1991

Mutual exclusion for uniprocessors

Brian N. Bershad
Carnegie Mellon University

Published In
.

Mutual Exclusion for Uniprocessors

Brian N. Bershad
April 2, 1991
CMU-CS-91-1163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We discuss and evaluate four mutual exclusion primitives appropriate for uniprocessors: memory interlocked instructions, software reservation, kernel emulation and *restartable atomic sequences*. A restartable atomic sequence is a code fragment that, if interrupted, is resumed by software at the beginning of the sequence, guaranteeing that the sequence is eventually executed to completion atomically.

We describe two implementations of restartable atomic sequences for the Mach operating system, and show that restartable atomic sequences perform significantly better than either kernel emulation or software reservation, making them an attractive alternative for use on uniprocessors that do not support atomic read-modify-write instructions. Further, on many processor architectures that do support such instructions, we show that restartable atomic sequences can have better performance. We show that improving the performance of low-level mutual exclusion mechanisms can have a substantial effect on application performance.

This research was sponsored in part by The Defense Advanced Research Projects Agency, Information Science and Technology Office, under the title "Research on Parallel Computing", ARPA Order No. 7330, issued by DARPA/CMO under Contract MDA972-90-C-0035 and in part by the Open Software Foundation (OSF).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF or the U.S. government.

Keywords: Operating Systems, Mutual Exclusion, Performance

1 Introduction

Multithreaded programs use mutual exclusion to guarantee consistency of shared data structures. Even on a uniprocessor, mutual exclusion is necessary to protect shared data against an interleaved thread schedule. Interleaving can occur when a thread is preempted, when an interrupt or exception occurs, or when a thread voluntarily relinquishes the processor. Programmers tend to think of mutual exclusion in terms of high-level primitives such as *P, V* [Dijkstra 68a] and *acquire_mutex, release_mutex* [Birrell 89]. These primitives, however, must be implemented with low-level operations that grant one of several threads mutually exclusive access to some data structure.

This paper discusses several mechanisms that may be used for mutual exclusion in multithreaded programs on uniprocessors. We describe three mechanisms that are explicitly insulated against interrupts during atomic operations, and a fourth mechanism, called a *restartable atomic sequence.* The use of a restartable atomic sequence assumes that interrupts during atomic operations occur infrequently, and that a simple recovery protocol can be used when untimely interrupts do occur. Using programs running on the Mach operating system [Accetta et al. 86], we show that restartable atomic sequences are significantly more efficient than other software techniques. We have measured performance improvements of up to 50% for some applications on the MIPS R3000-based [Kane 87] DECstation 5000, which does not have hardware support for atomic operations. In addition, we show that restartable atomic sequences can even outperform hardware mechanisms on processors which do provide explicit support for atomic operations.

1.1 Motivation

Efficient mutual exclusion mechanisms are becoming increasingly important on uniprocessors for two reasons. First, modern applications now use multiple threads as a program structuring device, and as a way to manage I/O and server parallelism even when no true CPU parallelism is available. Second, many operating systems today are built using the "small-kernel" model in which the kernel supports a few services such as thread scheduling, virtual memory and interprocess communication [Mullender et al. 90, Cheriton 88, Rozier et al. 88, Accetta et al. 86, Thacker et al. 88]. Other services such as the file system and networking are implemented as multithreaded user-level applications. The small-kernel approach exposes the performance of a system's mutual exclusion primitives because even single threaded programs rely on basic operating system services that are implemented out of the kernel using multiple threads. The performance of all applications is therefore ultimately influenced by the performance of the underlying mutual exclusion facilities.

The mechanisms that are generally used to implement atomic operations on a uniprocessor (i.e., those described in every undergraduate operating systems textbook) can be characterized as *pessimistic.* That is, they are designed with the assumption that atomicity can be violated at any time (e.g., with an interrupt), and therefore guard against this potential violation. In so doing, pessimistic mechanisms have a high overhead, and can have an adverse effect on the performance of multithreaded programs.

In this paper, we introduce an *optimistic* approach for handling atomic operations on a uniprocessor. We assume that atomicity is rarely violated and use a fast solution for the common case of uninterrupted execution during a code sequence that must be atomic. To ensure correctness, though, in the cases when an interrupt does occur, we rely on a separate recovery mechanism to take corrective action. As we show, the assumption that atomic operations on a uniprocessor are almost never interrupted is a good one, since it is true, and since it allows us to implement atomic operations with greater efficiency than is possible with pessimistic approaches.

In the next section we review several ways to provide atomic operations on a uniprocessor and

describe an optimistic approach based on restartable atomic sequences. In Section 3 we discuss two implementations of restartable atomic sequences for the Mach operating system. In Section 4 we examine the effect that restartable atomic sequences have on the performance of multithreaded programs and the operating system. In Section 5 we show that restartable atomic sequences can outperform atomic hardware operations on several processor architectures. We discuss related work in Section 6. Finally, in Section 7 we present our conclusions.

2 Implementing Atomic Operations on a Uniprocessor

This section describes four techniques for implementing atomic operations on a uniprocessor. Three of the techniques, memory interlocked instructions, software reservation and kernel emulation, are pessimistic. The fourth, restartable atomic sequences, is based on the optimistic approach. Throughout this paper, we focus on an atomic *Test-And-Set* operation, although other atomic operations such as *Compare-And-Swap* or *Fetch-And-Add* could be constructed using the techniques we describe.

2.1 Memory Interlocked Read-Modify-Write Instructions

Memory interlocked instructions (or instruction sequences) require special hardware support from the processor and bus to ensure that a given memory location can be read, modified and written without interruption. Memory interlocked instructions are primarily intended to support multiprocessing, but can be used on uniprocessor systems as well.

