ and, for $s \in \{0, 1\}$, arrays $B_s[0..n-1][0..n-1]$ and $R_s[1..\ell]$, where $\ell = \lfloor \log n \rfloor + 1$. Array A_s is used by processes to keep track of their sequence number and communicate their "status". Array B_s implements the backpack functionality, and R_s is used for the random "oracle" mechanism described earlier. Registers

Variables and Notation:
$\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ is the set of non-negative integers, and $\mathcal{P} = \{0, \ldots, n-1\}$ is the set of processes. Lower case Latin and Greek symbols indicate (process-)local variables; the names of shared objects are capitalized. The scope of all process-local variables is global, i.e., they maintain their values between method calls. All shared variables are remote to all processes, except for array B, where $B[p][q]$ is in p's local memory segment.

Shared Objects:

- $A[0..n-1]$ is an array of registers, each storing a pair (seq, str), where $seq \in \mathbb{N}_0$ and $str \in \{want, done\} \cup (\mathcal{P} \times \mathbb{N}_0)$. Each array entry is initially $(0, done)$.
- $B_s[0..n-1][0..n-1]$, for $s \in \{0,1\}$, is an array of registers, each storing a pair $(seq, stat)$, where $seq \in \mathbb{N}_0$ and $stat \in \{trying, waiting, promoted, done\}$. Each array entry is initially $(0, done)$. Subarray $B[p][\cdot]$ is in process p's memory segment.
- $R_s[1..\ell]$, for $s \in \{0,1\}$ and $\ell = \lfloor \log n \rfloor + 1$ is an array of registers, each storing a pair in $\mathcal{P} \times \mathbb{N}_0$ that is initially $(0,0)$.
- S_s, for $s \in \{0,1\}$ is a CAS object which stores values in $(\mathcal{P} \times \mathbb{N}_0) \cup \{(\perp, \perp)\}$, and is initially $(\perp, \perp)$.
- L is a 4-process lock (`4PLock`) which can be accessed by the processes $0, \ldots, 3$.
- Bit_s, for $s \in \{0,1\}$, is a Boolean register that is initially 0.

lock$_p$():

```
1  while True do
2      c := A[p].seq + 1;   A[p] := (c, want)                                   //Announce that I want the lock
3      Choose α : Pr(α = j) = 1/2 for j ∈ {0,1}                                 //Choose a side uniformly at random
4      Choose λ : Pr(λ = j) = 2^{-j} for j ∈ {1,...,ℓ-1}, and Pr(λ = ℓ) = 2^{-ℓ+1}   //Choose a random location in R_α
5      R_α[λ] := (p, c)                                                         //Write to that location
6      (w, d) := S_α.CAS((⊥,⊥), (p,c))                                          //Try to become a leader
7      if w = ⊥ then                                                            //I am the leader
8          bit := Bit_α                                                         //Read bit to compute parity for access on L
9          L.lock(2α + bit)                                                     //Prevent anyone else from executing the code below

           //Collect processes from R_α:
10         found := ∅
11         for j = 1, ..., ℓ do
12             (r, d) := R_α[j]
13             if A[r] ∉ {(d, want), (d, (p, c))} then break                    //Break if r won't see me
14             found := found ∪ {(r, d)}

15         for each (r, d) ∈ found − {(p, c)} do                                //Wait for each process in found to join my backpack
16             await(B_α[p][r].seq ≥ d)
17         promote_p()                                                          //Promote processes that have applied in time
18         A[p] := (c, done)                                                    //Close backpack
19         promote_p()                                                          //Promote late processes
20         return                                                               //Enter the critical section
21     else                                                                     //I failed to become a leader
22         A[p] := (c, (w, d))                                                  //Announce that I'm trying to enter w's backpack
23         (w, d) := S_α.read()                                                 //Perhaps there's a new leader?
24         B_α[w][p] := (c, trying)                                             //Join w's backpack
25         if A[w] = (d, want) then                                             //The backpack is still open
26             B_α[w][p] := (c, waiting)                                        //Tell w that I'm ready to get promoted
27             await(B_α[p][w] = (c, promoted))                                 //Wait to be promoted
28             return                                                           //Enter the critical section
29         else B_α[w][p] := (c, done)                                          //May be too late for promotion, better give up
```

release$_p$():

```
30 if w = ⊥ then                                                                //I am the leader
31     Bit_α := 1 − bit                                                         //Force next leader on side α to use different ID on L
32     S_α.CAS((p, d),(⊥,⊥))                                                    //Reset the CAS object
33     L.release(2α + bit)                                                      //Release lock L
34 else B_α[w][p] := (c, done)                                                  //Notify leader that I left the critical section
```

