

Constant-RMR Implementations of CAS and Other Synchronization Primitives Using Read and Write Operations

[Extended Abstract]

Wojciech Golab^{*}
Department of Computer Science
University of Toronto
Toronto, Canada
wgolab@cs.toronto.edu

Danny Hendler
Department of Computer Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
hendlerd@cs.bgu.ac.il

Vassos Hadzilacos
Department of Computer Science
University of Toronto
Toronto, Canada
vassos@cs.toronto.edu

Philipp Woelfel[†]
Department of Computer Science
University of Toronto
Toronto, Canada
pwoelfel@cs.toronto.edu

ABSTRACT

We consider asynchronous multiprocessors where processes communicate only by reading or writing shared memory. We show how to implement consensus, all comparison primitives (such as CAS and TAS), and load-linked/store-conditional using only a *constant* number of remote memory references (RMRs), in both the cache-coherent and the distributed-shared-memory models of such multiprocessors. Our implementations are blocking, rather than wait-free: they ensure progress provided all processes that invoke the implemented primitive are live.

Our results imply that *any* algorithm using read and write operations, comparison primitives, and load-linked/store-conditional, can be simulated by an algorithm that uses read and write operations only, with at most a constant blowup in RMR complexity.

Categories and Subject Descriptors

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

^{*}Supported in part by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

[†]Supported by DFG grant WO1232/1-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PODC'07, August 12–15, 2007, Portland, Oregon, USA.
Copyright 2007 ACM 978-1-59593-616-5/07/0008 ...\$5.00.

General Terms

Algorithms, Performance, Theory

Keywords

Comparison primitives, consensus, mutual exclusion, remote memory references, shared memory.

1. INTRODUCTION

Work on synchronization in shared memory multiprocessors has largely focused on the asynchronous model, either with or without crash failures. In wait-free synchronization, a process must make progress through its own steps regardless of the execution speeds or crash failures of others; in blocking synchronization, processes may busy-wait for others by repeatedly accessing shared memory, and so progress is guaranteed only when every active process is live. (A process is live if, whenever it begins executing an algorithm, it continues to take steps until its algorithm terminates.)

In this paper we focus on blocking implementations in asynchronous multiprocessors where processes communicate only by reading or writing shared memory. A natural way to measure the time complexity of algorithms in such multiprocessors is to count the number of memory accesses. This measure is problematic for blocking implementations because, in this case, a process may perform an unbounded number of memory accesses while busy-waiting for another process. (For example, this happens in a mutual exclusion algorithm when a process waits for another to clear the critical section.) Instead, we can measure the time complexity of an algorithm by counting only *remote memory references* (RMRs), i.e., memory accesses that traverse the processor-to-memory interconnect. *Local-spin* algorithms, which perform busy-waiting by repeatedly reading *locally* accessible shared variables, achieve bounded RMR complexity and have practical performance benefits [4].

The classification of memory accesses into local and remote depends on the type of multiprocessor: In the *dis-*

tributed shared memory (DSM) model, each shared variable is local to exactly one process and remote to all others. In the *cache-coherent* (CC) model, each process maintains local copies of shared variables inside a cache; the consistency of copies in different caches is ensured by a coherence protocol. At any given time, a variable is local to a process if the coherence protocol guarantees that the corresponding cache contains an up-to-date copy of the variable, and is remote otherwise.

Summary of results.

(1) All *comparison* primitives [1], a class of synchronization primitives that includes the popular *compare-and-swap* (CAS), can be implemented using read and write operations with only a constant number of RMRs, in both the DSM and CC models. Using previously known techniques [11], we can also obtain such implementations for the *load-linked* and *store-conditional* (LL/SC) pair of primitives.

(2) Our constant-RMR implementations of comparison primitives can be made locally accessible just like their hardware-implemented counterparts. In the DSM model, this means that a designated process can access the implemented object without performing any RMRs. Similarly, in the CC model certain operations on an “in-cache” object cost no RMRs; whether an object is “in-cache” depends on the coherence protocol and the prior history of the execution.

(3) As a consequence of (1) and (2), any CC or DSM shared memory algorithm using read and write operations, LL/SC, and comparison primitives, can be simulated by an algorithm that uses only read and write operations, with only a constant blowup in the RMR complexity.

(4) A super-constant lower bound, in the DSM model, on the worst-case number of RMRs required by an operation in any implementation using only read and write operations of a particular type of resettable counter over the domain $\{0, 1, 2\}$. More precisely, the concurrent access of such a counter by N processes requires $\Omega(N \log \log N)$ RMRs. We will discuss the significance of this result later.

Our constant-RMR implementation of comparison primitives is obtained in a series of steps. We first show how to transform *any* constant-RMR *leader election* algorithm that uses read and write operations into a constant-RMR *name consensus* algorithm that uses read and write operations. (In a leader election algorithm, exactly one active process declares itself the winner, and all others declare themselves losers. In a name consensus algorithm, all active processes agree on one of their IDs.) Since there is a constant-RMR leader election algorithm [8], this transformation gives a constant-RMR name-consensus algorithm. This efficient name consensus algorithm is used, in turn, to obtain a constant-RMR CAS implementation. Finally, we observe that using CAS and no additional RMRs, one can easily implement any comparison primitive.

Related work and implications of our results. Herlihy has shown that synchronization primitives vary widely in their ability to support *wait-free* implementations, and can be classified in the *wait-free hierarchy*, where the level of a primitive corresponds to its power [9]. For example, CAS together with read and write operations supports wait-free implementations of arbitrary objects shared by *any* number of processes; as a result, CAS is at the top level of the wait-free hierarchy. In contrast, the primitive *fetch-and-store* (FAS), together with read and write operations, supports wait-free

implementation of arbitrary objects shared by at most *two* processes; as a result, FAS is at level two of the wait-free hierarchy.

As regards *blocking* synchronization, however, all primitives can be implemented using only read and write operations, by using such operations to implement mutual exclusion [7]. Thus, it is not meaningful to compare the power of two primitives by asking whether one can be used, along with read and write operations, to implement the other. Instead of comparing the power of two primitives based on computability, it is natural to ask whether we can base such a comparison on complexity — specifically, the RMR complexities of implementing each of the two primitives using read and write operations. From this point of view, we say that a primitive S is stronger than a primitive W if the RMR cost of implementing S using read and write operations is (asymptotically) higher than that of implementing W .

