An O(1)-Barriers Optimal RMRs Mutual Exclusion Algorithm[*]

(Extended Abstract)

Hagit Attiya
Department of Computer-Science
Technion
hagit@cs.technion.ac.il

Danny Hendler
Department of Computer-Science & Telekom Innovation Laboratories
Ben-Gurion University of the Negev
hendlerd@cs.bgu.ac.il

Smadar Levy
Department of Computer-Science
Ben-Gurion University of the Negev
raykgold@bgu.ac.il

ABSTRACT

Mutual exclusion is a fundamental coordination problem. Over the last 20 years, shared-memory mutual exclusion research focuses on *local-spin* algorithms and uses the *remote memory references* (RMRs) metric.

To ensure the correctness of concurrent algorithms in general, and mutual exclusion algorithms in particular, it is often required to prohibit certain re-orderings of memory instructions that may compromise correctness, by inserting *memory barrier* instructions. Memory barriers incur non-negligible overhead and may significantly increase the algorithm's time complexity.

This paper presents the first read/write mutual exclusion algorithm with asymptotically optimal complexity under both the RMRs and barriers metrics: each passage through the critical section incurs $O(\log n)$ RMRs and a constant number of barriers. The algorithm works in the popular *Total Store Ordering* model.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Concurrent programming*; F.1.2 [**Computation By Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General Terms

Algorithms, Architecture

Keywords

Shared memory, mutual exclusion, total store ordering

[*]Partially supported by the Israel Science Foundation (grant number 1227/10)

1. INTRODUCTION

Concurrent algorithms and, in particular, mutual exclusion (mutex) algorithms, are almost always designed under the assumption that memory accesses are atomic, i.e. linearizable [11], or at least sequentially consistent [18]. In practice, however, modern compilers optimize code so as to issue certain instructions out of order, based on the memory model supported by the architecture. The memory model dictates which operation pairs can be reordered [1, Figure 8]. For example, the *total store ordering* (*TSO*) model [19] ensures that writes are not reordered, but it is possible to perform a read from address a before a write to address $b \neq a$ that is earlier in program order is performed. The TSO model is supported by several common architectures, including SPARC [19] and x86 [12].[1] This model is weaker than sequential consistency, and hence, also weaker than linearizability.

To ensure the correctness of a concurrent algorithm, it is possible to prohibit the reordering of memory instructions, by inserting a *barrier* (also called a *fence*) instruction between the memory instructions. Inserting a barrier incurs significant overhead, but it is unavoidable since it has been shown [3] that every mutex algorithm must ensure that there is *a read after a write to a different location*, unless strong atomic operations such as, e.g., *compare-and-swap* (CAS) are used (and these also incur significant overhead).

Over the last 20 years, shared-memory mutual exclusion research investigates the *remote memory references* (RMR) complexity of local-spin mutex algorithms; much of this work focuses on (deterministic) read/write mutual exclusion (see [6, 13, 14, 15, 26] for some examples).

There are such algorithms that incur only a logarithmic number of RMRs per *passage* (entry and corresponding exit of the critical section) [26], which is asymptotically optimal [4][2], but they require a logarithmic number of barriers as well. On the other hand, it is easily shown that the well-known Bakery mutex algorithm [17] requires only a constant number of barriers per passage. However, it incurs a linear number of RMRs, even in un-contended, solo, passages. So

[1]Owens et al. [22] prove Intel x86 is equivalent to Sparc TSO.

[2]The lower bound of [4] applies also to mutual exclusion algorithms that may use *comparison* primitives, such as CAS, in addition to reads and writes.

is there an inherent tradeoff between the RMR and barrier complexity of read/write mutual exclusion?

This paper shows that it is possible to win on both measures, by presenting a read/write mutual exclusion algorithm that combines optimal RMR complexity with a constant number of memory barriers, in the popular TSO model.

Local-spin mutex research considers both the *Distributed Shared Memory* (DSM) and the *Cache Coherent* (CC) system models. The prevailing DSM memory model is much weaker than TSO [7], however. We therefore consider the CC model in this work.

Mutex algorithms that incur $O(1)$ RMRs and memory barriers per passage can be devised if atomic operations such as swap and fetch-and-add are available. Two key such algorithms are the CLH lock [5, 20] and the MCS lock [21] Both these algorithms use the swap operation.

The rest of this paper is organized as follows. In Section 2, we provide required background, describe the high-level ideas underlying our algorithm and provide intuitions as to why it works under TSO. Section 3 presents a more precise model. Section 4 presents the algorithm in detail (using CAS) and Section 5 proves its correctness; Section 6 explains how to modify the algorithm so it uses only read and write operations. We conclude, in Section 6, with a short discussion of our results and directions for future research. The appendix present preliminary results of an experimental evaluation of our algorithm.

2. BACKGROUND AND OVERVIEW OF THE ALGORITHM

To understand the importance of the execution order of memory access instructions, consider the following simplified code for the *Bakery* mutex algorithm [17].

```
1: Choosing[i] = true
2: Num[i]=1 + max(Num[1],..., Num[n])
3: Choosing[i]=false
4: for (j = 1; j ≤ n; j++)
5:     while (Choosing[j]) nop
6:     while ((Num[j] ≠ 0) and ((Num[j], j) < (Num[i], i))) nop
Critical Section
7: Number[i]=0
```

Roughly, the key for showing the safety of the Bakery algorithm is that a process first announces its participation (by setting its number to a positive value) and then reads to verify that no other process holds a smaller number. However, when the architecture only guarantees total store ordering, the read of Line 6 may be performed early, before the writes of Lines 2 and 3 are visible to the other processes. Thus, it is possible that two processes will read zero from each other's entry of the Num array, and both will enter the critical section. This reordering of instructions can be excluded by inserting a barrier between Line 3 and Line 4. Adding two addition barriers (after Lines 1 and 7) ensures that the Bakery algorithm is correct under the TSO model.

Unfortunately, the Bakery algorithm incurs significant overhead, apparently because of high RMR complexity: a process needs to read the numbers of all potential contenders for the critical section. This produces $O(n)$ RMRs, where n is the number of processes.

Figure 1(a) shows the time taken by a single thread to perform 1M solo passages in the Bakery algorithm, as a function of n. (More information about the tests on which these graphs are based is provided in the appendix.) A second running thread caused the thread performing passages to incur RMRs. We tested performance with and without the algorithm's barriers[3]. Thus, per every x-axis value, there are 2 bars showing the time taken with (light bars) and without (dark bars) barriers. As shown by Figure 1(a), barriers account for a relatively small fraction of the time complexity of the Bakery algorithm. However, the algorithm's time complexity grows quickly with n.

