Fast Mutual Exclusion, Even With Contention*

Maged M. Michael Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY 14627-0226

{michael,scott}@cs.rochester.edu

June 1993

Abstract

We present a mutual exclusion algorithm that performs well both with and without contention, on machines with no atomic instructions other than read and write. The algorithm capitalizes on the ability of memory systems to read and write at both full- and half-word granularities. It depends on predictable processor execution rates, but requires no bound on the length of critical sections, performs only $O(n)$ total references to shared memory when arbitrating among conflicting requests (rather than $O(n^2)$ in the general version of Lamport's fast mutual exclusion algorithm), and performs only 2 reads and 4 writes (a new lower bound) in the absence of contention. We provide a correctness proof.

We also investigate the utility of exponential backoff in fast mutual exclusion, with experimental results on the Silicon Graphics Iris multiprocessor and on a larger, simulated machine. With backoff in place, we find that Lamport's algorithm, our new algorithm, and a recent algorithm due to Alur and Taubenfeld all work extremely well, outperforming the native hardware locks of the Silicon Graphics machine, even with heavy contention.

1 Introduction

Many researchers have addressed the problem of n-process mutual exclusion under a shared-memory programming model in which reads and writes are the only atomic operations. Early solutions to the problem entail a lock acquisition/release protocol in which each process that wishes to execute the critical section makes $\Omega(n)$ references to shared memory, where n is the total number of processes [3, 7].

On the assumption that contention is relatively rare, Lamport in 1987 suggested two mutual exclusion algorithms [4] in which a process performs only a constant number of shared memory references in its acquisition/release protocol, so long as no other process attempts to do so simultaneously. The first algorithm requires a bound on the length of a critical section (which is not always possible), and a bound on the relative rates of process execution. The second algorithm performs $\Omega(n^2)$ total references to shared memory ($\Omega(n)$ in each process) when attempting to arbitrate among n concurrent lock acquisition attempts (see section 2).

We have developed an algorithm that retains the $O(1)$ bound of Lamport's algorithms in the absence of contention, while arranging to elect a winner after only $O(n)$ shared memory references in the presence of contention. Like Lamport's first algorithm, the new algorithm requires a bound on relative rates of process execution; it does not however require a bound on the length of a critical section.

Several researchers have recently presented algorithms with similar characteristics. These are summarized in table 1. The first column of the table indicates the number of references a process makes to shared memory

*This work was supported in part by NSF Institutional Infrastructure award number CDA-8822724, NSF grant number CCR-9005633, and ONR research contract number N00014-92-J-1801 (in conjunction with the ARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order No. 8930).

Algorithm	shared memory references to choose winner		needs speed bound?	comments
	no contention	contention		
Lamport 1 [4]	2 reads, 3 writes	$O(n)$	yes	requires bound on critical section length
Lamport 2 [4]	2 reads, 5 writes	$\Omega(n^2)$	no	
Styer [8]	3 reads, $(4+l)$ writes	$\Omega(ln^{2/l})$	no	l can be chosen anywhere in $(O(1), O(\log n))$
Yang and Anderson 1 [10]	$O(\log n)$	$O(\log n)$	no	starvation free no remote spins
Yang and Anderson 2 [10]	6 reads, 9 writes[a]	$O(n)$ [$O(\log n)$ "typical"]	no	starvation free no remote spins
Alur and Taubenfeld [1]	3 reads, 5 writes	$O(n)$	yes	
new	2 reads, 4 writes	$O(n)$	yes	requires multi-grain atomic reads and writes

Table 1: Comparative characteristics of fast mutual exclusion algorithms.

[a] With appropriate assignment of variables to local memory locations, 5 of the 9 writes need not traverse the processor-memory interconnection network.

when acquiring and releasing a lock for which there is no contention. The second column indicates the number of references that may need to execute sequentially in order for some process to enter its critical section when n processes wish to do so. (This notion of "time" differs from that of most other researchers; we assume that references may serialize if they are made by the same process or require the use of the same memory bank or communication link.) Our algorithm performs fewer shared memory references than any but the bounded-critical-section version of Lamport's algorithm. It is also substantially simpler than the algorithms of Styer or Yang and Anderson, both of which employ a hierarchical collection of sub-n-process locks. There is a strong resemblance between our algorithm and that of Alur and Taubenfeld, though the two were developed independently. In effect, we reduce the number of shared-memory operations by exploiting the ability of most memory systems to read and write atomically at both full- and half-word granularities.

Following the presentation of our algorithm in section 2, we present a correctness proof in section 3, experimental performance results in section 4, and conclusions in section 5. In our experiments, we employ limited exponential backoff to reduce the amount of contention caused by concurrent attempts to acquire a lock. This technique, originally suggested by T. Anderson, works very well for `test_and_set` locks [2, 5], and our results show it to be equally effective for locks based on reads and writes. In fact, on our Silicon Graphics multiprocessor, fast mutual exclusion algorithms with backoff (and the new algorithm in particular) outperform the native hardware spin locks by a significant margin, with or without contention. Results on a larger, simulated machine also show the new algorithm outperforming both Lamport's second algorithm and Alur and Taubenfeld's algorithm.

2 Algorithms

Lamport [4] presents two mutual exclusion algorithms. Both allow a process to enter its critical section in constant time. The first algorithm requires a bound on the relative rates of execution of different processes, and on the time required to execute critical sections. In the absence of contention a process requires five accesses to shared memory to acquire and release the lock. Process i executes the code on the left side of figure 1. Variable Y is initialized to `free`, and the delay in line 7 is assumed to be long enough for any process that has already read $Y =$ `free` in line 3 to complete lines 5, 6, and (if appropriate) 10 and 11.

