A Complexity Separation Between the Cache-Coherent and Distributed Shared Memory Models

Wojciech Golab*
Hewlett-Packard Labs
Palo Alto, California, USA
wojciech.golab@hp.com

September 26, 2011

We consider asynchronous multiprocessor systems where processes communicate by accessing shared memory. Exchange of information among processes in such a multiprocessor necessitates costly memory accesses called *remote memory references* (RMRs), which generate communication on the interconnect joining processors and main memory. In this paper we compare two popular shared memory architecture models, namely the *cache-coherent* (CC) and *distributed shared memory* (DSM) models, in terms of their power for solving synchronization problems efficiently with respect to RMRs. The particular problem we consider entails one process sending a "signal" to a subset of other processes. We show that a variant of this problem can be solved very efficiently with respect to RMRs in the CC model, but not so in the DSM model, even when we consider amortized RMR complexity.

To our knowledge, this is the first separation in terms of amortized RMR complexity between the CC and DSM models. It is also the first separation in terms of RMR complexity (for asynchronous systems) that does not rely in any way on wait-freedom—the requirement that a process makes progress in a bounded number of its own steps.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles, *Shared memory*; F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems.

General Terms: Algorithms, theory.

Keywords: Shared memory models, remote memory references, complexity.

*This research was conducted mostly during a postdoctoral fellowship at the University of Calgary, under the supervision of Prof. Philipp Woelfel. Author partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

1 Introduction

Shared memory multiprocessors in the form of multi-core chips can be found in most servers and desktop computers today, as well as many embedded systems. Due to the large gap between memory and processor speed, such systems rely heavily on architectural features that mitigate the relatively high cost of accessing memory. Two models of such architectures, illustrated in Figure 1, are the cache-coherent (CC) model and the distributed shared memory (DSM) model [3]. Cache-coherent systems are most common in practice, and often use a shared bus as the interconnect between processors and memory. Memory references that can be resolved entirely using a processor's cache (e.g., in-cache reads) are called *local* and are much faster than ones that traverse the interconnect (e.g., cache misses), called *remote memory references* (RMRs). The fact that any memory location can be cached by any process simplifies greatly the design of efficient algorithms in the CC model. In contrast, in the DSM model memory is partitioned into modules that are tied to specific processors. Different memory modules can be accessed in parallel by those processors using separate memory controllers, which provides superior memory bandwidth. As in the CC model, we can classify memory references in the DSM model as fast local references versus more costly RMRs. The classification in the DSM model is based only on the memory location (as opposed to the state of caches): A reference to a memory location in a processor's own memory module is local, and a reference to another processor's memory module is an RMR.

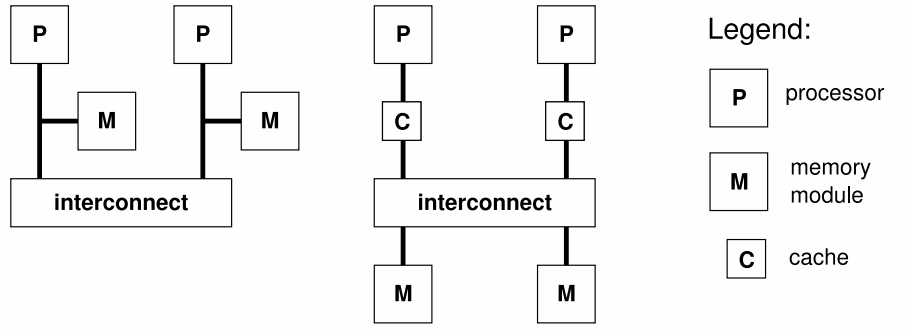

Figure 1: Models of shared memory architecture—DSM (left) and CC (right).

In this paper we consider efficient algorithms for solving synchronization problems in asynchronous multiprocessors that conform to either the CC or DSM model. In particular, we consider algorithms that use *blocking synchronization*, whereby processes may busy-wait by repeatedly reading the values of shared variables. In this context, RMR complexity has been shown to be a meaningful indicator of real world performance (e.g., [4]). The fundamental technique in the design of RMR-efficient algorithms is to co-locate variables with processes that access them most heavily. Unfortunately such techniques are specific to a shared memory model. Consequently, an algorithm that is very

RMR-efficient in one model is not necessarily efficient with respect to RMRs in another model (e.g., see Section 5 of [3]).

An interesting open problem is to compare the relative power of the CC and DSM models for solving synchronization problems efficiently with respect to RMRs. To settle this question, we must fix a synchronization problem and a set of synchronization primitives (e.g., atomic reads and writes) that are available for accessing memory. Consider first the mutual exclusion (ME) problem [9], where processes contend for a shared resource and must coordinate with each other to ensure that at most one process has access to the resource at any given time. Tight bounds for RMR complexity of N-process mutual exclusion are known for popular combinations of primitives, and do not show evidence that the CC model is more powerful than the DSM model, or vice-versa [3, 10, 5]. That is, although the RMR complexity may depend on the combination of primitives, for each combination studied, the tight bound is the same for the CC model as for the DSM model.

Surprisingly, Hadzilacos and Danek [8] discovered a separation between the CC model and DSM model by looking at the RMR complexity of solving *group mutual exclusion* (GME). This problem generalizes ordinary mutual exclusion by annotating each request for the shared resource with a *session* ID, and allowing multiple processes to access the resource concurrently provided that they request the same session. For a certain combination of primitives, it turns out that the RMR complexity of two-session N-process GME is less in the CC model than in the DSM model, by a factor of $\Theta(N/\log N)$.

Although we know that in one case the CC model is more powerful than the DSM model with respect to RMR complexity of a synchronization problem, the relative power of these two models is not well understood in the broader sense. For example, we do not know whether the CC model is at least as powerful as the DSM model for all problems, or whether perhaps the two models are incomparable because for some problem DSM is more powerful than CC. We also do not know how the two models compare under other notions of power, particularly the power to solve problems efficiently with respect to amortized (as opposed to worst-case) RMR complexity.

Answers to the above questions have interesting implications regarding the possibility of an *RMR-preserving simulation* of one model using another other. Such a simulation, if it exists, could be used to transform an algorithm that solves a given problem in one model to an algorithm that solves the same problem in another model, with at most a constant-factor increase in RMR complexity. A simulation of the CC model using the DSM model would be particularly interesting for both software and hardware designers because the CC model is arguably easier to program in, whereas the DSM model is easier to implement in hardware. Known results show only that such a simulation cannot exist if we define RMR complexity in the worst-case sense, leaving open the possibility that a simulation could at least preserve amortized RMR complexity.

