due to the fact that each two-process entry and exit section of Peterson's algorithm requires fewer remote operations outside of busy-waiting loops than does ours. This gives rise to the following open question: how many remote operations outside of busy-waiting loops are required in two-process read/write algorithms based on local spinning?

We end the paper with a few observations concerning the time complexity of concurrent algorithms. A natural approach to measuring the time complexity of such an algorithm would be to simply count the number of operations. However, a straightforward application of such an approach does not provide any insight into the behavior of mutual exclusion algorithms under heavy contention. In particular, in any algorithm in which processes busy-wait, the number of operations needed for one process to get to its critical section is unbounded. In order to serve as a measure of time complexity, a measure should be both intuitive and easy to compute. In sequential programming, the usual measure of time complexity, which is obtained by simply counting operations, satisfies these criteria. By contrast, there has been much disagreement on how time complexity should be measured in concurrent programs, and a complexity measure satisfying these criteria has yet to be adopted. We believe that an appropriate time complexity measure for concurrent algorithms is one based on the number of remote memory references. As seen in this paper, such a measure can be used to make meaningful distinctions concerning the performance of concurrent programs.

Acknowledgement: We would like to thank Howard Gobioff for helping with the performance studies given in Section 5. We would also like to acknowledge Argonne National Laboratories for providing us with access to the machines used in these studies. We are particularly grateful to Terry Gaasterland at Argonne for her help. We would also like to thank Nir Shavit and the program committee for their comments on an earlier draft of this paper.

References

[1] A. Agarwal and M. Cherian, "Adaptive Backoff Synchronization Techniques", *Proceedings of the 16th International Symposium on Computer Architecture*, May. 1989, pp. 396-406.

[2] J. Anderson, "A Fine-Grained Solution to the Mutual Exclusion Problem", to appear in *Acta Informatica*.

[3] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January, 1990, pp. 6-16.

[4] BBN Advanced Computers, *Inside the TC2000 Computer*, February, 1990.

[5] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[6] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 9, 1965, pp. 569.

[7] J. Goodman, M. Vernon, and P. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 64-75.

[8] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors", *IEEE Computer*, Vol. 23, June, 1990, pp. 60-69.

[9] J. Kessels, "Arbitration Without Common Modifiable Variables", *Acta Informatica*, Vol. 17, 1982, pp. 135-141.

[10] C. Kruskal, L. Rudolph, M. Snir, "Efficient Synchronization on Multiprocessors with Shared Memory", *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, October, 1988, pp. 579-601.

[11] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February, 1987, pp. 1-11.

[12] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991, pp. 21-65.

[13] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System", *Proceedings of the 9th ACM Symposium on Theory of Computing*, May, 1977, pp. 91-97.

[14] G. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, No. 3, June, 1981, pp. 115-116.

[15] E. Styer, "Improving Fast Mutual Exclusion", *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, 1992, pp. 159-168.

[16] P. Yew, N. Tzeng, and D. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors", *IEEE Transactions on Computers*, Vol. C-36, No. 4, April, 1987, pp. 388-395.

6.2 Software Combining

It is, of course, possible to implement any coarse-grained atomic operation using our mutual exclusion algorithm. However, when implementing read-modify-write operations, performance can be increased by exploiting a special characteristic possessed by many such operations, namely their combinability. For such operations, the technique of software combining can be incorporated within our algorithm to yield greater parallelism. Examples of combinable operations include fetch-and-store and fetch-and-add; Kruskal, Rudolph, and Snir provide an extensive treatment of combinable operations in [10].

The technique of software combining was first proposed by Yew, Tzeng, and Lawrie in [16]. The first (and only published) application of this technique for the purpose of implementing read-modify-write operations is given by Goodman, Vernon, and Woest in [7]. The basic idea behind software combining is to combine separate operations issued by different processes into a single operation, executing the single operation in place of those originally issued. The combining of operations and the decombining of corresponding return values take place within a "combining tree", with the former propagating up the tree, and the latter propagating down. Any algorithm that employs software combining must satisfy the following requirements. First, the algorithm must be able to detect operation requests that could potentially be combined (and should maximize the number of combined requests to the greatest extent possible). Second, the algorithm must choose a process to take charge of the combined request. Third, processes that are not selected to take charge of a combined request must be delayed until their return values are available.

For brevity, we defer a complete description of our software combining algorithm and its correctness proof to the full paper. Nonetheless, it is easy to see that our mutual exclusion algorithm has the required mechanisms for meeting the above requirements for combining. The second requirement, i.e., that of choosing a process to handle a combined request, is straightforward. In particular, by the properties of our two-process mutual exclusion algorithm, at each node in the binary arbitration tree, the process that first updates the tie-breaker variable T will enter its critical section before any other process "located within" the subtree rooted at that node. This "winning" process can therefore handle the task of servicing any request that is obtained by combining the requests of processes within the subtree. To detect those operation requests that could potentially be combined (the first requirement mentioned above), the winning process merely has to re-read the tie-breaker variable to see whether there is another process within the subtree that is waiting. Finally, note that the local-spin locations employed in the algorithm provide the necessary synchronization for decombining a combined return value into individual return values.

Although our combining algorithm is similar to that of Goodman, Vernon, and Woest, two important differences between the two bear mentioning. First, our algorithm requires only atomic read and write operations, whereas their algorithm requires a special-purpose hardware locking primitive (the QOLB primitive). Second, our algorithm employs only local spins, while theirs relies on global spinning. As such, only $O(\log_2 N)$ remote memory references are required to execute any operation in the worst case in our algorithm, whereas an unbounded number of such operations are generated in the worst case in their algorithm.

7 Concluding Remarks

We have presented a scalable mutual exclusion algorithm for shared memory multiprocessors that does not require any hardware support other than atomic read and write operations. Our algorithm has better worst-case time complexity than any previously published mutual exclusion algorithm based on read/write atomicity, requiring $O(\log_2 N)$ remote operations under any amount of contention. We have also presented extensions of our algorithm for fast mutual exclusion in the absence of contention and for incorporating the technique of software combining. In addition, we have suggested two open problems concerning the power of atomic reads and writes as synchronization primitives.

In the complexity calculations given in this paper, the distinction between remote and local operations is based upon a static assignment of shared variables to processes. Other definitions, which incorporate specific architectural details of systems, are also possible. For example, for programs intended for machines with coherent caching, it might be appropriate to consider a read of a shared variable x by a process p to be local if x has not been written by another process since p's most recent access of x. However, because of the many parameters that go into defining a cache-coherence protocol, such definitions can be problematic. We therefore choose to leave this as a subject for further study.

