A version of this paper will appear in the Fifth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS V). October 1992.

Fast Mutual Exclusion for Uniprocessors

Brian N. Bershad

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

David D. Redell and John R. Ellis

Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

Abstract

In this paper we describe restartable atomic sequences, an *optimistic* mechanism for implementing simple atomic operations (such as *Test-And-Set*) on a uniprocessor. A thread that is suspended within a restartable atomic sequence is resumed by the operating system at the beginning of the sequence, rather than at the point of suspension. This guarantees that the thread eventually executes the sequence *atomically.* A restartable atomic sequence has significantly less overhead than other software-based synchronization mechanisms, such as kernel emulation or software reservation. Consequently, it is an attractive alternative for use on uniprocessors that do not support atomic operations. Even on processors that do support atomic operations in hardware, restartable atomic sequences can have lower overhead.

We describe different implementations of restartable atomic sequences for the Mach 3.0 and Taos operating systems. These systems' thread management packages rely on atomic operations to implement higher-level mutual exclusion facilities. We show that improving the performance of low-level atomic operations, and therefore mutual exclusion mechanisms, improves 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, by the Open Software Foundation (OSF), and by a grant from the Digital Equipment Corporation (DEC). Bershad was partially supported by a National Science Foundation Presidential Young Investigator Award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, OSF, DEC, the NSF, or the U.S. government.

1 Introduction

In this paper we describe restartable atomic sequences, an *optimistic* mechanism for implementing atomic operations on a uniprocessor. Our approach assumes that short, atomic sequences are rarely interrupted. If a thread is interrupted during an atomic sequence, we rely on a recovery mechanism provided by the kernel that resumes the thread at the beginning of the sequence. We have implemented restartable atomic sequences in the Mach 3.0 [Accetta et al. 86] and Taos [Thacker et al. 88] operating systems, using a different method in each. 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 5000/200, which does not have hardware support for atomic operations. In addition, we show that restartable atomic sequences outperform hardware mechanisms on processors that do provide explicit support for atomic operations.

1.1 Motivation

Multithreaded programs use mutual exclusion to guarantee consistency of shared data structures. Mutual exclusion mechanisms such as *P, V* [Dijkstra 68a] and *acquire_mutex, release_mutex* [Birrell 91] are implemented using lower-level operations such as *Test-And-Set* that grant one of several threads mutually exclusive access to some data structure. Even on a uniprocessor, mutual exclusion is necessary to protect shared data against an interleaved thread schedule. Interleaving can occur when a thread is *suspended* (due to a synchronous fault or an asynchronous preemption), or when a thread *blocks* (due to the thread voluntarily relinquishing the processor).

Efficient mutual exclusion mechanisms are becoming increasingly important on uniprocessors for two reasons. First, modern applications use multiple threads as a program structuring device, as a mechanism for portability to multiprocessors, and as a way to manage I/O and server concurrency even when no true CPU parallelism is available. Second, many operating systems today are built on top of a microkernel that supports relatively few services; for example 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 microkernel approach exposes the performance of a system's mutual exclusion primitives; even single threaded programs rely on basic operating system services that are implemented outside the kernel using multiple threads. The performance of all applications is therefore ultimately influenced by the performance of the underlying mutual exclusion mechanisms.

The mechanisms that have been 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, their design assumes that atomicity may be violated at any moment (e.g., with an interrupt), and therefore guards against this potential violation every time the atomic operation is executed. This approach, though, can incur a high overhead that affects the performance of applications relying on mutual exclusion, either directly or indirectly.

In contrast, the optimistic mechanism described in this paper assumes that atomic sequences are rarely interrupted, and adopts an inexpensive solution for this assumed common case. We show that this assumption is both accurate, and effective at reducing the overhead of mutual exclusion.

1.2 The rest of this paper

In the next section we describe restartable atomic sequences after reviewing several pessimistic techniques for ensuring mutual exclusion on a uniprocessor. In Section 3 we discuss implementations of restartable atomic sequences for the Mach and Taos operating systems. In Section 4 we discuss some of the kernel design issues that arise when implementing restartable atomic sequences. In Section 5 we show the performance impact of using restartable atomic sequences in the Mach operating system. In Section 6 we show that restartable atomic sequences have less overhead than equivalent hardware mechanisms on several processor architectures. In Section 7 we discuss related work. In Section 8 we present our conclusions.

