Timing-based Mutual Exclusion with Local Spinning*

Yong-Jik Kim and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
Email: {kimy,anderson}@cs.unc.edu

July 2003

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. Also, algorithms with $O(\log N)$ time complexity are known. Thus, for algorithms in this class, logarithmic or near-logarithmic RMR time complexity is fundamentally required.

In this paper, we show that lower RMR time complexity is attainable in semi-synchronous systems with *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. 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. It follows from these results that semi-synchronous systems allow mutual exclusion algorithms with substantially lower RMR time complexities than completely asynchronous systems, regardless of how one resolves the issues noted above.

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* [12]. 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. (A detailed discussion of the CC and DSM machine models can be found in [10].)

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

*Work supported by NSF grant CCR 0208289.

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

For example, *compare-and-swap* can be specified by defining $f(old, new) = new$ and $g(temp, old, new) = old$. (In this paper, we assume that *compare-and-swap* returns the accessed variable's old value.)

In earlier work, Cypher [15] 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 [6], we presented for this class of algorithms a substantially improved lower bound of $\Omega(\log N/\log\log N)$ RMRs. This lower bound 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 [27].[1] The proofs of these 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, 20, 21]. Such algorithms are the focus of this paper. In this paper, we exclusively consider the *known-delay* model [4, 20, 21], in which there is a known upper bound, denoted Δ, on the time required to access (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, 23]. Thus, it suffices 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.

Our specific objective is to determine whether lower RMR time complexity is attainable in semi-synchronous systems with delay statements. Specifically, we wish to determine if timing-based algorithms with $o(\log N/\log\log N)$ RMR time complexity exist. Note that, from the lower bound mentioned above [6], $o(\log N/\log\log N)$ time is *not* possible in completely asynchronous systems.

When assessing the RMR time complexity of timing-based algorithms, the question of whether delays should be counted arises. To see why this is an issue, consider the *test-and-set*/*reset* implementation in Figure 1, which is taken from [23].[3] A process p performs a successful *test-and-set* by reading $Z = \perp$ and by then writing $Z := p$. Of course, many processes may concurrently find $Z = \perp$ and then write Z. The delay at line 3 causes line 4 to be executed *after* all such "concurrent" writes have been performed, so only the *last* process to update Z returns *true* at line 4.

This usage of delays is typical of timing-based algorithms: a process performs a delay that is long enough to ensure that any execution of a certain code fragment by other processes completes before the delay ends.

[1] In contrast, several $\Theta(1)$ algorithms are known that are based on noncomparison primitives (*e.g.*, [9, 13, 16, 22]). A detailed discussion of such algorithms can be found in [10]. 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 Δ 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 Δ is unknown *a priori*, and hence, cannot be used directly in an algorithm.

[3] It is assumed here that a process may invoke *reset* only if it is has previously performed a successful *test-and-set*, and has not invoked *reset* since.

```
shared variable Z: {0..N − 1} ∪ {⊥}
procedure test-and-set()                    procedure reset()
  returns boolean                           5 : Z := ⊥
1 : v := Z;
2 : if v = ⊥ then Z := p fi;
3 : delay(2 · Δ);
4 : return (v = ⊥ ∧ Z = p)
```

Figure 1: *test-and-set* and *reset*.

(Note that if such a code fragment is short, then it may be reasonable to execute it with interrupts disabled, in which case determining an appropriate delay value is relatively straightforward.) Such a code fragment will typically contain at least one RMR. Given this, it may make sense to include delays when assessing time complexity. Accordingly, we define the *RMR-Δ time complexity* of an algorithm to be the total number of RMRs and *delay*(Δ) 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 Δ is fixed 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-Δ measure as the better choice.

As we shall see, the exact semantics assumed of the statement *delay*(Δ) is of relevance as well. It is reasonable to assume that a process is delayed by *at least* Δ time units when invoking this statement. However, it is not clear whether a specific upper bound on the delay duration should be assumed. For completeness, we once again consider both possibilities. (Note that assuming an upper bound places constraints on the way processes are scheduled. Such constraints may be problematic in practice if delay durations are long.)

Our results are summarized in Table 1. The headings "Delays Bounded/Unbounded" indicate whether delay durations are assumed to be upper bounded. The other headings should be self-explanatory. Each table entry gives a time-complexity figure that is shown to be optimal by giving an algorithm, and for the $\Theta(\log \log N)$ entries, a lower bound. The main conclusion to be drawn from these results is the following: *in semi-synchronous systems in which delay statements are supported, substantial improvements in RMR time complexity are possible when devising mutual exclusion algorithms, regardless of how one resolves the issues of whether to count delays and how to define the semantics of the* delay *statement.*

The results summarized in Table 1 complete a body of work developed by us over the last few years on time-complexity limits pertaining to mutual exclusion algorithms. Such algorithms can be categorized according to the assumptions made about the underlying hardware or system. In our prior work, we have studied algorithms based on atomic reads and writes [5, 7], nonatomic reads and writes [8], and stronger synchronization primitives (such as *compare-and-swap* and *fetch-and-store*) [9]. Most mutual exclusion algorithms presented in the literature fit within one of these categories, with one notable exception: timing-based algorithms, the topic of this paper.

In the following sections, we establish the results of Table 1. First, we present ALGORITHMS DSM, CC, and T in Sections 2–4, respectively. We then establish our lower bounds in Sections 5 and 6. We conclude in Section 7.

2 $\Theta(1)$ DSM Algorithm

In this section, we describe ALGORITHM DSM, which has $\Theta(1)$ RMR-Δ time complexity (and hence $\Theta(1)$ RMR time complexity) on DSM machines. Upper bounds on delays are not required. We first consider an unbounded version, which is illustrated in Figure 2. In this and subsequent figures, we assume that each labeled sequence of statements is atomic; in each figure, each labeled sequence reads or writes at most one shared variable. (References to unspecified code fragments, such as in line 11 in Figure 2, should be interpreted as *branches* to these code fragments.)

Arch.	RMR Time Complexity: Delays Bounded	RMR Time Complexity: Delays Unbounded	RMR-Δ Time Complexity: 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, Theorem 5}	$\Theta(\log \log N)$ {Alg. T, Theorem 4}

Table 1: Summary of results. Each entry gives a time-complexity figure that is shown to be optimal.

Overview of Algorithm DSM**.** The basic structure of Algorithm DSM is as follows. A process p tries to enter its critical section by performing a *compare-and-swap* operation on the shared variable *Lock* (line 2 of Figure 2). (Recall that *compare-and-swap* can be implemented in $\Theta(1)$ time using delays [3, 23].) *Lock* acts as a basic *test-and-set* lock, except that *compare-and-swap* is used instead: if the *compare-and-swap* operation succeeds, then p enters its critical section.

In an ordinary *test-and-set* lock, if the *test-and-set* operation fails, then p repeats the operation until it succeeds, resulting in unbounded RMRs. In order to avoid this, Algorithm DSM is constructed as follows: if p fails to acquire *Lock*, then it is eventually "promoted" to its critical section by some other process.

To arbitrate between these two possibilities, an additional two-process mutual exclusion algorithm is used, which can be easily implemented in $\Theta(1)$ time [27]. The two-process entry section is invoked with a process identifier of "0" by processes that successfully acquire *Lock* (line 11), and a process identifier of "1" by promoted processes (line 14). To ensure that multiple processes are not concurrently promoted, a shared variable *Promoted* is used to hold the identity of a promoted process (if one exists), as described shortly.

We now consider the case that p fails to acquire *Lock* because another process q has already acquired it. In that case, p notifies q of its presence by writing a shared variable $Waiting[q][p]$, which is local to q (line 12). (Recall that variable locality is statically determined in DSM machines.) Since each of $Waiting[q]$ $[0..N-1]$ is local to q, q may examine these variables in its exit section, and construct a queue $LocalQueue[q]$ of found processes (*i.e.*, $\{r\colon Waiting[q][r] = true\}$) without executing any RMRs (lines 18–21). q then merges $LocalQueue[q]$ into a serial waiting queue, named *GlobalQueue* (line 22). This queue is accessed only within exit sections. A "barrier" mechanism (lines 6 and 10) is used that ensures that multiple processes do not execute their exit sections concurrently. Because `Wait` is effectively part of a critical section, these procedures can easily be implemented in $\Theta(1)$ time, as explained shortly.

Thanks to the barrier mechanism, *GlobalQueue* can be implemented as a sequential data structure. The queues *GlobalQueue* and *LocalQueue* are accessible by the usual *Enqueue* and *Dequeue* operations, plus a *Merge* operation (which merges the content of $LocalQueue[q]$ into *GlobalQueue*), each of which can be implemented in $\Theta(1)$ time. We assume that *Dequeue* returns $\perp$ when applied to an empty queue.

Note that, in order to implement *Merge* in $\Theta(1)$ time, the nodes of $LocalQueue[q]$ must be linked onto *GlobalQueue*. (If we instead *copy* the content of $LocalQueue[q]$ to *GlobalQueue*, then the copy operation could incur $\Theta(N)$ RMRs, since we cannot in general expect the variables that comprise *GlobalQueue* to be local to q.) Thus, $LocalQueue[q]$ must be implemented with *shared* variables that are local to q.

Process q thus adds processes that have written $Waiting[q][\ldots]$ (including p in the case considered here) to *GlobalQueue* (lines 19–22). In addition, q (and each process later executing its exit section) checks if a promoted process currently exists (line 24), and if not, dequeues a process r from *GlobalQueue* (if it is nonempty), and "promotes" r to its critical section (lines 25–27). (This is rather similar to helping mechanisms used in wait-free algorithms [17].)

There are two key additional ideas in this algorithm. First, a process q that has acquired *Lock* resets *Lock* in its exit section (line 16) and delays (line 17), until any process p that read $Lock = q$ has finished writing $Waiting[q][p]$, *i.e.*, Δ_0 is large enough to satisfy following property.

Property 1 A process that reads $Lock = q$ at line 2 executes line 12 in Δ_0 time.

Second, even though scanning $Waiting[q][\ldots]$ and generating a local queue requires $\Theta(N)$ local operations (accessing $\Theta(N)$ variables), these operations do not contribute to RMR-Δ time complexity. Interestingly, this is only possible in DSM machines, because in CC machines, the first access of each of these variables causes a cache miss. Thus, the algorithm's time complexity is $\Theta(1)$ in DSM machines, but $\Theta(N)$ in CC

```
shared variables                                              private variables
    Spin: array[0..N − 1] of boolean;                             lock, next, q: (⊥, 0..N − 1)
    Lock, Promoted: (⊥, 0..N − 1) initially ⊥;
    Waiting: array[0..N − 1][0..N − 1] of boolean initially false;
    GlobalQueue: queue of {0..N − 1} initially empty;
    LocalQueue: array[0..N − 1] of queue of {0..N − 1}

process p ::    /* 0 ≤ p < N */
while true do
0:  Noncritical Section;
1:  Spin[p] := false;
2:  lock := CAS(&Lock, ⊥, p);
    if lock = ⊥ then
3:      LockWinnerEntry()
    else
4:      PromotedEntry()
    fi;
5:  Critical Section;
6:  Wait();            /* wait at the barrier */
    if lock = ⊥ then
7:      LockWinnerExit()
    else
8:      PromotedExit()
    fi
9:  TryToPromote();
10: Signal()           /* open the barrier */
od

procedure LockWinnerEntry()
11: Entry_2(0)

procedure PromotedEntry()
12: Waiting[lock][p] := true;
13: await Spin[p];
14: Entry_2(1)

procedure LockWinnerExit()
15: Exit_2(0);
16: Lock := ⊥
17: delay(Δ_0);
18: LocalQueue[p] := new copy of an empty queue;
    for q := 0 to N − 1
19:     if Waiting[p][q] then
20:         Enqueue(LocalQueue[p], q);
21:         Waiting[p][q] = false
        fi
    od;
22: Merge(GlobalQueue, LocalQueue[p])

procedure PromotedExit()
23: Exit_2(1)

procedure TryToPromote()
/* promote a waiting process (if any) */
24: q := Promoted;
    if (q = p) ∨ (q = ⊥) then
25:     next := Dequeue(GlobalQueue);
26:     Promoted := next;
27:     if next ≠ ⊥ then Spin[next] := true fi
    fi
```

Figure 2: ALGORITHM DSM, unbounded space version.

machines. (In contrast to this situation, it is usually the case that designing efficient local-spin algorithms is easier for CC machines than for DSM machines.)

The barrier mechanism. We now explain how to implement the barrier mechanism. Procedures `Wait` and `Signal` can be implemented by the code fragments on the left and right below, respectively.

```
a: Waiter := p;                                   g: if CAS(B, closed, open) = waiting then
b: Spin′[p] := false;                             h:     next := Waiter;
c: if CAS(B, closed, waiting) = closed then       i:     Spin′[next] := true
d:     await Spin′[p]                                 fi
   fi;
e: B := closed;
f: Waiter := ⊥
```

Variable B ranges over *open*, *closed*, and *waiting*, and indicates whether the barrier is open, closed without a waiting process, or closed with a waiting process. If B equals *waiting*, then *Waiter* indicates the current waiting process. $Spin'[p]$ is a spin variable used exclusively by process p (and, hence, it can be stored in memory local to p). Since there can be at most one process that may execute within `a`–`e` (respectively,

g–i) at any time, it is straightforward to establish the correctness of this implementation.

Correctness Proof. We now formally prove the correctness of ALGORITHM DSM. We begin by stating several notational conventions that will be used throughout this paper.

