Local-spin Mutual Exclusion Using Fetch-and-ϕ Primitives*

(Extended Abstract)

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

Abstract

We present a generic *fetch-and-ϕ*-based local-spin mutual exclusion algorithm, with $O(1)$ time complexity under the remote-memory-references time measure. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$, where N is the number of processes. The *rank* of a *fetch-and-ϕ* primitive is a notion introduced herein; informally, it expresses the extent to which processes may "order themselves" using that primitive. This algorithm breaks new ground because it shows that $O(1)$ time complexity is possible using a wide range of primitives. In addition, previously published *fetch-and-ϕ*-based algorithms either use multiple primitives or require an underlying cache-coherence mechanism for spins to be local, while ours does not. By applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\log_r N)$ algorithm using any primitive of rank r, where $2 \leq r < N$. For certain primitives of constant rank, we present a slightly more efficient $\Theta(\log N / \log \log N)$ algorithm. It follows from a previously-presented lower bound proof that this algorithm is time-optimal.

Keywords: Fetch-and-ϕ primitives, local spinning, mutual exclusion, scalability, shared-memory systems

*Work supported by NSF grants CCR 9972211, CCR 9988327, and ITR 0082866.

1 Introduction

Recent work on mutual exclusion has focused on the design of algorithms that minimize contention for the processors-to-memory interconnection network 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 spin 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.[1] As explained later, it is generally more difficult to design local-spin algorithms for DSM machines than for CC machines.

In this paper, several results concerning the time complexity of local-spin mutual exclusion algorithms are presented. The notion of time complexity assumed is that given by the *remote-memory-references measure* [2]. Under this measure, an algorithm's time complexity is defined as the total number of remote memory references required in the worst case by one process to enter and then exit its critical section once. An algorithm may have different time complexities on CC and DSM machines, because on CC machines, variable locality is dynamically determined, while on DSM machines, it is statically determined.

The main focus of this paper is mutual exclusion algorithms implemented using *fetch-and-ϕ* primitives. A *fetch-and-ϕ* primitive is characterized by a particular function ϕ, accesses a single variable atomically, and has the effect of the following pseudo-code, where *var* is the variable accessed.

```
fetch-and-φ(var, input)
    old := var;
    var := φ(old, input);
    return(old)
```

In this paper, we distinguish between *fetch-and-ϕ* primitives that are comparison primitives and those that are not. A *comparison primitive* conditionally updates a shared variable after first testing that its value meets some condition; examples include *compare-and-swap* and *test-and-set*.[2] Non-comparison primitives update variables unconditionally; examples include *fetch-and-increment* and *fetch-and-store*.

In recent work, we established a time-complexity lower bound of $\Omega(\log N / \log\log N)$ remote memory references for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives. In contrast, several constant-time algorithms are known that are based on noncomparison *fetch-and-ϕ* primitives [3, 4, 8]. This suggests that noncomparison primitives may be the best choice to provide in hardware, if one is interested in implementing efficient blocking synchronization mechanisms.

Constant-time local-spin mutual algorithms that use noncomparison primitives have been proposed by T. Anderson [3], Graunke and Thakkar [4], and Mellor-Crummey and Scott [8]. In each of these algorithms, blocked processes wait within a "spin queue." A process enqueues itself by using a *fetch-and-ϕ* primitive to update a shared "tail" pointer; a process's predecessor (if any) in the queue is indicated by the primitive's return value. A process in the spin queue waits (if necessary) until released by its predecessor. Although these algorithms follow a common strategy, they vary in the primitives used and the progress properties ensured. Some important attributes of each algorithm are listed below.

- T. Anderson's algorithm uses *fetch-and-increment* and requires an underlying cache-coherence mechanism for spins to be local. Thus, it has $O(1)$ time complexity only on CC machines.
- Graunke and Thakkar's algorithm uses *fetch-and-store*. Time complexity is $O(1)$ only on CC machines.
- Mellor-Crummey and Scott actually presented two variants of their algorithm, one that uses *fetch-and-store*, and a second that uses both *fetch-and-store* and *compare-and-swap*. In both, spins are local on both CC and DSM machines. However, the *fetch-and-store* variant is not starvation-free, and hence actually has unbounded time complexity. The variant that also uses *compare-and-swap is* starvation-free and has $O(1)$ time complexity on both CC and DSM machines.

[1] If a DSM machine has a cache-coherence mechanism, then we consider it to be a CC machine.

[2] *compare-and-swap* and *test-and-set* are ordinarily defined to return a boolean condition indicating if the comparison succeeded. In this paper, we instead assume that each returns the accessed variable's original value, as in [5]. It is straightforward to modify any algorithm that uses the boolean versions of these primitives to instead use the versions considered in this paper.

The existence of these algorithms gives rise to a number of intriguing questions regarding mutual exclusion algorithms. Is it possible to devise an $O(1)$ algorithm for DSM machines that uses a single *fetch-and-ϕ* primitive? Can such an algorithm be devised using primitives other than *fetch-and-increment* and *fetch-and-store*? Is it possible to automatically transform a local-spin algorithm for CC machines so that it has the same time complexity on DSM machines? Given that the $\Omega(\log N/\log\log N)$ lower bound mentioned above applies to algorithms that use comparison primitives, we know that there exist *fetch-and-ϕ* primitives that are not sufficient for constructing $O(1)$ algorithms. For such primitives, what is the most efficient algorithm that can be devised? Can we devise a *ranking* of synchronization primitives that indicates the singular characteristic of a primitive that enables a certain time complexity (for mutual exclusion) to be achieved? Such a ranking would provide information relevant to the implementation of *blocking* synchronization mechanisms that is similar to that provided by Herlihy's wait-free hierarchy [5], which is relevant to *nonblocking* mechanisms.

Contributions of this paper. In this paper, we partially address the questions raised above. Our main contribution is a generic N-process *fetch-and-ϕ*-based local-spin mutual exclusion algorithm that has $O(1)$ time complexity on both CC and DSM machines. This algorithm is "generic" in the sense that it can be implemented using any *fetch-and-ϕ* primitive of rank $2N$. Informally, a primitive of rank r has sufficient symmetry-breaking power to linearly order up to r invocations of that primitive. Our generic algorithm breaks new ground because it shows that $O(1)$ time complexity is possible using a wide range of primitives, on both CC and DSM machines. Thus, introducing *additional* primitives to ensure local spinning on DSM machines, as Mellor-Crummey and Scott did, is not necessary.