Looking at the relative power of primitives through this lens reveals a landscape very different from that of Herlihy’s wait-free hierarchy. Some primitives classified as strong in the wait-free hierarchy have low RMR-cost implementations from read and write operations. Conversely, some primitives classified as weak in the wait-free hierarchy have inherently high RMR-cost implementations from reads and writes. For example, CAS is at the top of the wait-free hierarchy but, as we show in this paper, it can be implemented from read and write operations using only a constant number of RMRs. On the other hand, FAS is only at level two of the wait-free hierarchy, but any implementation of FAS from read and write operations requires $\Omega(\log N / \log \log N)$ RMRs in the worst case. This follows from the fact that mutual exclusion can be solved with $O(1)$ RMRs per passage through the critical section using FAS along with read and write operations [5], but requires $\Omega(\log N / \log \log N)$ RMRs per passage in the worst case using only reads and writes [1]. The same holds for the primitive *fetch-and-add* (FAA), which is also at level two of the wait-free hierarchy [9], but can be used, along with read and write operations, to solve mutual exclusion in $O(1)$ RMRs per passage [4].

Our results also have some interesting implications regarding mutual exclusion. To explain these we first recall certain facts about the RMR complexity of mutual exclusion. The most RMR-efficient mutual exclusion algorithm that uses only read and write operations known to date is one devised by Yang and Anderson; it performs $\Theta(\log N)$ RMRs per passage through the critical section [13]. On the other hand, Anderson and Kim have shown that any mutual exclusion algorithm that uses read and write operations and comparison primitives requires $\Omega(\log N / \log \log N)$ RMRs per passage in the worst case [1]. Closing the gap between these bounds remains an interesting open question.

Note that the lower bound encompasses algorithms that use comparison primitives, in addition to read and write operations. Thus, one question (raised by Anderson, Kim, and Herman [3]) is whether we can improve the upper bound by employing algorithms that, in addition to read and write operations, also use comparison primitives. Our result (3) implies that the answer is no.

Anderson and Kim have defined a resettable, two-bounded counter that, together with read and write operations, can be used to solve mutual exclusion in $O(\log N / \log \log N)$ RMRs [2]. Thus, one approach to closing the gap on the RMR complexity of mutual exclusion is to find a constant-

RMR implementation of the Anderson-Kim counter using read and write operations. (This would yield a mutual exclusion algorithm with $O(\log N / \log \log N)$ RMR complexity using only read and write operations, beating the Yang-Anderson upper bound and meeting the Anderson-Kim lower bound.) Our result (4) shows that this is not possible in the DSM model: Any implementation of this counter requires a worst-case *average* of $\Omega(\log \log N)$ RMRs per operation. Cypher obtained an $\Omega(\log \log N / \log \log \log N)$ lower bound on the average RMR complexity of mutual exclusion [6]. In the DSM model, our lower bound on the Anderson-Kim counter is stronger than Cypher’s result.

Notation. We apply the following notational conventions. The set \mathcal{P} of processes is $\{p_1, p_2, \dots, p_N\}$. For any $p_i \in \mathcal{P}$, we treat p_i as a synonym for i , its *ID*. In the pseudo-code of an algorithm, PID denotes the ID of the executing process. We denote by $B \leftarrow \text{read}(A)$ a read of shared variable A to private variable B . Similarly, we denote by $\text{write } D \leftarrow C$ a write of the constant or private variable C to shared variable D . We omit the keywords **read** and **write** above if A and D are local to the executing process in the DSM model, or if they are private variables. Whenever a separate copy of a shared variable V is defined for each process, we denote process p ’s copy by V_p , which in the DSM model means that V_p is local to p . We denote by **await cond** a busy-wait loop that repeatedly evaluates condition *cond* and terminates when *cond* evaluates to true.

2. CONSENSUS

In this section, we obtain an $O(1)$ RMRs implementation of consensus from reads and writes. We consider a special variant of consensus that we call *name consensus*, from which ordinary consensus follows by a straight-forward transformation that preserves RMR complexity.

In the name consensus problem, an arbitrary subset of processes run until they all agree on a common value, which is the name (ID) of one of them. A process *wins* if its name is agreed upon and *loses* otherwise. More formally, each process executes a procedure **NameDecide**, which returns the ID of one of the invoking processes. **NameDecide** may be invoked at most once by each process, and if called by multiple processes, returns the same value to all callers. Thus, in sequential executions, **NameDecide** returns the ID of the first process that executes it.

Name consensus is straight-forward to solve in the CC model using leader election. Informally, the algorithm works as follows. Every process first competes in leader election. Then, the winner writes its ID to a register *leader* initialized to \perp , and each loser locally spins on *leader* to discover the winner’s ID. We defer further details of the CC algorithm to the full version of the paper, and focus on the more interesting DSM case for the remainder of this section.

2.1 Name Consensus in the DSM Model: A High-Level Description

Our implementation of name consensus uses a *leader election* object. In the leader election problem, exactly one process, the leader, must be distinguished from all others. The leader election type supports a single operation, **LeaderElect**. The single process elected as leader must receive response *win* from **LeaderElect** and all other participating processes must receive response *lose*.

We show how to implement name consensus at a cost of $O(1)$ RMRs per process in the worst case. Our implementation uses an $O(1)$ -RMRs leader election object L , such as the one presented in [8], and atomic read/write registers. The way we use L is derived from the following observation: *After any execution of a leader election algorithm in which all active processes terminate, there is a flow of data (either direct or indirect) between the process elected leader and any other participating process.* More precisely, a (direct) “flow of data” from process p to process q means that q reads a variable last written by p . In order to use this information, our algorithm uses an “instrumented” version of L (denoted \hat{L}) as a base object rather than using L directly. In addition to returning the usual response, the instrumented leader election object returns to a calling process p a set S_p consisting of IDs of processes with which p exchanged data (i.e., the flow could be to or from p) by way of an RMR.

The sets S_p that arise after every active process finishes leader election induce a connected data flow graph G in which nodes represent processes, and (p, q) is an edge whenever $q \in S_p$ or $p \in S_q$. (The graph may also contain edges arising when p writes to q ’s memory but q is not active, in which case $q \in S_p$ but there is no flow of data between p and q .) One of the nodes in this graph represents the leader. The main challenge of the algorithm is to synchronize the efficient dissemination of the leader’s ID across G . This challenge is complicated by two key technical difficulties. We now describe these difficulties and explain how we cope with them.

Reliable Inter-Process Communication. The asynchrony of the system makes it difficult to establish reliable communication between pairs of processes. To exemplify this difficulty, consider the information represented by the sets S_p . This information cannot be used directly for inter-process communication. For example, if there was flow of data from q to p during leader election, it is possible that p is subsequently delayed while q proceeds further in the computation or even terminates. Thus, if p writes to q ’s memory and then busy-waits for a response from q , it might wait forever. To overcome this problem, we use an inter-process handshaking protocol, introduced in [8], through which p and q can efficiently agree on whether or not they can communicate. In the terminology of [8], p and q agree on whether or not they share a “communication link”.