Unfortunately, not all processors support memory interlocked instructions, and many that do, do so reluctantly; i.e., the cycle time for an interlocked access is several times greater than that for a non-interlocked access. The reasons for the higher cost are increased complexity [Intel860 89], an overly "rich" set of atomic operations [Leonard 87, Intel386 90], support for non-aligned atomic updates [Leonard 87], and the fact that atomic operations can bypass the on-chip cache [Motorola 88100 88]. A good survey of atomic hardware operations, characteristics and implementations can be found in [Glew & Hwu 91].

2.2 Software reservation

Instead of using hardware directly, atomic operations can be constructed with software reservation algorithms, such as Dekker's [Dijkstra 68b], Peterson's [Peterson 81] or Lamport's [Lamport 87]. Roughly speaking, software reservation algorithms work by having a thread first register its intent to perform an atomic operation, then check if any other thread has registered a similar intent, and if not to then complete the operation.

Although a large number of reservation-based mutual exclusion algorithms are described in the literature, we use Lamport's "fast mutual exclusion algorithm" [Lamport 87] to investigate software reservation schemes, since it has been proven correct and shown to be optimal.[1] In Lamport's algorithm, shown in Figure 1, each thread has a unique identifier, i, which is used to place reservations into the variable x, and to indicate ownership of the lock via the variable y. In the normal case (no contention, no collision), Lamport's algorithm requires two loads and five stores, executing in order the lines [1,2,3,9,19,21,22]. If a thread reaches line 3, though, and finds

[1] If one is willing to put an upper bound on the length of the critical section, then it is possible to implement multiprocessor mutual exclusion with fewer instructions than required by Lamport's algorithm. Such a limitation, though, is generally not feasible on a multiprocessor, and would be nearly impossible on a uniprocessor.

```
start:
     1      <b[i] := true;>
     2      <x := i;>
     3      if < y <> 0 > then       { Contention }
     4           < b[i] := false; >
     5           await < y = 0; >
     6           goto start;
     7      endif
     8
     9      < y := i; >
    10      if < x <> i > then       { Collision }
    11           < b[i] := false; >
    12           for j := 1 to N do await < b[j] = false; > end;
    13           if <y <> i > then
    14              await <y = 0;>
    15              goto start;
    16           endif;
    17      endif
    18
    19      CRITICAL SECTION
    20
    21      < y := 0; >
    22      < b[i] := false; >
```

Figure 1: Lamport's Fast Mutual Exclusion Algorithm

that the lock is held by another thread, there is contention and the thread must wait until the lock is released. The array b is used to resolve collisions, which occur whenever two or more threads find that the lock is free at line 3 and proceed to line 9 simultaneously (or through an interleaved schedule on a uniprocessor). A collision by n threads will be detected at line 10 by $n-1$ of them; those $n-1$ will enter the loop at line 12 and wait until the collisions have settled out (lines 12 through 15). The `<await>` used at lines 5, 12 and 14 is necessary when there is contention or collision, and can be implemented on a uniprocessor by having the awaiting thread yield its processor to the scheduler.

Although reservation-based algorithms such as Lamport's are correct in principle, they are in practice unwieldy, having worst-case waiting times that are $O(n)$ and storage requirements that are $O(n \times l)$, where n is the maximum number of threads that may be simultaneously active, and l is the maximum number of synchronization objects.

The space requirement can be reduced to $O(n)$ with a single "meta-atomic object" that is used to control access to all "regular atomic objects." In this case, the CRITICAL SECTION at line 19 in Figure 1 becomes a code sequence to access the "regular atomic object." For example, we can bundle the reservation algorithm inside a *Test-And-Set* procedure (see Figure 2).

Even though bundling reduces the space requirement for an atomic *Test-And-Set* variable to one bit (space for the meta variables x, y, and b can be counted as "constant" system overhead), it increases the number of memory accesses to enter and exit a critical section to (at least)

```
function Meta-Atomic-Test-And-Set(var p: integer): integer;
var result: integer;
begin
    [ lines 1 through 18 from Lamport's algorithm ]
    if (p = 0)  then
         result := 0;
         p = 1;
    else
         result := 1;
    end;

    [ lines 21 through 22 from Lamport's algorithm ]
    return result;
end AtomicTest-And-Set;

procedure AtomicClear(var p: integer)
begin
    p := 0;
end AtomicClear;
```

Figure 2: Bundled Test-And-Set Using Lamport's Algorithm

three loads and seven stores. Additionally, bundling serializes all atomic operations, even those for unrelated synchronization objects. On a uniprocessor, for example, preemption during the `Meta-Atomic-Test-And-Set` in Figure 2 operation would be disastrous to performance as it would prevent other threads from executing any other atomic operation.

2.3 Kernel Emulation

Memory interlocked instructions and software reservation protocols work on uniprocessors and multiprocessors alike. A strictly uniprocessor solution with low space overhead and not requiring special hardware is to have the kernel export its own mutual exclusion mechanism to applications by means of a system call that does an atomic read-modify-write on a memory location in the caller's address space. In the kernel, processor interrupts must be disabled during the execution of the atomic operation.

A particularly convenient implementation strategy is to define an unused opcode in a processor's instruction set to be a *Test-And-Set* pseudo-instruction; when a user-level program tries to execute the pseudo-instruction, a processor exception (*invalid opcode*) is raised and control transfers to the kernel's exception handler. There, the kernel determines the application's intentions, simulates the *Test-And-Set* with interrupts disabled (and therefore atomically), and returns to user-mode. Several versions of the Mach operating system for MIPS-based architectures implemented mutual exclusion in this way.

Involving the kernel on each synchronization operation has two problems. First, it simply increases the latency to enter and exit a critical section by a large number of instructions [Anderson et al. 91]. Not only must the trap be fielded and dispatched by the kernel, but registers must be saved and restored as control transfers across the user-kernel boundary, and the kernel must check

to ensure that the application is specifying a valid memory location as the operand to the *Test-And-Set* instruction. On the MIPS R3000, for example, emulating a *Test-And-Set* in the kernel takes about 100 instructions.