The tree-based approach that we employ in our algorithm can be employed to extend any two-process algorithm to an N-process one. Recently, we used this tree-based scheme to obtain an N-process algorithm using the simple two-process algorithm given by Peterson in [14]. This algorithm performs well on machines with coherent caching, as all spins in Peterson's algorithm are local on such machines. (By contrast, on distributed shared memory machines, the spins of this algorithm involve remote variables.) In fact, if critical sections are sufficiently small (in which case busy-waiting is not the dominant aspect of performance), then this algorithm outperforms ours. Its better performance in this case is

```
shared var B : array[1..N] of boolean;
            X, Y : 0..N;
            Z : boolean
initially   Y = 0  ∧  Z = false  ∧  (∀i :: B[i] = false)

process i
private var flag : boolean;
            n : 1..N

while true do
  TOP:   Noncritical Section;
         X := i;
         if Y ≠ 0 then goto SLOW fi;
         Y := i;
         if X ≠ i then goto SLOW fi;
         B[i] := true;
         if Z then goto SLOW fi;
         if Y ≠ i then goto SLOW fi;
         ENTRY2;                          /* Two-Process Entry Section */
           Critical Section;
         EXIT2;                           /* Two-Process Exit Section */
         Y := 0;
         B[i] := false;
         goto TOP;

  SLOW:  ENTRYN;                          /* Arbitration Tree */
           ENTRY2;                        /* Two-Process Entry Section */
             Critical Section;
             B[i] := false;
             if X = i then
               Z := true;
               flag := true;
               for n = 1 to N do
                 if B[n] then flag := false fi
               od;
               if flag then Y := 0 fi;
               Z := false
             fi;
           EXIT2;                         /* Two-Process Exit Section */
         EXITN                            /* Arbitration Tree */
od
```

Figure 4: Fast, scalable mutual exclusion algorithm.

Scott reported in [12] that Anderson's algorithm produced far fewer remote operations than the test-and-set algorithm.

Sequent Symmetry

The Sequent Symmetry is a shared memory multiprocessor whose processor and memory nodes are interconnected via a shared bus. A processor node consists of an Intel 80386 and a 64 Kbyte, two-way set-associative cache. Cache coherence is maintained by a snoopy protocol. The Symmetry provides an atomic fetch-and-store instruction. Because other strong primitives are not provided, we used a version of Mellor-Crummey and Scott's algorithm that is implemented with fetch-and-store and that does not ensure starvation-freedom [12]. Fetch-and-add, which is used in T. Anderson's algorithm, was simulated by a test-and-set algorithm with randomized backoff, as Anderson did in [3].

The experiments on the Symmetry show similar results to that for the TC2000. However, on the Symmetry, T. Anderson's algorithm has the best overall performance, mainly because the availability of coherent caches makes all spins in his algorithm local. The performance of Lamport's algorithm on the Symmetry is far better than that on the TC2000. This seems partly due to the fact that his algorithm is not starvation-free. Specifically, when a process enters its critical section, it can keep all needed variables in its own cache and repeatedly enter its critical section, without yielding to the other processes. In one of our tests for the two-process case, one process executed 50,000 critical sections during a period of time in which the other process executed only 120 critical sections.

Dependence on coherent caching for efficient synchronization [3, 8] is questionable, as many caching schemes do not cache shared writable data. Our solution neither requires a coherent cache for efficient implementation nor any strong primitives. An efficient implementation of our algorithm requires only that each processor has some part of shared memory that is locally accessible, and that read and write operations are atomic. We consider these to be minimal hardware requirements for efficient synchronization. It is worth noting that, without fetch-and-add and compare-and-swap primitives, T. Anderson's algorithm and Mellor-Crummey and Scott's algorithm are not starvation-free.

6 Discussion

In this section, we discuss two modified versions of our mutual exclusion algorithm. In the first modification, we alter the algorithm so that in the absence of contention only $O(1)$ remote operations are required. In the second modification, the technique of software combining is introduced for the purpose of increasing concurrency when using the algorithm to implement combinable read-modify-write operations.

6.1 Fast Mutual Exclusion

As discussed in Section 1, most early mutual exclusion algorithms based on read/write atomicity are neither fast in the absence of contention, nor able to cope with high contention. Because Lamport's fast mutual exclusion algorithm induces $O(1)$ remote operations in the absence of contention, and our mutual exclusion algorithm requires $O(\log_2 N)$ remote operations given any level of contention, it seems reasonable to expect a solution to exist that induces $O(1)$ remote operations when contention is absent, and $O(\log_2 N)$ remote operations when contention is high.

The first extension of our algorithm, which is given in Figure 4, almost achieves that goal. The basic idea of this modification is to combine Lamport's fast mutual exclusion algorithm and our scalable algorithm, specifically by placing an extra two-process version of our algorithm "on top" of the arbitration tree. The "left" entry section of this extra two-process program (i.e., process u's code in Figure 1) is executed by a process if that process detects no contention. The "right" entry section of this extra program (i.e., process v's code in Figure 1) is executed by the winning process from the arbitration tree. A process will compete within the arbitration tree (as before) if it detects any contention. As seen in Figure 4, the scheme used to detect contention is similar to that used in Lamport's algorithm.

It should be clear that, in the absence of contention, a process enters its critical section after executing $O(1)$ remote operations. Also, in the presence of contention, a process enters its critical section after executing $O(\log_2 N)$ remote operations. However, when a period of contention ends, N remote operations might be required in order to re-open the fast entry section — see the **for** loop in Figure 4. Nonetheless, performance studies show that, under high contention, these statements are rarely executed. (Under low contention, they are obviously never executed.) For example, out of the 100,000 critical section executions in one experiment, these N statements were performed after only 320 critical section executions in the two-process case, after only 55 in the four-process case, and after only one in the eight- and sixteen-process cases.

In the absence of contention, our algorithm generates about twice as many remote memory operations as Lamport's. However, under high contention, our algorithm is clearly superior, as Lamport's induces an unbounded number of remote operations. Also, our modified algorithm ensures starvation-freedom, whereas Lamport's algorithm does not.