We present our generic algorithm by first giving a variant that is designed for CC machines. We then present a very general transformation that can be used to convert algorithms that locally spin on CC machines to ones that locally spin on DSM machines. This transformation is then applied to our algorithm. (This same transformation can also be applied to the algorithms of T. Anderson and Graunke and Thakkar.)

By applying our generic algorithm within an arbitration tree, one can easily construct a $\Theta(\log_r N)$ algorithm using any primitive of rank r, where $2 \leq r < N$. For the case $r = \Theta(N)$, this algorithm is clearly optimal. However, we show that there exists a class of primitives with constant rank for which $\Theta(\log_r N)$ is *not* optimal. We show this by presenting an optimal $\Theta(\log N/\log\log N)$ algorithm that can be implemented using any primitive in this class. As explained later, the optimality of this algorithm follows from the $\Omega(\log N/\log\log N)$ lower bound mentioned above.

Organization. The rest of this paper is organized as follows. In Sec. 2, we present needed definitions. Then, in Sec. 3, we present our generic algorithm. The $\Theta(\log N/\log\log N)$ algorithm mentioned above is then presented in Sec. 4. We end the paper with concluding remarks in Sec. 5.

2 Definitions

Due to space constraints, we refrain from giving a definition of the mutual exclusion problem; such a definition can be found in any concurrent algorithms textbook (*e.g.*, [7]). We hereafter let N denote the number of processes in the system, and assume that each process has a unique process identifier in the range $0, \ldots, N-1$.

We assume the existence of a *generic fetch-and-ϕ* primitive, as defined in Sec. 1. We will use "*Vartype*" to denote the type of the accessed variable *var*. (The accessed variable's type is part of the definition of such a primitive.) For example, for a *fetch-and-increment* primitive, *Vartype* would be **integer**, and for a *test-and-set* primitive, it would be **boolean**. In our algorithms, we use $\perp$ to denote the initial value of a variable accessed by a *fetch-and-ϕ* primitive (*e.g.*, if *Vartype* is **boolean**, then $\perp$ would denote either *true* or *false*). We now define the notion of a "rank," mentioned earlier.

Definition: The *rank* of a *fetch-and-ϕ* primitive is the largest integer r satisfying the following.

> For each process p, there exists a constant array $\alpha[p][0..k_p - 1]$ of input values (for some k_p), such that if p performs a sequence of *fetch-and-ϕ* invocations as specified below on a variable v (of type *Vartype*) that is initially $\perp$ (for some choice of $\perp$),
>
> **for** $i := a_p$ **to** ∞ **do** *fetch-and-ϕ*$(v, \alpha[p][i \bmod k_p])$ **od**

shared variables

CurrentQueue: $0, 1$;
Tail: **array**$[0, 1]$ **of** *Vartype* **initially** $\perp$;
Position: **array**$[0, 1]$ **of** $0..2N - 1$ **initially** 0;
Signal: **array**$[0, 1][Vartype]$ **of boolean initially** *false*;
Active: **array**$[0..N - 1]$ **of boolean initially** *false*;
QueueIdx: **array**$[0..N - 1]$ **of** $(\perp, 0, 1)$;
*Waiter*1: **array**$[0..N - 1]$ **of** $(\perp, 0..N - 1)$;
*Waiter*2: **array**$[0, 1][Vartype]$ **of** $(\perp, 0..N - 1)$;
Spin: **array**$[0..N - 1]$ **of boolean initially** *false*

private variables

idx: $0, 1$;
counter: **integer**;
prev, *self*, *old*: *Vartype*;
pos: $0..2N - 1$;
q: $0..N - 1$;
next: $(\perp, 0..N - 1)$;
flag: **boolean**

Figure 1: Variables used in Algorithms G-CC and G-DSM.

where a_p is some integer value, then in any interleaving of these invocations by the N different processes, **(i)** any two invocations among the first $r - 1$ by different processes write different values to v, **(ii)** any two *successive* invocations among the first $r - 1$ by the same process write different values to v, and **(iii)** of the first r invocations, only the first invocation returns $\perp$.

A *fetch-and-ϕ* primitive has *infinite rank* if the condition above is satisfied for arbitrarily large values of r. □

As our generic algorithm shows, a *fetch-and-ϕ* primitive with rank r has enough power to linearly order r invocations by possibly different processes unambiguously. Note that it is not necessary for the primitive to fully order invocations by the same process, since each process can keep its own execution history.

Examples. An r-bounded *fetch-and-increment* primitive on a variable v with range $0, \ldots, r - 1$ is defined by $\phi(old, input) = \min(r - 1, old + 1)$. (In this primitive, the input parameter is not used, and hence we may simply assume $\alpha[p][j] = \perp$ for all p and j.) If v is initially 0, then any r consecutive invocations on v return distinct values, $0, 1, \ldots, r - 1$. Moreover, any further invocation (after the r^{th}) returns $r - 1$, which is the same as the return value of the r^{th} invocation. Therefore, an r-bounded *fetch-and-increment* primitive has rank r, and an unbounded *fetch-and-increment* primitive has infinite rank.

For *fetch-and-increment* primitives, the input parameter α is extraneous. However, this is not the case for other primitives. As a second example, consider a *fetch-and-store* primitive on a variable that is large enough to hold $2N + 1$ distinct values ($2N$ pairs $(p, 0)$ and $(p, 1)$, where p is a process, and one for the initial value $\perp$). It is easily shown that *fetch-and-store* has infinite rank. This follows by defining $\alpha[p][j] = (p, j \bmod 2)$. (Informally, each process may write the information "this is an (even/odd)-indexed invocation by process p" each time.) It also follows that an unbounded *fetch-and-store* primitive has infinite rank.

3 A Constant-time Generic Algorithm

In this section, we present an $O(1)$ mutual exclusion algorithm that uses a generic *fetch-and-ϕ* primitive, which is assumed to have rank at least $2N$. Two variants of the algorithm are presented, one for CC machines and one for DSM machines. In local-spin algorithms for DSM machines, each process must have its own dedicated spin variables (which must be stored in its local memory module). In contrast, in algorithms for CC machines, processes may *share* spin variables, because each process can read a different cached copy. Because of this flexibility, algorithms for CC machines tend to be a bit simpler than those for DSM machines. This is why we present separate algorithms. Our CC algorithm, denoted G-CC, is presented first, and then its DSM counterpart, denoted G-DSM, is obtained by means of a fairly simple transformation.