The handshaking mechanism is implemented by the functions **LinkRequest** and **LinkReceive** described in [8]. For presentation completeness, we now provide a short explanation of how these functions work and list their code in the Appendix.

Process p calls **LinkRequest** in an attempt to establish a “communication link” with some process it is aware of, say q . **LinkRequest** receives as input the ID of a process (q , in our example) and returns a response of 0 or 1. If it returns 1, then we say that *a link from q to p is established*. As proved in [8], if a link from q to p is successfully established, then q eventually becomes aware of p . In that case, p and q can communicate safely using blocking techniques without risk of an endless loop. The **LinkReceive** function receives no arguments, and returns a subset of \mathcal{P} .

The key properties of **LinkRequest/LinkReceive** are captured by the following lemma, whose proof appears in [8].

LEMMA 2.1. Consider an execution where some process q calls `LinkReceive` at most once, every process $p \in \mathcal{P}$ calls `LinkRequest(q)` at most once, and every active process is live. Then the following claims hold in this execution:

- (a) each call to `LinkRequest` and `LinkReceive` terminates; and
- (b) p is in the set returned by q 's call to `LinkReceive` if and only if a link from q to p is eventually established (i.e., p 's call to `LinkRequest(q)` returns 1); and
- (c) if q 's call to `LinkRequest(p)` terminates before q starts executing `LinkReceive`, then a link from q to p is established; and
- (d) each call to `LinkRequest` costs $O(1)$ RMRs, whereas each call to `LinkReceive` costs zero RMRs.

Communication Load Balancing. As mentioned before, our name consensus algorithm disseminates the leader's ID across the data flow graph G . A straightforward algorithm for doing that is to have each process p notify all its neighbors in G about the leader's ID. With this algorithm, however, the number of RMRs incurred by p might be super-constant (if p has a super-constant number of neighbors in G). Rather than having p notify all its neighbors directly, our algorithm uses a notification mechanism that guarantees that the communication load of p 's notification task is shared between its neighbors in G .

Informally, the notification mechanism works as follows. When p needs to communicate the leader's ID to a subset \mathcal{N}_p of its neighbors (i.e., those with which p has established a communication link), it builds a chain of the IDs in \mathcal{N}_p in its local memory. Process p then signals the first process in the chain, say q , by writing p 's ID in a designated location in q 's memory. This costs p a single RMR. Each such process q then reads the leader's ID from p 's memory and signals the next process in the chain (if any), whose ID it also reads from p 's local memory. Our algorithm makes sure that, for each such process p , all the processes in \mathcal{N}_p wait for p 's (either direct or indirect) notification, as otherwise the notification mechanism will fail.

The notification mechanism is implemented by the pair of helper functions `signal` and `wait`, presented below. The argument of `signal` is a non-empty subset $P \subseteq \mathcal{P}$ of processes to notify. On lines 1–5, the calling process p uses its local `Work` array to create a “chain” of identifiers from P (all elements of `Workp` are initially \perp). This chain determines the order in which the processes in P notify each other. To start the notification mechanism, p assigns `true` to p 'th entry of the array D that is local to the process at the beginning of the chain (line 6).

Function `wait` is the counterpart of `signal` whereby a process waits to be notified. The argument of `wait` is a non-empty subset $Q \subseteq \mathcal{P}$ of processes that may notify the caller. The process p executing `wait` waits for notifications from other processes by locally spinning on entries of the array D_p (each element initially `false`) until a process that precedes it in a notification chain writes `true` to some entry (see lines 2–6). On line 7, the identifier of the next process in the chain (if any) is read. If such a process exists (i.e., the caller is not the last one in the chain), then it is notified (lines 8–9). Finally, the ID of the process that signalled the caller is returned on line 11.

The behavior of `signal/wait` is formally stated by the following lemma, whose proof is provided in the full paper.

LEMMA 2.2. Consider an execution involving `signal` and `wait`, where the argument of each call to these functions is a nonempty subset of \mathcal{P} , and where every active process is live. Then the following claims hold in this execution:

1. each call to `signal` and `wait` costs $O(1)$ RMRs; and
2. each call to `signal` terminates in a bounded number of steps; and
3. if process p calls `wait(Q)` and receives response q then $q \in Q$, and moreover q called `signal(P)` with $p \in P$; and
4. if process q completes a call to `signal(P)` where it writes $D_p[q]$ on line 6, and if p calls `wait(Q)` with $q \in Q$, then p 's call terminates; and
5. if some process completes a call to `wait(Q)` where it reads p 's ID on line 7 and returns response q (after writing $D_p[q]$ on line 9), and if p calls `wait(Q)` with $q \in Q$, then p 's call terminates.

Function `signal(P)`

Input: $P \subseteq \mathcal{P}, P \neq \emptyset$

```

1  $prev \leftarrow \perp$ 
2 foreach  $next \in P$  do
3   | write  $Work_{PID}[next] \leftarrow prev$ 
4   |  $prev \leftarrow next$ 
5 end
6 write  $D_{prev}[PID] \leftarrow true$ 
7 return OK

```

Function `wait(Q)`

Input: $Q \subseteq \mathcal{P}, Q \neq \emptyset$

Output: $q \in Q$

```

1  $q \leftarrow \perp$ 
2 repeat
3   | foreach  $t \in Q$  do
4     | if  $D_{PID}[t] = true$  then  $q \leftarrow t$ ; break loop
5     | end
6 until  $q \neq \perp$ 
7  $next \leftarrow read(Work_q[PID])$ 
8 if  $next \neq \perp$  then
9   | write  $D_{next}[q] \leftarrow true$ 
10 end
11 return  $q$ 

```

2.2 Name Consensus in the DSM Model: A Detailed Description

As mentioned in Section 2.1, we implement the DSM name consensus algorithm using an “instrumented” leader election object \hat{L} , which we construct using a given leader election object L with $O(1)$ worst-case RMR complexity.

The high-level idea is the following: Processes simulate a call to `L.LeaderElect`. Whenever a process p writes a value v to a register during this call, it now writes (v, p) , instead. This way, a process reading that register can find out who has written to it last. If process p reads register r and notices that some other process q has written to that register, then it stores the name q in set S_p . Later, it can try to establish communication links with all processes in S_p . Since the RMR complexity of establishing communication links with all processes in S_p will be linear in the size of S_p , we have to make sure that $|S_p|$ is constant. This is no problem, as long as p records only those processes of which it becomes aware

by reading remotely, because p can only read from a constant number of remote registers. However, p can read arbitrarily many local registers. Therefore, instead of adding q to S_p when p becomes aware of q due to a local read, process q adds p to its own set S_q when it writes remotely to one of p 's registers. This way, we ensure that all sets S_p , $p \in \mathcal{P}$, have constant size, and if p becomes aware that q participates in the leader election, then either $q \in S_p$ or $p \in S_q$.