2 Implementing mutual exclusion on a uniprocessor

This section describes four techniques for implementing atomic primitives suitable for use by mutual exclusion mechanisms on a uniprocessor. We concentrate on the specific atomic primitive *Test-And-Set*, although other primitives, such as *Fetch-And-Add*, *Load-Linked/Store-Conditional*, and *Memory-Register-Exchange* could be similarly constructed. Each of these primitives performs an atomic read-modify-write of a single memory location. 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.

2.1 Memory-interlocked 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 atomic updates on arbitrary bit boundaries [Leonard 87], and the fact that atomic operations may bypass the on-chip cache [Motorola 88100 88]. A good survey of memory-interlocked instructions and their implementations can be found in [Glew & Hwu 91].

2.2 Software reservation

Atomic operations can also be constructed using software reservation algorithms, such as Dekker's [Dijkstra 68b], Peterson's [Peterson 81] or Lamport's [Lamport 87]. Roughly speaking, with software reservation algorithms, a thread must register its intent to perform an atomic operation and then wait until no other thread has registered a similar intent before proceeding. We use Lamport's fast mutual exclusion algorithm to evaluate software reservation schemes since it has been proven correct and shown to be optimal. If one is willing to put an upper bound on the duration 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.

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 [1,2,3,9,10,19,21,22]. If a thread reaches line 3, though, and finds 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.

```
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  end;
 8
 9  y := i;
10  if x <> i then      { Collision }
11    b[i] := false;
12    for j := 1 to N await (b[j] = false);
13    if y <> i  then
14        await (y = 0);
15        goto start;
16    end;
17  end;
18
19  CRITICAL SECTION
20
21  y := 0;
22  b[i] := false;
```

Figure 1: Lamport's fast mutual exclusion algorithm.

Although reservation-based algorithms such as Lamport's are correct in principle, they are in practice unwieldy, having 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" which 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).

```
function Meta-Atomic-Test-And-Set(var p: integer)
        :integer;
var result: integer;
begin
   [ lines 1-18 from Lamport's algorithm ]
   if (p = 0)  then
      result := 0;
      p = 1;
   else
      result := 1;
   end;
   [ lines 21-22 from Lamport's algorithm ]
  return result;
end Meta-Atomic-Test-And-Set;

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

Figure 2: Bundled *Test-And-Set* using Lamport's algorithm.

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 three loads and seven stores. Additionally, bundling serializes all atomic operations, even those for unrelated synchronization objects. On a uniprocessor, for example, a thread preempted during the function `Meta-Atomic-Test-And-Set` would prevent other threads from executing *any* atomic operation.

2.3 Kernel emulation

Memory-interlocked instructions and software reservation protocols work on both uniprocessors and multiprocessors. A strictly uniprocessor solution has the kernel export its ability to perform atomic operations 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 are disabled during the execution of the atomic operation.

Although kernel emulation requires no special hardware, its runtime cost is high. The kernel must be invoked on every synchronization operation, requiring that a trap be fielded and dispatched, state saved and restored, and arguments checked. On the MIPS R3000, for example, building a *Test-And-Set* with kernel emulation takes about 100 instructions.