The two algorithms and associated variable declarations are shown in Figs. 1–3. Each algorithm is specified by giving *Acquire* and *Release* procedures, which are invoked by a process to perform its entry and exit sections, respectively. In both algorithms, "**await** B," where B is a boolean expression, is used as a shorthand for the busy-waiting loop "**while** $\neg B$ **do** /* null */ **od**."

Reset mechanism. When trying to implement a mutual exclusion algorithm using a generic *fetch-and-ϕ* primitive — of which only its rank r is known — the primary problem that arises is the following.

```
process p ::    /* 0 ≤ p < N */
procedure Acquire()
1:  QueueIdx[p] := ⊥;
2:  Active[p] := true;
3:  idx := CurrentQueue;
4:  QueueIdx[p] := idx;
5:  prev := fetch-and-φ(Tail[idx], α[p][counter]);
6:  self := φ(prev, α[p][counter]);
7:  counter := counter + 1 mod k_p;
8:  if prev ≠ ⊥ then
9:      await Signal[idx][prev];
10:     Signal[idx][prev] := false
    fi;
11: Acquire_2(idx)

procedure Release()
12: pos := Position[idx];
13: Position[idx] := pos + 1;
14: Release_2(idx);
15: if (pos < N) ∧ (pos ≠ p) ∧ (Active[pos]) then
16:     q := pos;
17:     await ¬Active[q] ∨
18:           (QueueIdx[q] = idx)
19: elseif pos = N then
20:     Tail[1 − idx] := ⊥;
21:     Position[1 − idx] := 0;
22:     CurrentQueue := 1 − idx
    fi;
23: Signal[idx][self] := true;
24: Active[p] := false
```

Figure 2: Algorithm G-CC: Generic mutual exclusion algorithm for CC machines.

```
process p ::    /* 0 ≤ p < N */
procedure Acquire()
1:      QueueIdx[p] := ⊥;
2:      Active[p] := true;
3:      idx := CurrentQueue;
4:      Acquire_2(p, 1);
5:          QueueIdx[p] := idx;
6:          q := Waiter[p];
7:      Release_2(p, 1);
8:      if q ≠ ⊥ then Spin[q] := true fi;
9:      prev := fetch-and-φ(Tail[idx], α[p][counter]);
10:     self := φ(prev, α[p][counter]);
11:     counter := counter + 1 mod k_p;
12:     if prev ≠ ⊥ then
13:         Acquire_2((idx, prev), 0);
14:             flag := Signal[idx][prev];
15:             Waiter[idx][prev] := if flag then ⊥ else p;
16:             Spin[p] := false;
17:         Release_2((idx, prev), 0);
18:         if ¬flag then
19:             await Spin[p];
20:             Waiter[idx][prev] := ⊥
            fi;
21:         Signal[idx][prev] := false
        fi;
22:     Acquire_2(idx)

procedure Release()
23:     pos := Position[idx];
24:     Position[idx] := pos + 1;
25:     Release_2(idx);
26:     if (pos < N) ∧ (pos ≠ p) ∧ (Active[pos]) then
27:         q := pos;
28:         Acquire_2(q, 0);
29:             flag :=¬Active[q] ∨
30:                    (QueueIdx[q] = idx);
31:             Waiter[q] := if flag then ⊥ else p;
32:             Spin[p] := false;
33:         Release_2(q, 0);
34:         if ¬flag then
35:             await Spin[p];
36:             Waiter[q] := ⊥
            fi
37:     elseif pos = N then
38:         Tail[1 − idx] := ⊥;
39:         Position[1 − idx] := 0;
40:         CurrentQueue := 1 − idx
        fi;
41:     Acquire_2((idx, self), 1);
42:         Signal[idx][self] := true;
43:         next := Waiter[idx][self];
44:     Release_2((idx, self), 1);
45:     if next ≠ ⊥ then Spin[next] := true fi;
46:     Acquire_2(p, 1);
47:         Active[p] := false;
48:         next := Waiter[p];
49:     Release_2(p, 1);
50:     if next ≠ ⊥ then Spin[next] := true fi
```

Figure 3: Algorithm G-DSM: Generic algorithm for DSM machines. Lines different from Fig. 2 are shown with **boldface** line numbers.