We construct the “instrumented” object \hat{L} from L as follows. For every register r initialized to value v by L , initialize r to (\perp, v) . For every uninitialized register r that may be accessed by L , initialize r to (\perp, \tilde{v}) for some arbitrary value \tilde{v} . Each process $p \in \mathcal{P}$ records a set $S_p \subseteq \mathcal{P}$, initially $S_p = \emptyset$. To execute $\hat{L}.$ LeaderElect, a process p begins simulating its steps in $L.$ LeaderElect. If its next step in $L.$ LeaderElect writes value v to register r in process q 's memory module, then p writes (p, v) to r , adds q to S_p if $q \neq p$, and then performs the same local computation it does in that step of $L.$ LeaderElect (if any). If its next step in $L.$ LeaderElect reads register r in process q 's memory module, then p reads r , and decodes the value read into a pair (z, v) . In addition, p adds z to S_p if $z \neq \perp \wedge z \neq p \wedge q \neq p$ holds, and then performs the same local computation it does in that step of $L.$ LeaderElect (if any) using v as the value read from r . Thus, p repeatedly simulates its steps in $L.$ LeaderElect until termination, at which time p 's execution also terminates and produces a set S_p containing the IDs of certain other processes with which p exchanged information during the execution. $\hat{L}.$ LeaderElect returns to the caller the same response as $L.$ LeaderElect upon termination.

The access procedure for the NameDecide-DSM operation is presented below. The algorithm uses the following variables. For each $p \in \mathcal{P}$, register $leader_p$ eventually stores the ID of the process elected as leader; $leader_p$ is initialized with \perp . We say that p *knows the leader* if and only if $leader_p \neq \perp$. The set S_{PID} , referenced on line 5, is the set computed during execution of $\hat{L}.$ LeaderElect() on line 1, as described earlier. We refer to p and q as *neighbors* iff $p \in S_q$ or $q \in S_p$ (i.e., there is an edge between p and q in the data flow graph G defined in Section 2.1). As we prove in the full paper, any two processes active in the execution are connected by some chain of neighbors that are also active.

After performing line 1, process p needs to communicate with its neighbors regarding the winner's ID. In particular, p must communicate with the subset of neighbors in S_p . For that, we employ the handshaking functions `LinkRequest` and `LinkReceive` defined in [8] and described above. We use two instances of these functions, whereby each instance has its own distinct copy of the underlying shared variables. In the pseudo-code below we distinguish between these two instances by using the subscripts 1 and 2.

Now, consider the outcome of executing line 1. If p is the single winner of \hat{L} , then $leader_p$ is set to p 's identifier on line 2. On lines 4–11, p tries to establish links with the subset of its neighbors from S_p . The identifiers of neighbors with which a link is successfully established are stored in the set U , which is initialized on line 4. If p fails to establish a link with some $q \in S_p$, then q is active (by the properties of `LinkRequest/LinkReceive`) and, as our proofs show, q already knows the leader. In this case, p may obtain the leader's ID from $leader_q$ (lines 8–9).

If p is not the winner on line 1 and it successfully establishes links with each $q \in S_p$, then p still does not know

Function NameDecide-DSM

Output: PID of leader

```

1 if  $\hat{L}.$ LeaderElect() = win then
2   |  $leader_{PID} \leftarrow PID$ 
3 end
4  $U \leftarrow \emptyset$ 
5 foreach  $q \in S_{PID}$  do
6   | if LinkRequest1( $q$ ) then
7     |  $U \leftarrow U \cup \{q\}$ 
8   | else if  $leader_{PID} = \perp$  then
9     |  $leader_{PID} \leftarrow read(leader_q)$ 
10  | end
11 end
12 if  $leader_{PID} = \perp$  then
13   |  $s \leftarrow wait(\mathcal{P})$ 
14   |  $leader_{PID} \leftarrow read(leader_s)$ 
15 end
16  $V \leftarrow \emptyset$ 
17 foreach  $q \in U$  do
18   | if LinkRequest2( $q$ ) then
19     |  $V \leftarrow V \cup \{q\}$ 
20   | end
21   | signal( $\{q\}$ )
22 end
23  $X \leftarrow LinkReceive_1()$ 
24  $Y \leftarrow LinkReceive_2()$ 
25  $Z \leftarrow (X \setminus Y) \setminus U$ 
26 if  $Z \neq \emptyset$  then signal( $Z$ )
27  $W \leftarrow U \setminus V$ 
28 while  $W \neq \emptyset$  do
29   |  $s \leftarrow wait(W)$ 
30   |  $W \leftarrow W \setminus \{s\}$ 
31 end
32 return  $leader_{PID}$ 

```

the leader when it reaches line 12. In this case, p waits on line 13 by calling the `wait` function. We prove in the full paper that this call eventually returns the ID of a neighbor s of p that knows the leader. On line 14, p obtains the ID of the leader from $leader_s$. Thus, by the time p reaches line 16, it already knows the leader's ID.

On lines 16–22, p makes sure that its neighbors from U also know the leader. At the same time, p computes a subset $V \subseteq U$ of neighbors that are either inactive in the computation, or have not yet completed line 24. This step is required since some neighbors of p from U may later try to notify p of the leader (via `signal`), and will rely on p to call `wait` and assist in the notification mechanism; at the same time, U may contain IDs of inactive processes that will never notify p . After successfully establishing a link with a neighbor via `LinkRequest`₂, p knows it must not wait for that neighbor later. Thus, by executing lines 16–22, p fulfills its duty to share the leader's ID with its neighbors in S_p , and determines which of these neighbors it may have to assist later.

Next, p attempts to communicate with the subset of its neighbors that are not in S_p . These neighbors may rely on p to learn the leader's ID. To determine the IDs of such neighbors, p calls two instances of `LinkReceive` on lines 23–24 and stores the responses to sets X and Y , respectively. These sets contain the identifiers of neighbors that success-

fully established a link with p by calling `LinkRequest` (see line 6 and line 18, respectively). Then, on lines 25–26, p ensures that all the processes in $X \setminus Y$ know the leader (those in Y have all reached line 16 and so, as explained earlier, they already know the leader). Note that processes in U have all been notified by p on line 21, and so p excludes them on line 25 where it computes the set of processes that remain to be informed. Since the set Z may be large, p cannot directly communicate with all the processes in it. Rather, p calls `signal(Z)` on line 26 to start a “notification chain” in which all of p ’s neighbors in set Z get to know the leader, and in which each process incurs only $O(1)$ RMRs.