The second algorithm does not require any bounds on execution rates or lengths of critical sections. In the absence of contention a process requires seven accesses to shared memory to acquire and release the lock,

```
1:  START:
2:        X ← i
3:        if Y ≠ free
4:              goto START
5:        Y ← i
6:        if X ≠ i
7:              { delay }
8:              if Y ≠ i
9:                    goto START
10: { critical section }
11:       Y ← free
12: { non-critical section }
13:       goto START
```

```
1:  START:
2:        B[i] ← true
3:        X ← i
4:        if Y ≠ free
5:              B[i] ← false
6:              repeat until Y = free
7:              goto START
8:        Y ← i
9:        if X ≠ i
10:             B[i] ← false
11:             for j ← 1 to N
12:                   repeat while B[j]
13:             if Y ≠ i
14:                   repeat until Y = free
15:                   goto START
16: { critical section }
17:       Y ← free
18:       B[i] ← false
19: { non-critical section }
20:       goto START
```

Figure 1: Lamport's fast mutual exclusion algorithms.

and $O(n^2)$ time with contention.[1] Process i executes the code on the right side of figure 1. Variable Y is initialized to `free` and each element of the B array is initialized to `false`.

We have devised a new mutual exclusion algorithm that allows a process to enter its critical section with only six shared memory references in the absence of contention. In the presence of contention, it requires $O(n)$ time. As in Lamport's first algorithm, we assume a bound on relative rates of process execution. Such an assumption is permissible if the algorithm is executed by an embedded system, or by an operating system routine that executes with hardware interrupts disabled. We do not, however, require a bound on the length of critical sections. Process i in our algorithm executes the code in figure 2. Variables Y and F are initialized to `free` and `out`, respectively. They are assumed to occupy adjacent half-words in memory, where they can be read or written either separately or together, atomically. The delay in line 7 is assumed to be long enough for any process that has already read $Y =$ `free` in line 3 to complete line 5, and any process that has already set Y in line 5 to complete line 6 and (if not delayed) line 10.

A similar algorithm, due to Alur and Taubenfeld [1], appears in figure 3. Rather than read and write at multiple granularities, this algorithm relies on an additional flag variable (Z) to determine whether any process has entered the critical section by the end of the delay. When releasing the lock, process i first clears Z, and then clears Y *only if* Y still equals i. If Y has changed, the last process to change it is permitted to enter the critical section as soon as Z is cleared. Both our algorithm and Alur and Taubenfeld's assume a bound on relative rates of process execution, with identical delays on the slow code path, when contention is detected. Both algorithms require only $O(n)$ time when arbitrating among n concurrent lock acquisitions, and $O(1)$ time in the absence of contention. On the fast code path, however, our algorithm performs 25% fewer shared memory references. As shown in section 4, this translates not only into lower overhead in the no-contention case, but also, given backoff, in most cases of contention as well.

[1] As noted in section 1, we assume that a reference to shared memory may take time linear in the number of processes attempting to access the same location concurrently.

```
1:  START:
2:      X ← i
3:      if Y ≠ free
4:          goto START
5:      Y ← i
6:      if X ≠ i
7:          { delay }
8:          if (Y, F) ≠ (i, out)
9:              goto START
10:     F ← in
11: { critical section }
12:     (Y, F) ← (free, out)
13: { non-critical section }
14:     goto START
```

Figure 2: A new fast mutual exclusion algorithm.

```
1:  START:
2:      X ← i
3:      repeat until Y = free
4:      Y ← i
5:      if X ≠ i
6:          { delay }
7:          if Y ≠ i
8:              goto START
9:          repeat until Z = 0
10:     else
11:         Z ← 1
12: { critical section }
13:     Z ← 0
14:     if Y = i
15:         Y ← free
16: { non-critical section }
17:     goto START
```

Figure 3: Alur and Taubenfeld's algorithm.

3 Correctness

In this section we present proofs of mutual exclusion and livelock freedom for our algorithm.

As it concerns the algorithm, each process i can be conceptualized as a sequence of non-looping subprocesses. Thus, the execution time of a subprocess is bounded except for the critical section. A subprocess either acquires the lock, executes the critical section, releases the lock, and terminates; or fails to acquire the lock at some point, terminates, and the next subprocess begins execution from START. It is clear that at any moment each process has at most one subprocess running.

Let U be the set of (non-looping) subprocesses running at time t. U can be partitioned into five disjoint sets A, B, C, D, and E, defined in terms of the truth of the four conditions in lines 3, 6, and 8 in the algorithm, where the condition in line 8 can be considered as two sequential conditions, the first testing Y and if it is equal to i, the second testing F. The sets are defined as follows:

$$A = \left\{ i \,\middle|\, \begin{array}{c} Y = \texttt{free} \\ X = i \end{array} \right\},\ B = \left\{ i \,\middle|\, \begin{array}{c} Y = \texttt{free} \\ X \neq i \\ Y = i \\ F = \texttt{out} \end{array} \right\},\ C = \left\{ i \,\middle|\, \begin{array}{c} Y = \texttt{free} \\ X \neq i \\ Y \neq i \end{array} \right\},\ D = \left\{ i \,\middle|\, \begin{array}{c} Y = \texttt{free} \\ X \neq i \\ Y = i \\ F \neq \texttt{out} \end{array} \right\},\ \text{and}$$

$$E = \left\{ i \,\middle|\, Y \neq \texttt{free} \right\}.$$

In the proof we use the following notation: $\forall i, j \in U$, il denotes the time at which subprocess i executes line l in the algorithm, and $il < jm$ denotes that i executes line l before j executes line m.

3.1 Mutual Exclusion

Let W be the set of subprocesses executing their critical sections at time t. $W = \{i | i \in A \cup B$ and $i10 \leq t \leq i12\}$. To prove mutual exclusion it suffices to prove that $\forall t$, $|W| \leq 1$.

Lemma 1: $\forall i \in A \cup B$, $\nexists j \in A \cup B$ such that $j5 < i12 < j12$.

By defining the "order" of a subprocess to be the number of subprocesses that set Y to `free` before it does, Lemma 1 can be proved by induction on the order of subprocesses in $A \cup B$. A complete proof is presented in the appendix.

Now we can define supersets for A and B by transforming the conditions on the values of state variables to conditions on the order of setting and reading them by the subprocess under consideration and other concurrent subprocesses.