```
   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 Test-And-Set;
```

Figure 3: Generic *Test-And-Set* using a restartable atomic sequence.

2.4 Restartable atomic sequences

The three mechanisms described so far are pessimistic. A memory-interlocked instruction implicitly delays interrupts until the instruction completes; a software reservation algorithm explicitly guards against arbitrary interleaving; 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 mechanism that guards against interrupts, we can instead recognize when an interrupt occurs and recover. For any read-modify-write sequence, 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 occurs 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. The corresponding clear operation can store a zero into p as long as single-word memory accesses execute atomically.

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 a recovery action to ensure atomicity. In the next section we describe two recovery strategies.

3 Implementing restartable atomic sequences

Restartable atomic sequences require kernel support to ensure that a suspended thread is resumed at the beginning of the sequence. This section describes two strategies for implementing that kernel support. The first strategy, used by the Mach 3.0 kernel, places a restartable atomic sequence at a designated code range within a program. The second strategy, used by the Taos kernel, constructs restartable atomic sequences out of unique code fragments against which a suspended thread's current instruction stream is compared. Both strategies have been implemented in versions of the operating systems running on the MIPS R3000-based DECstation 5000/200.

3.1 Explicit registration in Mach

The Mach operating system implements a strategy based on explicit registration. The kernel keeps track of each address space's restartable atomic sequence. If a thread is suspended within that sequence, it is resumed at the beginning. An application registers the starting address and length of the sequence with the kernel. The registration is done automatically during program initialization by C-Threads [Cooper & Draves 88], Mach's thread management package.

An address space may register only one restartable atomic sequence at a time. This restriction simplifies the kernel's task of determining if a suspended thread was executing within a restartable sequence. When the thread management system attempts to register a restartable atomic sequence with a kernel that does not support such sequences, 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 ensures binary portability between uniprocessors and multiprocessors, and binary compatibility with older kernels that do not support restartable atomic sequences.

A registered *Test-And-Set* function can be implemented with a single four-word (and four cycle) sequence on a load/store RISC architecture. For example, the assembly code for this function on a MIPS R3000 is shown in Figure 4. Line 1 loads the current value of the *Test-And-Set* location, passed in register `a0`, into the return value register, `v0`. Line 2 loads a temporary register with the value 1. Line 3 returns 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 thread's most recent load at line 1.

```
# Test-And-Set procedure.
Test-And-Set:
lw  v0, (a0)  #v0 = contents of a0
li  t0, 1     #temporary t0 gets 1
j   ra        #return to caller, result in v0
sw  t0, (a0)  #store 1 in Test-And-Set
              #location
```

Figure 4: Restartable *Test-And-Set* procedure using explicit registration in Mach 3.0.

Costs of explicit registration

There are two runtime costs associated with explicit registration. Because the kernel identifies restartable atomic sequences by a single PC range per address space, they cannot be inlined. 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 whenever a thread is suspended. Although this test adds a few tens of cycles to the kernel's thread suspension path (which is already several hundred cycles long), thread suspensions occur far less often than atomic operations, making the additional scheduling overhead worthwhile.

3.2 Designated sequences in Taos

Taos uses designated code sequences to recognize when a thread has been suspended within an atomic sequence. The kernel compares the instruction stream of a suspended thread against a designated sequence. The comparison allows restartable atomic sequences to occur anywhere in a program, enabling inlining and eliminating the branch overhead of explicit registration.

The kernel's comparison must recognize every interrupted sequence and reject any other similar looking sequence since mistakenly changing the PC in such a situation could cause code to malfunction. Taos uses a two-stage check to unambiguously recognize atomic sequences.

The first stage is a fast test which rejects most interrupted code sequences that are not restartable. The opcode of the suspended instruction is used as an index into a hash table containing instructions eligible to appear in a restartable atomic sequence. If the opcode matches the contents of the indexed entry, the test proceeds to the second stage. The first check is quite fast, yet succeeds in rejecting a large majority of the non-atomic cases and none of the atomic ones. The few that pass this check, comprising all of the suspended atomic sequences, plus a much larger number of false alarms, move on to the second stage of the check.

The second stage uses another table, again indexed by opcode, to determine the expected offset from the suspended instruction to a "landmark" no-op. The landmark no-op is never emitted by the compiler under normal circumstances, but is present within every restartable atomic sequence. On the R3000, the landmark no-op is a non-destructive register move which fills an otherwise useless branch delay slot. If the second stage finds the landmark in the expected position, it recognizes the sequence as atomic and restarts it. Otherwise, the sequence is rejected as a false alarm.

The designated sequence for acquiring a mutex is shown in Figure 5. The sequence is optimistic in two distinct senses: it assumes both that it will not be interrupted, and that it will find the mutex unlocked. Both assumptions model the frequent case, but either or both can fail independently. The sequence is essentially a *Test-And-Set* of an entire word, where the unlocked value of the mutex is 0, and the locked-but-no-waiters value is 0x80000000. Typically, the sequence finds that the mutex has the former value and atomically sets it to the latter. The infrequent case is handled with an out-of-line kernel call via `SlowAcquire`. The sequence for mutex release (*Test-And-Clear*) is similar.