Finally, p must assist some of its neighbors by sharing work in the notification chains initiated by them on line 26. This is done on lines 27–31. Process p first computes the set W of processes with which it established a link on line 6 but not on line 18 (see line 27). The set W contains every neighbor of p that eventually signals a group of processes that includes p on line 26, hence it is the subset of p ’s neighbors that may rely on p for sharing work. Note that W may also contain neighbors that signal p from line 21, but in that case no work sharing is required (in particular, such a neighbor does not signal p again on line 26). Process p receives signals from its neighbors and performs its share of work in the corresponding notification chains by executing lines 28–31. Finally, p returns the leader’s ID on line 32.

The correctness of `NameDecide` is established by the following theorem, whose proof appears in the full paper. Here we provide a very short proof sketch.

THEOREM 2.1. *NameDecide-DSM is a correct name consensus algorithm and its RMR complexity is $O(1)$.*

Proof sketch. From the correctness of the leader election object \hat{L} used by `NameDecide-DSM`, the sets S_p of neighbors, for $p \in \mathcal{P}$, constructed by the calls on line 1, induce an (undirected) connected graph between processes in \mathcal{P} . Termination follows from the fact that the leader elected by \hat{L} .`LeaderElect` starts a notification mechanism whereby it communicates its identity to all active processes. Since \hat{L} is correctly accessed once by each active process, and since each process eventually returns the ID of the winner of \hat{L} , the return value of `NameDecide-DSM` is correct. $O(1)$ RMR complexity follows from the fact that $|S_p|$, for any process p , is bounded from above by the number of RMRs performed by p while executing \hat{L} .`LeaderElect`.

3. COMPARE-AND-SWAP

In this section we show how to simulate a compare-and-swap (CAS) object using read/write registers and name consensus objects. This is a strong variant of CAS in the sense that it returns the previous value of the object instead of merely a Boolean success indicator. The simulation applies to both the CC and DSM model, given an appropriate choice of base objects.

Our implementation of CAS uses a variant of the helper functions `signal` and `wait` used in `NameDecide-DSM`. The new functions assume that there is exactly one *signalling process* p such that only p calls `s-signal`, and any process calling `s-wait` knows p ’s name and passes it as an argument. Furthermore, the signaller can pass an argument to `s-signal` that is received by the notified process as the return value of `s-wait`.

Implementations of `s-signal` and `s-wait` for the CC and DSM models are shown below. In the CC model, we use a Boolean atomic register *flag*, initially `false`, as a spin variable for all waiting processes, and an uninitialized register *value* to store the signaler’s argument. In the DSM model, the implementation is more complex; it relies on an instance of `LinkRequest/LinkReceive` where the signalling process p is the process that calls `LinkReceive`, as well as an instance of `signal/wait`. Formal correctness proofs for these functions are provided in the full paper.

Function `s-signal-DSM(val)`

```
1 write value ← val
2 P ← LinkReceive
3 signal(P)
```

Function `s-wait-DSM(p)`

```
1 if LinkRequest( $p$ ) then wait({ $p$ })
2 return read(value)
```

Function `s-signal-CC(val)`

```
1 write value ← val
2 write flag ← true
```

Function `s-wait-CC(p)`

```
1 await flag = true
2 return read(value)
```

We present an implementation of a CAS object τ that uses the `s-signal` and `s-wait` functions. The set of base objects includes a pointer D to a data structure we refer to as a *page*. Initially, D points to a page representing the initial value. A page consists of the following objects: an atomic register V storing τ ’s value, a constant-RMR name consensus object NC (e.g., implemented as per Section 2), and an instance of `s-signal/s-wait`. In the pseudo-code below we use the operator \triangleright to denote the pointer dereference operation (e.g., $d \triangleright V$ where d is an address read from D).

The set of execution histories corresponds to the access procedure for the CAS operation, presented below. The algorithm relies on a helper function `getNewPage` that returns the address of a page where the corresponding instances of V , NC , and `s-signal/s-wait` are in their initial states. The use of pages is motivated by the need to recycle memory, which allows us to obtain a bounded-memory implementation of CAS. For lack of space, we defer details of the implementation of `getNewPage` and, in particular, the memory recycling scheme, to the full paper. For now, we may think of `getNewPage` as simply returning the address of a previously-unused page, where the corresponding objects and function instances have already been initialized.

At a high level, the CAS algorithm works as follows. The current state of τ is stored in $D \triangleright V$. Each execution of CAS first reads the current state from the page pointed to by D , and compares that state against its *cmp* argument. A CAS operation execution that “fails” on line 3 because *cmp* differs from the current state, simply returns the previous state on line 14, and is linearized to the point where it read D on line 1. Executions where the *cmp* argument matches the current state on line 3 are partitioned into groups based on the page address they read on line 1. Executions in each group perform additional synchronization to determine their

Function CAS(*cmp*, *new*)

Input: *cmp* – comparison value, *new* – value to be swapped in

Output: previous value of object

```
1  $d \leftarrow \text{read}(D)$ 
2  $old \leftarrow \text{read}(d \triangleright V)$ 
3 if  $old = cmp \wedge cmp \neq new$  then
4    $winner \leftarrow d \triangleright NC.\text{NameDecide}$ 
5   if  $winner = \text{PID}$  then
6      $d' \leftarrow \text{getNewPage}$ 
7     write  $d' \triangleright V \leftarrow new$ 
8     write  $D \leftarrow d'$ 
9      $d \triangleright s\text{-signal}(new)$ 
10  else
11     $old \leftarrow d \triangleright s\text{-wait}(winner)$ 
12  end
13 end
14 return  $old$ 
```

relative linearization order. The processes of each group compete using the call to `NameDecide` on line 4 on the name consensus object of the page read on line 1. The winner is linearized first among the processes of this group, and succeeds in changing the state of τ . That is, the winner allocates a new page on line 6, writes the new state to this page on line 7, and writes the address of the new page to D on line 8. The winner’s operation execution is linearized at the latter step. Other processes in the winner’s group wait on line 11 for the winner to complete line 9 (hence line 8 as well). The pending operation executions of these processes are linearized to a point just after the winner’s execution of line 8, at which time the state of τ changed. In other words, these executions become unsuccessful since their *cmp* arguments do not match the new state written by the winner. The return value of `s-wait` allows these operation executions to discover this new state on line 11, and that value is returned on line 14.