promote$_p$():

```
35 for r = 0, ..., n − 1 do                                                     //Scan through backpack
36     d := B_α[p][r].seq                                                       //Get r's sequence number
37     await(B_α[p][r] ≠ (d, trying))                                           //Wait for r's decision: promote?
38     if B_α[p][r] = (d, waiting) then                                         //r wants promotion
39         B_α[r][p] := (d, promoted)                                           //Promote process q
40         await(B_α[p][r] ≠ (d, waiting))                                      //Wait until process q is done
```

Figure 1. Lock Implementation.

$B_s[p][q]$ are in process p's local memory segment, and all other registers are remote to all processes. We also use compare&swap objects S_s, $s \in \{0,1\}$, to elect leaders, and registers Bit_s to indicate the parity of the number of leaders that have been elected on S_s. (The reason for this will be explained below.) Finally, we use a 4-process lock L. To capture L, a process calls L.`lock(`i`)`, where i is a *virtual* ID in the set $\{0,1,2,3\}$.

In line 2 of the `lock()` method, process p increments its sequence number stored in $A[p].seq$ and writes the pair $(c, want)$ to $A[p]$, where c is the incremented sequence number. The status indicates that p has started the entry section. Then, in lines 3 and 4, p chooses $\alpha \in \{0,1\}$ uniformly at random, and $\lambda \in \{0,\ldots,\ell\}$ according to a geometric probability distribution that guarantees that $\lambda = i$ with probability $\Theta(1/2^i)$. In line 5, p writes its ID/sequence-number pair (p,c) to $R_\alpha[\lambda]$. Then, in line 6, p tries to compare-and-swap (p,c) into S_α.

If p's `CAS()` succeeds, p becomes the leader on side α. In that case, it reads Bit_α into bit and then tries to capture lock L (in lines 8 and 9), using a virtual 2-bit ID with high-order bit α and low-order bit bit. Then, in lines 10–14, the process scans array R_α, from left to right. Each time it finds a process/sequence-number pair (r,d), it checks the status of process r in $A[r]$. If that status is $want$ or (p,c), then it is guaranteed that r will get promoted. All such pairs are added to a local set $found$, and as soon as the leader sees a process on R_α that does not meet the above criterion, it stops scanning R_α. In lines 15 and 16, the leader p waits for each process r added to $found$ (except for itself), until r has joined p's backpack by writing to $B_\alpha[p][r]$. In line 17, p calls `promote()`, which is a method that promotes processes that wrote to $B_\alpha[p][\cdot]$, i.e., coordinates their entry into the critical section. The `promote()` method guarantees that p will successfully promote each process r that, before the invocation of that method, wrote to $B_\alpha[p][r]$ the pair $(c, trying)$ or $(c, waiting)$, where c is p's current sequence number. Upon termination of `promote()`, all processes that p promoted will have exited the critical section. In lines 18 and 19, p first closes its backpack by writing $(c, done)$ to $A[p]$, and then calls `promote()` again. This second promotion call ensures that all processes that joined p's backpack before p closed it, will get promoted. After that, p enters the critical section.

Now suppose p lost the leader election, i.e., its S_α.`CAS()` in line 6 returned (w,d), where w is the leader of the `CAS()` operation at the time, and d is its sequence number. Then, in line 22, p writes the pair $(c,(w,d))$ to $A[p]$ to indicate its new status, namely that it now aims to join w's backpack. In line 23, p reads a pair (w,d) from S_α again, in case the leader has changed after p updated its status. This avoids deadlock, because otherwise p might try to join the backpack of an "old" leader, while the "new" leader may have seen $(c, want)$ when it read $A[p]$ in line 13, and thus might try to promote p. Then, in line 24, p tries to enter w's backpack by writing $(c, trying)$ to $B_\alpha[w][p]$. In line 25, it reads $A[w]$ to check whether w's backpack is still open. If not, p "gives up" and indicates that is not trying to get promoted anymore by writing $(c, done)$ to $B_\alpha[w][p]$, in line 29. Otherwise, in lines 26 and 27, p indicates that it is ready to get promoted by writing $(c, waiting)$ to $B_\alpha[w][p]$, and then it busy-waits on $B_\alpha[p][w]$ until w promotes it. After that p can enter the critical section by returning from its `lock()` call, in line 28.