```
lw   v0, (a0)          #get value of mutex
lui  t0, 8000H         #temporary t0=0x80000000
bne  v0, SlowAcquire   #branch if not common case
no-op                  #special landmark value
sw   t0, (a0)          #store locked value
```

Figure 5: A restartable atomic sequence for mutex acquisition using an inlined designated sequence.

Costs of designated sequences

Designated sequences have several costs. There is the measurable cost of the two-stage check on every thread switch. The check is currently implemented in Modula2+, the language in which the operating system is written [Rovner et al. 85]. As with Mach's explicit registration, the check adds a few tens of instructions to the kernel's context switch path (counting instructions in the generated code shows that the check adds about 2 μsecs on a MIPS R3000 in the common case).

Unlike explicit registration, which uses only one sequence that can be overwritten at runtime if restartable atomic sequences are not supported on a given system, designated sequences are not portable between uniprocessors and multiprocessors. The compiler must generate a different code sequence for each.

More generally, the use of a designated sequence requires a strong alliance between the compiler and the operating system, since changes in the way that one handles atomic operations must be reflected in the other. The global design properties of the Taos operating system make this linkage feasible, however. The crucial property of Taos is that both the kernel and its multithreaded clients are written in Modula-2+. In this context, the kernel and the compiler can cooperate closely to support fast mutual exclusion using designated inlined sequences. In contrast, for Mach, which is not intended to be used with any one language and any one compiler, such a close alliance between the compiler and the operating system kernel is not feasible.

4 Kernel design considerations

Section 3 described two kernel techniques that support fast mutual exclusion with restartable atomic sequence. The implications of these techniques for the inner workings of the kernel depend both on the exact technique chosen (explicit registration, or designated sequences) and on the design details of the specific kernel involved. In this section we discuss some of these implications.

4.1 Placement of the PC check

The most obvious question about kernel structure is: when should the kernel check/adjust the PC of a suspended thread? The two points at which the thread can be checked are when it is first suspended, and when it is about to be resumed. One could consider intermediate points, but they are less natural than either point where the kernel already has the thread in hand.

When using designated sequences, checking the PC can cause a page fault since it involves reading arbitrary user memory. If the kernel path leading to suspension is restricted in its ability to incur additional faults, as it is in Taos and many other systems, early checking of the PC with designated sequences can be problematic. Checking the PC late solves this problem, since there are generally fewer restrictions on kernel exceptions when coming out of a context switch.

In Mach, the PC is checked when the thread is suspended rather than when it returns to user level. Since only the PC, but not its contents, are inspected, there is no concern about touching user memory at inopportune times. The check is done early because the return PC and reason for entry into the kernel are conveniently available at that point.

Detection at user level

Explicit registration and designated sequences place with the kernel the responsibility for detecting and correcting atomicity violations. An alternative approach places that responsibility with the application itself: whenever a suspended thread is resumed by the kernel, it returns to a fixed user-level sequence that determines if the thread was suspended within a restartable atomic sequence. If so, the user-level recovery code branches to the beginning of the sequence, otherwise it branches to the suspended instruction.

User-level detection is attractive because the kernel provides only the mechanism to ensure atomicity. The policy lies 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, 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 suspended instruction involves more complexity and overhead than the simple check made by the kernel in either of the other two strategies. There is a level of control indirection requiring that the real return address be saved and restored on the thread's user-level stack at each suspension. Because of these problems, and because there is little motivation to create a clean policy/mechanism separation when there is only one policy, neither Taos nor Mach provide for user-level detection.[1]

4.2 Mutual exclusion in the kernel

The kernel is itself a client of thread management facilities in both Mach and Taos. It is tempting to regard the kernel's ability to disable interrupts as a sweeping solution to the mutual exclusion problem on a uniprocessor. Mach implicitly adopts this approach as the kernel is non-preemptive, but is compiled for uniprocessors with all low-level synchronization operations removed. The Taos kernel, however, is preemptive, and uses designated sequences just as applications do. There are two reasons for this. The first is a minor performance gain, since explicit disabling and reenabling of interrupts would more than double the cost of synchronization operations. The second reason is a desire to use the same Modula-2+ compiler for all code, whether it be user code or kernel code.