The correctness of the above implementation is formally stated by the following theorem, whose proof is provided in the full paper.

THEOREM 3.1. *The implementation defined above of a CAS object τ is linearizable. Furthermore, each operation on τ costs $O(1)$ RMRs given that each operation on a base object costs $O(1)$ RMRs.*

Locally Accessible Implementations. Consider an algorithm \mathcal{A} that uses a CAS object C . Our goal in this section is to show that one can “plug” our simulation of C into \mathcal{A} with only a constant blowup in \mathcal{A} ’s RMR complexity. To achieve this, our simulation of CAS must meet the following requirements, in addition to a constant worst-case RMR complexity: (1) in the DSM model, a designated process (to which C is local) can execute *any* CAS operation without incurring any RMRs; and (2) in the CC model, certain CAS operations on an “in-cache” CAS object incur no RMRs.

Before we formalize the latter property, let us review the RMR cost of operations in the CC model under the two families of coherence protocols that we consider in this paper: *write-through* and *write-back* [12]. (Each of these protocols comes in two versions: with cache invalidation and with cache update upon write. In this paper we consider only the invalidation versions, as they are the more common ones in practice.)

In a write-through protocol, to read a variable X a process p must have a (valid) cached copy of X . If it does, p reads that copy without causing an RMR; otherwise, p causes an RMR that creates a cached copy of X . To write X , p causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of X and writes X to memory.

In a write-back protocol, each cached copy is held in either “shared” or “exclusive” mode. To read a variable X , a process p must hold a cached copy of X in either mode. If it does, p reads that copy without causing an RMR; otherwise, p causes an RMR that (a) creates a cached copy of X held in shared mode, and (b) eliminates any copy of X held in exclusive mode, typically by downgrading the status to shared and, if the exclusive copy was modified, writing X back to memory. To write X , p must have a cached copy of X held in exclusive mode. If it does, p writes that copy without causing RMRs; otherwise, p causes an RMR that (a) creates a cached copy of X held in exclusive mode, and (b) invalidates all other cached copies of X (regardless of the mode in which they are held) and writes X back to memory.

In both protocols, a read of variable X by process p causes an RMR if and only if p has no (valid) cached copy of X . The protocols differ in the RMRs caused by writes: in write-through, every write causes an RMR; in write-back, a write of X by p causes an RMR if and only if X was not previously written by p or X was accessed (read or written) by another process since it was last written by p . (We ignore RMRs due to constraints on cache size and associativity.)

Consider now a CC multiprocessor that supports in hardware other operations, such as CAS, in addition to reads and writes. What is the RMR cost of such operations? Operations that are “read-like”, in that they do not change the state of the variable to which they are applied (e.g., unsuccessful CAS operations), are treated like reads in this regard, while other operations are treated like writes. Note that this way of counting RMRs is conservative because, in some hardware implementations of CAS, even an unsuccessful CAS operation is treated like a write, causing cache invalidations as dictated by the coherence protocol.

Consider a (linearizable) implementation I of an object using only read and write operations. For the sake of concreteness, suppose the object is a CAS, though what follows applies to objects of any type. As stated earlier, we would like the implementation I to have the following *RMR-preservation property* in the two families of CC multiprocessors under consideration: Suppose we take *any* algorithm that uses CAS objects, as well as read and write operations. If we replace the hardware CAS objects that the algorithm uses by I , then for any execution of the resulting algorithm (that uses only read and write operations), the number of RMRs when running on a particular CC multiprocessor is within a constant factor of the number of RMRs performed by the original algorithm (that uses CAS operations, in addition to reads and writes) on the same multiprocessor.

We now state conditions on the implementation I that are sufficient for it to satisfy this RMR preservation property for the two coherence protocols under consideration. For the write-back protocol, the following condition suffices:

(R) *For any execution of I and any process p , there is a linearization σ of the operations in that execution such that if σ' is a contiguous subsequence of σ consisting only of read-like operations, then the operations in σ' issued by p cause only a constant number of RMRs.*

For the write-through protocol, the implementation I is RMR-preserving if it satisfies condition (R), and

(W) For any execution of I and any process p , there is a linearization σ of the operations in that execution such that if σ' is a contiguous subsequence of σ consisting only of operations issued by p , then the operations in σ' cause only a constant number of RMRs.

We now explain how to modify our implementation of CAS to ensure that it satisfies these properties. When running on a CC machine with write-through caching, no modifications are required. Consider a sequence of failed CAS operations that is contiguous in the linearization order, and a process p . It is easy to see that all operations by p , except possibly the first and last, access the same page, read the same variables (i.e., D and $D \triangleright V$), and write no variables. Consequently, at most the first two operations and the last in the sequence perform any RMRs at all, and each of these performs at most a constant number of RMRs.

Our CAS algorithm must be modified for CC machines with write-back caching, otherwise a process performs RMRs each time it executes a successful CAS operation (because it must execute name consensus and initialize a new page). To remedy this problem, we allow a process p to reuse the current page upon a successful CAS operation under the following conditions: the page was allocated by p , and no other process has accessed the page previously in the linearization order (other than possibly reading its address). To ensure the latter condition, we introduce a reader-writer lock on each page that must be acquired each time the page is accessed. Before acquiring the lock, p checks whether the page was allocated by it. If this is the case, and if no other process has previously acquired the lock in read mode, then p acquires the lock in write mode. Otherwise, p acquires the lock in read mode. In the former case, p is able to perform a successful CAS by reusing the current page. In the latter case, p follows the original execution path, possibly participating in name consensus and allocating a new page. It is easy to construct the reader-writer lock so that the overhead introduced is $O(1)$ RMRs per process per page.

Unlike the CC implementation, our CAS algorithm for the DSM model must be modified considerably to support local operations. Two key modifications are needed in order to make a CAS object local to a designated process p . First, the fields of a page, namely the register V and the name consensus object, must be local to p (for name consensus we can use a technique similar to the one presented in [8] for leader election). Second, the signalling mechanism called on line 9 and line 11 must be modified so that p does not write remote spin variables. Specifically, when p performs a successful CAS operation execution, no process q may wait on line 11 for p to update D on line 8. Instead, q “helps” p complete its pending operation and only then completes its own operation. The helping mechanism requires that we replace the shared variable D with a pair of variables, D_p and D_{other} , where D_p is written only by p , and D_{other} may be written by any process other than p . Instead of reading D on line 1, a process now reads both D_p and D_{other} , and adopts the most recently written value. To determine which value is more recent, we can tag each page with a sequence number that increases each time the CAS object changes states. Thus, we arrive at the following result.

