A Fast and General Software Solution to Mutual Exclusion on Uniprocessors

David Mosberger, Peter Druschel, and Larry L. Peterson [1]
{davidm,druschel,llp}@cs.arizona.edu

TR 94-07

Abstract

This paper presents a technique to solve the mutual exclusion problem for uniprocessors purely in software. The idea is to execute atomic sequences without any hardware protection and, in the rare case that the atomic sequence is interrupted, to rollforward to the end of the sequence. The main contribution of this paper is to discuss the OS-related issues of this technique and to demonstrate its practicality, both in terms of flexibility and performance. It proposes a purely software-based technique that achieves mutual exclusion *without any memory-accesses*. Experiments show that this technique has the potential to outperform equivalent hardware mechanisms.

June 24, 1994

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1] This work supported in part by ARPA Contract DABT63-91-C-0030, by Digital Equipment Corporation, and Hewlett-Packard.

1 Introduction

Atomic sequences—a sequence of instructions that needs to execute without interference—are fundamental to concurrent programs, especially operating systems. For this reason, processor architectures generally provide primitive operations, such as test-and-set and compare-and-swap, that can be used to implement the mutual exclusion required by atomic sequences. Due to the trend of implementing processors that are suitable for use in shared-memory multiprocessors, however, these primitives have become quite expensive. For example, the Alpha architecture [Sit92] provides a multiprocessor-safe load-linked/store-conditionally instruction that, even on a uniprocessor, takes over 100 cycles to move a 64 bit integer atomically between two memory locations. This is slow and—without any doubt—it is possible reduce this overhead significantly. Nevertheless, it shows that hardware primitives are designed and optimized for the multiprocessor case.

On a uniprocessor, it is possible to implement atomic sequences by disabling interrupts, which are the only possible source of interference[1]. Although an order of magnitude faster than load-linked/store-conditionally, operations for disabling interrupts are usually privileged, meaning that they cannot be directly invoked by user processes. It is also the case that support for hierarchical priority levels is more expensive than one might hope [SCB93].

Because of the limitations of these two hardware-based approaches to implementing atomic sequences, Bershad, Redell, and Ellis have proposed a software-only implementation of mutual exclusion on uniprocessors [BRE92]. However, their solution is applicable only to simple atomic sequences—those that involve only one store to shared memory. While this is sufficient to implement synchronization primitives that can in turn be used to construct higher-level synchronization objects, the approach is still limited. First, when used to build lock-based synchronization mechanisms, this approach is not appropriate for data-structures that are shared with interrupt handlers, as this code cannot risk blocking. (In general, lock-based solutions introduce the complexity of deadlock.) Second, while it is possible to create lock-free data-structures based on some of these primitives, it is well-known that this approach can incur significant overheads, often requiring reference counts and/or shadow copies of shared objects [Mas92, Her93].

This paper describes an alternative software-based approach to implementing atomic sequences on uniprocessors. Like the Bershad et al. solution, the idea is to address the only source of interference—interrupts. Unlike their approach, which uses rollback (an atomic sequence that has been interrupted is rolled back to the beginning), our solution uses *rollforward* (when an interrupt interferes with an atomic sequence, the interrupt handler arranges for the sequence to be executed to the end before resuming interrupt processing). This approach can be used to implement lock-free data structures efficiently and puts few restrictions on the code that can make up the atomic sequence. In particular, the number of stores is practically unlimited. As the technique is purely software-based it can be used to achieve mutual exclusion at any privilege level. This makes it equally well suited for implementation within the OS kernel (e.g., for device drivers) and in user-level processes (e.g., to protect against asynchronous events such as Unix signals [LMKQ88] or VMS Asynchronous System Traps [GS92]). Finally, even though this technique is not directly applicable to multiprocessor synchronization, it is useful in constructing lock-free data-structures on machines that do not have sufficiently powerful hardware-primitives [Ber93].

While the idea is straight-forward, there are several issues that make this technique non-trivial in an OS context. In particular, it is necessary to address issues like interactions with the memory system, multiple address spaces, and multiple trust domains. The main contributions of this paper is to explore this design space, and to quantify the performance of specific points in that space.

The rest of the paper is organized as follows. Section 2 describes the issues involved in using rollforward as a technique to achieve mutual exclusion. Section 3 then presents three concrete solutions and Section 4 compares the performance of these implementations with several hardware-based mutual exclusion schemes. Finally, Section 5 identifies related work and Section 6 offers some concluding remarks.

[1] DMA by devices is not of concern here as device interfaces usually have there own synchronization mechanisms.

2 Design Issues

The idea to use rollforward as the mechanism to ensure the atomic execution of any atomic sequence is straight-forward. If an interrupt handler determines that it pre-empted an atomic sequence, it arranges to execute the sequence to the end before proceeding with interrupt handling. The practicality of this mechanism therefore depends largely on the potential for unwanted effects due to rollforward.

This section first lays out the design space for using rollforward as a technique to achieve mutual exclusion. Two orthogonal dimensions are involved: the mechanism to register an atomic sequence and the mechanism to rollforward an atomic sequence. A final subsection then examines the issues that impact the practicality of rollforward.

2.1 Registering Atomic Sequences

The first choice in designing a software-based atomicity scheme involves the mechanism to register an atomic sequence. Clearly, to perform a rollforward (or rollback), an interrupt handler must know about the ranges (extents) of code that constitute atomic sequences. Many registration schemes are conceivable and we present only four possibilities in that spectrum. The first two were previously suggested by Bershad et al. [BRE92]; we repeat them here for completeness. We found that both approaches rather seriously limit the flexibility with which software mutual exclusion can be used. The third and fourth approach rid these limitations at the cost of slightly higher overheads.

2.1.1 Designated Sequences

The first proposal in [BRE92] is to use designated instruction sequences to mark atomic sequences. This is also the technique used in [MK87]. Designated sequences must not appear anywhere but in atomic sequences. An interrupt handler can check whether the PC is inside an atomic sequence by matching the surrounding code according to a set of templates. If a template matches, the interrupted handler concludes that it pre-empted an atomic sequence and performs the appropriate recovery action. The advantage of designated sequences is that they do not incur any overheads per execution of an atomic sequence, yet allow for inlining. The difficulty in using designated sequences lies in finding code sequences that do not occur outside of atomic sequences. Also, matching templates against the code in the vicinity of the saved PC can be time consuming.