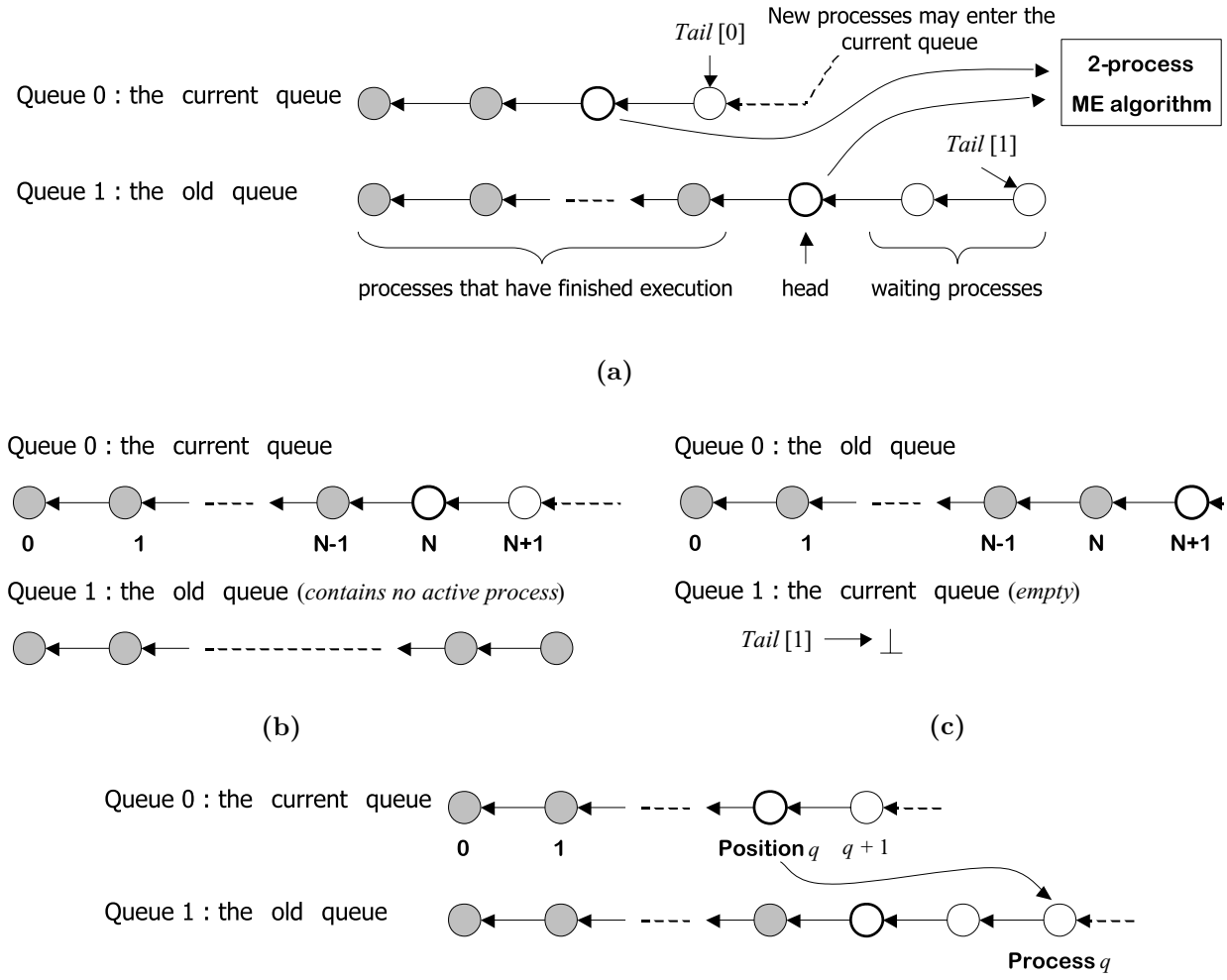

Figure 4: The structure of Algorithm G-CC. **(a)** The overall structure. This figure shows a possible state of execution when the current queue is queue 0. (The "finished" processes may be duplicated, because a process may execute its critical section multiple times.) **(b)** A state just before *CurrentQueue* is updated. **(c)** A state just after *CurrentQueue* is updated. **(d)** A process (in its exit section) in the current queue waiting for another in the old queue.

> If the primitive is invoked more than r times to access a variable, then it may not be able to provide enough information for processes to order themselves. Therefore, *the algorithm must provide a means of resetting such a variable before it is accessed* $2N$ *times*.

We solve this problem in Algorithm G-CC by using two "waiting queues," indexed 0 and 1. Associated with each queue j is a "tail pointer," $Tail[j]$. In its entry section, a process enqueues itself onto one of these two queues by using the *fetch-and-*ϕ primitive to update its tail pointer, and waits on its predecessor, if necessary. At any time, one of the queues is designated as the "current" queue, which is indicated by the shared variable *CurrentQueue*. The other queue is called the "old" queue. The algorithm switches between the two queues over time in a way that ensures that each tail pointer is reset before being accessed $2N$ times. We now describe the reset mechanism in detail.

When a process invokes the *Acquire* routine, it determines which queue is the current queue by reading the variable *CurrentQueue* (line 3 of Fig. 2), and then enqueues itself onto that queue using the *fetch-and-*ϕ primitive (lines 5–7). If p is not at the head of its queue ($p.prev \neq \perp$), then it waits until its predecessor in the queue updates the spin variable $Signal[p.idx][p.prev]$ (line 9), which p then resets (line 10).

Note that it is possible for a process q to read *CurrentQueue* before another process updates its value to switch to the other queue. Such a process q will then enqueue itself onto the old queue. Thus, *both* queues may possibly hold waiting processes. To arbitrate between processes in the two queues, an extra two-process mutual exclusion algorithm is used. A process competes in this two-process algorithm after reaching the head of its waiting queue using the routines $\texttt{Acquire}_2$ and $\texttt{Release}_2$, with the index of its queue as a "process identifier" (lines 11 and 14). This is illustrated in Fig. 4(a), where the current queue is queue 0. Note that this extra two-process algorithm can be implemented from reads and writes in $O(1)$ time [9].

As explained above, some process must reset the current queue before it is accessed $2N$ times. To facilitate this, each queue j has an associated shared variable $Position[j]$. This variable indicates the relative position of the current head of the queue, starting from 0. For example, in Fig. 4(a), the head of queue 0 is at position 2, and hence $Position[0] = 2$. A process in queue j updates $Position[j]$ while still effectively in its critical section (lines 12 and 13). Thus, $Position[j]$ cannot be concurrently updated by different processes.

A process exchanges the role of the two queues after completing its critical section if it is at position N in the current queue (lines 20–22). Insets (b) and (c) of Fig. 4 show the state of the two queues before and after such an exchange. In order to exchange the queues, we must ensure the following invariant.

Invariant If a process executes its critical section after having acquired position N of the current queue, then no process is in the old queue. (I1)

(A process is considered to be "in" the old queue if it read the index of that queue from *CurrentQueue*. In particular, that process may be yet to update the queue's tail pointer.) Given this invariant, a process at position N may safely reset the old queue and exchange the queues. Invariant (I1) is a direct consequence of the following invariant. (Recall that process identifiers range from 0 to $N - 1$.)

Invariant If a process executes its critical section after having acquired position *pos* of the current queue, and if $pos > q$, then process q is not in the old queue. (I2)

To maintain (I2), each process p has two associated variables, $Active[p]$ and $QueueIdx[p]$, which indicate (respectively) whether process p is active, and if so, which queue it is executing in (lines 1, 2, 4, and 24). If a process p executes at position q ($< N$) in the current queue, then in its exit section, p checks $QueueIdx[q]$ in order to see if process q is in the old queue (line 15); if that is the case, then p waits until q finishes its critical section (lines 17 and 18), before signaling to its successor (*i.e.*, a process at position $q + 1$ in the current queue) that it is now at the head of that queue (line 23). This situation is depicted in Fig. 4(d).

Although p waits for q, starvation freedom is guaranteed, because q is in the old queue, and hence makes progress independent of the current queue. Only the current queue is stalled until q finishes execution. (The fact that p may have to wait for a significant duration in its exit section may be a cause for concern. However, with a slightly more complicated handshake, such waiting can be eliminated. The idea is to require process p to instruct process q to signal p's successor *after* q finishes its critical section. Therefore, p may finish execution without waiting for q. For simplicity, this handshake has not been added to Algorithm G-CC.)