The second problem with kernel emulation is that it can increase the perceived occupancy time of critical sections, thereby increasing the amount of time that a thread holds a shared resource. Specifically, with kernel emulation, the *Test-And-Set* lock is held not only during the actual critical section as coded by the programmer, but also during the time taken to emulate the instruction in the kernel. In a preemptive system, this can increase contention for critical sections, as it increases the likelihood that a thread is preempted while holding a lock. This, in turn, can increase the number of context switches that must occur during the execution of a multithreaded program; on a uniprocessor, the only reasonable action to take when a *Test-And-Set* fails is to voluntarily relinquish the processor in the hopes that the thread for which the *Test-And-Set* last succeeded will soon clear it. Preemptive scheduling policies have been shown to interact badly with synchronization mechanisms on shared memory multiprocessors [Zahorjan et al. 89]. In Section 4 we show that preemptive scheduling combined with inflated critical sections can also affect performance on a uniprocessor.

2.4 Restartable Atomic Sequences – Optimistic Atomic Operations

The three mechanisms described so far are pessimistic. That is, they are designed to work in the face of untimely interrupts; a memory interlocked instruction implicitly delays interrupts until the instruction completes, a software reservation algorithm works in the presence of arbitrary interleaving, and kernel emulation explicitly disables interrupts during operations that must execute atomically.

On a uniprocessor, an atomic read-modify-write operation can be performed optimistically. Instead of using a heavyweight mechanism that works even if a thread is interrupted, it is easier to assume that an interrupt won't occur, but to recognize when it does and to recover. For any read-modify-write sequence that performs only one memory write as its final instruction, the recovery process is straightforward: *restart the sequence.* In this way, when the sequence eventually completes, it will have completed without interruption, i.e., atomically. An atomic *Test-And-Set* operation is shown in Figure 3. As long as statements 3 through 7 execute without interruption on a uniprocessor, this code will atomically read and write the variable p. If an interrupt does occur that would allow another thread to possibly modify the variable p, then the interrupted thread must resume execution at line 3 when it is next scheduled. As long as simple memory accesses execute atomically, the corresponding atomic clear operation can simply store a zero into p.

Restartable atomic sequences are attractive because they do not not require hardware support, have a short code path with one load and one store per atomic read-modify-write (in the common case of no interruptions), and do not involve the kernel on every atomic operation. Only when an atomic instruction sequence might not have executed atomically is it necessary to perform some "cleanup" action to ensure atomicity.

3 Kernel Support for Restartable Atomic Sequences

Restartable atomic sequences require a small amount of kernel support to ensure that a thread that is interrupted within an atomic sequence resumes at the beginning. This section describes two strategies for implementing restartable atomic sequences in the Mach operating system. The first strategy, and the simpler of the two, places the responsibility for detecting and recovering from an

```
   function Test-And-Set(var p: integer): integer;
   var result: integer;
   begin
1          result := 1;
2          BEGIN RESTARTABLE ATOMIC SEQUENCE
3          if p = 1 then
4              result := 0;
5          else
6              p := 1;
7          end;
8          END RESTARTABLE ATOMIC SEQUENCE
9          return result;
   end;
```

Figure 3: Test-And-Set using a Restartable Atomic Sequence

interrupted atomic sequence with the kernel. The second strategy places the responsibility with the application itself. We describe the strategies in terms of the MIPS R3000-based DECstation 5000, which does not have hardware support for memory-interlocked instructions.

3.1 Explicit Registration

With explicit registration, the kernel keeps track of each address space's restartable atomic sequence. Whenever the kernel resumes a thread that has been interrupted, it checks if the thread is being resumed within a restartable atomic sequence. If so, the thread is instead resumed at the top of the sequence. During program initialization, an application registers with the kernel the starting address of the sequence. The registration is done by the thread management system, which is automatically called at program startup.

Implementing explicit registration in the Mach kernel was straightforward. We added a new kernel call so that an address space can notify the kernel of its restartable atomic sequence, added a word to the kernel's address space control block to record the start of the sequence, and added a few lines of code which examines a thread's next user-level PC at the point where threads block in the kernel. We also added about a dozen lines of code to CThreads [Cooper & Draves 88], the user-level thread management package, to perform the registration at program startup.

An address space may register only one restartable atomic sequence at a time to simplify the kernel's task of checking if a thread had been interrupted within an atomic sequence. One sixteen byte sequence is sufficient for implementing a four cycle *Test-And-Set* function for the MIPS R3000. The assembly code for this function is shown in Figure 3.1. Line 1 loads the current value of the *Test-And-Set* location, passed in register a0, into the return value register, v0. Line 2 uses the load delay slot to load a temporary with the value 1. Line 3 transfers control back to the caller. Line 4, which executes in the branch delay slot following the return, stores a 1 into the *Test-And-Set* location. Lines 1–4 form the restartable atomic sequence: when the store *finally* occurs at the end of line 4, no other thread will have executed since the storing thread's most recent load at line 1.

There are two runtime costs associated with explicit registration. The first comes from the inability to inline atomic sequences. The kernel identifies restartable atomic sequences by a single

PC range, so the compiler cannot inline atomic sequences if an address space is to have more than one. The inability to inline slightly increases the overhead of atomic operations because of the cost of subroutine linkage. The second cost comes from having to check the return PC when a thread blocks. This test adds a few dozen cycles to a context-switch path that is already several hundred cycles long. As we show in the next section, blocks occur orders of magnitude less frequently than atomic operations. Consequently, we feel that it is worth spending a few extra cycles at context-switch time if we can significantly improve the performance of atomic operations.

```
    LEAF(Test-And-Set)
        # Test-And-Set location address in register a0
1       lw  v0, (a0)    #v0 = contents of a0
2       li  t0, 1       #temporary t0 gets 1
3       j   ra          #return to caller, result in v0
4       sw  t0, (a0)    #store 1 in Test-And-Set location
    END(Test-And-Set)
```

Figure 4: Restartable *Test-And-Set* Using Explicit Registration

3.1.1 Portability and Backward Compatibility

A diverse computing environment may have both uniprocessors and multiprocessors based on the same processor architecture. For example, at CMU, Mach runs on laptops and several different shared memory multiprocessors, all of which are based on the Intel 386 microprocessor. On a multiprocessor, though, restartable atomic sequences cannot be used to implement mutual exclusion. Either memory-interlocked instructions or software reservation must be used. Nevertheless, binary compatibility between the systems is important from the standpoint of software development and system maintenance.