> **Notational Conventions:** We use $s.p$ to denote the statement with label s of process p, and $p.v$ to represent p's private variable v. Let S be a subset of the statement labels in process p. Then, $p@S$ holds if and only if the program counter for process p equals some value in S. (Note that if s is a statement label, then $p@\{s\}$ means that statement s of process p is *enabled*, *i.e.*, p has not yet executed s.)
>
> As stated earlier, we assume that each labeled sequence of statements is atomic. For example, consider statement $2.p$. If $2.p$ is executed while $Lock = \perp$ holds, then it establishes $p@\{3\}$. On the other hand, if $2.p$ is executed while $Lock \neq \perp$ holds, then it establishes $p@\{4\}$.
>
> We number statements in this way to reduce the number of cases that must be considered in the proof. Note that each numbered sequence of statements reads or writes at most one shared variable. (Because the `Entry`, `Exit`, `Wait`, and `Signal` routines are assumed to be correct, we can assume that they execute atomically and do not access any of the shared variables of ALGORITHM DSM.) □

We now give the list of invariants needed to prove the correctness of ALGORITHM DSM. Informally speaking, invariant (I1) states that at most one process may "hold" *Lock* at a time. Invariants (I2) and (I3) state that the identity of a promoted process is indicated by *Promoted*. (In particular, there exists at most one promoted process.) Invariant (I4) and (I5) state the mutual exclusion requirement and the correctness of the barrier, respectively.

invariant $(Lock = p) \ = \ (p@\{3, 11, 5, 6, 15, 16\} \ \wedge \ p.lock = \perp)$ (I1)

invariant $p@\{13\} \ \wedge \ Spin[p] = true \ \Rightarrow \ Promoted = p$ (I2)

invariant $p@\{14, 5, 6, 8, 23, 9, 24..26\} \ \wedge \ p.lock \neq \perp \ \Rightarrow \ Promoted = p$ (I3)

invariant (Mutual exclusion) $\left|\{p\colon p@\{5, 6\}\}\right| \leq 1$ (I4)

invariant $\left|\{p\colon p@\{7..10, 15..27\}\}\right| \leq 1$ (I5)

invariant If p is contained in any queue (*GlobalQueue* or *LocalQueue*$[q]$ for some q), then $p@\{13\}$ holds. (I6)

We now prove that each of (I1)–(I6) is an invariant. For each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.)

invariant $(Lock = p) \ = \ (p@\{3, 11, 5, 6, 15, 16\} \ \wedge \ p.lock = \perp)$ (I1)

Proof: The only statement that may establish the left-hand side is $2.p$, which also establishes the right-hand side. The only statement that may falsify the left-hand side is $16.q$, where q is any arbitrary process. The structure of the algorithm implies that, in this case, $q@\{16\} \wedge q.lock = Null$ holds. Hence, if $16.q$ is executed while the left-hand side holds, then (I1) implies $q = p$, and hence $16.q$ also falsifies the right-hand side.

The only statements that may establish or falsify the right-hand side are $2.p$ and $16.p$, which preserve (I1) as shown above. □

invariant $p@\{13\} \ \wedge \ Spin[p] = true \ \Rightarrow \ Promoted = p$ (I2)

Proof: The only statement that may establish $p@\{13\}$ is $12.p$, which may establish the antecedent only if executed when $Spin[p] = true$ holds. Since p has established $Spin[p] = false$ at line 1, this may happen only if some other process q has executed line 27 while $q.next = p$ holds. However, this in turn implies that

statement $25.q$ dequeued p from *GlobalQueue*. By (I6), statement $25.q$ was executed while $p@\{13\}$ was true. However, after p executes line 1, $p@\{13\}$ is continuously false until $12.p$ is executed, a contradiction. It follows that statement $12.p$ cannot establish the antecedent.

The only statement that may establish $Spin[p] = true$ is $27.q$, where q is any arbitrary process. In this case, q has established the consequent by executing $26.q$. Note that *Promoted* cannot change between the executions of $26.q$ and $27.q$, thanks to the barrier mechanism (I5).

The only statement that may falsify the consequent is $26.q$. However, due to the barrier mechanism, this may happen only if $24.q$ is executed while $Promoted = p$ holds, in which case $26.q$ may be executed only if $q = p$. Thus, the antecedent is false before and after the execution of $26.q$. $\square$

invariant $p@\{14, 5, 6, 8, 23, 9, 24..26\} \wedge p.lock \neq \bot \Rightarrow Promoted = p$ (I3)

Proof: The only statement that may establish $p@\{14, 5, 6, 8, 23, 9, 24..26\}$ is $13.p$, which may do so only if $Spin[p] = true$ holds. In this case, (I2) implies the consequent.

The only statement that may falsify the consequent is $26.q$, which may do so only if $q = p$, as shown in the proof of (I2). Thus, $26.q$ falsifies the antecedent in this case. $\square$

invariant $\left|\{p\colon p@\{5, 6\}\}\right| \leq 1$ (I4)

Proof: By (I1), there exists at most one non-promoted process that may execute $\texttt{Entry}_2(0)$ and $\texttt{Exit}_2(0)$. Similarly, by (I3), there exists at most one promoted process that may execute $\texttt{Entry}_2(1)$ and $\texttt{Exit}_2(1)$. Because the `Entry` and `Exit` routines are assumed to be correct, we have (I4). $\square$

invariant $\left|\{p\colon p@\{7..10, 15..27\}\}\right| \leq 1$ (I5)

Proof: By (I4), there exists at most one process that may invoke `Wait` (line 6). Similarly, while (I5) holds, there exists at most one process that may invoke `Signal` (line 10). Because the `Wait` and `Signal` routines are assumed to be correct (under this condition), we have (I5). $\square$

invariant If p is contained in any queue (*GlobalQueue* or $LocalQueue[q]$ for some q), then $p@\{13\}$ holds. (I6)

Proof: The antecedent may be established only if some process q finds $Waiting[q][p] = true$ at line 19 and enqueues p at line 20. However, for this to happen, p must have executed line 12 and established $p@\{13\}$. Moreover, once $p@\{13\}$ is established, it may be falsified only if $Spin[p] = true$ is established, which may happen only if some process dequeues q from *GlobalQueue* at line 25.

Thus, the antecedent is established only if $p@\{13\}$ is true, and $p@\{13\}$ continues to hold as long as the antecedent holds. $\square$

By (I4), we have mutual exclusion. By Property 1, every process that fails to acquire *Lock* is eventually enqueued onto $LocalQueue[q]$ (for some q), and hence p is eventually dequeued and promoted by some process (lines 25–27). Thus, we also have starvation freedom.

A bounded version. We now describe a simple change needed to bound the space complexity of Algorithm DSM. At line 18, $LocalQueue[p]$ is assigned a pointer to a local queue Q that is not concurrently accessed by any other process. Some processes are then enqueued onto Q at line 20, which is in turn merged into *GlobalQueue* at line 22. Note that the length of *GlobalQueue* is at most $N-1$ at any given moment. As a consequence, any process that is represented by a node in Q is dequeued from *GlobalQueue* after at most $N-1$ subsequent critical-section executions. Therefore, after these $N-1$ executions, Q is again "clean."

It follows that we can keep N copies of local queues (each capable of holding N process identifiers) *per each process*, and then use them in a circular order at line 18. With this transformation, the space complexity of Algorithm DSM becomes $\Theta(N^3)$. (Each of N processes has N copies of local queues, each of which incurs $\Theta(N)$ space complexity.) From Algorithm DSM, we have the following theorem.

```
shared variables                                                    private variables
    Spin: array[0..N−1] of boolean;                                     lock, next, q: (⊥, 0..N−1)
    Lock, Promoted: (⊥, 0..N−1) initially ⊥;
    GlobalQueue: queue of {0..N−1} initially empty;
    LocalQueue: array[0..N−1] of queue of {0..N−1}

process p ::   /* 0 ≤ p < N */

while true do
0:  Noncritical Section;

1:  Spin[p] := false;
2:  LocalQueue[p] := new copy of an empty queue;
3:  lock := CAS(&Lock, ⊥, p);
    if lock = ⊥ then
4:      LockWinnerEntry()
    else
5:      PromotedEntry()
    fi;

6:  Critical Section;

7:  Wait();                  /* wait at the barrier */
    if lock = ⊥ then
8:      LockWinnerExit()
    else
9:      PromotedExit()
    fi
10: TryToPromote();
11: Signal()                 /* open the barrier */
od

procedure LockWinnerEntry()
    /* execute a queue-lock entry section
       (lines 12 and 14)                       */
12: Enqueue_N(p);
13: Lock := ⊥;
14: Spin_N(p);
15: Entry_2(0)

procedure PromotedEntry()
16: delay(Δ_p^−, Δ_p^+);
17: Enqueue(LocalQueue[lock], p);
18: await Spin[p];
19: Entry_2(1)

procedure LockWinnerExit()
20: Exit_2(0);
21: delay(Δ_N^−, Δ_N^+);
22: Merge(GlobalQueue, LocalQueue[p]);
23: Exit_N(p);    /* execute a queue-lock exit section */

procedure PromotedExit()
24: Exit_2(1)

procedure TryToPromote()
/* promote a waiting process (if any) */
25: q := Promoted;
    if (q = p) ∨ (q = ⊥) then
26:     next := Dequeue(GlobalQueue);
27:     Promoted := next;
28:     if next ≠ ⊥ then Spin[next] := true fi
    fi
```

Figure 3: ALGORITHM CC, unbounded space version.

Theorem 1 *The mutual exclusion problem can be solved with $\Theta(1)$ RMR-Δ time complexity (and hence $\Theta(1)$ RMR time complexity) on DSM machines in the known-delay model.* □

3 $\Theta(1)$ Bounded-delay Algorithm

We now describe ALGORITHM CC, which has $\Theta(1)$ RMR time complexity on CC machines, provided delays can be upper bounded. As before, we present a version with unbounded space complexity, which is shown in Figure 3. (Space can be bounded in the same way as for ALGORITHM DSM.)

ALGORITHMS DSM and CC share a common structure: if a process p fails to acquire *Lock* because of another process q, then p is enqueued onto q's local waiting queue *LocalQueue*[q], and q later merges its local queue with *GlobalQueue* in its exit section. However, unlike ALGORITHM DSM, q cannot examine one variable for every potential waiter, since this would incur $\Theta(N)$ RMR time complexity in CC systems. Instead, the burden of enqueueing p onto q's local queue is placed on p. In order to enqueue itself, p must ensure that other processes do not access *LocalQueue*[q] concurrently. Toward this goal, we use a queue-based mutual exclusion algorithm as a subroutine, represented by procedures `Enqueue`, `Spin`, and `Exit`, as explained shortly. The manner in which these queues are accessed is illustrated in Figure 4, and explained

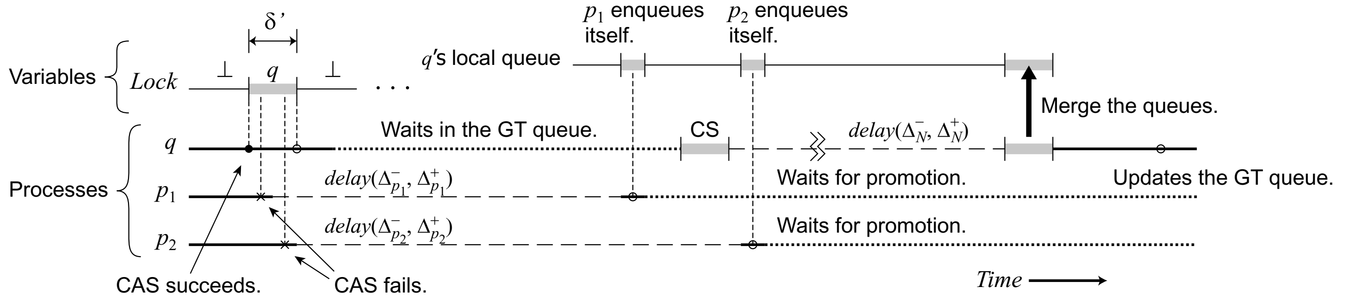

Figure 4: A possible timeline of ALGORITHM CC. Time flows from left to right. Black circles (•) represent successful *compare-and-swap* operations, crosses (×) represent unsuccessful ones, and white circles (∘) represent one or more simple write(s). Dotted and dashed lines represent local spinning and delay-statement executions, respectively.

below.

Since delays are upper-bounded, we can write a delay statement as $delay(\Delta^-, \Delta^+)$, where Δ^- (Δ^+) specifies the lower (upper) bound on the duration of the delay. (In general, Δ^- and Δ^+ are not independent.) We can then create $N+1$ possible delay durations, denoted (Δ_p^-, Δ_p^+) (for $0 \le p \le N$), such that

$$\Delta_{p+1}^- - \Delta_p^+ \ge \delta \tag{1}$$

holds for each p and some δ.

As explained below, the algorithm ensures the following.

Property 1 If a process q acquires *Lock* (line 3 of Figure 3), then q establishes $Lock = \perp$ (line 13) within $\delta' = \Theta(\Delta)$ time, for some δ'.

Therefore, if multiple processes p_i (for some range of i) obtain $Lock = q$ for some q at line 3, then they must have done so in a duration of length δ', as shown in Figure 4. Each process p_i then executes $delay(\Delta_{p_i}^-, \Delta_{p_i}^+)$ (line 16), and then enqueues itself onto *LocalQueue*[q] (line 17). Note that, by (1), the actual delay duration of each p_i differs by at least δ. Hence, by Property 1, and by choosing δ large enough, we can ensure that such processes finish their delay-statement executions at different times, so that accesses of *LocalQueue*[q] by these processes never overlap. Thus, each p_i may enqueue itself in a sequential operation, as shown in Figure 4. (Meanwhile, q may execute its critical section.)

In its exit section, process q executes $delay(\Delta_N^-, \Delta_N^+)$ (line 21), and thus ensures that every p_i (as defined above) has enqueued itself onto *LocalQueue*[q]. q then merges its queue with the global queue (line 22).

Note that a process cannot simply acquire *Lock* (by a successful *compare-and-swap* operation), execute its critical section, and then release *Lock* as in ALGORITHM DSM, because critical-section executions may take unbounded time and hence Property 1 would be violated. A possible alternative might be to release *Lock* before executing the critical section. However, this is clearly problematic, because the critical section would be unprotected by *Lock*.