The difficulty in finding designated sequences is ameliorated by the fact that with rollforward, the matching does not need to be perfect. It is acceptable to mistakenly classify a code sequence as an atomic sequence as long as all atomic sequences are detected (i.e., false hits are tolerable, missed hits are not). No harm is done by executing a short piece of code needlessly. Of course, the probability of false hits should be small. Otherwise, the overheads due to rollforward could negate the advantage of software-based mutual exclusion. Note that the same does not hold for rollback, where the interrupted atomic sequence is *restarted*. Restarting a code sequence that in reality is not a naturally restartable sequence may lead to unpredictable results. In this sense, designated sequences can be used more readily with rollforward than with rollback.

2.1.2 Static Registration

The second approach is static registration. At program startup time, all atomic sequences are registered with the interrupt system by specifying their starting addresses and lengths. As with designated sequences, there is no registration overhead once the program is initialized. However, testing whether an interrupted process was executing inside an atomic sequence can be rather expensive if many atomic sequences are registered. Proliferation on the number of atomic sequences can be avoided by having only a single atomic sequence (such as test-and-set) which is then used to build higher level synchronization objects. Limiting the number of atomic sequences unfortunately makes it essentially impossible to inline atomic sequences. Even though there is no direct runtime cost per atomic sequence, a hidden cost of a function call per atomic sequence has to be paid. Also, limiting the number of atomic sequences is detrimental to the motivation for using rollforward—the desire for the flexibility it promises is the very reason for our interest.

2.1.3 Dynamic Registration

A third approach is dynamic registration. A process notifies the interrupt system of an atomic sequence just before entering it, executes it, and finally cancels the registration. This has the disadvantage of incurring an overhead every time an atomic sequence is executed. On the positive side, it combines the advantages of designated sequences and static registration. Inlining poses no problem. The extent of an atomic sequence can be computed at run time, so even late forms of inlining, such as code synthesis [Mas92], work readily. Checking whether an atomic sequence was interrupted is efficient as well. It involves no more than two comparisons: one to determine whether an atomic sequence is registered and a second to check whether the sequence has finished execution already. (Depending on implementation details, the second check may not be necessary.)

2.1.4 Hybrid Registration

A fourth approach is what we call hybrid registration. It is a combination of designated sequences and dynamic registration. The hybrid scheme registers an atomic action just like the dynamic approach. However, after executing an atomic sequence, the registration is not cancelled explicitly. Instead, a designated sequence is used to delineate the end of the sequence. If a process is interrupted just a few instructions before reaching the designated sequence, the interrupt handler will determine that it interrupted an atomic sequence and rollforward the remaining instructions. Just as with (imperfect) designated sequences, there is a potential for false hits. However, we observed already that this is harmless as long as the sequence rolled forward terminates quickly.

The advantage of this scheme is twofold: not having to cancel registrations explicitly saves at least one instruction per atomic sequence, and registrations can be understood as hints. Extraneous “registrations” due to optimizations are tolerable as long as they occur infrequently. Not explicitly cancelling a registration is just one optimization made possible by the hint nature of registrations. As we will show in Section 3, exploiting this property can lead to a very efficient registration scheme that does not involve any loads or stores to memory.

2.2 Rolling Forward Atomic Sequences

The second dimension involves how the interrupt handler regains control of the processor once it has reached the end of the atomic sequence. We consider four possibilities.

2.2.1 Code Rewriting

An obvious solution is to rewrite the interrupted code by temporarily replacing the first instruction following the atomic sequence with a jump to the interrupt handler. A nice property of code rewriting is that it has no overheads except in the rare case of an interrupt interfering with an atomic sequence. It requires that instructions are located in a writable segment of the address space. After modifying the code, coherency has to be established between data space (where the new code is written to) and instruction space (where the new code will be executed from). Older processors that do not employ separate instruction and data caches guarantee this automatically. On many modern processors, however, this requires explicit cache flushing, which can be costly.

2.2.2 Cloning

In [ABLL91], a technique was proposed that avoids overheads per atomic sequence without requiring write access to the code segment. The idea is to clone every atomic sequence. The original copy is left unmodified while the cloned copy ends with an instruction that relinquishes control back to the interrupt handler. Given the short size of atomic sequences, we do not believe that code growth due to cloning would be a serious drawback. However, a difficulty with this scheme is the need for an efficient mechanism to map a PC in the original atomic sequence into the corresponding PC in the cloned copy of the atomic sequence. The interrupt handler has to perform this mapping in order to locate the right cloned code given a PC pointing to any instruction in the interrupted sequence. This mapping problem is aggravated if atomic sequences can be inlined.

A more subtle problem is that cloned code may be located at an address different from the original one. While it is possible to write code that produces results that are a function of the address at which it is located at we do not expect this to be a real problem for atomic sequences. More realistically, moving code might necessitate small changes in the code in order to preserve overall semantics. For example, an address constant may suddenly no longer fit into an instruction field. In the cloned sequence, such an instruction would have to be replaced by two or more instructions. Clearly, this could make the mapping problem even more difficult.

2.2.3 Computed Jumps

Computed jumps provide an alternative solution. With this approach, every atomic sequence ends with a jump to an address stored in some variable, say *dest*. Before the atomic sequence is entered, the address of the instruction immediately following the atomic sequence is stored in *dest*. Thus, if the atomic sequence executes without interruption, *dest* points to the instruction immediately after the jump and the jump acts as a "no operation" (NOP). Conversely, if the atomic sequence is interrupted, the interrupt handler overwrites *dest* with the address of the instruction where it wants to resume execution once the atomic sequence has finished. This scheme should work on almost any imaginable system. While widely and easily applicable, it has the disadvantage of adding a jump to every atomic sequence.

2.2.4 Controlled Faults

Sometimes it is more convenient for an interrupt handler to regain control via a fault (trap) instead of a jump. This is particularly useful if the interrupt handler executes at a higher privilege level than the interrupted code. Just like with computed jumps, an instruction is placed at the end of the atomic sequence that normally acts as a NOP. If properly chosen, an interrupt handler can then changes the state of the system such that this "NOP" causes a fault during rollforward. For example, in a system with memory protection, a dummy read from a special page could be performed. Normally, processes would have read access to that page. But before performing a rollforward, the interrupt handler would remove read access from that page. When leaving the atomic sequence, the CPU will attempt to read from that special page and cause an access violation fault. The access fault handler can then check whether a rollforward is in progress, and if so, pass control back to the interrupt handler. Other kinds of faults that are easily exploited for this purpose include unaligned memory access faults and division by zero faults. The only condition on the kinds of faults that can be used for this purpose is that it must be possible to resume a process after taking such a fault. Any precise fault guarantees this [HP90a].

