# Timing-based Mutual Exclusion with Local Spinning⋆
## (Extended Abstract)

Yong-Jik Kim and James H. Anderson

Department of Computer Science
University of North Carolina at Chapel Hill
Email: {kimy,anderson}@cs.unc.edu

**Abstract.** We consider the time complexity of shared-memory mutual exclusion algorithms based on reads, writes, and comparison primitives under the remote-memory-reference (RMR) time measure. For asynchronous systems, a lower bound of $\Omega(\log N/ \log \log N)$ RMRs per critical-section entry has been established in previous work, where $N$ is the number of processes. In this paper, we show that lower RMR time complexity is attainable in semi-synchronous systems in which processes may execute *delay* statements. When assessing the time complexity of delay-based algorithms, the question of whether delays should be counted arises. We consider both possibilities. Also of relevance is whether delay durations are upper-bounded. (They are lower-bounded by definition.) Again, we consider both possibilities. For each of these possibilities, we present an algorithm with either $\Theta(1)$ or $\Theta(\log \log N)$ time complexity. For the cases in which a $\Theta(\log \log N)$ algorithm is given, we establish matching $\Omega(\log \log N)$ lower bounds.

## 1 Introduction

Recent work on shared-memory mutual exclusion has focused on the design of algorithms that minimize interconnect contention through the use of *local spinning*. In local-spin algorithms, all busy waiting is by means of read-only loops in which one or more "spin variables" are repeatedly tested. Such variables must be either locally cacheable or stored in a local memory module that can be accessed without an interconnection network traversal. The former is possible on cache-coherent (CC) machines, while the latter is possible on distributed shared-memory (DSM) machines.

In this paper, several results concerning the time complexity of local-spin mutual exclusion algorithms are given. Time complexity is defined herein using the *remote-memory-reference* (*RMR*) *measure* [8]. Under this measure, an algorithm's time complexity is defined as the total number of RMRs required in the worst case by one process to enter and then exit its critical section. An algorithm may have different RMR time complexities on CC and DSM machines, because variable locality is dynamically determined on CC machines and statically on DSM machines (see [7]).

In this paper, we consider mutual exclusion algorithms based on reads, writes, and comparison primitives such as *test-and-set* and *compare-and-swap* (CAS). A *comparison primitive* is an atomic operation on a shared variable $v$ that is expressible using the following pseudo-code.

---

```
compare_and_fg(v, old, new)
    temp := v;
    if v = old then v := f(old, new) fi;
    return g(temp, old, new)
```

For example, CAS can be speci£ed by de£ning $f(old, \ new) = new$ and $g(temp, old, \ new) = old$.

In earlier work, Cypher [11] established a time-complexity lower bound of $\Omega(\log \log N / \log \log \log N)$ RMRs for any asynchronous $N$-process mutual exclusion algorithm based on reads, writes, and comparison primitives. In recent work [5], we presented for this class of algorithms a substantially improved lower bound of $\Omega(\log N / \log \log N)$ RMRs, which is within a factor of $\Theta(\log \log N)$ of being optimal, since algorithms based only on reads and writes with $\Theta(\log N)$ RMR time complexity are known [22].[1] The proofs of these lower bounds use the ability to "stall" some processes for arbitrarily long durations, and hence are not applicable to *semi-synchronous systems*, in which the time required to execute a statement is upper-bounded.

A number of interesting "timing-based" mutual exclusion algorithms have been devised in recent years in which such bounds are exploited, and processes have the ability to delay their execution [2, 4, 16, 17]. Such algorithms are the focus of this paper. We exclusively consider the *known-delay* model [4, 16, 17], in which there is a known upper bound, denoted $\Delta$, on the time required to read or write a shared variable.[2] For simplicity, all process delays are assumed to be implemented via the statement $delay(\Delta)$. (Longer delays can be obtained by concatenating such statements; we will use $delay(c \cdot \Delta)$ as a shorthand for $c$ such statements in sequence.)

In prior work on timing-based algorithms, the development of algorithms that are fast in the *absence* of contention has been the main focus. In fact, to the best of our knowledge, all timing-based algorithms previously proposed use non-local busy-waiting. Hence, these algorithms have unbounded RMR time complexity under contention.

**Contributions.** In this paper, we present time-complexity bounds for timing-based algorithms under the known-delay model in which all busy-waiting is by local spinning. (Our results are summarized in Table 1, which is explained below.) Under this model, the class of algorithms considered in this paper can be restricted somewhat with no loss of generality. In particular, comparison primitives can be implemented in constant time from reads and writes by using delays [3, 19]. Thus, it suf£ces to consider only timing-based algorithms based on reads and writes. In the rest of the paper, all claims are assumed to apply to this class of algorithms, unless noted otherwise.

When assessing the RMR time complexity of timing-based algorithms, the question of whether delays should be counted arises. Given that $\Delta$ is necessarily at least the duration of one RMR, it may make sense to count delays. Accordingly, we de£ne the *RMR-$\Delta$*

---

[1] In contrast, several $\Theta(1)$ algorithms are known that are based on noncomparison primitives (*e.g.*, [6, 9, 12, 18]). We do not consider such algorithms in this paper.

[2] Equivalently, statement executions can be considered to take place instantaneously (*i.e.*, atomically), with consecutive statement executions of the same process occurring at most $\Delta$ time units apart. We adopt this model in our lower bound proof. The known-delay model differs from the *unknown-delay* model [2], wherein the upper bound $\Delta$ is unknown *a priori*, and hence, cannot be used directly in an algorithm.

| Arch. | RMR Time Complexity | | RMR-$\Delta$ Time Complexity |
| --- | --- | --- | --- |
| | Delays Bounded | Delays Unbounded | Delays Bounded/Unbounded |
| DSM | $\Theta(1)$ {ALG. DSM} | $\Theta(1)$ {ALG. DSM} | $\Theta(1)$ {ALG. DSM} |
| CC | $\Theta(1)$ {ALG. CC} | $\Theta(\log \log N)$ {ALG. T, Thm. 3} | $\Theta(\log \log N)$ {ALG. T, Thm. 2} |

**Table 1.** Summary of results. Each entry gives a time-complexity £gure that is shown to be optimal.

*time complexity* of an algorithm to be the total number of RMRs and $delay(\Delta)$ statements required in the worst case by one process to enter and then exit its critical section. (Note that this measure includes the total delay duration as well as the number of delay statements, since $\Delta$ is £xed for a given system.)