$\forall i \in A$, and $\forall j \in U - \{i\}$, for Y to be equal to `free` at $i3$, either $i3 < j5$ or $j12 < i3$ (Lemma 1). And for X to be equal to i at $i6$, either $j2 < i2$ or $i6 < j2$.

$$\text{Therefore, } A \subseteq \left\{ i \,\middle|\, \begin{array}{c} i \in U \text{ and } \forall j \in U - \{i\}, \\ j2 < i2 \text{ and } i3 < j5 \text{ or} \\ i6 < j2 \text{ or} \\ j2 < i2 \text{ and } j12 < i3 \end{array} \right\}.$$

$\forall i \in B$ and $\forall j \in U - \{i\}$, for Y to be equal to `free` at $i3$, either $i3 < j5$ or $j12 < i3$ (Lemma 1). For Y to be equal to i at $i8$, either $i8 < j5$, $j5 < i5$ and $i8 < j12$, or $j12 < i5$. And for F to be equal to `out` at $i8$, either $i8 < j10$ or $j12 < i8$.

$$\text{Therefore, } B \subseteq \left\{ i \,\middle|\, \begin{array}{c} i \in U \text{ and } \forall j \in U - \{i\}, \\ i8 < j5 \text{ or} \\ i3 < j5 < i5 \text{ and } i8 < j10 \text{ or} \\ i3 < j5 \text{ and } j12 < i5 \text{ or} \\ j12 < i3 \end{array} \right\}.$$

$$\text{Let } AA = \{(i.j) | i, j \in A \text{ and } i \neq j\} \text{ then } AA \subseteq \left\{ (i,j) \,\middle|\, \begin{array}{c} i, j \in U \text{ such that} \\ j6 < i2 \text{ and } j12 < i3 \text{ or} \\ i6 < j2 \text{ and } i12 < j3 \end{array} \right\}.$$

Let $BB = \{(i,j) | i,j \in B \text{ and } i \neq j\}$ then $BB \subseteq \left\{ (i,j) \left| \begin{array}{c} i,j \in U \text{ such that} \\ j3 < i5 \text{ and } i8 < j5 \text{ and } j8 < i10 \text{ or} \\ j3 < i5 \text{ and } i12 < j5 \text{ or} \\ i12 < j3 \text{ or} \\ i3 < j5 \text{ and } j8 < i5 \text{ and } i8 < j10 \text{ or} \\ i3 < j5 \text{ and } j12 < i5 \text{ or} \\ j12 < i3 \end{array} \right. \right\}$.

Let $AB = \{(i,j) | i \in A \text{ and } j \in B\}$, then $AB \subseteq \left\{ (i,j) \left| \begin{array}{c} i,j \in U \text{ such that} \\ j2 < i2 \text{ and } i3 < j5 \text{ and } j8 < i5 \text{ or} \\ j2 < i2 \text{ and } j3 < i5 < j5 \text{ and } j8 < i10 \text{ or} \\ j2 < i2 \text{ and } j3 < i5 \text{ and } i12 < j5 \text{ or} \\ i6 < j2 \text{ and } i12 < j3 \text{ or} \\ j2 < i2 \text{ and } j12 < i3 \end{array} \right. \right\}$.

Let $BA = \{(i,j) | i \in B \text{ and } j \in A\}$ then $BA = \{(i,j) | (j,i) \in AB\}$.

With sufficient delay,

$BB \subseteq \left\{ (i,j) \left| \begin{array}{c} i,j \in U \text{ such that} \\ j3 < i5 \text{ and } i12 < j5 \text{ or} \\ i12 < j3 \text{ or} \\ i3 < j5 \text{ and } j12 < i5 \text{ or} \\ j12 < i3 \end{array} \right. \right\}$ and $AB \subseteq \left\{ (i,j) \left| \begin{array}{c} i,j \in U \text{ such that} \\ j2 < i2 \text{ and } j3 < i5 \text{ and } i12 < j5 \text{ or} \\ i6 < j2 \text{ and } i12 < j3 \text{ or} \\ j2 < i2 \text{ and } j12 < i3 \end{array} \right. \right\}$.

Let $W_2 = \{(i,j) | i,j \in W \text{ and } i \neq j\}$, then $W_2 \subseteq AA \cup BB \cup AB \cup BA$. Then, $W_2 = \emptyset$. Then, $|W| \leq 1$. This completes the proof of mutual exclusion □

3.2 Livelock Freedom

Lemma 2: $\forall i \in U$, **if** i **sets** Y **to** i **and terminates at time** t **while** $Y = i$, **then** $\exists j \in A \cup B$ **such that** $j10 < t < j12$

Proof:

If $i \in A \cup B$ then i sets Y to `free` and terminates. If $i \in C$ then i terminates while $Y \neq i$. If $i \in E$ then i never sets Y to i. If $i \in D$ then at $i8$, $F =$ `in` then it must be the case that $\exists j \in A \cup B$ such that $j10 < i8 < j12$. If $j12 < i8$ then i terminates while $Y \neq i$, otherwise i terminates after $j10$ and before $j12$ □

Lemma 3: $\forall i \in C$, $A \cup B \cup D \neq \emptyset$, **for some** t, $i5 \leq t \leq i8$

Proof:

Assume that the lemma is false i.e. $\exists i \in C$, $\forall t$, $i5 \leq t \leq i8$, $A \cup B \cup D = \emptyset$. Since $i \in C$ then at $i8$, $Y \neq i$. Hence either $Y =$ `free` or $Y = j \neq i$, where $j \in U$. If $Y =$ `free` then $\exists k \in A \cup B$ such that $i5 < k12 < i8$, which contradicts the initial assumption. Thus $Y = j$ i.e. $\exists j \in A \cup B \cup C \cup D$ such that j is the last process to set Y between $i5$ and $i8$. Then according to the initial assumption, $j \in C$. Therefore $\exists k \in A \cup B \cup C \cup D$ that sets Y between $j5$ and $j8$. This again implies that $k \in C$. Since j is the last subprocess to set Y before $i8$, $i8 < k5$. If $i8 < k3$ then $\exists l \in A \cup B$ such that $i8 < l12 < k3$ i.e. $j5 < l12 < j8$. Contradiction. Therefore, $k3 < i8$. If $i5 < k3$ then $\exists l \in A \cup B$ such that $i5 < l12 < k3$ i.e. $i5 < l12 < i8$. Contradiction. Therefore $k3 < i5$. Thus $k3 < i5$ and $i8 < k5$ which is not possible with sufficient delay. Therefore, the initial assumption is false and the lemma is true □

Define t_1 and t_2 such that $\forall i \in U$ i starts after t_1 and terminates before t_2. Assuming that the critical sections are finite, (t_1, t_2) can be chosen to be finite. Let U' be the set of subprocesses running in the interval (t_1, t_2), and similarly define A', B', C', D', and E'. It is clear that $U \subseteq U'$, $A \subseteq A'$, $B \subseteq B'$, $C \subseteq C'$, $D \subseteq D'$, and $E \subseteq E'$.