2.3 Difficulties in Realizing Rollforward

There are two major difficulties in realizing rollforward. The first is how to ensure that a rollforward does not delay an interrupt excessively. The second is how to deal with faults induced by a rollforward. We now consider each issue in turn.

2.3.1 Limiting the Duration of a Rollforward

So far, we implicitly assumed that atomic sequences terminate quickly; i.e., they terminate within a given deadline or, at least, do not lead to endless looping. If the interrupt handler can trust in the code being rolled forward, this is generally not a problem because atomic sequences will be short. However, if there is a potential for malicious code being rolled forward, more care has to be taken. In the worst case, blindly rolling forward any "atomic sequence" can cause the system to lose interrupts in an unbounded fashion.

This could be handled by punishing such malicious code. For example, if an "atomic sequence" is found to execute for longer than, say, one timer interrupt period, the corresponding address space could be terminated. A less drastic method is possible in systems that support timer interrupts with very fine granularity. A "watchdog" timer could be started before rolling forward the atomic sequence. If the atomic sequence has not finished by the time the watchdog timer expires, the system simply resumes interrupt processing and therefore does not guarantee atomic execution of the malicious "atomic sequence." Yet another solution is to inspect the code before executing it. This is feasible as long as the inspected code is not self-modifying (e.g., it is in a page that has execute, but not write permission). Note that this restriction does not preclude the use of run-time generated code. Another quite reasonable restriction could be to disallow any backward jumps or subroutine calls in atomic sequences. With these restrictions in place, checking that a given sequence of instructions will terminate has a time complexity that is linear in the number of instructions. Fortunately, many recent CPUs have regular instruction encodings such that it is efficient to analyze code prior to its execution. In some cases it may also be possible to perform this check earlier than at interrupt time. It could be as early as compilation time or as late as program startup time.

Note that the above considerations also apply if the registration technique has a potential for false hits. With false hits, arbitrary code sequences may be mistakenly identified as atomic sequences. It is therefore necessary to check the safety of the code before initiating a rollforward, even if there is trust.

2.3.2 Faults During Rollforward

The second major issue in using rollforward is that it may cause unexpected faults. Clearly, it would be unacceptable to delay interrupt servicing excessively because a rollforward caused a page-fault. Similarly, it might be impossible to rollforward an atomic sequence if it deterministically causes, for example, a division by zero fault. We therefore have to impose the rule that atomicity of atomic sequences is guaranteed only for sequences that execute fault-free. Except for page faults, guaranteeing fault-free execution is not very difficult in practice. Nevertheless, it should be pointed out that there can be rather subtle sources of faults. For example, many architectures define a few instructions that are subsettable. Implementations may choose to emulate such instructions in software to reduce hardware cost. Unless the emulation is uninterruptible, atomic sequences should not contain such instructions.

We emphasize that page faults are often transparent. For a group of processes, page faults are only detectable if one process might execute while another is suspended on a (shared) page miss. For example, in a traditional Unix process sharing data with signal handlers, page faults are fully transparent. On the other hand, if several Unix processes cooperate via shared memory, page faults are not transparent. If page faults are not transparent, they should be treated just like interrupts. That is, a page fault in an atomic sequence will cause a rollforward. As a rollforward can cause further page faults, the rollforward mechanism has to be re-entrant. It is important to keep this in mind while reading the rest of this section.

Page faults can be either instruction or data page faults. Instruction page faults do not pose a problem as long as none of the atomic sequences cross page boundaries. A page fault might occur right before starting the atomic sequence, but once execution started, the page is guaranteed to stay resident up to the next pre-emption point, which in the assumed environment, must be an interrupt. A similar trick can be applied to avoid data-page faults. Rollforward is feasible as long as a given shared data object is fully contained in a single page. A process executing in an atomic sequence might be pre-empted due to a page fault to a shared page. While being blocked waiting for the data page, other processes might attempt to access the same page as well. This is not a problem because none of these processes could have read or modified any shared state yet (otherwise they would have caused the page fault). When the shared page finally arrives, the blocked processes can be resumed in any order.

While the one-page restriction is quite acceptable for the code of atomic sequences, it poses a serious limitation for shared data objects. A more practical solution might be to avoid page faults in the first place. This is possible if shared memory can be pinned. Pinned memory is never paged out by the virtual memory system and shared memory would therefore always stay resident in physical memory. This is certainly a realistic solution for the parts of the kernel that interact with interrupt handlers. There, page-faults must be avoided anyway, so this solution comes at no extra cost. However, it is undesirable to allow user processes to pin down large amounts of memory.

Fortunately, other solutions exist that work well for user processes. For example, it is possible to add a "gate" to every shared object. If the gate is open, atomic sequences accessing that object can be executed. If it is closed, attempting to execute such an atomic sequence causes the process to be suspended until the gate is re-opened again. This idea can be used in a straight-forward manner to deal with page faults: a gate is closed whenever a process executing an atomic sequence causes a page fault on the gate's shared object. It remains closed until the fault-causing atomic sequence has terminated. This ensures proper operation in the case of an atomic sequence causing multiple page faults. In essence, processes may execute independently, as long as they do not attempt to access a shared object for which a page fault is pending. We believe this is a reasonable tradeoff between keeping overheads low and minimizing the time processes are blocked due to page-faults of *other* processes. A gate can be implemented in many different ways. One possibility would be to assign each shared object to a distinct memory segment (or set of pages, in a paged memory system). Closing the gate could then be implemented as turning off read and write access to the object's memory segment (set of pages). Clearly, changing segment protection is a rather heavy-weight mechanism. But it should be observed that these overheads arise only in response to page faults caused by atomic sequences. In particular, in the normal case of no page faults, atomic sequences would not experience any additional overhead.

3 Implementation

This section describes three specific software mutual exclusion schemes using rollforward. Given the large design space, the limitation to only three implementations is somewhat arbitrary. However, the three implementations have unique features that make them worthwhile to highlight: the first uses the dynamic registration scheme and adds a computed jump to every atomic sequence to allow an interrupt handler to regain control of the CPU after performing a rollforward. It is the most obvious implementation, and is therefore easy to explain. It also serves as a benchmark against which the other, more involved schemes, can be compared. The second is a slight variation of the first—instead of a jump, a fault is used to provide the interrupt handler with the means of regaining control. This might be an appropriate choice in situations where atomic sequences execute at a lower privilege level than the interrupt handler. The third implementation uses the hybrid registration scheme and, like the first one, provides a jump to the interrupt handler as the means to regain control after a rollforward. This implementation incurs the lowest overheads per atomic sequence, and has the potential to outperform even the best hardware-based schemes.