One way to achieve portability would be to put a conditional at the beginning of each restartable atomic sequence: if running on a uniprocessor, use a restartable atomic sequence, otherwise use a technique that works on a multiprocessor. Unfortunately, the test would add several cycles to each atomic operation.

A better approach, and the one we've taken, is to execute the conditional only once at program initialization time. When a program's thread management system attempts to register its restartable atomic sequence with a kernel that does not support such sequences (for example, on a multiprocessor), the registration fails. In response to the failure, the thread management system overwrites the restartable atomic sequence with code that uses a conventional mechanism. Overwriting is also done to ensure backward compatibility so that new programs can run on old kernels.

3.2 User-Level Restart

Explicit registration places the responsibility for the detection and correction of atomicity violations with the kernel. An alternative approach places that responsibility with the application itself: a thread that blocks in the kernel is returned to user level at a fixed location, whereby its "execution state" can be determined by code at that location. If the most recent instruction executed at user level by the just-resumed thread is part of an atomic sequence, then the user-level code branches

to the beginning of the sequence, otherwise it branches to the instruction where the thread left off. Figure 5 illustrates this control flow for two threads in the same address space; thread 1 is preempted during a restartable atomic sequence, whereas thread 2 is preempted while executing undistinguished code. On reschedule, each thread resumes at the fixed sequence in its address space; for thread 1, control returns to the beginning of its atomic sequence, for thread 2, control returns to where it left off.

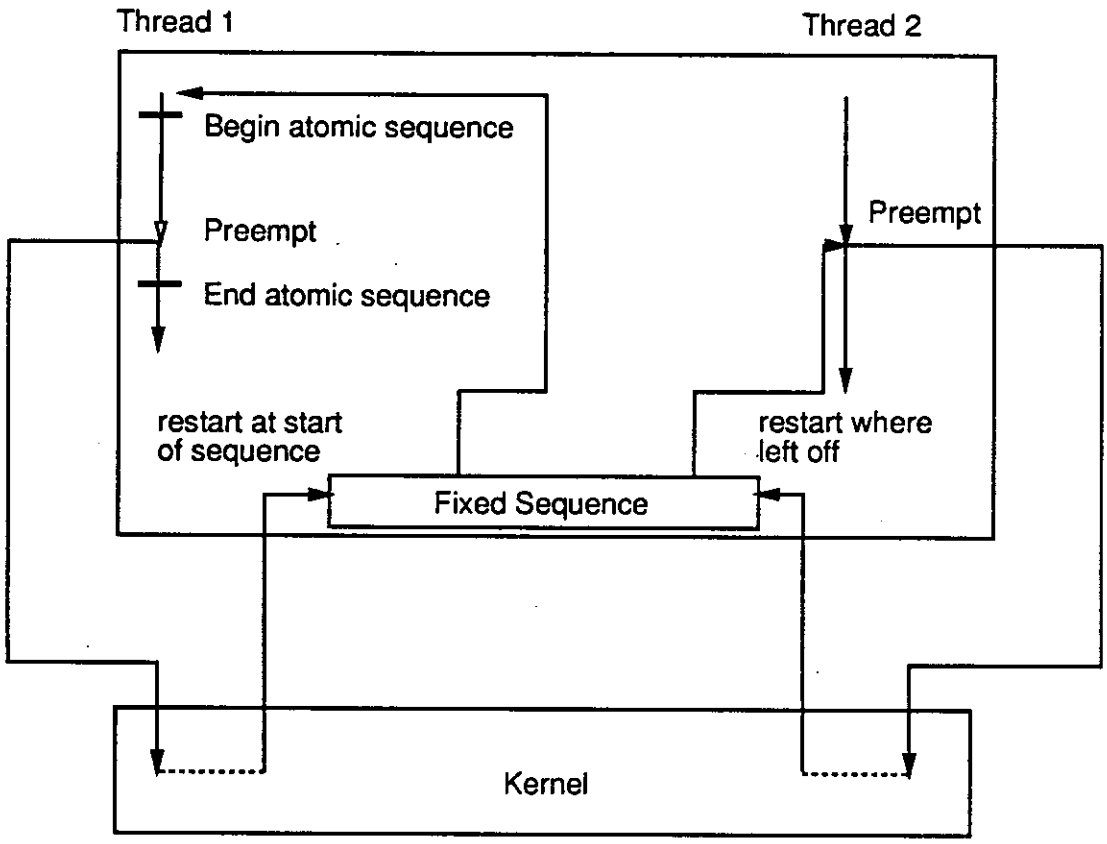

Figure 5: User Level Support for Restartable Atomic Sequences

Relative to explicit registration, user-level detection is attractive because the kernel provides only the mechanism to ensure atomicity; the policy determining what must be atomic is with the application. Since the kernel is not involved in either detection or correction, those processes can be made as rich as necessary to satisfy the atomicity constraints of *any* instruction sequence. For example, restartable atomic sequences can be inlined by having the fixed sequence at user level inspect the instruction stream to determine if a thread was interrupted within an atomic operation. User-level restart makes it possible to support a diverse set of synchronization mechanisms, such as those that manipulate wait-free data structures [Herlihy 91], as well as the more conventional *Test-And-Set*.