We still must show that using a *fetch-and-*ϕ primitive of rank $2N$ is sufficient. Suppose that process p acquires position N of queue 0 when it is the current queue. We claim that at most $N - 1$ processes may be enqueued onto queue 0 after p and before the queues are exchanged again. For a process q to enqueue itself onto queue 0 after p, it must have read the value of *CurrentQueue* before it was updated by p. For q to enqueue itself a *second* time onto queue 0, it must read $CurrentQueue = 0$ again, *after* $CurrentQueue = 1$ was established by p. This implies that the two queues have been exchanged again. (We remind the reader that, by the explanation above, the queues will not be exchanged again until there are no processes in queue 0.) Thus, after p establishes that queue 1 is current, and while queue 0 continues to be the old queue, at most $N - 1$ processes may be enqueued (after p) onto queue 0. Thus, a rank of $2N$ is sufficient.

Time complexity. The busy-waiting loops at lines 9, 17, and 18 in Fig. 2 are read-only loops in which variables are read that may be updated by a unique process. On a CC machine, the first read of such a variable generates a cached copy, and hence subsequent reads until the variable is updated are handled in-cache. In all cases, such a variable can be updated a constant number of times before the waiting condition is established. Thus, each busy-waiting loop generates a constant number of remote memory references. (This

analysis ignores any invalidations or displacements of cached variables due to cache associativity or capacity constraints.) Because there are no loops in the algorithm aside from busy-waiting loops, it follows that the time complexity of Algorithm G-CC is $O(1)$ on CC machines. Thus, we have the following lemma.

Lemma 1 *If the underlying fetch-and-ϕ primitive has rank at least $2N$, then Algorithm G-CC is a correct, starvation-free mutual exclusion algorithm with $O(1)$ time complexity in CC machines.* □

Algorithm G-DSM. We now explain how to convert Algorithm G-CC into Algorithm G-DSM. The key idea of this conversion is a simple transformation of each busy-waiting loop, which we examine here in isolation. This transformation generalizes one presented earlier by us [6]. In Algorithm G-CC, all busy waiting is by means of statements of the form "**await** B," where B is some boolean condition. Moreover, if a process p is waiting for condition B to hold, then there is a unique process that can establish B, and once B is established, it remains true, until p's "**await** B" statement terminates.

In Algorithm G-DSM, each statement of the form "**await** B" has been replaced by the code fragment on the left below (see lines 13–20 and 28–36 in Fig. 3), and each statement of the form "$B := \mathit{true}$" by the code fragment on the right (see lines 4–8, 41–45, and 46–50).

```
a: Acquire_2(J, 0);
b:     flag := B;
c:     Waiter[J] := if flag then ⊥ else p;
d:     Spin[p] := false;
e: Release_2(J, 0);
f: if ¬flag then
g:     await Spin[p];
h:     Waiter[J] := ⊥
   fi
```

```
i: Acquire_2(J, 1);
j:     B := true;
k:     next := Waiter[J];
l: Release_2(J, 1);
m: if next ≠ ⊥ then Spin[next] := true fi
```

The variable $\mathit{Waiter}[\mathcal{J}]$ is assumed to be initially $\perp$, and $\mathit{Spin}[p]$ is a spin variable used exclusively by process p (and, hence, it can be stored in memory local to p). $\texttt{Acquire}_2$ and $\texttt{Release}_2$ represent an instance of a two-process mutual exclusion algorithm, indexed by $\mathcal{J}$. To see that this transformation is correct, assume that a process p executes lines a–h while another process q executes lines i–m. Since lines b–d and j–k execute within a critical section, lines b–d precede lines j–k, or *vice versa*. If b–d precede j–k, and if $B = \mathit{false}$ holds before the execution of b–d, then p assigns $\mathit{Waiter}[\mathcal{J}] := p$ at line c, and initializes its spin variable at line d. Process q subsequently reads $\mathit{Waiter}[\mathcal{J}] = p$ at line k, and establishes $\mathit{Spin}[p] = \mathit{true}$ at line m, which ensures that p is not blocked. On the other hand, if lines j–k precede lines b–d, then process q reads $\mathit{Waiter}[\mathcal{J}] = \perp$ (the initial value) at line k, and does not update any spin variable at line m. Since process p executes line b after q executes line j, p preserves $\mathit{Waiter}[\mathcal{J}] = \perp$, and does not execute lines g and h. Given the correctness of this transformation, we have the following.

Lemma 2 *If the underlying fetch-and-ϕ primitive has rank at least $2N$, then Algorithm G-DSM is a correct, starvation-free mutual exclusion algorithm with $O(1)$ time complexity in DSM machines.* □

The transformation above also can be applied to the algorithms of T. Anderson [3] and Graunke and Thakkar [4]. In each case, the two-process mutual algorithm actually can be avoided by utilizing the specific *fetch-and-ϕ* primitive used (*fetch-and-increment* and *fetch-and-store*, respectively).

If we have a *fetch-and-ϕ* primitive with rank r ($4 \leq r < 2N$), then we can arrange instances of Algorithm G-DSM in an arbitration tree, where each process is statically assigned a leaf node and each non-leaf node consists of an $\lfloor r/2 \rfloor$-process mutual exclusion algorithm, implemented using Algorithm G-DSM. Because this arbitration tree is of $\Theta(\log_r N)$ height, we have the following theorem. (Note that for $r = 2$ or 3, a $\Theta(\log_r N)$ algorithm is possible without even using the *fetch-and-ϕ* primitive [9].)

Theorem 1 *Using any fetch-and-ϕ primitive of rank $r \geq 2$, starvation-free mutual exclusion can be implemented with $\Theta(\log_{\min(r,N)} N)$ time complexity on either CC or DSM machines.* □

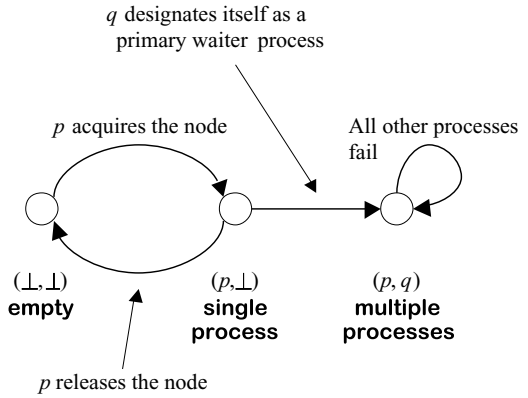

Figure 5: Transitions allowed by *fetch-and-toggle*.