3.1 Dynamic Registration Scheme With Jump (Dyn/Jump)

The most straight-forward dynamic registration scheme is shown below. First, variable *destAddr* is set to the address of the instruction designated by label theEnd. To indicate that the process is executing an atomic sequence, *inAS* is set to true. Then, the code of the actual atomic sequence is executed. Afterwards, *inAS* is reset to false and a jump to the address stored in *destAddr* is performed. In the normal case, this will jump to label theEnd and therefore act as a

NOP. Notice that this implementation depends on the fact that word reads and writes execute atomically.

```
        $destAddr \leftarrow addressOf$(theEnd)
        $inAS \leftarrow$ TRUE
        $\langle$atomic sequence $\ldots\rangle$
        $inAS \leftarrow$ FALSE
        jump $destAddr$
theEnd:
```

If every atomic sequence is wrapped up in this manner, the state of variable *inAS* is an accurate indication of whether a process is executing in an atomic sequence. An interrupt handler therefore has to read a single word to check whether a rollforward is necessary. If so, the interrupt handler changes *destAddr* to point to an instruction in its own code, restores the state of the interrupted process, and jumps to the saved PC. This causes the interrupted atomic sequence to be executed to the end. The computed jump then transfers control back to the interrupt handler where it reestablishes its own state and continues with the actual interrupt processing.

On a DEC Alpha, this produces the following code for an atomic sequence that increments variable *sharedCounter*. As can be seen, an overhead of five instructions has to be paid per atomic sequence: two load address instructions, two stores, and a jump instruction (overhead instructions are marked with an asterisk).

```
   * lda  r4, inAS          # load address of inAS
   * lda  r1, theEnd        # load address of theEnd into r1
   * stl  zero, (r4)        # inAS <- TRUE (0 = TRUE)
     lda  r3, sharedCounter # load address of sharedCounter
     ldl  r2, (r3)          # load value of sharedCounter
     addl r2, 1, r2         # increment counter
     stl  r2, (r3)          # store back new value
   * stl  r1, (r4)          # reset inAS to FALSE (not 0 = FALSE)
   * jmp  (r1)              # jump to address stored in r1
theEnd:
```

3.2 Dynamic Registration Scheme With Fault (Dyn/Fault)

The pseudo-code for the second implementation is given below. An asterisk denotes pointer dereferencing. The instruction that potentially causes a fault is the dereferencing of pointer variable *falseOrFault*. Normally, this variable points to a memory location holding value FALSE. Thus, in the absence of interference, the code sequence will reset *inAS* to false after executing the atomic sequence. However, before performing a rollforward, an interrupt handler changes *falseOrFault* to an unaligned address. This fault is intercepted by the interrupt handler and thus allows it to regain control after a rollforward. Obviously, this implementation only works on machines that require aligned memory accesses, as is the case for most RISC processors. After regaining control, the interrupt handler has to change the state of the faulting process such that it will be able to resume execution. This is most easily done by resetting *inAS* to false and advancing the saved PC to the address stored in *destAddr* (i.e., by skipping the faulting instruction).

```
        $destAddr \leftarrow addressOf$(theEnd)
        $inAS \leftarrow$ TRUE
        $\langle$atomic sequence $\ldots\rangle$
theEnd: $inAS \leftarrow *falseOrFault$
```

This implementation weakens the semantics of *inAS* in a rather subtle way: the fault-causing instruction is executed *before inAS* is reset to false. This is done to keep the number of overhead instructions low and to avert the danger of a compiler optimizing away the fault causing instruction. Unfortunately, this means that an interrupt handler has to be more careful before initiating a rollforward. As before, it first checks whether *inAS* is true. In addition, it has to check whether the saved PC of the interrupted process points to an instruction before label theEnd. Only if both conditions hold should a rollforward be initiated. Otherwise, it could happen that a process is interrupted after dereferencing *falseOrFault* but before storing false to *inAS*. If only *inAS* were used, the interrupt handler would initiate a rollforward. However, the fault causing instruction has been executed already and the interrupt handler would not be able to regain control of the CPU; the interrupt would be lost. For the second check to work properly, the atomic sequence must not consist of any instructions beyond label theEnd. In practice, this means that function calls cannot be permitted in atomic sequences.

The above pseudo-code can be optimized in several ways. For example, *destAddr* and *inAS* can be merged into a single variable if there is a distinguished address that cannot be a valid instruction address (e.g., zero or an odd address). For brevity, we omit the assembly code produced when applying this optimization. On the Alpha, the resulting code has six overhead instructions: four loads and two stores.

3.3 Hybrid Registration Scheme With Jump (Hyb/Jump)

The pseudo-code for the hybrid registration scheme is given below:

```
              $destAddr \leftarrow addressOf$(theEnd)
              $inAS \leftarrow$ TRUE
              ⟨atomic sequence . . .⟩
              jump $destAddr$
  theEnd:
```

It is identical to the one for the Dyn/Jump implementation, except that the final assignment “*inAS*←FALSE” is missing. This is possible because registration is only a hint. The above code guarantees that whenever executing an atomic sequence, *inAS* will be true and *destAddr* will point to the end of the atomic sequence. The opposite is not necessarily true. An interrupt handler first checks whether *inAS* is true. If it is, the interrupted process *might* have been executing an atomic sequence. For this to be the case, the saved PC of the process has to point to an instruction before label theEnd and *destAddr* must point to an instruction preceded by a jump instruction. If these conditions hold and the code between the saved PC and theEnd is “safe” to execute (i.e., does not involve any endless loops etc.), the interrupt handler will rollforward the interrupted code sequence.

As registration is only a hint, interesting optimizations can be applied. Like before, *destAddr* and *inAS* can be merged into a single variable. In addition, it is now possible to allocate this merged variable into some register that is well-known to the interrupt handler. The above pseudo-code can therefore be implemented without any memory load or store operations and yet the well-known register has a special purpose only while executing in an atomic sequence. Using a register also has the added benefit that code following the atomic sequence is likely to quickly overwrite the value in that register. This has the effect of automatically cancelling the registration of the previous atomic sequence. For this purpose, it is best if the register is frequently used (such as a compiler temporary) and—to reduce the probability of false hits—if instruction addresses have values that are unlikely to occur in ordinary computations.

