USENIX Association

Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference

San Antonio, Texas, USA
June 9-14, 2003

An Implementation of User-level Restartable Atomic Sequences on the NetBSD Operating System

Gregory McGarry
g.mcgarry@ieee.org

Abstract

This paper outlines an implementation of restartable atomic sequences on the NetBSD operating system as a mechanism for implementing atomic operations in a mutual-exclusion facility on uniprocessor systems. Kernel-level and user-level interfaces are discussed along with implementation details. Issues associated with protecting restartable atomic sequences from violation are considered. The performance of restartable atomic sequences is demonstrated to outperform syscall-based and emulation-based atomic operations. Performance comparisons with memory-interlocked instructions demonstrate that restartable atomic sequences continue to provide performance advantages on modern hardware. Restartable atomic sequences are now the preferred mechanism for atomic operations in the NetBSD threads library on all uniprocessor systems.

1 Introduction

The NetBSD Project is currently adopting a new threads system based on scheduler activations[6]. Part of this project is the implementation of a POSIX-compliant threads library that utilises the scheduler activations interface. The motivation for the threads project is to support the multi-threaded programming model which is becoming increasingly popular for application development. Multi-threaded applications use multiple threads to aid portability to multiprocessor systems, and as a way to manage server concurrency even when no true system parallelism is available. To support multi-threaded applications, the POSIX standard specifies a mutual-exclusion facility to serialise access and guarantee consistency of shared data. Even on uniprocessor systems, mutual-exclusion facilities are necessary to protect shared data against an interleaved thread schedule. Interleaving can occur when a thread blocks on a resource or when a thread is preempted, causing another thread to assume control of the processor.

The scheduler activations threading model also places additional demand on the mutual exclusion facility. The primary advantage of scheduler activations is that it combines the simplicity of a kernel-based threading system and the performance of a user-level threading system[1]. While the kernel component of the model controls the switching of execution contexts, the user-level component is responsible for scheduling threads onto the available execution contexts. The user-level component contains a complete scheduler implementation with shared scheduling data which must be protected by the mutual-exclusion facility. Consequently, the mutual-exclusion facility is a critical component of the scheduler activations threading model.

The basic building block for any mutual-exclusion facility, whether it be a *blocking* or *busy-wait* construct, is the *fetch-modify-write* operation[5]. The fetch-modify-write operation reads a boolean flag that indicates ownership of shared data. The operation modifies the flag from false to true, thereby acquiring ownership. The fetch-modify-write operation must execute atomically (without interruption) to ensure the flag state is maintained consistent across all contending threads.

Modern systems generally provide sophisticated processor primitives within the hardware in the form of memory-interlocked instructions and bus support to ensure that a given memory location can be read, modified and written atomically. The specific primitive varies between processors, however common instructions include *test-and-set*, *fetch-and-store* (swap), *fetch-and-add*, *load-locked/store-conditional* and *compare-and-swap*[5, 2]. Unfortunately, there are two problems associated with the use of memory-interlocked instructions within a user-level threads library:

- Memory-interlocked instructions incur overhead since the cycle time for an interlocked memory access is several times greater than that for a non-interlocked access. The overhead associated with memory-interlocked instructions is due to memory and interconnection contention.
- Not all processors support memory-interlocked instructions and therefore cannot provide atomic operations for a mutual-exclusion facility. Example processors include the MIPS R3000, VR4100 and ARM2 processors. Interestingly, processor manufacturers are choosing to introduce new processors without memory-interlocked instructions. These processors generally provide a subset of the complete processor specification and primarily target

embedded or low-power applications.

Both of these problems are important to the NetBSD Project due to the large number of hardware systems that are supported.

On multiprocessor systems, the hardware is expected to provide the necessary atomic operations. Multiprocessor atomic operations for synchronisation have been extensively investigated by Mellor-Crummey and Scott[5]. However, the majority of systems supported by the NetBSD operating system are uniprocessor systems. Therefore, it is desirable to find an alternate mechanism for atomic operations which works efficiently on all uniprocessor systems.

The simplest mechanism for implementing an atomic operation is to request the kernel to perform the operation of behalf of the user-level thread. A call to a specific *system call* can be used to enter the kernel. While inside the kernel, the scheduler can be paused so that the thread is guaranteed not to be preempted while a non-atomic fetch-modify-write operation is performed. Alternatively, an invalid instruction can be executed by the thread to cause an exception which can be intercepted by the kernel to *emulate* an atomic fetch-modify-write operation. An advantage of instruction emulation is that the invalid instruction can be forward compatible with newer versions of the processor which do support explicit atomic operations. However, both syscall-based and emulation-based solutions incur significant overhead by entering the kernel.

Atomic operations can also be implemented using a software reservation mechanism. With the software reservation mechanism, a thread must register its intent to perform an atomic operation and then wait until no other thread has registered a similar intent before proceeding. The most widely recognised algorithm based on software reservation is Lamport's mutual-exclusion algorithm[3].