On the other hand, one might argue that delays should be ignored when assessing time complexity, just like local memory references. For completeness, we consider this possibility as well by also considering the standard RMR measure (which ignores delays). One limitation of the RMR measure is that it allows algorithms with long delays to be categorized as having low time complexity. For this reason, we view the RMR-$\Delta$ measure as the better choice.

As we shall see, the exact semantics assumed of the statement $delay(\Delta)$ is of relevance as well. It is reasonable to assume that a process is delayed by *at least* $\Delta$ time units when invoking this statement. However, it is not clear whether a speci£c upper bound on the delay duration should be assumed. For completeness, we once again consider both possibilities.

Our results are summarized in Table 1. The headings "Delays Bounded/Unbounded" indicate whether delay durations are assumed to be upper bounded. Each table entry gives a time-complexity £gure that is shown to be optimal by giving an algorithm, and for the $\Theta(\log \log N)$ entries, a lower bound. Due to space constraints, ALGORITHMS DSM and CC, as well as one of our lower bounds, are presented only in the full version of this paper [14]. The main conclusion to be drawn from these results is the following: *in semi-synchronous systems in which* delay *statements are supported, substantially smaller RMR time complexity is possible than in asynchronous systems when devising mutual exclusion algorithms using reads, writes, and comparison primitives, regardless of how one resolves the issues of whether to count delays and how to de£ne the semantics of the* delay *statement*.

In the following sections, we present ALGORITHM T, a $\Theta(\log \log N)$ algorithm, and a matching $\Omega(\log \log N)$ lower bound for CC machines under the RMR-$\Delta$ measure.

## 2   A $\Theta(\log \log N)$ Algorithm

In this section, we describe ALGORITHM T (for "tree"), illustrated in Fig. 3, in which each process executes $\Theta(\log \log N)$ RMRs and $\Theta(\log \log N)$ delay statements in order to enter and then exit its critical section. Upper bounds on delays are not required.

ALGORITHM T is constructed by combining smaller instances of a mutual exclusion algorithm in a binary arbitration tree. A similar approach has been used in algorithms in which each tree node represents an instance of a two-process mutual exclusion algorithm [13, 22]. If each node takes $\Theta(1)$ time, then $\Theta(\log N)$ time is required for a process to enter (and then exit) its critical section.
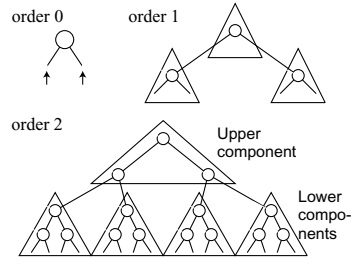
**Fig. 1.** Structure of arbitration trees used in ALGORITHM T. A tree of order $k > 0$ has an upper component and $2^{2^{k-1}}$ lower components, each of order $k - 1$.
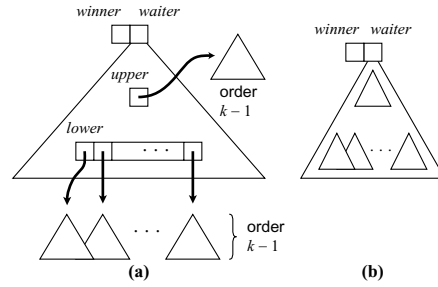
**Fig. 2.** Structure of an arbitration tree of *TreeType*, of order $k$. **(a)** A "verbose" depiction, showing dynamic links for its components. **(b)** A simpli£ed version.

In order to obtain a faster algorithm, we give the tree an additional structure, as follows. For the sake of simplicity, we assume that $N = 2^{2^K}$ holds for some integer $K > 0$. (Otherwise, we add "dummy processes" to the nearest such number. Since $\log \log 2^{2^K} = K$, such padding increases the algorithm's time complexity by only a constant factor.) We say that a binary arbitration tree has *order $k$* if it has $2^k$ (non-leaf) levels and $2^{2^k}$ leaves, as shown in Fig. 1. A tree of order zero is a two-process mutual exclusion algorithm. A tree $T$ of order $k > 0$ is divided into the top $2^{k-1}$ levels and the bottom $2^{k-1}$ levels: the top levels constitute a single tree of order $k - 1$, and the bottom levels, $2^{2^{k-1}}$ distinct trees of order $k - 1$. (This structure is rather similar to the van Emde Boas tree [21], which implements a $\Theta(\log \log u)$-time search structure over a £xed set of integers in the range $1..u$.) We call these subtrees $T$'s *components*. Thus, $T$ consists of a single *upper component* and $2^{2^{k-1}}$ *lower components*, where the root node of each lower component corresponds to a leaf node of the upper component. These components are linked into $T$ dynamically by pointers, so a process can exchange a particular component $S$ with another tree $S'$ (of order $k - 1$) in $\Theta(1)$ time.

We also say that tree $S$ is a *constituent* of tree $T$ if either $S$ is $T$ or $S$ is a constituent of another component of $T$. Associated with each tree $T$ is a £eld called *winner*, which is accessed by CAS operations. (As noted earlier, CAS can be implemented in $\Theta(1)$ time using delays [3, 19].) A process $p$ attempts to establish $T.winner = p$ by invoking CAS, in which case it is said to have *acquired $T$*. The structure of an arbitration tree explained thus far is depicted in Fig. 2. (The *waiter* £eld is explained later.)

**Arbitration tree and waiting queue.** We start with a high-level overview of ALGO-RITHM T. A tree $T_0$ of order $K$ and $N$ leaves is used, in which each process is statically assigned to a leaf node. The algorithm is constructed by recursively combining instances of a mutual exclusion algorithm for each component of $T_0$. The process that wins the outermost instance of the algorithm (*i.e.*, that associated with $T_0$) enters its critical section.