To solve this problem, we use *Lock* to execute a queue-based mutual exclusion algorithm with $\Theta(1)$ RMR time complexity. Several such algorithms are known [13, 16, 22] that have the following structure.

```
while true do
    Noncritical Section;
    Enqueue_N(p);    /* enqueues p onto the waiting queue in Θ(1) steps; may use strong atomic primitives */
    Spin_N(p);                                                                      /* local spinning */
    Critical Section;
    Exit_N(p)
od
```

For example, Graunke and Thakkar's algorithm [16] can be implemented as follows, in which the shared variables *Tail* and *Slots* implement a waiting queue, which we will call the "GT queue."

shared variables
  Tail: $(0..N-1) \times$ **boolean initially** $(0, \mathit{false})$;
  Slots: **array**$[0..N-1]$ **of boolean initially** *true*

private variables
  tail: $(\perp, 0..N-1)$;
  bit: **boolean**

$\texttt{Enqueue}_N(p)$
  a: $(\mathit{tail}, \mathit{bit}) := \mathit{fetch\text{-}and\text{-}store}(\mathit{Tail}, (p, \mathit{Slots}[p]))$

$\texttt{Spin}_N(p)$
  b: **await** $\mathit{Slots}[\mathit{tail}] \neq \mathit{bit}$

$\texttt{Exit}_N(p)$
  c: $\mathit{bit} := \mathit{Slots}[p]$;
  d: $\mathit{Slots}[p] := \neg \mathit{bit}$

In Algorithm CC, these procedures are invoked by processes that have acquired *Lock* (lines 12, 14, and 23). Note we allow `Enqueue` to contain noncomparison primitives (such as *fetch-and-store* and *fetch-and-add*). Since executions of `Enqueue` (in Algorithm CC) are serialized by means of *Lock*, these primitives can be implemented with ordinary reads and writes.

Since q releases *Lock* at line 13, Property 1 is ensured. As shown in Figure 4, q enqueues itself onto the GT queue (line 12), waits for its predecessor (in the GT queue) to finish (line 14), and later signals its successor in the GT queue (line 23).

The two-process mutual exclusion algorithm and the barrier and promotion mechanisms are the same as in Algorithm DSM. In fact, the barrier mechanism can be considerably simplified in CC machines, as follows: `Wait` can be defined as "**await** *Flag*; *Flag* := *false*" and `Signal` as "*Flag* := *true*," where *Flag* is a shared boolean variable.

Due to the similarity between Algorithms DSM and CC, we omit a formal correctness proof for Algorithm CC. From this discussion, we have the following theorem.

Theorem 2 *The mutual exclusion problem can be solved with* $\Theta(1)$ *RMR time complexity on CC machines in the known-delay model, assuming delays can be upper bounded.* □

4 A $\Theta(\log \log N)$ Algorithm

In this section, we describe Algorithm T (for "tree"), 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. As a stepping stone toward our final algorithm, we first present a version with unbounded space, which is given in Figure 5.

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 [18, 27]. 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.

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 levels and 2^{2^k} leaves, as shown in Figure 6. We do not count leaves when counting the number of levels. (This structure is somewhat similar to the van Emde Boas tree [26], which implements a $\Theta(\log \log u)$-time search structure over a fixed set of integers $1..u$.) A tree of order zero is a two-process mutual exclusion algorithm, as shown

```
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;
    if (q = p) ∨ (q = ⊥) then
6:      next := Dequeue(WaitingQueue);
7:      Promoted := next;
8:      if next ≠ ⊥ then Spin[next] := true fi
    fi

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

procedure AccessTree(
    ptrT: pointer to TreeType,
    pos: 0..N-1)
14: k := ptrT -> order;
    if k = 0 then                   /* base case */
15:     ptrT -> waiter := p;
16:     ExecuteCS(1);
17:     return
    fi;
18: indx := ⌊pos/2^(2^(k-1))⌋;
    ptrL := ptrT -> lower[indx];
19: if CAS(ptrL -> winner, ⊥, p) ≠ ⊥ then
20:     AccessTree(ptrL, pos mod 2^(2^(k-1)));
                /* recurse into the lower component */
21:     return
    fi;
22: ptrU := ptrT -> upper;
23: if CAS(ptrU -> winner, ⊥, p) ≠ ⊥ then
24:     AccessTree(ptrU, indx)
                /* recurse into the upper component */
    else
25:     if ptrT = &T_0 then
26:         ExecuteCS(0)
        else
27:         ptrT -> waiter := p;
                /* p is now the primary waiter of T */
28:         ExecuteCS(1)
        fi;
        /* update the upper component */
29:     ptrC := GetCleanTree(k - 1);
30:     ptrT -> upper := ptrC;
31:     delay(Δ_0);
32:     Enqueue(WaitingQueue, ptrU -> waiter)
    fi;
    /* update the lower component */
33: ptrC := GetCleanTree(k - 1);
34: ptrT -> lower[indx] := ptrC;
35: delay(Δ_0);
36: Enqueue(WaitingQueue, ptrL -> waiter)
```

Figure 5: ALGORITHM T, unbounded space version. (Each private variable used in *AccessTree* is assumed to be on the call stack.)

in the figure. 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 $k-1$. 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 component of another constituent of T. Associated with each tree T is a field called *winner*, which is accessed by *compare-and-swap* operations. A process p attempts to establish $T.winner = p$ by invoking *compare-and-swap*, in which case it is said to have *acquired* T. The structure of an arbitration tree explained thus far is depicted in Figure 7. (The *waiter* field is explained later.)

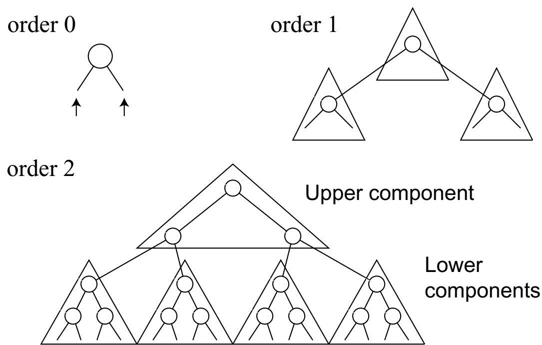

Figure 6: Structure of arbitration trees used in ALGORITHM T. A tree of order $k > 0$ has an upper component and $2^{2^{k-1}}$ lower components, each of order $k-1$.

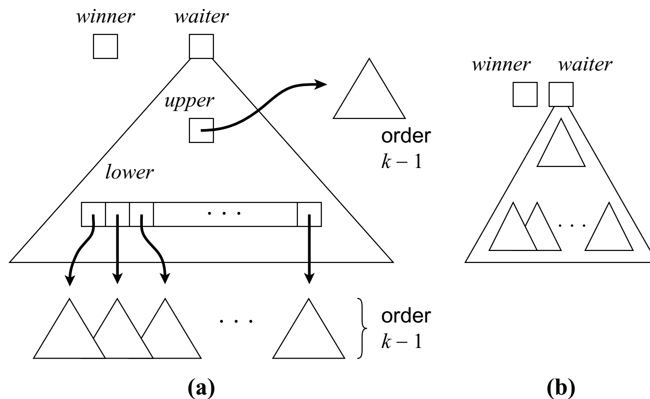

Figure 7: Structure of an arbitration tree of *TreeType*, of order k. **(a)** A "verbose" depiction, showing dynamic links for its components. **(b)** A simplified version.

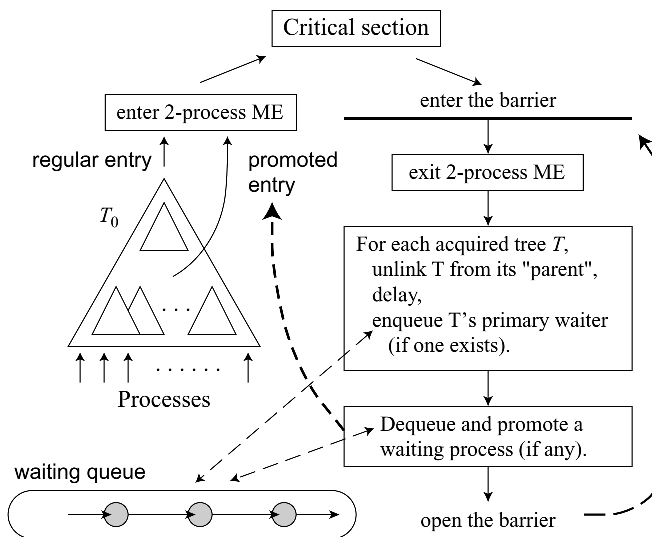

Figure 8: Overall structure of ALGORITHM T. Dashed lines represent information/signal flow.

Arbitration tree and waiting queue. We start with a high-level overview, as illustrated in Figure 8. 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 (namely, the one 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 *compare-and-swap* on $L_i.winner$, and then on $U.winner$). If p acquires both components, then it may enter its critical section. As explained shortly, p may also be "promoted" to its critical section after failing to acquire either tree. (In that case, p may have acquired only L_i, or neither L_i nor U.) 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. As before, a barrier mechanism (lines 4 and 12 in Figure 5) 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. The way in which *WaitingQueue* and *Promoted* are used (lines 5–8, 32, and 36) is the same as in previous algorithms.

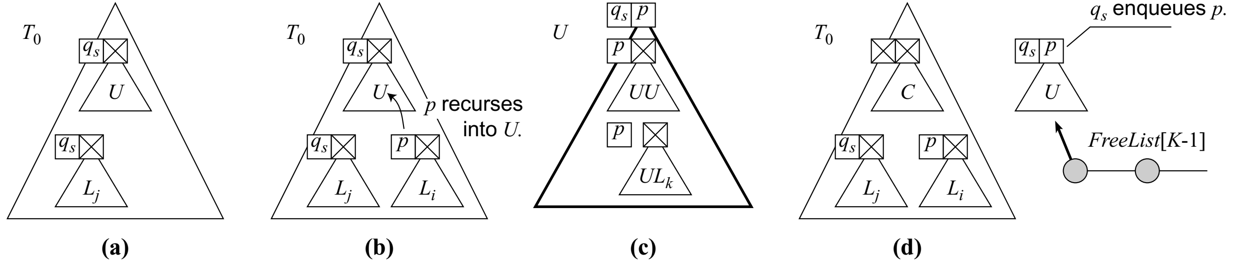

Figure 9: An example of recursive execution. The left-side boxes represent the *winner* field; the right-side ones, *waiter*. **(a)** A process q_s acquires both L_j and U and performs a regular (non-promoted) entry. **(b)** Process p acquires L_i, but fails to acquire U. **(c)** p recurses into U, acquires 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*. In the bounded space version (Figure 10), the old component U is now enqueued onto *FreeList*$[K-1]$, so that it can be reused later.

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 defined 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 Figure 9. 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 field *waiter* of S (Figure 9(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 the following.

Property 1 A process p that enters a tree S 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 first 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) first delays itself by $\Delta_0 = \Theta(\Delta)$ time, and then examines its path in order to discover other waiting processes (Figure 9(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 completes its delay. By Property 2, and because p has entered S, q_s's delay ensures that q_s indeed finds 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 Figure 9(d). (The variable $FreeList[K-1]$ shown in the figure pertains to the bounded space version, as explained shortly.) In the unbounded version (Figure 5), 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.) 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 Δ_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-Δ 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-Δ time complexity is incurred at T, and $\Theta(K) = \Theta(\log \log N)$ at T_0.

A bounded version. We now explain how to bound the space complexity of ALGORITHM T. As explained above, each process, in its exit section, may need to update a pointer to a component so that it points to a "clean" tree. (A tree is *clean* if it is properly initialized and no process is concurrently accessing it.)

In an unbounded version, we may assume that a previously unused tree always exists. However, this is impossible in the bounded version, illustrated in Figure 10, and hence we must "recycle" trees that have been used before. However, since trees are accessed in entry sections which execute asynchronously, it is difficult to ensure that a given tree T is not concurrently accessed by any other process. (Note that, if we poll every other process to ensure that it does not access T, then such a check would incur $\Theta(N)$ RMRs, which is clearly undesirable.) In order to facilitate this, we use variables *InUse* and *FreeList*, which keep track of trees that are possibly clean.

$FreeList[k]$ is a queue of trees of order k that are not currently linked from T_0, and is accessed by the usual *Enqueue* and *Dequeue* operations, plus the *MoveToTail* operation: if a tree S is in $FreeList[k]$, then $MoveToTail(FreeList[k], \&S)$ moves S to the end of the queue; otherwise, it does nothing. If these queues are implemented as a doubly-linked list, then each of these operations can be performed in $\Theta(1)$ time. We stress that these operations are invoked only in exit sections, and hence are executed sequentially, due to the barrier. When a process p accesses a particular tree T (of order k), then p marks T as being "in use" by establishing $InUse[p][s][j] = \&T$ for some j (lines 23 and 28 in Figure 10). Lines 3–5 are executed to ensure that no tree currently "in use" can propagate to the head of *FreeList*. In particular, if p is waiting for promotion while accessing S, then while it is waiting, S will be moved to the end of $FreeList[k]$ by every N^{th} critical section execution. A variable *Check* cycles through $0..N-1$ for this purpose. (This mechanism is taken from [11], and is also used in [5].)

Further details. Having explained the basic structure of the algorithm, we now present a more detailed overview. (In the rest of this section, all line numbers refer to Figure 10.)

A process p in its entry section first initializes its spin variable (line 1), and enters T_0 (line 2). Procedure $AccessTree(\&T, pos)$ executes a mutual exclusion algorithm at tree T, starting from position *pos* among its leaf nodes (indexed $0..2^{2^k}$, where k is the order of T). After returning from *AccessTree*, p moves "dirty" trees to the tail of *FreeList* as explained before (lines 3–5), and promotes a process from *WaitingQueue*, if one exists and if no other process is currently promoted (lines 8–11). Finally, it opens the barrier (line 7).