In Lamport's algorithm, the atomic operation is protected by two variables; one to indicate ownership of the atomic operation and another to place reservations. Each thread registers in a global array its intent to perform an atomic operation. The thread then places a reservation into the reservation variable before testing the ownership variable. If a thread determines that ownership is held by another thread, there is *contention*, and the thread must wait until ownership is relinquished. When the thread determines that ownership has been relinquished, it assigns its ownership to the atomic operation then checks that it still holds the reservation. If another thread holds the reservation then a *collision* has occurred. The thread then waits for all contending threads to acquire ownership and remove their intent from the global array. The thread then restarts the procedure from the beginning.

The software reservation mechanism works equally well on both uniprocessor and multiprocessor systems. However, even for the case when no contention and no collisions occur, reservation-based algorithms require several memory accesses per atomic operation. Additionally, these algorithms have storage requirements that increase linearly with the number of potential contending threads. For a user-level threads library, it is not always possible to know the maximum number of threads likely to be used in an application. For these reasons, the software reservation mechanism was not considered further.

The mechanism of restartable atomic sequences has been proposed to address the problems outlined above for implementing atomic operations on uniprocessor systems[2]. The basic concept of restartable atomic sequences is that a user-level thread which is preempted within a restartable atomic sequence is resumed by the kernel at the beginning of the atomic sequence rather than at the point of preemption. This guarantees that the thread eventually executes the instruction sequence atomically.

This paper outlines the implementation of restartable atomic sequences to provide the atomic operations required by the POSIX-compliant threads library on the NetBSD operating system. Section 2 discusses the concept of restartable atomic sequences. Section 3 presents details of the implementation on the NetBSD operating system. Section 4 compares the performance of restartable atomic sequences with other mechanisms for implementing atomic operations including memory-interlocked instructions, instruction emulation and a syscall-based mechanism. Section 4 also examines the cost associated with restartable atomic sequences. Section 5 discusses the application of restartable atomic sequences in the POSIX-compliant threads library and presents some benchmarks of the mutual-exclusion facility in the threads library. Finally, Section 6 presents conclusions.

2 Restartable Atomic Sequences

Restartable atomic sequences is a mechanism to implement atomic operations on uniprocessor systems. Responsibility for executing the instruction sequence atomically is shared between the user-level thread and the kernel. When a thread is preempted within a restartable atomic sequence, it is resumed by the kernel at the beginning of the atomic sequence rather than at the point of preemption.

Most mechanisms used to implement atomic operations on uniprocessor systems can be called pessimistic. Their design assumes that atomicity may be violated at any moment and guards against this potential violation for every atomic operation. Restartable atomic sequences use an optimistic mechanism that assumes that

atomic sequences are rarely preempted and are inexpensive for this common case. Only when an atomic sequence is not executed atomically is it necessary to perform a recovery action to ensure atomicity.

Restartable atomic sequences require kernel support to ensure that a preempted thread is resumed at the beginning of the sequence. An application registers the address range of the atomic sequence with the kernel. If a user-level thread is preempted within a registered atomic sequence, then its execution is rolled-back to the start of the sequence by resetting the program counter saved in its process control block.

Consider the test-and-set atomic operation in Figure 1 which reads the memory address, writes to the memory address and returns the read value. The memory address could be a flag in a mutual-exclusion operation. If the test-and-set operation executes atomically, two conditions can occur. If `*addr` is unset, then it is set and the function returns the unset value. If `*addr` is set, then it remains unchanged and the function returns the set value.

Now consider what will happen if the test-and-set operation is preempted between the memory read and write operations and a collision occurs when accessing `*addr`. Again, two conditions occur. If `*addr` is unset, `old` is unset. The thread is then preempted. At this point the new thread will read `*addr` as unset, and set `*addr`. When the original thread resumes, its recorded value of `old` no longer reflects the true value of `*addr` and will also set `*addr`. Both threads will have the test-and-set operation return success to setting the memory address. If this condition occurred while the test-and-set operation was being used in a mutual-exclusion operation, then both threads would assume ownership of a shared resource.

The other collision condition occurs if `*addr` is read as set before the thread is preempted. The new thread clears `*addr`. When the original thread resumes, its recorded value of `old` no longer reflects the true value of `*addr`, it will set `*addr` but return the unset value. In this condition, the original thread believes that the memory address was already set, while the second thread has cleared the memory address. If this condition occurred while the test-and-set operation was being used in a mutual-exclusion operation, then neither thread would assume ownership of the shared resource. Neither thread would be able to reacquire ownership of the resource and *deadlock* would occur.

Atomicity of the test-and-set operation is assured by making the adjacent memory accesses between `__ras_start` and `__ras_end` a restartable atomic sequence. Now consider what will happen if the test-and-set operation is preempted between the memory read and write operations. Again, two conditions occur. If

```
int test_and_set(int *addr)
{
  int old;

  __asm __volatile("__ras_start:");
  old = *addr;
  *addr = 1;
  __asm __volatile("__ras_end:");

  return (old);
}
```

Figure 1: Implementation of a test-and-set atomic operation protected as a restartable atomic sequence.