The use of restartable atomic sequences in both user programs and the kernel raises the question of system structuring due to potential recursion. Two events, a page fault or a thread preemption, can trigger a thread switch in the middle of a restartable atomic sequence. Since the sequence may be in either user or kernel code, there are then four events that must be considered in the light of recursion: user page fault, user preemption, kernel page fault, and kernel preemption. The kernel uses mutexes while handling these events, so it is important to ensure that recursion does not lead to deadlock. For example, a thread could incur a user page fault, be preempted while handling it in the kernel, and upon resuming from the preemption, incur a second page fault when trying to do its PC check. If the preemption happened while holding a lock in the virtual memory system, the recursion could cause the thread to deadlock with itself.

The problem here is that careless ordering of the PC check could lead to mutual recursion between the thread scheduler and the virtual memory system. Such

[1] At CMU, we rely on user-level restart in a preemptive coroutine package for Unix systems that is used in teaching an undergraduate operating systems course. We examine the interrupted PC within the Unix signal handler, and roll it back if necessary. With this, we avoid disabling and enabling Unix signals during every synchronization operation.

problems are avoided in Taos because the system is structured to impose a strict ordering on the four events listed above. The handling of any event can cause only lower-level events. A user page fault can incur kernel page faults and kernel preemptions, but a kernel preemption (including the PC check at restart) can not incur kernel page faults. Resuming from a user preemption, by contrast, is allowed to incur page faults. By consistently ordering the PC checks, Taos is able to use restartable atomic sequences at all levels of the system without risk of deadlock or endless recursion.

5 The performance of three software techniques for mutual exclusion

In this section we compare the performance of restartable atomic sequences, kernel emulation and software reservation on a RISC-based DECstation 5000/200 running the Mach 3.0 kernel (version MK42) and CMU's Unix server (version UX23) [Golub et al. 90]. The DECstation 5000/200 has a 25 Mhz MIPS R3000 processor which 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 various mechanisms. Next, we examine their effect on the performance of common thread management operations. Finally, we take a system-wide perspective and look at the effect that mutual exclusion overhead has on the performance of several 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.
- Thread suspensions occur much less frequently than atomic operations, justifying the small amount of extra work done during thread switch in order to improve the performance of atomic operations.
- Restartable atomic sequences are almost never interrupted, validating the optimistic approach.

Although we have not collected detailed performance information in Taos, we believe that the results would be similar.

5.1 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. The test uses only one thread, 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 that of the basic processor architecture, memory system and mutual exclusion mechanism. 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 should be considered when evaluating a mutual exclusion mechanism. For example, a scheme requiring several writes will not work well on a memory system with a write-through cache and a shallow write-buffer [Bershad et al. 92].

The elapsed times to execute the various software-based mutual exclusion algorithms are shown in Table 1. The values in the table were determined by executing the test in a tight loop 1,000,000 times, computing the average elapsed time of each pass through the loop, and subtracting off the loop overhead. There was only negligible variation in times over several runs of the benchmarks on an unloaded system.

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

Table 1: Microbenchmark results for the DECstation 5000/200.

Restartable atomic sequences were measured with 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 overhead is sufficiently high that there is little to be gained by inlining. 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 accesses over Protocol (a), executes more quickly on the DECstation 5000/200 because of the cost of having to compute a thread's unique identifier and the address of its "busy" bit. With protocol (a), these must be computed on entry *and* exit to a critical section, whereas with protocol (b), they need only be computed on entry. A dedicated per-thread hardware register would reverse this disparity.

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 number of instructions and memory accesses required. Despite their better performance, both reservation strategies have practical problems that make them difficult to use (see Section 2.2). Consequently, in the rest of this section, we restrict our comparisons to systems using only restartable atomic sequences and kernel emulation.

5.2 Thread management overhead

