A Space- and Time-efficient Local-spin Spin Lock

Yong-Jik Kim and James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill

March 2001

Abstract

A simple code transformation is presented that reduces the space complexity of Yang and Anderson's local-spin mutual exclusion algorithm. In this algorithm, only atomic read and write instructions are used; each process generates $\Theta(\log N)$ remote memory references per lock request, where N is the number of processes. The space complexity of the transformed algorithm is $\Theta(N)$, which is clearly optimal. Its time complexity is $\Theta(\log N)$, like the original.

Keywords: Mutual exclusion, local spinning, read/write atomicity, shared-memory systems, space complexity

1 Introduction

In the mutual exclusion problem [4], each of a set of N processes repeatedly executes a "critical section" of code. Each process's critical section is preceded by an "entry section" and followed by an "exit section." The objective is to design the entry and exit sections so that the following requirement holds.

- **Exclusion:** At most one process executes its critical section at any time.

In the variant of the problem considered here, we also require the following.

- **Starvation-freedom:** Each process in its entry section eventually enters its critical section.

In [7], Yang and Anderson presented a mutual exclusion algorithm that uses only atomic read and write instructions, and showed that its performance is comparable to that of algorithms based on strong synchronization primitives [3, 5, 6]. The per-process time complexity of their algorithm is $\Theta(\log N)$, where "time" is measured by counting only memory references that cause a traversal of the interconnect between processors and shared memory. The algorithm is designed so that all busy-waiting is by means of "local-spin" loops in which shared variables are read locally without an interconnect traversal. There are two architectural paradigms that support local accesses of shared variables: on a distributed shared-memory machine, a shared variable is locally accessible if it is stored in a local memory module, and on a cache-coherent machine, a shared variable is locally accessible if it is stored in a local cache line.

Yang and Anderson's algorithm is listed as ALGORITHM YA in Fig. 2. ALGORITHM YA is constructed by embedding instances of a two-process mutual exclusion algorithm within a binary arbitration tree. The two-process algorithm has $O(1)$ time complexity, so the overall per-process time complexity is $\Theta(\log N)$ per lock request. In order to guarantee that different nodes in the arbitration tree do not interfere with each other, a distinct spin variable is used for each process for each level of the arbitration tree. Thus, the algorithm's space complexity $\Theta(N \log N)$.

In this paper, we present a simple code transformation that reduces the space complexity of ALGORITHM YA from $\Theta(N \log N)$ to $\Theta(N)$. In our new algorithm, each process uses the same spin variable for all levels of the arbitration tree. Like ALGORITHM YA, the time complexity of our new algorithm is $\Theta(\log N)$. In a recent paper [2], we established a time-complexity lower bound of $\Omega(\log N / \log \log N)$ remote memory

```
const                                              /* for simplicity, we assume N = 2^L */
   L = log N;                             /* (depth of arbitration tree) + 1 = O(log N) */
   Tsize = 2^L − 1 = N − 1                          /* size of arbitration tree = O(N) */

shared variables
   T: array[1..Tsize] of 0..N − 1;
   C: array[1..Tsize][0, 1] of (0..N − 1, ⊥) initially ⊥;
   R: array[1..Tsize][0, 1] of (0..N − 1, ⊥);                          /* ALGORITHM LS */
   Q: array[0..N − 1] of 0..2 initially 0;                             /* ALGORITHM YA */
   P: array[1..Tsize][0, 1] of 0..2 initially 0;            /* ALGORITHMS CC, LS and F */
   S: array[0..N − 1] of boolean initially false                /* ALGORITHMS LS and F */

private variables
   h: 1..L;
   node: 1..2T+1;
   side: 0, 1;                                            /* 0 = left side, 1 = right side */
   rival: 0..N − 1
```

Figure 1: Variable declarations.

references for mutual exclusion algorithms under read/write atomicity. We conjecture that $\Omega(\log N)$ is actually a tight lower bound for this class of algorithms. If this conjecture is true, then the algorithm of this paper is both time- and space-optimal.

The rest of the paper is organized as follows. In Sec. 2, we show that our new algorithm can be derived from ALGORITHM YA in a manner that preserves the Exclusion and Starvation-freedom properties. In Sec. 3, we prove that the algorithm's time complexity is $\Theta(\log N)$. Sec. 4 concludes the paper.