In method `release()`, a loser p simply indicates in line 34 that it is done with the critical section by writing $(c, done)$ to $B_\alpha[w][p]$, where w is the current leader that promoted p. If p is a leader, then it executes lines 31–33. First it flips the bit Bit_α, then it resets the CAS object S_α to the initial value $(\perp,\perp)$, and finally it releases lock L. Note that it releases L only after resetting S_α. This is necessary to ensure that there is always a good interval, either on side 0 or side 1 (see the high level description in Section III-A). However, this also means that a new leader may get elected on side α before the previous leader on side α has released L. Since processes use the bit Bit_α to compute their virtual IDs, it is ensured that access to lock L is still safe. (This is the reason why we need to use a 4-process lock L, instead of a 2-process one.)

Method `promote()` uses a straightforward handshaking mechanism to facilitate the promotion. The leader p scans array $B_\alpha[p][\cdot]$, and waits for each process r until either its sequence number stored in $B_\alpha[p][r]$ changes, or until r is not trying to get promoted, as indicated by $B_\alpha[p][r].status$ (lines 36 and 37). When this happens, r has made a decision whether it still wants to be promoted, by writing to that array entry that it is waiting for promotion, in line 26, or that it has given up, in line 29. (In the latter case, r may also subsequently have increased the sequence number stored in $B_\alpha[p][r].seq$, if meanwhile it started another attempt.) If r still wants to be promoted, it wrote $(d, waiting)$, and is now busy-waiting in line 27. In this case, in line 39 process p notifies process r that it can enter the critical section, and then in line 40, p waits until r writes to $B_\alpha[p][r]$ again; r will do so at the end of its exit section in line 34.

IV. Complexity Analysis

In this section we analyze the RMR complexity of our algorithm. The proofs of deadlock freedom and mutual exclusion, as well as the proofs of some claims used for the complexity analysis are omitted due to space restrictions.

We start with some terminology and some simple facts. A subscript attached to a local variable indicates the process to which the local variable belongs to; e.g., c_x denotes local variable c of process x.

We consider an execution of the algorithm, and fix linearization points of all `CAS()` operations in an arbitrary but unique way. When we say a process *executes* a `CAS()` operation op at time t, we mean that t is the linearization

point of op. Sometimes we talk about the value that the CAS object has at a certain point; with that we mean the value it would have at that point, if all `CAS()` operations occurred atomically at their linearization points.

For $s \in \{0,1\}$ and $i \geq 0$, we define a *phase* (s,i). Phase $(s,0)$ starts at the beginning of the execution, and phase (s,i), for $i \geq 1$, starts when for the i-th time a S_s.`CAS()` operation in line 32 linearizes. Phase (s,i) ends when phase $(s,i+1)$ starts. The *leader* of phase (s,i) is the unique process p that executes a successful S_s.`CAS()` operation in line 6 which linearizes during phase (s,i). We say process x gets *promoted* during a `promote()` call by process y, if x enters the critical section while y's `promote()` call is pending. Note that y must own lock L when it calls `promote()`, so no two `promote()` calls can overlap, and thus a process can only get promoted during a single `promote()` call.

Theorem 1. *Consider the random execution of the algorithm scheduled by a locality-aware adversary, and let τ_m be the point when the implemented lock method has been invoked $m \geq 1$ times. The expected total number of RMRs incurred until point τ_m is $O(m)$.*

W.l.o.g. we assume that after point τ_m, the adversary does not schedule any new invocations to the `lock()` method; it only schedules processes that have already started a `lock()` operation until all such pending operations are completed. Let τ'_m denote that completion point. We will bound the number of RMRs incurred until point τ'_m, as this is clearly an upper bound on the RMRs incurred until τ_m. The next lemma says that it suffices to bound instead the number of writes to arrays R_s in line 5.

Lemma 2. *The total number of RMRs incurred until point τ'_m is $O(W_m)$, where W_m is the total number of write operations on the two arrays R_0 and R_1 until τ'_m.*

Proof: We consider the number of RMRs incurred by process p in a single iteration of the while-loop in the `lock()` method. Up to (and including) line 7, p incurs $O(1)$ RMRs. The number of RMRs incurred in the rest of the while-loop iteration depends on whether or not p becomes a leader (i.e., wins the CAS object S_{α_p} in line 6). If p does not become a leader, then it executes the else-part of the if-else statement, in lines 21–21, which requires just $O(1)$ RMRs. If p does become a leader, then it executes lines 8–20, which may incur more than $O(1)$ RMRs. Precisely, more than $O(1)$ RMRs may be needed in the for-loop in lines 11–11, and also in each of the two `promote()` operations in lines 17 and 19 (the remaining operations incur $O(1)$ RMRs). In each iteration of the for-loop in lines 11–11, p incurs two RMRs, in lines 12 and 13, and in each `promote()` operation, an RMR is incurred every time p executes line 39. We explain next how to charge those RMRs to other processes, without charging more than $O(1)$ RMRs in total to each process, per iteration of the while-loop.