On the Alpha, the example to increment a shared counter translates to the code shown below. Notice that this code has only two instructions beyond what is needed for the atomic sequence: a load address and a jump (again, overhead instructions are marked with an asterisk). That is, *no memory accesses* are necessary to implement mutual exclusion.[2] This is probably close to the minimum instruction count overhead that any dynamic registration scheme will ever have. In essence, the first load corresponds to a “disable interrupts” instruction and the jump corresponds to an “enable interrupts” instruction.

```
          * lda  r1, theEnd        # load address of theEnd into r1
            lda  r3, sharedCounter # load address of sharedCounter
            ldl  r2, (r3)          # load value of sharedCounter
            addl r2, 1, r2         # increment counter
            stl  r2, (r3)          # store back new value
          * jmp  (r1)              # jump to address stored in r1
theEnd:
```

Note that implementations that use a jump also admit the implementation of interrupt priority levels in a straightforward and efficient manner. One could encode the priority level at which the sequence executes in an integer that is placed between jump instruction and label theEnd. An interrupt handler can check this word and perform a rollforward only if the interrupt priority is not higher than that integer. It is interesting to observe that this works properly even in the presence of false hits leading to extraneous rollforwards. The integer following the jump simply *decreases* the probability of false hits. Except for a slight increase in code size and the associated cache effects, this is a zero cost extension.

3.4 Summary

The three implementations presented above can be summarized according to efficiency and the flexibility they afford. For completeness, we also include Bershad et al.’s rollback technique. The table is ordered according to increasing flexibility. In general, the more flexible a solution, the higher the overheads it imposes. The only exception to this rule is Dyn/Fault, which is more restrictive, yet has one overhead instruction more than Dyn/Jump.

Technique:	**Restrictions on Atomic Sequences:**
Rollback:	At most *one store* to shared memory.
Hyb/Jump:	No backward jumps or function calls and limit on maximum code size.
Dyn/Fault:	No function calls.
Dyn/Jump:	No limitations.

4 Experimental Results

This section presents the performance of the three implementations introduced in the previous section. For comparison, we also report on the performance of various hardware based schemes and of *sigprocmask*, the Unix user-level equivalent of disabling interrupts. There are three performance parameters associated with a software-based mutual exclusion scheme: (1) overhead per atomic sequence, (2) overhead per interrupt, and (3) overhead per rollforward. In practice, the overhead per interrupt is minimal (typically an integer range check) and insignificant compared to the cost of fielding an interrupt and switching context to the interrupt handler. We therefore do not report on this overhead.

[2] On the Alpha, the load address instruction is usually translated into a load instruction, but this is mainly due to a limitation of the OSF/1 assembler. There are other, potentially more efficient, schemes to load a 64 bit constant.

4.1 Overhead Per Atomic Sequence

Table 1 presents the overhead, measured in CPU cycles, that occurs with every execution of an atomic sequence. It includes the execution time of everything that has to be done in addition to the actual atomic sequence in order to guarantee atomicity. For a hardware-based scheme, this is typically the time to disable interrupts before entering the atomic sequence and to re-enable interrupts after leaving the atomic sequence. For software-based schemes, this includes dynamic registration overheads, for example. The hardware on which we obtained these results consisted of a DEC 3000 Model 600 AXP workstation with an Alpha CPU operating at 175 MHz and an HP 9000/735 with a PA-RISC 1.1 CPU operating at 99 MHz [Sit92, HP90b]. All tests were small enough to fit in the cache and the reported results are the execution times when running in the cache (i.e., with a "warm" cache). Both machines provide timers with a resolution of a single CPU cycle. Measuring execution time via these timers adds up to 3 cycles. For consistency, we did not account for this overhead in any of the measurements. The numbers reported are the mode of the execution time histograms obtained after running each test 1000 times. In almost all the measurements, more than 995 samples had the value of the mode.

	DEC Alpha			HP PA-RISC 1.1		
Technique	NULL	LIFO	FIFO	NULL	LIFO	FIFO
sigprocmask	1682	3045	3363	1787	3578	3590
Dyn/Fault	13	27	24	12	24	27
Dyn/Jump	9	16	13	11	21	27
Hyb/Jump	6	5	6	5	8	12
DI	4	3	4	4	5	12
CIPL	4	5	6	14	24	29
splx	44	89	88	30	63	73
PALcode	$\geq$ 13	$\geq$ 13	$\geq$ 13	n/a	n/a	n/a
LL/STC	n/a	$\geq$ 118	$\geq$ 118	n/a	n/a	n/a

Table 1: Overheads of Different Atomicity Schemes in Cycles

The first test program, NULL, is an empty atomic sequence. In theory, the overheads measured with this test should be *the* constant that gets added to the execution time of any atomic sequence. However, the code implementing atomicity interacts with the code implementing the atomic sequence. For example, the former may compete with the latter for registers and/or cache memory. Thus, the effective overheads may be bigger than what is observed for an empty atomic sequence. The opposite can occur as well: an empty atomic sequences causes the atomicity code that runs before the atomic sequence and the one that runs afterwards to be executed back to back. This can result in additional CPU stalls. In this case, the overheads for an empty atomic sequence would be pessimistic. With these considerations in mind, we also timed two non-trivial test programs. They are simple as well, but realistic. Test program LIFO measures the time to add an element to a singly-linked stack and to remove that same element again. This code involves two atomic sequences. Test program FIFO measures the time to enqueue an element into an empty, singly-linked queue and to dequeue that same element. It also involves two atomic sequences. In both cases, care was taken to write the programs in such a way that the compiler's optimizer could not take any shortcuts. For these two programs, the overheads should be twice as high as those for the NULL program. When comparing the numbers reported in the NULL column with those in the LIFO and FIFO columns, it becomes apparent that this simple relationship does not hold; the interaction between the atomicity code and the actual atomic sequence is not negligible.

We measured the following mutual exclusion techniques:

sigprocmask: Using *sigprocmask* to disable signal SIGALRM before entering the atomic sequence and to restore the old signal mask after leaving the atomic sequence.

Dyn/Fault, Dyn/Jump, Hyb/Jump: See Section 3. For the Dyn/Jump and Hyb/Jump implementations, the plain version and the version encoding an interrupt priority level in the word following the jump instruction had the same execution times, so we report only one number.

DI: Disabling *all* interrupts before entering an atomic sequence and enabling *all* interrupts after leaving it; i.e., this scheme does not admit interrupt priority levels. The Alpha *architecture* does not support this. However, the implementation described in [Dig92] provides the required facilities in a chip specific fashion. These low-level facilities normally are not accessible to kernel-level software. The measurements therefore had to be performed in a small stand-alone system. For the PA-RISC, the technique was measured in a Mach kernel that was extended with the test programs.

CIPL: Changing interrupt priority level via inlined code. The same comments as for DI apply.

splx: This is the same as CIPL except that it adds the overhead of a function call. That is, the code to change the interrupt priority level is no longer inlined.