Mach's user-level thread management system, C-Threads, 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 higher level facilities such as threads, locks and condition variables. We looked at several benchmarks to understand the influence that atomic operations have on the performance of these higher level facilities using two different versions of C-Threads. One version relies on kernel emulation for synchronization. The other uses restartable atomic sequences. The benchmarks, which contain the kinds of operations typically found in multithreaded programs, are:

- *Spinlock.* One thread repeatedly acquires and releases a spinlock. The spinlock is implemented with a *Test-And-Set* sequence.

- *Mutexlock.* One thread repeatedly acquires and releases a relinquishing mutex. Unlike a spinlock, if a thread tries to acquire a held mutex, it relinquishes the processor. The mutex is implemented using a spinlock and a queue of waiting threads.

- *Forktest.* Threads are recursively forked in succession; 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 alternately.

The performance of these benchmarks running on a DECstation 5000/200 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). The table shows that the performance of thread management operations depends upon the performance of the underlying synchronization mechanism. When using kernel emulation for *Test-And-Set*, thread management functions spend the majority of their time in the kernel executing synchronization code. With restartable atomic sequences, synchronization overhead becomes negligible. Even *PingPong*, with its profligate synchronization (26 *Test-And-Set*s per cycle), spends less than 10% of its time synchronizing when using restartable atomic sequences.

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

Table 2: The effect of synchronization on thread management overhead under Mach 3.0 on a DECstation 5000/200.

5.3 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 examine the effect that restartable atomic sequences have on the performance of several applications running on Mach 3.0. The applications are:

- *text-format.* Format a version of this paper using LaTeX.

- *afs-bench.* A script of file system intensive programs such as copy, compile and search that execute out of 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 64 byte buffer.

Table 3 shows the behavior of the applications when run under two 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. 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.

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

Program	Elapsed Time (secs)		Emulation Traps	Restarts	Thread Suspensions	
	Emul.	R.A.S.			Emul.	R.A.S.
text-format	10.1	9.8	57305	0	295	317
afs-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	106969	91494

Table 3: Effect of synchronization overhead on application performance.

improve by about 30% and 50% respectively. Single-threaded "vanilla Unix" applications also benefit indirectly through the improved performance of multi-threaded user-level operating system services. For example, the performance of the text-formatter and the file system benchmarks, which are themselves single threaded but rely on the multithreaded Unix server, improves by about 3%.

The column labeled "Emulation Traps" reflects the number of synchronizations that occurred when atomic operations were implemented in the kernel. The column labeled "Restarts" shows the average number of atomic sequence restarts that had to be performed when *Test-And-Set* was implemented with explicit registration. The restart count demonstrates that the likelihood of a thread being suspended during a restartable atomic sequence is extremely small.

The last two columns show the number of times that the kernel suspended a thread. 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 more compelling justification, of course, is the reduced execution time for the applications.

The number of emulation traps can be used to account for the performance difference between the two versions of the system. For example, *parthenon-10*, with its 1.57 million kernel emulations, should improve by about 1.57 million $\times$ 3.7 μsecs (4.3 μsecs$-$.58 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 traps is neither strictly negative nor strictly positive. Hence, the number of emulation traps is only a good, but not exact, predictor of performance improvement. Second, some of the 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 critical section itself being suspended is small. With kernel emulation, though, each *Test-And-Set* takes about 100 instructions, and nearly all are executed with processor interrupts disabled. When control returns out of the kernel, 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 relinquish its processor to the scheduler.

We looked more closely at *parthenon-10* to determine the influence of inflated critical sections on program behavior. The program synchronizes often, but most synchronization 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 (suspended) thread. When using kernel emulation in *parthenon-10*, a thread found a *Test-And-Set* lock held about twice as often as with restartable atomic sequences.

6 Software vs. hardware support for mutual exclusion

The lack of hardware support for atomic operations offered the initial motivation to investigate efficient software solutions [Anderson et al. 91]. Most processors, however, do support some type of atomic read-modify-write instruction. In this section we evaluate the use of restartable atomic sequences on such processors.