2 Transformation of the Algorithm

Starting with ALGORITHM YA, we construct three other algorithms, each obtained from its predecessor by means of a simple code transformation. These other algorithms are ALGORITHM CC (for cache-coherent), ALGORITHM LS (for linear space), and ALGORITHM F (the final algorithm). The first two algorithms are shown in Fig. 2, and the second two in Fig. 3. Variable declarations for all the algorithms are given in Fig. 1. In Figs. 2 and 3, we have used "**await** B," where B is a boolean expression, as a shorthand for the busy-waiting loop "**while** B **do od**."

ALGORITHM YA. We begin with a brief, informal description of ALGORITHM YA. At each node n in the arbitration tree, the following variables are used: $C[n][0]$, $C[n][1]$, $T[n]$, and $Q[0], \ldots, Q[N-1]$. Variable $C[n][0]$ ranges over $\{0, \ldots, N-1, \perp\}$ and is used by a process from the left subtree rooted at n to inform a process from the right subtree rooted at n of its intent to enter its critical section. Variable $C[n][1]$ is similarly used by processes from the right subtree. Variable $T[n]$ ranges over $\{0, \ldots, N-1\}$ and is used as a tie-breaker in the event that two processes attempt to "acquire" node n at the same time. In such a case, the process that first updates $T[n]$ is favored. Variable $Q[p]$ is the spin variable used by process p.

Loosely speaking, the two-process algorithm at node n works as follows. A process l from the left subtree rooted at n "announces" its arrival at node n by establishing $C[n][0] = l$. It then assigns its identifier l to the tie-breaker variable $T[n]$, and initializes its spin variable $Q[l]$. If no process from the right-side has attempted to acquire node n, *i.e.*, if $C[n][1] = \perp$ holds when l executes statement 5, then process l proceeds directly to the next level of the arbitration tree (or to its critical section if n is the root). Otherwise, if $C[n][1] = r$, where r is some right-side process, then l reads the tie-breaker variable $T[n]$. If $T[n] \neq l$, then process r has updated $T[n]$ *after* process l, so l can enter its critical section (recall that ties are broken in favor of the first process to update $T[n]$). If $T[n] = l$ holds, then either process r executed statement 3 before process l, or process r has executed statement 2 but not statement 3. In the first case, l should wait until r "releases" node n in its exit section, whereas, in the second case, l should be able to proceed past

/* lines of ALGORITHM CC that are different from ALGORITHM YA are shown in **boldface** */

ALGORITHM YA /* the original algorithm in [7] */

```
process p ::   /* 0 ≤ p < N */
while true do
1:  Noncritical Section;

    for h := 1 to L do
        node := ⌊(N + p)/2^h⌋;
        side := ⌊(N + p)/2^(h−1)⌋ mod 2;
2:      C[node][side] := p;
3:      T[node] := p;
4:      Q[p] := 0;
5:      rival := C[node][1 − side];
        if (rival ≠ ⊥ ∧
6:              T[node] = p) then
7:          if Q[rival] = 0 then
8:              Q[rival] := 1;
            fi;
9:          await Q[p] ≥ 1;
10:         if T[node] = p then
11:             await Q[p] = 2
            fi
        fi
    od;

12: Critical Section;

    for h := L downto 1 do
        node := ⌊(N + p)/2^h⌋;
        side := ⌊(N + p)/2^(h−1)⌋ mod 2;
13:     C[node][side] := ⊥;
14:     rival := T[node];
        if rival ≠ p then
15:         Q[rival] := 2
        fi
    od
od
```

ALGORITHM CC /* the cache-coherent algorithm */

```
process p ::   /* 0 ≤ p < N */
while true do
1:  Noncritical Section;

    for h := 1 to L do
        node := ⌊(N + p)/2^h⌋;
        side := ⌊(N + p)/2^(h−1)⌋ mod 2;
2:      C[node][side] := p;
3:      T[node] := p;
4:      P[node][side] := 0;
5:      rival := C[node][1 − side];
        if (rival ≠ ⊥ ∧
6:              T[node] = p) then
7:          if P[node][1 − side] = 0 then
8:              P[node][1 − side] := 1;
            fi;
9:          await P[node][side] ≥ 1;
10:         if T[node] = p then
11:             await P[node][side] = 2
            fi
        fi
    od;

12: Critical Section;

    for h := L downto 1 do
        node := ⌊(N + p)/2^h⌋;
        side := ⌊(N + p)/2^(h−1)⌋ mod 2;
13:     C[node][side] := ⊥;
14:     rival := T[node];
        if rival ≠ p then
15:         P[node][1 − side] := 2
        fi
    od
od
```