Another open question is whether it is possible to show a separation in the RMR complexity of a problem between the CC and DSM models without leaning on *wait-freedom*—the requirement that a process must make progress

in a bounded number of its own steps [16]. The complexity separation shown in [8] depends crucially on a restricted form of wait-freedom in the specification of the GME problem, which makes it more difficult to synchronize when one process releases the shared resource and allows a subset of other processes to emerge from busy-wait loops and make progress. This aspect of the problem specification tends to favor the CC model, where the problem can be solved by having the former process signal the others through a single spin variable. In contrast, in the DSM model spin variables cannot be shared by processes (or else RMR complexity becomes unbounded), and so more elaborate synchronization mechanisms must be used. Wait-freedom restricts the possible mechanisms that can be used, and in that sense penalizes the DSM model. Consequently, we wonder whether removing wait-freedom from the problem specification might create a more level playing field within which to judge the power of the CC and DSM models.

Summary of contributions The key contribution of this paper is the proof of a separation between the DSM and CC models in terms of the amortized RMR complexity of solving a simple synchronization problem. The "direction" of the separation is consistent with the one discovered by Hadzilacos and Danek [8]; the problem under consideration is solved more efficiently in the CC model than in the DSM model. However, our result is stronger in two ways. First, it applies to amortized RMR complexity and not only worst-case RMR complexity. Second, it is insensitive to progress properties in the sense that it holds for both the wait-free version of the synchronization problem and the version that allows busy-waiting.

Our result implies that the CC model cannot be simulated using the DSM model without introducing more than a constant-factor overhead in terms of the total number of RMRs performed by all processes executing an algorithm.

Road map We give the model and definitions in Section 2. We then survey related work in Section 3. In Section 4, we specify a simple synchronization problem, called the *signaling problem*. In Section 5 we give a simple algorithm that solves this problem in the CC model using very few RMRs. Section 6 presents a lower bound for the DSM model, which establishes a complexity separation from the CC model. We then discuss the complexity of variations on the signaling problem in Section 7. Finally, we consider the practical implications of our main result in Section 8, and conclude the paper in Section 9.

2 Model

There are N asynchronous processors that communicate by accessing shared memory using the following atomic primitives: reads, writes, Compare-And-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC). (For definitions of CAS and LL/SC see [17].)

Processes and steps There are up to N *processes* running on the processors, at most one process per processor. The set of processes is denoted $\mathcal{P} = \{p_1, p_2, ..., p_N\}$, and we say that p_i has *ID* i. Each process is a sequential thread of control that repeatedly applies *steps*, where each step entails a memory access and some local computation. A step may cause a process to *terminate*, meaning that it stops performing steps. Processes can be modeled formally as input/output automata [27], but here we adopt a more informal approach by describing their possible behaviors through a shared memory algorithm. The algorithm is expressed through pseudo-code for each process, which is a collection of procedures that a process may call, one at a time. A procedure may accept some input arguments and may return some response to the caller. We specify a process by defining the possible sequences of procedure calls a process may make before terminating. We say that a process *crashes* if it terminates while performing a procedure call.

Histories A *history* is a finite or infinite sequence of steps that describes an execution of the multiprocessor from well-defined initial conditions. Process steps can be scheduled arbitrarily, and there is no bound on the number of steps that can be interleaved between two steps of the same process. A process *participates* in a history if it takes at least one step in that history. A history is *fair* if every process that participates either takes infinitely many steps, or terminates eventually.

Progress properties We will analyze the algorithms presented in this paper with respect to two progress properties: *wait-free* and *terminating*. An algorithm is wait-free if there is an upper bound B such that for any history H of the algorithm, each (partially or fully completed) call to a procedure in H incurs at most B steps. An algorithm is terminating if, for any fair history H of the algorithm where no process crashes, each (partially or fully completed) call to a procedure in H incurs a finite number of steps. (That is, in H each process that participates either terminates after completing a finite number of procedure calls, or else it makes infinitely many procedure calls.)

Remote Memory References RMRs were introduced in Section 1. In the DSM model, a memory access is an RMR if and only if the address accessed by the processor maps to a memory module tied to another processor. In the CC model, the definition of an RMR is more complex; it depends on the state of each processor's cache, as well as the type of coherence protocol used to maintain consistency among caches. For our purposes, we need only a loose definition of RMRs in the CC model that makes it possible to establish upper bounds on RMR complexity. To that end, we assume that if a process reads some memory location several times, then this entire sequence of reads incurs only one RMR in total provided that between the first and last of these reads there is no nontrivial operation performed by another process on that memory

location. (A nontrivial operation overwrites a memory location, possibly with the same value as before.)

3 Related Work

A number of interesting complexity results have appeared in literature on algorithms for asynchronous shared memory multiprocessors. Many of these pertain to mutual exclusion (ME) [9, 25], the problem of ensuring exclusive access to a shared resource among competing processes. RMRs were originally motivated in this context as an alternative to traditional step complexity. (In an asynchronous model, ME cannot be solved with bounded step complexity per process.) The key result in RMR complexity of ME is a separation between the complexity (per passage through the critical section) of two classes of algorithms characterized by the set of primitives used. For the class based on reads and writes, the tight bound is $\Theta(\log N)$ RMRs per process in the worst case [30, 22, 10, 5]. In contrast, for the class that uses reads, writes, and Fetch-And-Increment or Fetch-And-Store, the tight bound is $O(1)$ RMRs [4, 14]. Analogous bounds hold for first-come-first-served (FCFS) ME [24, 3, 7].

RMR complexity bounds for the class of algorithms that use reads and writes only can be generalized to the class that in addition uses comparison primitives (e.g., Compare-And-Swap) [3]. For ME, the reason is that comparison primitives can be simulated efficiently using reads and writes. For example, any comparison primitive can be implemented using reads and writes with only $O(1)$ RMRs per operation in the CC and DSM models [13, 12]. Note that in such implementations *every* operation incurs RMRs, in contrast to a comparison primitive implemented in hardware, which can sometimes be applied locally. *Locally-accessible* implementations address this issue and can be used to transform any algorithm that uses reads, writes, and comparison primitives into one that uses reads and write only, and has the same RMR complexity asymptotically [12, 11]. Note that because this transformation necessarily introduces busy-waiting ([16]), it can break certain correctness properties of the algorithm, such as bounded exit in ME, and bounded doorway in FCFS ME [3]. (This is precisely why the transformation was not used in [7].)