Procedure *ExecuteCS* is called to execute critical sections. As explained before, processes exiting T_0 form two groups: the promoted processes and the non-promoted processes (*i.e.*, those that successfully acquire the upper component of T_0). To arbitrate between these two groups, an additional two-process mutual exclusion algorithm is used. Promoted processes invoke $ExecuteCS(side)$ with $side = 1$, and other processes with $side = 0$. Inside *ExecuteCS*, a process first waits for promotion if *side* equals 1 (line 12), executes a two-process entry section (line 13), executes its critical section (line 14), and waits until the barrier is opened (line 15), so that exit-section executions are serialized. Finally, it executes the two-process exit section (line 16) and returns.

```
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);
    Check: 0..N-1;
    Spin: array[0..N-1] of boolean;
    Promoted: (⊥, 0..N-1) initially ⊥;
    FreeList: array[0..K-1] of queue of
        pointers to TreeType;
    InUse: array[0..N-1][0..K-1][1, 2] of
        pointer to TreeType;
    WaitingQueue: queue of {0..N-1}

initially
    WaitingQueue: empty;
    FreeList[k]: contains 4N + 1 clean trees
        of order k

process p ::    /* 0 ≤ p < N */

while true do
0:  Noncritical Section;

1:  Spin[p] := false;
2:  AccessTree(&T_0, p);

/* move "dirty" trees to tail */
3:  ptr := Check;
    for k := 0 to K-1 do
        for j := 1 to 2 do
4:          MoveToTail(FreeList[k], InUse[ptr][k][j])
    od od;
5:  Check := ptr + 1 mod N;

6:  TryToPromote();
7:  Signal()                    /* open the barrier */
od

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

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

procedure AccessTree(
    ptrT: pointer to TreeType,
    pos: 0..N-1)

17: k := ptrT -> order;
    if k = 0 then                       /* base case */
18:     ptrT -> waiter := p;
19:     ExecuteCS(1);
20:     ptrT -> waiter := ⊥;
21:     return
    fi;

22: indx := ⌊pos/2^(2^(k-1))⌋;
    ptrL := ptrT -> lower[indx];
23: InUse[p][k-1][1] := ptrL;

24: if CAS(ptrL -> winner, ⊥, p) ≠ ⊥ then
25:     AccessTree(ptrL, pos mod 2^(2^(k-1)));
                /* recurse into the lower component */
26:     return
    fi;

27: ptrU := ptrT -> upper;
28: InUse[p][k-1][2] := ptrU;

29: if CAS(ptrU -> winner, ⊥, p) ≠ ⊥ then
30:     AccessTree(ptrU, indx)
                /* recurse into the upper component */
    else
31:     if ptrT = &T_0 then
32:         ExecuteCS(0)
        else
33:         ptrT -> waiter := p;
                /* p is now the primary waiter of T */
34:         ExecuteCS(1)
        fi;

        /* update the upper component */
35:     ptrC := Dequeue(FreeList[k-1]);
36:     ptrT -> upper := ptrC;
37:     delay(Δ_0);
38:     Enqueue(WaitingQueue, ptrU -> waiter);
39:     ptrU -> winner := ⊥;
40:     Enqueue(FreeList[k-1], ptrU)
    fi;

    /* update the lower component */
41: ptrC := Dequeue(FreeList[k-1]);
42: ptrT -> lower[indx] := ptrC;
43: delay(Δ_0);
44: Enqueue(WaitingQueue, ptrL -> waiter);
45: ptrL -> winner := ⊥;
46: Enqueue(FreeList[k-1], ptrL)
```

Figure 10: ALGORITHM T, bounded space version.

We now explain *AccessTree*(&T, *pos*) in detail. Lines 18–21 implement the base case in which T has order zero. In this case, T is a single binary node. Actually, T is simpler than a two-process mutual exclusion, because p may enter T only if some other process q has already acquired T. Since T may be concurrently accessed by at most two processes, p is the sole waiter of T.

Thus, p designates itself as the primary waiter of T (line 18), invokes *ExecuteCS* as a promoted process, resets *ptrT* -> *waiter* (line 20), and returns (line 21).

If $k > 0$, then p first tries to acquire its lower component L of T (line 24). If p fails, then it recurses into L (line 25). Otherwise, p tries to acquire the upper component U as well (line 29). If it fails, then it recurses into U (line 30). Otherwise, there are two cases to consider. If T equals T_0, then p invokes *ExecuteCS* as a non-promoted process (line 32); otherwise, p designates itself as T's primary waiter (line 33), and invokes *ExecuteCS* as a promoted process (line 34).

In its exit section, p replaces the components it has acquired with clean trees (lines 35–40 for U, and lines 41–46 for L). We only explain lines 35–40 here; the other case is similar. (See also Figure 9(d).) First, p dequeues a clean tree of order $k-1$ (line 35), updates T's *upper* pointer (line 36), and delays itself by Δ_0 (line 37). Δ_0 is chosen to satisfy the following two properties.

Property 4 For each tree S, if a process q establishes $q.ptrL := \&S$ at line 22 while $S.winner \neq \bot$ holds, then some process r (not necessarily q) becomes the primary waiter of S by executing line 18 or 33 of *AccessTree*(&$S, \ldots$) in Δ_0 time.

Property 5 For each tree S, if a process q establishes $q.ptrU := \&S$ at line 27 while $S.winner \neq \bot$ holds, then some process r (not necessarily q) becomes the primary waiter of S by executing line 18 or 33 of *AccessTree*(&$S, \ldots$) in Δ_0 time.

By Property 2, we have $\Delta_0 = \Theta(\Delta)$. Therefore, considering again lines 35–40, if a process has entered U, then p enqueues its primary waiter at line 38. It then resets *U.winner* (line 39), and enqueues U onto *FreeList* so that it may be reused later (line 40).

Correctness Proof. We now formally prove the correctness of ALGORITHM T. The following is the list of invariants needed to prove the correctness of ALGORITHM T. Informally speaking, invariant (I1) implies that only the winner of the upper component (of a tree T) may exchange the upper component (by executing line 36). Invariant (I2) implies that the identity of such a winner is indicated by the *winner* field. Invariant (I3) states that the identity of a promoted process is indicated by *Promoted*. (In particular, there exists at most one promoted process.) Invariant (I4) and (I5) state the mutual exclusion requirement and the correctness of the barrier, respectively.

invariant $p@\{12..16, 31..36\} \wedge p.ptrT \texttt{->} order > 0 \Rightarrow p.ptrU = p.ptrT \texttt{->} upper$ (I1)

invariant $p@\{12..16, 31..39\} \wedge p.ptrT \texttt{->} order > 0 \Rightarrow p.ptrU \texttt{->} winner = p$ (I2)

invariant $p@\{2..6, 8..10, 12..46\} \wedge Spin[p] = true \Rightarrow Promoted = p$ (I3)

invariant **(Mutual exclusion)** $\left|\{p\colon p@\{14..16\}\}\right| \leq 1$ (I4)

invariant $\left|\{p\colon p@\{3..11, 16, 20, 21, 26, 35..46\}\}\right| \leq 1$ (I5)

We now prove that each of (I1)–(I5) is an invariant. As before, for each invariant I, we prove that for any pair of consecutive states t and u, if all invariants hold at t, then I holds at u. (It is easy to see that each invariant is initially true, so we leave this part of the proof to the reader.)

invariant $p@\{12..16, 31..36\} \wedge p.ptrT \texttt{->} order > 0 \Rightarrow p.ptrU = p.ptrT \texttt{->} upper$ (I1)

Proof: The only statement that may establish the antecedent is 29.p, which may do so only if executed when $p.ptrU \texttt{->} winner = \bot$ holds. Note that p has established the consequent by executing statement 27.p. Therefore, 29.p may falsify (I1) only if some process q has changed $p.ptrT \texttt{->} upper$ (by executing 36.q while $q.ptrT = p.ptrT$ holds). However, for this to happen, q must have established $q.ptrU \texttt{->} winner = q$ by

executing 29.q. Moreover, $q.ptrU = q.ptrT$ -> $upper$ holds. (Otherwise, q would have violated (I1).) It follows that p reads $p.ptrU$ -> $winner = q$ at line 29 in this case, and hence cannot establish the antecedent.

The only statement that may falsify the consequent is 36.q, where q is any arbitrary process. Statement 36.q may falsify the consequent only if executed when $q.ptrT = p.ptrT$ holds. However, by applying (I1) and (I2) to q, we have $q.ptrT$ -> $upper$ -> $winner = q$. Similarly, the antecedent of (I1) implies (by (I2)) $p.ptrT$ -> $upper$ -> $winner = p$.

It follows that 36.q may falsify (I1) only if $q = p$ holds. However, in this case, 36.q falsifies the antecedent, and thus preserves (I1). □

invariant $p@\{12..16, 31..39\} \wedge p.ptrT$ -> $order > 0 \Rightarrow p.ptrU$ -> $winner = p$ (I2)

Proof: The only statement that may establish the antecedent is 29.p, which establishes the consequent. The only statements that may falsify the consequent are 39.q and 45.q, where q is any arbitrary process. However, the structure of the algorithm implies that a tree cannot simultaneously be the upper component of some tree *and* the lower component of some tree. Thus, 45.q cannot falsify the consequent.

Statement 39.q may falsify the consequent only if executed when $q.ptrU = p.ptrU$ holds. However, by applying (I2) to p and q, we have $p.ptrU$ -> $winner = p$ and $q.ptrU$ -> $winner = q$, and hence $p = q$ holds. Thus, 39.q falsifies the antecedent in this case. □

invariant $p@\{2..6, 8..10, 12..46\} \wedge Spin[p] = true \Rightarrow Promoted = p$ (I3)

Proof: The only statement that may establish $p@\{2..6, 8..10, 12..46\}$ is 1.p. However, the antecedent is false after the execution of 1.p. The only statement that may establish $Spin[p] = true$ is 11.q, where q is any arbitrary process. In this case, q has established the consequent by executing 10.q. Note that *Promoted* cannot change between the executions of 10.q and 11.q, due to the barrier mechanism (I5).

The only statement that may falsify the consequent is 10.q. However, due to the barrier mechanism, this may happen only if 8.q is executed while $Promoted = p$ holds, in which case 10.q may be executed only if $q = p$. Thus, 10.q falsifies the antecedent in this case. □

invariant (Mutual exclusion) $\left|\{p\colon p@\{14..16\}\}\right| \leq 1$ (I4)

Proof: A process p may execute `Entry`$_2(0)$ and `Exit`$_2(0)$ only if it executes lines 31 and 32. By (I1) and (I2), this implies T_0 -> $upper$ -> $winner = p$. Therefore, there exists at most one non-promoted process that may execute `Entry`$_2(0)$ and `Exit`$_2(0)$. Similarly, by (I3), there exists at most one promoted process that may execute `Entry`$_2(1)$ and `Exit`$_2(1)$. Because the `Entry` and `Exit` routines are assumed to be correct, we have (I4). □

invariant $\left|\{p\colon p@\{3..11, 16, 20, 21, 26, 35..46\}\}\right| \leq 1$ (I5)

Proof: By (I4), there exists at most one process that may invoke `Wait` (line 15). Similarly, while (I5) holds, there exists at most one process that may invoke `Signal` (line 7). Because the `Wait` and `Signal` routines are assumed to be correct (under this condition), we have (I5). □

By (I4), we have mutual exclusion. As explained before, Property 1 ensures that a process p (unless it wins the outermost instance of the mutual exclusion algorithm) eventually becomes a primary waiter of some tree S. By inductively applying Properties 2 and 3 over the order of trees, it easily follows that some process eventually enqueues p onto *WaitingQueue*. Thus, p eventually enters its critical section.

The space complexity of Algorithm T is $O(N^3 \log\log N)$ if we take into account adding dummy processes to ensure $N = 2^{2^K}$. From Algorithm T, we have the following theorem.

Theorem 3 *The mutual exclusion problem can be solved with* $\Theta(\log\log N)$ *RMR-*Δ *(or RMR) time complexity on CC or DSM machines in the known-delay model.* □

5 Lower Bound: System Model

In this section, we present the model of a shared-memory system used in our lower-bound proof. Our system model is similar to that used in [6, 12].

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 finite 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 Op the *operation* of event e, denoted $op(e)$. 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 α from v. We call e_p a *read event*.
- $e_p = [p, \mathsf{write}(v), \alpha]$. In this case, e_p writes the value α to v. We call e_p a *write event*.
- $e_p = [p, \mathsf{delay}]$. In this case, e_p delays p by a fixed amount Δ, defined so that each event execution finishes in Δ time. We call e_p a *delay event*. □

We also say that e_p *accesses* v if it writes or reads v. 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 $writer_event(v, H)$,[4] 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 $writer(v, H) = \perp$ and $writer_event(v, H) = \perp$. The execution time of the last event of H is denoted $last(H)$.

Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes.) An *initial value* is associated with each variable. An event is *local* if it accesses a local variable, and *remote* if it accesses a remote variable. An event is *nonlocal* if it is either a remote event or a delay event.

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.[5] If G is a subcomputation of H, then $H - G$ is the computation obtained by removing all events in G from H. Computations H and G are *equivalent with respect to* Y if and only if $H \mid Y = G \mid Y$. A computation H is a *Y-computation* if and only if $H = H \mid Y$. For simplicity, we abbreviate these definitions when applied to a singleton set of processes (*e.g.*, $H \mid p$ instead of $H \mid \{p\}$).

Mutual exclusion systems. We now define 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 satisfies 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. The allowable transitions of $stat_p$ are as follows: for all $H \in C$,

[4] Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other. (For simplicity, we do not extend our notion of an event to include an additional identifier for distinguishability.)

[5] $H \mid Y$ and $H \circ G$ are not necessarily valid computations in a given system $\mathcal{S}$, *i.e.*, elements of C. However, we can always consider them as computations in a technical sense, *i.e.*, each is a sequence of events.

$H \circ \langle (Enter_p, \exists t)\rangle \in C$ if and only if $value(stat_p, H) = ncs$;
$H \circ \langle (CS_p, \exists t)\rangle \in C$ if $value(stat_p, H) = entry$;
$H \circ \langle (Exit_p, \exists t)\rangle \in C$ if $value(stat_p, H) = exit$.

We also say that a process is *active* if it is not in its noncritical section:

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

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

Exclusion: For all $H \in C$, if both $H \circ \langle (CS_p, t)\rangle \in C$ and $H \circ \langle (CS_q, t')\rangle \in C$ hold (for some t and t'), then $p = q$.