`*addr` is unset, `old` is unset. The thread is then preempted. At this point the new thread will read `*addr` as unset, and set `*addr`. When the original thread resumes, its execution is rolled-back to `__ras_start` which corresponds to the instruction to read `*addr`. The value of `old` now corresponds to the correct value of `*addr` set by the other thread. The value of `*addr` remains unchanged and the correct value is returned by the test-and-set operation. The other collision condition occurs analogously.

Restartable atomic sequences should adhere to the following requirements:

1. have a single entry point;
2. have a single exit point;
3. restrict modifications to global/shared data;
4. not execute emulated instructions or invoke system calls; and
5. not invoke any functions.

The first requirement is to ensure that the roll-back of the program counter is valid. Nevertheless, the kernel cannot guarantee that the sequence is successfully restartable; it assumes that the application knows what it is doing. The second and third requirements are linked. Access to global/shared data should be restricted to ensure that the restartable atomic sequence is *idempotent*. An operation is idempotent if it achieves the same result irrespective of the number of times it executes. Restricting the restartable atomic sequence to a single global write in the last instruction of the restartable atomic sequence ensures that the operation is idempotent. Accordingly, a single exit point is desirable. However, if the atomic operation chooses not to modify any global data, the restartable atomic sequences may be exited at any point.

The fourth requirement is to ensure that the kernel is not entered and thus providing an opportunity for the

thread to block on a resource and be preempted. In this case, the kernel will always roll-back execution to the beginning of the restartable atomic sequence whenever the thread is unblocked, and the thread will never exit the restartable atomic sequence. The fifth requirement is to ensure that the program counter remains within the range of the registered atomic sequence.

There are two run-time costs associated with restartable atomic sequences. Because the kernel identifies restartable atomic sequences by an address range, restartable atomic sequences cannot be inlined. The inability to inline atomic sequences slightly increases the overhead of atomic operations due to the cost of subroutine calls. The second run-time cost comes from checking the program counter at each context switch. Although this test can add several cycles to the kernel's context switch path, many applications use many more atomic operations than the number of context switches, making the additional scheduling overhead negligible.

3 Implementation

The NetBSD operating system is well-known for its portability and support for many architectures. The portability of the operating systems stems from a clear separation of machine-dependent and machine-independent subsystems. An implementation of restartable atomic sequences clearly requires access to machine-dependent information such as the thread program counter. However, much of the implementation can be shared between all architectures and is machine-independent. Additionally, the user-level interface should be uniform across all architectures. Consequently, the primary objective of this implementation was to provide a generic interface to be utilised by all supported architectures. To that end, the implementation consists of a simple system-call interface and a largely machine-independent kernel implementation.

Initially, the implementation was intended to support atomic operations only for use by the threads library. However, it was recognised that a generic user-level interface would allow applications to find new and innovative uses of restartable atomic sequences. For example, benchmark utilities, performance counters, and profiling tools may make use of restartable atomic sequences to ensure that an analysed instruction sequence is executed without interruption. Supporting new and innovative uses seemed like a worthwhile goal.

The initial design decision was that only thread-synchronous events will cause an atomic sequence to be restarted and asynchronous events, such as interrupts, do not cause a restart. Therefore, *a restartable atomic sequence will only be restarted if the thread execution context is switched from the processor*. Asynchronous events are difficult to protect, since they must be identified outside the context of the executing thread. Additionally, the handling of asynchronous events is a machine-dependent operation and would require invasive changes to the machine-dependent kernel. For example, on architectures such as i386 and m68k, it is difficult to provide a central location to check if the program counter is within a restartable atomic sequence since interrupts are dispatched via an interrupt table.

Another important consideration is how registered restartable atomic sequences are handled by the *fork* and *exec* system calls. Restartable atomic sequences are inherited from the parent by the child during the *fork* system call. This allows restartable atomic sequences to continue to work on children of the parent process that registered the sequences. Restartable atomic sequences are removed during the *exec* system call. This property is intuitive given that the program text has changed and the instruction sequence for a registered atomic sequence is different.

The implementation supports the registration of multiple restartable atomic sequence for a process. A list of address ranges is maintained for each process. A per-process limit of the maximum number of registered restartable atomic sequences is imposed to limit resource exhaustion.

3.1 Kernel interface

All atomic sequences for a process are manipulated by the *rasctl* system call. Its prototype can be found in `<sys/ras.h>` and is implemented within the standard C library. It has a prototype given by

```
int
rasctl(void *addr, size_t len, int op)
```

The prototype is intended to be similar to the *mmap* system call, since both system calls affect the process address space. If a restartable atomic sequence is registered and the process is preempted within the range `addr` and `addr+len`, then the process is resumed at `addr`. The operations that can be applied to a restartable atomic sequence are specified by the `op` argument. Possible operations are:

- `RAS_INSTALL`: register a new atomic sequence;
- `RAS_PURGE`: remove a registered atomic sequence for this process; and
- `RAS_PURGE_ALL`: remove all registered atomic sequences for this process.

The operation affects the restartable atomic sequence immediately.