The number of RMRs can be significantly reduced by employing a *tournament tree* mutex algorithm [26]. The algorithm uses a balanced binary tree with n nodes, each associated with a two-process mutual exclusion algorithm. A process starts at a dedicated (virtual) leaf and climbs up the tree, competing in a two-process mutex algorithm at each node, until it wins at the root and enters the critical section. This algorithm incurs $O(\log n)$ RMRs per passage through the critical section, which is optimal [4]. To ensure the correctness of the two-process mutex performed at the nodes on the path to the root, it is necessary to insert a barrier in the code of each internal node (see Figure 2(a)). This means that a process performs $\Theta(\log n)$ barriers on each passage to the critical section, leading to a significant overhead, as can be seen in Figure 1(b).

A balanced binary tree with n leaves is a key data structure also in our mutual exclusion algorithm; however, we use it for *collecting* a list of waiting processes, rather than as an arbitration mechanism. Entry to the critical section is guarded by two mechanisms: The first, which is expected to play the main role in lightly-contended situations, is a lock manipulated with *CAS* operations (we explain how the CAS can be replaced with reads and writes in Section 6). The second mechanism is a *promotion queue* of waiting processes, collected on the binary tree.

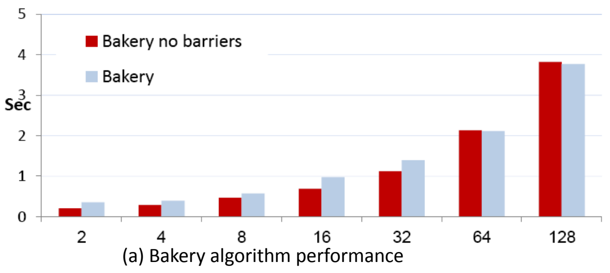
A process exiting the critical section goes down the path from the root to its leaf, collecting waiting processes (whose identifiers it reads from each path node and its child nodes) and adding them to the promotion queue. If the queue is not empty, the exiting process *promotes* one of the processes in the queue into the critical section. The promotion queue is manipulated only by processes exiting the critical section, that is, in exclusion.

A similar promotion mechanism has been used in prior mutex algorithms (e.g., [10]). The key novelty of our algorithm is the simple manner in which processes announce their wish to enter the critical section. A process wishing to enter the critical section writes its id in all nodes on the path from its leaf to the root and tries to grab the lock. If it fails, then it is ensured that some exiting process will find it and add it to the promotion queue; therefore, it can wait for its promotion, spinning on a dedicated variable (Figure 2(b)).

To understand where a barrier should be inserted, it is instrumental to think about the core argument in showing why a process p eventually enters the critical section, namely, the proof that a waiting process will eventually be enqueued to the promotion queue. For the current discussion alone, assume that the memory is sequentially consistent and that each process enters the critical section only once.

Consider the highest node v in which process p's id is not overwritten by another process (its own leaf, in the worst case); let u be its parent. We can argue, by induction, that

[3]Removing the barriers makes the algorithm incorrect in general but not in solo executions.



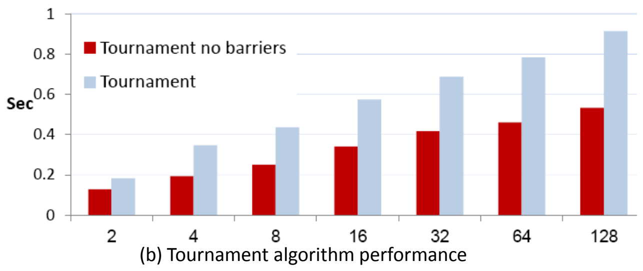


Figure 1: Time to perform 1M passages as a function of the number of threads supported in the Bakery (a) and Tournament tree (b) algorithms.

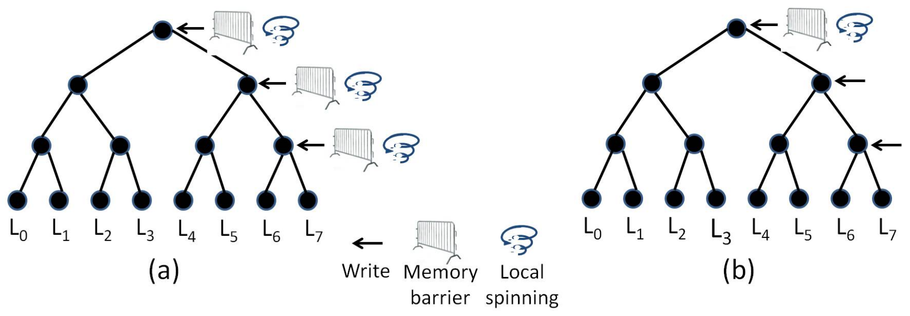


Figure 2: Entry section synchronization pattern in previous optimal-RMR algorithms (a) and in our algorithm (b).

q, the last process to overwrite p's id in u, eventually enters the critical section. The sequence of operations p and q perform (shown below) guarantees that q reads p from v and enqueues it.

```
p                 q
write to v
write to u
                  write to u          (overwrites p)
                  enters CS
                  exits CS
                  reads p from v      (p not overwritten in v)
```

While this description assumes sequential consistency (by assuming all operations are ordered sequentially), it relies in fact on two orders: that the writes of p (to v and u) are executed in program order, and that the writes of p are executed before it fails to win the lock. TSO enforces the first ordering, and *a single* barrier is inserted to ensure the second ordering.

Thus, in our algorithm processes use a synchronization pattern fundamentally different than that used by previous optimal-RMR algorithms as they access the tree while performing their entry section. Previous optimal-RMR mutex algorithms used a synchronization pattern as in the tournament tree (Figure 2(a)). As a process climbs up the path from its leaf to a new node v in the course of the entry section, it performs the following operations. First, it has to write to node v. Then, it has to perform a memory barrier for ensuring the visibility of the write. Finally, it reads v and may need to locally spin until some condition is met.

In our algorithm, a process performing the entry section *only writes* to all non-root nodes along its path, as shown in Figure 2(b); it does not perform memory barrier operations on such nodes, nor does it busy wait on them or even read them. A process may need to perform the write-fence-read sequence of operations only when it reaches the root node. A detailed description of the algorithm appears in Section 4.