— *Informally, 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 traffic. In order to apply our lower bound to such systems, we do not count every read/write event, but only critical events, as defined below.

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 $writer(v, F) \neq p$. □

Definition: We say that an event e_p is *critical* if and only if 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. □

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 a write event of v by p is a cache-miss event if **(i)** it is the first write of v by p, or **(ii)** it is the first write of v by p after some other process has written v.

Also, if p both reads and writes v, then both its first read of v and first write of v are considered critical. Depending on the system implementation, the latter of these two events might not generate a cache miss. However, even in such a case, the first such event will always generate a cache miss, and hence at least half of all such critical reads and writes will actually incur real global traffic. Hence, our lower bound remains asymptotically unchanged for such systems.

In a *write-through* cache scheme, writes always generate a cache miss. On the other hand, with a *write-back* scheme, a remote write to a variable v may create a cached copy of v, so that subsequent writes to v do not cause cache misses. In the definition above, if e_p is not the first write to v by p, then it is considered a cache-miss event only if $writer(v, F) = q \neq p$ holds, which implies that v is stored in the local cache of another process q.[6] In such a case, e_p either invalidates or updates the cached copy of v (depending on the system), thereby generating global traffic.

As explained later, the definition of cache-miss events given above is too narrow to Theorem 5, a lower bound that pertains to the RMR measure. Thus, in order to prove Theorem 5, we need to slightly broaden our definition of a cache-miss event, as shown in Section 6.

[6] Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable. (Note that $writer(v, F) = q$ implies that q's cached copy of v has not be invalidated.) Cache size and associativity limitations can only *increase* the number of cache misses.

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 $t \le t'$ holds.

T2: For any timed event (e_p, t) in H, if $e_p \ne \mathit{Exit}_p$ and if $\mathit{last}(H) > t + \Delta$, 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:

$$\begin{cases} t' = t + \Delta, & \text{if } e_p \text{ is a delay event,} \\ t + \Delta_c \le t' \le t + \Delta, & \text{if } e_p \text{ is a cache-miss event,} \\ t \le t' \le t + \Delta, & \text{otherwise,} \end{cases}$$

where Δ_c is a lower bound (less than Δ) 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). Thus, in such systems (which has a known lower bound on the execution time of a noncritical event), the RMR measure and the RMR-Δ measure become identical.

Note that we allow noncritical events to take zero duration. 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. (As noted earlier in footnote 2, events occur *instantaneously* in our model; durations are enforced by properly spacing events apart. Whenever two events occur at the same instant, their effect is determined by their relative order in the computation under consideration.)

On the other hand, cache-miss events take some duration between Δ_c and Δ, 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 Δ. Thus, the issue of whether delays are upper bounded does not arise in our proof.

We say that two computations H and G are *congruent*, denoted $H \sim G$, if they have the same sequence of events and only the timings differ. (That is, we can write $H = \langle (e^1, t^1),\ (e^2, t^2),\ \ldots \rangle$ and $G = \langle (e^1, u^1),\ (e^2, u^2),\ \ldots \rangle$).

The properties below apply to any mutual exclusion system.

P1: If $H \in C$ and G is a prefix of H, then $G \in C$.

— *Informally, every prefix of a valid computation is also a valid computation.*

P2: Assume that $H \circ \langle (e_p, t) \rangle \in C$, $G \in C$, $G \mid p \sim H \mid p$, and the following: either e_p is not a read event, or e_p reads v and $\mathit{value}(v, G)$ equals $\mathit{value}(v, H)$. Then, $G \circ \langle (e_p, t') \rangle \in C$ holds, provided that $G \circ \langle (e_p, t') \rangle$ satisfies T1–T3.

— *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e_p after H, and if a variable read by e_p (if any), after G, has the same value as after H, then p can execute e_p after G.*

P3: If $H \circ \langle (e_p, t) \rangle \in C$, $G \in C$, and $G \mid p \sim H \mid p$, then $G \circ \langle (e'_p, t') \rangle \in C$ for some event e'_p such that $op(e_p) = op(e'_p)$.

— *Informally, if two computations H and G are not distinguishable to process p, and if p can execute event e after H, then p can execute the same kind of operation after G. (Note that the values read or written might be different.)*

P4: For any $H \in C$ and a read event $e_p = [p, \mathsf{read}(v), \alpha]$ of v, $H \circ \langle (e_p, t) \rangle \in C$ implies $value(v, H) = \alpha$.

— *Informally, only the last value written to a variable may be read.*

P5: If $H \in C$, $H \sim G$, and G satisfies T1–T3, then $G \in C$.

— *Informally, we can change a computation's timing, as long as T1–T3 are satisfied.*

6 Lower Bound: Proof Sketch

In the appendix, we show that for any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation H such that some process p executes $\Omega(\log \log N)$ critical events to enter and exit its critical section, where $N = |P|$. In this section, we sketch the key ideas of the proof.

The 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 its entry, exit, or noncritical 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 mth 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 will be the case only if some variable v exists that is written 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, the desired regular computation H_{m+1} can be constructed.

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, 6, 14, 19, 24]. We now define the regularity condition.

A regular computation must meet certain conditions, which are needed for the induction to continue. These conditions are given as R1–R4 in the following definition. Informally, R1 ensures that active processes have no knowledge of each other; R2 bounds the number of possible conflicts caused by appending a critical event; R3 ensures that the active and finished processes behave as explained above; R4 ensures that erasing an invisible process preserve criticality of events.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system. We say that $H \in C$ is *regular* if and only if there exist two disjoint sets of processes, $\mathrm{Inv}(H)$, the set of *invisible processes*, and $\mathrm{Vis}(H)$, the set of *visible processes*, satisfying

$$\mathrm{Inv}(H) \cup \mathrm{Vis}(H) = \{p \in P \colon H \mid p \neq \langle\rangle\},$$

such that the following conditions hold.

R1: Assume that H can be written as $E \circ \langle (e_p, t) \rangle \circ F \circ \langle (f_q, t') \rangle \circ G$, and consider a variable $v \in V$. If e_p writes v, f_q reads v, and F does not contain a write to v (*i.e.*, $\mathit{writer_event}(v, F) = \perp$), then $p \in \mathrm{Vis}(H)$.
— *Informally, if a process p writes to a variable v, and if another process q reads that value from v, then p is visible.*

R2: Consider a variable $v \in V$ and two different events e_p and f_q in H. Assume that both p and q are in $\mathrm{Inv}(H)$, $p \neq q$, both e_p and f_q accesses v, and there exists a write to v in H. Then, $\mathit{writer}(v, H) \in \mathrm{Vis}(H)$ holds.
— *Informally, if a variable v is accessed by more than one process in* $\mathrm{Inv}(H)$, *then the last process in H to write to v (if any) belongs to* $\mathrm{Vis}(H)$.

R3: For any process $p \in \mathrm{Inv}(H)$, $\mathit{value}(\mathit{stat}_p, H) = \mathit{entry}$ holds.
— *Informally, every invisible process is in its entry section.*

R4: Assume that H can be written as $E \circ \langle (e_p, t) \rangle \circ F \circ \langle (f_p, t') \rangle \circ G$, for some $p \in \mathrm{Inv}(H)$, in which both e_p and f_p accesses a variable v. If a process $q \neq p$ writes v in F, then some process $r \in \mathrm{Vis}(H)$ writes v in F.
— *Informally, 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 in between. This condition is used to show that the property of being a critical event is preserved after erasing invisible processes.* □

Detailed proof overview. Initially, we start with a regular computation H_1, where $\mathrm{Inv}(H_1) = P$, $\mathrm{Vis}(H_1) = \{\}$, 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 mth induction step, we consider a computation $H = H_m$ in which each process in $\mathrm{Inv}(H)$ executes m critical events, $\mathrm{Vis}(H)$ consists of at most m processes, and each active process executes its last event at the same time $t = t_m$. Furthermore, we assume

$$\log\log n = \Theta(\log\log N) \quad \text{and} \quad m = O(\log\log N), \tag{2}$$

where $n = |\mathrm{Inv}(H)|$. We construct a regular computation $G = H_{m+1}$ such that $\mathrm{Act}(G)$ consists of $\Omega(\sqrt{n}/(\log\log N)^2)$ processes, each of which executes $m+1$ critical events in G. (To see why (2) 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 $p \in \mathrm{Inv}(H)$ is *blocked after H* if and only if 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. These cases are depicted in Figs. 11 and 12, respectively.

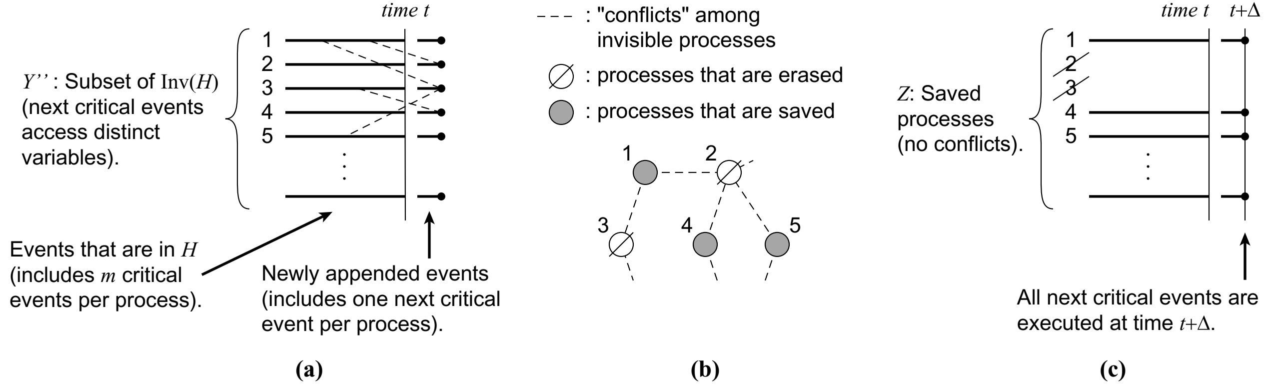

Figure 11: Erasing strategy. Here and in later figures, black circles (•) respresent critical events.

Case 2: Erasing strategy. 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 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 (2), 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 (Figure 11(a)). We now eliminate remaining possible conflicts among processes in Y'' by constructing a "conflict graph" $\mathcal{G}$ as follows (see Figure 11(b)).

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, it is clear that each process generates at most m edges. By applying Turán's theorem (Theorem 6 in the appendix), we can find a subset $Z \subseteq Y''$ with no conflicts such that $|Z| = \Omega(\sqrt{n}/m)$. By retaining Z and erasing all other invisible processes, we can eliminate all conflicts (Figure 11(c)) 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, there exists a variable v that is accessed by the next critical events of $\Omega(\sqrt{n})$ processes. 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 at time $t + \Delta$, as shown in Figure 12. In this way, the only information flow among processes in Y_v is that from the "last writer" of v, denoted $p_{\rm LW}$, to any subsequent reader (of v). We then add $p_{\rm LW}$ to the set of visible processes, *i.e.*, $p_{\rm LW} \in \mathrm{Vis}(G)$ 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 conflict with at most one invisible process, so we additionally erase at most m invisible processes, which is $o(\sqrt{n}/m)$, by (2).

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

Lemma 1 *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 and is unblocked after 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: $|\mathrm{Inv}(G)| = \Omega(\sqrt{n}/m)$, each $p \in \mathrm{Inv}(G)$ has exactly $m+1$ critical events in G, $|\mathrm{Vis}(G)| \leq m+1$, and each active process in G executes its last event at time $t + \Delta$. □

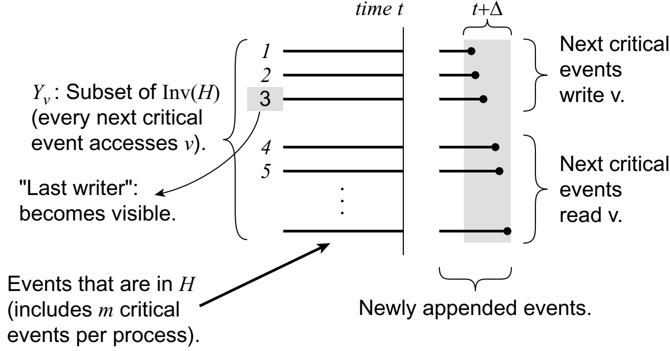

Figure 12: Covering strategy. Here, process 3 becomes p_{LW} and is added to $\mathrm{Vis}(G)$. Time instant $t + \Delta$ is shown expanded.

Finding unblocked processes. We now explain how we can find "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 Δ. By extending F in such a way until all visible processes finish execution, we have an extension $F \circ D$, which can be decomposed into $F \circ D_1 \circ D_2 \circ \cdots \circ D_{k'}$, 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 Δ. Continuing in this way, we can define 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), E is decomposed into at most $m \log\log N + 1$ segments, *i.e.*, $k \le m \log\log N + 1$.

For each E_j, starting with E_1, we do the following to find n' invisible processes that are unblocked simultaneously, where $n' = n/(k+1) - m$. We first append the *noncritical* events of E_j. It can be shown that noncritical events do not cause information flow. After that, we determine how many invisible processes are unblocked. If n' or more processes are unblocked, then we can erase all blocked invisible processes and construct G by applying Lemma 1. (This situation is depicted in Figure 13.) 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 Figure 13.) 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 flow. 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 find n' invisible unblocked processes (after appending $E_1 \circ \cdots \circ E_j$, for some $0 \le j \le k$), 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 (2) 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*

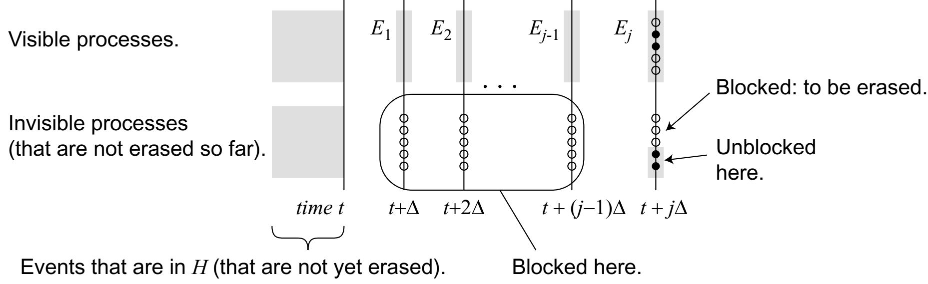

Figure 13: Finding unblocked processes. In this figure, white circles (∘) respresent noncritical events, and shaded boxes represent computations made of possibly many events.