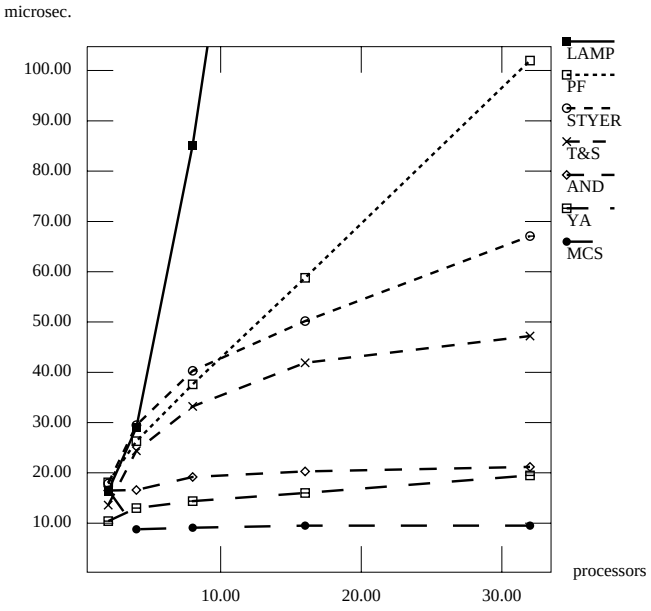

Figure 2: TC2000, small critical section.

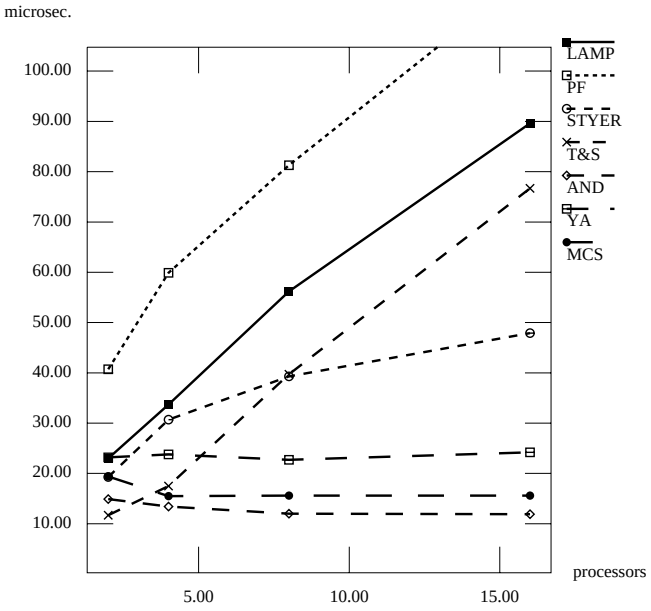

Figure 3: Symmetry, small critical section.

MCS algorithm was the best overall performer of the alternatives considered here. The graph depicted for the MCS algorithm is mostly flat, except at the point for two processors. This anomaly at two processors coincides with results reported by Mellor-Crummey and Scott on the Sequent Symmetry, and was attributed by them to the lack of a compare-and-swap instruction on the Symmetry [12]. As our implementation of their algorithm did employ compare-and-swap, we have not found a satisfying explanation for this behavior on the TC2000.

T. Anderson's algorithm requires only local spinning when implemented on a machine with coherent caches. On the Symmetry, where each process can spin on its own coherent cache, Anderson's algorithm outperforms the MCS algorithm. However, on the TC2000, which does not support coherent caching, Anderson's algorithm requires remote spinning, slowing its performance by a large margin.

The simple T&S algorithm exhibited poor scalability. Where there is a possibility of contention among a large number of processors, it should be avoided, or used with good backoff scheme [1].

Three algorithms based on atomic reads and writes – Lamport's, Peterson and Fischer's, and Styer's – also showed poor scalability. In particular, the performance of Lamport's algorithm degrades dramatically as the number of contenders increases. The average execution time for the 16 and 32 processor cases, which are not depicted in Figure 2, are 230 microseconds and 620 microseconds, respectively. The performance of Styer's algorithm, which is better than that of Lamport's, is due to the tree structure employed. Styer's algorithm generates $O(\log_2 N)$ remote operations outside of busy-waiting loops. Even though Peterson and Fischer's algorithm is also tree-based, it induces $O(N)$ remote operations outside of busy-waiting loops, which results in poorer scalability.

Our mutual exclusion algorithm shows performance that is comparable to that of T. Anderson's and Mellor-Crummey and Scott's algorithms. Its good scalability emphasizes the importance of local spinning. The difference seen between our mutual exclusion algorithm and the MCS algorithm is explained by the amount of global traffic generated by each algorithm. The MCS algorithm generates $O(1)$ remote operations per critical section execution, whereas ours generates $O(\log_2 N)$. The global traffic of the other five algorithms is unbounded, as each employs global spinning. The performance of T. Anderson's algorithm is far better than that of the simple test-and-set algorithm. Because the processes in Anderson's algorithm spin globally on the TC2000, this might be interpreted as a counterexample to our belief that minimizing remote operations is important for good scalability. However, Mellor-Crummey and

when $R > 0$ holds increments $j.pc$, and hence decreases the value of R. (I13) implies that if $i@\{8\} \wedge j@\{13\}$ holds, then $T = i$ holds. Thus, the execution of $13.j$ when $R > 0$ holds decreases the value of R. Finally, statement $14.j$ establishes $P[i] = 2$, establishing $R = 0$. □

Observe that if $i@\{8\} \wedge P[i] \geq 1$ holds then $P[i] \geq 1$ is not falsified unless $\neg i@\{8\}$ is established. (Only statement $3.i$ may falsify $P[i] \geq 1$.) Thus, by the definition of a fair history, (L0) implies that the following assertion holds.

$$i@\{8\} \ \mapsto \ \neg i@\{8\} \qquad \text{(L1)}$$

We next prove the following assertion.

$$i@\{10\} \ \mapsto \ \neg i@\{10\} \ \vee \ P[i] = 2 \qquad \text{(L2)}$$

We define a well-founded ranking R as follows:

$$R = \begin{cases} 0 & \text{if } \neg i@\{10\} \ \vee \ P[i] = 2 \\ 15 - j.pc & \text{if } i@\{10\} \ \wedge \ P[i] < 2 \end{cases}$$

By definition, R is always non-negative, $R = 0 \Rightarrow \neg i@\{10\} \vee P[i] = 2$, and $R > 0 \Rightarrow i@\{10\} \wedge P[i] < 2$. To establish that (L2) holds, it suffices to prove requirements (i) and (ii) given above.