For ME and FCFS ME, the same RMR complexity bounds hold in the CC model as in the DSM model, with the exception of so-called Local-Failed Comparison with write-Update (LFCU) systems [1]. An LFCU system is a type of cache-coherent machine that is almost never implemented in practice. In such systems, ME can be solved using reads, writes and Test-And-Set in $O(1)$ RMRs, which beats the $\Theta(\log N)$ tight bound for the DSM model. The complexity results presented in this paper for the CC model hold just as well for LFCU systems as for the more standard write-through and write-back systems [29].

Mutual exclusion has been studied not only asynchronous systems, but also in semi-synchronous systems, where consecutive steps by the same process occur at most Δ time units apart for some Δ [3]. In one class of such systems, every

process knows Δ, and processes have the ability to delay their own execution by at least Δ time units in order to force others to make progress. Given reads, writes and comparison primitives, ME can be solved in such systems using $O(1)$ RMRs in the DSM model, but in the CC model $\Omega(\log \log N)$ RMRs are needed in the worst case [23]. To our knowledge, this is the first result that separates the CC and DSM models in terms of RMR complexity for solving a fundamental synchronization problem. (In this context we ignore complexity bounds for LFCU systems because they are not representative of the more common variants of the CC model.)

An interesting complexity separation has also been shown for the group mutual exclusion (GME) problem [19] in asynchronous systems. GME is a generalization of ME where requests for the shared resource are annotated with session IDs, and two processes can access the shared resource concurrently provided that they request the same session. Several specifications for this problem have been proposed, differing in fairness and progress properties [19, 20, 15, 18, 6]. Upper bounds for RMR complexity of GME in asynchronous systems range from $O(\log N)$ to $O(N)$ depending on the particular specification, and are subject to any lower bound known for ME. Some algorithms use only atomic reads and writes, while others rely also on CAS and/or Fetch-And-Add primitives. To our knowledge, the only known lower bound on RMRs for GME, except those for ME, is the $\Omega(N)$ bound by Hadzilacos and Danek for the DSM model [8], which applies to the version of GME defined by Hadzilacos [15] and holds even when there are only two sessions. This result separates the DSM model from the CC model, in which the two-session case can be solved using only $O(\log N)$ RMRs [8]. The "direction" of the separation is opposite to the one for ME in semi-synchronous systems.

Another line of research related to this paper pertains to transforming an algorithm that solves some synchronization problem in one shared memory model into an algorithm that solves the same problem in another model, with the same RMR complexity up to a constant factor. A few transformations have been proposed for mapping mutual exclusion algorithms from the CC model to the DSM model [26, 2]. (In [2], see footnote 7.) These transformations work only for a restricted class of ME algorithms, and to our knowledge no general transformation exists for ME or any other widely studied synchronization problem.

4 Problem specification

In this section we specify the *signaling problem*, for which we establish RMR complexity bounds in the remainder of the paper. The problem belongs to the family of problems where two types of processes, called *signalers* and *waiters*, must exchange information regarding some event (e.g., a shared resource has been released). That is, the signalers must ensure that the waiters are aware that the event has occurred. There are several important dimensions within which the safety properties for the signaling problem problem can be pinned down: Is there one signaler/waiter or are there many? Are the IDs of the

signalers/waiters fixed in advance or decided arbitrarily at runtime? How do waiters learn about the signal?

We also consider two ways to specify the semantics of the signaling problem. With *polling semantics*, a solution to the problem consists of two procedures, called `Signal()` and `Poll()`. A signaler issues the signal by calling `Signal()`, which has no return value. A waiter learns about the signal by calling `Poll()`, which returns a Boolean indicating whether the signal has been issued. A process may call `Signal()` at most once and may call `Poll()` arbitrarily many times until such a call returns `true`. (Alternately, we can require that waiters and signalers be distinct. This has no effect on the complexity bounds presented in this paper.) The safety properties for the procedures `Signal()` and `Poll()` are stated formally below in Definition 4.1.

Specification 4.1. *For any history where each process makes zero or more calls to* `Signal()` *and zero or more calls to* `Poll()`*, in any order, the following hold:*

- *If some call to* `Poll()` *returns* `true`*, then some call to* `Signal()` *has already begun.*
- *If some call to* `Poll()` *returns* `false`*, then no call to* `Signal()` *completed before this call to* `Poll()` *began.*

With *blocking semantics*, a solution to the problem consists of procedures `Signal()` and `Wait()`. Procedure `Signal()` is specified as for polling semantics. A waiter learns about the signal by calling `Wait()`, which returns (with a trivial response) only after some call to `Signal()` has begun. If the signal is never issued, then `Wait()` never returns. As before, a process may call the two procedures arbitrarily many times, in any order.

To derive a complexity separation between the CC and DSM models, we consider one of the most difficult variations of the signaling problem: there is one signaler and there are many waiters, whose IDs are not fixed in advance; polling semantics are used; and waiters can terminate after a finite number of calls to `Poll()` even if no such call returned `true`—a key point exploited in our lower bound proof.

Orthogonal to the above safety properties are the progress properties a solution may satisfy. Terminating solutions are certainly possible, and with polling semantics wait-free solutions can also be considered. Note that in terminating solutions, one process may busy-wait for another during a call to `Poll()` regardless of the values returned by such calls. However, if the signal has not yet been issued, each call to `Poll()` must eventually terminate provided that the history is fair.

5 Upper bound for CC model

We are interested in solutions to the signaling problem that are efficient with respect to RMRs. More precisely, we would like to minimize the total number of RMRs a process incurs across all the procedure calls it makes in a given history.

In the CC model, a very simple and RMR-efficient solution is obtained using a single Boolean shared variable, call it B, set to false initially. With polling semantics, procedure `Signal()` assigns $B :=$ true, and `Poll()` reads and returns the value of B. With blocking semantics, `Signal()` also assigns $B :=$ true, and `Wait()` busy-waits until $B =$ true holds before returning.

The solution described above is wait-free, has $O(1)$ RMR complexity per process in the CC model, and uses only atomic reads and writes. As we show next in Section 6, such a solution cannot be obtained in the DSM model, even if we care only about terminating solutions, settle for $O(1)$ amortized RMR complexity, and allow additional synchronization primitives. We then discuss additional upper bounds for variations of signaling problem in Section 7.

6 Lower bound for DSM model

Our main result, to which we devote the remainder of this section, is captured in Theorem 6.2 below. (In this section, "signaling problem" refers to the particular variation described at the end of Section 4.)