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

By applying Lemma 2 inductively, we have the following.

Theorem 4 *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation in which a process incurs $\Omega(\log\log N)$ RMR-Δ time complexity in order to enter and then exit its critical section.* □

We now explain sketch the key ideas for adapting the preceding system model and proof for the RMR time complexity measure with unbounded delays. Three changes are required. First, in order to compute RMR time complexity, we count only non-delay critical events.

Second, consider a regular computation H_m, in which "most" unblocked invisible processes execute delay events as their next critical events. Since delays do not contribute to the RMR measure, appending these delay events does not suffice. In this case, we first erase all other invisible processes, and then let the remaining invisible processes delay until all visible processes finish execution (which is possible since delays are not bounded). Some invisible processes may have to be erased during the procedure, since visible processes may execute critical events that cause conflicts.

After all visible processes have finished execution, each of the remaining (delayed) invisible processes has no knowledge of any other active process, and hence it can be shown that each of them (except at most one) must execute a non-delay critical event before entering its critical section. By appending these non-delay critical events, we can construct H_{m+1}.

Third, consider a regular computation H_m, in which most visible processes are blocked. Since delays do not contribute to RMR time complexity, visible processes may execute an unbounded number of further delay events. Consider the extension E of H_m (in the proof of Lemma 6) obtained by making visible processes execute further events, decomposed into k segments of duration Δ each. E may contain "noncritical-delay" segments in which visible processes execute only noncritical and delay events. Since they contain delay events, they cannot be merged with neighboring segments. Moreover, since they do not contribute to RMR time complexity, their number is unbounded.

Thus, in the worst case, we may have $k = \Theta(n)$. Moreover, noncritical events by visible processes may "wake up" invisible processes one by one. For example, suppose that invisible processes are indexed as p_i, and assume that each p_i is executing a local-spin loop **while** $v \neq p_i$ **do od**. Furthermore, assume that a visible process q is the only writer of v, and that q has already written to v. In this case, in each segment E_i, q may execute a *noncritical* write of v, by writing $v := p_i$, and then execute a delay event. Thus, after each segment E_i, exactly one invisible process is unblocked, and the induction fails.

The main problem here is that q's writes of v are considered noncritical (except for the first), even though they are preceded by reads of v by invisible processes. However, in any realistic CC system, each invisible process creates a local copy of v by reading it, and hence q must either invalidate or update these cached copies. Thus, we must slightly broaden the definition of cache-miss events:

Definition: Let e_p be an event in $H \in C$. Then, we can write H as $F \circ \langle e_p \rangle \circ \cdots$, 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 *the last process to access v in H (if any) is not p*. □

With this new definition, the scenario described above cannot happen. In particular, if we encounter a "noncritical-delay" segment E_j during the procedure of Figure 15, then we can ensure that any invisible process that is blocked before E_j is also blocked after E_j, and hence we can append E_j without erasing any invisible processes.

In order to allow unbounded delays, Conditions T2 and T3 must be changed as follows.

T2: For any timed event (e_p, t) in H, if $e_p \neq Exit_p$, e_p is not a delay event, and $last(H) > t + \Delta$, 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:

$$\begin{cases} t' \geq t + \Delta, & \text{if } e_p \text{ is a delay event,} \\ t + \Delta_c \leq t' \leq t + \Delta, & \text{if } e_p \text{ is a cache-miss event,} \\ t \leq t' \leq t + \Delta, & \text{otherwise,} \end{cases}$$

where Δ_c is a lower bound (less than Δ) on the duration of a cache-miss event.

With these changes, we have the following theorem.

Theorem 5 *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.* □

7 Concluding Remarks

To the best of our knowledge, this paper is the first work on timing-based mutual exclusion algorithms in which all busy-waiting is by local spinning. Our specific interest has been to determine whether lower RMR time complexity is possible 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 efficient 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. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, October 2000.

[6] 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.

[7] J. Anderson and Y.-J. Kim. A new fast-path mechanism for mutual exclusion. *Distributed Computing*, 14(1):17–29, January 2001.

[8] J. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12. ACM, July 2002.

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

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

[11] J. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM, August 1995.

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

[13] 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.

[14] 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.

[15] 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.

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

[17] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

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

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

[26] 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.

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

Appendix: Detailed Lower-bound Proofs

In this appendix, we present a detailed proof of Theorem 4, and explain in detail how to adapt the proof to establish Theorem 5. Throughout the rest of this appendix, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$. Moreover, we assume the following.

- For each computation H in C, if H contains at most one $Enter_p$, then p executes at most $(\log \log N)$ critical events in H. (3)

We can assume (3) because otherwise our lower bound is already proved. We now establish several lemmas needed to establish Theorem 4. The first lemma states that we can safely "erase" any active process.

Lemma 3 *Consider a regular computation H in C. For any set Y of processes such that $\mathrm{Vis}(H) \subseteq Y$, the following hold: $H \mid Y \in C$, $H \mid Y$ is regular, and $\mathrm{Vis}(H \mid Y) = \mathrm{Vis}(H)$. Moreover, for each event e_p in $H \mid Y$,* **(i)** *if e_p is critical in $H \mid Y$, then it is also critical in H, and* **(ii)** *if e_p is critical in H and if $p \in \mathrm{Inv}(H)$, then it is also critical in $H \mid Y$.*

Proof: By the definition of a critical event, (i) is straightforward. Therefore, if an event e_p in $H \mid Y$ is noncritical in H, then it is also noncritical in $H \mid Y$. It follows that $H \mid Y$ satisfies T1–T3. Moreover, since H satisfies R1, if a process p is in $\mathrm{Inv}(H)$, then no process other than p reads a value written by p. Therefore, $H \mid Y \in C$. The conditions R1–R4 can be individually checked to hold in $H \mid Y$, which implies that $H \mid Y$ is regular and $\mathrm{Vis}(H \mid Y) = \mathrm{Vis}(H)$ holds.

To show (ii), consider an event e_p in $H \mid Y$, where $p \in \mathrm{Inv}(H)$. Transition events, critical reads, and delay events are straightforward, because their definitions depend only on the previous events of the same process, which are identical in both H and $H \mid Y$. For critical writes, the only problematic case is as follows: e_p writes v, f_p is the last write of v by p before e_p in H, and e_p is critical in H because of a write to v between f_p and e_p by another process that does not participate in $H \mid Y$ (and hence does not write to v in $H \mid Y$). However, R4 ensures that in such a case there exists some process in $\mathrm{Vis}(H)$ that writes to v between f_p and e_p, and hence e is also critical in $H \mid Y$. □

The next lemma gives us means for extending a computation by appending noncritical events by invisible processes.

Lemma 4 *Consider a regular computation H in C, and a set of processes $Y = \{p_1, p_2, \ldots, p_k\}$, where $Y \subseteq \mathrm{Inv}(H)$. Assume that for each p_j in Y, $last(H \mid p_j) = t$ (for some fixed t), and there exists a p_j-computation L_{p_j}, such that $H \circ L_{p_j} \in C$ and L_{p_j} has no critical events in $H \circ L_{p_j}$. Moreover, assume $t \leq last(H) \leq t + \Delta$.*

Then, there exists a computation L such that $L \sim L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_k}$, satisfying the following: $H \circ L \in C$, $H \circ L$ is regular, $\mathrm{Vis}(H \circ L) = \mathrm{Vis}(H)$, L contains no critical events in $H \circ L$, and every event in L is executed at time $t + \Delta$.

Proof: Since noncritical events can be executed arbitrarily fast, we can change event timings in L_1, L_2, $\ldots$, L_k and construct L such that every event is execute at time $t + \Delta$. It can be easily shown that $H \circ L$ satisfies T1–T3, provided L contains no critical events in $H \circ L$. (Regardless of whether the last event by p_j in H is a delay event, a cache-miss event, or a noncritical event, the first event by p_j in L is executed late enough to satisfy T3.)

We now show that each event e_p in L (for some $p \in Y$) is noncritical, by induction on the number of events in L. As in the proof of Lemma 3, it suffices to consider the case in which e_p writes v and is critical in $H \circ L$ because of a write event f_q of v (where $q \neq p$) prior to e_p in $H \circ L$. Since e_p is noncritical in $H \circ L_p$, we have $writer(v, H) = p$. By induction, either f_q is in H, or f_q is in L and is noncritical. In the latter case, by definition, q writes v in H. Therefore, in either case, q writes v in H. However, since $writer(v, H) = p$, by applying R2, we have $p \in \mathrm{Vis}(H)$, a contradiction.

Since L contains no critical events, it can be easily shown that events in L do not cause information flow among processes in Y, and hence $H \circ L$ is in C. The conditions R1–R4 can be individually checked to hold in $H \circ L$, which implies that $H \circ L$ is regular and $\mathrm{Vis}(H \circ L) = \mathrm{Vis}(H)$ holds. □

The next theorem by Turán [25] will be used in proving Lemma 5.

Theorem 6 (Turán) *Let $\mathcal{G} = (V, E)$ be an undirected graph, where V is a set of vertices and E is a set of edges. If the average degree of $\mathcal{G}$ is d, then there exists an independent set*[7] *with at least $\lceil |V|/(d+1) \rceil$ vertices.* □

The following lemma is a formal version of Lemma 1: it extends a regular computation, provided that it has enough unblocked invisible processes.

Lemma 5 *Let H be a regular computation. Define $n = |\mathrm{Inv}(H)|$, and assume the following.*

- *for each process $p \in \mathrm{Inv}(H)$,*
  - *p executes exactly m critical events in H,* (4)
  - *p is unblocked after H, and* (5)
  - *$last(H \mid p) = t$, for some fixed t;* (6)
- *$n > (m \log\log N)^2$;* (7)
- *$|\mathrm{Vis}(H)| \leq m$;* (8)
- *$last(H) \leq t + \Delta$;* (9)
- *for each $p \in \mathrm{Act}(H) \cap \mathrm{Vis}(H)$, either $last(H \mid p) = t$, or $t \leq last(H \mid p) \leq t + \Delta$ and the last event by p in H is noncritical.* (10)

Then, there exist a regular computation G in C such that

- *each process in $\mathrm{Inv}(G)$ executes exactly $m+1$ critical events in G.* (11)
- *$|\mathrm{Inv}(G)| \geq \dfrac{\sqrt{n} - m \log\log N}{2m+1} - m$;* (12)
- *$|\mathrm{Vis}(G)| \leq m + 1$;* (13)
- *$last(G) = t + \Delta$;* (14)
- *for each $p \in \mathrm{Act}(G)$, $last(G \mid p) = t + \Delta$.* (15)

Proof: For each $p \in \mathrm{Inv}(H)$, by (5), there exist a p-computation L and p's "next critical event" e_p such that

- $H \circ L_p \circ \langle (e_p, t_p) \rangle \in C$ for some t_p, (16)
- L_p contains no critical events in $H \circ L_p$, and (17)
- e_p is a critical event in $H \circ L_p \circ \langle e_p \rangle$. (18)

Therefore, by (6) and (9), and applying Lemma 4 with 'Y' $\leftarrow \mathrm{Inv}(H)$, we have a $\mathrm{Inv}(H)$-computation L satisfying the following.

- for each $p \in \mathrm{Inv}(H)$, $L \mid p \sim L_p$; (19)
- $H \circ L$ is a regular computation in C; (20)
- $\mathrm{Vis}(H \circ L) = \mathrm{Vis}(H)$; (21)
- L contains no critical events in $H \circ L$; (22)
- every event in L is executed at time $t + \Delta$. (23)

If $e_p = CS_p$ holds for some $p \in \mathrm{Inv}(H)$, then since e_p is not a read event, by P2, we have $H \circ L \circ \langle (CS_p, t + \Delta) \rangle \in C$. Thus, by the Exclusion property, there exists at most one process in $\mathrm{Inv}(H)$ that satisfies $e_p = CS_p$. Define $Y = \mathrm{Inv}(H) - \{p\}$ if such a process p exists, and $Y = \mathrm{Inv}(H)$ otherwise. Then,

$$n - 1 \leq |Y| \leq n. \tag{24}$$

[7] An *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $V' \subseteq V$ such that no edge in E is incident to two vertices in V'.

We now establish the following claim.

Claim 1: There exists a computation $\overline{G}$ satisfying the following.

- $\overline{G}$ is a regular computation in C; (25)
- $\text{Inv}(\overline{G}) \subseteq Y$; (26)
- $|\text{Inv}(\overline{G})| \geq (\sqrt{n} - m \log\log N)/(2m+1)$; (27)
- $|\text{Vis}(\overline{G})| \leq m+1$; (28)
- each process $p \in \text{Inv}(\overline{G})$ executes exactly $m+1$ critical events in $\overline{G}$; (29)
- $last(\overline{G}) = t + \Delta$; (30)
- for each $p \in \text{Inv}(\overline{G})$, $last(\overline{G} \mid p) = t + \Delta$. (31)

Proof of Claim: We consider three cases, as discussed in Section 6.

Case 1: Delay events. Assume that there exists a subset Y' of Y such that $|Y'| \geq \sqrt{n}$ and for each $p \in Y'$, e_p is a delay event. We apply Lemma 3 with 'H' $\leftarrow H \circ L$ and 'Y' $\leftarrow \text{Vis}(H) \cup Y'$. The assumptions stated in Lemma 3 follow from (20) and (21). Thus, we obtain a computation $F = (H \circ L) \mid (\text{Vis}(H) \cup Y')$, satisfying the following.