PALcode: The Alpha architecture defines a Privileged Architecture Library (PALcode). This library code is invoked via traps and executes in privileged kernel mode with interrupts turned off [Sit92]. We did not have the opportunity to implement the benchmarks as PALcode yet, but found that it takes at least 13 cycles just to invoke PALcode and return immediately from it. It is not possible to inline PALcode.

LL/STC: The Alpha architecture provides load-linked and store-conditionally instructions to implement multiprocessor-safe shared data structures. We did not implement our benchmarks using these instructions but note that a single atomic word move between two memory locations already has the high cost of 118 cycles.

The data in Table 1 shows that the overhead of *sigprocmask* is two orders of magnitude higher than that of any of the other software based mutual exclusion schemes. Even if rollforward were expensive, the software-based methods using rollforward would likely be more efficient overall. The table also indicates that the rollforward approach is highly competitive even when compared to hardware-based methods. On the Alpha, even though DI is slightly faster and CIPL is about equally fast as Hyb/Jump, it should be stressed that the former two rely on CPU and system implementation specific details that are likely to change frequently. The fastest architecturally defined hardware-based method on the Alpha is to implement each atomic sequence as a separate PALcode function. As the data shows, Hyb/Jump is at least twice as fast as PALcode. On the PA-RISC, Hyb/Jump performs even better. Only DI is slightly faster while all other hardware-based techniques are slower by at least a factor of two.

Furthermore, it should be pointed out that many OS kernels use a function call to change the interrupt priority level. As shown in row “splx,” this is rather expensive. Finally, as the last row indicates, a pair of load-linked/store-conditionally instructions has a surprisingly high cost. Considering this and the rather long list of conditions under which a store-conditionally is (almost) guaranteed to fail [Sit92], this does not appear to be an effective scheme to provide atomicity in a uniprocessor (and it probably was not meant to be so).

4.2 Overhead per Rollforward

The overhead per rollforward consists of two components: the time to check whether it is safe to do a rollforward and the time to activate and return from the rollforward (i.e., switch context to the interrupted process and regain control of the CPU). The second component is difficult to measure in a meaningful way as its cost is highly dependent on the details of an implementation. Fortunately, if the rollforward handling is placed early on in the interrupt handling, switching context to the interrupted process involves saving and restoring only a minimal amount of processor state. That is, if implemented properly, this overhead should quite small (compared to other interrupt-overheads). The first component can be measured easily and its cost should remain comparable across different implementations and systems. It is non-zero if either the interrupt system does not trust in the pre-empted code or if the mutual exclusion scheme has the potential for "false hits" (i.e., an interrupted sequence can be mistakenly identified as an atomic sequence).

A rollforward is safe if performing it will not take "too much time" (meet a certain deadline or, at least, not take forever) and if it ends with an instruction that will return control of the CPU to the interrupt handler. It is a well-known result that the halting problem for any universal language is undecidable. It is therefore necessary to restrict the code in atomic sequences such that safety can be decided efficiently. This is achieved by limiting the maximum length of the atomic sequence and imposing the rule that no branch is allowed unless it can be statically determined that it branches forward and that it does not jump outside the atomic sequence. If the mechanism to relinquish control to the interrupt handler relies on a well-known register (such as Dyn/Jump and Hyb/Jump), it is also necessary to ensure that the code sequence does not attempt to modify the well-known register. If the registration technique has the potential for false hits (e.g., Hyb/Jump), it is also necessary to disallow any uses of the well-known register. Otherwise, rolling forward code that was mistakenly identified as an atomic sequence could produce an incorrect result. Finally, to avoid security holes, the code should be rolled forward in the context of the interrupted process instead of the context of the interrupt handler.

On the Alpha, we measured the worst-case time to check the safety of a code sequence. As described in the previous paragraph, checking safety for the Hyb/Jump scheme represents the worst case, because it uses a well-known register and it has the potential for false hits. During these measurements, the maximum length of an atomic sequence was restricted to a conservative 32 instructions. For example, the atomic sequences in the above benchmark programs are all shorter than 7 instructions when using the Hyb/Jump technique.

It is important to do measurements under realistic conditions. In particular, the memory system state greatly influences the results. The DEC 3000/600 workstation has an 8 KB instruction cache, an 8 KB data cache, and a 2 MB unified secondary cache. Suppose an interrupt handler has to check the safety of the code starting at address A. Just prior to the interrupt, the CPU was executing in that address range. It is therefore likely that the instructions around A are in the instruction as well as in the secondary level cache (the latter is a superset of the primary caches). As programs typically do not load data from the address range they are executing in, it is unlikely that any of the instructions around A reside in the primary data cache. We chose to perform our measurements in exactly this scenario: the instructions to be checked are all in the secondary cache but none of them are in the data cache.

Our measurements indicate that the overhead O_R to check the safety of a sequence that is N instructions long is bounded as follows:

$$O_R \quad \leq \quad 73.375\,\mathrm{cyc} + N \cdot 25.375\,\mathrm{cyc}.$$

For example, checking the safety of a sequence that is 16 instructions long takes less than 480 cycles or, at a clock rate of 175 MHz, less than $2.7\,\mu$s .

4.3 Overall Benefit of Software Mutual Exclusion

Software mutual exclusion is an optimistic approach. It attempts to improve overall performance by optimizing the common case at the cost of the, hopefully, rare case. This implies that the optimistic approach may fail if the rare case occurs more frequently than anticipated. We have not yet had the opportunity to perform macro experiments that would provide this information. However, we can at least test the idea based on a simple execution model.

We assume an execution model in which interrupts occur at a fixed rate R_I. Similarly, atomic sequences are executed at a fixed rate R_A and have a duration of D_A cycles each. This is illustrated in Figure 1. If we assume

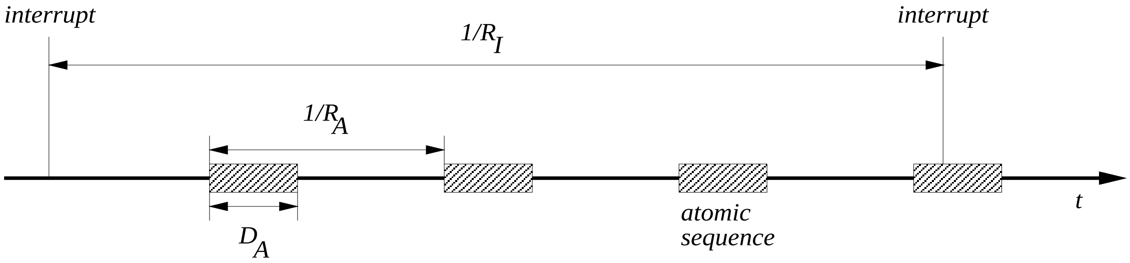

Figure 1: Model of operation.