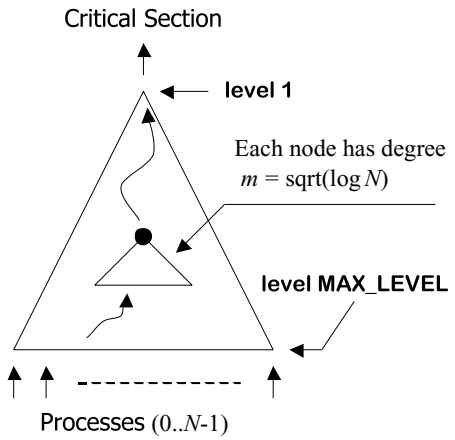

Figure 6: Arbitration tree of Algorithm T.

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

The time-complexity bound in Theorem 1 is clearly tight for $r = \Theta(N)$. In this section, we show that for $r = 3$, it is *not* tight. This follows from Algorithm T, shown in Fig. 9, which has $\Theta(\log / \log\log N)$ time complexity on both DSM and CC machines. This algorithm is based on a (somewhat contrived) *fetch-and-ϕ* primitive of rank three. This primitive, termed *fetch-and-toggle*, is defined by the following:

$$\begin{aligned} \mathit{Vartype} &= ((\perp, 0..N-1), (\perp, 0..N-1)), \\ \phi((q,r),p) &= \begin{cases} (p,\perp), & \text{if } (q,r) = (\perp,\perp) \\ (\perp,\perp), & \text{if } (q,r) = (p,\perp) \\ (q,p), & \text{if } q \neq \perp,\ q \neq p, \text{ and } r = \perp \\ (q,r), & \text{otherwise,} \end{cases} \end{aligned}$$

where (q, r) denotes the old value of the accessed variable, and p denotes the process invoking the primitive.

Algorithm T uses an arbitration tree, each node n of which is represented by a variable $\mathit{Lock}[n]$ of type *Vartype*. Informally, a value of $(\perp, \perp)$ represents an available node; $(p, \perp)$, where $p \neq \perp$, represents a situation in which p has acquired the node and no other process has since accessed the node; (p, q), where $p \neq \perp$ and $q \neq \perp$, represents a situation in which p has acquired the node, and q waiting at that node (perhaps along with some other processes). Transitions allowed by the *fetch-and-toggle* primitive are shown in Fig. 5. By examining the transition diagram, it is easy to see that *fetch-and-toggle* has rank three.

Arbitration tree and waiting queue. The structure of the arbitration tree is illustrated in Fig. 6. The tree is of degree $m = \sqrt{\log N}$. Each process is statically assigned to a leaf node, which is at level `MAX_LEVEL`. (The root is at level 1.) Since the tree has N leaf nodes, $\texttt{MAX\_LEVEL} = \Theta(\log_m N) = \Theta(\log N / \log\log N)$.

To enter its critical section, a process p traverses the path from its leaf up to the root and attempts to acquire each node on this path. If p acquires the root node, then it may enter its critical section. As explained shortly, p may also be "promoted" to its critical section while still executing within the tree. (In that case, p may have acquired only some of the nodes on its path.) In either case, upon exiting its critical section, p traverses its path in reverse, releasing each node it has acquired.

In addition to the arbitration tree, a serial waiting queue, *WaitingQueue*, is used. This queue is accessed by a process only within its exit section. A "barrier" mechanism is used that ensures that multiple processes do not execute their exit sections concurrently. As a result, the waiting queue can be implemented as a sequential data structure. It is accessible by the usual *Enqueue* and *Dequeue* operations, and also an operation *Remove*(*WaitingQueue*, p), which removes process p from inside the queue, if present; it is straightforward to implement each of these operations in $O(1)$ time. When a process p, inside its exit section, discovers another waiting process q, p adds q to the waiting queue. In addition, p dequeues a process r from the queue (if the

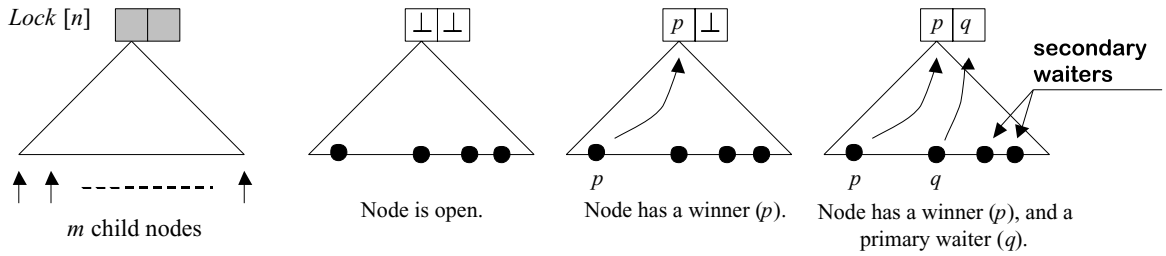

Figure 7: Structure of a node used in Algorithm T.

queue is nonempty), and "promotes" r to its critical section. (This mechanism is rather similar to helping mechanisms used in wait-free algorithms [5].)

Arbitration at a node. As mentioned above, associated with each (non-leaf) node n is a "lock variable" $Lock[n]$ of type *Vartype*, which represents the state of that node. The structure of such a node is illustrated in Fig. 7. In its entry section, a process p may try to acquire node n only if it has already acquired some child of n. In order to acquire node n, p executes *fetch-and-toggle*($Lock[n]$). Assume that the old value of $Lock[n]$ is returned in the private variables $(winner, waiter)$. There are three possibilities to consider.

- If $winner = \perp$ holds, then p has established $Lock[n] = (p, \perp)$ and has acquired the node. In this case, p proceeds to the next level of the tree.
- If $winner = q$ (for some process q) and $waiter = \perp$ hold, then p has established $Lock[n] = (q, p)$, in which case it becomes the "primary waiter" at node n. In this case, p stops at node n and waits until it is "promoted" to its critical section by some other process.
- Otherwise, p's *fetch-and-toggle* does not change $Lock[n]$, in which case p is a "secondary waiter" at node n. In this case, p also waits at node n until it is promoted.

Next, consider the behavior of a process p in its exit section. There are two possibilities to consider, depending on p's execution history in its entry section.

- If p acquired node n in its entry section, then p has established $Lock[n] = (p, \perp)$. In this case, p tries to release node n by executing *fetch-and-toggle*($Lock[n]$) once again. If no other process has updated $Lock[n]$ between p's two invocations of *fetch-and-toggle*($Lock[n]$), then node n is successfully released (*i.e.*, $Lock[n]$ transits to $(\perp, \perp)$, as depicted in Fig. 5). In this case, p descends the tree and continues to release other nodes it has acquired.