THEOREM 3.2. Any CC or DSM shared memory algorithm using read and write operations and compare-and-swap, can be simulated by an algorithm that uses only read and write operations, with at most a constant blowup in the RMR complexity.

4. GENERALIZATION TO COMPARISON PRIMITIVES AND LL/SC

The results presented in Section 3 can be generalized to additional operations. Anderson and Kim define a class of primitives that they name *comparison primitives* [1]. This class contains operations such as CAS and TAS. The pseudo-code of a generic comparison primitive, which must be executed atomically, is shown below.

Function `compare_and_fg`(v, cmp, new)

```

1  $old \leftarrow v$ 
2 if  $v = cmp$  then  $v \leftarrow f(cmp, new)$ 
3 return  $g(old, cmp, new)$ 

```

Any such primitive can be implemented using the CAS algorithm of Section 3 as a black box to update v , namely by replacing lines 1–2 with the following statement:

$old \leftarrow v.CAS(cmp, f(cmp, new))$

Linearizability and liveness are obvious since the pseudo-code modified this way contains only one shared memory operation (i.e., the CAS on v).

It is also possible to implement the pair of primitives *load-linked/store-conditional* (LL/SC) using our CAS algorithm, namely via the constant-time wait-free construction of Moir [11]. Thus, we get the following theorem, whose formal proof we defer to the extended version of the paper.

THEOREM 4.1. Any CC or DSM shared memory algorithm using read and write operations, comparison primitives, and LL/SC, can be simulated by an algorithm that uses only read and write operations, with at most a constant blowup in the RMR complexity.

5. BOUNDED COUNTERS

We have shown for several types of important primitives that they have constant-RMR implementations from reads and writes. It is natural to ask, for what kind of problems are such constant-RMR solutions *not* possible. Mutual exclusion is one example of a problem which cannot be realized with a constant number of RMRs using reads and writes, but it is quite strong. In this section we explore the limitations of constant-RMR implementable objects more closely.

A *k*-bounded counter is an object which supports the operations **Increase** and **Reset**. The object stores a value in $\{0, \dots, k\}$. The operation **Increase** increments the value of the object by one, unless it already has value k . The operation **Reset** sets the value of the object back to 0. Both operations return the previous value of the object. A *blind* *k*-bounded counter supports the operation **BlindReset** instead of **Reset**, which also resets the object but does not return any value.

Using the techniques devised in this paper, it is easy to implement blind *k*-bounded counters in such a way that any **Increase**-operation needs only $O(k)$ RMRs, and a **BlindReset**-operation needs only $O(1)$ RMRs. However, for (non-blind) two-bounded counters, no constant-RMR implementation exists for the DSM-model.

THEOREM 5.1. *In the DSM-model, for any linearizable implementation of a two-bounded counter using read and write operations, comparison primitives, and LL/SC, there is an execution where N processes concurrently access the counter, each applying at most two operations sequentially, such that the total number of RMRs performed in the execution is $\Omega(N \cdot \log \log N)$.*

One nice consequence of our results from the previous sections is, that it suffices to prove this theorem only for implementations with reads and writes, and then we get the result for the stronger primitives for free. The proof is based on very similar ideas as the lower bound for adaptive mutual exclusion by Kim and Anderson [10]. We provide some intuition for the proof at the end of this section.

Note that our lower bound implies an $\Omega(\log \log N)$ lower bound for the amortized (per process) RMR cost of mutual exclusion. Thus, our lower bound beats the previously best amortized lower bound of $\Omega(\log \log N / \log \log \log N)$ RMRs by Cypher [6] (but note that Cypher’s bound also holds for the CC-model).

One may also consider counters with an additional operation **ResetOnOne**, which resets the counter only if its value is one, and returns the previous value. Anderson and Kim have devised an algorithm which uses counters supporting the operations **Increase**, **BlindReset**, and **ResetOnOne** together with read and write registers in order to solve mutual exclusion with $O(\log N / \log \log N)$ RMRs [2]. Our lower bound for two-bounded counters implies that this approach cannot lead to $O(\log N / \log \log N)$ RMR implementations of mutual exclusion using only reads, writes, comparison and LL/SC-primitives. In our proof of Theorem 5.1, we construct the execution with high RMR cost for an algorithm, where each process first calls **Increase**, and only if this operation returns zero, then calls **Reset**. In such an algorithm, the **Reset** operation can be replaced by a **ResetOnOne**-operation, which – if not successful – is followed by a **BlindReset**. Hence, we obtain the same lower bound as in Theorem 5.1 for counters supporting **Increase**, **BlindReset**, and **ResetOnOne**.

Sketch of the Proof of Theorem 5.1. For the remainder of this section, we consider the DSM model with a set \mathcal{P} of N processes. We assume that there are linearizable implementations for the operations **Increase** and **Reset** using only read and write operations. We use these in an algorithm **SimpleAlgo** which works as follows (see the pseudo-code below): A process calling **SimpleAlgo** first tries to increase the counter by executing **Increase** and prints out the return value of that function call. (We can assume w.l.o.g. that each process realizes the operation **Print**(v) by writing value v to some designated local register.) If the counter is increased from 1 to 2, or not increased at all, then the process is done. Otherwise the process prints “lock”, and then resets the counter, and prints the value of the counter at the moment just before the reset occurred.

Algorithm: SimpleAlgo

```

1  $r \leftarrow \text{Increase}$ 
2 Print(PID, increase,  $r$ )
3 if  $r = 0$  then
4   | Print(PID, lock)
5   |  $r' \leftarrow \text{Reset}$ 
6   | Print(PID, reset,  $r$ )
7 end

```

This algorithm realizes something very similar to mutual exclusion: If one process manages to increase the counter from 0 to 1, it enters a *critical section*: Until it resets the counter, no other process can print “lock”.

In the full version of the paper we provide a proof of the following theorem, from which Theorem 5.1 follows easily.

THEOREM 5.2. *Let $n \leq N$ and $\mathcal{P}' \subseteq \mathcal{P}$ be a set of size n . Under the assumptions described above, there is an execution, where $O(\log \log n)$ processes in \mathcal{P}' concurrently execute **SimpleAlgo** exactly once and the total number of RMRs during that execution is $\Omega((\log \log n)^2)$.*

We now sketch the high-level idea of the proof of this theorem. We construct an execution in which a subset of the processes in \mathcal{P} participate and execute operations in rounds. During one round, each participating process that has not finished executes exactly one RMR. We construct executions E_1, E_2, \dots, E_K inductively (for some appropriate value of K), where E_k , $1 \leq k \leq K$, is an execution over k rounds. However, with increasing k we decrease the number of processes participating in E_k . Let P_k denote the set of processes participating in execution E_k . We also maintain some sets of “special processes”, $Z_1, \dots, Z_k \subseteq P_k$. Each set Z_i is either the empty set, or it contains exactly one process p_i . The executions E_k are constructed essentially in the same way as the execution in [10].