Figure 2: ALGORITHM YA and ALGORITHM CC.

node n. This ambiguity is resolved by having process l execute statements 7 through 11. Statements 7 and 8 are executed by process l to release process r in the event that it is waiting for l to update the tie-breaker variable (*i.e.*, r is busy-waiting at node n at statement 9). Statements 8-10 are executed by l to determine which process updated the tie-breaker variable first. Note that $Q[l] \geq 1$ implies that r has already updated the tie-breaker, and $Q[l] = 2$ implies that l has released node n. To handle these two cases, process l first waits until $Q[l] \geq 1$ holds (*i.e.*, until r has updated the tie-breaker), re-examines $T[n]$ to see which process updated it last, and finally, if necessary, waits until $Q[l] = 2$ holds (*i.e.*, until process r releases node n).

After executing its critical section, process l releases node n by establishing $C[n][0] = 0$. If $T[n] = r$, in which case process r is waiting to enter its critical section, then process l updates $Q[r]$ in order to terminate r's busy-waiting loop.

ALGORITHM CC. In ALGORITHM CC, each node n has two associated spin variables, $P[n][0]$ and $P[n][1]$. $P[n][0]$ is used by *all* processes that try to acquire node n from the left side. $P[n][1]$ is similarly used by all right-side processes. This algorithm has $\Theta(N)$ space complexity, and also has $\Theta(\log N)$ time complexity on cache-coherent systems, because each spin variable is waited on by at most one process at any time. However, because the association of spin variables to processes is dynamic, we cannot statically allocate spin variables to processes. Therefore, ALGORITHM CC has *unbounded* time complexity on distributed shared-memory systems without coherent caches. (Although virtually every modern multiprocessor is cache-coherent, non-cache-coherent systems are still used in embedded applications, where cheaper computing technology often must be used due to cost limitations. Thus, algorithms for non-cache-coherent systems are still of interest.) The remaining algorithms correct this problem.

The correctness of Algorithm CC follows easily from the correctness of Algorithm YA. In fact, Algorithm YA is trickier to prove correct than Algorithm CC, because in Algorithm YA, a node becomes dynamically associated with different spin variables, depending on the identity of the processes competing at that node.

Algorithm LS. Algorithm LS has been obtained from Algorithm CC by applying a simple transformation, which we examine here in isolation. In Algorithm 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 LS, each statement of the form "**await** B" has been replaced by the following code fragment:

a: $R := p$;
b: **while** $\neg B$ **do**
c: await $S[p]$;
d: $S[p] := \mathit{false}$
    od

where $S[p]$ is initially *false* (see statements 9 and 11). In addition, each assignment of the form "$B := \mathit{true}$" has been replaced by the following:

e: $B := \mathit{true}$;
f: $\mathit{rival} := R$;
g: $S[\mathit{rival}] := \mathit{true}$

(see statements 8 and 15). Note that the code implementing "**await** B" can terminate only if B is *true*, *i.e.*, it terminates only when it should. Moreover, if a process p finds that B is *false* at statement b, and if another process q subsequently establishes B by executing statement e, then because p's execution of statement a precedes q's execution of statement f, q establishes $S[p] = \mathit{true}$ when it executes statement g. Thus, if a process p is waiting for condition B to hold, and B is established by another process, then p must eventually exit the **while** loop at statement b.

Algorithm F. Algorithm F has been obtained from Algorithm LS by removing the shared array R, which was introduced in applying the transformation above at each node of the arbitration tree. The fact that this array is unnecessary follows from several invariants of Algorithm LS, which are stated below. In stating these invariants, the following notation is used.

- $p@S$ is true if and only if process p's program counter equals some value in the set S.
- $p.v$ denotes the private variable v of process p.

Before stating the required invariants, first note that removing array R does not affect safety, because the **while** conditions at statements 9b and 11b still ensure that the busy-waiting loops terminate only when they should. However, there is now a potential danger that a process may not correctly update its rival's spin variable, and hence that process may wait forever at either statement 9c or 11c. The following invariants (of Algorithm LS) imply that this is not possible.

