Implementing Atomic Sequences on Uniprocessors Using Rollforward

David Mosberger, Peter Druschel,* and Larry L. Peterson †

Department of Computer Science
University of Arizona
Tucson, AZ 85721

{davidm,druschel,llp}@cs.arizona.edu

Summary

This article presents a software-only solution to the synchronization problem for uniprocessors. The idea is to execute atomic sequences without any hardware protection, and in the rare case of pre-emption, to roll the sequence forward to the end, thereby preserving atomicity. One of the proposed implementations protects atomic sequences *without any memory-accesses*. This is significant as it enables execution at CPU-speeds, rather than memory-speeds. The benefit of this method increases with the frequency at which atomic sequences are executed. It therefore encourages the building of systems with fine-grained synchronization. This has the additional advantage of reducing average latency. Experiments demonstrate that this technique has the potential to outperform even the best hardware mechanisms. The main contribution of this article is to discuss operating-system related issues of rollforward and to demonstrate its practicality, both in terms of flexibility and performance.

1 Introduction

Atomic sequences—a sequence of instructions that needs to execute without interference—are fundamental to concurrent programs, including operating systems.[1] For this reason, processors generally provide synchronization primitives, such as `test-and-set` or `compare-and-swap` in their instruction-set architecture [2, 3, 4, 5, 6, 7, 8]. Due to the trend of implementing processors that are suitable for use in shared-memory multiprocessors, these primitives have become quite expensive. For example, the Alpha architecture [7] provides a multiprocessor-safe load-linked/store-conditionally instruction that, on some uniprocessors, requires

*Author's current address: Department of Computer Science, Rice University, P.O. Box 1892, Houston, TX 77251, USA

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

[1]This article uses the term "atomic sequence" rather than "critical section" as the latter is one particular technique that uses mutual exclusion to guarantee atomic execution. It is by no means to only technique to implement atomic sequences [1].

over 100 cycles to move a 64 bit quantity atomically between two memory locations. While it is certainly possible to reduce this overhead, this example does illustrate that hardware synchronization primitives are designed and optimized for the multiprocessor case.

On a uniprocessor, where interrupts are the only source of interference, it is possible to implement atomic sequences by disabling interrupts. 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 [9].

Because of the limitations of these two hardware-based approaches, Bershad, Redell, and Ellis have proposed a software-only implementation of mutual exclusion on uniprocessors [10]. However, the solution is applicable only to simple atomic sequences that involve at most one write to shared memory. While this is sufficient to implement synchronization primitives powerful enough to construct higher-level synchronization objects, the approach is still limited. First, when used to build lock-based synchronization mechanisms, it cannot be used for data-structures that are shared with interrupt handlers, as this code cannot risk blocking. (In general, lock-based solutions introduce the problem 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 [11, 12].

To alleviate these limitations, we propose the use of *rollforward*. When an atomic sequence is pre-empted, the system arranges for the sequence to be executed to the end before resuming the new task. This approach puts few restrictions on the code that constitutes atomic sequences. In particular, the number of writes is essentially unlimited. Consequently, it is well-suited for constructing efficient lock-free data structures. Also, 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 [13] or VMS Asynchronous System Traps [14]). Even though the 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 [15].

While the idea of rollforward is straight-forward, there are several interesting issues. For example, we found that it is possible to trade flexibility for performance. As different applications require different performance/flexibility tradeoffs, no single solution will suite all of them. Also, it is not immediately clear what kind of events ought to be interpreted as pre-emption points. On a uniprocessor, the ultimate source of pre-emption is the interrupt. However, in many cases it is possible to work with higher-level, less frequent events, such as Unix signal-delivery. Finally, in some applications, it may also be necessary to address issues like interactions with the memory system, multiple address spaces, and multiple trust domains. This indicates that rollforward has a rather rich design-space. The main contribution of this article is to explore this space and to quantify the performance of specific design-points.

The rest of the article is organized as follows. Section 2 is a general treatment of the rollforward technique.

Section 3 presents three concrete rollforward mechanisms. The performance of these mechanisms is reported in Section 4. For comparison, it also reports on the performance of other hardware and software-based synchronization primitives. Section 6 offers some concluding remarks.