By (I7), $R > 0$ implies that $\neg j@\{0\}$ holds. Thus, requirement (i) follows by the deadlock-freedom invariants given earlier. We now show that requirement (ii) holds. First, observe that if $R > 0$ then no statement of i is enabled. Statements of j other than $8.j$, $10.j$, $13.j$ and $14.j$ decrease the value of R if executed when $R > 0$ holds, because each statement increases $j.pc$. (I2), (I12), and (I16) imply that the execution of $8.j$ when $R > 0$ holds increments $j.pc$, and hence decreases the value of R. Also, (I11) implies that if $R > 0$ holds, then $\neg j@\{10\}$ holds. (I13) implies that if $i@\{10\} \wedge j@\{13\}$ holds, then $T = i$ holds. Thus, the execution of $13.j$ when $R > 0$ holds decreases the value of R. Finally, statement $14.j$ establishes $P[i] = 2$, establishing $R = 0$. □

Observe that if $i@\{10\} \wedge P[i] = 2$ holds then $P[i] = 2$ is not falsified unless $\neg i@\{10\}$ is established. (Only statements $3.i$ and $7.j$ may falsify $P[i] = 2$. However, (I3) implies that statement $7.j$ does not falsify $P[i] = 2$.) Thus, by the definition of a fair history, (L2) implies that the following assertion holds.

$$i@\{10\} \ \mapsto \ \neg i@\{10\} \qquad \text{(L3)}$$

By (L1), (L3), the program text, and the definition of a fair history, we conclude that the program in Figure 1 is free from starvation.

5 Performance Results

To compare the scalability of our mutual exclusion algorithm with that of other algorithms, we conducted a number of experiments on the BBN TC2000 and Sequent Symmetry multiprocessors. Results from some of those experiments are presented in this section. Results from additional experiments will be presented in the full paper.

BBN TC2000

The BBN TC2000 is a distributed shared memory multiprocessor, each node of which contains a processor and a memory unit. The nodes are connected via a multistage interconnection network, known as the Butterfly switch. Each access to a remote memory location (i.e., one that requires a traversal of the interconnection network) takes about 2 microseconds, whereas each local reference takes about 0.6 microseconds. Each node's processor, a Motorola 88100, provides an atomic fetch-and-store instruction called `xmem`. Other strong primitives such as compare-and-swap and fetch-and-add are provided using the TC2000 hardware locking protocol [4]. The TC2000 has cache memory, but does not provide a cache coherence mechanism.

We tested seven mutual exclusion algorithms on the TC2000: a simple test-and-set algorithm; the queue-based algorithm using compare-and-swap given by Mellor-Crummey and Scott in [12]; the queue-based algorithm using fetch-and-add given by T. Anderson in [3]; the fast mutual exclusion algorithm given by Lamport in [11]; the tree-based algorithm given by Styer in [15]; the tree-based algorithm given by Peterson and Fischer in [13]; and the mutual exclusion algorithm described in Section 3. Performance results obtained by running these seven algorithms on the TC2000 are summarized in Figure 2. Each point (x, y) in each graph represents the average time y for one critical section execution with x competing processors. The timing results summarized in the graph were obtained by averaging over 10^5 critical section executions. The critical section consists of a read and an increment of shared counter. Results obtained using larger critical sections, which for brevity are not presented here, show similar performance to that depicted in Figure 2. The timing results presented include the execution time of critical sections.

The performance of the test-and-set algorithm is given by the graph labeled T&S, Mellor-Crummey and Scott's algorithm by the graph labeled MCS, T. Anderson's algorithm by the graph labeled AND, Lamport's algorithm by the graph labeled LAMP, Styer's algorithm by the graph labeled STYER, Peterson and Fischer's algorithm by the graph labeled PF, and our algorithm by the graph labeled YA. On the TC2000, the

antecedent may be established only by statements $5.i$, $6.j$, $7.j$, and $2.j$. Only statement $3.i$ may falsify the consequent. Statement $5.i$ establishes $i@\{6\}$ only when $T \neq j$ holds. Statement $6.j$ can establish $j@\{8\}$ only when $P[i] \geq 1$ holds. Statement $7.j$ establishes $P[i] = 1$. When statement $2.j$ establishes $T = j$, it also establishes $j@\{3\}$. When statement $3.i$ falsifies $P[i] \geq 1$, it establishes $i@\{4\}$. □

invariant $i@\{11..14, 0..2\} \wedge j@\{3..10\} \Rightarrow T = j$ (I13)

Initially $j@\{3..10\}$ is false, and hence (I13) is true. The antecedent may be established only by statements $4.i$, $5.i$, $9.i$, $10.i$, and $2.j$. The consequent may be falsified only by statement $2.i$. (I1) implies that statement $4.i$ establishes $i@\{11\}$ only if $\neg j@\{3..10\}$ holds. By (I2), statements $5.i$ and $9.i$ establish $i@\{11\}$ only when $T = j$ holds. Statement $10.i$ establishes the antecedent only when $P[i] = 2 \wedge j@\{3..10\}$ holds. In the resulting state, $i@\{11\} \wedge P[i] = 2 \wedge j@\{3..10\}$ holds. This implies, by (I1) and (I5), that $T = j$ holds. When statement $2.j$ establishes $j@\{3\}$, it also establishes $T = j$. Finally, statement $2.i$ establishes $i@\{3\}$. □

invariant $i@\{0..2\} \wedge j@\{4..8\} \Rightarrow P[j] = 0 \vee P[j] = 2$ (I14)

Initially $j@\{4..8\}$ is false, and hence (I14) is true. The antecedent may be established only by statements $13.i$, $14.i$, and $3.j$. The consequent may be falsified only by statement $7.i$. Statement $13.i$ may establish the antecedent only when $i@\{13\} \wedge j@\{4..8\} \wedge T = i$ holds. However, $i@\{13\} \wedge j@\{4..8\}$ implies, by (I13), that $T = j$ holds. Thus, statement $13.i$ does not establish the antecedent. Statement $14.i$ establishes $P[j] = 2$. When statement $3.j$ establishes $j@\{4\}$, it also establishes $P[j] = 0$. Finally, statement $7.i$ establishes $i@\{8\}$. □

invariant $i@\{0..2\} \wedge j@\{9, 10\} \Rightarrow P[j] = 2$ (I15)