3. SHARED-MEMORY SYSTEM WITH TOTAL STORE ORDERING (TSO)

We now present an operational model for the behavior of a shared-memory system with relaxed memory ordering, which is a simplified version of the model used by Park and Dill [23]. The model is tailored to describe TSO, but it can be extended to describe other memory models (see [16, 23, 25]).

There is a set of n processes, $p_1, \ldots, p_n$, each of which has its own abstract *store queue*. A process executes memory operations—read, write and compare-and-swap—in the order specified by its algorithm, called *program order*. Write operations may be delayed and executed after load operations following them in the program order, due to various implementation techniques or compiler optimizations. This is modeled by having write operations go to the store queue rather than directly to the shared memory.

A *configuration* describes the state of a system: It contains the local state of each process, including its location in its algorithm and the contents of its store queue. It also contains the value of each shared variable. In the *initial configuration*, all processes are in their initial state and their store queues are empty; all shared variables hold their initial values.

An *execution fragment* E is a sequence of steps, picked by an adversarial scheduler. The adversary picks a process for the next step, and then decides whether to let the process execute another step according to its algorithm or to *perform* the first write operation in the store queue (if any). If the adversary takes a write operation from the store buffer, then it changes the value of the relevant shared variable to the parameter of the write. The write operation is *performed* at this step.

What happens when a process takes a step depends on the type of step:

1. A *write* operation is placed at the end of the store queue. The write operation is *issued* at this step.

2. A *read* operation returns the value of the variable, and the process changes its local state accordingly. If there are writes to this variable in the store queue, the value is read from the most recent write in the queue; otherwise, the value of the variable is read from the shared memory.[4] The read operation is performed at this step.

3. A *barrier* operation enforces the adversary to perform all the writes in the store queue (if any) in the order they were issued. The writes become visible and the queue is emptied. The barrier operation is performed at this step.

4. A *compare-and-swap* (*CAS*) operation atomically compares the value of the shared variable with its first parameter, and if they are equal, sets the value of the variable to its second parameter. The CAS operation is performed at this step. If there are writes to the variable in the store queue when the CAS is issued, then the adversary performs all these writes (and those preceding them in the buffer, if any) and removes them from the store queue before the CAS step is taken.

Note that in this model, writes become visible to all other processes at the same time (this means that the memory is *coherent*). An *execution* is an execution fragment that starts in the initial configuration.

We assume a *cache-coherent* (CC) system, in which each processor[5] maintains local copies of shared variables it reads 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). Each write to a shared variable incurs an RMR.[6] A read by process p of a shared variable v incurs an RMR if (1) it is the first read of v by p, or (2) it is preceded by a write of another process to v that was performed after p's previous read of v.

4. THE ALGORITHM

We now provide a detailed description of our algorithm, whose pseudo-code is presented by Algorithm 1. Note that Algorithm 1 uses the CAS operation. In Section 6, we explain how it can be transformed to a read-write algorithm that maintains the properties we require.

The data structure underlying the algorithm is a binary tree with leaves $L_0, \ldots, L_{n-1}$ (statement **1**). Leaf L_p, for $p \in \{0, \ldots, n-1\}$, is statically assigned to process p. Tree nodes store integer values and are initialized to -1.

Our algorithm uses a *promotion* mechanism. This mechanism allows a process performing its exit section to facilitate the entry of other processes whose identifiers it reads along the path from the root to its leaf. Identifiers of promoted processes that did not yet enter the critical section are stored in a FIFO queue named *promQ* (statement **3**). Processes apply operations to *promQ* before they release the lock, hence its implementation is not required to support concurrent access.

Each process has an entry in the *inPromQ* array, indicating whether or not it is currently in the promotion queue. Process identifiers are enqueued to the promotion queue only on condition that they are not already in it. The *inPromQ* array is used for checking this condition by performing a single read operation.

The *apply* array is used by processes to apply for promotion (statement **5**). A promoted process busy-waits on its entry of the *signal* array until it is signalled to enter the critical section (statement **7**). The *lock* integer either stores the identifier of the process currently holding the lock, or -1 if the lock is free (statement **11**). It is manipulated using read, write and CAS operations.

The *exits* shared integer counts the number of critical section exits. It is required for preventing a possible scenario in which a process busy-waiting on *lock* incurs a non-constant number of RMRs if the lock owner releases and re-captures *lock* multiple times.

The Entry Section. To enter the critical section, process p first initializes its entry of the *signal* array and sets its entry of the *apply* array to apply for promotion (statements **17**–**18**). Process p then traverses up the path from L_p to the root, writing its identifier to each node along this path (statements **19**–**21**). After having completed this sequence of writes, p performs a single memory barrier to ensure that these writes become visible before it proceeds (statement **22**).

Process p then attempts to capture the lock using a CAS operation (statement **23**). (We note that, in some architectures, the execution of a CAS operation triggers the execution of a memory barrier. In such architectures, the memory barrier of statement **22** is not required.) If the CAS succeeds, then p enters the critical section. If the CAS fails, however, then the process that owns the lock when p's unsuccessful CAS is performed (henceforth denoted by r) may fail to observe p, in which case it will not promote it. This may happen if r has already begun its exit section and already read the nodes on the intersection of p's and r's paths to the root before p performed its memory barrier.

To overcome this potential race condition, p has to busy wait until either another complete execution of the exit section takes place since its CAS failed, the lock becomes free, or the lock is handed to it (statement **25**). (It is also possible that the CAS of statement **23** fails because the lock is handed to p before it performs the CAS.) Then, p attempts to capture the lock again (statement **26**). If it succeeds, it enters the critical section. If it fails, implying that either the lock was re-captured since p's CAS in statement **23** failed or that the lock was handed to p, then p awaits to be signalled (statement **27**). As we prove, this must eventually happen, and when it does, p enters the critical section.

[4]In our algorithm, a read accesses a variable that was previously written to by the same thread only after a barrier, so the first condition is never satisfied.
[5]The model assumes that distinct processes run on distinct processors.
[6]This is not necessarily the case in write-back CC systems. Any upper bound on the RMR complexity in our model holds also for the write-back model.