  On the other hand, if some other process has updated $Lock[n]$ between p's two invocations, then let q be the first such process. As explained above, q must have changed $Lock[n]$ from $(p, \perp)$ to (p, q), thus designating itself as the primary waiter of node n. In this case, p adds q to the waiting queue. (Note that p does *not* enqueue any secondary waiters, *i.e.*, processes that accessed $Lock[n]$ after q.) Process p then releases node n by writing (not via *fetch-and-toggle*) $Lock[n] := (\perp, \perp)$, and descends the tree.
- If p was promoted at node n, then p has *not* acquired node n, and hence is not responsible for releasing node n. Instead, p examines every child of node n (specifically, $Lock[child]$, where *child* is a child of n) to determine if any "secondary waiters" at node n exist. p adds such processes to the waiting queue.

It is straightforward to show each process eventually either acquires the root, or is added to the waiting queue by some other process. Since the waiting queue is checked every time a process executes its exit section, it follows that the algorithm is starvation free.

As explained above, processes exiting the arbitration tree form two groups: the promoted processes and the non-promoted processes (*i.e.*, those that successfully acquire the root). To arbitrate between these two groups, an additional two-process mutual exclusion algorithm is used. The manner in which this algorithm and the barrier mentioned previously are used is illustrated in Fig. 8.

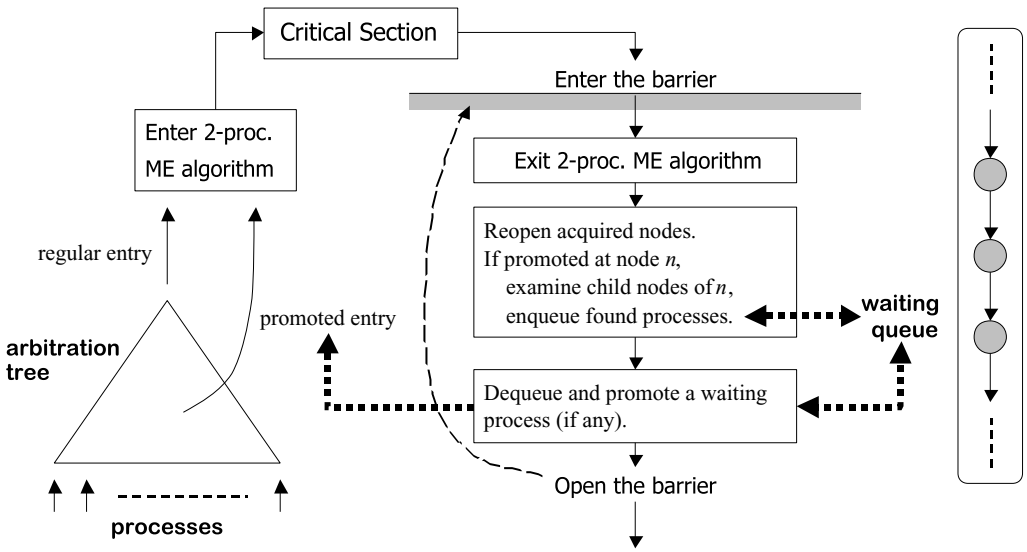

Figure 8: The exit section of Algorithm T. Dashed and dotted lines represent information/signal flow.

Further details. Having explained the basic structure of the algorithm, we now present a more detailed overview. We begin by considering the shared variables used in the algorithm, which are listed in Fig. 9. *Lock* and *WaitingQueue* have already been explained. *Spin*[*p*] is a dedicated spin variable for process p. *Promoted* is used to hold the identity of any promoted process. This variable is used to ensure that multiple processes are not promoted concurrently, which is required in order to ensure that the additional two-process mutual exclusion algorithm is accessed by only one promoted process at a time.

We now consider the *Acquire* and *Release* procedures in some detail. A process p in its entry section first initializes its spin variable (line 1), and automatically acquires its leaf node (line 2). It then ascends the arbitration tree (lines 3–9). p tries to acquire each node it visits by executing line 5. If it succeeds, then it ascends to the next level; otherwise, it spins at line 7 until it is promoted by some other process. If p acquires the root node, then it executes the two-process entry section using "0" as a process identifier (line 11). Otherwise, it executes the two-process entry section using "1" as a process identifier (line 9). The private variable *break_level* stores the level at which p exited the **for** loop (lines 8 and 10).

In its exit section, p waits until the barrier is opened (line 12) and then executes the two-process exit section (line 13). The barrier is specified by two procedures `Wait` and `Signal`, which ensure that p waits at line 12 if another process is executing within lines 13–36. Because `Wait` is invoked within a critical section, it is straightforward to implement these procedures in $O(1)$ time. In CC machines, `Wait` can be defined as "**await** *Flag*; *Flag* := *false*" and `Signal` as "*Flag* := *true*," where *Flag* is a shared boolean variable. In DSM machines, a slightly more complicated implementation is required, which we omit due to space limitations.

If p was promoted at node n (*i.e.*, *break_level* > 0), then it examines *Lock*[n] (line 16). If p finds *Lock*[n] = ($winner, p$) at line 16, then p is the primary waiter at n, and was promoted *before* the winning process at node n (given by *winner*, which was assigned at line 5) entered its critical section. This can happen because p may actually have been promoted by a primary waiter at a lower level. In this case, p resets node n in place of the winning process, and adds that process to the waiting queue (lines 17–18). After that, p adds any process that has acquired a child node of n to the waiting queue (lines 19–22).