The algorithm is livelock free if $U \neq \emptyset$ implies that $A' \cup B' \neq \emptyset$. Assuming that $U \neq \emptyset$, $\exists i \in A \cup B \cup C \cup D \cup E$.

If $i \in E$ then at $i3$, $Y \neq$ `free`. Hence $\exists j$, $Y = j$ and $j \in A' \cup B' \cup C' \cup D'$ which is the last to set Y before $i3$. j is either running at $i3$ or has already terminated. If j is running then $A \cup B \cup C \cup D \neq \emptyset$. If j has already terminated then according to Lemma 2, $\exists k \in A \cup B$ that was executing its critical section while j terminated. It cannot be the case that $k12 < i3$ because j is the last to set Y before $i3$. Therefore it must be the case that $k10 < i3 < k12$ then $A \cup B \neq \emptyset$. Therefore $E \neq \emptyset$ implies that $A' \cup B' \cup C' \cup D' \neq \emptyset$. If $i \in C$ then according to Lemma 3, $A' \cup B' \cup D' \neq \emptyset$. Finally, if $i \in D$ then at $i8$, $F \neq$ `out`. Hence $\exists j \in A \cup B$ such that $j10 < i8 < j12$. Therefore, $A' \cup B' \neq \emptyset$.

Therefore, $U \neq \emptyset$ implies that $A' \cup B' \neq \emptyset$. This completes the proof of livelock freedom □

4 Experiments

In this section we present the experimental results of implementing three mutual exclusion algorithms—Lamport's second, Alur and Taubenfeld's, and ours—on an 8-processor Silicon Graphics (SGI) Iris 4D/480 multiprocessor and on a larger simulated machine. Based on relative numbers of shared-memory reads and writes (see table 1), we expected these algorithms to dominate the others. Among them, we expected the new algorithm to perform the best, both with and without contention. We also expected exponential backoff to substantially improve the performance of all three algorithms.

It was not clear to us *a priori* whether Alur and Taubenfeld's algorithm would perform better or worse than Lamport's algorithm. The former performs more shared memory references on its fast code path, but has a lower asymptotic complexity on its slow code path. How often each path would execute seemed likely to depend on the effectiveness of backoff. For similar reasons, it was unclear how large the performance differences among the algorithms would be. Our experiments therefore serve to verify expected relative orderings, determine unknown orderings, and quantify differences in performance.

4.1 Real Performance on a Small Machine

To obtain a bound on relative rates of processor execution, we exploited the real-time features of SGI's IRIX operating system, dedicating one processor to system activity, and running our test on the remaining seven processors, with interrupts disabled. The system processor itself was lightly loaded, leaving the bus essentially free. We disabled caching for the shared variables used by the lock algorithms, but enabled it for private variables and code. We compiled all three locks with the MIPS compiler's highest (−O3) level of optimization.

We tested two versions of each algorithm: one with limited exponential backoff and one without. The no-backoff version of Lamport's algorithm matches the pseudo-code on the right side of figure 1. The no-backoff version of the new algorithm matches the pseudo-code in figure 2, except that after discovering that $Y \neq$ `free` in line 3, or that $(Y, F) \neq$ (i, `out`) in line 8, we wait for $Y =$ `free` before returning to START. Similarly, the no-backoff version of Alur and Taubenfeld's lock matches the pseudo-code in figure 3, except that (1) if $Y \neq$ `free` in line 3, we return to START after Y becomes `free`, rather than continuing, and (2) if $Y \neq i$ at line 7, we wait for $Y =$ `free` before returning to START. In the backoff versions of all three algorithms, each repeat loop includes a delay that increases geometrically in consecutive iterations, subject to a cap. The base, multiplier, and cap were chosen by trial and error to maximize performance. C code for our experiments can be obtained via anonymous ftp from cayuga.cs.rochester.edu (directory pub/scalable_sync/fast).

Performance results appear in figures 4 and 5. In both graphs, point (x, y) indicates the number of microseconds required for one processor to acquire and release the lock, when x processors are attempting to do so simultaneously. These numbers are derived from program runs in which each processor executes 100,000 critical sections. Within the critical section, each processor increments a shared variable. After releasing the lock, the processor executes only loop overhead before attempting to acquire the lock again. Program runs were repeated several times; reported results are stable to about ±2 in the third significant digit. The one-processor points indicate the time to acquire and release the lock (plus loop overhead) in the absence of contention. Points for two or more processors indicate the time for one processor to pass the lock on to the next.