Note that, for each process $p$, its path from its leaf up to the root in $T_0$ is contained in two components, namely, some lower component $L_i$ and the upper component $U$. To enter its critical section, $p$ attempts to acquire both components on this path (by invoking CAS on $L_i.winner$, and then on $U.winner$). If $p$ acquires both components, then it may enter its critical section, by invoking *ExecuteCS*. As explained shortly, $p$ may also be "promoted" to its critical section after failing to acquire either tree. (In that case, $p$ may

```
TreeType = record
    order: 0..K;
    upper: pointer to TreeType;
    lower: array[0..(2^{2^{order-1}} − 1)] of
       pointer to TreeType;
    winner, waiter: (⊥, 0..N − 1)
end

shared variables
    T_0: (a tree with order K);
    Spin: array[0..N − 1] of boolean;
    Promoted: (⊥, 0..N − 1) initially ⊥;
    WaitingQueue: queue of {0..N − 1}
       initially empty;

process p ::    /* 0 ≤ p < N */

while true do
0:   Noncritical Section;

1:   Spin[p] := false;
2:   AccessTree(&T_0, p);
3:   TryToPromote();
4:   Signal()       /* open the barrier */
od

procedure TryToPromote()
/* promote a waiting process (if any) */
5:   q := Promoted;
6:   if (q = p) ∨ (q = ⊥) then
7:      next := Dequeue(WaitingQueue);
8:      Promoted := next;
9:      if next ≠ ⊥ then
10:        Spin[next] := true fi
     fi

procedure ExecuteCS(side: 0, 1)
11: if side = 1 then await Spin[p] fi;
12: Entry_2(side);
13: Critical Section;
14: Wait();       /* wait at the barrier */
15: Exit_2(side)
```

```
procedure AccessTree(
    ptrT: pointer to TreeType, pos: 0..N − 1)
16: k := ptrT -> order;
17: if k = 0 then               /* base case */
18:    ptrT -> waiter := p;
19:    ExecuteCS(1);
20:    return
    fi;

21: indx := ⌊pos/2^{2^{k-1}}⌋;
22: ptrL := ptrT -> lower[indx];
23: if CAS(ptrL -> winner, ⊥, p) ≠ ⊥ then
24:    AccessTree(ptrL, pos mod 2^{2^{k-1}});
            /* recurse into the lower component */
25:    return
    fi;

26: ptrU := ptrT -> upper;
27: if CAS(ptrU -> winner, ⊥, p) ≠ ⊥ then
28:    AccessTree(ptrU, indx)
            /* recurse into the upper component */
    else
29:    if ptrT = &T_0 then
30:       ExecuteCS(0)
       else
31:       ptrT -> waiter := p;
32:       ExecuteCS(1)
       fi
    /* update the upper component */
33:    ptrC := GetCleanTree(k − 1);
34:    ptrT -> upper := ptrC;
35:    delay(Δ_0);
36:    Enqueue(WaitingQueue, ptrU -> waiter)
    fi;

    /* update the lower component */
37: ptrC := GetCleanTree(k − 1);
38: ptrT -> lower[indx] := ptrC;
39: delay(Δ_0);
40: Enqueue(WaitingQueue, ptrL -> waiter)
```

**Fig. 3.** ALGORITHM T, unbounded space version. (Each private variable used in *AccessTree* is assumed to be on the call stack.)

have acquired only $L_i$, or neither $L_i$ nor $U$.) To arbitrate between these two possibilities, an additional two-process mutual exclusion algorithm is invoked inside *ExecuteCS* (lines 12 and 15 in Fig. 3), which can be easily implemented in $\Theta(1)$ time [22]. Promoted processes invoke *ExecuteCS(side)* with $side = 1$, and other processes with $side = 0$. In any case, $p$ later resets any component(s) it has acquired.

The algorithm also uses a serial waiting queue, named *WaitingQueue*, which is accessed only within exit sections. A "barrier" mechanism (lines 4 and 14) is used that ensures that multiple processes do not execute their exit sections concurrently. As a result, *WaitingQueue* can be implemented as a sequential data structure, in which each operation takes $\Theta(1)$ time. When a process $p$, inside its exit section, discovers another waiting process $q$, $p$ adds $q$ to the waiting queue. In addition, $p$ dequeues a process $r$ from the queue (if the queue is nonempty), and "promotes" $r$ to its critical section (lines 5–10).
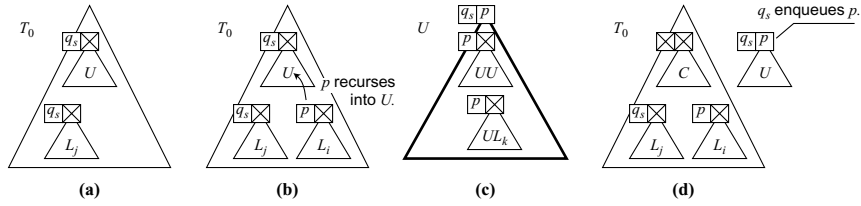
**Fig. 4.** An example of recursive execution. The left-side boxes represent the *winner* £eld; the right-side ones, *waiter*. **(a)** A process $q_s$ acquires both $L_j$ and $U$ and performs a regular (non-promoted) entry. **(b)** Process $p$ acquires $L_i$, but fails to acquire $U$. **(c)** $p$ recurses into $U$, acquires its two components $UL_k$ and $UU$, and becomes $U$'s primary waiter. (Note that the tree depicted here is $U$, not $T_0$.) **(d)** Process $q_s$, in its exit section, updates $T_0.upper$ to point to a clean tree, $C$, delays itself, and enqueus $p$ onto *WaitingQueue*.

The barrier is speci£ed by two procedures `Wait` and `Signal`. Since executions of `Wait` are serialized by $\text{Entry}_2$ and $\text{Exit}_2$, we can easily implement these procedures in $O(1)$ time. In CC machines, `Wait` can be de£ned as "**await** *Flag*; *Flag* := *false*" and `Signal` as "*Flag* := *true*," where *Flag* is a shared boolean variable. In DSM machines, a slightly more complicated implementation is required, which can be found in [14].