Definition 6.1. *For any algorithm $\mathcal{A}$ that solves the signaling problem with polling semantics, let $\mathcal{H}_{\mathcal{A}}$ denote the set of histories (and all their finite prefixes) where each process makes zero or more calls to* `Poll()` *and zero or more calls to* `Signal()`*, in arbitrary order, and then terminates.*

Theorem 6.2. *For any deterministic terminating algorithm $\mathcal{A}$ that solves the signaling problem (with polling semantics) using atomic reads and writes, and for any constant $c \in \mathbb{Z}^+$, there exists a constant $k \in \mathbb{Z}^+$ and a history $H \in \mathcal{H}_{\mathcal{A}}$ where k processes participate and incur more than ck RMRs in total in the DSM model.*

At the end of this section (see Corollary 6.14), we generalize the above result to algorithms that use compare-and-swap (CAS) or load-linked/store-conditional (LL/SC) in addition to reads and writes.

6.1 Overview of proof

To establish Theorem 6.2, we apply a two-part proof whose first part is very similar to Kim and Anderson's construction for proving a lower bound on the RMR complexity of adaptive mutual exclusion [21]. The key idea we borrow from them is to construct inductively a history where communication among processes is minimized using the strategies of erasing and rolling forward. (Eventually there are no more processes to erase or roll forward, and the construction halts.) The main difference between their construction and ours is the end goal. Whereas Kim and Anderson apply the maximum possible number of rounds in order to trigger many RMRs, our goal is to apply just enough rounds so that waiters "stabilize," meaning that they stop performing RMRs and start busy-waiting on local memory. In fact, for our purposes it helps to use as few rounds as

possible, as that maximizes the number of waiters remaining. Part two of our proof then shows how to extend this construction so that a signaler executes many (i.e., more than ck) RMRs communicating with the waiters.

In the inductive construction, we begin with all N processes participating as waiters, making repeated calls to `Poll()`. The strategies for erasing and rolling forward are analogous to Kim and Anderson's. In our context, rolling forward means that a waiter is allowed to complete any ongoing call to `Poll()` it may have, and terminate. After applying the inductive construction for only a constant number of rounds, there are a few processes that have been rolled forward, and many more "invisible" processes that have not communicated with each other. Next, we proceed to part two. Here we show that many of the invisible processes created in part one have stabilized, and that if a judiciously chosen process calls `Signal()`, then it can be forced into an expensive (in terms of RMRs) "wild goose chase." Since the signaler does not know who the waiters are, it must discover them by performing RMRs, but in that case we intervene and erase the waiter (if it is invisible) just before the RMR.

In the remainder of this section, we describe in more detail our two-part proof. We proceed by contradiction, supposing that Theorem 6.2 is false. That is, we suppose that some deterministic terminating algorithm $\mathcal{A}$ (consisting of subroutines `Poll()` and `Signal()` for each process) exists that solves the signaling problem using atomic reads and writes with $O(1)$ amortized RMR complexity. Then there is a constant $c \in \mathbb{Z}^+$ such that for any $H \in \mathcal{H}_\mathcal{A}$ (see Definition 6.1), if k processes participate in H then the total number of RMRs incurred by these processes in the DSM model is at most ck.

6.2 Proof—Part 1

We first give some definitions based on those of [21]. These definitions are specialized for the DSM model, which makes them different from Kim and Anderson's more elaborate definitions that deal with the CC and DSM models simultaneously.

Definition 6.3. *For any $H \in \mathcal{H}_\mathcal{A}$, let $Par(H)$ denote the set of processes that participate in H. The set of* finished processes *in H, denoted $Fin(H)$, is the subset of $Par(H)$ consisting of processes that have terminated by the end of H. The set of* active processes *in H, denoted $Act(H)$, is defined as $Par(H) \setminus Fin(H)$ (i.e., participating processes that have not yet terminated).*

Definition 6.4. *For any $H \in \mathcal{H}_\mathcal{A}$ and any $p, q \in Par(H)$, we say that p* sees *q in H if and only if p reads a variable that was last written by q.*

Definition 6.5. *For any $H \in \mathcal{H}_\mathcal{A}$ and any $p, q \in Par(H)$, we say that p* touches *q in H if and only if p accesses a variable local to q.*

Definition 6.6. *For any $H \in \mathcal{H}_\mathcal{A}$, we say that H is* regular *if and only if all of the following conditions hold:*

1. *For any distinct $p, q \in Par(H)$, if p sees q in H then $q \in Fin(H)$.*

2. *For any distinct $p, q \in Par(H)$, if p touches q in H then $q \in Fin(H)$.*
3. *For any variable v written in H, if v is written by more than one process and the last write is by $p \in Par(H)$, then $p \in Fin(H)$.*

Lemma 6.7. *For any $H \in \mathcal{H}_{\mathcal{A}}$ and any $p \in Act(H)$, if no $q \in Par(H)$ sees p in H, then the history H' obtained by erasing all steps of p from H is also an element of $\mathcal{H}_{\mathcal{A}}$.*

We will use Lemma 6.7 implicitly in the proof sketches that follow whenever we need to erase an active process from a history.

At this point we depart from Kim and Anderson's definitions [21] and introduce a few of our own.

Definition 6.8. *For any regular $H \in \mathcal{H}_{\mathcal{A}}$ and any $p \in Act(H)$, p is* stable *if and only if for any extension H' of H where p runs solo and continues calling* `Poll()` *repeatedly, p incurs zero RMRs in H' after the prefix H. Otherwise p is* unstable.

Our end goal in this part of the proof is to show that $\mathcal{H}_{\mathcal{A}}$ contains a regular history such that $Act(H)$ contains many more stable processes than $Fin(H)$. To that end, we will construct inductively a sequence of regular computations H_1, H_2, H_3, ..., H_c and prove the following invariant in the case when N is large enough (with respect to c):

Definition 6.9. *For any i, $0 \leq i \leq c$, let $S(i)$ denote the statement that $\mathcal{H}_{\mathcal{A}}$ contains a regular history H_i satisfying all of the following properties:*

1. $|Fin(H_i)| \leq i$
2. $|Act(H_i)| \geq N^{1/3^i}$
3. *Each $p \in Act(H_i)$ incurs at most i RMRs.*
4. *Each unstable $p \in Act(H_i)$ incurs exactly i RMRs.*
5. *Each $p \in Fin(H_i)$ incurs at most ci RMRs.*

Lemma 6.10. *If N is large enough then for any i, $0 \leq i \leq c$, $S(i)$ holds.*

Proof sketch: The analysis is similar to that of [21] at a high level, with some technical differences. We proceed by induction. For $S(0)$, note that in H_0 there N active processes and no finished processes, and no process has performed an RMR. Now for any $0 \leq i < c$ suppose that $S(i)$ holds. We must prove $S(i+1)$. To that end, we will construct H_{i+1} from H_i. If there are no unstable processes in $Act(H_i)$, we simply let each active process take one more step, which is necessarily local step.