The purge operation should be considered to have undefined behaviour if there are any other runnable threads in the process which might be executing within the registered atomic sequence at the time of the purge. The

```
#include <sys/ras.h>

extern void __ras_start(void);
extern void __ras_end(void);

...
if (rasctl((void *)__ras_start,
   (size_t)(__ras_end - __ras_start),
   RAS_INSTALL))
  errx(1, "rasctl failed");
```

Figure 2: The registration process for the restartable atomic sequence presented in Figure 1.

application must be responsible for ensuring that there is some form of coordination between threads.

The *rasctl* system call will fail with `EINVAL` if an invalid operation is specified, if `addr` or `addr+len` are invalid user addresses, or if the maximum number of restartable atomic sequences per process is exceeded. It will return `ESRCH` if the restartable atomic sequence cannot be found during a purge operation.

Figure 2 shows an example of registering the restartable atomic sequence for the test-and-set atomic operation presented in Figure 1.

3.2 Kernel implementation

All registered restartable atomic sequences for a process are recorded in a linked list, `p_raslist`, located in `struct proc` of the process. Each element in the linked list records the start address and end address of a single registered restartable atomic sequence. Additionally, a counter is available to record the number of restarts actioned for the restartable atomic sequence. This counter can provide some interesting information but is rarely useful for most applications. Currently there is not a user-level interface to access the counter.

A counter, `p_nras` in `struct proc` records the number of registered atomic sequences and is used to simplify the program-counter check. The program counter is checked within the `cpu_switch()` function. The `cpu_switch()` function is a machine-dependent function which is responsible for switching the context of the active thread on the processor. The `cpu_switch()` function has a pointer to the `proc` structure passed as the first argument, and a check if `p_nras` is non-zero is an inexpensive test. The machine-dependent implementation code on the i386 adds merely three instructions to the main execution path for the case of no registered atomic sequences. If `p_nras` is non-zero, then `ras_lookup()` is invoked to compare the program counter with all registered restartable atomic sequences. The `ras_lookup()` function is machine-independent and has the function prototype

```
caddr_t
ras_lookup(struct proc *p,
           caddr_t addr)
```

It searches the registered restartable atomic sequences for process `p` which contains the user address `addr`. If the address `addr` is found within a restartable atomic sequence, then the restart address of the restartable atomic sequence is returned, otherwise -1 is returned. In the case of a match, the machine-dependent code in `cpu_switch()` uses the start address to reset the program counter in the process control block.

The `RAS_INSTALL` and `RAS_PURGE` operations of the *rasctl* system call invoke the `ras_install()` and `ras_purge()` functions. They have the prototypes

```
int
ras_install(struct proc *p,
            caddr_t addr, size_t len)
int
ras_purge(struct proc *p,
          caddr_t addr, size_t len)
```

The `ras_install()` function will return `EINVAL` if `addr` or `addr+len` are invalid user addresses, or if the maximum number of restartable atomic sequences per process is exceeded. The `ras_purge()` function will return `ESRCH` if the specified restartable atomic sequence has not been registered.

The `ras_fork()` function is used to copy all registered restartable atomic sequences for a process to another. It is primarily during the *fork* system call when the sequences are inherited from the parent by the child. It has the prototype

```
int
ras_fork(struct proc *p1,
         struct proc *p2)
```

The `ras_purgeall()` function is used to remove all registered restartable atomic sequences for a process. It is primarily used to remove all registered restartable atomic sequences for a process during the *exec* system call and to perform the `RAS_PURGE_ALL` operation for the *rasctl* system call. It has the prototype

```
int
ras_purgeall(struct proc *p)
```

The `ras_fork()` and `ras_purgeall()` functions are guaranteed to complete successfully.

3.3 Additional kernel issues

Restartable atomic sequences are user-level instruction sequences that receive special consideration by the kernel. In addition to checking if the program counter is within a restartable atomic sequence when a thread context is restored, it is also important for the kernel not to violate the prerequisites for their correct operation. One potential problem for the kernel is the *ptrace* facility.

The *ptrace* facility provides tracing and debugging facilities. It allows a process (the tracing process) to control another (the traced process). Most of the time, the traced process runs normally, however the *ptrace* facility provides some kernel capabilities to modify the behaviour of the traced process. If the traced process contains registered restartable atomic sequences then the kernel must ensure that they are protected from modification by the tracing process, otherwise one or both of the processes will fail to perform as expected.

There are two specific cases which must be considered:

- The tracing process attempts to write to a restartable atomic sequence (`PT_WRITE_I`). An example is when the tracing process attempts to set a breakpoint.
- The traced process attempts to single-step into a restartable atomic sequence (`PT_STEP`).

The first case is handled within the machine-independent *ptrace* facility. Each write to the code segment is first checked to ensure that the write is not to a restartable atomic sequence. Attempting to write to a restartable atomic sequence fails.