Claim 3. *Let t be some point at which process w^* has finished the for-loop in lines 11–11 but not yet started the loop in lines 15 and 16. If $(r^*, d^*) \in found_{w^*}$ at point t, then at this point r^* executes some iteration of the while-loop in a `lock()` operation, and enters the critical section in the same iteration while $c_{r^*} = d^*$.*

From Claim 3, it follows that in each iteration of the for-loop in lines 11–11, except possibly for the last one, process p adds to set $found_p$ a distinct pair (r,d), not added to any other $found$ set. For each such pair (r,d), we have that process r executes a different iteration of the while-loop in the `lock()` method. We charge to r the two RMRs incurred in the for-loop iteration in which (r,d) is added to $found_p$, and charge to p the two RMRs of the last for-loop iteration. It follows that for each iteration of the while-loop executed by r or p, this process is charged at most $O(1)$ of the RMRs incurred (by any process) in lines 11–11.

Claim 4. *If during a `promote()` operation process w^* finishes line 40 $k \geq 1$ times, then during that `promote()` call at least k `lock()` calls respond and k `release()` calls get invoked.*

Claim 4 allows us to distribute the RMR cost of a `promote()` operation to the processes that go through the critical section during that call, charging one RMR to each process q for each of its entries to the critical section. Since only one `promote()` operation can be in progress at a time (as the leader must own lock L during the call), no process is charged twice for the same passage through the critical section. Finally, we observe that each `release()` operation incurs $O(1)$ RMRs.

Combining the above, we obtain that the total number of RMRs is asymptotically the same as the total number of iterations of the while-loop in the `lock()` method, executed by all processes, plus the total number of passages of processes through the critical section. Since for each passage, a process must execute at least one iteration of the while-loop, we conclude that the total number of RMRs is asymptotically the same as the total number of iterations of the while-loop, which is equal to the number of write operations on R_0 and R_1. This completes the proof of Lemma 2. ∎

Next we introduce the notions of *good intervals* and *good writes*, and show that in expectation at least half of the writes to arrays R_0 and R_1 are good. Thus, it suffice to bound the number of good writes.

For $s \in \{0,1\}$ and $i \geq 0$, the *good interval* $I_{s,i}$ starts at the beginning of phase (s,i), and ends when the L.`lock()` operation in line 9 by the leader of phase (s,i) responds (i.e., when the leader has acquired that lock). A write operation on array R_s in line 5 is *good* if it takes place in some good interval $I_{s,i}$.

Good intervals $I_{s,i}$ and $I_{s,i'}$ with $i \neq i'$ do not overlap, but good intervals for different sides s may overlap. A critical observation for our analysis is that the union of all good intervals covers the complete execution. Since each process chooses the side α at random before writing to R_α, with probability 1/2 that write will occur during a good interval on side α. A straight-forward application of Wald's Theorem yields the following lemma.

Lemma 5. *In expectation, at least half of all write operations on arrays R_0 and R_1 are good.*

Next we look at a single phase (s, i), and bound from below the number of times processes go through the critical section during that phase, in terms of the number of good writes to R_s in the phase. Let $k_{s,i}$ be the number of good writes to array R_s in phase (s, i), and let $\ell_{s,i}$ be the number of passages through the critical section by processes in phase (s, i); if phase (s, i) does not exists, then $k_{s,i} = \ell_{s,i} = 0$.

Lemma 6. $\mathbf{E}[\ell_{s,i}] \geq \mathbf{E}[k_{s,i}]/36$.