Otherwise, we let each unstable process in $Act(H_i)$ take steps until it is about to perform an RMR, and determine for each such process its next RMR. We will refer to these as the "next RMRs" of the unstable processes. Allowing each such process to apply its next RMR may yield a history H_i' that is not regular because it violates one of the properties in Definition 6.6. As in Kim and Anderson's proof, properties 1 and 2 of Definition 6.6 are dealt with easily

by erasing at most a constant fraction of active processes. To that end, we construct a "conflict graph" where vertices represent active processes and an edge $\{p, q\}$ exists if and only if p sees or touches $q \neq p$ in its next RMR (or vice versa). Since a process can see or touch at most one other process by performing an RMR, the conflict graph has average degree $d \leq 4$. (For each process we add at most two edges, and each edge contributes to the degree of two vertices.) By Turán's theorem, the conflict graph contains an independent set containing at least $\frac{1}{d+1} = \frac{1}{5}$ of the active processes. Keeping these and erasing the remaining active processes resolves all the conflicts. Once this is done, we apply any next RMRs that perform a read, and consider the remaining RMRs for property 3 of Definition 6.6. We consider two cases, as in Kim and Anderson's proof: one dealt with by rolling forward, the other by erasing. Let X denote the number of unstable processes remaining after properties 1–2 of Definition 6.6 are dealt with.

Roll-forward case If at least $\lfloor\sqrt{X}\rfloor$ unstable processes are about to access the same shared variable v in their next RMRs, we erase all other unstable processes, and allow the ones remaining to apply their next RMRs on v in some arbitrary order. The last process to write v, call it r, is then rolled forward. As we roll forward r, it may see or touch other processes. If r sees or touches an active process p, then this is an RMR for r and we erase p. (It follows from H_i being regular that r cannot see p via a local access prior to r's next RMR.) If r executes more than $c(i+1)$ RMRs in total as a result of being rolled forward, then we obtain a contradiction easily: Erasing all other active processes we obtain a history where there are at most $i+1$ finished processes, namely i from H_i (by property 1 of Definition 6.9) plus r, and no other processes. The total number of RMRs in this history is more than $c(i+1)$, which contradicts our definition of $\mathcal{A}$. Thus, by rolling forward r we erase at most $c(i+1) \leq c^2 + c$ active processes. The number of active processes remaining is at least $\lfloor\sqrt{X}\rfloor - (c^2 + c + 1)$.

Erasing case If there is no single variable that is about to be accessed by at least $\lfloor\sqrt{X}\rfloor$ unstable processes in their next RMRs, then these RMRs collectively access at least $\lfloor\sqrt{X}\rfloor$ distinct variables. For each such variable, we choose arbitrarily an unstable process that will access it, and we erase all the other unstable processes. This leaves at least $\lfloor\sqrt{X}\rfloor$ active processes. It is possible that some of the next RMRs of these remaining processes are about to write registers that have already been written by active processes. We can eliminate this problem by erasing some of the active processes. To that end, we construct a conflict graph where each vertex is an active process and an edge $\{p, q\}$ exists if and only if p writes in its next RMR a variable previously written by $q \neq p$. Since each active process has at most one next RMR, the conflict graph has average degree $d \leq 2$. (For each process we add at most one edge, and each edge contributes to the degree of two vertices.) By Turán's theorem, the conflict graph contains an independent set containing at least $\frac{1}{d+1} = \frac{1}{3}$ of the active processes. Keeping these and erasing the remaining active processes resolves all

the conflicts, leaving $\left\lceil \lfloor \sqrt{X} \rfloor \times \frac{1}{3} \right\rceil$ active processes.

Let H_{i+1} denote the history obtained from H_i by our construction. It follows from our construction that H_{i+1} is regular, and so it remains to show that it satisfies properties 1–5 of $S(i+1)$ given that $S(i)$ holds. Property 1 holds because $|Fin(H_{i+1})| \leq |Fin(H_i)| + 1 \leq i$. Property 2 holds because $|Act(H_{i+1})| \geq |Act(H_i)|^{1/3} \geq N^{1/3^{(i+1)}}$ for N large enough. Properties 3 and 4 follow from $S(i)$ and the application of the next RMRs round $i+1$ of the construction. Property 5 follows from $S(i)$ and the fact that any process being rolled forward in round $i+1$ incurs at most $c(i+1)$ RMRs in H_{i+1} under our original supposition that Theorem 6.2 is false. Thus, $S(i+1)$ holds. □

6.3 Proof—Part 2

Lemma 6.11. *In the history H_c referred to by Lemma 6.10 there are at least $\lfloor N^{1/3^c}/2 \rfloor$ stable processes.*

Proof sketch: Recall that, by Lemma 6.10, the following hold: $|Fin(H_c)| \leq c$, $|Act(H_c)| \geq N^{1/3^c}$, and that each unstable process in $Act(H_c)$ has performed exactly c RMRs in H_c. Now suppose for contradiction that fewer than $\lfloor N^{1/3^c}/2 \rfloor$ of the active processes are stable. Then at least $\lfloor N^{1/3^c}/2 \rfloor$ are unstable, and moreover each one has performed exactly c RMRs in H_c. Let H'_c be the history obtained from H_c by erasing all active processes except for exactly $\lfloor N^{1/3^c}/2 \rfloor$ unstable ones, and then letting each unstable process make one more RMR (in any order). This history satisfies the following: $|Fin(H'_c)| = |Fin(H_c)| \leq c$, $|Act(H'_c)| = \lfloor N^{1/3^c}/2 \rfloor$, and each process in $Act(H'_c)$ has incurred $c+1$ RMRs. Consequently, $|Par(H'_c)| \leq c + \lfloor N^{1/3^c}/2 \rfloor$, and the total number of RMRs incurred in H'_c is at least $(c+1)\lfloor N^{1/3^c}/2 \rfloor = c\lfloor N^{1/3^c}/2 \rfloor + \lfloor N^{1/3^c}/2 \rfloor$, which is greater than c times $|Par(H'_c)|$ when N is large enough. This contradicts the RMR complexity of $\mathcal{A}$. □

Lemma 6.12. *There exists a regular history $H \in \mathcal{H}_\mathcal{A}$ in which there are exactly $\lfloor N^{1/3^c}/2 \rfloor$ stable active processes, at most c finished processes, and no other processes participating. Furthermore, in H, each process that participates incurs at most c^2 RMRs in the DSM model.*

Proof sketch: By Lemma 6.11, the history H_c referred to by Lemma 6.10 contains at least $\lfloor N^{1/3^c}/2 \rfloor$ stable processes. It also contains at most c finished processes by Lemma 6.10. To construct H, take H_c and erase all active processes except $\lfloor N^{1/3^c}/2 \rfloor$ unstable ones. The remaining processes incur the same number of RMRs in H as in H_c, which is bounded in Lemma 6.10: each active process in H_c incurs at most c RMRs, and each finished process incurs at most c^2 RMRs. Thus, each process incurs at most c^2 RMRs, as wanted. □

We now describe how to use the history H referred to by Lemma 6.12 to derive a contradiction.