2 Design Issues

The basic idea of using rollforward to ensure the atomicity is simple: at every pre-emption point, check whether an atomic sequence has been executing, and if so, roll it forward before resuming the new task. Any rollforward scheme thus needs to provide two mechanisms: one to *detect* that an atomic sequence has been executing, and one to *recover* from this situation. A third, orthogonal issue, is pre-emption *granularity*—defining what kind of events constitute a pre-emption point.

2.1 Pre-Emption Granularity

With rollforward, every pre-emption point incurs some extra overhead due to atomic sequence detection and recovery. Therefore, the lower the frequency at which pre-emption events occur, the better rollforward will work. On a uniprocessor the ultimate, and therefore finest-grained, pre-emption event is the interrupt. Solving the problem at this level will lead to a general solution. In many cases, however, this may be either not feasible, for example because a general user may not have permission to modify the operation system, or overly general, and thus, overly costly in terms of implementation and performance.

In general, it is desirable to define pre-emption at as high a level as possible. For any given application, the appropriate level can be determined quite easily. For example, in a Unix environment where the intent is to protect processes from asynchronously executing signal handlers, the pre-emption event is signal delivery. In a system with multi-threaded processes, pre-emption events are user-level scheduling decisions. In contrast, a system that allows sharing of data across address-spaces must also consider kernel-level scheduling decisions as pre-emption points. Only when sharing data with interrupt handlers will it be necessary to descend to the lowest level and define interrupts as the source of pre-emption.

For the sake of generality and concreteness, the rest of this article focuses on the case where interrupts are the pre-emption points. Unless noted otherwise, the following discussions apply directly or in a straightforward manner to coarser grained applications.

2.2 Registering Atomic Sequences

To detect the execution of atomic sequences, it is necessary to register their existence with the interrupt system. Many registration schemes are conceivable and we present only four possibilities in that spectrum: designated sequences, static registration, dynamic registration, and hybrid registration. The first two were previously suggested by Bershad et al. [10]; we repeat them here for completeness. We found that both approaches rather seriously limit flexibility. The third and fourth approach remove these limitations at the cost of slightly higher overheads.

2.2.1 Designated Sequences

The first proposal in [10] is to use designated instruction sequences to mark atomic sequences. This is also the technique used in [16]. Designated sequences must not appear anywhere but in atomic sequences. An interrupt handler can check whether the program-counter (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 precise. It is acceptable to mistakenly classify a code sequence as atomic (i.e., false hits are tolerable). No harm is done by executing a short piece of code prematurely. 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 imprecise designated sequences are not acceptable with rollback because restarting a code sequence that in reality is not naturally restartable may lead to unpredictable results. In this sense, designated sequences can be used more readily with rollforward than with rollback.

2.2.2 Static Registration

A 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 a software-based `test-and-set` operation, which is then used to build higher level synchronization objects. Limiting the number of atomic sequences unfortunately makes it impossible to inline atomic sequences. This imposes an indirect cost of a function call per atomic sequence. Also, a major reason for our interest in rollforward is the flexibility it affords. Restricting the number of atomic sequences to one would completely circumvent this flexibility.

2.2.3 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 [11], would work. 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.

2.2.4 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 sequence 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. An interrupt handler checks whether an atomic sequence has been pre-empted by first checking whether a registration is pending. If so, it must ensure the atomic sequence has not terminated already. This is achieved by checking whether a designated sequence appears within the next few instructions beyond the point of interruption. If both conditions hold (registration is pending and designated sequence is present), the interrupt handler concludes an atomic sequence was pre-empted and initiates a rollforward. Just as with (imprecise) designated sequences, there is a potential for false hits. As already observed, this is harmless as long as the sequence rolled forward terminates quickly.

The advantage of this scheme is twofold. First, not having to cancel registrations explicitly saves at least one instruction per atomic sequence. Second, registrations can be understood as hints: spurious "registrations" are tolerable as long as they occur infrequently. Not explicitly cancelling a registration is just one optimization enabled by the hint nature of registrations. As shown in Section 3, exploiting this property can lead to a highly efficient registration scheme that does not involve any loads or stores to memory.

2.3 Rolling Forward Atomic Sequences

A recovery mechanism is needed for the case where an atomic sequence has been pre-empted. With rollforward, recovery is achieved by finishing execution of the interrupted sequence before resuming the new task. Of course, rollforward should occur in a controlled fashion and must guarantee that control eventually will return to the interrupt handler.