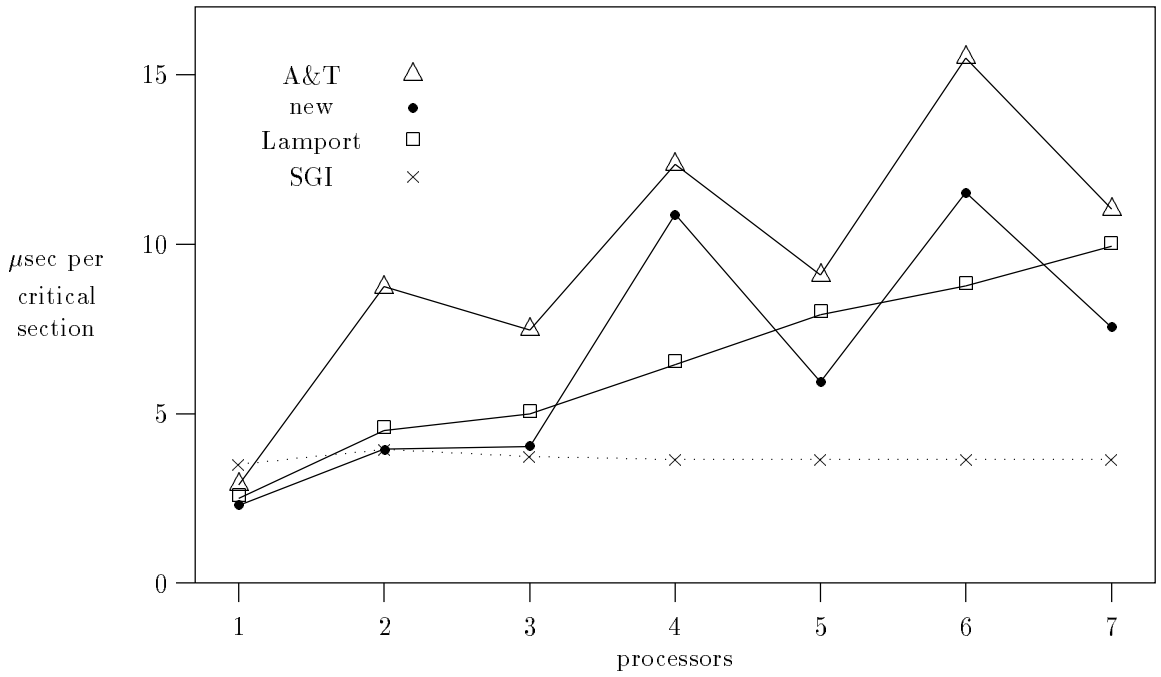

Figure 4: Performance results without backoff on the SGI Iris.

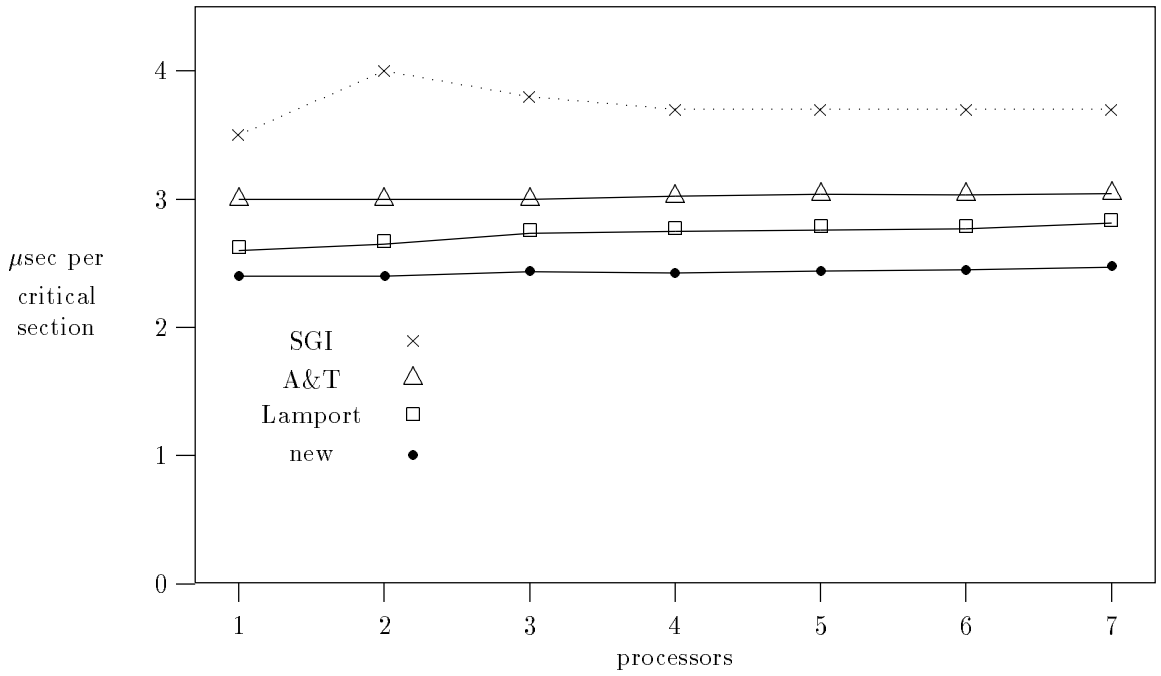

Figure 5: Performance results with backoff on the SGI Iris.

	Processors						
Algorithm	1	2	3	4	5	6	7
Alur & Taubenfeld	100	92.9	94.2	29.5	90.0	29.3	85.0
new	100	51.6	65.8	22.8	61.2	15.1	53.9

Table 2: Percentage of critical section entries in the no-backoff experiments made via the fast (no delay) code path of the delay-based algorithms.

Backoff is clearly important. Without it, performance degrades rapidly with increasing contention. Lamport's algorithm degrades smoothly, while the new algorithm and that of Alur and Taubenfeld behave erratically (see below). By contrast, with backoff, performance of all three algorithms is excellent, and roughly proportional to the number of shared memory references on the fast code path. With only six such references, the new algorithm is the fastest.

We instrumented the two delay-based algorithms in an attempt to explain the strange (but highly repeatable) behavior of the new algorithm and that of Alur and Taubenfeld in the no-backoff experiments. The results appear in table 2. We hypothesize that with odd numbers of processors there is usually one that is able to enter its critical section without executing a delay, while with even numbers of processors the test falls into a mode in which all processors are frequently delayed simultaneously, with none in the critical section. This hypothesis is consistent with memory reference traces recorded for similarly anomalous points in the simulation experiments, as discussed in the following section.

Surprisingly, all three algorithms with backoff outperform the native `test_and_set`locks supported in hardware on the SGI machine. These native locks employ a separate synchronization bus, and are generally considered very fast. With backoff, processes execute the fast path in almost every lock acquisition. This explains the observation that relative performance of the three locks is proportional to the number of shared memory references in the fast paths in their acquisition/release protocols.