**Recursive execution.** We now consider the case that a process $p$ fails to acquire both of its components of $T_0$. (Until we consider the exit section below, $p$ is assumed to be de£ned as such.) Assume that $p$ fails to acquire $S$ (which may be either $L_i$ or $U$), because some other process $q_s$ has already acquired it. The case for $S = U$ is illustrated in Fig. 4. In this case, $p$ recurses into $S$ (we say that $p$ "enters" $S$), and executes an identical mutual exclusion algorithm, except for one difference: if $p$ acquires both components of $S$ along its path inside $S$ (which we denote by $SL_k$ and $SU$, respectively), then instead of entering its critical section, it writes its identity into another £eld *waiter* of $S$ (Fig. 4(c)). We say that $p$ is the *primary waiter* of $S$ in this case. If $p$ still fails to acquire both $SL_k$ and $SU$, then it recurses further into the component it failed to acquire. Therefore, we have:

**Property 1** A process $p$ that enters a tree $S$ ($\neq T_0$) eventually becomes the primary waiter of some constituent $S'$ of $S$.

Once $p$ becomes a primary waiter, it stops and waits until it is promoted by some other process.

After $p$ enters $S$, it tries to acquire $SL_k$ in $\Theta(1)$ steps. If $p$ succeeds, then it tries to acquire $SU$ in $\Theta(1)$ steps. Otherwise, some other process $r$ has already acquired $SL_k$. That process will eventually attempt to acquire $SU$ in $\Theta(1)$ steps, unless it has already done so. Since the £rst process to attempt to acquire $SU$ succeeds, we have the following.

**Property 2** If some process enters a tree $S$, then some process becomes $S$'s primary waiter in $\Theta(1)$ steps, that is, in $O(\Delta)$ time.

Inside its exit section, process $q_s$ (which has acquired $S$) £rst delays itself by $\Delta_0 = \Theta(\Delta)$ time, and then examines its path in order to discover other waiting processes (Fig. 4(d)). In particular, for each component $q_s$ has acquired (including $S$), $q_s$ determines if that component has a primary waiter. Thus:

**Property 3** If a process $q$ acquires a tree $S$, then $q$ enqueues $S$'s primary waiter (if any)

in $q$'s exit section.

As explained shortly, $p$ may enter $S$ only before $q_s$ £nishes its delay. Because $p$ has entered $S$, by Property 2, $q_s$'s delay ensures that $q_s$ indeed £nds a primary waiter of $S$.

If $p$ is the primary waiter of $S$, then $q_s$ enqueues $p$ onto the waiting queue; otherwise, $q_s$ enqueues the primary waiter of $S$, which eventually executes its exit section and examines the components of $S$ it has acquired. Continuing in this manner, every process that stopped inside $S$, including $p$, is eventually enqueued onto the waiting queue. Thus, $p$ eventually enters its critical section.

**Exit-section execution.** We now consider the exit section of a process $p$. As explained before, the barrier mechanism ensures that exit-section executions are serialized. For each component $S$ of $T_0$ that is acquired by $p$ (which may be $L_i$ or $U$), $p$ updates $T_0$'s pointer for $S$ so that it now points a "clean" tree $C$, as shown in Fig. 4(d). We assume the existence of a function $GetCleanTree$, which returns a pointer to a previously unused tree of a given order. (This results in unbounded space complexity. Space can be bounded by recycling used trees, as shown in [14].) Note that some process may still be executing inside $S$, which is now unlinked from $T_0$.

As explained above, after unlinking $S$, $p$ delays itself by $\Delta_0$, and thus ensures that if any process has entered $S$ (the "old" component), then some process has become its primary waiter. $p$ then checks for the primary waiter of $S$, and enqueues the waiter if it exists.

From the discussion so far, it is clear that the mutual exclusion algorithm at each tree $T$ of order $k$ incurs $\Theta(1)$ RMR-$\Delta$ time complexity, plus the time required for a recursive invocation at some component (of order $k-1$) of $T$. (Note that $p$ may recurse into either $L_i$ or $U$, but not both.) Thus, $\Theta(k)$ RMR-$\Delta$ time complexity is incurred at $T$, and $\Theta(K) = \Theta(\log \log N)$ at $T_0$.

A version of ALGORITHM T with bounded space is presented in detail in [14]. From ALGORITHM T, we have the following theorem.

**Theorem 1.** *The mutual exclusion problem can be solved with* $\Theta(\log \log N)$ *RMR-$\Delta$ (or RMR) time complexity on CC or DSM machines in the known-delay model.* $\qquad\square$

## 3 Lower Bound: System Model

In this section, we present the model of a shared-memory system used in our lower-bound proof. Due to space limitations, we only prove the lower bound in Table 1 pertaining to the RMR-$\Delta$ measure. Our system model is similar to that used in [5, 8].

**Shared-memory systems.** A *shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of computations $C$, a set of processes $P$, and a set of variables $V$. A *computation* is a £nite sequence of timed events. A *timed event* is a pair $(e, t)$, where $e$ is an event and $t$ is a nonnegative real value, specifying the time $e$ is executed. An *event* $e$, executed by a process $p \in P$, has the form of $[p, \mathsf{Op}, \ldots]$. We call $\mathsf{Op}$ the *operation* of event $e$, denoted $op(e)$. $\mathsf{Op}$ can be one of the following: $\mathsf{read}(v)$, $\mathsf{write}(v)$, or delay, where $v$ is a variable in $V$. For brevity, we sometimes use $e_p$ to denote an event executed by process $p$. The following assumption formalizes requirements regarding the atomicity of events.