The user-level approach is not without problems, however. Transferring first to a fixed instruction sequence, and then to the actual return address involves slightly more overhead than the simple check made by the kernel in the explicit registration scheme. In addition to the extra level of indirection, the "real" return address has to be saved and restored on the thread's user-level stack; our user-level detection and restart code for the MIPS adds about 45 instructions to every

preemptive context switch. Finally, if user-level restart is used to inline, portability and backward compatibility become difficult to achieve.[2]

Although we have implemented the kernel mechanisms to support user-level detection and restart for Mach, it is only being supported experimentally. Backward compatibility and portability are important aspects of Mach, the performance degradation due to not being able to inline code sequences is small, and the policy/mechanism separation is less attractive when there is only one policy to support (Mach's user-level thread management system uses only *Test-And-Set* internally). These points have caused us to concentrate on the explicit registration scheme. As we gain more experience with using restartable atomic sequences, diverse synchronization mechanisms, and inexpensive synchronization, we may choose to abandon explicit registration in favor of the more flexible user-level approach.

We *are* using user-level restart in a preemptive coroutine package for Unix processes in which time-slicing is implemented with Unix signals. Within the user-level signal handler, we examine the interrupted PC and if it points to code within a *Test-And-Set* sequence, we roll the PC back up to the beginning of the sequence. This has proven especially valuable in the coroutine package, because it means that we do not have to disable and reenable Unix signals across each atomic operation.

4 Comparing the Performance of Three Software Techniques

In this section we compare the performance of restartable atomic sequences, kernel emulation and software reservation for the RISC-based DECstation 5000 (DS5000). The DS5000 has a 25 Mhz MIPS R3000 processor, but does not support atomic read-modify-write memory accesses in hardware.

We discuss performance at three levels. First, we examine the basic overhead of the three mechanisms. Next, we examine the effect that each has on the performance of common thread management operations. Finally, we take a system-wide perspective by looking at the effect that synchronization overhead has on the performance of several real applications. In brief, we show that:

- Using restartable atomic sequences instead of kernel-emulation, the performance of multithreaded applications can be improved substantially.
- Even single threaded applications, because they deal with multithreaded operating system servers, can benefit indirectly from inexpensive mutual exclusion.
- Preemptive context switches occur much less frequently than atomic operations, justifying the small amount of extra work done during context switch in order to improve the performance of atomic operations.
- Atomic operations, even when expanded out to multi-instruction atomic sequences, are almost never interrupted. It other words, restart is almost never required.
- Short critical sections that become inflated when synchronization operations are costly can increase lock contention. This can degrade performance by increasing the frequency with which threads find locks held and must therefore block.

[2]While dynamic code rewriting could be used for inlined code, it would incur such a large startup overhead that it is unlikely to be practical.

4.1 Microbenchmarks

We compare the performance of the three software-based mutual exclusion mechanisms with a test that enters a critical section using a *Test-And-Set* lock, increments a counter, and leaves the critical section by clearing the *Test-And-Set* lock. A single thread performs the test inside a loop a large number of times. The update to the counter is included so as to model a real critical section: interactions between the atomic operation, the code in the critical section, and the memory system must be considered when evaluating a mutual exclusion mechanism (e.g., a scheme requiring several writes will not work well on a memory system that has a write-through cache and a shallow write-buffer). The test has only one thread passing through the critical section, so the *Test-And-Set* always succeeds. Consequently, we are not measuring the performance of the thread management system itself (context switching, scheduling, etc), but rather the basic processor architecture, memory system and mutual exclusion mechanism.

The elapsed times to execute the three software-based mutual exclusion algorithms are shown in Table 1. The values in the table were determined by setting the loop limit to 1,000,000, computing the average elapsed time of each pass through the loop, and subtracting off the loop overhead. We ran the benchmarks several times on an unloaded system and observed little variance in the times.

Mechanism	Time (μsecs)
Restartable Atomic Sequences (branch)	.64
Restartable Atomic Sequences (inlined)	.51
Kernel Emulation	4.15
Software-reservation (a)	1.51
Software-reservation (b)	1.16

Table 1: Microbenchmark results for the DS5000

Software-reservation protocol (a) is an implementation of Lamport's fast mutual exclusion algorithm in which each lock is represented by a data structure containing an owner and a reservation field (one word each), and an array of booleans indexed by a thread identifier. It is the most direct implementation of the algorithm, but suffers from the high storage requirements described in Section 2.2. Protocol (b) uses Lamport's algorithm to implement the "meta" mutual exclusion function shown in Figure 2. Protocol (b), despite an increase in the number of memory access over Protocol (a), executes more quickly on the DS5000 because of the cost of having to compute a thread's unique identifier and the address of its "busy" bit. In essence, with protocol (a), these have to be computed on entry *and* exit to a critical section, whereas with protocol (b), they need only be computed on entry.

Restartable atomic sequences were measured with out-of-line branches to an explicitly registered sequence, and also with inlined code. The performance difference between the two approaches is due to the subroutine linkage overhead on the MIPS. Kernel emulation and both reservation schemes use out-of-line calls to implement the atomic operations. For these mechanisms, the number of instructions executed is already sufficiently high that there is little to be gained by inlining.

The table shows that kernel emulation is by far the most expensive approach; the trap and exception dispatch in the kernel are the main sources of overhead. Both software reservations schemes are faster than kernel emulation, but much slower than restartable atomic sequences due to the large number of instructions and memory accesses required. In contrast to the reservation protocols, a restartable atomic sequence involves very few instructions on the fast path.