2.3.1 Code Rewriting

An obvious solution is to rewrite the interrupted code by temporarily replacing the first instruction past the atomic sequence with a jump that leads back to the interrupt handler. A nice property of code rewriting is that it has no overheads except in the rare case of actual interference. 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, establishing coherence requires explicit cache flushing, which can be costly.

2.3.2 Cloning

The designers of Scheduler Activations [17] proposed a technique that has no overheads per atomic sequence but does not require write access to the code segment. The idea was to clone every atomic sequence. The original copy is left unmodified while the cloned copy is modified to end with an instruction that relinquishes control back to the interrupt handler. Given the short size of atomic sequences, code growth due to cloning is not likely to be a problem. 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.

2.3.3 Computed Jumps

Computed jumps provide an alternative solution. With this approach, every atomic sequence ends with a jump to an address stored in a variable, say *dest*. Before the atomic sequence is entered, the address of the instruction immediately following the atomic sequence is stored in *dest*. If the atomic sequence executes without interruption, *dest* points to the instruction after the jump and the jump acts as a "no operation" (NOP). Conversely, if the atomic sequence is interrupted, the interrupt handler can overwrite *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.

2.3.4 Controlled Faults

Sometimes it is more convenient for an interrupt handler to regain control via a fault (trap or software-interrupt) instead of a jump. This is particularly useful if the interrupt handler executes at a higher privilege level than the interrupted code. Just as with computed jumps, an instruction is placed at the end of the atomic sequence that normally acts as a NOP. If properly chosen, the interrupt handler can then change 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 during recovery, the interrupt handler would remove read access from that page. After 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 [18].

2.4 Challenges in Realizing Rollforward

So far, we have proposed mechanisms to regain control once an atomic sequence has finished execution. This is not sufficient, as it is also necessary to guarantee that a rollforward does indeed terminate. In fact, it is usually desirable that it will terminate within a certain time-limit. This problem becomes quite challenging if a rollforward can cause further faults.

2.4.1 Limiting the Duration of a Rollforward

If a precise registration scheme is used and the interrupt handler can trust in the code being rolled forward, limiting rollforward-duration is not an issue because atomic sequences will be short by design.

An imprecise registration scheme can result in the rollforward of arbitrary code sequences. Even if all code is trusted, it is necessary to employ a mechanism to limit rollforward duration. This is because perfectly reasonable code may contain endless loops (e.g., the idle-loop in an operating system). Similar care has to be taken if there is a potential for malicious code being rolled forward.

Excessively long rollforwards can 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 underlying address space could be terminated. A less aggressive solution would be to simply give no atomicity guarantees to such atomic sequences. That is, after the timer expires, execution is pre-empted and interrupt-handling is resumed. This solution could work reasonably well on systems with fine-grained, low-overhead watchdog timer support.

Yet another solution is to inspect the code prior to execution. This is feasible provided the inspected code does not modify itself; e.g., it is trusted or in a page with no write permission. If we also disallow backward jumps and subroutine calls, 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 this analysis can be done efficiently. 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.

2.4.2 Faults During Rollforward

Limiting rollforward duration is not necessarily equivalent to limiting the number of instructions in an atomic sequence. Particularly if there are faults during rollforward, the time required for a rollforward may easily exceed the desired limit. Worse still, if an atomic sequence repeatedly causes, for example, a division-by-zero fault, it may not be possible to ever complete the atomic sequence.

It is therefore necessary to impose the rule that atomicity is guaranteed only for sequences that execute fault-free. Except for page faults, this 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.

A simple solution is to avoid page faults in the first place. This is possible if the code of atomic sequences and shared memory can be pinned. By definition, pinned memory is never paged out and therefore stays 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.

If it is not possible to avoid page faults altogether, it may be possible to avoid them at inopportune places. If both the code of an atomic sequence and the data that it accesses are contained in a single page each, rollforward is applicable without any additional mechanisms: the atomic sequence obviously cannot be started before the instruction page is present. In contrast, the data page may cause a fault at an arbitrary point in the atomic sequence. However, as all data is contained in a single page, the atomic sequence cannot have read or modified any state yet. It is thus acceptable to block the atomic sequence while loading the data page. Eventually, both the instruction and data pages are present and at that point the sequence can finish execution without further faulting. If the limitation to single pages is unacceptable, prologue code touching all needed pages could be executed before starting the atomic sequence. Thus, by the time the atomic sequence starts execution, it is guaranteed that all pages are present. Note that the prologue code itself must be executed atomically as well (otherwise a page fault may lead to a needed page being paged out again). As the prologue does not modify state, it is a naturally restartable atomic sequence that can be implemented via rollback. In essence, the prologue code is executed repeatedly until it can execute without incurring a page-fault, at which point the rollforward-protected atomic sequence is started. Since all pages are now present, it is guaranteed that the atomic sequence can finish without further faulting.