**Atomicity Property:** Each event $e_p$ is one of the following.

- $e_p = [p, \mathsf{read}(v), \alpha]$. In this case, $e_p$ reads the value $\alpha$ from $v$. We call $e_p$ a *read event*.
- $e_p = [p, \mathsf{write}(v), \alpha]$. In this case, $e_p$ writes the value $\alpha$ to $v$. We call $e_p$ a *write event*.
- $e_p = [p, \mathsf{delay}]$. In this case, $e_p$ delays $p$ by a £xed amount $\Delta$, de£ned so that each event execution £nishes in $\Delta$ time. We call $e_p$ a *delay event*. □

In a computation, event timings must appear in nondecreasing order. (When multiple events are executed at the same time, their effect is determined by the order they appear in a computation.) The value of variable $v$ at the end of computation $H$, denoted $value(v, H)$, is the last value written to $v$ in $H$ (or the initial value of $v$ if $v$ is not written in $H$). The last event to write to $v$ in $H$ is denoted $writer\_event(v, H)$,[3] and the process that executes the event is denoted $writer(v, H)$. If $v$ is not written by any event in $H$, then we let $writer(v, H) = \bot$ and $writer\_event(v, H) = \bot$. The execution time of the last event of $H$ is denoted $last(H)$.

We use $\langle (e, t), \ldots \rangle$ to denote a computation that begins with the event $e$ executed at time $t$, $\langle \rangle$ to denote the empty computation, and $H \circ G$ to denote the computation obtained by concatenating computations $H$ and $G$. For a computation $H$ and a set of processes $Y$, $H \mid Y$ denotes the subcomputation of $H$ that contains all events of processes in $Y$. A computation $H$ is a $Y$-*computation* iff $H = H \mid Y$. For simplicity, we abbreviate these de£nitions when applied to a singleton set of processes (*e.g.*, $H \mid p$ instead of $H \mid \{p\}$).

**Mutual exclusion systems.** We now de£ne a special kind of shared-memory system, namely mutual exclusion systems, which are our main interest.

A *mutual exclusion system* $\mathcal{S} = (C, P, V)$ is a shared-memory system that satis£es the following properties. Each process $p \in P$ has an auxiliary variable $stat_p$ that ranges over $\{ncs, \ entry, \ exit\}$. The variable $stat_p$ is initially $ncs$ and is accessed only by the following events: $Enter_p = [p, \mathsf{write}(stat_p), entry]$, $CS_p = [p, \mathsf{write}(stat_p), exit]$, and $Exit_p = [p, \mathsf{write}(stat_p), ncs]$. We call these events *transition events*. These events represent the start of $p$'s entry section, $p$'s critical-section execution, and the end of $p$'s exit section, respectively.

We henceforth assume each computation contains at most one $Enter_p$ event for each process $p$, because this is suf£cient for our proof. The remaining requirements of a mutual exclusion system are as follows.

**De£nition:** For a computation $H$, we de£ne $\mathrm{Act}(H)$, the set of *active processes* in $H$, as $\{p \in P : value(stat_p, H) \text{ is } entry \text{ or } exit\}$. □

**Exclusion:** At most one process may be enabled to execute $CS_p$ after any $H \in C$.

**Progress (Livelock freedom):** Given $H \in C$, if some process is active after $H$, then $H$ can be extended by events of active processes so that some such process $p$ eventually executes either $CS_p$ or $Exit_p$.

**Cache-coherent systems.** On cache-coherent systems, some variable accesses may be handled locally, without causing interconnect traf£c. In order to apply our lower bound to such systems, we do not count every read/write event, but only critical events, as de£ned below. As shown in [5, 14], the number of critical events by any process is an asymptotic lower bound for the number of events (by that process) that incur interconnect traf£c in

---

[3] Although our de£nition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other.

systems with any combinations of write-through/write-back and write-invalidate/write-update caching schemes.

**Definition:** Let $e_p$ be an event in $H \in C$, and let $H = F \circ \langle e_p \rangle \circ \cdots$, where $F$ is a subcomputation of $H$. We say that $e_p$ is a *cache-miss event* in $H$ if one of the following conditions holds: **(i)** it is the first read of a variable $v$ by $p$; **(ii)** it writes a variable $v$ such that $writer(v, F) \neq p$. $\square$

**Definition:** We say that an event $e_p$ is *critical* iff one of the following conditions holds. **(i)** $e_p$ accesses $stat_p$. (In this case, $e_p$ is called a *transition event*.) **(ii)** $e_p$ is a delay event. **(iii)** $e_p$ is a cache-miss event. $\square$

Transition events are defined as critical because this allows us to combine certain cases in the proofs that follow. Since a process executes only three transition events per critical-section execution, this has no asymptotic impact.

Note that the above definition of a cache-miss event depends on the particular computation that contains the event, specifically the prefix of the computation preceding the event. Therefore, when saying that an event is (or is not) a cache-miss event or a critical event, the computation containing the event must be specified.

**Properties of computations.** The timing requirements of a mutual exclusion system are captured by requiring the following for each $H \in C$.

**T1:** For any two timed events $(e_p, t)$ and $(f_q, t')$ in $H$, if $e_p$ precedes $f_q$, then $t \leq t'$ holds.

**T2:** For any timed event $(e_p, t)$ in $H$, if $e_p \neq Exit_p$ and if $last(H) > t + \Delta$, then $e_p$ is not the last event in $H \mid p$.

**T3:** For any two consecutive timed events $(e_p, t)$ and $(f_p, t')$ in $H \mid p$, the following holds:
$$\begin{cases} t' = t + \Delta, & \text{if } e_p \text{ is a delay event,} \\ t + \Delta_c \leq t' \leq t + \Delta, & \text{if } e_p \text{ is a cache-miss event,} \\ t \leq t' \leq t + \Delta, & \text{otherwise,} \end{cases}$$
where $\Delta_c$ is a lower bound (less than $\Delta$) on the duration of a cache-miss event.

Note that T3 allows noncritical events to execute arbitrarily fast. If this were not the case, then a "free" delay statement that is not counted when assessing time complexity could be implemented by repeatedly executing noncritical events (*e.g.*, by reading a dummy variable). In fact, our model allows noncritical events that take zero duration. (Thus, our proof does not apply to completely synchronous systems.) We could have instead required them to have durations lower-bounded by an arbitrarily small positive constant, at the expense of more complicated bookkeeping in our proofs.

On the other hand, cache-miss events take some duration between $\Delta_c$ and $\Delta$, and hence our lower bound applies to systems with both upper and lower bounds on the execution time of such events. All delay events are assumed to have an exact duration of $\Delta$. Thus, the issue of whether delays are upper bounded does not arise in our proof.