4.2 The Effect On Thread Management Overhead

The Mach user-level thread management system, CThreads, like other thread management packages [Anderson et al. 89, Bershad et al. 88, Weiser et al. 89], relies heavily on simple atomic operations to implement high-level abstractions such as threads, locks and condition variables. To evaluate the effect that atomic operations have on the performance of the higher level facilities, we examine the performance of several benchmarks that stress different thread management functions in two different versions of CThreads: one uses kernel emulation for *Test-And-Set* and the other uses restartable atomic sequences. For the reasons mentioned earlier, our CThreads implementation relies on explicit registration rather than user-level detection, or software reservation,

We selected four thread management benchmarks typical of the kinds of thread management operations found in many multithreaded programs. The benchmarks were:

- *Spinlock.* Repeatedly acquire and release a spinlock. If the spinlock, which is implemented with a *Test-And-Set* lock, is held the acquire fails and is immediately retried.
- *Mutexlock.* Repeatedly acquire and release a relinquishing mutex. If a thread attempts to acquire a held mutex, which is implemented with a *Test-And-Set* lock and a queue of waiting threads, the thread puts itself on the wait queue and relinquishes its processor.
- *Forktest.* Recursively fork off a large number of threads; i.e., thread 1 forks thread 2 which forks thread 3, etc. After forking, a thread immediately terminates.
- *Pingpong.* Two threads "pingpong" off one another in a tight loop, using a mutex and condition variable to execute in alternation.

The performance of these benchmarks is shown in Table 2. Each entry in the table represents the elapsed time per operation (i.e, one spinlock acquire and release, one mutex lock and unlock, one fork and exit, one ping and pong).

Benchmark	**Time (μsecs)**	
	Emulation	**R.A.S**
Spinlock	4.3	.58
MutexLock	4.6	.91
ForkTest	43.7	23.8
PingPong	230.8	115.2

Table 2: Effect of Synchronization on Thread Management Overhead

Table 2 shows that the performance of thread management operations depends heavily on the performance of the underlying synchronization mechanism. When using kernel emulation for *Test-And-Set*, thread management functions spend the majority of their time getting in and out of the kernel to access synchronization code. With restartable atomic sequences, though, synchronization overhead becomes negligible. Even *PingPong*, with its profligate synchronization (26 *Test-And-Sets* per cycle), spends less than 10% of the time synchronizing when using restartable atomic sequences.

4.3 Application Performance

The microbenchmarks and thread management benchmarks indicate that restartable atomic sequences can have a large effect on individual operations. Ultimately, though, we are concerned

with performance system-wide. In this subsection we evaluate the overall performance improvement that comes from using restartable atomic sequences as opposed to kernel emulation when running "typical" applications on the DS5000. We selected four applications:

- *text-format.* Format this paper using LaTeX.
- *fs-bench.* A script of file system intensive programs such as copy, compile and search, run using the Andrew File System [Satyanaranyanyan et al. 85].
- *parthenon-n.* A resolution-based theorem prover that uses n threads to exploit or-parallelism [Bose et al. 89].
- *procon-64.* A producer-consumer application in which one consumer thread coordinates with one producer thread to read data from a large file into a fixed-size buffer of size 64.

Table 3 shows the behavior of the applications when run two under different versions of the operating system. The columns labeled "Emul" reflect runs using kernel emulation for the application and for Mach's user-level Unix server which implements much of the operating system environment [Golub et al. 90]. The columns labeled "R.A.S." reflect runs using restartable atomic sequences for the applications and for the Unix server. Each program was run several times and the average values for measurements taken during the runs are given in the table.

Program	Time (seconds)		Emulation Faults	Restarts	Thread Blocks	
	Emul.	R.A.S.			Emul.	R.A.S.
text-format	10.1	9.8	57305	0	295	317
fs-bench	239.4	231.1	2191276	42	8856	9876
parthenon-1	25.8	18.5	1395534	4	412	354
parthenon-10	26.1	18.6	1576714	7	610	499
procon-64	30.4	15.7	2738168	4	91494	106969

Table 3: Effect of Synchronization Overhead on Application Performance

Restartable atomic sequences have the greatest effect on applications that use threads explicitly, such as *parthenon* with 1 or 10 threads and *procon-64* which improve by about 30% and 50% respectively. Single-threaded "vanilla Unix" applications also benefit (indirectly) through the improved performance of the out-of-kernel multithreaded operating system services. For example, the performance of the file system benchmark, which itself uses only single threaded programs but relies on the multithreaded Unix server, improves by a little over 3%. The text formatter benefits the least of all the benchmark applications because Mach uses file mapping to reduce the frequency of server interaction for the file system operations performed most commonly by the text formatter – simple reads and writes to small files.

The column labeled "Emulation Faults" is a count of the number of kernel emulations that occurred during the running of each benchmark when *Test-And-Set* was implemented in the kernel. The column labeled "Restarts" shows the average number of atomic sequence restarts that had to be performed during the running of each program when *Test-And-Set* was implemented at user level with explicit registration. The restart count demonstrates that the likelihood of a thread being preempted during a restartable atomic sequence is extremely small. More than anything else, this column highlights the tremendous degree of pessimism present in the traditional mechanisms used

for mutual exclusion on uniprocessors. The optimistic approach instead represents the point of view that one should never have to pay to avoid something that almost never happens.

The number of emulation faults can be used to account for a large part of the performance difference between the two versions of the system. For example, we would expect *parthenon-10*, with its 1.57 million kernel emulations, to improve by about 1.57 million $\times$ 3.7 μsecs, or about 5.8 seconds. The actual improvement is slightly greater than this for two reasons. First, the correlation between elapsed time and number of emulation faults, even for the emulation based kernel, is neither strictly negative nor strictly positive. Hence, the number of emulation faults is only a good, but not exact, predictor of execution time. Second, some of the remaining performance improvement is due to the reduction in scheduling overhead that comes with a decrease in critical section service time.