Proof: We first give an overview of the proof. We fix the set of processes that perform a good write to array R_s during phase (s, i), and condition on the event $\mathcal{E}$ that all the first $\kappa = \lfloor \log(k_{s,i}) \rfloor$ positions in R_s get written by those good writes. Event $\mathcal{E}$ has constant probability, and implies that the leader of phase (s, i) will execute at least κ iterations of the for-loop comprising lines 11–11; and for each $1 \leq j \leq \kappa$, it will add to its $found$ set some process that wrote to $R_s[j]$ after the beginning of the phase. The leader will then have to wait in line 16 until all these processes have executed line 24, and in particular until the process p^* that wrote to $R_s[\kappa]$ has done so. Given $\mathcal{E}$, the conditional distribution of the position λ in which a process writes to R_s in phase (s, i) is very close to the unconditional one, described in line 4. In particular, the probability that a process writes to the κ-th position $R_s[\kappa]$ is $O(1/2^\kappa) = O(1/k_{s,i})$. It follows that any schedule by the adversary will result in an expected number of at least $\Omega(k_{s,i})$ processes that write to R_s in phase (s, i), and execute line 24 before process p^* does. All these processes will be promoted to the critical section before the end of the phase, as the leader has to wait for p^* in line 16 before it invokes a `promote()` operation in line 17, and thus it will see those $\Omega(k_{s,i})$ processes (in expectation) when it scans through its backpack.

We now give the detailed proof. We define three sets of processes, $K_{s,i}$, $M_{s,i}$, and $P_{s,i}$, as follows. Set $K_{s,i}$ consists of all processes that perform a good write to R_s during phase (s, i). Set $M_{s,i}$ consists of the processes that write to R_s between the beginning of phase (s, i), and the point right after the leader of the phase has either completed the κ-th iteration of the for-loop in lines 11–11, where $\kappa := \lfloor \log(k_{s,i}) \rfloor$, or has broken out of the loop in line 13 (whichever of the two happens first). Note that $K_{s,i} \subseteq M_{s,i}$. Finally, set $P_{s,i}$ contains all processes $p \in M_{s,i}$ that write to array B_s in line 24 before the leader invokes the `promote()` operation in line 17.

We let $\mathcal{E}$ be the event that for each position $1 \leq j \leq \kappa$ of array R_s, at least one of the $k_{s,i}$ good write operations on R_s during phase (s, i) is performed on register $R_s[j]$.

The following claim establishes that no process, which writes to R_s in phase (s, i), can proceed past line 27, or, unless it is the leader of that phase, read an entry of R_s, before the leader of that phase executes line 17.

Claim 7. *Let t be some point after the leader w^* of phase (s, i) finished line 9, but has not yet started line 17, and let c^* be the value of c_{w^*} at that point. Then at point t,*

(a) no process other than w^* has performed a read operation on R_s since the beginning of the phase; and

(b) for each process $q \neq w^*$ that has written to R_s at some point $t' < t$ during phase (s, i):

(b1) at point t, q is poised to execute a shared memory step in line 6 or in one of lines 22–27; and

(b2) if in interval $[t', t]$ process q executes a `read()` or `CAS()` operation on S_s in line 6 or 23, respectively, then this operation returns (w^*, c^*), and if it executes line 25 during $[t', t]$, then the if-condition in that line evaluates to true.

Between the beginning of phase (s, i) and the point when good interval $I_{s,i}$ ends, the leader does not access R_s. Claim 7 implies that the position λ in R_s, where a process $p \in K_{s,i}$ writes to when it executes line 5 during $I_{s,i}$ does not affect any other processes' steps, or p's steps starting with line 5 and until interval $I_{s,i}$ ends. Since the adversary does not know which position λ a process p chooses, and only the location of p's write to R_s depends on λ, that random choice does not affect the schedule up to the point interval $I_{s,i}$ ends.

Fix some execution prefix E that ends at the beginning of good interval $I_{s,i}$. In addition, fix all remaining random choices made by processes until the end of $I_{s,i}$, except for the choice of λ that each process makes on side s. Then for every infinite sequence $\vec{\lambda} = (\lambda_1, \lambda_2, \dots)$, the adversary schedules an execution that is uniquely determined by $\vec{\lambda}$ up to the end of $I_{s,i}$, where the j-th process that executes line 4 on side s during $I_{s,i}$ chooses the value λ_j in that line. Let $E_{\vec{\lambda}}$ denote that unique execution up to the point when $I_{s,i}$ ends. Then we have that for any two $\vec{\lambda}, \vec{\lambda'}$, the adversary cannot distinguish between $E_{\vec{\lambda}}$ and $E_{\vec{\lambda'}}$, and the only difference any process sees (if any) is the random value it chooses in line 4 on side s during $I_{s,i}$. Since the adversary cannot distinguish between those two executions, $K_{s,i}$ is the same for both, and so is the order of all steps. Hence, κ is also fixed, and the probability of event $\mathcal{E}$, that each of the values $1, \dots, \kappa$ is chosen at least once by processes in $K_{s,i}$, is