We assume the following standard properties for computations: changing only the timings of a valid computation leaves it valid (provided that T1–T3 are preserved); a prefix of a valid computation is also valid; a process determines its next event only based on its execution history; and reading a variable returns the value last written to it, or its initial value if it has not been written.

## 4   Lower Bound: Proof Sketch

Our lower-bound proof focuses on a special class of computations called "regular" computations. A regular computation consists of events of two groups of processes, "invisible processes" and "visible processes." Informally, only a visible process may be known to other processes. Each invisible process is in its entry section, competing with other (invisible and visible) processes. A visible process may be in any section.

At the end of this section, a detailed overview of the proof is given. Here, we give cursory overview, so that the definition of a regular computation will make sense. Initially, we start with a regular computation in which all the processes in $P$ are invisible. The proof proceeds by inductively constructing longer regular computations, until the desired lower bound is attained. At the $m^{\text{th}}$ induction step, we consider a regular computation $H_m$ with $n$ invisible processes and at most $m$ visible processes. The regularity condition defined below ensures that *no participating process has knowledge of any other process that is invisible*. Thus, we can "erase" any invisible process (*i.e.*, remove its events from the computation) and still get a valid computation.

After $H_m$, some invisible processes may be "blocked" due to knowledge of visible processes — that is, they may start repeatedly reading variables read previously, not executing any critical event until visible processes take further steps. In order to construct a longer computation $H_{m+1}$, we need a "sufficient" number of unblocked processes. As shown later, it is possible to extend $H_m$ to obtain a computation $A$ by letting visible processes execute some further steps (and by possibly erasing some invisible processes) such that, after $A$, enough invisible processes are unblocked.

To construct $H_{m+1}$, we append to $A$ one next critical event for each such unblocked process. Since these next critical events may introduce information flow, some invisible processes may need to be erased to ensure regularity. Sometimes erasing alone does not leave enough active processes for the next induction step. This may happen only if some variable $v$ exists that is accessed by "many" of the critical events we are trying to append. In that case, we erase processes accessing other variables and then apply the "covering" strategy: we add the last process to write to $v$ to the set of visible processes. All subsequent reads of $v$ must read the value written by this process, and hence information flow from invisible processes is prevented. Thus, we can construct. $H_{m+1}$.

The induction continues until the desired lower bound of $\Omega(\log \log N)$ critical events is achieved. This basic proof strategy of erasing and covering has been used previously to prove several other lower bounds for concurrent systems ([1, 5, 10, 15, 20] — see also [7]). Note that, in asynchronous systems, we can "stall" *all* invisible processes until all visible processes finish execution. Thus, blocked processes pose no problem. However, this is clearly impossible in semi-synchronous systems, resulting in additional complications (*e.g.*, finding enough unblocked processes). We now define the regularity condition.

**Definition:** Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system. We say that $H \in C$ is *regular* iff there exist two disjoint sets of processes, $\text{Inv}(H)$, the set of *invisible processes*, and $\text{Vis}(H)$, the set of *visible processes*, satisfying $\text{Inv}(H) \cup \text{Vis}(H) = \{p \in P:$ $H \mid p \neq \langle \rangle \}$, such that the following conditions hold.

- **R1:** If a process $p$ writes to a variable $v$, and if another process $q$ reads *that* value from $v$, then $p$ is visible (*i.e.*, $p \in \text{Vis}(H)$).

- **R2:** If a variable $v$ is accessed by more than one process in $H$ and if $H$ contains a write to $v$, then $writer(v, H) \in \text{Vis}(H)$ holds.[4]
- **R3:** Every invisible process is in its entry section.
- **R4:** If two events $e_p$ and $f_p$ access a variable $v$, and if some process writes $v$ in between, then some visible process writes $v$ between $e_p$ and $f_p$. (This condition is used to show that the property of being a critical event is preserved after erasing some invisible processes.) □

**Detailed proof overview.** Initially, we start with a regular computation $H_1$, where $\text{Inv}(H_1) = P$, $\text{Vis}(H_1) = \{\}$, and each process has one critical event, executed at time 0. We then inductively show that other longer computations exist, the last of which establishes our lower bound. Each computation is obtained by covering some variable and/or erasing some processes.

At the $m^{\text{th}}$ induction step, we consider a computation $H = H_m$ in which each process in $\text{Inv}(H)$ executes $m$ critical events, $|\text{Vis}(H)| \leq m$, and each active process executes its last event at the same time $t = t_m$. Furthermore, we assume

$$\log \log n = \Theta(\log \log N) \quad \text{and} \quad m = O(\log \log N), \tag{1}$$

where $n = |\text{Inv}(H)|$. We construct a regular computation $G = H_{m+1}$ such that $\text{Act}(G)$ consists of $\Omega(\sqrt{n}/(\log \log N)^2)$ processes, each of which executes $m + 1$ critical events in $G$. (To see why (1) is reasonable, note that $\log \log n$ is initially $\log \log N$ and decreases by $\Theta(1)$ at each induction step.) The construction method is explained below. For now, we assume that all $n$ invisible processes are unblocked. At the end of this section, we explain how to adjust the argument if this is not the case.

**Definition:** If $H$ is a regular computation, then a process $p \in \text{Inv}(H)$ is *blocked after $H$* iff there exists no $p$-computation $A$ such that $A$ contains a critical event (in $H \circ A$) and $H \circ A \in C$ holds. □

Due to the Exclusion property, each unblocked process (except at most one) executes a critical event *before* entering its critical section. We call these events "next" critical events, and denote the corresponding set of processes by $Y$. We consider three cases, based on the variables accessed by these next critical events.

**Case 1: Delay events.** If there exist $\Omega(\sqrt{n})$ processes that have a next critical event that is a delay event, then we erase all other invisible processes and append these delay events. Since a delay event does not cause information flow, the resulting computation is regular. (Such delays may force visible processes to take steps. We explain how to handle this after Case 3.)

In the remaining two cases, we can assume that $\Theta(n)$ next critical events are critical reads or writes.