The second case is more difficult to handle, since different architectures handle single-stepping mode differently. For example, the MIPS R3000 processor does not have a single-stepping or tracing mode and the facility is generally handled through software emulation. Software emulation works by replacing the next instruction with an invalid opcode which generates an exception that the kernel identifies and handles specifically. Protecting the restartable atomic sequences during emulation is the same as for the first case discussed above. However, care must be taken, since the same emulation technique is used to single-step the kernel inside the kernel debugger, and restartable atomic sequences are not supported within the kernel.

Other architectures such as i386 and m68k provide tracing support in hardware. On these architectures the trace trap must check if the program counter is within a restartable atomic sequence before dispatching the event to the tracing process via the *ptrace* facility. The usual procedure is to continue stepping through the restartable atomic sequence and only dispatch the event on the first instruction after the atomic sequence.

4 Performance Evaluation

In this section, the performance of restartable atomic sequences is compared with the competing mechanisms. The test-and-set atomic operation based on restartable atomic sequences is compared with a syscall-based mechanism, instruction emulation on the MIPS R3000 processor and memory-interlocked instructions. The overhead associated with checking the program counter during a context switch is also investigated.

All microbenchmarks presented in this section are based on mutual exclusion mechanisms with a test that enters a critical section using a *test-and-set* lock and leaves the critical section by clearing the *test-and-set* lock. The benchmark uses a single thread, so that no contention occurs. The benchmark is measuring the performance of the basic processor architecture, memory system and mutual exclusion mechanism. The measures are determined by executing the benchmark in a loop one million times. The loop overhead and overhead for clearing the lock is eliminated in the published measures.

As already mentioned, restartable atomic sequences use an optimistic mechanism that assumes that atomic sequences are rarely preempted and are inexpensive for this common case. By way of example, a system with a 100MHz i486DX processor executes one million iterations of the benchmark outlined above in 0.38 seconds with only 4 restarts actioned.

4.1 Syscall-based atomic operations

The syscall-based mechanism uses the kernel to perform all the necessary actions to ensure atomicity. The mechanism only works on uniprocessor systems and is successful because the NetBSD kernel is not preemptable[4]. Support is provided by two system calls to acquire and release the lock on behalf of the thread. These system calls were added to a NetBSD kernel for the explicit purpose of comparing its performance with restartable atomic sequences.

The system calls take the address in the process address of the lock. The *acquire* system call will block the current thread until the lock is released. The *release* system call will release the lock to any blocked threads. The system calls are implemented using the `tsleep()`/`wakeup()` kernel facility.

The elapsed times to execute the test-and-set atomic operation based on the syscall mechanism and restartable atomic sequences for various processors are shown in Table 1. From this benchmark it can be seen that on the MIPS R3000 processor, restartable atomic sequences provide almost ninety-fold performance improvement over syscall-based atomic operations. The run-time cost for syscall-based atomic operations is high. The kernel must be invoked on every atomic operation, requiring that a trap be fielded, dispatched and ar-

system	*RAS*	*syscall*
DECstation 5000/25	0.40	35.6
100MHz i486DX	0.32	21.5
DECstation 5000/260	0.17	9.7
Alchemy Pb1000 EVB	0.04	2.4
Broadcom BCM91250A EVB	0.04	1.4

Table 1: Elapsed times (in microseconds) to execute the test-and-set atomic operation based on the syscall mechanism and restartable atomic sequences (RAS).

guments checked. Modern processors in the MIPS family appear to be less sensitive to system-call overhead, however restartable atomic sequences continue to provide a significant performance improvement.

4.2 Instruction emulation

The original MIPS instruction set architecture (ISA) implemented in the R3000 processor did not provide any memory-interlocked instructions. The MIPS II ISA implemented in the R4000 and most subsequent processors introduced a load-locked/store-conditional instruction pair. The MIPS R3000 processor will generate a trap if it attempts to execute a load-locked or store-conditional instruction. This trap can be intercepted by the kernel and the necessary actions performed to adequately emulate the instruction pair.

Instruction emulation has the advantage that software can be written for any processor in the family and the unsupported instructions are supported transparently.

The NetBSD kernel currently emulates the load-locked and store-conditional instructions for the MIPS R3000 processor. The current implementation only operates on uniprocessor systems and is successful because the kernel is not preemptable[4].

Although instruction emulation requires no special hardware, its run-time cost is high. Similar to the syscall-based mechanism, the kernel must be invoked on every emulated instruction, requiring that a trap be fielded, dispatched and arguments checked. A single test-and-set atomic operation on the DECstation 5000/25 based on restartable atomic sequences can execute in 0.4 microseconds. The test-and-set atomic operation using instruction emulation executes in 56 microseconds. The cost of instruction emulation is higher than the syscall-based test-and-set atomic operation, since each test-and-set operation uses a load-locked and store-conditional instruction, which generates two traps to the kernel to emulate the instructions. Therefore, on the MIPS R3000 processor, the syscall-based mechanism is faster than instruction emulation.