$$\Pr(\mathcal{E}) = 1 - \Pr(\bar{\mathcal{E}}) \geq 1 - \sum_{1 \leq i \leq \kappa} (1 - 1/2^i)^{k_{s,i}} \geq$$
$$1 - \sum_{1 \leq i \leq \kappa} e^{-k_{s,i}/2^i} \geq 1 - \sum_{j \geq 1} e^{-j} = \frac{e-2}{e-1} > \frac{1}{3}. \quad (1)$$

(We used that $\kappa = \lfloor \log(k_{s,i}) \rfloor$.)

Claim 8. *If event $\mathcal{E}$ occurs, then the leader of phase (s,i) does not break out of the for-loop in lines 11–11 during the first κ iterations.*

Let $\kappa' \geq \kappa$ be the number of iterations of the for-loop in lines 11–11 completed by the leader of phase (s,i), without including the last iteration if it ends with a break. Let T be the point right after the end of the last iteration of the for-loop.

In the following we condition on event $\mathcal{E}$ and also on the value of κ'. Note that $\mathcal{E}$ implies $\kappa' \geq \kappa$, by Claim 8.

We are interested in the conditional distribution of the λ-value of each process $p \in M_{s,i}$, given $\mathcal{E}$ and $\kappa' = k$, for $k \geq \kappa$. At point T the adversary has no knowledge of those λ-values, except for what can be inferred from the value of κ'. Hence, conditionally on events $\mathcal{E}$ and $\kappa' = k$, the value of λ chosen by each $p \in M_{s,i}$ has no effect on the schedule and thus on p's steps during the interval starting with p's good write to R_s and ending at point T. Then, for each $p \in M_{s,i}$ and $1 \leq j \leq \kappa$, the probability that a given $p \in M_{s,i}$ chooses $\lambda = j$ is

$$\Pr(\lambda = j \mid \mathcal{E}, \kappa' = k) \leq \Pr(\lambda = j \mid \mathcal{E}, \kappa' = \kappa)$$
$$= \frac{\Pr(\lambda = j \wedge \mathcal{E} \mid \kappa' = \kappa)}{\Pr(\mathcal{E} \mid \kappa' = \kappa)}.$$

Since

$$\Pr(\lambda = j \wedge \mathcal{E} \mid \kappa' = k) \leq \Pr(\lambda = j \mid \kappa' = \kappa)$$
$$\leq \Pr(\lambda = j \mid \lambda \leq \kappa) \leq (1/2^j)/(1 - 1/2^j),$$

and $\Pr(\mathcal{E} \mid \kappa' = \kappa) \geq \Pr(\mathcal{E}) \geq 1/3$, by (1), it follows

$$\Pr(\lambda = j \mid \mathcal{E}, \kappa' = k) \leq (3/2^j)/(1 - 1/2^j). \quad (2)$$

The next claim says that the leader will wait in line 16 until all processes in its $found$ set (other than the leader itself) have executed line 24.

Claim 9. *Let t_1 be the point when phase (s,i) begins, and $t_2 > t_1$ the point during phase (s,i) when the leader of that phase, w^*, finishes the for-loop in line 16. If at point t_2, $(r^*, d^*) \in found_{w^*}$ and $r^* \neq w^*$, then r^* writes $(d^*, trying)$ to $B_s[w^*][r^*]$ at some point $t^* \in [t_1, t_2]$.*

We are interested in the total number of processes $p \in M_{s,i}$ that execute line 24 until all processes in the leader's $found$ set (other than the leader) have executed that line. This is lower bounded by the number Y of processes $p \in M_{s,i}$ that execute line 24 until the first process $p^* \in M_{s,i}$ with $\lambda_{p^*} = \kappa$ executes that line, provided that the λ-value of the leader is not κ.