invariant $q@\{\mathsf{9b}, \mathsf{9c}, \mathsf{9d}, \mathsf{11b}, \mathsf{11c}, \mathsf{11d}\} \Rightarrow R[q.\mathit{node}][q.\mathit{side}] = q \wedge C[q.\mathit{node}][q.\mathit{side}] = q$ (I1)

invariant $p@\{6..\mathsf{8f}\} \wedge p.\mathit{rival} \neq C[p.\mathit{node}][1 - p.\mathit{side}] \wedge$
$q@\{3..\mathsf{11d}\} \wedge q.\mathit{node} = p.\mathit{node} \wedge q.\mathit{side} = 1 - p.\mathit{side} \Rightarrow$
$q@\{3\} \vee T[p.\mathit{node}] = q$ (I2)

/* lines that are different from ALGORITHM CC are marked in **boldface** */

ALGORITHM LS /* the linear-space algorithm */

```
      process p ::   /* 0 ≤ p < N */
      while true do
1:      Noncritical Section;
        for h := 1 to L do
            node := ⌊(N + p)/2^h⌋;
            side := ⌊(N + p)/2^(h−1)⌋ mod 2;
2:          C[node][side] := p;
3:          T[node] := p;
4:          P[node][side] := 0;
5:          rival := C[node][1 − side];
            if (rival ≠ ⊥ ∧
6:                T[node] = p) then
7:              if P[node][1 − side] = 0 then
8:  8e              P[node][1 − side] := 1;
    8f              rival := R[node][1 − side];
    8g              S[rival] := true
                fi;
9:  9a          R[node][side] := p;
    9b          while P[node][side] = 0 do
    9c              await S[p];
    9d              S[p] := false
                od;
10:             if T[node] = p then
11: 11a             R[node][side] := p;
    11b             while P[node][side] ≤ 1 do
    11c                 await S[p];
    11d                 S[p] := false
                    od
                fi
            fi
        od;
12:     Critical Section;
        for h := L downto 1 do
            node := ⌊(N + p)/2^h⌋;
            side := ⌊(N + p)/2^(h−1)⌋ mod 2;
13:         C[node][side] := ⊥;
14:         rival := T[node];
            if rival ≠ p then
15: 15e         P[node][1 − side] := 2;
    15f         rival := R[node][1 − side];
    15g         S[rival] := true
            fi
        od
      od
```

ALGORITHM F /* the final algorithm */

```
      process p ::   /* 0 ≤ p < N */
      while true do
1:      Noncritical Section;
        for h := 1 to L do
            node := ⌊(N + p)/2^h⌋;
            side := ⌊(N + p)/2^(h−1)⌋ mod 2;
2:          C[node][side] := p;
3:          T[node] := p;
4:          P[node][side] := 0;
5:          rival := C[node][1 − side];
            if (rival ≠ ⊥ ∧
6:                T[node] = p) then
7:              if P[node][1 − side] = 0 then
8:  8e              P[node][1 − side] := 1;
    8f              —
    8g              S[rival] := true
                fi;
9:  9a          —
    9b          while P[node][side] = 0 do
    9c              await S[p];
    9d              S[p] := false
                od;
10:             if T[node] = p then
11: 11a             —
    11b             while P[node][side] ≤ 1 do
    11c                 await S[p];
    11d                 S[p] := false
                    od
                fi
            fi
        od;
12:     Critical Section;
        for h := L downto 1 do
            node := ⌊(N + p)/2^h⌋;
            side := ⌊(N + p)/2^(h−1)⌋ mod 2;
13:         C[node][side] := ⊥;
14:         rival := T[node];
            if rival ≠ p then
15: 15e         P[node][1 − side] := 2;
    15f         —
    15g         S[rival] := true
            fi
        od
      od
```

Figure 3: ALGORITHM LS and ALGORITHM F.

invariant $p@\{\mathsf{8f}\} \wedge q@\{\mathsf{9b}, \mathsf{9c}, \mathsf{9d}, \mathsf{11b}, \mathsf{11c}, \mathsf{11d}\} \wedge q.node = p.node \wedge q.side = 1 - p.side \Rightarrow$
$p.rival = R[p.node][1 - p.side] \vee T[p.node] = q$ (I3)

invariant $p@\{14, \mathsf{15e}, \mathsf{15f}, \mathsf{15g}\} \wedge q@\{\mathsf{9b}, \mathsf{9c}, \mathsf{9d}, \mathsf{11b}, \mathsf{11c}, \mathsf{11d}\} \Rightarrow T[p.node] = p \vee T[p.node] = q$ (I4)

Invariant (I1) follows easily from the correctness of the arbitration-tree mechanism used in ALGORITHM LS and the fact that process q establishes $C[q.node][q.side] = q$ (statement 2) before it establishes $R[q.node][q.side] = q$ (statements 9a and 11a).