4.3 Memory-interlocked instructions

The performance of atomic operations based on restartable atomic sequences is compared with memory-interlocked instructions on processors that provide such functionality. Two memory-interlocked implementations are considered. An *inlined* version uses compiler and assembler optimisations to schedule the memory-interlocked instructions in the instruction stream at the point of invocation. A *non-inlined* version wraps the memory-interlocked instructions in a function call. Due to the overhead associated with dispatching a function call, the inlined version is almost always expected to provide a performance gain. As mentioned in Section 2, restartable atomic sequences cannot be inlined.

To compare the performance of the two implementations of memory-interlocked instructions and restartable atomic sequences, a performance index is introduced. The metric is calculated by

$$M = 100 \left(\frac{t_{NI} - t}{t_{NI}} \right),$$

where t is the execution time of the atomic-operation mechanism and t_{NI} is the execution time for a non-inlined memory-interlocked instruction. Therefore, the performance index provides an indication of improvement over a non-inlined memory-interlocked instruction. A performance index of zero indicates no performance improvement. A performance index of one-hundred indicates zero cost associated with an operation, or effectively infinite performance improvement.

The performance indices comparing atomic operations based on restartable atomic sequences and inlined memory-interlocked instructions are shown in Table 2. The table is ordered approximately corresponding to the processing power (or age) of the processor.

Table 2 shows that for older processors, restartable atomic sequences exhibit an additional cost over memory-interlocked instructions. On these machines, restartable atomic sequences are paying the penalty of the function call. The modern processors tend to indicate a potential performance improvement of restartable atomic sequences over memory-interlocked instructions. This improvement is mainly attributed to the introduction of on-chip caches and memory controllers.

Table 2 clearly shows that the memory subsystem is crucial for efficient performance of lock operations. The Chalice CATS, Digital DNARD and Netwinder systems which have an ARM processor show significant performance improvement with restartable atomic sequences. This processor appears to be very sensitive to memory performance. However, newer processors in the ARM family such as the Xscale show the memory subsystem has been significantly improved so that inlined memory-interlocked instructions outperform restartable atomic

sequences. For the Xscale case, restartable atomic sequences are paying the penalty of being non-inlined. On this particular Xscale system, the memory controller is integrated into the CPU, the memory the that memory-interlocked instruction is manipulating is cacheable, and so the memory-interlocked instruction is inexpensive.

The systems based on the Alpha processor also show significant performance improvement with restartable atomic sequences. These systems generally have small performance indices for the inlined memory-interlocked instructions; an indication that the cost associated with function invocation is negligible. The AlphaServer 1200 is a multiprocessor system, so the hardware causes significant extra bus activity. However, the benchmark was run with a uniprocessor kernel.

The microbenchmark results presented in Table 2 demonstrate that an architecture and memory system cannot necessarily provide efficient functionality compared with a combination of kernel and compiler optimisation. On modern processors, restartable atomic sequences can provide improved performance in atomic operations.

4.4 Run-time overhead of restartable atomic sequences

As mentioned in Section 2, there is a run-time overhead associated with checking if the program counter is within a restartable atomic sequence on every context switch. The context-switch time was measured to obtain an indication of the cost of checking the program counter. The context-switch time was determined by measuring the time it takes a token to be passed through a pipe between two processes. The token is passed between the two processes twenty times for a single measurement. Of 16000 measurements, the shortest time was chosen as the true context-switch measurement, since it most-likely represents the uninterrupted time to perform the context switch. Context-switch times are measured for increasing number of registered atomic sequences and are shown for the DECstation 5000/25 and 100MHz i486DX in Figure 3.

Since the restartable atomic sequences are recorded in a linked list for each process the context-switch times increase linearly with increasing number of registered atomic sequences. By one-hundred registered atomic sequences the context-switch time for the DECstation 5000/25 has doubled. About 130 registered atomic sequences are required to double the context-switch time on the 100MHz i486DX.

For a small number of sequences in the list, the performance is not likely to be a significant issue. Nevertheless, one way to improve the performance might be to order the list according to the start address. Then the `ras_lookup()` function could quickly abort the search when it finds a sequence with a start address higher than the program counter. Similarly, the first and last addresses of all restartable atomic sequences could be recorded in a header which would allow the `ras_lookup()` function to quickly check if it is necessary to traverse the list. Realistically, applications will not make use of so many restartable atomic sequences. Indeed, the current implementation places an arbitrary restriction of sixteen sequences per process. Since the threads library is currently the only user of restartable atomic sequences, only one restartable atomic sequence is expected to be registered for an application.

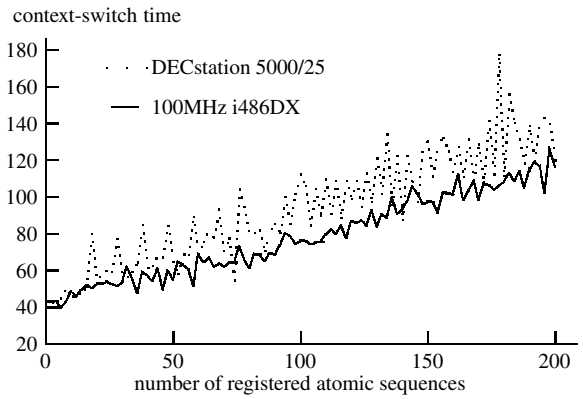

Figure 3: Context-switch time (milliseconds) for DECstation 5000/25 and 100MHz i486DX with increasing number of registered atomic sequences.