Algorithm 1: mutual exclusion algorithm for process $p \in \{0, \ldots, n-1\}$

```
1  shared $\mathcal{T}$: Binary tree with leaves $L_0, \ldots, L_{n-1}$
   /* Integer nodes initialized to $-1$ */
2
3  shared $promQ$: Queue of int init $\phi$          /* Queue of
   processes to be promoted */
4
5  shared $apply$: array $[0..n-1]$ of boolean init false
   /* Applying-for-promotion array */
6
7  shared $signal$: array $[0..n-1]$ of boolean   /* Array for
   signalling promoted processes */
8
9  shared $inPromQ$: array $[0..n-1]$ of boolean init false
   /* $promQ$ membership array */
10
11 shared $lock$: int init $-1$      /* Lock, supporting read,
   write and CAS operations */
12
13 shared $exits$: int init 0          /* Number of CS exits,
   incremented on exit section */
14
15 local $r$: boolean, int $next$, $e$;
16 Procedure $Entry()${
17 $signal[p] \leftarrow$ false
18 $apply[p] \leftarrow$ true
19 foreach node $n$ on the path from $L_p$ to the root do
20     $n \leftarrow p$
21 end
22 Barrier
23 if $CAS($lock$, -1, p) \neq -1$ then
24     $e \leftarrow exits$
25     await $((exits - e \geq 2) \vee lock \in \{p, -1\})$
26     if $CAS($lock$, -1, p) \neq -1$ then
27         await $(signal[p])$
28     end
29 end
30 Procedure $Exit()${
31 $apply[p] \leftarrow$ false
32 $exits \leftarrow exits+1$
33 foreach node $n$ on the path from the root to $L_p$'s parent do
34     $q_1 \leftarrow n$, $q_2 \leftarrow n.left$, $q_3 \leftarrow n.$right
35     foreach $q \in \{q_1, q_2, q_3\} \setminus \{-1, p\}$ do
36         if $apply[q] \wedge \neg inPromQ[q]$ then
37             $promQ.enqueue(q)$
38             $inPromQ[q] \leftarrow$ true
39         end
40     end
41 end
42 if $promQ.isEmpty()$ then
43     $lock \leftarrow -1$
44 else
45     $next \leftarrow Q.dequeue()$
46     $inPromQ[next] \leftarrow$ false
47     $lock \leftarrow next$
48     $signal[next] \leftarrow$ true
49 end
50 Barrier
```

The Exit Section. To exit the critical section, process p first resets its entry of the *apply* array as it no longer requires promotion (statement **31**). Process p then descends down the path from the root to its leaf (statement **33**), reading the identifiers written at every internal node along this path and its child nodes (statement **34**). Process p promotes any process that applies for promotion whose identifier it reads, by enqueueing it to the promotion queue if it is not already there (statements **35**–**40**).

After having descended to its leaf, p checks the promotion queue (statement **42**). If it is empty, then p releases the lock (statement **43**), otherwise p dequeues the first process from the promotion queue, resets its *inPromQ* entry, hands over the lock to it and signals it that is may now enter the critical section (statements **45**–**48**). Before exiting, p performs a memory barrier to ensure that the write operations it performed in its critical and exit sections are all visible (statement **50**).

5. CORRECTNESS AND COMPLEXITY PROOFS

In this section, we prove that our algorithm satisfies mutual exclusion and starvation freedom in TSO systems. Then we show that the algorithm provides $(\log n + 3)$-bounded waiting (formally defined later). This is followed by a proof that the algorithm has optimal RMR complexity and uses only a constant number of barriers.

A *passage* is the sequence of steps taken by a process as it performs the entry, critical and exit sections. For the purposes of the proofs in this section, statement **50**, where a process performs a memory barrier to ensure that its writes in the critical and exit sections are performed, is not considered part of the exit section.

A mutual exclusion algorithm provides *k-bounded waiting* if, in every execution, no process enters the critical section more than k times while another process is waiting in its its entry section.

The next theorem summarizes the properties of the algorithm.

Theorem 1. *Algorithm 1 satisfies mutual exclusion, starvation-freedom and* $(\log n + 3)$*-bounded waiting under TSO. Each passage of the algorithm incurs* $O(\log n)$ *RMRs and a constant number of barriers.*

We start with a few definitions. For execution fragments E, F, we write $E \preceq F$ if E is a prefix of F. Let $pc(C, p)$ denote the value of p's program counter in configuration C, that is, the number of the next pseudocode statement that p will execute after configuration C. $C.v$ denotes the value of shared variable v in configuration C. Similarly, $pc(E, p)$ and $E.v$ denote the value of p's program counter and the value of shared variable v in the configuration reached after execution E.

Definition 1. *We say that a* passage starts *when a process issues statement* **17**. *We say that a process* completes its exit section *and* completes a passage *when statement* **43** *or statement* **48** *are performed. We say that a process is* in a passage *in configuration C if there is a passage that p started in the execution leading to C but did not yet complete. Let p be a process in its entry section. We say that p is in the* doorway *if it did not yet perform statement* **23**; *otherwise, we say that p is in the* waiting section. *A process p* owns the lock *in configuration C if $C.lock = p$ holds.*

Invariant (d) of the following lemma immediately implies mutual exclusion.

Lemma 2. *The following invariants hold in every reachable configuration C of the algorithm.*

(a) If p is in the critical section in C, then p owns the lock in C.

*(b) If p is in the exit section in C and $pc(C,p) \neq 48$, then p owns the lock in C. If $pc(C,p) = 48$, then the lock owner is some process q, where q's identifier was dequeued by p in statement **45** of its current passage.*

(c) $\left|\{r|(C.signal[r] = true) \wedge (17 < pc(C,r) < 49)\}\right| \leq 1$. If there is such a process r, then r owns the lock in C and no process $q \neq r$ is in the critical or exit sections in C.

(d) At most one process is in the critical or exit sections.

Proof. All invariants clearly hold in the initial configuration. The only pseudo-code statements that may violate these invariants are the following.

1. Statements **23** and **26**, if the CAS is successful: this makes the executing process enter the critical section and, it is also the lock owner.
2. Statement **27**, if performed by process p when $signal[p] = true$ holds, as this makes p enter the critical section.
3. Statement **43**: the execution of this statement by process p changes the value of *lock* and after its execution p is no longer the lock owner. Also, from Definition 1, after performing statement **43** p completes the exit section.
4. Statement **47**: the execution of this statement changes the identity of the lock owner.
5. Statement **48**: the execution of this statement sets an entry of the *signal* array. Also, from Definition 1, after performing statement **48** p completes the exit section.