To see that (I2) holds, note that $p.rival = C[p.node][1 - p.side]$ holds when $p@\{6\}$ is established by process p. The only other statements that may establish its antecedent are statements 2 and 13 of some other process q with $q.node = p.node \wedge q.side = 1 - p.side$. These statements may establish $q@\{3..11\mathsf{d}\}$ or falsify $p.rival = C[p.node][1 - p.side]$. However, $q@\{3..11\mathsf{d}\}$ is false after q executes statement 13 (moreover, by the correctness of the arbitration-tree mechanism, $r@\{3..11\mathsf{d}\} \wedge r.node = p.node \wedge r.side = 1 - p.side$ is false after the execution of statement 13 by q for *any* choice of r). Also, statement 2 establishes $q@\{3\}$, *i.e.*, the consequent of (I2) holds. Finally, note that whenever $q@\{3\}$ is falsified, $T[p.node] = q$ is established.

Invariant (I3) follows from (I1) and (I2) by considering two cases. Assume that its antecedent holds. If $p.rival = C[p.node][1 - p.side]$ holds, then $p.rival = R[p.node][1 - p.side]$ follows by (I1). Otherwise, $T[p.node] = q$ follows by (I2).

By the correctness of the arbitration-tree mechanism, it easily follows that after both p and q have executed statement 3 with $p.node = n \wedge q.node = n$ and before either of them leaves node n, no other process can enter node n. Therefore, the consequent of (I4) is not falsified while its antecedent holds.

We now claim that these invariants imply the correctness of ALGORITHM F. Assume that process p executes statement 15f of ALGORITHM LS while process q is waiting at statements $\{9\mathsf{b}, \ldots, 9\mathsf{d}, 11\mathsf{b}, \ldots, 11\mathsf{d}\}$. For this to happen, by (I4), p must have found $T[p.node] = q$ at statement 14. However, by (I1), p also finds $R[p.node][1 - p.side] = q$ at statement 15f. It follows that statement 15f reads the same value that is already read by statement 14, and hence is dispensable.

The other case is more subtle. Assume that process p executes statement 8f instead. In this case, the antecedent of (I3) holds. If $p.rival = R[p.node][1 - p.side]$ holds, statement 8f does not modify $p.rival$, and thus statement 8f is clearly dispensable. On the other hand, if $T[p.node] = q$ holds, then by the tie-breaking strategy used in ALGORITHM LS, p enters its critical section before q does. Therefore, in this case, p will eventually execute statements $\{15\mathsf{e}, \ldots, 15\mathsf{g}\}$ and release q from its spinning. Hence, the progress property is preserved even if we remove statement 8f from the algorithm.

Finally, now that we have dropped statements 8f and 15f, we can also drop statements 9a and 11a. Thus, we have the following theorem.

Theorem 1: ALGORITHM F *is a correct, starvation-free mutual exclusion algorithm, with* $\Theta(N)$ *space complexity.* □

3 Time Complexity

Although we showed in the previous section that ALGORITHM F is a starvation-free algorithm, we didn't establish its time complexity. (Starvation-freedom merely indicates that a process *eventually* enters its critical section; it does not specify how soon.) In this section, we show that each process performs $\Theta(\log N)$ remote memory references in ALGORITHM F to enter and then exit its critical section. Because the spins at statements 9c and 11c are local, it clearly suffices to establish a $\Theta(\log N)$ bound on the total number of iterations of the **while** loops at statements 9b and 11b for one complete entry-section execution.

Consider a process p. During its entry section, the total iteration count of the **while** loops at statements 9b and 11b is bounded by the number of statement executions that establish $S[p] = true$. This can happen only if some other process q executes statement 8g or 15g while $q.rival = p$ holds. The arbitration-tree structure implies that this can happen only if

- process p always enters node n from side s, where $s = 0$ or 1, and
- process q executes either statement 8g or 15g while $q.node = n \wedge q.side = 1 - s \wedge q.rival = p$ holds.

Since there are $\Theta(\log N)$ nodes along the path taken by process p to reach its critical section, it suffices to prove the following lemma (the term *event*, which is used in the lemma, refers to a single statement execution).

Lemma 1: Consider a process p and a node n. Assume that p always enters node n from side s ($s = 0$ or 1) during its entry section. During an interval in which p is in its entry section, there can be at most seven[1] events e such that e is an execution of statement 8g or 15g by some process q with $q.node = n \wedge q.side = 1 - s \wedge q.rival = p$.