5 Discussion

Based on the performance results presented in Section 4, restartable atomic sequences is the default default mechanism for implementing atomic operations in the threads library on the NetBSD operating system. The mechanism provides the foundation for the implementation of a POSIX-compliant mutual-exclusion facility and the primitives for mutual exclusion in the library scheduler.

The threads library uses a *blocking* construct in the mutual-exclusion facility. This facility provides the `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions. The implementation uses a test-and-set atomic operation to test the ownership of a mutual-exclusion flag (mutex). If a mutex is owned by another thread, the scheduler blocks the current thread and switches another thread onto the available execution context. Eventually a thread will release ownership of the mutex and any blocked threads waiting on the mutex will be given the execution context.

The blocking construct within the threads library uses a single test-and-set atomic operation and therefore uses

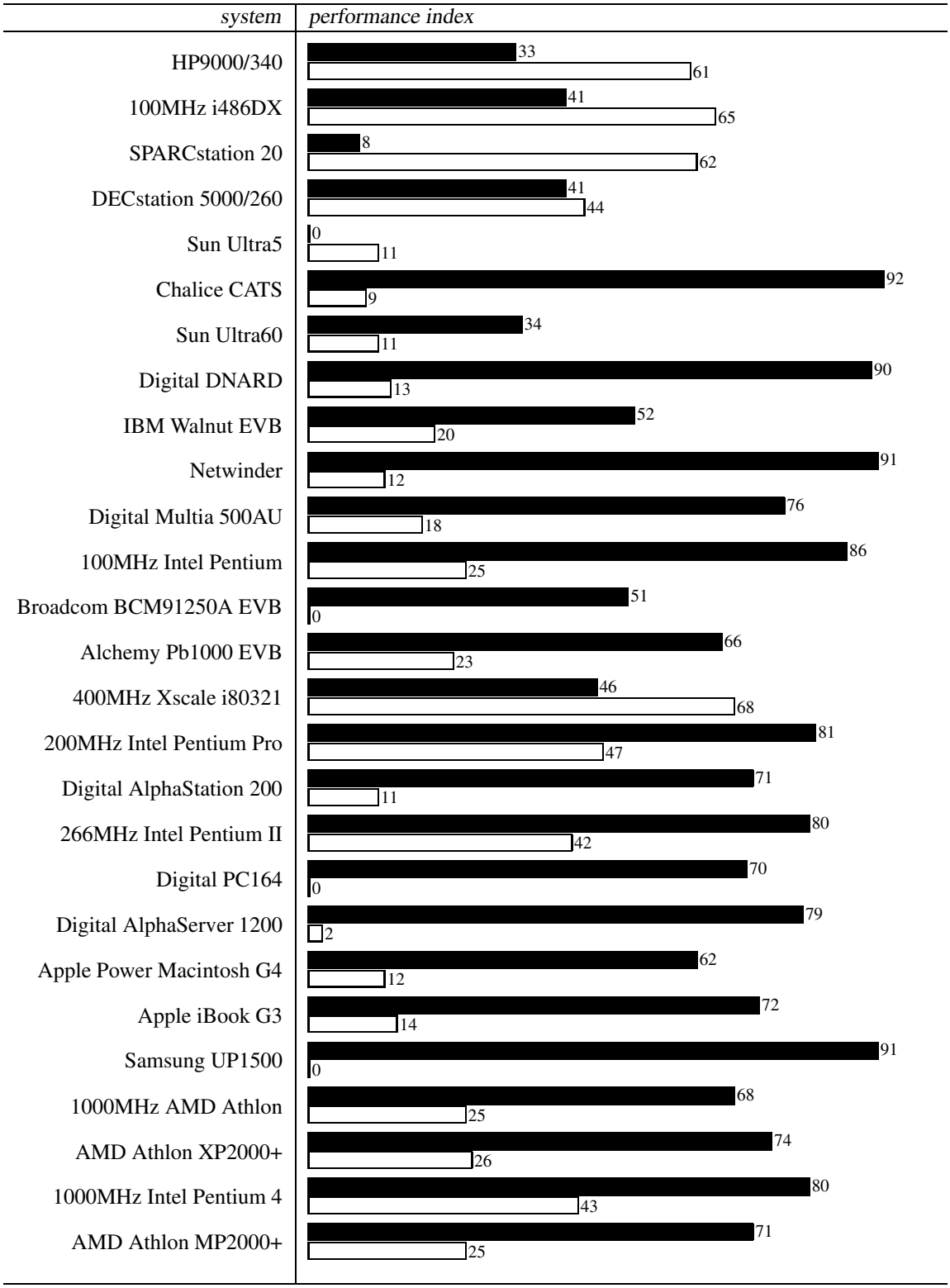

Table 2: Performance index of a test-and-set atomic operation based on restartable atomic sequences (black) and inlined memory-interlocked instructions (white). Larger values of the performance index indicate improved performance over non-inlined memory-interlocked instructions.

a single restartable atomic sequence. When the threads library is loaded, an initialisation function invokes the *rasctl* system call to register the restartable atomic sequence of the test-and-set atomic operation. Since the library must support atomic operations transparently on both uniprocessor and multiprocessor systems, the test-and-set atomic operation invokes the underlying test-and-set mechanism through function pointers. The initialisation function checks for a multiprocessor system with the `hw.ncpu` *sysctl* variable. On a multiprocessor system the test-and-set atomic operation uses memory-interlocked instructions.