After point T, the leader executes the loop in lines 15–16 throughout which all shared memory steps are reads of registers in the leader's own local memory segment, and thus do not incur RMRs. Hence, the adversary does not gain any information about how many iterations of the for-loop have been executed by the leader, until the leader finishes that loop. Therefore, to determine Y we can assume that the λ-value of each process $p \in M_{s,i}$ (other than the leader) remain unknown until the leader has finished its loop in lines 15–16. Given $\mathcal{E}$ and κ', the probability that any given process $p \in M_{s,i}$ (including the leader) chooses $\lambda = \kappa$ is at most $\pi = (3/2^\kappa)/(1 - 1/2^\kappa)$, by (1). From the union bound, the probability that neither the leader nor any of the first $j-1$ processes $p \in M_{s,i}$ that execute line 24 choose $\lambda = \kappa$ is at least $1 - j\pi$. Thus, $E[Y \mid \mathcal{E}, \kappa']$ is at least

$$\sum_{j \geq 1} \max\{0, 1 - j\pi\} \geq \frac{2^\kappa(1 - 1/2^\kappa)}{6} - \frac{1}{2}.$$

Therefore, the expected number of processes $p \in M_{s,i}$ that execute line 24 before the leader initiates a promotion is $\mathbf{E}[|P_{s,i}| \mid \mathcal{E}] \geq \frac{2^\kappa(1-1/2^\kappa)}{6} - \frac{1}{2}$, and thus

$$\mathbf{E}[|P_{s,i}|] \geq \mathbf{E}[|P_{s,i}| \mid \mathcal{E}] \cdot \Pr(\mathcal{E}) \overset{(1)}{\geq} \frac{2^\kappa(1 - 1/2^\kappa)}{18} - \frac{1}{6}. \quad (3)$$

The next claim says that the leader and all $p \in P_{s,i}$ go through the critical section in phase (s,i). cl

Claim 10. *The leader of phase (s,i) finishes its* `lock()` *operation in the same iteration of its while-loop in which it becomes the leader, and each process $q \in P_{s,i}$ in the iteration of its while-loop in which it writes to R_s for the first time during phase (s,i).*

Since the leader does not belong to $P_{s,i}$, it follows from Claim 10 and (3), that the expected total number of processes that go through the critical section in phase (s,i) is at least $\frac{2^\kappa(1-1/2^\kappa)}{18} - \frac{1}{6} + 1 \geq k_{s,i}/36$, as $\kappa = \lfloor \log(k_{s,i}) \rfloor$. This completes the proof of Lemma 6. ∎

We now have all the pieces we need to prove the main result of the section, the bound on the total number of RMRs.

Proof of Theorem 1: Recall, we have assumed w.l.o.g. that the adversary schedules exactly m invocations to the implemented `lock()` operation. We will bound the total number of RMRs until the point τ'_m when all those operations are completed. The total number of passages by processes through the critical section is also m. Thus, $m = \sum_{s,i} \ell_{s,i}$, where the summation is over all $s \in \{0,1\}$ and $i \geq 0$ ($\ell_{s,i} = 0$ if phase (s,i) does not exist). Since m is fixed and finite, it follows $m = \mathbf{E}[\sum_{s,i} \ell_{s,i}] = \sum_{s,i} \mathbf{E}[\ell_{s,i}]$.. Using that $\mathbf{E}[\ell_{s,i}] \geq \mathbf{E}[k_{s,i}]/36$, from Lemma 6, gives $m \geq \sum_{s,i} \mathbf{E}[k_{s,i}]/36 = \mathbf{E}[\sum_{s,i} k_{s,i}]/36$.

Further, from Lemma 5, we have for the total number W_m of writes to arrays R_0 and R_1, $\mathbf{E}[W_m] \leq 2\,\mathbf{E}[\sum_{s,i} k_{s,i}]$. Combining this with the inequality above, yields $\mathbf{E}[W_m] \leq 72m$. The theorem now follows from this and Lemma 2, which bounds the total number of RMRs by $O(W_m)$. ■

V. Conclusion

For the CC model, there is currently no randomized algorithm that achieves constant RMR complexity. We believe that our techniques can be extended to the CC model. To achieve this, we are working on a mechanism that allows processes to join the backpack of a leader in a similar way as in our DSM algorithm. The naive algorithm requires the leader to scan an array of size n and incurs $\Omega(n)$ RMRs in the CC model, but we have a randomized technique that achieves something similar in $O(1)$ RMRs. However, it is a technical challenge to combine this with the "oracle" mechanism of our DSM algorithm.

Acknowledgements

This research was undertaken, in part, thanks to funding from the Canada Research Chairs program, the Discovery Grants program of NSERC, and the INRIA Associate Team RADCON.

References

[1] Z. Aghazadeh, W. Golab, and P. Woelfel, "Making objects writable," in *Proc. 33rd PODC*, 2014, pp. 385–395.