Initially $j@\{9, 10\}$ is false, and hence (I15) is true. The antecedent may be established only by statements $13.i$, $14.i$, and $8.j$. The consequent may be falsified only by statements $7.i$ and $3.j$. Statement $13.i$ may establish the antecedent only when $i@\{13\} \wedge j@\{9, 10\} \wedge T = i$ holds. However, $i@\{13\} \wedge j@\{10\}$ implies, by (I13), that $T = j$ holds. Thus, statement $13.i$ does not establish the antecedent. Statement $14.i$ establishes $P[j] = 2$. Statement $8.j$ establishes the antecedent only when $i@\{0..2\} \wedge j@\{8\} \wedge P[j] > 0$ holds. By (I14), this implies that $P[j] = 2$ holds. The consequent could be falsified only by statements $7.i$ or $3.j$. However, statement $7.i$ establishes $i@\{8\}$, and statement $3.j$ establishes $j@\{4\}$. □

invariant $i@\{3..8\} \wedge j@\{10\} \wedge T = i \Rightarrow P[j] = 2$ (I16)

Initially $i@\{3..8\}$ is false, and hence (I16) is true. The antecedent may be established only by statements $2.i$ and $9.j$. The consequent could be falsified only by statements $7.i$ or $3.j$. Statement $2.i$ establishes the antecedent only when $i@\{2\} \wedge j@\{10\}$ holds, which, by (I15), implies that $P[j] = 2$ holds. Statement $9.j$ establishes $j@\{10\}$ only when $T = j$ holds. Thus, statement $9.j$ does not establish the antecedent. (I3) implies that statement $7.i$ does not falsify $P[j] = 2$. Statement $3.j$ establishes $j@\{4\}$. □

(I7) implies that if only one of i and j is in its entry section, then that process's busy-waiting loop will terminate. (I8), (I11), (I12), and (I16) imply that if both processes are in their entry sections then at least one of them can make progress. Hence, we conclude that the program in Figure 1 is free from deadlock.

Now, we prove that the algorithm is free from starvation. We begin by proving the following assertion.

$$i@\{8\} \mapsto \neg i@\{8\} \vee P[i] \geq 1 \quad \text{(L0)}$$

Let $j.pc$ denote the program counter of process j; i.e., $j.pc = k$ iff $j@\{k\}$ holds. We define a well-founded ranking R as follows:

$$R = \begin{cases} 0 & \text{if } \neg i@\{8\} \vee P[i] \geq 1 \\ 15 - j.pc & \text{if } i@\{8\} \wedge P[i] = 0 \end{cases}$$

By definition, R is always non-negative, $R = 0 \Rightarrow \neg i@\{8\} \vee P[i] \geq 1$, and $R > 0 \Rightarrow i@\{8\} \wedge P[i] = 0$. For the sake of the proof, we say that a statement is *enabled* iff its execution changes the corresponding process's program counter. (Only statements $8.j$ and $10.j$ of process j may be disabled – the former when $P[j] = 0$ holds, and the latter when $P[j] \leq 1$ holds.) To establish that (L0) holds, it suffices to prove that if $R > 0$, then (i) there exists some enabled statement other than $0.j$ (note that process j may halt in its noncritical section), and (ii) the execution of any enabled statement decreases the value of R.

By (I7), $R > 0$ implies that $\neg j@\{0\}$ holds. Thus, requirement (i) follows by the deadlock-freedom invariants given earlier. We now show that requirement (ii) holds. First, observe that if $R > 0$ then no statement of i is enabled. Statements of j other than $8.j$, $10.j$, $13.j$, and $14.j$ decrease the value of R if executed when $R > 0$ holds, because each such statement increases $j.pc$. (I8) implies that the execution of $8.j$ when $R > 0$ holds increments $j.pc$, and hence decreases the value of R. Also, (I2), (I12), and (I16) imply that the execution of $10.j$

invariant $\quad i@\{4..12\} \ \wedge \ P[i] = 2 \ \wedge \ C[j] = j \ \Rightarrow$
$T = j \ \vee \ j@\{2\}$ (I4)

Initially $i@\{4..12\}$ is false, and hence (I4) is true. To prove that (I4) is not falsified, it suffices to consider only those statements that may establish the antecedent or falsify the consequent. The antecedent may be established only by statements $3.i$, $14.j$, and $1.j$. The consequent may be falsified only by statements $2.i$ and $2.j$. When statement $3.i$ establishes $i@\{4..12\}$, it also falsifies $P[i] = 2$. When statement $14.j$ establishes $P[i] = 2$, it also establishes $j@\{0\}$, which, by (I0), implies that $C[j] = 0$ holds. When statement $1.j$ establishes $C[j] = j$, it also establishes $j@\{2\}$. Only statement $2.i$ falsifies $T = j$, and it establishes $i@\{3\}$. Finally, statement $2.j$ establishes $T = j$. □

invariant $\quad i@\{11..14, 0..2\} \ \wedge \ C[j] = j \ \Rightarrow$
$T = j \ \vee \ j@\{2\}$ (I5)

Initially $C[j] = j$ is false, and hence (I5) is true. The antecedent may be established only by statements $4.i$, $5.i$, $9.i$, $10.i$, and $1.j$. The consequent may be falsified only by statements $2.i$ and $2.j$. Statement $4.i$ establishes $i@\{11\}$ only when $C[j] = 0$ holds. Statements $5.i$ and $9.i$ establish $i@\{11\}$ only when $T \neq i$ holds, which, by (I2), implies that the consequent holds. Statement $10.i$ establishes the antecedent only when $P[i] = 2 \ \wedge \ C[j] = j$ holds. In the resulting state, $i@\{11\} \ \wedge \ P[i] = 2 \ \wedge \ C[j] = j$ holds, which, by (I4), implies that the consequent holds. $C[j] = j$ can be established only by statement $1.j$. When statement $1.j$ establishes $C[j] = j$, it also establishes $j@\{2\}$. Only statement $2.i$ can falsify $T = j$, and it establishes $i@\{3\}$. Finally, statement $2.j$ establishes $T = j$. □

We now prove that the mutual exclusion property holds.

invariant $\quad \neg i@\{11\} \ \vee \ \neg j@\{11\}$ (I6)

To see that mutual exclusion is guaranteed, assume $i@\{11\} \wedge j@\{11\}$. Then, by (I1), $i@\{11\} \wedge j@\{11\} \wedge C[i] = i \ \wedge \ C[j] = j$ holds. By (I5), this implies that $T = i \ \wedge \ T = j$ holds, which is a contradiction. Thus, (I6) is an invariant. □