We say that process p “becomes aware of process q ”, if p reads a register which was last written by q . The most important property of the executions E_k and the sets Z_1, \dots, Z_k can be summarized as follows:

(I) *During execution E_k , if in round i process $p \in P_k$ becomes aware of process $q \in P_k$, then q has already finished its execution, and $q \in Z_1 \cup \dots \cup Z_i$.*

This property leads to another observation:

(II) *All processes that finish the algorithm during execution E_k increase the counter from 0 to 1, enter the “critical section”, and reset the counter from 1 to 0.*

This is proved by induction on the number of processes that have finished the algorithm. The idea of the induction is the following: The “first process” finishing the algorithm does not become aware of any other process, so it has to behave as claimed (it doesn’t know whether it runs solo). The “next” process finishing the algorithm, p , must also increase the counter from 0 to 1, because it is not aware of any process which has not already reset the counter from 1 to 0. Thus, p also prints “lock”. Finally, p ’s **Reset**-operation must return 1, because p is not aware of any process that might have increased the counter from 1 to 2.

Using this observation, we can conclude:

(III) *There are no two processes $p, q \in P_k - (Z_1 \cup \dots \cup Z_k)$ that finish the algorithm during execution E_k .*

Assume, by way of contradiction, that such processes p, q do exist. Then both of them have to print “lock”, and w.l.o.g. p prints “lock” not later than q . Since no process in P_k becomes aware of p during execution E_k , we can halt p right after it printed “lock” without changing the behaviour of any other process. But then q eventually prints “lock”, too, and resets the counter before p can reset the counter.

This means that p and q are in the critical section at the same time—a contradiction.

Now let $Z = Z_1 \cup \dots \cup Z_k$ and $Z \subseteq S \subseteq P_k$. The execution $E_k|_S$ is the execution where only the processes in S participate and these processes are scheduled relative to each other exactly the same way as in E_k . Since during E_k , any process $p \in S$ may only become aware of the processes in Z , and all processes in Z also participate in $E_k|_S$, for any process $p \in S$ the executions E_k and $E_k|_S$ are indistinguishable. Using this idea, we obtain another important property of the executions E_k :

(IV) For $Z \subseteq S \subseteq P_k$, the processes in S behave in execution $E_k|_S$ exactly in the same way as they behave in execution E_k .

By (III), we conclude that at most one of the processes in $P_k - Z$ finishes the algorithm during execution E_k . In fact, by (IV) we know that for any $Z \subseteq S \subseteq P_k$, at most one of the processes in $S - Z$ finishes the algorithm in execution $E_k|_S$. Recall that $|Z| = |Z_1| + \dots + |Z_k| \leq k$. Since E_k has k rounds and each process executes one RMR per round, at least $|S - Z| - 1 \geq |S| - k - 1$ processes perform k RMRs, each. So if $|P_k| \geq 2k$, then we can find a set S of size exactly $2k$ such that the total number of RMRs performed in execution $E_k|_S$ is $\Omega(k^2)$. Our construction of the executions guarantees $|P_k| = \Omega(n^{2^{-k}})$, so $|P_k| \geq 2k$ for some $k = \Omega(\log \log n)$, and the lower bound follows.

6. DISCUSSION

In this paper, we have presented linearizable constant-RMR implementations of consensus, comparison primitives (specifically CAS), and LL/SC, from read and write operations for both the DSM and CC shared memory models. To the best of our knowledge, our CAS algorithm is the first constant-RMR implementation of a long-lived read-modify-write object from reads and writes. Our results imply that comparison primitives are no stronger than reads and writes in terms of RMR complexity for blocking synchronization. We have also shown that any implementation of the resettable bounded counter defined by Anderson and Kim in [2] incurs a worst-case average complexity of $\Omega(\log \log n)$ RMRs.

Acknowledgement. We thank Robert Danek for his feedback on an earlier draft of this paper. We are also grateful to the anonymous referees for their insightful comments.

7. 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. Local-spin mutual exclusion using fetch-and-phi primitives. In *Proc. of 23rd ICDCS*, pages 538–547, 2003.
- [3] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.
- [4] 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] T. Craig. Queuing spin lock algorithms to support timing predictability. In *Proc. of 14th RTSS*, pages 148–156, 1993.
- [6] R. Cypher. The communication requirements of mutual exclusion. In *Proc. of 7th SPAA*, pages 147–156, 1995.
- [7] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.
- [8] W. Golab, D. Hendler, and P. Woelfel. An $O(1)$ RMRs leader election algorithm. In *Proc. of 25th PODC*, pages 238–247, 2006, Denver, CO, USA.
- [9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.
- [10] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proc. 15th DISC*, pages 1–15, London, UK, 2001.
- [11] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of 16th PODC*, pages 219–228, New York, NY, USA, 1997.
- [12] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, San Francisco, California, 1994.
- [13] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

APPENDIX

In this Appendix we provide the code of the `LinkRequest` and `LinkReceive` functions used by the DSM name consensus algorithm. The reader is referred to [8] for a more detailed description and correctness proofs.

The handshaking protocol to establish communication links with process p uses the array $A_p[]$ and the variable B_p . Processes p and q use B_p and entry q of A_p to agree on whether or not q succeeds in establishing a link with p . The output of this protocol is recorded in the LINK_p array: entry q of LINK_p is set if a link from p to q was established and reset otherwise.

Function LinkRequest(p)

Input: Process ID p
Output: a value in $\{0, 1\}$ indicating link establishment failure or success, respectively

```

1 write  $A_p[\text{PID}] \leftarrow 1$ 
2  $s \leftarrow \text{read}(B_p)$ 
3 if  $s = \perp$  then  $link \leftarrow 1$  else  $link \leftarrow 0$ 
4 write  $\text{LINK}_p[\text{PID}] \leftarrow link$ 
5 return  $link$ 

```

Function LinkReceive

Output: set of processes to which link was established

```

1  $B \leftarrow 1$ 
2 forall process IDs  $q \neq \text{PID}$  do
3   if  $A[q] = \perp$  then  $\text{LINK}[q] \leftarrow 0$ 
4   else await  $\text{LINK}[q] \neq \perp$ 
5 end
6 return  $\{q \mid \text{LINK}[q] = 1\}$ 

```