that interrupts sample the interval $1/R_A$ randomly, the probability of an interrupt pre-empting an atomic sequence is $R_A \cdot D_A$. Suppose that the software mutual exclusion scheme has a fixed overhead of O_S cycles per atomic sequence and an overhead of O_R cycles per rollforward. Note that we ignore the *per interrupt* overhead. This cost is so small compared to the cost of fielding an interrupt and switching context that we believe this is justified (on the Alpha, this overhead is roughly 20 instructions). Also, O_R does not include the time spent executing in the atomic sequence because a hardware-based scheme would have delayed the interrupt by that much. On the other hand, the only overhead associated with a traditional approach, such as disabling interrupts, is the one that arises per atomic sequence. We denote this by O_H. Given these definitions, it is easy to derive an equation for the maximum interrupt rate below which the software-based approach is faster:

$$R_I \quad < \quad \frac{O_H - O_S}{D_A \cdot O_R}.$$

Note that the right hand side is independent of the rate R_A at which atomic sequences are executed. As R_A increases, the probability of interference—and with it the overhead—increases. However, because atomic sequences are executed more frequently, the benefit of using a software-based mutual exclusion scheme increases as well. Overall, the increase in overhead is balanced by an equal increase in benefit and the cross-over point remains the same.

For concreteness, we work out an example for the Alpha workstation using the Hyb/Jump implementation. As reported in Table 1, O_S is roughly 6 cycles when using the Hyb/Jump technique on the Alpha. Assume that the maximum length of an atomic sequence is 32 instructions. As the Alpha can issue up to two instructions per cycle, we conservatively assume an average execution rate of 1 instruction per cycle giving D_A as 32 cycles. Of the 32 instructions, we assume that, on average, half of them have to be rolled forward. According to the measurements presented in the previous subsection, it takes at most 480 cycles to check the safety of a code sequence that is 16 instructions long. Conservatively, we set $O_R = 480$ cycles. On the other side, the best architecturally defined hardware-based scheme on the Alpha is PALcode [Sit92]. As shown in Table 1, the overhead per PALcode invocation is at least 13 cycles. Optimistically, O_H is 13 cycles. With these values, we have:

$$R_I \quad < \quad \frac{O_H - O_S}{D_A \cdot O_R} = \frac{13\,\mathrm{cyc} - 6\,\mathrm{cyc}}{32\,\mathrm{cyc} \cdot 480\,\mathrm{cyc}} \approx 456 \times 10^{-6}\,\mathrm{cyc}^{-1}.$$

At the machine's clock-rate of 175 MHz, this corresponds to an interrupt rate of approximately 80 kHz. Under the given model, the software-based method therefore outperforms the hardware-based scheme with any but the most severe interrupt loads.

5 Related Work

As mentioned in the introduction, Bershad et al. propose a software-based technique for mutual exclusion on uniprocessors [BRE92]. Their approach is based on rollback instead of rollforward. It also differs from our approach in several more specific ways. First, their solution is limited to atomic sequences that contain only a single store to shared memory, whereas our approach permits multiple stores. It is often more convenient, and in the end more efficient, to implement shared data-structures via sequences that require multiple stores to shared memory (e.g., see [Mas92]).

Second, our approach supports efficient lock-free solutions, whereas the earlier solution was designed for a lock-based scenario. We are interested in lock-free data-structures for four reasons: they can be used to share data with interrupt handlers, there is no potential for deadlock, there is no locking overhead, and non-updating operations can proceed and complete concurrently. This is in contrast to the motivation for using lock-free data-structures on multiprocessors. There, as argued by Herlihy [Her93], the interest is founded mainly on the fault-tolerance properties of lock-free data-structures.

Third, our hybrid registration scheme allows inlining of atomic sequences, whereas their static registration scheme severely restricts the number of atomic sequences. The bottom line is that rollforward makes software-based mutual exclusion a much more widely applicable technique.

Using rollforward to guarantee atomicity is not a new idea. For example, it has been used successfully in the VAX runtime system of the Trellis/Owl language [MK87]. There, rollforward was implemented by emulating the instructions from the point of interruption to the end of the atomic sequence. We do not believe that emulation is feasible in the applications we envision. The complexity and overheads involved would be simply too big. Another shortcoming is that the Trellis/Owl work does not address any of the operating system issues raised by software mutual exclusion. Being a language system, it is unclear whether the same techniques could be generalized to other languages or supported in a language-independent manner.

Rollforward is also mentioned in [Ber93]. That work appears to dismiss rollforward as a practical solution for two reasons: (a) execution of code on behalf of another thread and (b) page-faults. It is true that executing code on behalf of another thread is difficult. However, we contend that it is feasible and efficient to let rollforward occur in the preempted thread's context simply by ensuring (e.g., via code-inspection) that it will relinquish control of the CPU in a timely manner. We also discussed how page-faults can be handled. In particular, restricting code and data to be (individually) contained in a single page is practical for the problem studied in [Ber93]. We would also like to point out that with rollback it would be difficult to implement primitives that need to update multiple words atomically. For example, a simple compare-and-swap2 operation that compares and conditionally updates two words is much easier to implement with rollforward. In [Mas92], Massalin found this primitive to greatly simplify the implementation of certain lock-free data-structures. There is good reason to belief that there are other multi-word operations that would be useful for constructing lock-free data-structures. Rollforward gives the flexibility to do so.

Another system that employed rollforward is Scheduler Activations [ABLL91]. Unlike Trellis/Owl it does not use rollforward to achieve mutual exclusion. Instead, it is used to avoid deadlock. Scheduler Activations use lock-based synchronization. For performance and liveness reasons, it is undesirable to suspend a process while it is holding a lock. One could either *avoid* this situation or *recover* from it. Scheduler Activations take the latter approach. That is, whenever a process is suspended while holding a lock, it is rolled forward to the point where it releases the lock. Even though the reason for rollforward is very different from achieving mutual exclusion, the implementation the authors describe is not limited to Scheduler Activations. Unfortunately, the paper gives only a superficial treatment of the issues involved with rollforward. It does not give any indications as to the cost or limitations of the technique.

Finally, [SCB93] presents a technique called *optimistic interrupt protection*. The idea is to use delayed (lazy) evaluation to reduce the number of times that expensive interrupt level changing instructions have to be invoked. What is interesting is that the implementation of this technique is one particular realization of the more general rollforward technique. Due to the entirely different focus, that work makes no attempt to generalize the technique into a synchronization mechanism that is useful at the user as well as kernel-level. As a matter of fact, the paper does not make a connection between optimistic interrupt protection and rollforward.