Next, we prove a few invariants that are needed to establish that starvation-freedom holds.

invariant $\quad i@\{0, 1\} \ \wedge \ j@\{5..10\} \ \Rightarrow \ P[j] = 2$ (I7)

Initially $j@\{5..10\}$ is false, and hence (I7) is true. The antecedent may be established only by statements $13.i$, $14.i$, and $4.j$. The consequent may be falsified only by statements $7.i$ and $3.j$. Statement $13.i$ establishes the antecedent only when $T = i \ \wedge \ j@\{5..10\}$ holds. In the resulting state, $i@\{0\} \ \wedge \ T = i \ \wedge \ j@\{5..10\}$ holds, which, by (I1), implies that $i@\{0\} \ \wedge \ C[j] = j \ \wedge \ T \neq j$ holds. By (I5), this implies that $\neg j@\{5..10\}$. Hence, statement $13.i$ cannot establish the antecedent. Statement $14.i$ establishes $P[j] = 2$. Statement $4.j$ establishes $j@\{5..10\}$ only if $C[i] \neq 0$ holds, which, by (I0), implies that $\neg i@\{0, 1\}$ holds. The consequent could be falsified only by statements $7.i$ or $3.j$. However, statement $7.i$ establishes $i@\{8\}$, and statement $3.j$ establishes $j@\{4\}$. □

invariant $\quad i@\{8\} \ \wedge \ j@\{8\} \ \Rightarrow$
$P[i] \geq 1 \ \vee \ P[j] \geq 1$ (I8)

Initially $i@\{8\}$ is false, and hence (I8) is true. The statements that can establish the antecedent are statements $6.i$, $7.i$, $6.j$, and $7.j$. The consequent may be falsified only by statements $3.i$ and $3.j$. Statement $6.i$ can establish $i@\{8\}$ only if $P[j] \neq 0$ holds. Statement $7.i$ establishes $P[j] = 1$. Only statement $3.i$ can falsify $P[i] \geq 1$, and it establishes $i@\{4\}$. Similarly, statements $3.j$, $6.j$, and $7.j$ preserve (I8). □

invariant $\quad i@\{9, 10\} \ \Rightarrow \ P[i] \ \geq \ 1$ (I9)

Initially $i@\{9, 10\}$ is false, and hence (I9) is true. Statement $8.i$ establishes $i@\{9\}$ only if $P[i] \geq 1$ holds. Only statement $3.i$ falsifies $P[i] \geq 1$, and it establishes $i@\{4\}$. □

invariant $\quad T = j \ \wedge \ j@\{4..10\} \ \Rightarrow$
$\neg i@\{10\} \ \vee \ P[j] = 0$ (I10)

Initially $j@\{4..10\}$ is false, and hence (I10) is true. The antecedent may be established only by statements $2.j$ and $3.j$. The consequent may be falsified only by statements $9.i$, $7.i$, and $14.i$. Statement $2.j$ establishes $j@\{3\}$. Statement $3.j$ establishes $P[j] = 0$. $i@\{10\}$ can be established by statement $9.i$ only when $T = i$. Only statements $7.i$ and $14.i$ falsify $P[j] = 0$, and they establish $\neg i@\{10\}$. □

invariant $\quad \neg i@\{10\} \ \vee \ \neg j@\{10\}$ (I11)

Assume that $i@\{10\} \ \wedge \ j@\{10\}$ holds. Then, by (I9), $i@\{10\} \ \wedge \ j@\{10\} \ \wedge \ P[i] \geq 1 \ \wedge \ P[j] \geq 1$ holds, which, by (I2), implies that $i@\{10\} \ \wedge \ j@\{10\} \ \wedge \ P[i] \geq 1 \ \wedge \ P[j] \geq 1 \ \wedge \ (T = i \ \vee \ T = j)$ holds. By (I10), this leads to a contradiction. Thus, (I11) is an invariant. □

invariant $\quad i@\{6..8\} \ \wedge \ j@\{8..10\} \ \wedge \ T = j \ \Rightarrow$
$P[i] \geq 1$ (I12)

Initially $i@\{6..8\}$ is false, and hence (I12) is true. The

```
shared var C : array[u, v] of {0, u, v};
            P : array[u, v] of 0..2;
            T : {u, v}
initially   C[u] = 0 ∧ C[v] = 0 ∧ P[u] = 0 ∧ P[v] = 0
```

process u

```
while true do
   0: Noncritical Section;
   1: C[u] := u;
   2: T := u;
   3: P[u] := 0;
   4: if C[v] ≠ 0 then
   5:    if T = u then
   6:       if P[v] = 0 then
   7:          P[v] := 1 fi;
   8:       while P[u] = 0 do /* null */ od;
   9:       if T = u then
  10:          while P[u] ≤ 1 do /* null */ od fi
         fi
      fi;
  11: Critical Section;
  12: C[u] := 0;
  13: if T ≠ u then
  14:    P[v] := 2 fi
od
```

process v

```
while true do
   0: Noncritical Section;
   1: C[v] := v;
   2: T := v;
   3: P[v] := 0;
   4: if C[u] ≠ 0 then
   5:    if T = v then
   6:       if P[u] = 0 then
   7:          P[u] := 1 fi;
   8:       while P[v] = 0 do /* null */ od;
   9:       if T = v then
  10:          while P[v] ≤ 1 do /* null */ od fi
         fi
      fi;
  11: Critical Section;
  12: C[v] := 0;
  13: if T ≠ v then
  14:    P[u] := 2 fi
od
```

Figure 1: Two-process mutual exclusion algorithm.

verses this path in reverse, this time executing the exit section of each link. For brevity, we defer a complete description of the N-process solution and its correctness proof to the full paper. Note that if variable $P[i]$ is local to process i in the two process algorithm, then process i executes a constant number of remote operations in its entry and exit sections. It follows that, in the N-process algorithm, each process executes $O(\log_2 N)$ remote operations in its (N-process) entry and exit sections.

4 Correctness Proof

As explained above, the N-process solution is obtained by associating an instance of the two process solution of Figure 1 with each internal node of a binary arbitration tree. By induction on the depth of the tree, it can be shown that the mutual exclusion and starvation-freedom properties hold for the N-process program provided they hold for the two-process program. A detailed version of this inductive argument will be given in the full paper. In the rest of this section, we prove that the mutual exclusion and starvation-freedom properties hold for the two-process program.