We call the above statements *i-statements* and prove the lemma by induction on the number of these statements that are performed in an execution. For the base case, consider an execution E in which no i-statements are performed. Since no successful CAS is performed in E, no process could have entered the CS from statement **23**. This also implies that no process could have been signalled in E, which implies in turn that Invariant (c) holds, since upon starting the entry section a process resets its entry of the *signal* array (statement **17**). As no process was signalled in E, no process could have entered the CS from statement **27**, hence Invariants (a), (b) and (d) also hold.

We note that the first statement of the entry code by a process resets its entry of the *signal* array and therefore cannot violate Invariant (c) (clearly it cannot violate Invariants (a), (b) or (d)).

For the induction step, assume the claim holds for any execution where some $k \geq 0$ i-statements are performed, and let $E = E'sE''$ be an execution where exactly $k+1$ i-statements are performed, the last of which is s. Let C be the configuration after E'. The following cases exist.

1. Step s is the execution of a successful CAS in statement **23** or statement **26** by some process p. Process p is in the critical section in Cs and is the lock owner, thus Invariant (a) is not violated by s. Since the CAS is successful, *C.lock=-1* holds and so no process is the lock owner in C. By the induction hypothesis, applied to Invariant (a), this implies that no process is in the critical section in C. By the induction hypothesis, applied to Invariant (b), this also implies that there is no process in the exit section in C, hence Invariants (b) and (d) hold in Cs. Finally, since no process was signalled in s, Invariant (c) is not violated by s.
2. Step s is the execution of statement **27** by process p and $C.signal[p] = true$ holds. Process p is in the critical section in Cs. By the induction hypothesis, applied to Invariant (c), p is the lock owner in C hence also in Cs, so Invariant (a) holds in Cs. Also by the induction hypothesis, applied to Invariant (c), there is no process in the critical or exit sections in C, so both Invariants (b) and (d) hold in Cs. By the induction hypothesis, applied to Invariant (c), p is the only process whose entry of the *signal* array is set in C, hence it is also the only such process in Cs, so Invariant (c) holds also in Cs.
3. Step s is the execution of statement **43** by some process p. By the induction hypothesis, applied to Invariant (d), p is the only process in the critical and exit sections in C, hence there is no process in the critical or exit sections in Cs, implying that Invariants (a), (b) and (d) hold in Cs. Invariant (c) also holds in Cs, since statement **43** does not set an entry of the *signal* array.
4. Step s is the execution of statement **47** by some process p that hands over the lock to some process q. By the induction hypothesis, applied to Invariant (d), p is the only process in the critical or exit sections in C, hence, in Cs, there is no process in the critical section and p is the only process in the exit section, thus Invariants (a) and (d) hold in Cs. Since $pc(C,p) = 47$, it follows from the induction hypothesis, applied to Invariant (b) that p is the lock owner in C. Consequently, it follows from $pc(Cs,p) = 48$ that Invariant (b) also holds in Cs. Finally, since statement **47** does not set an entry of the *signal* array and $17 < pc(C,p), pc(Cs,p) < 49$, Invariant (c) cannot be violated by the execution of s.
5. Step s is the execution of statement **48** by some process p. By the induction hypothesis, applied to Invariant (d), p is the only process in the critical and exit sections in C, thus there is no process in these sections in Cs and Invariants (a), (b) and (d) hold in Cs. Since p is in the exit section in C, it follows by the induction hypothesis, applied to Invariant (c) that p is the only process not in its remainder section that may have its entry of the *signal* array set in C. Since $pc(C,p) = 48$, it follows from the induction hypothesis, applied to Invariant (b) that there is a process q, whose identifier is written in p's *next* variable in C, that holds the lock in C. Since p exits the critical section once having performed statement **47**, $\{r|(C.signal[r] = true) \wedge (17 < pc(C,r) < \textit{statement } \mathbf{49})\} = \{q\}$ and Invariant (c) holds in Cs.

□

Corollary 3. *The algorithm satisfies mutual exclusion.*

We proceed to prove that the algorithm satisfies starvation freedom. Corollary 3 implies:

COROLLARY 4. *The promotion queue is accessed in a sequential manner.*

LEMMA 5. *Let q be a process that is enqueued by some process p to promQ in some point t of the execution, then (a) q is in its entry section in t, and (b) q eventually enters the critical section after t.*

PROOF. A process q sets its entry of the *apply* array in the beginning of its entry section (statement **18**) and resets it at the beginning of its exit section (statement **31**). Since p is in the exit section in t, it follows from Invariant (d) of Lemma 2 that q cannot be in its critical or exit sections, hence claim (a) holds.

By Corollary 4, operations on *promQ* are applied in a sequential manner. From statements **42**–**48**, process p eventually dequeues the oldest item in *promQ*, say process q', hands over the lock to it, and signals it. By claim (a), q' is in its entry section. Regardless of whether q' performs the CAS operations of statement **23** and statement **26** before or after the lock is handed over to it, these CAS operations will fail and q' is bound to reach statement **25** and will eventually enter the critical section. Proceeding inductively, all processes that precede q in *promQ* (if any) will eventually enter and exit the critical section. It follows that process q eventually enters the critical section. □

LEMMA 6. *Let p be a process that performs statement* **23** *and receives its own identifier p as the result of the CAS operation. Then p eventually enters the critical section.*

PROOF. Since *lock* $= p$ when p performs statement **23**, some process q hands the lock to p during its exit section and signals p. By Invariant (c) of Lemma 2, once p is signalled no other process enter the critical section before it, hence, *lock*$=p$ continues to hold as long as p is in its entry section. Thus, p eventually evaluates the second condition of statement **25** as true and fails the CAS of statement **26**. It follows that p eventually evaluates the condition of statement **27** as true and enters the critical section. □

DEFINITION 2. *Let p be a process and let $l \in \{0, \ldots, \log n\}$ be a tree level (the root is in level 0). We let n_p^l denote the node along the path from L_p to the root that is at level l, where the leaves are at level $\log n$. Let E be an execution. We let $l_p(E)$ denote the highest (smallest) level l such that $n_p^l = p$ holds after E.*

LEMMA 7. *The algorithm satisfies starvation freedom.*

PROOF. Assume, by way of contradiction, that there is an infinite execution EE' such that, at the end of E, a non-empty set $\mathcal{S}$ of *starved processes* are busy-waiting either at statement **25** or at statement **27** but none of them enter the critical section in the (infinite) execution fragment E'. The following claim states that there is a prefix of E' after which the values l_p are "stable" for every starved process p.