Lemma 6.13. *For large enough N, there exists a history $H' \in \mathcal{H}_{\mathcal{A}}$ and a constant $k \in \mathbb{Z}^+$, such that at most k processes participate in H and yet the total number of RMRs incurred in H is more than ck.*

PROOF SKETCH: Let H be the history referred to by Lemma 6.12. In this history, each process that participates incurs at most c^2 RMRs, and at most $c + \lfloor N^{1/3^c}/2 \rfloor$ processes participate. Thus, in total the participating processes write to at most $(c+\lfloor N^{1/3^c}/2 \rfloor)(1+c^2)$ distinct memory modules. (Each process may write its own module, and at most c^2 remote ones.) For N large enough, this means there is some process s whose memory module is not written in H.

Now construct H' from H as follows. First, let each stable process run solo, one by one, completing any pending call to `Poll()` that it may have. Since each process is stable, by Definition 6.8, it will not incur any RMRs doing so. Now let s run solo and make a call to `Signal()`, which must eventually terminate. As s performs this call, each time s is about to see a process p that is active in H, or is about to write memory local to p, erase p and then allow s to take its step. Note that this step by s must be an RMR because $s \neq p$, and because by our choice of s, process p has never written memory local to s. It follows that s performs one such RMR for each stable process $p \in Act(H)$, otherwise after s completes its call to `Signal()` there is some p that remains stable and whose local memory is in the same state as at the end of H. Consequently, if p now calls `Poll()`, then its call returns the same response as p's last call, namely false, contradicting the specification of the signaling problem (see Definition 4.1). Thus, s performs at least $\lfloor N^{1/3^c}/2 \rfloor$ RMRs. Finally, erase any active process that remains, which leaves only s and at most c finished processes participating in the history. Let H' denote this history. Note that at most $k = c+1$ processes participate in H', and yet s incurs at least $\lfloor N^{1/3^c}/2 \rfloor$ RMRs in H' in the DSM model. For large enough N, this means that the number of RMRs is more than $ck = c(c+1)$, as wanted. □

PROOF OF THEOREM 6.2: Lemma 6.13 contradicts our assumption on the RMR complexity of $\mathcal{A}$. Thus, $\mathcal{A}$ does not exist and Theorem 6.2 holds. □

Corollary 6.14. *For any deterministic algorithm $\mathcal{A}$ that solves the signaling problem (with polling semantics) using atomic reads and writes, and either CAS or LL/SC, and for any constant $c \in \mathbb{Z}^+$, there exists a constant $k \in \mathbb{Z}^+$ and a history $H \in \mathcal{H}_{\mathcal{A}}$ where k processes participate and incur more than ck RMRs in total in the DSM model.*

Proof. Suppose for contradiction that Corollary 6.14 is false, namely that some deterministic terminating algorithm $\mathcal{A}$ exists that solves signaling problem using the stated set of base object types, and there is a constant $c \in \mathbb{Z}^+$ such that for any $H \in \mathcal{H}_{\mathcal{A}}$ the total number of RMRs incurred in the DSM model in H is at most c times the number of processes participating. Replacing the variables accessed via CAS or LL/SC in $\mathcal{A}$ with the locally-accessible $O(1)$-RMR implementations of these primitives presented in [11, 12], we obtain another terminating algorithm $\mathcal{A}'$ that solves the signaling problem using only atomic reads and writes, and there exists a constant $c' \in \mathbb{Z}^+$ such that for any $H \in \mathcal{H}_{\mathcal{A}'}$

the total number of RMRs incurred in the DSM model in H is at most c' times the number of processes participating. The existence of $\mathcal{A}'$ contradicts Theorem 6.2. □

7 Additional complexity bounds

In this section we discuss solutions to variations of the signaling problem defined in Section 4. Recall that there are two flavors of the problem: with polling semantics, waiters repeatedly call `Poll()` to determine if the signal has been issued; with blocking semantics, waiters instead call `Wait()`, which does not return until the signal has been issued. Both flavors of the signaling problem are solved easily in the CC model using $O(1)$ RMRs per process worst-case. The solution is presented in Section 5.

As one might suspect, in light of our lower bound result, the solution space for variations of the signaling problem in the DSM model is somewhat more complex. The solutions proposed for the CC model do not work "out of the box" in the DSM model in the sense that they have unbounded RMR complexity. Nevertheless, RMR-efficient algorithms for the DSM model can be devised using more elaborate synchronization techniques. We explore such solutions in detail in the remainder of this section. For each problem variation we consider, we will describe the polling solution, from which blocking solution can be achieved easily by implementing `Wait()` via repeated execution of the code for `Poll()`. In some cases, a more efficient solution is possible for blocking semantics, as we indicate below.

Single waiter
If there is at most a single waiter, and its ID is not necessarily fixed in advance, the problem can be solved using two global variables, say W (process ID, initially NIL) and S (Boolean, initially false), as well as an array $V[1..N]$ of Boolean variables (initially false), where $V[i]$ is local to process p_i. The first call to `Poll()` writes the waiter's ID to W, and then reads and returns the value of S. On subsequent invocations by process p_i, `Poll()` reads and returns $V[i]$ instead. `Signal()` sets S to true, and then reads W. If NIL is read, `Signal()` returns immediately. If a non-NIL value is read, it must be the waiter's ID, say p_j, in which case the signaler writes true to $V[j]$. This algorithm has $O(1)$ RMR complexity per process in the worst case, matching the upper bound for the CC model.