4.2 Simulated Performance on a Large Machine

To investigate the effect of backoff on fast mutual exclusion algorithms with only atomic read and write, and to evaluate the relative performance of the three algorithms when there is a higher level of contention on a large number of processors, we simulated the execution of these three lock algorithms on a hypothetical large machine with 128 processors.

Our simulations use the same executable program employed on the SGI machine. It runs this program under Veenstra's MIPS interpreter, Mint [9], with a simple back end that determines the latency of each reference to shared memory. We assume that shared memory is uncached, that each memory request spends 36 cycles in each direction traversing some sort of processor/memory interconnect, that competing requests queue up at the memory, and that the memory can retire one request every 10 cycles. The minimum time for a shared-memory reference is therefore 82 cycles. For the delay-based algorithms, we used a delay of 2500 cycles, which provides enough time for the memory to service 2 requests from each of 128 processors.

Figures 6 and 7 show that the performance of all three algorithms (and Lamport's in particular) improves substantially with the use of exponential backoff. Thus backoff makes mutual exclusion feasible even for large numbers of processors, with no atomic instructions other than read and write.

For figure 7, backoff constants (base, multiplier, and cap) were selected for each algorithm to maximize its performance on 128 processors. On smaller numbers of processors this backoff is too high, and performance is unstable. With greater than 32 processors, the relative order of the algorithms remains the same over a wide range of possible backoff constants. Most of the individual data points reflect simulation runs in which each processor executes 100 critical sections. We ran longer simulations on a subset of the points in order to verify that the total number of elapsed cycles was linearly proportional to the number of critical section executions.

All three algorithms were found to be sensitive not only to the choice of backoff constants, but also to critical and non-critical section lengths. With many variations of these parameters, the overall relative performance of the three algorithms was always found to be the same. The presented results are with a single shared-memory update in each critical section, and nothing but loop overhead in the non-critical sections.

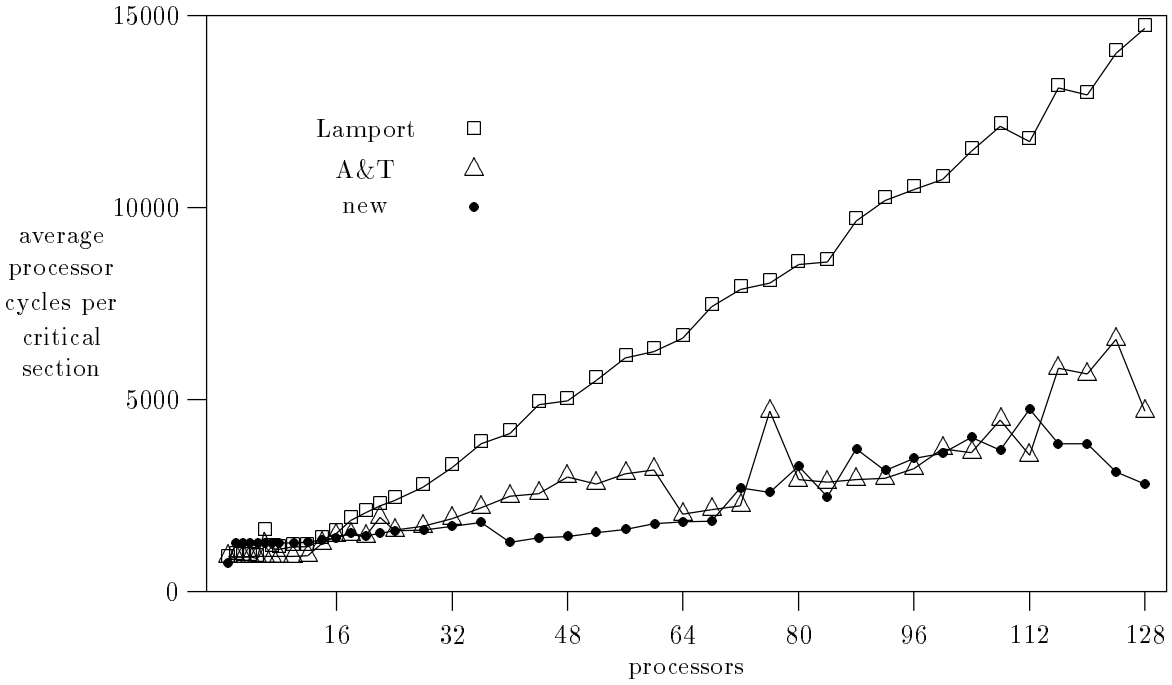

Figure 6: Performance results without backoff on a (simulated) 128-node machine.