CLAIM 1. *There is a prefix E_1 of E' such that $\forall p \in \mathcal{S}, \forall E_1E_2 \preceq E' : l_p(EE_1E_2) = l_p(EE_1)$.*

PROOF. Let p be a starved process. First we note that $l_p(F)$ is well defined for all $E \preceq F \preceq EE'$, since $L_p = p$ always holds for every process in the waiting section. From the definitions of E and of starved processes, p does not issue a write during E'. It follows that l_p is non-decreasing during E', that is, $F_1 \preceq F_2 \preceq E' \Longrightarrow l_p(EF_1) \leq l_p(EF_2)$. Finally, since $l_p \in \{0, \ldots, \log n\}$, l_p may increase at most $\log n$ times during E' and the claim follows. □

We reach a contradiction by proving the following inductive claim.

CLAIM 2. *Let p be a process. If $l_p(EE_1) = k$, for $k \in \{0, \ldots, \log n\}$, then p is not starved in EE'.*

PROOF. We prove the claim by induction on $l_p(EE_1)$.

Base: for the base case, $l_p(EE_1) = 0$ so $n_p^{l_p}$ after EE_1 is the root. Consider process p's last execution of the doorway when it last wrote its identifer to the root node and then performed the barrier in statement **22**. Clearly, p's identifier is not overwritten in the root in the rest of the execution. From the definition of execution E and since p is starved, when p proceeded to perform the CAS of statement **23** during E, the CAS must have failed. Let t be the point in the execution when p performs this CAS. The preceding barrier performed by p in statement **22** guarantees that p's identifier is written at the root in t. If the lock owner at t is p, then by Lemma 6, p eventually enters the critical section. Assume otherwise, then some other process r holds the lock in t. We have the following possibilities:

1. Process r did not yet read the root node in statement **34** of its exit section. In this case, r is bound to read p's identifier in its exit section and to enqueue p to the promotion queue. By Lemma 5, p eventually enters the critical section.

2. Process r reads the root node in statement **34** of its exit section before t but, as it is still the lock owner in t, did not complete its passage. Eventually r exits the critical section. If process p executes statements **25**–**26** before the lock is re-captured, then the CAS of statement **26** succeeds and p enters the critical section. Otherwise, some process r' (possibly $r' = r$) enters the critical section after t. Hence, when r' performs its exit section, it reads p's identifier from the root node and promotes it if it was not promoted earlier, and by Lemma 5(b), p eventually enters the critical section.

Induction: Assume the claim holds for all levels l, $0 \leq l \leq k < \log n$, we prove that it holds also for level $k + 1$. Let process p be a starved process such that $l_p(EE_1) = k + 1$, so n_p^{k+1} is the highest node along the path from L_p to the root such that $n_p^{k+1} = p$ after EE_1.

Consider process p's last execution of the doorway when it last wrote its identifer to node n_p^{k+1} and let t denote the point in the execution when this write is performed. Clearly, p's identifier is not overwritten in n_p^{k+1} in the rest of the execution. After writing to n_p^{k+1}, p proceeds to write its identifier to higher nodes along the path to the root and then performs the barrier in statement **22**. Let t_1 be the point in the execution when p's next write to n_p^k is performed and let t_2 be the point when it performs statement **22**. By total store ordering, $t \leq t_1 \leq t_2$. From the definition of l_p, n_p^k is overwritten after t_1. Let $t' > t_1$ be the point when n_p^k is last written in EE_1 and let q be the process that issues this write operation. Since q is the last process to write to n_p^k in EE_1, $l_q(EE_1) \leq k$. It follows from the induction hypothesis that q is not starved and therefore q enters and exits the critical section after t'. Since $t' > t_1 \geq t$, q reads p's identifier in the course of its exit section in statements **35**–**40**. By Lemma 5(b), p eventually enters the critical section. □

Claim 2, capturing the informal argument presented in Section 2, implies Lemma 7.

Bounded Waiting.

We now prove that our algorithm provides $O(\log n)$-bounded waiting.

Lemma 8. *The algorithm provides* $(\log n + 3)$*-bounded waiting.*

The key to the proof is Lemma 9 below. Essentially, the lemma establishes that if a process p is "overtaken" $\log n+2$ times by some other process q, then some process is bound to read L_p in the course of its exit section and promote p before q will enter the critical section once again.

We use the following notation in the proofs that follow. Let $\mathcal{P}$ be a passage by some process p. Then $c(\mathcal{P})$ is the point in the execution when p completes the doorway during $\mathcal{P}$ and $s(\mathcal{P})$ is the point when p either succeeds in its CAS operation on the lock or is enqueued by some other process to the promotion queue (in statement **37**) during $\mathcal{P}$. For $l \in \{0, \ldots, \log n\}$, let $t^l(\mathcal{P})$ be the point when p's write to n_p^l in the course of $\mathcal{P}$'s doorway is performed.

Lemma 9. *Consider a passage $\mathcal{P}$ by process p in which it does not enter the critical section directly from statement* **23**. *The following holds for all $l \in \{0, \ldots, \log n\}$: if some process q enters the critical section $l+2$ times during $(c(\mathcal{P}), s(\mathcal{P}))$, then some process reads node n_p^l during $(t^l(\mathcal{P}), s(\mathcal{P}))$ as it performs statement* **34** *during its exit section before the $(l+2)$'th entry of q.*

Proof. We prove the claim by induction on l.

Base: for the base case, $l = 0$ and n_p^0 is the root node. If p enters the critical section from statement **26** and no other process enters the critical section during $(c(\mathcal{P}), s(\mathcal{P}))$, then the claim clearly holds. Otherwise, let p' be the first process to enter the critical section at some point $t' \in (t^0(\mathcal{P}), s(\mathcal{P}))$. Clearly from the code, p' reads the root node in statement **34** in the course of its exit section. Assume it reads the root node at point t'' after t'. Then no process q enters the critical section twice during $(c(\mathcal{P}), t'')$ and $t'' \in (t^0(\mathcal{P}), s(\mathcal{P}))$ holds, which proves the claim.