**Case 2: Erasing strategy (Fig. 5).** Assume that there exist $\Omega(\sqrt{n})$ distinct variables that are accessed by some next critical events. For each such variable $v$, we select one process whose next critical event accesses $v$. Let $Y'$ be the set of selected processes. Since

---

[4] Note that R2 is not a consequence of R1. For example, consider a case in which a process $q = writer(v, H)$ writes $v$ *after* another process $p$ reads $v$.
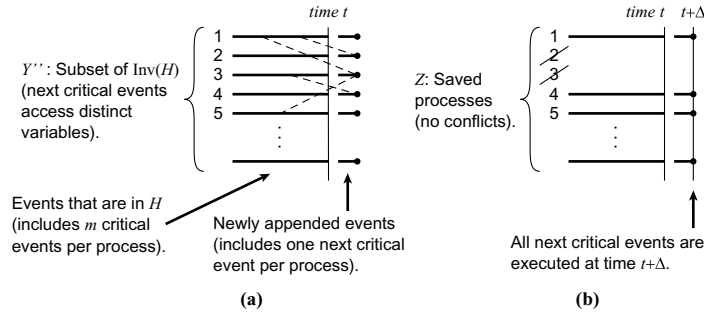
**Fig. 5.** Erasing strategy. Here and in later £gures, black circles (●) respresent critical events.

we have at most $m$ visible processes, and since each visible process executes at most $\log \log N$ critical events in $H$ (otherwise, our lower bound is achieved), they collectively access at most $m \log \log N$ distinct variables. We erase processes in $Y'$ that access these variables. By (1), we have $m \log \log N = o(\sqrt{n})$, so we still have $\Omega(\sqrt{n})$ processes left. Let $Y''$ be the subset of $Y'$ that is not erased (Fig. 5(a)). We now eliminate remaining possible con¤icts among processes in $Y''$ by constructing a "con¤ict graph" $\mathcal{G}$ as follows.

Each process $p$ in $Y''$ is considered a vertex in $\mathcal{G}$. By induction, $p$ has $m$ critical events in $H$. If such an event accesses the same variable as the next critical event of some other process $q$, then introduce the edge $\{p, q\}$.

Since the next critical events of processes in $Y''$ access distinct variables, each process generates at most $m$ edges. By applying Turán's theorem (see [14]), we can £nd a subset $Z \subseteq Y''$ with no con¤icts such that $|Z| = \Omega(\sqrt{n}/m)$. By retaining $Z$ and erasing all other invisible processes, we can eliminate all con¤icts (Fig. 5(b)) and construct $G$.

**Case 3: Covering strategy.** Assume that the next critical events collectively access $O(\sqrt{n})$ distinct variables. Since there are $\Theta(n)$ next critical reads/writes, some variable $v$ is accessed by $\Omega(\sqrt{n})$ next critical events. Let $Y_v$ be the set of these processes. First, we retain $Y_v$ and erase all other invisible processes. Let the resulting computation be $H'$. We then arrange the next critical events of $Y_v$ by placing all writes before all reads, and letting all writes and reads execute at time $t + \Delta$. In this way, the only information ¤ow among processes in $Y_v$ is that from the "last writer" of $v$, denoted $p_{\text{LW}}$, to any subsequent reader (of $v$). We then add $p_{\text{LW}}$ to the set of visible processes, *i.e.*, $p_{\text{LW}} \in \text{Vis}(G)$ holds. Thus, no invisible process is known to other processes, and we can construct $G$.

Finally, after any of the three cases, we let each active visible process execute one more event (critical or noncritical), so that it "keeps pace" with the invisible processes and preserves T2 and T3. Each such event may cause a con¤ict with at most one invisible process, so we erase at most $m$ more invisible processes, which is $o(\sqrt{n}/m)$, by (1).

From the discussion so far, we have the following lemma.

**Lemma 1.** *Let $H$ be a regular computation satisfying the following: $n = |\text{Inv}(H)|$, each $p \in \text{Inv}(H)$ has exactly $m$ critical events in $H$ and is unblocked after $H$, $|\text{Vis}(H)| \leq m$, and each active process executes its last event at time $t$.*

*Then, there exists a regular computation $G$ satisfying the following: $|\text{Inv}(G)| = \Omega(\sqrt{n}/m)$, each $p \in \text{Inv}(G)$ has exactly $m + 1$ critical events in $G$, $|\text{Vis}(G)| \leq m + 1$, and each active process in $G$ executes its last event at time $t + \Delta$.* □
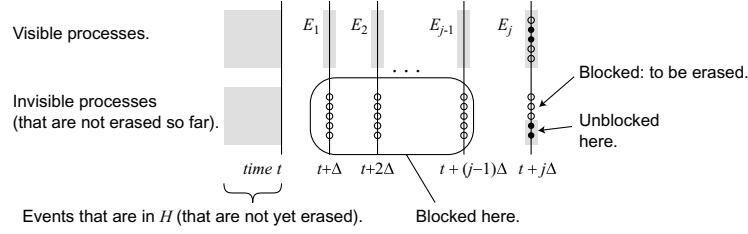
**Fig. 6.** Finding unblocked processes. In this £gure, black and white circles (●, ○) respresent critical and noncritical events, and shaded boxes represent computations made of possibly many events.

**Finding unblocked processes.** We now explain how we can £nd "enough" unblocked invisible processes. Consider $F = H \mid \mathrm{Vis}(H)$, a computation obtained by erasing all invisible processes. It can be easily shown that $F$ is a regular computation in $C$. Let the processes in $\mathrm{Vis}(H)$ execute in "lockstep," *i.e.*, let each process in $\mathrm{Vis}(H)$ execute exactly one event per each interval of length $\Delta$. By extending $F$ in such a way until all visible processes £nish execution, we have an extension $F \circ D = F \circ D_1 \circ D_2 \circ \cdots \circ D_{k'}$, where each $D_j$ contains exactly one event by each process in $\mathrm{Vis}(H)$, executed at time $t + j\Delta$. Since we allow noncritical events to take zero time, if a segment $D_j$ (where $j < k'$) consists of only noncritical events, then we can "merge" it into the next segment and create a segment of length $\Delta$. Continuing in this way, we can de£ne an extension $F \circ E = F \circ E_1 \circ E_2 \circ \cdots \circ E_k$, such that: **(i)** $D$ and $E$ consist of the same sequence of events (only event timings are different); **(ii)** every event in $E_j$ is executed at time $t + j\Delta$; **(iii)** every $E_j$ (except perhaps the last one) contains some critical event; **(iv)** any critical event in $E_j$ is the last event by that process in $E_j$.