Finally, note that instruction or data page faults during atomic sequences are permissible as long as no other process accesses any of the shared data being accessed by an atomic sequence that is blocked on a page fault. This can be achieved in a number of ways. A simple solution is to modify the registration protocol such that entering an atomic sequence would cause a fault if another atomic sequence is blocked on a page fault already. For example, a dummy read from a shared page could be performed when entering atomic sequences. Upon a page fault within an atomic sequence, read-permission is turned off and future processes will be prevented from entering atomic sequences. Once the blocked atomic sequence has terminated, read-permission is turned back on and all processes that faulted on the read to the dummy page can be resumed.

Of course, finer-grained schemes are possible. In the extreme case, one could control access to individual shared objects, thus maximizing concurrency. With fine-grained control, it might be more economical to control access via unaligned access faults rather than page protection faults. The general mechanism remains the same, however.

3 Implementation

This section describes three implementations of registration schemes. Given the large design space, the limitation to only three implementations is somewhat arbitrary. However, they were chosen based on unique features that make them worthwhile to highlight: the first uses dynamic registration and a computed jump to relinquish control back to the interrupt handler. 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 relinquish control. 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 ← addressOf(theEnd)
        inAS ← TRUE
        ⟨atomic sequence . . . ⟩
        inAS ← FALSE
        jump destAddr
theEnd:
```

If every atomic sequence is wrapped 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). It should be noted that on the Alpha, the load address instruction (`lda`) is an assembler-provided pseudo-instruction that usually expands into a load word instruction. However, in some cases it expands into a short sequence of ALU operations or several load instructions. The exact choice depends on the assembler in use and the address being loaded. Thus, the number of overhead instructions may be higher in some cases.

```
        * 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. Just as in C, 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).

	$destAddr \leftarrow addressOf(\text{theEnd})$
	$inAS \leftarrow \text{TRUE}$
	$\langle$atomic sequence...$\rangle$
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 protected with this registration scheme.

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. 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 code below 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 the instruction before address *destAddr* must be a jump (i.e., the jump instruction serves as the designated sequence). If these conditions hold and the code between the saved PC and theEnd is "safe" to execute (does not involve endless looping etc.), the interrupt handler initiates a rollforward.

```
        destAddr ← addressOf(theEnd)
        inAS ← TRUE
        ⟨atomic sequence . . . ⟩
        jump destAddr
theEnd:
```

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 to a 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 is tied up 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 overhead instructions: a load address and a jump (again, overhead instructions are marked with an asterisk). That is, *no memory accesses* are necessary to register an atomic sequence.[2] For dynamic registration schemes, this is probably close to the minimum achievable instruction count overhead. 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 straight-forward 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 more overhead instruction than Dyn/Jump.

Technique:	**Restrictions on Atomic Sequences:**
Rollback:	At most *one write* to shared memory.

[2] As noted before, the load address instruction is often translated into a load instruction, but this is mainly due to a limitation of the DEC UNIX assembler. There are other, potentially more efficient, schemes to load a 64 bit constant. Also, if the Alpha architecture would provide a PC-relative addressing mode, the first overhead instruction could be implemented by a single non-memory touching instruction, assuming the atomic sequence is reasonably short.

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 rollforward: (1) overhead per atomic sequence, (2) overhead per interrupt, and (3) overhead per rollforward. It is difficult to measure the overhead per interrupt in isolation. Instead, we report the overall performance of rollforward in an experimental OS kernel.

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 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 [7, 3]. 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 all measurements, except the ones for *sigprocmask*, more than 995 samples had the value of the mode.

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

Technique	DEC Alpha			HP PA-RISC 1.1		
	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	≥ 13	≥ 13	≥ 13	n/a	n/a	n/a
LL/STC	n/a	≥ 118	≥ 118	n/a	n/a	n/a

Table 1: Overheads of Different Atomicity Schemes in Cycles

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 a way that prevented the compiler's optimizer from taking 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 always 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 [19] 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 [20] 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 [7]. 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 [7], this does not appear to be an effective scheme to provide atomicity in a uniprocessor.