Induction: Assume the claim holds for all $0 \leq l \leq m < \log n$, we prove it holds for $m+1$. Assume there is a process q that enters the critical section $m+3$ times during $(c(\mathcal{P}), s(\mathcal{P}))$. Let t' be the point in the execution when q enters the critical section for the $(m+2)$'th time during $(c(\mathcal{P}), s(\mathcal{P}))$. By the induction hypothesis, there is a point $t'' \in (t^m(\mathcal{P}), s(\mathcal{P}))$, before t', when some process q' reads node n_p^m as it executes statement **34** in the course of its exit section.

Let r be the value read by q' in t''. Since $t'' < t' < s(\mathcal{P})$, $r \neq p$, as otherwise p would have been promoted by q' before t' hence before $s(\mathcal{P})$, contradicting the definition of $s(\mathcal{P})$. Since $t'' \in (t^m(\mathcal{P}), s(\mathcal{P}))$, it follows that r has a passage $\mathcal{R}$ whose doorway completes during $(t^m(\mathcal{P}), s(\mathcal{P}))$. By total store ordering, $t^{m+1}(\mathcal{P}) \leq t^m(\mathcal{P})$ hence r completes $\mathcal{R}$'s doorway during $(t^{m+1}(\mathcal{P}), s(\mathcal{P}))$.

The following possibilities exist:

1. r is in the entry section of $\mathcal{R}$ in t''. In this case, q' promotes r during its exit section before q enters the critical section for the $(m+2)$'th time. Since q may appear in the promotion queue at most once, r will read n_p^{m+1} (say, at point t^*) during its exit section, before q enters the critical section for the $(m+3)$'th time. Since q enters the critical section for the $(m+3)$'th time before $s(\mathcal{P})$, it follows that $t^* \in (t^{m+1}, s(\mathcal{P}))$ holds, hence the claim holds for $m+1$.

2. r already completed passage $\mathcal{R}$ before t''. This implies that r performed the exit section of $\mathcal{R}$ during $(t^{m+1}(\mathcal{P}), t'')$ and read n_p^{m+1} before q enters the critical section for the $(m+2)$'th time. The claim for $m+1$ follows.

□

Proof of Lemma 8: Let p a process that performs a passage $\mathcal{P}$. If p enters the critical section from statement **23**, then it shifts directly from the doorway to the critical section without waiting and the claim is not violated. Assume otherwise, then the conditions of Lemma 9 hold.

Assume that some process q enters the critical section $\log n + 3$ times during $(c(\mathcal{P}), s(\mathcal{P}))$ and let t' be the point in the execution when q enters for the $(\log n + 2)$'th time. By Lemma 9, there is a point execution t'', that follows $t^{\log_n}(\mathcal{P})$ and precedes t', which in turn, precedes $s(\mathcal{P})$), in which some process q' reads L_p as it performs statement **34** during its exit section. Since $L_p = p \wedge apply[p] = true$ holds as long as p is waiting, q' promotes p in this passage. After p is promoted, q may enter the critical section at most once more—for the $(\log n+2)$'th time—before p, since every process appears at most once in the promotion queue. This is a contradiction. □

RMR and Barrier Complexity

Lemma 10. *A process performs a constant number of barriers and incurs $O(\log n)$ RMRs during a passage.*

Proof. The first part of the lemma is clear from the pseudocode. To prove the second part, we only need to consider the number of RMRs incurred while a process p busy waits in statement **25** and statement **27**, since p traverses $O(\log n)$ nodes in its entry and exit sections. Since p reads variable *exits* before executing statement **25**, the first condition of this statement guarantees that p incurs at most 4 RMRs while executing it: at most twice when it reads variable *exits* and at most twice when it reads *lock*. As for statement **27**, since *signal*$[p]$ is only reset by p, p cannot incur more than a single RMR in the course of busy waiting in this statement. □

All the variables used by our algorithms are of bounded size, with the exception of the *exits* variable. This variable is used to prevent a scenario in which a busy-waiting process incurs a non-constant number of RMRs while reading the same identifier from *lock* again and again and the proof of Lemma 10 above relies on its usage. We get the following from Lemma 8:

Observation 1. *Lemma 10 continues to hold if the* exits *variable supports $(n-1)(\log n + 2) + 1$ values and is incremented modulo this value.*

6. A READ/WRITE IMPLEMENTATION

We now show that we can avoid CAS operations and transform the algorithm to an optimal-RMR, constant-barriers algorithm that uses only reads and writes.

Golab et al. [8, Theorem 42] establish that every mutual exclusion algorithm A that uses reads, writes, and *comparison* operations such as compare-and-swap, can be transformed to a mutual exclusion algorithm A' that uses only reads and writes and has the same RMR complexity up to a constant factor. However, we also require that the algorithm obtained by applying the transformation of [8] to Algorithm 1 will need only a constant number of barriers per passage.

The transformation of [8] for cache-coherent algorithms is based on an implementation of an $O(1)$ RMRs, locally-accessible and writable CAS object from reads and writes. This is a complex implementation composed of multiple implementation layers [8, Figure 1]. The bottom-most layer is an $O(1)$ RMRs implementation of *Leader Election* (LE) from reads and writes [9], and the implementations of the other layers appear in [8].

The proof of the following lemma is provided in the full paper. It establishes that *Read* operations on the simulated comparison object that do not incur RMRs perform no writes. *Read* operations that do incur RMRs, as well as *Write* or *ECAS* operations (see [8]) applied to the simulated object, perform a constant number of writes.

LEMMA 11. *Operations on a comparison object simulated by [8] perform at most a constant number of writes. Moreover, a* Read *operation on a simulated comparison object performs write operations only if the simulated* Read *operation incurs an RMR.*

THEOREM 12. *There is a read-write mutual exclusion algorithm for the TSO model that satisfies starvation-freedom and* $(\log n + 3)$*-bounded waiting, for which each passage incurs* $O(\log n)$ *RMRs and a constant number of barriers.*

PROOF. Let A' be the read-write algorithm obtained by applying the transformation of [8] to Algorithm 1. We show that A' retains the properties specified in Theorem 1. From [8, Theorem 42] and Theorem 1, A' satisfies mutual exclusion and starvation-freedom and has $O(\log n)$ RMR complexity. Since Algorithm 1 does not apply CAS operations in its doorway, $(\log n+3)$-bounded waiting is also retained.[7]

We now show that A' has constant barrier complexity. The only shared variable on which CAS operations are performed in Algorithm 1 is the *lock* variable. From Lemma 11, each CAS operation applied in Algorithm 1 to *lock* causes a constant number of write operations in A'. At most two CAS operations are applied to *lock* in every passage of Algorithm 1 (in statement **23** and statement **26**) and only a single Write operation (in statement **43** or statement **47**), hence their transformation causes at most a constant increase in the number of barriers required per passage.