Many waiters, fixed in advance
In this formulation of the problem, the signaler knows in advance the IDs of the waiters that will participate eventually (i.e., if the history is extended by sufficiently many steps). A simple solution in this case is to use an array of Boolean variables $V[1..N]$, initially all false, $V[i]$ local to process p_i. A call to `Poll()` by p_i reads and returns $V[i]$, and `Signal()` sets $V[j]$ for each waiter p_j whose ID is fixed in advance. Letting W denote the number of waiters, this solution

has $O(W)$ RMR complexity per process worst-case. However, amortized RMR complexity may be more than $O(1)$ RMRs if the signaler performs W RMRs but only $o(W)$ waiters participate so far in the history. (This is possible if the history is not yet long enough for every fixed waiter to begin participating.)

For terminating solutions, it is easy to reduce the amortized RMR complexity to $O(1)$ in all histories. The signaler can simply wait for each waiter to participate, using another array of Boolean flags, before writing any element of V. With blocking semantics, the worst-case RMR complexity per process can also be reduced to $O(1)$ using the work sharing techniques described in [13].

For wait-free solutions using reads, writes and comparison primitives, $O(1)$ amortized RMR complexity is impossible to achieve for all histories when W is large enough (e.g., $W \in \Theta(N)$). This result follows by a lower bound proof similar to the one given in Section 6. (Although the signaler knows which waiters will participate eventually, it does not know which waiters participate at the time when it calls `Signal()`, which must return in a bounded number of steps.)

For terminating solutions with polling semantics, it is also possible to show that in the worst case the signaler must perform $\Omega(W)$ RMRs if all W waiters participate by the time `Signal()` is called. This follows by a simplified version of the lower bound proof from Section 6. (This simplified proof is similar in spirit to the lower bound proof of Hadzilacos and Danek [8], but does not rely on any form of wait-freedom.) The main idea is as follows: First, W waiters call `Poll()` repeatedly until they are all stable (i.e., accessing only local memory). We then allow each waiter to complete any ongoing call to `Poll()` it may have. Next, a signaler makes a call to `Signal()`, which must terminate in a finite number of steps because the signaler is running solo starting from a state where each waiter is not required to take any more steps (i.e., it is "in between" calls to `Poll()`). Furthermore, before this call to `Signal()` terminates, the signaler must write remotely to the local memory of each waiter, except possibly itself, which incurs $\Omega(W)$ RMRs. To see this, suppose that the signaler neglects to write p_i's memory for some waiter p_i different from itself. In that case, after the call to `Signal()` completes, p_i may make another call to `Poll()` that will incorrectly return the same value as p_i's previous call (i.e., false).

Many waiters not fixed in advance, one signaler fixed in advance
The solution is similar to the case with fixed waiters, with a few additional steps. Upon calling `Poll()` for the first time, a waiter "registers" with the signaler by setting a dedicated Boolean flag in the signaler's local memory. The signaler, upon calling `Signal()`, checks for each i whether p_i has registered, and if so, performs a remote write to $V[i]$. In addition, we must handle correctly the race condition when waiters register while the signaler is calling `Signal()`. To that end, it suffices to use an additional global variable analogous to S from the single waiter case. The signaler writes S at the beginning of `Signal()`, and waiters check S at the end of their first call to `Poll()` (i.e., after registering).

A terminating algorithm for this version of the problem appears in [12]. It uses using atomic reads and writes only and incurs $O(1)$ RMRs per process in the worst case.

Many waiters not fixed in advance, one signaler not fixed in advance
With polling semantics the problem is subject to the lower bound from Section 6. That is, if only reads, writes and CAS or LL/SC are used, the problem can be solved more efficiently in the CC model than in the DSM model with respect to amortized RMR complexity. It is possible to close this gap by using stronger primitives. Recall from Section 3 that if Fetch-And-Increment or Fetch-And-Add are available in addition to reads and writes, then it is possible to solve mutual exclusion using $O(1)$ RMRs per process, which can be used to construct a shared queue with the same RMR complexity. Waiters and signalers can leverage such a queue as follows: During its first call to `Poll()`, a waiter adds itself to the queue, and also checks a global flag to see if a signaler has started a call to `Signal()`. During subsequent calls to `Poll()`, a waiter only checks a dedicated flag in its own local memory. During a call to `Signal()`, the signaler sets the global flag, then retrieves the set of all waiters from the shared queue, and writes the dedicated flag for each waiter found in the queue. The worst-case RMR complexity per process of this solution is $O(1)$ for waiters, and $O(k)$ for the signaler when there are k waiters participating. Thus, amortized RMR complexity is $O(1)$.

With blocking semantics, the problem can be reduced to the single-waiter case by having the waiters elect a leader, which learns about the signal and then ensures that the signal is propagated to the remaining waiters. Using the synchronization techniques described in [12], such a solution is possible with $O(1)$ RMR complexity per process worst-case, using only atomic reads and writes. (The leader election algorithm must tell each waiter the ID of the leader rather than merely telling each waiter whether it is the leader.)

Many waiters not fixed in advance, many signalers
One possibility is to reduce this case to "one signaler not fixed in advance" by having signalers elect a leader that will signal the waiters. Leader election can be solved in $O(1)$ RMRs per process worst-case using atomic reads and writes by a terminating algorithm [13], or in one step per process using virtually any

read-modify-write primitive (e.g., Test-And-Set or Fetch-And-Store).

8 RMRs vs. Observed Performance

In this section, we discuss the relationship between RMR complexity and observed performance in real world multiprocessors, and comment on the practical interpretation of our complexity separation result.

The RMR complexity measure attempts to quantify inter-process communication by counting memory accesses that engage the interconnect joining processors and memory (see Figure 1). Since the interconnect is slow relative to processor speed, observed performance tends to degrade as communication over the interconnect increases. This effect has been demonstrated in the context of mutual exclusion algorithms on the Sequent Symmetry (a CC machine) and the BBN Butterfly (a DSM machine) [4, 14, 28, 30]. The key lesson from this body of literature is that algorithms with bounded RMR complexity (i.e., so-called *local-spin* algorithms) outperform algorithms that have unbounded RMR complexity by a significant margin under worst-case conditions (i.e., maximum concurrency).

Although the benefits of local-spin algorithms are well understood, it is important to note that RMR complexity is not a very precise tool for predicting real world performance, especially for cache-coherent machines. For example, to derive our upper bound in Section 5, we made the simplifying assumption that a cache behaves "ideally," meaning that it never drops data spuriously (see Section 2). Since this assumption does not hold in a preemptive multitasking environment, especially under high load, theoretical RMR complexity bounds are prone to underestimate the actual number of RMRs. An even more important concern is the imprecise relationship between RMRs and communication. We focus on the latter issue in more detail in the remainder of this section.