Regardless of whether p was promoted, it tries to reopen each node that it acquired in its entry section (lines 23–29). For each such node n, p checks if it is still the winner (line 24); this may not be the case, if the primary waiter at node n executed lines 16–18 before p entered its critical section. If p is indeed the winner at node n, then it tries to reopen node n (line 26). p may fail to reopen node n only if node n has a primary waiter, in which case p enqueues the waiter and reopens the node using an ordinary write (lines 28, 29). Finally, p resets its leaf node (line 30), makes sure that it is not contained in the waiting queue (line 31), and checks if there is any unfinished promoted process (lines 32, 33). If not, then p dequeues and promotes a process from the waiting queue (if one exists) (lines 34–36). As a last step, p opens the barrier (line 37).

```
shared variables
    Lock : array[1..MAX_NODE] of Vartype
        initially (⊥, ⊥);
    Spin : array[0..N − 1] of boolean;
    WaitingQueue : (serial waiting queue)
        initially empty;
    Promoted : (⊥, 0..N − 1) initially ⊥

private variables
    lev, break_level : 1..MAX_LEVEL;
    n, child : 1..MAX_NODE;
    winner, waiter, last, q, next : ⊥, 0..N − 1

process p ::    /* 0 ≤ p < N */

procedure Acquire()
1:  Spin[p] := false;
2:  Lock[Node(p, MAX_LEVEL)] := (p, ⊥);
3:  for lev := MAX_LEVEL − 1 downto 1 do
4:      n := Node(p, lev);
5:      (winner, waiter) :=
            fetch-and-toggle(Lock[n]);
6:      if winner ≠ ⊥ then
7:          await Spin[p];
                    /* wait until promoted */
8:          break_level := lev;
9:          Acquire_2(1);   /* promoted entry */
            return
        fi
    od;
10: break_level := 0;
11: Acquire_2(0)            /* normal entry */

procedure Release()
12: Wait();                 /* wait at the barrier */
13: if break_level = 0 then Release_2(0) else Release_2(1) fi;
14: if break_level > 0 then
15:     n := Node(p, break_level);
16:     if Lock[n] = (winner, p) then
17:         Lock[n] := (⊥, ⊥);
18:         Enqueue(winner)
        fi;
19:     for each child := (a child of n) do
20:         (winner, waiter) := Lock[child];
21:         if (winner ≠ ⊥) then
22:             Enqueue(WaitingQueue, winner) fi
        od
    fi;
23: for lev := break_level + 1 to MAX_LEVEL − 1 do
        /* reopen each node p has acquired */
24:     n := Node(p, lev);    (winner, waiter) := Lock[n];
25:     if winner = p then
26:         (winner, waiter) := fetch-and-toggle(Lock[n]);
27:         if waiter ≠ ⊥ then
28:             Enqueue(WaitingQueue, waiter);
29:             Lock[n] := (⊥, ⊥)
        fi fi
    od;
30: Lock[Node(p, MAX_LEVEL)] := (⊥, ⊥);
31: Remove(WaitingQueue, p);
32: q := Promoted;
33: if (q = p) ∨ (q = ⊥) then
34:     next := Dequeue(WaitingQueue);
35:     Promoted := next;
36:     if next ≠ ⊥ then Spin[next] := true fi
    fi;
37: Signal()                /* open the barrier */
```

Figure 9: Algorithm T: A tree-structured algorithm using *fetch-and-toggle* primitive.

In order to compute the time complexity of the algorithm, note that $\texttt{MAX\_LEVEL} = \Theta(\log N / \log\log N)$ holds, and that the busy-waiting loop (line 7) spins on a local variable. Therefore, the **for** loops of lines 3–9 and lines 23–29 generate $\Theta(\log N / \log\log N)$ remote memory references. Since the arbitration tree has degree $\Theta(\sqrt{\log N})$, the **for** loop of lines 19–22 generates $\Theta(\sqrt{\log N})$ remote memory references, which is asymptotically dominated by $\Theta(\log N / \log\log N)$. Thus, we have the following theorem.

Theorem 2 *Using fetch-and-toggle, which is of rank three, starvation-free mutual exclusion can be implemented with $\Theta(\log N / \log\log N)$ time complexity on either CC or DSM machines.* □

It can be shown that our previous $\Omega(\log N / \log\log N)$ lower-bound proof [1] applies to certain systems that use *fetch-and-ϕ* primitives of constant rank. The proof inductively extends computations so that information flow among processes is limited. If, at some induction step, a variable v is accessed by many processes, then information flow is kept low by ensuring that v may be assigned $O(1)$ different values during this induction step. Therefore, our lower bound applies to any *fetch-and-ϕ* primitive satisfying the following: *any consecutive invocations of the primitive by different processes can be reordered to return $O(1)$ different values.* Since *fetch-and-toggle* clearly satisfies this condition, we conclude that Algorithm T is time-optimal. It is possible to adapt Algorithm T to use primitives other than *fetch-and-toggle*; examples include a *fetch-and-increment/decrement* primitive with bounded range 0..2, a variant of *compare-and-swap* that allows two

different compare values to be specified, and a simultaneous execution of *test-and-set* and write. These algorithms are also time-optimal.

5 Concluding Remarks

We have shown that any *fetch-and-ϕ* primitive of rank r can be used to implement a $\Theta(\log_r N)$ mutual exclusion algorithm, on either DSM or CC machines. $\Theta(\log_r N)$ is clearly optimal for $r = \Theta(N)$. On the other hand, we have shown $\Theta(\log_r N)$ is *not* optimal for all primitives of constant rank by presenting an optimal $\Theta(\log N / \log\log N)$ algorithm that can be implemented using various such primitives.

We believe that the notion of rank defined in this paper may be a suitable way of characterizing the "power" of primitives from the standpoint of blocking synchronization, much like the notion of a *consensus number*, which is used in Herlihy's wait-free hierarchy [5], reflects the "power" of primitives from the standpoint of nonblocking synchronization. Interestingly, primitives like *compare-and-swap* that are considered to be powerful according to Herlihy's hierarchy are weak from a blocking synchronization standpoint (since they are subject to our $\Omega(\log N / \log\log N)$ lower bound [1]). Also, primitives like *fetch-and-increment* and *fetch-and-store* that are considered to be powerful from a blocking synchronization standpoint are considered quite weak according to Herlihy's hierarchy. (They have consensus number two.) This difference arises because in nonblocking algorithms, the need to reach consensus is fundamental (as shown by Herlihy), while in blocking algorithms, the need to order competing processes is important.

The $\Theta(\log N / \log\log N)$ algorithm in Sec. 4 shows that $\Omega(\log N / \log\log N)$ is a tight lower bound for *some* class of synchronization primitives. Unfortunately, we have been unable to adapt the algorithm to work with only reads, writes, and comparison primitives. Currently, we still believe that $\Omega(\log N)$ is a tight lower bound for algorithms based on such operations, as conjectured by us earlier [1]. Our $\Theta(\log N / \log\log N)$ algorithm may shed some light on this issue. In particular, it shows that a better bound must necessarily be based on proof techniques that exclude some of the primitives allowed by the current proof.

References

[1] 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, August 2001.

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

[3] 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.

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

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

[6] Y.-J. Kim and J. Anderson. A space- and time-efficient local-spin spin lock. *Information Processing Letters*, to appear.

[7] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, March 1996.

[8] 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.

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