If $E$ contains more than $m \log \log N$ critical events, then some visible process executes more than $\log \log N$ critical events in $E$, and hence our lower bound is achieved. Thus, assume otherwise. By (iii), we have $k \leq m \log \log N + 1$.

For each $E_j$, starting with $E_1$, we do the following to £nd $n'$ invisible processes that are unblocked simultaneously, where $n' = n/(k+1) - m$. We £rst append the *noncritical* events of $E_j$. It can be shown that noncritical events do not cause information ¤ow. After that, we determine how many invisible processes are unblocked. If $n'$ or more such processes are unblocked, then we can erase all blocked invisible processes and construct $G$ by applying Lemma 1, as depicted in Fig. 6. Otherwise, we erase the *unblocked* invisible processes, and let each remaining (blocked) invisible process execute one noncritical event, in order to keep pace. (See $E_1, \ldots, E_{j-1}$ of Fig. 6.) Finally, we append the critical events of $E_j$. By (iv), each visible process may execute at most one critical event in $E_j$. Thus, $E_j$ has at most $m$ critical events. If some critical event reads a variable written by an invisible process, then we erase that invisible process to prevent information ¤ow. Thus, we can append the critical events in $E_j$, erase at most $m$ invisible processes, and preserve regularity. We then repeat the same procedure for the next segment $E_{j+1}$, and so on.

Note that we erase at most $n' + m = n/(k+1)$ invisible processes to append each $E_j$. It is clear that either we £nd $n'$ invisible unblocked processes (after appending $E_1 \circ \cdots \circ E_j$, for some $0 \leq j \leq k$), or we append all of $E$. However, in the latter case, we have erased at most $kn/(k+1)$ invisible processes. Thus, at least $n/(k+1)$ invisible processes remain after $E$. Moreover, since each such process may only know of visible

processes, and since no visible process is active after $E$ (*i.e.*, they are in their noncritical sections), each remaining invisible process must make progress toward its critical section, *i.e.*, it cannot be blocked. Hence, in either case, we have at least $n/(k+1) - m = \Omega(n/(m \log \log N))$ unblocked invisible processes. (Note that (1) implies $m = o(n/k)$.) Therefore, by erasing all blocked processes and applying Lemma 1, we can construct $G$. Thus, we have the following lemma.

**Lemma 2.** *Let $H$ be a regular computation satisfying the following: $n = |\mathrm{Inv}(H)|$, each $p \in \mathrm{Inv}(H)$ has exactly $m$ critical events in $H$, $|\mathrm{Vis}(H)| \leq m$, and each active process executes its last event at time $t$.*

*Then, there exists a regular computation $G$ satisfying the following: $|\mathrm{Inv}(G)| = \Omega(\sqrt{n}/(\log \log N)^2)$, $|\mathrm{Vis}(G)| \leq m + 1$, each $p \in \mathrm{Inv}(G)$ has exactly $m + 1$ critical events in $G$, and each active process executes its last event at time $t'$, for some $t' > t$.* $\square$

By applying Lemma 2 inductively, we have the following.

**Theorem 2.** *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation in which a process incurs $\Omega(\log \log N)$ RMR-$\Delta$ time complexity in order to enter and then exit its critical section.* $\square$

As shown in [14], the above lower bound can be adapted for the RMR measure, provided delays are unbounded. Thus, we have the following.

**Theorem 3.** *For any mutual exclusion system $\mathcal{S} = (C, P, V)$ with unbounded delays, there exists a computation in which a process incurs $\Omega(\log \log N)$ RMR time complexity in order to enter and then exit its critical section.*

## 5   Concluding Remarks

To the best of our knowledge, this paper is the £rst work on timing-based mutual exclusion algorithms in which all busy-waiting is by local spinning. Our speci£c interest has been to determine whether lower RMR time complexity is possible (for mutual exclusion algorithms based on reads, writes, and comparison primitives) in semi-synchronous systems with delays. We have shown that this is indeed the case, regardless of whether delays are assumed to be counted when assessing time complexity, and whether delay values are assumed to be upper bounded. For each system model and time measure that arises by resolving these issues, we have presented an algorithm that is asymptotically time-optimal. Interestingly, under the RMR measure with unbounded delays, DSM machines allow provably lower time complexity than CC machines. In contrast to this situation, it is usually the case that designing ef£cient local-spin algorithms is easier for CC machines than for DSM machines.

## References

1. Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. ACM, July 2000.

2. R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 800–809. ACM, May 1994.

3. R. Alur and G. Taubenfeld. How to share an object: A fast timing-based solution. In *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pages 470–477. IEEE, 1993.

4. R. Alur and G. Taubenfeld. Fast timing-based algorithms. *Distributed Computing*, 10(1):1–10, 1996.

5. J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 90–99. ACM, August 2001.

6. J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-$\phi$ primitives. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, pages 538–547. IEEE, May 2003.

7. J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 2003 (to appear).

8. J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

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

10. J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

11. R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.

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

13. J. Kessels. Arbitration without common modifiable variables. *Acta Informatica*, 17:135–141, 1982.

14. Y.-J. Kim and J. Anderson. Timing-based mutual exclusion with local spinning. Manuscript, July 2003. Available from `http://www.cs.unc.edu/~anderson/papers.html`.

15. Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, October 2001.

16. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

17. N. Lynch and N. Shavit. Timing based mutual exclusion. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 2–11. IEEE, December 1992.

18. J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

19. S. Ramamurthy, M. Moir, and J. Anderson. Real-time object sharing with minimal support. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242. ACM, May 1996.

20. E. Styer and G. Peterson. Tight bounds for shared memory symmetric mutual exclusion. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–191. ACM, August 1989.

21. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, June 1977.

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