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 registration scheme has the potential for false hits, that is, 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, does 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 [21]. 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. Also, if the mechanism to relinquish control to the interrupt handler relies on a well-known register (such as Dyn/Jump and Hyb/Jump), the code sequence must not attempt to modify the well-known register. Uses of the well-known register are prohibited as well if the registration technique has the potential for false hits (e.g., Hyb/Jump). 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 [22]). 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:

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

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 μs. If the instructions reside in virtual memory and the data and instruction translation buffers (TLBs) are separate, an additional cost of one data TLB miss may be incurred. The exact cost of a TLB miss depends heavily on the organization of the page-tables and their state.

4.3 Overall Benefit

Rollforward 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. First, we provide a simple performance model that allows to estimate whether rollforward will be beneficial. We then provide some system-level measurements that were performed in an experimental kernel.

4.3.1 Performance 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

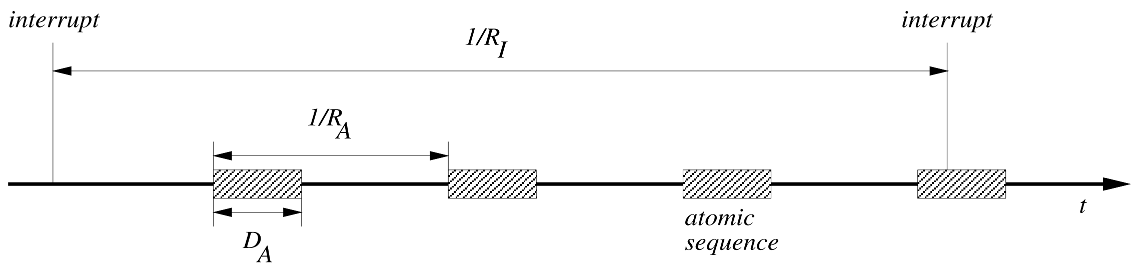

Figure 1: Model of operation.

we assume 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 rollforward 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 system measured in Section 4.3.2, 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 a proportional 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. The Alpha can issue up to two instructions per cycle, but we conservatively assume an average execution rate of only 1 instruction per cycle giving a D_A of 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 [7]. 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:

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

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.

4.3.2 System Measurements

We extended an experimental OS kernel with rollforward and measured its performance under different loads. The Dyn/Jump scheme was used to register atomic sequences. The kernel does not currently support paging, so page-faults were not an issue. For comparison, the same kernel was measured when using a hardware-based scheme to protect atomic sequences. On the Alpha, this is implemented via two PALcode calls per atomic sequence: one to raise and one to restore the interrupt priority level. The experiments were performed on a 133MHz DEC 3000/400 workstation, which is slower than a DEC 3000/600. However, the tests were small enough to fit in the secondary cache and the secondary cache performance scales with CPU clock-frequency, so the relative behavior should be identical. The workloads are:

IDLE: There is one process executing a tight loop incrementing a counter. Throughput is measured as the number of times the counter is incremented per 1000 cycles. This is a worst-case scenario for rollforward since almost no atomic sequences are executed, while timer interrupts occur at a frequency of 64 Hz.

BUSY1: Like IDLE, but the code incrementing the counter is now executed as an atomic sequence. This is a best-case scenario for rollforward because atomic sequences are executed at the highest possible frequency while interrupts occur at the relatively low frequency of 64 Hz. Notice that almost every interrupt will lead to a rollforward in this case.

BUSY2: Like BUSY1, except that the loop overhead increased to about 300 cycles. This increase comes from voluntarily yielding the processor once per loop. This was done to facilitate a fair comparison with the next workload (ETH). Interrupt frequency is still 64 Hz.

ETH: A process executes the same atomic sequence as in the BUSY2 case. At the same time, minimum sized Ethernet packets are exchanged between two machines. This increases interrupt frequency from 64 Hz to about 7.5 kHz.

The measured performance is summarized in Table 2. Column PAL reports the throughput for the kernel using PALcode calls, while RF reports the throughput for the kernel using rollforward.