For even very short critical sections (10 to 20 instructions) restartable atomic sequences add little extra overhead, and much of that overhead comes before the critical section has actually been entered. Consequently, a short critical section remains short, and the likelihood of the section being preempted is small. With kernel emulation, though, as mentioned earlier, each *Test-And-Set* takes about 100 instructions, and nearly all are executed with processor interrupts disabled. When control returns out of the kernel after an atomic operation has completed, interrupts are reenabled and any pending interrupts are delivered. If the delivered interrupt causes a preemption, then the thread that just performed the atomic operation will be descheduled and another thread will run. If that thread attempts to enter the same critical section, it will find the *Test-And-Set* variable already set and will be forced to relinquish its processor to CThread's user-level scheduler. As a result, the program pays the additional cost of descheduling and then rescheduling the unsatisfied thread.

We looked more closely at *parthenon-10* to determine the influence of inflated critical sections on program behavior. The program synchronizes often, but most synchronizations operations guard short critical sections that simply increment a counter, or dequeue an item from a linked list. In running the program, we counted the number of times that a thread was unable to enter a critical section because of a lock held by another (preempted) thread. When using kernel emulation in *parthenon-10*, a thread found a *Test-And-Set* lock held on average 603 times out of about 1.6 million attempts. In contrast, with restartable atomic sequence, only 387 times did a thread try to acquire a lock that was held. The only difference between the two runs was the lock holding time imposed by the synchronization mechanisms.

The last two columns show the number of times that a thread trapped into the kernel to block. For restartable atomic sequences, it indicates how many times a thread's execution state had to be checked to ensure that atomic operations eventually execute atomically. Comparing this column to the number of emulation faults justifies the small amount of extra work required by the restart strategies whenever a thread is rescheduled. The most compelling justification, of course, is the reduced execution time for all the applications.

5 Software vs. Hardware Support for Mutual Exclusion

We have so far used the lack of hardware support for atomic operations as motivation for investigating efficient software solutions. Many processors however do support some type of atomic read-modify-write instruction. Table 4 compares the overhead to acquire and release a *Test-And-Set* variable using memory-interlocked instructions and restartable atomic sequences on eight processor architectures. The table shows that restartable atomic sequences can be more efficient than memory interlocked instructions for the DEC CVAX, the Intel 486, the Motorola 88000, and Hewlett

Packard's Precision 9000.

For the interlocked cases, the *Test-And-Set* and subsequent release instructions were executed inline. In the restartable cases, only the release was inlined (implemented with a single clear memory instruction). The *Test-And-Set* code was called using a RISC-style linkage protocol: the address of the *Test-And-Set* variable, the return PC, and the result were passed in registers. The final column of Table 4 gives the number of μsecs required to transfer to the out-of-line restartable atomic sequence. The transfer is necessary when using explicit registration. If we were to instead rely on user-level detection, which would allow us to inline restartable atomic sequences, the software approach would outperform the hardware in nearly all cases (subtract the linkage overhead from the restartable atomic sequence overhead).

Processor	Interlocked *Test-And-Set*	Restartable *Test-And-Set*	Linkage Time μsecs
DEC CVAX	2.8	2.2	.6
Motorola 68030	1.1	2.0	.8
Intel 386	1.0	1.6	.7
Intel 486	.7	.6	.3
Intel 860	.3	.4	.2
Motorola 88000	.9	.3	.1
Sun SPARC	.8	1.0	.3
HP Precision 9000/835	5.5	2.1	.5

Table 4: Hardware and Software Costs of Atomic Operations

6 Related Work

We are aware of several other systems that use restartability to implement atomic operations on a uniprocessor. In the software arena, researchers at DEC SRC are using an approach in which the kernel detects preemptions that occur during inlined atomic sequences [Redell 90]. Their kernel "knows" the instruction sequences that implement the mutual exclusion operations P and V, and looks for these sequences whenever a thread is rescheduled. This approach combines the advantages of explicit registration (rapid kernel-level check) with user-level detection (atomic sequences can be inlined).

DEC SRC's approach requires a strong alliance between the compiler and the operating system, and requires that changes in the way that one handles synchronization be reflected in the other. In the DEC SRC environment, this is probably not a major concern, since the people who write the compilers work down the hall from those who build the kernels. They also have a system modelling technology that allows them to easily "remake" their entire installed software base when necessary. Finally, their intention to support only one language (Modula-2+ and its descendant Modula-3) makes it reasonable to embed information in the kernel about how the language runtime environment handle a small set of synchronization operations. In our environment, which is less cohesive than SRC's, it is not feasible for us to create such an alliance between the compiler and the operating system kernel because i) we import our compilers, ii) we have no real system modelling technology, and iii) we must support many different programming languages on top of Mach, each with its own calling convention and runtime environment.

The Synthesis kernel [Massalin & Pu 89] relies on a flavor of optimistic atomic operations that differs from ours. Synthesis' internal critical sections are implemented with Herlihy's wait-free

model of synchronization. Using *Compare-And-Swap* to update shared data structures, a thread can detect when it has been preempted within its critical section, and can restart the critical section on its own. This approach reduces lock contention and ensures that a thread that is preempted while updating a shared data structure does not prevent other threads from updating the same data structure. Nevertheless, the Synthesis kernel remains pessimistic ensuring atomicity with the processor's *Compare-And-Swap* instruction. Our count of the number of restarts required during scheduling-intensive applications indicates that a system like Synthesis would be able to use a lighter-weight, optimistic implementation of *Compare-And-Swap* on a uniprocessor.[3]

The Intel 860 processor [Intel860 89] has hardware support for restartable sequences. A thread begins a multi-instruction atomic sequence with a special instruction that sets a bit in the processor status word, disables interrupts, and locks the bus. The bit is cleared and the bus lock is automatically released on the next write through to memory, or after 32 cycles have elapsed, or if a processor exception occurs. The release on write covers the common case of a successful read-modify-write sequence; the release after 32 cycles ensures that a processor can't block out interrupts and lock out the bus indefinitely; the release in case of an exception ensures that program faults can be dealt with. On every path out of the kernel to user space after any type of exception, the kernel must check the bit in the processor status word and, if set, back the thread up to the point where it executed the special instruction.[4] Despite the 860's hardware support for restartable sequences (the bit in the processor status word subsumes the need to perform explicit registration or instruction stream inspection after a context switch), it offers relatively little advantage over software restart on a uniprocessor (see Table 4).