Proof: We represent process p's execution of statement z by $z.p[n, s]$, where n and s are the values of $p.node$ and $p.side$, respectively, before statement z is executed. We consider three cases, depending on the program counter of process p.

Case 1. Process p has not yet executed $2.p[n, s]$.

By the program text, $C[n][s] \neq p$ holds before statement $2.p[n, s]$ is executed. A process q, other than p, can establish $S[p] = true$ by executing either 8g.$q[n, 1-s]$ or 15g.$q[n, 1-s]$ *only once*. Note that, after that single event, and while Case 1 continues to hold, any process r (r could be q or another arbitrary process) executing with $r.node = n \wedge r.side = 1 - s$ will find $r.rival \neq p$ at statement 5 or 14, and hence cannot establish $S[p] = true$.

Case 2. Process p has executed $2.p[n, s]$ but not $3.p[n, s]$ (*i.e.*, $p@\{3\} \wedge p.node = n \wedge p.side = s$ holds).
Case 3. Process p has executed $3.p[n, s]$ but not $4.p[n, s]$ (*i.e.*, $p@\{4\} \wedge p.node = n \wedge p.side = s$ holds).
Case 4. Process p has executed $4.p[n, s]$.

While each of these cases holds, a process q, other than p, can establish $S[p] = true$ *twice* by executing statements 8g.$q[n, 1-s]$ and 15g.$q[n, 1-s]$.

Assume that after q exits node n, a process r (r could be q again, or another arbitrary process) enters node n from side $1 - s$. Note that statement 15g.$q[n, 1-s]$ has already established $P[n][s] \neq 0$, which is not falsified by any statement while one of Cases 2, 3, or 4 continues to hold. (By the correctness of the arbitration-tree mechanism, the only statement that may falsify $P[n][s] \neq 0$ is $4.p[n, s]$, which falsifies Case 3 and establishes Case 4.) Therefore, process r will find $P[n][s] \neq 0$ at statement 7 if any of Cases 2, 3, or 4 continues to hold, and will not execute statements 8e and 8g.

Similarly, note that process r itself establishes $T[n] = r$ by executing $3.r[n, 1-s]$, which is not falsified by any statement while one of Cases 2, 3, or 4 continues to hold. (By the correctness of the arbitration-tree mechanism, the only statement that may falsify $T[n] = r$ is $3.p[n, s]$, which falsifies Case 2 and establishes Case 3.) Therefore, process r will find $T[n] = r$ at statement 14 if any of Cases 2, 3, or 4 continues to hold, and will not execute statements 15e and 15g.

From these arguments, it follows that $S[p] = true$ can be established *at most twice* while *each* of Cases 2, 3, and 4 continues to hold. Hence, we have established the following bound on the number of events that may establish $S[p] = true$: 1 {Case 1} + 2 {Case 2} + 2 {Case 3} + 2 {Case 4} = 7. □

Finally, from Theorem 1 and Lemma 1, we have the following.

Theorem 2: ALGORITHM F *is a correct, starvation-free mutual exclusion algorithm, with* $\Theta(N)$ *space complexity and* $\Theta(\log N)$ *time complexity, on both cache-coherent and distributed shared-memory systems.* □

4 Conclusion

We have presented a mutual exclusion algorithm with $\Theta(N)$ space complexity and $\Theta(\log N)$ time complexity on both cache-coherent and distributed shared-memory systems. Our algorithm was created by applying a series of simple transformations to Yang and Anderson's mutual exclusion algorithm. The transformation used to obtain ALGORITHM LS may actually be of independent interest, because it can be applied to convert any algorithm in which spin variables are dynamically shared into one in which each process has a unique spin location. The resulting algorithm will be a local-spin algorithm on a distributed shared-memory machine without coherent caches as long as the **while** loops introduced in the transformation cannot iterate unboundedly.

[1] The number of events can be actually reduced to *five* with careful bookkeeping, but since this does not change the asymptotic argument, we will content ourselves with a less tight bound here.

In recent work, we showed that, by using ALGORITHM F as a subroutine, it is possible to construct an adaptive mutual exclusion algorithm with $\Theta(N)$ space complexity and $\Theta(\min(k, \log N))$ time complexity, where k is the number of processes simultaneously competing for their critical sections [1]. In fact, it was that work that motivated us to re-examine the space complexity of Yang and Anderson's algorithm.

References

[1] 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.

[2] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. In submission, January 2001.

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

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

[6] 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.

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