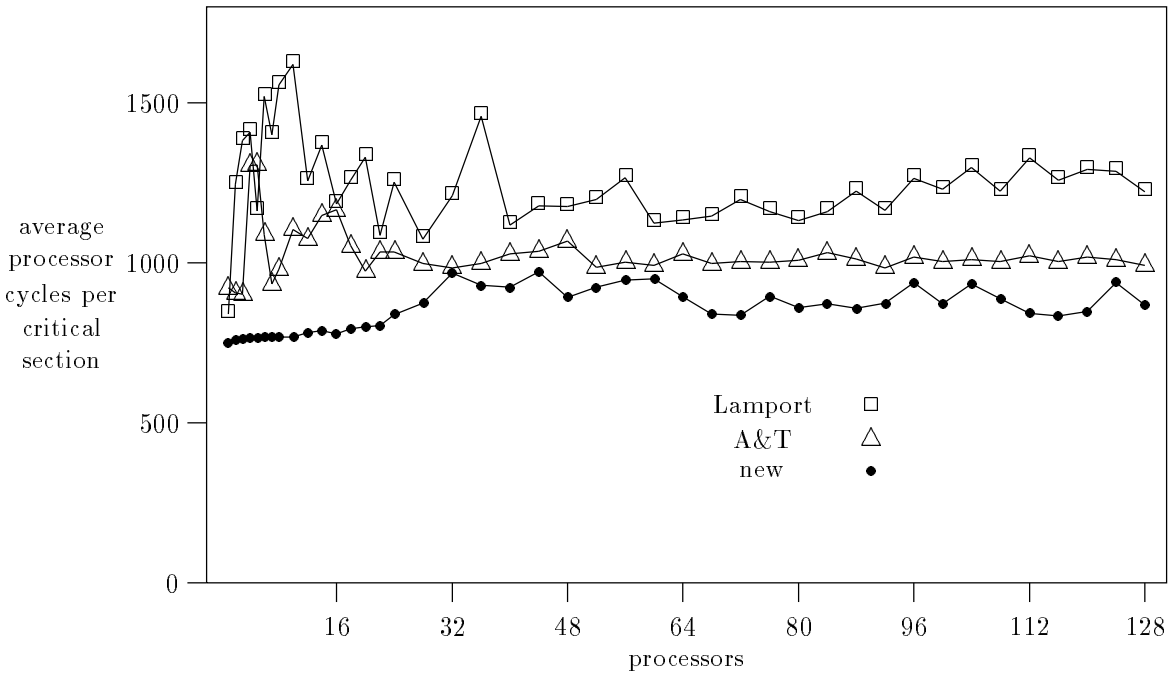

Figure 7: Performance results with backoff on a (simulated) 128-node machine.

Unstable parts of the graphs in figures 6 and 7 were investigated using detailed traces of shared memory references. The apparently anomalous points can be attributed to the big difference in execution time between the fast and the slow paths of the algorithms. With many variations of the backoff constants and length of critical and non-critical sections, there are always points (numbers of processors) where most of the time a processor executes the slow path to acquire and release the lock. But these points were found to change with different combinations of parameters.

The simulation results verify that the new algorithm outperforms the others with its low number of shared memory references in the fast path. For large numbers of processors, Alur and Taubenfeld's algorithm always outperforms Lamport's algorithm despite its higher number of shared memory references in the fast path, due to the increasing cost of the slow path of Lamport's algorithm.

5 Conclusions

Fast mutual exclusion with only reads and writes is a topic of considerable theoretical interest, and of some practical interest as well. We have presented a new fast mutual exclusion algorithm that has an asymptotic time complexity of $O(n)$ in the presence of contention, while requiring only 2 reads and 4 writes in the absence of contention. The algorithm capitalizes on the ability of most memory systems to read and write atomically at both full- and half-word granularities. The same asymptotic result has been obtained independently by Alur and Taubenfeld, without the need for multi-grain memory operations, but with a higher constant overhead: 3 reads and 5 writes on the fast code path.

From a practical point of view, our results confirm that mutual exclusion with only reads and writes is a viable, if not ideal, means of synchronization. Its most obvious potential problem—contention—can be mitigated to a large extent by the use of exponential backoff.

Most modern microprocessors intended for use in multiprocessors provide atomic instructions designed for synchronization (`test_and_set`, `swap`, `compare_and_swap`, `fetch_and_add`, `load_linked/store_conditional`, etc). For those that do not, system designers are left with the choice between implementing hardware synchronization outside the processor (as in the synchronization bus of Silicon Graphics machines), or employing an algorithm of the sort discussed in this paper. Backoff makes the latter option attractive.

On the SGI Iris, our new algorithm outperforms the native hardware locks by more than 30%. For arbitrary user-level programs, which cannot assume predictable execution rates, Lamport's second algorithm (with backoff) outperforms the native locks by 25%. These results are reminiscent of recent studies by Yang and Anderson, who found that their hierarchical read- and write-based mutual exclusion algorithm (line 4 in table 1) provided performance competitive with that of `fetch_and_`Φ-based algorithms on the BBN TC2000 [10]. Both Lamport's second algorithm and Yang and Anderson's algorithms require space per lock linear in the number of contending processes. For systems with very large numbers of processes, Merritt and Taubenfeld have proposed a technique that allows a process to register, on the fly, as a contender for only the locks that it will actually be using [6].

For the designers of microprocessors and multiprocessors, we remain convinced that the most cost-effective synchronization mechanisms are algorithms that use simple `fetch_and_`Φ instructions to establish links between processes that then spin on local locations [5]. For machines without appropriate instructions, however, fast mutual exclusion remains a viable option.

References

[1] R. Alur and G. Taubenfeld. Results about Fast Mutual Exclusion. Technical report, AT&T Bell Laboratories, 5 January 1993. Revised version of a paper presented at the *Thirteenth IEEE Real-Time Systems Symposium*, December 1992.

[2] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6-16, January 1990.

[3] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43-112. Academic Press, London, 1968.

[4] L. Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, February 1987.

[5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21-65, February 1991.

[6] M. Merritt and G. Taubenfeld. Speeding Lamport's Fast Mutual Exclusion Algorithm. *Information Processing Letters*, 45(3):137-142, March 1993.

[7] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115-116, June 1981.

[8] E. Styer. Improving Fast Mutual Exclusion. In *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 159-168, Vancouver, BC, Canada, 9-12 August 1992.

[9] J. E. Veenstra. Mint Tutorial and User Manual. TR 452, Computer Science Department, University of Rochester, May 1993.

[10] H. Yang and J. H. Anderson. Fast, Scalable Synchronization with Minimal Hardware Support (extended abstract). In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, (to appear) 15-18 August 1993.

A Proof of Lemma 1

Lemma 1: $\forall i \in A \cup B$, $\nexists j \in A \cup B$ such that $j5 < i12 < j12$.

Proof:

Let the order of a subprocess in $A \cup B$ be the number of subprocesses that set Y to `free` before it does. A proof by induction on the order of subprocesses involves proving that: (1) The lemma is true for the subprocess of order 0, and (2) If the lemma is true for subprocesses of order less than n, then it is true for the subprocess of order n.