User-level detection and restart is similar to the approach taken in [Anderson et al. 90] to support user-level thread management on shared memory multiprocessors. In that system, when a thread is preempted inside a critical section, it is immediately resumed not where it left off, but within code that gives the thread management system the opportunity to "clean up" from the unexpected preemption. The machinery described for implementing the clean up is sufficient for implementing restartable atomic sequences on a uniprocessor.

7 Conclusions

Restartable atomic sequences represent a "common case" approach to mutual exclusion on a uniprocessor. In the common case, an atomic operation runs uninterrupted. The uncommon case can be detected after it occurs and can be handled by means of a simple recovery process, either in the kernel or at user level. As such, restartable atomic sequences are appropriate for uniprocessors that do not support memory interlocked atomic instructions. Moreover, on processors that do have hardware support for synchronization, better performance may be possible with restartable atomic sequences.

---

[3]Synthesis runs only on 680x0-based systems, which support reasonably fast atomic instructions (see Table 4. Consequently, there's no reason to believe that the performance of the system on *that* processor architecture could be significantly improved by using restartable atomic sequences.

[4]In practice, this extra check doesn't add much to the cost of getting in and out of the kernel, since managing the exposed pipeline already takes several hundred machine instructions [Anderson et al. 91].

Acknowledgements

Richard Draves, Hank Levy, and Dan Stodolsky provided valuable feedback on earlier drafts of this paper. David Redell and John Ellis of DEC SRC suggested the idea of restartable sequences with kernel-level registration after hearing me complain about the deficiencies of a certain RISC processor. Both were helpful in evaluating the tradeoffs between the different approaches and in understanding Taos's mutual exclusion mechanism.

References

[Accetta et al. 86] Accetta, M. J., Baron, R. V., Bolosky, W., Golub, D. B., Rashid, R. F., Tevanian, Jr., A., and Young, M. W. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.

[Anderson et al. 89] Anderson, T., Lazowska, E., and Levy, H. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[Anderson et al. 90] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy., H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. Technical Report 90-04-02, University of Washington, Department of Computer Science and Engineering, April 1990. Submitted for publication.

[Anderson et al. 91] Anderson, T., Levy, H., Bershad, B., and Lazowska, E. The Interaction of Architecture and Operating System Design. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 1991.

[Bershad et al. 88] Bershad, B. N., Lazowska, E. D., and Levy, H. M. PRESTO: A System for Object-Oriented Parallel Programming. *Software: Practice and Experience*, 18(8):713–732, August 1988.

[Birrell 89] Birrell, A. An Introduction to Programming with Threads. Technical Report #35, Digital Equipment Corporation's Systems Research Center, January 1989.

[Bose et al. 89] Bose, S., Clarke, E., Long, D., and Michaylov, S. Parthenon: A Parallel Theorem Prover for Non-Horn Clauses. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, 1989.

[Cheriton 88] Cheriton, D. R. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[Cooper & Draves 88] Cooper, E. C. and Draves, R. P. C threads. Technical Report CMU-CS-88-54, School of Computer Science, Carnegie Mellon University, February 1988.

[Dijkstra 68a] Dijkstra, E. W. The Structure of the "THE" Multiprogramming System. *Communications of the ACM*, 11(5), May 1968.

[Dijkstra 68b] Dijkstra, E. W. *Cooperating Sequential Processes*, pages 43–112. Academic Press, New York, 1968.

[Glew & Hwu 91] Glew, A. and Hwu, W. A Feature Taxonomy and Survey and Synchronization Primitives Implementations. Technical Report UILU-ENG-91-2211, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, February 1991.

[Golub et al. 90] Golub, D., Dean, R., Forin, A., and Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*, pages 87–95, June 1990.

[Herlihy 91] Herlihy, M. Wait-free Synchronization. *ACM Transactions on Programming Languages*, 13(1), January 1991.

[Intel386 90] *386 Programmer's Reference Manual.* Intel, Mt. Prospect, IL, 1990.

[Intel860 89] *i860 64-bit Microprocessor Programmer's Reference Manual.* 1989.

[Kane 87] Kane, G. *MIPS R2000 RISC Architecture.* Prentice Hall, Englewood Cliffs, N.J., 1987.

[Lamport 87] Lamport, L. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[Leonard 87] Leonard, T. *VAX Architecture Reference Manual.* Digital Equipment Corporation, 1987.

[Massalin & Pu 89] Massalin, H. and Pu, C. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Motorola 88100 88] *MCS 88100 RISC Microprocessor User's Manual.* Phoenix, AZ, 1988.

[Mullender et al. 90] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer Magazine*, 23(5):44–54, May 1990.

[Peterson 81] Peterson, G. Myths About the Mutual Exclusion Problem. *Information Processing Letters*, 12(1), June 1981.

[Redell 90] Redell, D. Personal Communication, December 1990.

[Rozier et al. 88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., and Neuhauser, W. The Chorus Distributed Operating System. *Computing Systems*, 1(4), 1988.

[Satyanaranyanyan et al. 85] Satyanaranyanyan, M., Howard, J., Nichols, D., Sidebotham, R., and Spector, A. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 35–50, December 1985.

[Thacker et al. 88] Thacker, C. P., Stewart, L. C., and Satterthwaite, Jr., E. H. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[Weiser et al. 89] Weiser, M., Demers, A., and Hauser, C. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[Zahorjan et al. 89] Zahorjan, J., Lazowska, E., and Eager, D. The Effect of Scheduling Discipline on Spin Overhead for Shared Memory Parallel Processors. Technical Report 89-07-03, University of Washington, Department of Computer Science and Engineering, July 1989. To appear in IEEE Transactions on Parallel and Distributed Systems.