6 Concluding Remarks

This paper describes a software-based approach to mutual exclusion on a uniprocessor. The approach uses rollforward rather than rollback to recover when an atomic sequence is interrupted. Rollforward is more flexible than schemes based on rollback in that it allows multiple stores. Atomic sequences with multiple stores can be used to construct powerful "primitives" which, in turn, can be used to simplify the implementation of lock-free data-structures. Micro-experiments show our approach to be as efficient as, and in many cases more efficient than, hardware-based mutual exclusion mechanisms. We conclude this paper by making some observations about the mechanism.

First, we described that page faults present a more serious problem when using rollforward instead of rollback. With rollback, page faults are a problem only if an interrupt handler needs to inspect the interrupted code in order to determine whether it interrupted an atomic sequence [BRE92]. With rollforward, there is always a potential for a page fault during rollforward. On the other hand, it is much easier to use designated sequences with rollforward because false hits are tolerable. The Hyb/Jump techniques fully exploits this fact.

Second, even if a hardware-based mechanism is more efficient, a software-based method may be preferable because it is equally applicable in unprivileged user-mode as in kernel-mode. This is particularly useful for software that must execute efficiently in either environment, for example, OS servers that can be executed in either user space or the kernel.

Third, the ability to inline atomic sequences is important. Many modern machines use direct-mapped caches. Thus, every branch to a subroutine has the potential to evict cache lines due to collisions. For example, we reported the PALcode invocation overhead on the Alpha as 13 cycles (see Section 4). However, in practice, this overhead is often in the 30 cycle range due to cache effects (PALcode is located at an operating system fixed address and therefore not inlinable). As the gap between memory system and CPU performance continues to grow [HP90a, Pri94], techniques that yield good cache performance will become ever more important. Software-based methods that use designated sequences, a dynamic, or a hybrid scheme to register atomic sequences can all be inlined easily.

Fourth, our experience is that rollforward is very practical. For example, all of our methods can be implemented via a simple C pre-processor macro with the GNU C compiler. These macros take a single argument: a sequence of instructions that should be executed atomically. As each macro presents the same interface, software implementing atomic sequences can be written independently of which mutual exclusion scheme is used in the end. GNU C is powerful enough to allow writing the macros in a manner that is safe even when code optimizations are applied. This is a crucial feature. Without it, inlining of atomic sequences could not be done safely. Another demonstration of practicality is that it is surprisingly easy to implement rollforward recovery in the context of Unix signal processing. We used a slight variation of the Dyn/Jump technique to implement *sigprocmask* and its associated functions. Except for a few details, the implementation provides Unix semantics at a fraction of the cost of the traditional implementation involving a system call.[3]

Fifth, we enumerate a few applications of software-based mutual exclusion techniques using rollforward. As the technique is non-blocking, it can be used to share data between interrupt handlers and normal kernel mode or even user mode code. Also, it is directly applicable if page-faults are impossible (e.g., non-pageable part of an OS kernel) or if page-faults are transparent (e.g., within single Unix process). As discussed in Section 2, there are a number of ways to make rollforward work even if page-faults are not transparent. Once again, each solution provides a different

[3] The limitation of the user-level implementation of *sigprocmask* is that signal stacks cannot be handled properly. Also, for simplicity, no attempt was made to emulate the semantics for signals that can stop a process (e.g., SIGTSTP, SIGTTOU, and SIGTTIN).

tradeoff between flexibility and performance. Finally, the application where this technique can be applied most readily and extract the largest performance benefit is in Unix processes that frequently execute code that needs to be protected from asynchronously executing signal handlers.

Finally, there are countless low-level issues that arise when implementing an interrupt handler that supports software mutual exclusion. We conclude by mentioning just one. The question is whether a processor should disable *all* interrupts when dispatching to a handler or just *some* of them. It is intuitive that the code that checks for and initiates a rollforward is itself an atomic sequence. If a processor disables all interrupts before dispatching to this code, there is no problem. However, if only some interrupts are disabled, it is necessary to guarantee the atomicity via software. An interrupt handler could disable all remaining interrupts as quickly as possible, but this still leaves a small window during which an interrupt handler could be pre-empted. An interrupt-handler therefore has to check whether it interrupted another handler and, if so, take the appropriate recovery action. While not extremely difficult, this is clearly more complicated than if the number of outstanding interrupts were limited to one. For software mutual exclusion, it is therefore desirable that the processor invokes an interrupt handler with *all* interrupts disabled. This is the case for PA-RISC 1.1 but unfortunately not for an Alpha using either the VMS or OSF/1 PALcode. However, because PALcode is easy to replace—it is copied from non-volatile memory to normal RAM at boottime—this deficiency is easily corrected. As a matter of fact, it is even possible to move the software mutual exclusion part of an interrupt handler entirely into PALcode. From an operating system perspective, it would then appear as if software mutual exclusion were a part of the Alpha architecture.

References

[ABLL91] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Symposium on Operating System Principles*, 1991.

[Ber93] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–273, May 1993.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 223–233. ACM, October 1992.

[Dig92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*. Digital Press, Maynard, Massachusetts, first edition, October 1992. Order number EC-N0079-72.

[GS92] Ruth E. Goldenberg and Saro Saravanan. *VMS for Alpha Platforms—Internals and Data Structures*, volume 1. DEC Press, Burlington, Massachusetts, prelimenary edition, 1992. Order number EY-L466E-P1.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trancsactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HP90a] John L. Hennessey and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., Palo Alto, 1990.

[HP90b] Hewlett-Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*. Hewlett-Packard, Cupertino, California, first edition, November 1990. Part number 09740-90039.

[LMKQ88] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1988.

[Mas92] Henry Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, New York, NY 10027, September 1992.

[MK87] J. Eliot B. Moss and Walter H. Kohler. Concurrency features for the Trellis/Owl language. In *European Conference on Object-Oriented Programming*, number 276 in Lecture Notes in Computer Science, pages 171–180. Springer-Verlag, 1987.

[Pri94] Betty Prince. Memory in the fast lane. *IEEE Spectrum*, 31(2):38–41, February 1994.

[SCB93] Daniel Stodolsky, J. Bradley Chen, and Brian N. Bershad. Fast interrupt priority management in operating system kernels. In *Proceedings of the Second Usenix Workshop on Microkernels and Other Kernel Architectures*, pages 105–110. Usenix, September 1993.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Burlington, Massachusetts, 1992. Order number EY-L520E-DP.