	Throughput [incr/kcyc]		
Load	PAL	RF	Δ
IDLE	166.0	166.0	0%
BUSY1	19.5	55.4	+184%
BUSY2	4.2	5.1	+21%
ETH	3.8	4.3	+13%

Table 2: Measured Overall-Performance

The IDLE row shows that the per-interrupt overhead for rollforward indeed is insignificant. Both versions achieve a throughput of about 166 increments per 1000 instructions. The roughly 20 instructions that the interrupt handler executes to check whether a rollforward is necessary can be interleaved with the code that saves the machine state. Because the memory-system is the bottleneck at that point, executing a few additional integer operations costs very little. The BUSY1 row shows that rollforward significantly outperforms the PALcode-based kernel. In this case, throughput is almost 3 times higher—despite the fact that almost every interrupt causes a rollforward. BUSY2 executes atomic sequences at a lower frequency than BUSY1. Correspondingly, the performance advantage of rollforward drops to 21%. Comparing the ETH row to BUSY2 shows that adding the Ethernet load reduces throughput in the PALcode-based kernel by about 10%. The reduction in the rollforward case is with 16% somewhat higher. This can be explained by the fact that interrupt frequency increased by about a factor of 117. Nevertheless, rollforward is still faster by about 13%.

We make two final observations. First, in the ETH workload, the roundtrip time to exchange a packet between the two machines was slightly better (by about 1%) with rollforward than when using PALcode. Second, rollforward consistently showed a standard-deviation that was a factor of two smaller than PALcode. Minimizing variance, as opposed to minimizing latency, is a performance goal that we expect to increase in importance [23].

5 Related Work

As mentioned in the introduction, Bershad et al. propose a software-based technique for mutual exclusion on uniprocessors [10]. It is based on rollback instead of rollforward and differs from the latter in several specific ways. First, the solution is limited to atomic sequences that contain only a single write to shared memory, whereas the rollforward approach permits multiple writes. It is often more convenient, and in the end more efficient, to implement shared data-structures via sequences that require multiple writes to shared memory [11].

Second, rollforward supports efficient lock-free solutions, whereas rollback was designed for a lock-based scenario. For uniprocessors, lock-free data-structures are interesting 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. These advantages apply to multi-processors as well. However, as argued by Herlihy [12], the interest in multi-processor community appears to be based mainly on the second property (absence of deadlock), which makes it easier to build machines that are resilient against processor failures.

Third, dynamic registration schemes allow inlining of atomic sequences, whereas the static registration scheme proposed in [10] severely restricts the number of atomic sequences. Similarly, designated sequences are substantially easier to use with rollforward because false hits are acceptable. The bottom line is that rollforward makes software-based mutual exclusion a much more easily applicable technique.

Rollforward has been used successfully in the VAX runtime system of the Trellis/Owl language [16]. 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 rollforward. 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 [15]. 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 process's context simply by ensuring—for example, via code-inspection or a watch-dog timer—that it will relinquish control of the CPU in a timely manner. Also, we discussed scenarios in which page faults do not pose a problem and proposed several solutions on how page faults could be handled in an environment in which they are not transparent.

Another system that employed rollforward is Scheduler Activations [17]. 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. The paper gives only a superficial treatment of the issues involved. It does not give any indications as to the cost or limitations of rollforward.

Finally, Stodolsky et al. [9] present 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. It is interesting to note 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 outside the privileged kernel of an operating system. As a matter of fact, the paper does not make a connection between optimistic interrupt protection and rollforward.

6 Concluding Remarks

This article describes a rollforward-based technique to implement atomic sequences on a uniprocessor. Both, micro- and macro-experiments show our approach to be as efficient as, and in many cases significantly more efficient than, hardware-based mutual exclusion mechanisms. Rollforward is more flexible than schemes based on rollback in that it allows multiple writes. Atomic sequences with multiple writes can be used to construct powerful primitives which, in turn, can be used to simplify the implementation of lock-free data-structures. We conclude this article by making some observations about the mechanism.

First, we described that page faults present a bigger challenge with rollforward than with 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 [10]. With rollforward, there is the additional potential for a page fault during recovery. While this may pose a serious limitation in certain environments, we discussed several possible solutions and also showed that page faults are not an issue in at least two important cases: in-kernel synchronization and protection against Unix-style signal handlers. On the other hand rollforward makes it much easier to use designated sequences because false hits are tolerable. The Hyb/Jump techniques fully exploits this property.