Consider the simplified example of a cache-coherent system where processes communicate using read and write operations, and coherence is ensured by a *write-through* protocol [29]. In such a system, a read operation on a shared object O either finds a cached copy of O locally, or else it fetches O from main memory and creates a copy of O in the local cache. A write operation on O applies the new value for O to main memory, creates (or updates) a copy of O in the local cache, and invalidates (i.e., destroys) all copies of O in remote caches. Thus, while an RMR on read generates a fixed amount of communication, an RMR on write may trigger multiple "invalidation messages." This is in contrast to the DSM model where, in the absence of a coherence protocol, any RMR generates a fixed amount of communication.

The above example illustrates that RMRs in the CC model and RMRs in the DSM model are very different "currencies" for describing the cost of an algorithm. Consequently, a meaningful comparison between them requires that we define the "exchange rate." To that end, we must fix a particular cache-coherent architecture and coherence protocol. The simplest scenario occurs when the interconnect joining processors and memory is a shared bus, and any

message generated by the coherence protocol is broadcast to all processors. In that case, a single message suffices to invalidate all remote copies of an object on write, and so RMRs in this type of system are "at par" with RMRs in the DSM model.

Since a shared bus offers limited communication bandwidth, realistic large-scale cache-coherent systems use more elaborate interconnects. In such systems, an RMR on write may generate multiple invalidation messages, the exact number depending on the topology of the interconnect and the state information maintained by the coherence protocol. Consequently, RMR complexity per process can underestimate vastly the amount of communication triggered by a process. But what about amortized RMR complexity, which is the focus of this paper?

A key observation is that in any cache-coherent system, a cached copy of a shared object can be invalidated at most once, because invalidation destroys it. Since an RMR is necessary to create a cached copy in the first place, this means that the total number of *invalidations* (i.e., events where a cached copy of an object is destroyed) is bounded from above by the number of RMRs. The implications of this on message complexity depend on how the number of invalidations relates to the number of invalidation messages that trigger them.

Ideally, the cache coherence protocol would maintain sufficient information so that an invalidation message for a particular shared object O is only sent to remote caches that actually hold a copy of O. This is an unrealistic assumption because in an N-processor system, it requires roughly N bits of state for each cached object! However, it does ensure that at most one invalidation message is generated for each invalidation that actually occurs. In that case, amortized RMR complexity corresponds well to amortized message complexity.

In realistic large-scale cache-coherent systems, the coherence protocol maintains far less state than in an ideal system, and so RMRs may trigger superfluous invalidation messages (i.e., ones that do not actually cause an invalidation). In such systems, amortized RMR complexity may be lower asymptotically than amortized message complexity, and so our RMR complexity separation does not imply that in practice a large-scale cache-coherent machine could solve the signaling problem more efficiently than a DSM machine with the same number of processors.

9 Conclusion

In this paper, we showed that the signaling problem can be solved in a wait-free manner using only atomic reads and writes, with $O(1)$ RMRs per process in the CC model, and $O(1)$ space. We then showed that the same problem cannot be solved in the DSM model with the same parameters, even if we weaken these parameters in all of the following ways simultaneously: (1) we allow only one signaler instead of many, (2) we settle for $O(1)$ RMRs complexity in the amortized sense rather than in the worst case, (3) we allow unbounded space, (4) we weaken the progress condition from wait-free to terminating, and (5) we

allow use of Compare-And-Swap or Load-Linked/Store-Conditional in addition to reads and writes. Thus, we separate the CC and DSM models in terms of their power for solving the signaling problem efficiently with respect to amortized RMR complexity.

Acknowledgments Sincere thanks to Prof. Philipp Woelfel at the University of Calgary for stimulating discussions on the subject of RMRs and the relative power of the CC and DSM models for solving synchronization problems efficiently with respect to amortized RMR complexity. We are grateful to the anonymous referees for their insightful comments and suggestions. Special thanks to Dr. Mark Tuttle at Intel for his perspective on the performance of CC systems in the real world and the shortcomings of the RMR complexity measure. Thanks also to Dr. Ram Swaminathan at HP Labs and Dr. Robert Danek at Oanda for their careful proofreading of this work.

References

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

[2] J. Anderson and Y.-J. Kim. A generic local-spin fetch-and-phi-based mutual exclusion algorithm. *Journal of Parallel and Distributed Computing*, 67(5):551–580, 2007.

[3] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

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

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

[6] V. Bhatt and C.-C. Huang. Group mutual exclusion in $O(\log n)$ RMR. In *Proc. of the 29th ACM symposium on Principles of Distributed Computing*, pages 45–54, 2010.

[7] R. Danek and W. Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. *Distributed Computing*, 23(2):93–108, 2010.

[8] R. Danek and V. Hadzilacos. Local-spin group mutual exclusion algorithms. In *Proc. of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 71–85, 2004.

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

[10] R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *Proc. of the 25th annual ACM symposium on Principles of distributed computing*, pages 275–284, 2006.

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

[12] W. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In *Proc. of the 26th annual ACM symposium on Principles of distributed computing*, 2007.

[13] W. Golab, D. Hendler, and P. Woelfel. An O(1) RMRs leader election algorithm. In *Proc. of the 25th annual ACM symposium on Principles of distributed computing*, pages 238–247, 2006.

[14] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, 1990.

[15] V. Hadzilacos. A note on group mutual exclusion. In *Proc. of the twentieth annual ACM symposium on Principles of distributed computing*, pages 100–106, 2001.

[16] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1), 1991.

[17] P. Jayanti. A complete and constant time wait-free implementation of CAS from LL/SC and vice versa. In *Proc. of the 12th International Symposium on Distributed Computing*, pages 216–230, 1998.

[18] P. Jayanti, S. Petrovic, and K. Tan. Fair group mutual exclusion. In *Proc. of the 22nd ACM symposium on Principles of Distributed Computing*, pages 275–284, 2003.

[19] Y.-J. Joung. Asynchronous group mutual exclusion. *Distributed Computing*, 13(4):189–206, 2000.

[20] P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. *IEEE Trans. Parallel Distrib. Syst.*, 7(12):673–685, 2001.

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

[22] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, 84(1):47–55, 2002.

[23] Y.-J. Kim and J. Anderson. Timing-based mutual exclusion with local spinning. In *Proc. of the 17th International Symposium on Distributed Computing Systems*, pages 30–44, 2003.

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

[25] L. Lamport. The mutual exclusion problem: part II – statement and solutions. *J. ACM*, 33(2):327–348, 1986.

[26] H. Lee. Transformations of mutual exclusion algorithms from the cache-coherent model to the distributed shared memory model. In *Proc. of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 261–270, 2005.

[27] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

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

[29] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, second edition, 1997.

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