We begin by presenting notational conventions that will be used in the remainder of the paper.

Notational Conventions: Unless specified otherwise, we assume that i and j each range over $\{u, v\}$ and that $i \neq j$. We denote statement number k of process i as $k.i$. Let S be a subset of the statement labels in process i. Then, $i@\{S\}$ holds iff the program counter for process i equals some value in S. The following is a list of symbols we will use ordered by increasing binding power: $\equiv$, $\mapsto$, $\Rightarrow$, $\vee$, $\wedge$, $(=, \neq, >, <, \geq, \leq)$, $+$, $\neg$, $(., @)$. The symbols enclosed in parentheses have the same priority. We sometimes use parentheses to override this binding rule. □

The following invariants, which are stated without proof, follow directly from the program text.

invariant $\quad i@\{0, 1, 13, 14\} \;=\; (C[i] = 0)$ (I0)

invariant $\quad i@\{2..12\} \;=\; (C[i] = i)$ (I1)

invariant $\quad i@\{3..14\} \;\vee\; j@\{3..14\} \;\Rightarrow\; T = i \;\vee\; T = j$ (I2)

invariant $\quad i@\{7\} \;\Rightarrow\; P[j] = 0$ (I3)

We next prove two invariants that are needed to establish the mutual exclusion property.

be accessed by more than one process. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k. For an example of the syntax we use for programs, see Figure 1.

A program's semantics is defined by its set of "fair histories". As defined formally in the full paper, a *history* is a structure that represents a single execution of a program. In the usual way, we represent a history of a program as a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state of the program and $t_i \xrightarrow{s_i} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of statement s_i. Informally, a history of a program is *fair* if it is finite, with no statement enabled for execution in its last state, or if it is infinite, and each statement of the program is either infinitely often disabled for execution in the history or is infinitely often executed in the history. Note that this fairness requirement corresponds to weak fairness. Unless otherwise noted, we henceforth assume that all histories are fair.

When reasoning about the correctness of a concurrent program, safety properties are defined using invariants and progress properties are defined using leads-to assertions [5]. An assertion B (over program variables) is an *invariant* of a program iff B holds in each state of every history of the program. We say that predicate B *leads-to* predicate C in a program, denoted $B \mapsto C$, iff for each history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ of the program, if B is true at some state t_i, then C is true at some state t_j where $j \geq i$.

3 Mutual Exclusion Algorithm

In this section, we present our mutual exclusion algorithm. As in [9], we first solve the mutual exclusion problem for two processes, and then apply our two-process solution in a binary arbitration tree to get an N-process solution. The two-process algorithm is depicted in Figure 1. The two processes are denoted u and v, which are assumed to be distinct, positive integer values.

The algorithm employs five shared variables, $C[u]$, $C[v]$, T, $P[u]$, and $P[v]$. Variable $C[u]$ ranges over $\{0, u, v\}$ and is used by process u to inform process v of its intent to enter its critical section. Observe that $C[u] = u \neq 0$ holds while process u executes its statements 2 through 12, and $C[u] = 0$ holds otherwise. Variable $C[v]$ is used similarly. Variable T ranges over $\{u, v\}$ and is used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. The algorithm ensures that the two processes enter their critical sections according to the order in which they update T. Variable $P[u]$ ranges over $\{0, 1, 2\}$ and is used by process u whenever it needs to busy-wait. Note that $P[u]$ is waited on only by process u, and thus can be stored in a memory location that is locally accessible to process u (in which case all spins are local). Variable $P[v]$ is used similarly by process v.

Loosely speaking, the algorithm works as follows. When process u wants to enter its critical section, it informs process v of its intention by establishing $C[u] = u$. Then, process u assigns its identifier u to the tie-breaker variable T, and initializes its spinning location $P[u]$. If process v has not shown interest in entering its critical section, in other words, if $C[v] = 0$ holds when u executes statement 4, then process u proceeds directly to its critical section. Otherwise, u reads the tie-breaker variable T. If $T \neq u$, which implies that $T = v$, then u can enter its critical section, as the algorithm prohibits v from entering its critical section when $C[u] = u \wedge T = v$ holds (recall that ties are broken in favor of the first process to update T). If $T = u$ holds, then either process v executed statement 2 before process u, or process v has executed statement 1 but not statement 2. In the first case, u should wait until v exits its critical section, whereas, in the second case, u should be able to proceed to its critical section. This ambiguity is resolved by having process u execute statements 6 through 10. Statements 6 and 7 are executed by process u to release process v in the event that it is waiting for u to update the tie-breaker variable (i.e., v is busy-waiting at statement 8). Statements 8 through 10 are executed by u to determine which process updated the tie-breaker variable first. Note that $P[u] \geq 1$ implies that v has already updated the tie-breaker, and $P[u] = 2$ implies that v has finished its critical section. To handle these two cases, process u first waits until $P[u] \geq 1$ (i.e., until v has updated the tie-breaker), re-examines T to see which process updated T last, and finally, if necessary, waits until $P[u] = 2$ (i.e., until process v finishes its critical section).

After executing its critical section, process u informs process v that it is finished by establishing $C[u] = 0$. If $T = v$, in which case process v is waiting to enter its critical section, then process u updates $P[v]$ in order to terminate v's busy-waiting loop. In Section 4, we formally prove that our algorithm guarantees the mutual exclusion property and is free from starvation.

As discussed above, the N-process case is solved by applying the above two-process algorithm in a binary arbitration tree. Associated with each link in the tree is an entry section and an exit section. The entry and exit sections associated with the two links connecting a given node to its sons constitute a two-process mutual exclusion algorithm. Initially, all processes start at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each link on this path. Upon exiting its critical section, a process tra-

cessed. Recently, several queue-based mutual exclusion algorithms based on read-modify-write operations have been proposed in which this type of busy-waiting is avoided [3, 8, 12]. These algorithms exhibit good scalability when used on multiprocessors that permit shared variables to be locally accessible, as is the case if coherent caching schemes are employed, or if shared variables can be allocated in a local portion of distributed shared memory. The key to their good performance is the idea of "local spinning", i.e., busy-waiting on variables that are locally-accessible to the waiting process. By relying on local spinning as the sole mechanism by which processes wait, the number of remote operations is kept to a minimum.