Second, even if a hardware-based mechanism is more efficient, a software-based method may be preferable because it is applicable in unprivileged user-mode. This is particularly useful for software that may execute either in user-level or kernel-level mode. For example, OS servers may be developed and debugged in user-level space but moved into the kernel for a production environment.

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). In practice, this overhead is often in the 30 cycle range due to cache effects (PALcode is located at a platform-dependent fixed address and therefore not inlinable). As the gap between memory system and CPU performance continues to grow [18, 24], techniques that yield good cache performance will become ever more important. Software-based methods that use designated sequences, a dynamic, or hybrid scheme to register atomic sequences can all be inlined easily.

Fourth, our experience is that rollforward is very practical. 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 actually used. 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 a "fast signals" package (*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 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 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.

Sixth, the performance model and system measurements presented in Section 4.3 show that the benefit of rollforward increases with the frequency at which atomic sequences execute. Given the small overhead of dynamic registration, this encourages building systems with fine-grained synchronization. That is, instead of few large atomic sequences, it will usually be better to structure the system using several small ones. Not only does this reduce average interrupt latency, but it also implies that the system will be structured in a manner that is more suitable for shared-memory parallel machines, where small atomic sequences mean increased concurrency and thus a potential for increased overall performance. A good example is the user-level implementation of POSIX threads presented in [25]. Concerns about the overhead of signal-masking led to a design using one big monitor rather than small atomic sequences. As a result, a major redesign will be necessary should this package ever be extended to support multiprocessors. With rollforward, signal masking costs could have been reduced by an order of magnitude and would have led to a design that more naturally exhibits the inherent synchronization requirements of the package.

Finally, there are a number of low-level issues that arise when implementing an interrupt handler that

[3]The limitation of the user-level implementation of *sigprocmask* is that signal stacks are unsupported. Also, no attempt was made to emulate the semantics for signals that can stop a process (e.g., SIGTSTP, SIGTTOU, and SIGTTIN).

supports software mutual exclusion. They are not particularly difficult to solve but we would like to conclude by mentioning 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 not for an Alpha using either the VMS or DEC UNIX 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.

Source Code Availability

The source code for the fast-signals package and a stand-alone experimental OS kernel for the Alpha architecture that uses rollforward are available on request. Requests can be sent via e-mail to `davidm@cs.arizona.edu`.

References

[1] Maurice Herlihy, 'Wait-free synchronization', *ACM Trancsactions on Programming Languages and Systems*, **11**(1), 124–149 (1991).

[2] Gregory R. Andrews, *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, Menlo Park, 1991.

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

[4] Intel, *80960CA User's Manual*, 1989.

[5] *The PowerPC Architecture*, Cath May, Ed Silha, Rick Simpson, and Hank Warren (eds.), Morgan Kaufman, second edition, 1994.

[6] Charles Price, *MIPS IV Instruction Set*, MIPS Technologies, Inc., Mountain View, CA, 1995.

[7] *Alpha Architecture Reference Manual*, Richard L. Sites (ed.), Digital Press, Burlington, Massachusetts, 1992. Order number EY-L520E-DP.

[8] SPARC International, *The SPARC Architecture Manual, Version 8*, Prentice-Hall, 1992.

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

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

[11] Henry Massalin, 'Synthesis: An efficient implementation of fundamental operating system services', *Ph.D. Thesis*, Columbia University, New York, NY 10027, September 1992.

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

[13] 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.

[14] 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.

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

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

[17] 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', *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, 1991.

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

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

[20] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, 'Mach: A new kernel foundation for UNIX development', *USENIX Conference*, 1986, pp. 93–112.

[21] A. M. Turing, 'On computable numbers, with an application to the Entscheidungsproblem', *Proceedings, London Mathematical Society,*, 1936, pp. 230–265. Published as Proceedings, London Mathematical Society,, volume 2, number 42.

[22] DEC, *DEC 3000 300/400/500/600/800 Models, System Programmer's Manual*, Digital Equipment Corp., 1993. Order number EK-D3SYS-PM.A01.

[23] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman, 'Scout: A communications-oriented operating system', *Technical Report 94/20*, University of Arizona, Tucson, AZ 85721, June 1994.

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

[25] Frank Mueller, 'A library implementation of POSIX threads under UNIX', *USENIX Technical Conference Proceedings*. USENIX, Winter 1993, pp. 29–41.