Basis:

- Let i be the subprocess of order 0, and assume that $\exists j \in A \cup B$ such that $j5 < i12 < j12$.
- Assume that $i5 < j3$. Since $j5 < i12$ then $j3 < i12$. Then $i5 < j3 < i12$. Then for Y to be equal to `free` at $j3$ it must be the case that $\exists k \in A \cup B$ that sets Y to `free` before $j3$ i.e. before $i12$. Then i is of order greater than 0. Contradiction. Therefore, it must be the case that $j3 < i5$.
- Assume that $j5 < i3$. Since $i12 < j12$ then $i3 < j12$. Then $j5 < i3 < j12$. Then for Y to be equal to `free` at $i3$ it must be the case that $\exists k \in A \cup B$ that sets Y to `free` before $i3$ i.e. before $i12$. Then i is of order greater than 0. Contradiction. Therefore, it must be the case that $i3 < j5$.
- Considering the four possible cases for i and j belonging to A or B:
  1. i and $j \in A$: Since $i3 < j5$ then $i2 < j6$ then for X to be equal to j at $j6$ it must be the case that $i2 < j2$. Since $j3 < i5$ then $j2 < i6$. Then $i2 < j2 < i6$ then at $i6$ $X \neq i$. Then $i \notin A$. Contradiction.
  2. $i \in A$ and $j \in B$:
     - Assume that $j5 < i5$. For Y to be equal to j at $j8$ it must be the case that $j8 < i5$. Then $i3 < j5$ and $j8 < i5$, which is not possible with sufficient delay. Therefore, it must be the case that $i5 < j5$.
     - Since $i5 < j5$ then with sufficient delay it must be the case that $i10 < j8$. For F to be equal to `out` at $j8$ it must be the case that $\exists k \in A \cup B$ such that $i10 < k12 < j8$ (it is possible that $k = i$). If $k = i$ then for Y to be equal to j at $j8$, $i12 < j5$ which contradicts the initial assumption. If $k \neq i$ then $k12 < j5 < i12$, then i is not of order 0. Contradiction.
  3. $i \in B$ and $j \in A$:
     - Assume that $i5 < j5$. For Y to be equal to i at $i8$ it must be the case that $i8 < j5$. Then $j3 < i5$ and $i8 < j5$, which is not possible with sufficient delay. Therefore, it must be the case that $j5 < i5$.
     - Since $j5 < i5$ then with sufficient delay it must be the case that $j10 < i8$. For F to be equal to `out` at $i8$ it must be the case that $\exists k \in A \cup B$ such that $j10 < k12 < i8$ (it is possible that $k = j$). Then i is not of order 0. Contradiction.
  4. i and $j \in B$: For Y to be equal to i and j at $i8$ and $j8$ respectively, it must be either the case that $i8 < j5$ or $j8 < i5$. Then it must be either the case that $i3 < j5$ and $j8 < i5$; or $j3 < i5$ and $i8 < j5$. Both cases are not possible with sufficient delay.
- Therefore, $\nexists j \in A \cup B$ such that $j5 < i12 < j12$, i.e. the lemma is true for the subprocess of order 0.

Induction

- Assume that the lemma is true for all subprocesses of order less than n. Let $i \in A \cup B$ be the subprocess of order n.
- Assume that $\exists j \in A \cup B$ such that $j5 < i12 < j12$.

- Assume that $i5 < j3$. Since $j5 < i12$ then $j3 < i12$. Then $i5 < j3 < i12$. Then for Y to be equal to `free` at $j3$ it must be either the case that $\exists k \in A \cup B$ such that $i5 < k12 < j3$ i.e. $i5 < k12 < i12$. This contradicts the inductive hypothesis. Therefore, it must be the case that $j3 < i5$.

- Assume that $j5 < i3$. Since $i12 < j12$ then $i3 < j12$. Then $j5 < i3 < j12$. Then for Y to be equal to `free` at $i3$ it must be the case that $\exists k \in A \cup B$ such that $j5 < k12 < i3$ i.e. $j5 < k12 < j15$ and $k12 < i12$. This contradicts the inductive hypothesis. Therefore, it must be the case that $i3 < j5$.

- Considering the four possible cases for i and j belonging to A or B:

  1. i and $j \in A$: Since $i3 < j5$ then $i2 < j6$ then for X to be equal to j at $j6$ it must be the case that $i2 < j2$. Since $j3 < i5$ then $j2 < i6$. Then $i2 < j2 < i6$ then at $i6$ $X \neq i$. Then $i \notin A$. Contradiction.
  2. $i \in A$ and $j \in B$:
     - Assume that $j5 < i5$. For Y to be equal to j at $j8$ it must be the case that $j8 < i5$. Then $i3 < j5$ and $j8 < i5$, which is not possible with sufficient delay. Therefore, it must be the case that $i5 < j5$.
     - Since $i5 < j5$ then with sufficient delay it must be the case that $i10 < j8$. Then $j3 < i10 < j8$. For F to be equal to `out` at $j8$ it must be the case that $\exists k \in A \cup B$ such that $i10 < k12 < j8$ (it is possible that $k = i$). If $k = i$ then for Y to be equal to j at $j8$, $i12 < j5$ which contradicts the initial assumption. If $k \neq i$ then $k12 < j5 < i12$, then $i10 < k12 < i12$, which contradicts the inductive hypothesis.
  3. $i \in B$ and $j \in A$:
     - Assume that $i5 < j5$. For Y to be equal to i at $i8$ it must be the case that $i8 < j5$. Then $j3 < i5$ and $i8 < j5$, which is not possible with sufficient delay. Therefore, it must be the case that $j5 < i5$.
     - Since $j5 < i5$ then with sufficient delay it must be the case that $j10 < i8$. Then $i3 < j10 < i8$. For F to be equal to `out` at $i8$ it must be the case that $\exists k \in A \cup B$ such that $j10 < k12 < i8$ (it is possible that $k = j$). If $k = j$ then for Y to be equal to i at $i8$, $j12 < i5$ which contradicts the initial assumption. If $k \neq j$ then $k12 < i5 < j12$, then $j10 < k12 < j12$, which contradicts the inductive hypothesis.
  4. i and $j \in B$: For Y to be equal to i and j at $i8$ and $j8$ respectively, it must be either the case that $i8 < j5$ or $j8 < i5$. Then it must be either the case that $i3 < j5$ and $j8 < i5$; or $j3 < i5$ and $i8 < j5$. Both cases are not possible with sufficient delay.

- Therefore, $\nexists j \in A \cup B$ such that $j5 < i12 < j12$ $\square$