[2] D. Alistarh and J. Aspnes, "Sub-logarithmic test-and-set against a weak adversary," in *Proc. 25th DISC*, 2011, pp. 97–109.

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

[4] J. Anderson and Y.-J. Kim, "Fast and scalable mutual exclusion," in *Proc. 13th DISC*, 1999, pp. 180–194.

[5] ——, "Adaptive mutual exclusion with local spinning," in *Proc. 14th DISC*, 2000, pp. 29–43.

[6] ——, "An improved lower bound for the time complexity of mutual exclusion." *Distributed Computing*, vol. 15, pp. 221–253, 2002.

[7] T. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, pp. 6–16, 1990.

[8] J. Aspnes, "Faster randomized consensus with an oblivious adversary," in *Proc. 31st PODC*, 2012, pp. 1–8.

[9] H. Attiya, D. Hendler, and S. Levy, "An $O(1)$-barriers optimal RMRs mutual exclusion algorithm," in *Proc. 31st PODC*, 2013, pp. 220–229.

[10] H. Attiya, D. Hendler, and P. Woelfel, "Tight RMR lower bounds for mutual exclusion and other problems," in *Proc. 40th STOC*, 2008, pp. 217–226.

[11] M. Bender and S. Gilbert, "Mutual exclusion with $O(\log^2 \log n)$ amortized work," in *Proc. 52nd FOCS*, 2011, pp. 728–737.

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

[13] R. Cypher, "The communication requirements of mutual exclusion," in *Proc. 7th SPAA*, 1995, pp. 147–156.

[14] R. Danek and W. Golab, "Closing the complexity gap between FCFS mutual exclusion and mutual exclusion," *Distributed Computing*, vol. 23, no. 2, pp. 87–111, 2010.

[15] R. Danek and H. Lee, "Brief announcement: Local-spin algorithms for abortable mutual exclusion and related problems," in *Proc. 22nd DISC*, 2008, pp. 512–513.

[16] E. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, p. 569, 1965.

[17] R. Fan and N. Lynch, "An $\Omega(\log n)$ lower bound on the cost of mutual exclusion," in *Proc. 25th PODC*, 2006, pp. 275–284.

[18] G. Giakkoupis and P. Woelfel, "On the time and space complexity of randomized test-and-set," in *Proc. 31st PODC*, 2012, pp. 19–28.

[19] ——, "A tight RMR lower bound for randomized mutual exclusion," in *Proc. 44th STOC*, 2012, pp. 983–1002.

[20] W. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel, "RMR-efficient implementations of comparison primitives using read and write operations," *Distributed Computing*, vol. 25, no. 2, pp. 109–162, 2012.

[21] W. Golab, L. Higham, and P. Woelfel, "Linearizable implementations do not suffice for randomized distributed computation," in *Proc. 43rd STOC*, 2011, pp. 373–382.

[22] D. Hendler and P. Woelfel, "Adaptive randomized mutual exclusion in sub-logarithmic expected time," in *Proc. 29th PODC*, 2010, pp. 141–150.

[23] ——, "Randomized mutual exclusion with sub-logarithmic RMR-complexity," *Distributed Computing*, vol. 24, no. 1, pp. 3–19, 2011.

[24] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufman, 2008.

[25] P. Jayanti, "Adaptive and efficient abortable mutual exclusion," in *Proc. 22nd PODC*, 2003, pp. 295–304.

[26] P. Jayanti, S. Petrovic, and N. Narula, "Read/write based fast-path transformation for FCFS mutual exclusion," in *Proc. 31st SOFSEM*, 2005, pp. 209–218.

[27] Y.-J. Kim and J. Anderson, "A time complexity bound for adaptive mutual exclusion," in *Proc. 15th DISC*, 2001, pp. 1–15.

[28] ——, "Nonatomic mutual exclusion with local spinning," *Distributed Computing*, vol. 19, no. 1, pp. 19–61, 2006.

[29] H. Lee, "Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model," in *Proc. 25th ICDCS*, 2005, pp. 261–270.

[30] N. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[31] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[32] A. Pareek and P. Woelfel, "RMR-efficient randomized abortable mutual exclusion," in *Proc. 26th DISC*, 2012, pp. 267–281.

[33] M. Raynal, *Algorithms for Mutual Exclusion.* MIT Press, 1986.

[34] J.-H. Yang and J. Anderson, "A fast, scalable mutual exclusion algorithm," *Distributed Computing*, vol. 9, no. 1, pp. 51–60, 1995.