- F is a regular computation in C; (32)
- $\text{Inv}(F) = Y'$; (33)
- $\text{Vis}(F) = \text{Vis}(H)$; (34)
- for each $p \in Y'$, an event e_p is critical in F if and only if it is also critical in $H \circ L$. (35)

By combining (35) with (4) and (22), it follows that

- each process $p \in Y'$ executes exactly m critical events in F. (36)

We now append to F the next critical events by processes in Y'. For simplicity, we show how to append a single event; all other events can be appended similarly. Consider some process p in Y'. By Case 1, e_p is a delay event. Thus, by applying P2 with 'H' $\leftarrow H \circ L_p$, '(e_p, t)' $\leftarrow (e_p, t_p)$ (as defined in (16)), and 'G' $\leftarrow F$, it follows that $F' = F \circ \langle (e_p, t + \Delta) \rangle \in C$ holds, provided that F' satisfies T1–T3.

To see why F' satisfies T1–T3, let f_p be the last event by p in F. Then, f_p is contained in either L (and hence, by (19), in L_p) or in H. In the former case, by (22) and (23), f_p is a noncritical event executed at time $t + \Delta$, and hence e_p may execute at the same time. In the latter case, by (6), f_p is executed at time t, and hence, regardless of what kind of event f_p is, e_p may execute at $t + \Delta$.

Let $\overline{G}$ be the computation obtained by appending all next critical (delay) events by processes in Y'. By using (32), conditions R1–R4 can be individually checked to hold in $\overline{G}$, which implies the following: $\overline{G}$ is regular, $\text{Inv}(\overline{G}) = Y'$, and $\text{Vis}(\overline{G}) = \text{Vis}(H)$. Thus, $\overline{G}$ satisfies (25). By the definition of Y', and since $\sqrt{n} \geq (\sqrt{n} - m \log\log N)/(2m+1)$, we have (26) and (27). Since $\text{Vis}(\overline{G}) = \text{Vis}(H)$, by (8), we have (28). By (36), and since we have appended one delay event by each process in Y', we have (29). Finally, since these delay events are all executed at time $t + \Delta$, we also have (30) and (31).

Case 2: Erasing strategy. Assume that there exists a subset Y' of Y such that $|Y'| \geq \sqrt{n}$ and for each $p \in Y'$, e_p accesses a distinct variable. By (3) and (8), processes in $\text{Vis}(H)$ collectively execute at most $m \log\log N$ critical events in H, and hence, in $H \circ L$. (Recall that L is an $\text{Inv}(H)$-computation.) Hence, by the defintion of a cache-miss event, processes in $\text{Vis}(H)$ collectively access at most $m \log\log N$ distinct variables in $H \circ L$. Thus, there exists a subset Y'' of Y' satisfying the following.

- $|Y''| \geq \sqrt{n} - m \log\log N$, and (37)
- for each process $p \in Y''$, its next critical event e_p accesses a distinct variable that is not accessed by processes in $\mathrm{Vis}(H)$. (38)

Note that, by (7), Y'' is nonempty. We now construct a graph $\mathcal{G} = (Y'', E_{\mathcal{G}})$, where each vertex is a process in Y''. To each process p in Y'', we apply the following rule: for each critical event f_p by p in H, if f_p accesses a variable v that is accessed by q's next critical event e_q (for some $q \in Y''$), then introduce edge $\{p, q\}$. (The construction of the conflict graph is depicted in Figure 11 in Section 6.)

By (4) and (38), we introduce at most m edges per process. Thus, the average degree of $\mathcal{G}$ is at most $2m$. Hence, by Theorem 6, there exists an independent set $Z \subseteq Y''$ such that

$$|Z| \geq |Y''|/(2m+1) \geq \frac{\sqrt{n} - m \log\log N}{2m+1}, \quad (39)$$

where the latter inequality follows from (37). By applying Lemma 3 with 'H' $\leftarrow H \circ L$ and 'Y' $\leftarrow \mathrm{Vis}(H) \cup Z$, we obtain a computation $F = (H \circ L) \mid (\mathrm{Vis}(H) \cup Z)$, satisfying the following.

- F is a regular computation in C; (40)
- $\mathrm{Inv}(F) = Z$; (41)
- $\mathrm{Vis}(F) = \mathrm{Vis}(H)$; (42)
- each process $p \in Z$ executes exactly m critical events in F, (43)

where (43) is derived by applying (4) and (22), as in Case 1. Let $\overline{G}$ be the computation obtained by appending to F all next critical events by processes in Z, each executed at time $t + \Delta$. By using (18), (38), and (40), we can show the following: these events are also critical after F, $\overline{G}$ is regular, $\mathrm{Inv}(\overline{G}) = Z$, and $\mathrm{Vis}(\overline{G}) = \mathrm{Vis}(H)$. (In particular, the definition of Z implies that each critical event e_p by some $p \in Z$ accesses a variable v that is not accessed by any other critical events in H, and hence, in $H \circ L$. By the definition of a critical event, this implies that e_p is the only event that accesses v in $\overline{G}$.) Thus, we obtain a computation $\overline{G}$ satisfying (25)–(31).

Case 3: Covering strategy. Assume that neither Case 1 nor Case 2 is true. Then, by (24), there exist at least $n - \sqrt{n}$ processes in Y whose next critical events are *not* delay events. Among these processes, at least $(n - \sqrt{n})/\sqrt{n} = \sqrt{n} - 1$ processes access the same variable v in their next critical events, for some v. Let Y_v be the set of such processes. By applying Lemma 3 with 'H' $\leftarrow H \circ L$ and 'Y' $\leftarrow \mathrm{Vis}(H) \cup Y_v$, we obtain a computation $F = (H \circ L) \mid (\mathrm{Vis}(H) \cup Y_v)$, satisfying the following.

- F is a regular computation in C; (44)
- $\mathrm{Inv}(F) = Y_v$; (45)
- $\mathrm{Vis}(F) = \mathrm{Vis}(H)$; (46)
- each process $p \in Z$ executes exactly m critical events in F. (47)

We index processes in Y_v from p_1 to p_k, where $k = |Y_v|$, such that if e_{p_i} writes v and e_{p_j} reads v, then $i < j$ (*i.e.*, next critical writes of v precedes next critical reads of v). By (16) and P3, we can append to F the next critical events by Y_v as follows. (This strategy is depicted in Figure 12 in Section 6.)

- $\overline{G} \in C$, where $\overline{G} = F \circ \langle (e'_{p_1}, t+\Delta),\ (e'_{p_2}, t+\Delta),\ \ldots,\ (e'_{p_k}, t+\Delta) \rangle$;
- for each j, $op(e'_{p_j}) = op(e_{p_j})$.

By (18), it can be shown that each e'_{p_j} is critical in $\overline{G}$. Thus, each process in Y_v executes $m+1$ critical events in $\overline{G}$. Let p_{LW} be the last process to write to v in G (if such a process exists). If p_{LW} does not exist or if $p_{\mathrm{LW}} \in \mathrm{Inv}(H)$, then it can be shown that G is a regular computation that satisfies $\mathrm{Inv}(\overline{G}) = Y_v$, $\mathrm{Vis}(\overline{G}) = \mathrm{Vis}(H)$, and (25)–(31). On the other hand, if $p_{\mathrm{LW}} \notin \mathrm{Vis}(H)$, then

we have $p_{\rm LW} \in Y_v$. Define $p_{\rm LW}$ as visible (*i.e.*, $p_{\rm LW} \in \mathrm{Vis}(\overline{G})$.) It can be shown that G is a regular computation that satisfies $\mathrm{Inv}(\overline{G}) = Y_v - \{p_{\rm LW}\}$, $\mathrm{Vis}(\overline{G}) = \mathrm{Vis}(H) \cup \{p_{\rm LW}\}$, and (25)–(31). $\square$

We now extend $\overline{G}$ to construct a computation G that satisfies (15). Consider each process $p \in \mathrm{Act}(\overline{G})$. If $p \in \mathrm{Inv}(\overline{G})$, then by (31), we have $last(\overline{G} \mid p) = t + \Delta$. On the other hand, if $p \in \mathrm{Vis}(\overline{G})$, then from the proof of Claim 1, we have either $p = p_{\rm LW} \ \wedge \ last(\overline{G} \mid p) = t + \Delta$ or $p \in \mathrm{Vis}(H) \ \wedge \ last(\overline{G} \mid p) = t$. In the former case, p already satisfies (15). In the latter case, since p is active, it may execute some event f_p after $\overline{G}$. Thus, we can append $(f_p, t+\Delta)$ to $\overline{G}$. If f_p reads a variable v such that $writer(v, \overline{G}) = q \in \mathrm{Inv}(\overline{G})$ holds, then we erase q by applying Lemma 3 to preserve regularity. By (8), this case happens at most m times, and hence we erase at most m additional invisible processes during this procedure. Hence, by (27), G also satisfies (12).

Finally, assertions (11), (13), and (14) follow from (29), (28), and (30), respectively. Thus, G satisfies Lemma 5. $\square$

The following lemma is a formal version of Lemma 2: it provides the induction step that leads to the lower bound in Theorem 4.

Lemma 6 *Let m be a positive integer, t be a nonnegative real value, and H be a regular computation in C. Define $n = |\mathrm{Inv}(H)|$. Assume the following.*

- *each process in $\mathrm{Inv}(H)$ executes exactly m critical events in H;* (48)
- $n > (m \log\log N + 2)(4m^4(\log\log N)^2 + m)$; (49)
- $|\mathrm{Vis}(H)| \leq m$; (50)
- $last(H) = t$; (51)
- *for each $p \in \mathrm{Act}(H)$, $last(H \mid p) = t$.* (52)

Then, there exist a regular computation G in C and a real value t' $(> t)$ such that

- *each process in $\mathrm{Inv}(G)$ executes exactly $m+1$ critical events in G;* (53)
- $|\mathrm{Vis}(G)| \leq m + 1$; (54)
- $|\mathrm{Inv}(G)| = \Omega(\sqrt{n}/(\log\log N)^2)$; (55)
- $last(G) = t'$; (56)
- *for each $p \in \mathrm{Act}(G)$, $last(G \mid p) = t'$.* (57)

Proof: Define $F = H \mid \mathrm{Vis}(H)$, a computation obtained by erasing all invisible processes. By Lemma 3, F is regular. Let processes in $\mathrm{Vis}(H)$ execute in "lockstep," *i.e.*, let each active process in $\mathrm{Vis}(H)$ execute exactly one event at each time $t + j\Delta$, for $j = 1, 2, \ldots$. By the Progress property, eventually every process p in $\mathrm{Vis}(H)$ executes $Exit_p$. Thus, there exists an extension $F \circ D$, in which D is decomposed into a finite number of segments $D = D_1 \circ D_2 \circ \cdots \circ D_{k'}$, satisfying the following.

- $F \circ D \in C$;
- each D_j contains exactly one event by each process in $\mathrm{Act}(F \circ D_1 \circ \cdots D_{j-1})$, executed at time $t + j\Delta$.

As described in Section 6, we can "merge" segments so that every segment (except possibly the last one) contains some critical event. The merge procedure is described in Figure 14. Initially, we let $k = k'$, $C_j = E_j = D_j$, and $M_j = \langle\rangle$, for each j. Throughout the merge procedure, we maintain $E_j = M_j \circ C_j$, where M_j represents the "merged" segment (without critical events) and C_j represents the possibly "critical" segment, with at most one event by each process.

Since we allow noncritical events to take zero time, if a segment E_i (where $i < k$) consists entirely of noncritical events, then we reduce the execution time of $E_{i+1} \circ \cdots \circ E_k$ by Δ. Thus, events in E_i and E_{i+1} are now all executed at time $t + i\Delta$, and we merge E_i and E_{i+1} as follows: the new merged segment $M_i^{\rm new}$ equals $E_i \circ M_{i+1}$, the new critical segment $C_i^{\rm new}$ equals C_{i+1}, and the new segment $E_i^{\rm new}$ equals $E_i \circ E_{i+1}$. By assumption, the new merged segment has no critical events, and hence the loop invariant is preserved.

Continuing in this way, we can define an extension $F \circ E = F \circ E_1 \circ E_2 \circ \cdots \circ E_k$, satisfying the following.

initially $k = k'$, $C_i = E_i = D_i$, and $M_i = \langle\rangle$, for each $1 \le i \le k'$
while *true* **do**

Loop invariant:

1. $D \sim E_1 \circ E_2 \circ \cdots \circ E_k$;
2. $E_i = M_i \circ C_i$;
3. every event in E_i is executed at time $t + i\Delta$;
4. M_i does not contain any critical events;
5. C_i contains at most one event by each process.

if there exist a segment E_i $(i < k)$ without any critical event **then**
/* we now merge E_i and E_{i+1}; */
for $j := i + 1$ **to** k **do**
change timings of E_j by $-\Delta$, such that every event is executed at time $t + (j - 1)\Delta$
od;
$(M_i, C_i, E_i) := (E_i \circ M_{i+1},\ C_{i+1},\ E_i \circ E_{i+1})$;
for $j := i + 1$ **to** $k - 1$ **do** $E_j := E_{j+1}$ **od**;
$k := k - 1$
else
halt
fi
od

Figure 14: A procedure to merge segments.

- D and E consist of the same sequence of events (only event timings are different); (58)
- every event in E_j is executed at time $t + j\Delta$; (59)
- every E_j (except perhaps the last one) contains some critical event; (60)
- M_j does not contain any critical events; (61)
- C_j contains at most one event by each $p \in \mathrm{Vis}(H)$. (62)

By (3) and (50), processes in $\mathrm{Vis}(H)$ collectively execute at most $m \log\log N$ critical events in E. Thus, by (60), E is decomposed into at most $m \log\log N + 1$ segments, *i.e.*,

$$k \le m \log\log N + 1. \tag{63}$$

We now want to find "enough" unblocked processes, so that we can construct a computation G that satisfies (55). Define n', the number of unblocked processes we want to find, as

$$n' = n/(k + 1) - m. \tag{64}$$

In order to find this many unblocked processes, we apply the procedure shown in Figure 15. (The procedure is also illustrated in Figure 13 in Section 6.)