By Lemma 11, every read operation of *lock* in Algorithm 1 that incurs an RMR adds a constant number of write operations per passage of A'. Since the *exits* variable is incremented on every exit of the critical section, a process may incur only a constant number of RMRs while busy-waiting on *lock* in statement **25**. It follows that the transformation only adds to A' a constant number of write operations per passage. Inserting a barrier after each of these write operations preserves the correctness of the transformation. □

[7]As observed by [8], if the doorway code of algorithm A applies comparison primitives, then even if A satisfies k-bounded waiting, for any k, A' will not. This is because the implementations of [8, 9] are blocking but the doorway code is required to be wait-free.

We present the first mutual exclusion algorithm, using only reads and writes, that has optimal complexity under both the RMRs and barriers metrics. Each passage of the algorithm through the critical section incurs $O(\log n)$ RMRs and a constant number of barriers. This is also the first algorithm possessing such properties within the class of algorithms that may use comparison primitives such as CAS in addition to reads and writes.

It is easily seen that the progress guarantees of the new algorithm depend on TSO. Specifically, under *partial store ordering* (PSO), our algorithm does not guarantee deadlock-freedom, although mutual exclusion is not violated. Whether or not an algorithm with the same properties exists for PSO systems is an interesting open question. It is also unclear how *abortable* [13] or *adaptive* [2] mutex algorithms with the same properties may be constructed. We leave these questions for future work.

Acknowledgements.

We thank Dave Dice, Michael Kuperstein, Martin Vechev and Eran Yahav, for helpful discussions and the anonymous reviewers for their useful comments.

7. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *DISC*, pages 29–43, 2000.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[4] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *STOC*, pages 217–226, 2008.

[5] T. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical report, 1993.

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

[7] D. Dice. personal communication, May 2013.

[8] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012.

[9] W. M. Golab, D. Hendler, and P. Woelfel. An $O(1)$ rmrs leader election algorithm. *SIAM J. Comput.*, 39(7):2726–2760, 2010.

[10] D. Hendler and P. Woelfel. Randomized mutual exclusion with sub-logarithmic rmr-complexity. *Distributed Computing*, 24(1):3–19, 2011.

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

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[13] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, pages 295–304, 2003.

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

[15] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *DISC*, pages 1–15, 2001.

[16] M. Kuperstein, M. Vechev, and E. Yahav. Automatic inference of memory fences. In *FMCAD*, pages 111 –119, 2010.

[17] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[18] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[19] D. L.Weaver and T. Germond, editors. *The SPARC Architecture Manual*. Prentice Hall, 1994.

[20] P. Magnusson, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *IPPS*, pages 165–171, 1994.

[21] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[22] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics*, pages 391–407, 2009.

[23] S. Park and D. Dill. An executable specification and verifier for relaxed memory order. *Computers, IEEE Transactions on*, 48(2):227 –235, 1999.

[24] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[25] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

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

APPENDIX

A. EVALUATION

The empirical evaluation, based on which Figures 1 and 3 (below) were derived were conducted on a Sun Fire T200 machine with a single UltraSPARC T1 chip (1.2GHz), comprising 8 cores, each with 4 hyper threads, running the SunOS 5.11 operating system.

We tested 3 algorithms: the Bakery algorithm [17], a tournament-tree mutex algorithm implementing Peterson's 2-process mutex algorithm [24] in each internal node, and the new algorithm (using hardware CAS). All algorithms were implemented in C and compiled using Sun's CC compiler version 5.9 with the -O5 flag.[8]

We did not conduct a comprehensive evaluation of the new algorithm under contended workloads. Rather, in order to empirically validate the significance of the barriers metric, we compared the time complexity of solo passages with and without memory barriers. We note that, although removing the barriers makes these algorithm incorrect in general, it causes no correctness issues in solo executions.

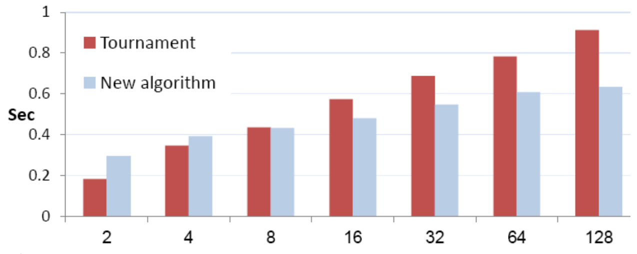


Figure 3: Time to perform 1M passages as a function of the number of threads supported.

For each algorithm, we measured the time it took a single thread (henceforth thread 1) to complete 1M solo passages (with and without barriers) as a function of the number of threads supported by the algorithm. Each data point in Figures 1 and 3 is the average of 100 different runs.

In order to verify that the thread performing passages incurs RMRs, a second thread (henceforth thread 2), bound to a different core, was also running throughout the test and writing to variables read by thread 1 in order to invalidate the corresponding cache lines, thus causing thread 1 to incur L1 cache-misses when reading these variables.

The writes of thread 2 are such that thread 1 never has to wait for thread 2. These are the variables to which thread 2 repeatedly writes in the course of the test for each of the 3 algorithms.

Bakery: thread 2 writes 0 to its *choosing* flag, thus causing thread 1 to incur an RMR whenever reading this flag in its entry section.

Tournament: thread 2 writes 0 to the b flag variable in Peterson's algorithm corresponding to thread 1's rival in each node along its path to the root, thus causing thread 1 to incur an RMR on each such node in its entry section.

New algorithm: thread 2 writes -1 to all the internal nodes along thread 1'th path to the root, thus causing thread 1 to incur an RMR on each such node in its entry section.

Figure 3 compares the time it takes a thread to perform 1M solo passages when performing the tournament and the new algorithm (using hardware CAS). For trees of 2 and 3 levels ($n = 2$ or $n = 4$), solo passages of the tournament tree are faster, since the new algorithm has more overhead, especially in its exit section. For $n \geq 16$, however, solo passages of the new algorithm are faster and the gap increases with n, since the number of barriers in the tournament algorithm grows with n.

[8]The code can be downloaded from http://code.google.com/p/optimal-rmrs-constant-fences-mutex/.