We measured the overhead to acquire and release a *Test-And-Set* lock using memory-interlocked instructions and restartable atomic sequences on eight processor architectures. The results are shown in Table 4. For the interlocked cases, the times do not include any linkage overhead, as the *Test-And-Set* and subsequent release instructions can be executed inline. In the cases of explicit registration, linkage overhead is included for the *Test-And-Set*, but not for the release, which can be inlined. The fourth column of Table 4 shows the call linkage overhead. Even with the linkage overhead, restartable atomic sequences are more efficient than memory-interlocked instructions on the DEC CVAX,

Processor	Interlocked Instruction (μsecs)	Explicit Registration (μsecs)	Linkage Overhead (μsecs)	Designated Sequence (μsecs)
DEC CVAX	2.8	2.2	.6	1.6
Motorola 68030	1.1	2.0	.8	1.2
Intel 386	1.0	1.6	.7	.9
Intel 486	.7	.6	.3	.3
Intel 860	.3	.4	.2	.2
Motorola 88000	.9	.3	.1	.2
Sun SPARC	.8	1.0	.3	.7
HP 9000 Series 700	.94	.17	.08	.09

Table 4: Hardware and software overheads of *Test-And-Set* using different implementation strategies.

the Intel 486, the Motorola 88000, and the Hewlett Packard 9000 (PA-RISC) Series 700.

Using designated sequences, the software approach outperforms the hardware in all cases (subtract the overhead of linkage from that of an explicitly registered sequence). As processor speeds increase relative to bus and memory speeds, we expect the optimistic software solution to continue its dominance. For interlocked instructions to outperform optimistic software techniques on uniprocessors, they must be implemented so that they exploit the simpler single processor case.

The table demonstrates that one should not necessarily rely on an architecture and memory system to provide functions that may be provided more cheaply with a combination of operating system, compiler, and runtime support.

7 Related work

The Trellis/Owl object-oriented language [Moss & Kohler 87] used optimistic synchronization techniques similar to those described in this paper. The Owl runtime system provided concurrency among several threads sharing a single VMS process, and used software interrupts from VMS to drive its multiplexing. It provided atomicity for its own needs and those of user programs by backing out of certain registered runtime routines, and by emulating forward through designated sequences. The most important difference between Owl and the work described in this paper is our integration of restartable atomic sequences with the operating system kernel.

User-level detection and restart is similar to the approach taken in [Anderson et al. 92] 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 recover from the preemption. This machinery is sufficient for implementing restartable atomic sequences on a uniprocessor.

The Intel i860 processor [Intel860 89] provides 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 to memory, after 32 cycles, or on a processor exception. The release on write covers the common case of a successful read-modify-write sequence. The kernel must check the bit on every transfer from the kernel to user level. If the bit is set, the kernel must back the thread up to the special instruction. Despite the i860's hardware support for restartable sequences (the bit in the processor status word eliminates the need to perform explicit registration or instruction stream inspection after every context switch), it offers little performance advantage over software techniques on a uniprocessor (see Table 4).

8 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. 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.

Acknowledgements

Richard Draves, Hank Levy, Chris Maeda, Dan Stodolsky and Terri Watson provided valuable feedback on earlier drafts of this paper. The use of restartable atomic sequences in Taos benefitted from discussions with Butler Lampson and Mike Burrows. The system structuring ideas in Section 4 were clarified during discussions with Jerry Saltzer.

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. 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.

[Anderson et al. 92] 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. *ACM Transactions on Computer Systems*, 9(1), February 1992.

[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.

[Bershad et al. 92] Bershad, B. N., Draves, R. P., and Forin, A. Using Microbenchmarks to Evaluate System Performance. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992.

[Birrell 91] Birrell, A. *An Introduction to Programming with Threads*. Prentice Hall, 1991.

[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 of Synchronization Primitive 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.

[Moss & Kohler 87] Moss, J. and Kohler, W. Concurrency Features for the Trellis/Owl Language. In *European Conference on Object-Oriented Programming*, June 1987. Appears in Springer-Verlag's Lecture Notes in Computer Science #276.

[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.

[Rovner et al. 85] Rovner, P., Levin, R., and Wick, J. On Extending Modula-2 for Building Large, Integrated Systems. Technical Report # 3, Digital Equipment Corporation's Systems Research Center, Palo Alto, California, January 1985.

[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.