At the jth iteration, we have appended all segments $E_1 \circ E_2 \circ \cdots \circ E_{j-1}$, plus the jth merged (noncritical) segment M_j. Denote the resulting (intermediate) computation as F_j, and let U be the set of unblocked invisible processes after F_j. First, if $|U| \ge n'$, then we erase the blocked invisible processes, and apply Lemma 5 in order to construct a computation G that satisfies Lemma 6. Note that assumption (7) of Lemma 5 is satisfied as follows:

$$\begin{aligned} n' &= n/(k+1) - m \\ &> \frac{m \log\log N + 2}{k+1} \cdot (4m^4(\log\log N)^2 + m) - m \\ &\ge 4m^4(\log\log N)^2 \\ &> (m \log\log N)^2, \end{aligned}$$

where we use (49) and (63) in succession.

$F_0 := H \circ M_1$;
for $j := 0$ **to** k **do**

Loop invariant:

1. *we have appended* $E_1 \circ E_2 \circ \cdots E_{j-1} \circ M_j$*;*
2. F_j *is a regular computation in* C*;*
3. *each* $p \in \mathrm{Inv}(F_j)$ *executes* m *critical events in* F_j*;*
4. *for each* $p \in \mathrm{Inv}(F_j)$*,* $last(F_j \mid p) = t + j\Delta$*;*
5. $\mathrm{Vis}(F_j) = \mathrm{Vis}(H)$*;*
6. $|\mathrm{Inv}(F_j)| \geq n - j(n' + m)$*;*
7. *for each* $p \in \mathrm{Act}(H) \cap \mathrm{Vis}(H)$*, the following holds: either* $last(H \mid p) = t + j\Delta$*, or* $last(H \mid p) = t + (j+1)\Delta$ *and the last event by* p *in* H *is noncritical.*

$U := \{p \in \mathrm{Inv}(F_j)$: p is unblocked$\}$;
if $|U| \geq n'$ **then**
apply Lemma 3 with 'H' $\leftarrow F_j$ and 'Y' $\leftarrow \mathrm{Vis}(H) \cup U$, and erase blocked processes;
apply Lemma 5 with 'n' $\leftarrow n'$ and construct a computation with at least $(\sqrt{n'} - m \log\log N)/(2m + 1) - m$ invisible processes;
let the resulting computation by G;
halt
else
apply Lemma 3 with 'H' $\leftarrow F_j$ and'Y' $\leftarrow \mathrm{Vis}(H) \cup (\mathrm{Inv}(F_j) - U)$, and erase processes in U;
append one noncritical event per each remaining invisible process, at time $t + (j+1)\Delta$;
append events in C_j, and erase at most m invisible processes (in the same manner as in the proof of Lemma 5);
if $j < k$ **then** append M_{j+1} **fi**;
let the resulting computation be F_{j+1}
fi
od

Figure 15: A procedure to find n' unblocked processes.

Second, if $|U| < n'$, then we erase processes in U. Note that each remaining invisible process p is in its entry section (by R3) and is blocked. Therefore, p may execute a noncritical event after F_j. We append these noncritical events at time $t + (j+1)\Delta$, and then append the jth critical segment in C_j. By (50) and (62), C_j may contain at most m critical events, and hence we can erase at most m processes and preserve regularity. (In particular, if a critical event f_p in C_j reads a variable v such that $writer(v, F_j) = q \in \mathrm{Inv}(F_j)$ holds, then we erase q by applying Lemma 3 to preserve regularity.) Finally, we append the $(j+1)^{\text{st}}$ merged segment M_{j+1}, and iterate again. (It can be shown that appending noncritical events by visible processes preserves regularity.)

It follows that, at each iteration, we erase at most $n' + m$ invisible processes. Assume the procedure does not halt prior to the kth iteration. Then, at the kth iteration, we have at least $n - k(n' + m)$ invisible processes. Then, by (64),

$$n - k(n' + m) = (k+1)(n' + m) - k(n' + m) = n' + m > n',$$

and hence, at the kth iteration, we have

$$|\mathrm{Inv}(F_k)| > n'. \tag{65}$$

Moreover, since we have appended all of E, visible processes are in their noncritical sections at step k. Hence, processes in $\mathrm{Inv}(F_k)$ cannot be blocked, *i.e.*, we have $\mathrm{Inv}(F_k) = U$. Combining this with (65), it follows that the procedure of Figure 15 eventually halts, with a computation G that satisfies

$$|\mathrm{Inv}(G)| \geq (\sqrt{n'} - m \log\log N)/(2m + 1) - m, \tag{66}$$

and (53), (54), (56), and (57). By (49), we also have

$$\sqrt{\frac{n}{m \log\log N + 2} - m} \;\geq\; \sqrt{4m^4 (\log\log N)^2 + m - m} \;=\; 2m^2 \log\log N \;>\; 2m \log\log N. \tag{67}$$

Thus,

$$
\begin{aligned}
|\mathrm{Inv}(G)| &\geq \frac{\sqrt{n'} - m \log\log N}{2m+1} - m && \{\text{by (66)}\} \\
&= \frac{\sqrt{n/(k+1) - m} - m \log\log N}{2m+1} - m && \{\text{by (64)}\} \\
&\geq \frac{1}{2m+1}\left(\sqrt{\frac{n}{m \log\log N + 2} - m} - m \log\log N\right) - m && \{\text{by (63)}\} \\
&\geq \frac{1}{2(2m+1)}\sqrt{\frac{n}{m \log\log N + 2} - m} \quad - \quad m. && \{\text{by (67)}\}
\end{aligned}
$$

By (67), we also have

$$
\begin{aligned}
\frac{1}{2(2m+1)}\sqrt{\frac{n}{m \log\log N + 2} - m} &\geq \frac{2m^2 \log\log N}{2(2m+1)} \\
&= \Theta(m \log\log N).
\end{aligned}
$$

Thus, the left-hand side dominates m. Combined with (49), it follows that

$$
|\mathrm{Inv}(G)| = \Omega\left(\frac{\sqrt{n}}{m \cdot \sqrt{m \log\log N}}\right).
$$

Finally, by (3), we have $m \leq \log\log N$, and hence we have (55). Thus, G satisfies Lemma 6. $\square$

By inductively applying Lemma 6, we have Theorem 4, as shown below.

Theorem 4 *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation in which a process incurs $\Omega(\log\log N)$ RMR-Δ time complexity in order to enter and then exit its critical section.*

Proof: Let $H_1 = \langle (Enter_1,\ 0),\ (Enter_2,\ 0),\ \ldots,\ (Enter_N,\ 0)\rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is regular and each process in $\mathrm{Inv}(H) = P$ has exactly one critical event in H_1. Starting with H_1, we repeatedly apply Lemma 6 and construct a sequence of computations $(H_1,\ H_2,\ \ldots,\ H_m)$, such that each process in $\mathrm{Inv}(H_j)$ has j critical events in H_j.

Define $n_j = |\mathrm{Inv}(H_j)|$. By applying (55) with 'H' $\leftarrow H_j$ and 'G' $\leftarrow H_{j+1}$, we have

$$
n_{j+1} = \Omega(\sqrt{n_j}/(\log\log N)^2),
$$

which also implies

$$
\log\log n_{j+1} \geq \log\log n_j - \Theta(1).
$$

Therefore, there exists a number $m = \Theta(\log\log N)$ satisfying $\log\log n_m = \Omega(\log\log N)$, which in turn implies $n_m = \Omega((m \log\log N)^3)$. Thus, we can apply Lemma 6 m times and still get a regular computation that satisfies (49), in which each invisible process executes $m = \Theta(\log\log N)$ critical events. $\square$

We now explain how to adapt the preceding lemmas in order to prove Theorem 5. Throughout the rest of this appendix, we will assume a modified definition of cache-miss events, given at the end of Section 6. First, we show that Lemma 3 still holds with the new definition.

Lemma 3 *Consider a regular computation H in C. For any set Y of processes such that $\mathrm{Vis}(H) \subseteq Y$, the following hold: $H \mid Y \in C$, $H \mid Y$ is regular, and $\mathrm{Vis}(H \mid Y) = \mathrm{Vis}(H)$. Moreover, for each event e_p in $H \mid Y$,* **(i)** *if e_p is critical in $H \mid Y$, then it is also critical in H, and* **(ii)** *if e_p is critical in H and if $p \in \mathrm{Inv}(H)$, then it is also critical in $H \mid Y$.*

Proof: It suffices to prove (ii). (The proof for the rest of the lemma is the same as before.)

To show (ii), consider an event e_p in $H \mid Y$, where $p \in \mathrm{Inv}(H)$. It suffices to consider critical writes. (The other cases are straightforward.) For critical writes, the only problematic case is as follows: e_p writes v, f_p is the last write of v by p before e_p in H, and e_p is critical in H because of an event g_q that *accesses* v between f_p and e_p. If q participates in $H \mid Y$, then e_p is clearly critical.

Therefore, assume that q does not participate in $H \mid Y$. If g_q writes v, then R4 ensures that there exists some process in $\mathrm{Vis}(H)$ that writes to v between f_p and e_p, and hence e_p is also critical in $H \mid Y$. On the other hand, if g_q reads v, then since p is invisible, g_q cannot read the value written by f_p, by R1. It follows that there exists a *visible* process r that writes v between f_p and g_q (and hence, between f_p and e_p). Thus, since $\mathrm{Vis}(H) \subseteq Y$, r also participates in $H \mid Y$, making e_p critical in $H \mid Y$. $\square$

The proof of Lemma 4 remains the same without any change. Lemma 5 remains the same, except that (5) is changed as follows:

- For each $p \in \mathrm{Inv}(H)$, there exists a p-computation L_p and p's "next critical event" e_p that satisfy (16)–(18). *Moreover, e_p is not a delay event.*

Thus, Case 1 (delay events) in the proof of Lemma 5 now does not arise. We now prove a modified version of Lemma 6.

Lemma 7 *Let m be a positive integer, t be a nonnegative real value, and H be a regular computation in C. Define $n = |\mathrm{Inv}(H)|$. Assume the following.*

- *each process in* $\mathrm{Inv}(H)$ *executes exactly m non-delay critical events in H;* (68)
- $n > (m \log\log N + 2)(4m^4(\log\log N)^2 + m)$; (69)
- $|\mathrm{Vis}(H)| \leq m$; (70)
- $last(H) = t$; (71)
- *for each* $p \in \mathrm{Act}(H)$, $last(H \mid p) = t$. (72)

Then, there exist a regular computation G in C and a real value t' ($> t$) such that

- *each process in* $\mathrm{Inv}(G)$ *executes exactly $m+1$ non-delay critical events in G;* (73)
- $|\mathrm{Vis}(G)| \leq m + 1$; (74)
- $|\mathrm{Inv}(G)| = \Omega(\sqrt{n}/(\log\log N)^2)$; (75)
- $last(G) = t'$; (76)
- *for each* $p \in \mathrm{Act}(G)$, $last(G \mid p) = t'$. (77)

Proof: Define $F = H \mid \mathrm{Vis}(H)$, a computation obtained by erasing all invisible processes. By Lemma 3, F is regular. Let processes in $\mathrm{Vis}(H)$ execute in "lockstep," *i.e.*, let each active process in $\mathrm{Vis}(H)$ execute exactly one event at each time $t + j\Delta$, for $j = 1, 2, \ldots$. By the Progress property, eventually every process p in $\mathrm{Vis}(H)$ executes $Exit_p$. Thus, there exists an extension $F \circ D$, in which D is decomposed into a finite number of segments $D = D_1 \circ D_2 \circ \cdots \circ D_{k'}$, satisfying the following.

- $F \circ D \in C$;
- each D_j contains exactly one event by each process in $\mathrm{Act}(F \circ D_1 \circ \cdots D_{j-1})$, executed at time $t + j\Delta$.

As in Lemma 6, we can construct an extension $F \circ E = F \circ E_1 \circ E_2 \circ \cdots \circ E_k$, satisfying the following.

- D and E consist of the same sequence of events (only event timings are different); (78)
- every event in E_j is executed at time $t + j\Delta$; (79)
- every E_j (except perhaps the last one) contains some critical event; (80)
- M_j does not contain any critical events; (81)
- C_j contains at most one event by each $p \in \mathrm{Vis}(H)$. (82)

By (70), processes in $\text{Vis}(H)$ collectively execute at most $m \log \log N$ non-delay critical events in E. (Otherwise, our lower bound is proved.) Thus, by (80), at most $m \log \log N + 1$ segments in E may contain a non-delay critical event.

We define a segment E_j of E to be a *noncritical-delay segment* if it entirely consists of noncritical and delay events, and a *useful segment* otherwise. Let k' be the number of useful segments. Then,

$$k' \leq m \log \log N + 1. \tag{83}$$

We now want to find "enough" unblocked processes, so that we can construct a computation G that satisfies (75). Define n', the number of unblocked processes we want to find, as

$$n' = n/(k' + 1) - m. \tag{84}$$

In order to find this many unblocked processes, we apply the procedure shown earlier in Figure 15. Since we have changed the definition of a cache-miss event, it can be shown that a noncritical-delay segment does not cause a blocked process to become unblocked. In particular, if a noncritical-delay segment E_{j+1} follows another noncritical-delay segment E_j, then the set of unblocked invisible process do not increase. Since E_{j+1} is appended only if the **else** part of the algorithm is executed, these invisible processes (U in the algorithm) have already been erased at the jth step.

It follows that, at the $(j + 1)^{\text{st}}$ step, there is no unblocked invisible process. (All unblocked invisible processes have been erased in the previous step, and no blocked processes may become unblocked after E_{j+1}.) Thus, appending E_{j+1} does not result in erasing any invisible process.

Thus, if the algorithm of Figure 15 does not halt until the last (k^{th}) segment, then we have erased at most $k'(n' + m)$ invisible processes, instead of $k(n' + m)$ (which may be much larger). Since k' is bounded by (83), it follows that we still have n' invisible processes that are not erased (see (65)).

Moreover, since we have appended all of E, visible processes are in their noncritical sections at step k. Hence, these invisible processes cannot be blocked. The rest of the proof is the same as in Lemma 6: we can construct G by applying Lemma 4 to these unblocked invisible processes. □

With these changes, we have Theorem 5.

Theorem 5 *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.* □