The use of function pointers to choose the underlying atomic mechanism introduces additional execution cost in the mutual-exclusion facility. It also means that the atomic locks are no longer inlined, and the performance of memory-interlocked instructions is relegated to the non-inlined case considered in Section 4.

The thread library introduces additional overhead to the mutual-exclusion facility and the full performance demonstrated in Table 2 is not attainable. For comparison, a microbenchmark similar to the one used in Section 4 was developed which invoked `pthread_mutex_lock()` and `pthread_mutex_unlock()` in a loop one million times. The benchmark uses a single thread so that no contention occurs. On a 1000MHz AMD Athlon processor, the benchmark of a test-and-set restartable atomic sequence completes in 75 milliseconds. On the same processor, the mutex operation (using restartable atomic sequences) completes in 125 milliseconds. Therefore, the mutual-exclusion facility of the threads library introduces 66% overhead. Nevertheless, restartable atomic sequences improve the performance of the mutex functions, where the mutex benchmark using non-inlined memory-interlocked instructions takes 135 milliseconds to complete. From this result, it can be seen that the performance improvement of restartable atomic sequences propagates directly into the mutual-exclusion facility of the threads library.

However, these microbenchmarks do not represent typical usage in multi-threaded applications. A microbenchmark based on a row-parallel LU matrix decomposition provides a complement of computation and mutex contention. An LU decomposition on a 1000×1000 matrix uses around 14,000 test-and-set atomic operations per thread. Running the benchmark on a 1000MHz AMD Athlon processor using from one to one-hundred threads, there was no statistical difference between restartable atomic sequences and memory-interlocked instructions (p-value 0.7). Similarly, a benchmark of the multi-threaded apache web server (version 2.0.44) using the *httpperf* HTTP performance measurement tool (version 0.8) also showed no statistical difference in either the request rate or data-transfer rate. Therefore, it may be concluded, from a performance perspective, that most applications will not be adversely affected by the implementation of restartable atomic sequences.

For processors which do not provide memory-interlocked instructions, restartable atomic sequences do provide a significant benefit. In the future this benefit may become more significant if restartable atomic sequences are extended for use within the NetBSD kernel. To realise improved performance on multiprocessor systems and attain targets for real-time latencies, a preemptable NetBSD kernel would place increased demand on atomic operations within the kernel. While much of the design decisions discussed in Section 3 are readily extended to a kernel-level implementation, actioning restarts only on context switches is likely to be insufficient. Interrupts must also action restarts. Consequently, a kernel-level implementation of restartable atomic sequences will require many more invasive changes to machine-dependent subsystems. The lessons learned from this implementation of user-level restartable atomic sequences serves as a good starting point.

6 Conclusion

Restartable atomic sequences have been implemented on the NetBSD operating system to provide a generic framework for atomic operations for use by the POSIX threads library. Restartable atomic sequences are appropriate for uniprocessor systems that do not support memory-interlocked instructions. Moreover, on modern processors that do have hardware support for synchronisation, better performance may be possible with restartable atomic sequences.

This paper has presented an overview of an implementation of user-level restartable atomic sequences on the NetBSD operating system and discussed design decisions encountered during its implementation. Performance comparisons between restartable atomic sequences, a syscall-based mechanism and instruction emulation for mutual exclusion demonstrated the advantages of restartable atomic sequences.

Availability

The kernel and user implementation discussed in this paper has been adopted by the NetBSD Project and is currently available under a BSD license from the NetBSD Project's source servers. A complete set of regression tools for memory-interlocked instructions and restartable atomic sequences is available within the source tree. The next formal release which will use the implementation will be NetBSD 2.0. Further information about the NetBSD Project can be found on the

Project's web server at `www.netbsd.org`.

Acknowledgements

Thanks to Artem Belevich, Allen Briggs, Simon Burge, Martin Husemann, Jason Thorpe, Valeriy Ushakov, Martin Weber, Nathan Williams, and Berndt Wulf for the performance data presented in Tables 1 and 2.

References

[1] 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. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] B. N. Bershad, D. R. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *Fifth Symposium on Architectural Support for Programming Languages and Operating Systems*, 1992.

[3] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[4] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The design and implementation of the 4.4BSD operating system.* Addison-Wesley, 1996.

[5] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[6] N. J. Williams. An implementation of scheduler activations on the NetBSD operating system. In *Proceedings of the 2002 Usenix Annual Technical Conference*, 2002.