In a recent paper [2], Anderson presented a mutual exclusion algorithm that uses only local spins and that requires only atomic read and write operations. In his algorithm, each process is required to perform $O(N)$ remote operations to enter its critical section whether there is contention or not, where N is the number of processes. (The architectural distinction between remote and local shared memory can be abstracted in a variety of ways for the purpose of defining the time complexity of concurrent programs. The abstraction adopted in this paper is based upon a static assignment of shared variables to processes: an operation of a process is *remote* if it accesses variables that are not assigned to that process, and is *local* otherwise. As explained in Section 7, other abstractions, which incorporate specific architectural details of systems such as coherent caching schemes, are also possible.) All other previously published mutual exclusion algorithms that are based on atomic reads and writes employ global busy-waiting and hence induce an unbounded number of remote operations under heavy contention. Most such algorithms also require $O(N)$ remote operations in the absence of contention. Some exceptions to the latter include algorithms given by Kessels in [9] and Lamport in [11]. Kessels' algorithm generates $O(\log_2 N)$ remote operations in the absence of contention, while Lamport's generates $O(1)$. A variant of Lamport's algorithm has recently been presented by Styer in [15]. Although Styer claims that his algorithm is more scalable than Lamport's, in terms of time complexity, they are actually very similar: both generate unbounded remote operations under heavy contention and $O(1)$ operations in the absence of contention. Styer's claims of scalability are predicated upon complexity calculations that ignore operations performed during busy-waiting loops. Because the processes in his algorithm busy-wait on remote variables, such complexity calculations do not give a true indication of scalability.

In this paper, we present a new mutual exclusion algorithm that requires only atomic reads and writes and in which all spins are local. Our algorithm induces $O(\log_2 N)$ remote operations under any amount of contention, and thus is an improvement over the algorithm given by Anderson in [2]. We also present two modified versions of this algorithm. The first modification results in an algorithm that requires only $O(1)$ remote operations in the absence of contention. Unfortunately, in this modified algorithm, worst-case complexity rises to $O(N)$. However, we argue that this $O(N)$ behavior is rare, occurring only when transiting from a period of high contention to a period of low contention. Under high contention, this modified algorithm induces only $O(\log_2 N)$ remote operations. In the second modified version of our basic algorithm, the technique of software combining is introduced for the purpose of increasing concurrency when using the algorithm to implement combinable read-modify-write operations. It is worth noting that our basic algorithm and its variations are starvation-free, whereas some of the aforementioned algorithms are not.

The results of this paper suggest two important open problems. First, note that the queue-based algorithms in [3, 8, 12] require $O(1)$ remote operations, whether there is contention or not (actually the algorithms in [3, 8] have $O(1)$ complexity only if coherent caches are provided). As our algorithm requires $O(\log_2 N)$ remote operations, it is natural to ask whether this gap is due to a fundamental weakness of atomic reads and writes, or whether there is a mutual exclusion algorithm based on such operations that has $O(1)$ complexity (which seems doubtful). If it turns out that the lower bound for mutual exclusion under read/write atomicity is $\Omega(\log_2 N)$, then a second question arises: namely, is it possible to develop a mutual exclusion algorithm under read/write atomicity that requires $O(1)$ remote operations in the absence of contention and $O(\log_2 N)$ remote operations in the presence of contention?

The rest of the paper is organized as follows. In Section 2, we present our model of concurrent programs. The above-mentioned mutual exclusion algorithm is then presented in Section 3, and its correctness proof is given in Section 4. In Section 5, we present results from performance studies conducted on the BBN TC2000 and Sequent Symmetry multiprocessors. In Section 6, we consider the two modified versions of the algorithm discussed above. Concluding remarks appear in Section 7.

2 Concurrent Programs

A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements. Each *variable* of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may

Fast, Scalable Synchronization with Minimal Hardware Support
(Extended Abstract)

Jae-Heon Yang* James H. Anderson*

Department of Computer Science
The University of Maryland at College Park
College Park, Maryland 20742

Abstract

This paper concerns synchronization under read/write atomicity in shared memory multiprocessors. We present a new algorithm for N-process mutual exclusion that requires only read and write operations and that has $O(\log_2 N)$ time complexity, where "time" is measured by counting remote memory references. The time complexity of this algorithm is better than that of all prior solutions to the mutual exclusion problem that are based upon atomic read and write instructions; in fact, the time complexity of most prior solutions is unbounded. Performance studies are presented that show that our mutual exclusion algorithm exhibits scalable performance under heavy contention. In the second part of the paper, we discuss two extensions of our mutual exclusion algorithm. In the first extension, we modify the algorithm so that in the absence of contention only $O(1)$ memory references are required. In the second extension, the technique of software combining is introduced for the purpose of increasing concurrency when using the algorithm to implement combinable read-modify-write operations.

1 Introduction

The mutual exclusion problem is a paradigm for resolving conflicting accesses to shared resources and has been studied for many years, dating back to the seminal paper of Dijkstra [6]. In this problem, each of a set of processes repeatedly executes a program fragment known as its "critical section". Before and after executing its critical section, a process executes two other program fragments, its "entry section" and "exit section", respectively. The entry and exit sections must be designed so that (i) at most one process executes its critical section at any time, and (ii) each process in its entry section eventually executes its critical section. The former is known as the *mutual exclusion* property, and the latter as the *starvation-freedom* property. In some variants of the problem, starvation-freedom is replaced by the weaker requirement of *livelock-freedom*: if some process is in its entry section, then some process eventually executes its critical section.

Most early solutions to the mutual exclusion problem required only minimal hardware support, specifically atomic read and write instructions. Although of theoretical importance, most such algorithms were judged to be impractical from a performance standpoint, leading to the development of solutions requiring stronger hardware support such as read-modify-write operations. The poor performance of the former stems partially from two factors. First, such algorithms are not scalable, i.e., performance degrades dramatically as the number of contending processes increases. Second, even in the absence of contention, such algorithms require a process contending for its critical section to execute many operations. Although the second problem has been subsequently addressed, specifically by Lamport in [11], the first has not. In this paper, we address this problem by considering the important question of whether poor performance under heavy contention is an inherent aspect of synchronization under read/write atomicity.

Many mutual exclusion algorithms that exhibit poor scalability do so, in part, because they require processes to busy-wait on shared variables that are remotely-accessible, i.e., that require a traversal of the global interconnect between processors and memory when ac-

*Work supported, in part, by NSF Contract CCR 9109497 and by the Center for Excellence in Space Data and Information Sciences. E-mail: {jhyang, jha}@cs.umd.edu.