Performance Measurements of a Multiprocessor Sprite Kernel

John H. Hartman
John K. Ousterhout

University of California at Berkeley
Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
{jhh,ouster}@sprite.Berkeley.EDU

ABSTRACT

This report presents performance measurements made of the Sprite operating system running on a multiprocessor. A variety of micro- and macro-benchmarks were run while varying the number of processors in the system, and both the elapsed time and the contention for kernel locks were recorded. A number of interesting conclusions are drawn from the results. First, the macro-benchmarks show acceptable performance on systems of up to five processors. Total system throughput increases almost linearly with the system size. Projections of the lock contention measurements show that the maximum performance will be reached with about seven processors in the system. Second, it is often difficult to predict the effect of a benchmark on particular kernel locks. It was anticipated that different benchmarks would saturate different kernel monitor locks. After running the benchmarks it was found that a single master lock was the biggest kernel bottleneck, and that one of the micro-benchmarks had saturated a different lock than the one at which it was targeted. The kernel locking structure has become so complex as the system has evolved that it is hard to determine cause and effect relationships. Third, although the kernel contains many locks, only a few of them are performance bottlenecks. Performance measurements such as those presented here allow the relevant parts of the kernel to be redesigned to eliminate the bottlenecks. Such a redesign is needed to allow the system to scale gracefully beyond about seven processors.

1. Introduction

Sprite is a network operating system being developed at Berkeley [7]. From its inception Sprite has been designed to run on a multiprocessor. To avoid performance bottlenecks due to kernel contention, the kernel is multi-threaded to allow more than one processor to execute kernel code at the same time. Exclusive access to kernel resources is ensured through the use of locks. There is a limit to the number of processors that can be in the kernel and doing useful work, however. Once a lock saturates, additional processors will not significantly increase system throughput. The goal of our study was to determine how well Sprite scales with the size of a multiprocessor, by running a variety of benchmarks and measuring both contention for kernel locks and overall system performance.

The benchmarks were chosen either to stress particular kernel locks or to resemble user workloads seen in the Sprite development environment. The former were used to measure how well the system behaves when locks became saturated, and to identify locks that are potential bottlenecks. This information can then be used to restructure the kernel to improve its behavior. The latter were used to measure the

The work described here was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-R-0269 and in part by the National Science Foundation under grant ECS-8351961.

system's ability to scale while running realistic workloads, determining both the limit on system size and the effect of adding another processor to the system.

The results indicate that contention for the kernel context switch code is the biggest limiting factor to scaling the system. The lock protecting this code becomes very heavily utilized with only five processors in the system. Measurements of the realistic workloads indicate that the lock will become saturated with about seven processors in the system.

The rest of the paper is structured as follows. Section 2 describes the types of locks used in the Sprite kernel, and Section 3 describes the multiprocessor hardware used to obtain the measurements. The instrumentation added to the Sprite kernel is outlined in Section 4, and a description of the benchmarks is in Section 5. The results are in Section 6, followed by comments in Section 7 and concluding remarks in Section 8.

2. Kernel locks

In order for a multi-threaded kernel to function correctly it must contain mechanisms for providing both mutual exclusion and synchronization between threads. Other efforts to design multiprocessor operating systems have used semaphores [1, 3, 8]. Semaphores are appealing because they can provide both mutual exclusion and synchronization, eliminating the need for separate mechanisms. Sprite, however, uses monitor-style locking and condition variables to provide these services.

There are two basic types of locks in the Sprite kernel: *monitor* locks and *master* locks. Monitor locks are used to implement monitors [5], with semantics similar to those in Mesa [6]. Monitor locks are acquired at the start of a procedure and released at the end. If a process tries to lock a monitor lock and another process has it locked already, then the process is put to sleep. The release of a monitor lock causes all processes that are waiting on the lock to be awakened and simultaneously try to reacquire the lock.

The other type of lock in the Sprite kernel is the *master* lock. Master locks are used in much the same manner as monitor locks, except that they are used to to provide mutual exclusion between processes and interrupt handlers. A master lock is simply a spin lock that is acquired with interrupts disabled. If a master lock is already in use when an attempt is made to grab it, then the processor retries the locking operation until it succeeds. Interrupts are disabled to prevent a situation where an interrupt is taken after a master lock has been acquired and the interrupt routine spins forever waiting for the lock to be released.

Locks have three different styles of usage in the Sprite kernel. These three styles correspond to different locking granularities. Fine-grained locks allow a high degree of concurrency, but they increase the number of locks that a particular thread must acquire, thereby decreasing its performance. Coarse-grained locks reduce the amount of concurrency, but improve the performance of a single thread. The trick in designing the kernel is deciding the placement and granularity of locks.

The coarsest granularity locks in the kernel are single locks that are used to protect sections of code. Locks used in this manner are referred to as *code locks*. If the lock is a monitor lock then the resulting construction is similar to the *monitored modules* described in [6]. An example of this style of use is the monitor lock around the Sprite virtual memory system. There is a single monitor lock that must be grabbed whenever a routine in the virtual memory system is called, thus synchronizing access to the virtual memory system as a whole.

The remaining two styles of usage are variations on a theme. They both associate locks with data rather than with sections of code. For this reason they are referred to as *data locks*. Data locks that use a monitor lock are referred to as *monitored objects* in the Mesa paper. One style of usage associates a lock with a data structure. In order to manipulate the data structure the lock must be held. A lock that protects a queue is an example. The Sprite file cache has a single lock that protects the various lists of cache blocks, such as the free list, dirty list, etc. In order to modify any of these lists the file cache lock must be held.

The finest granularity locks are associated with individual data objects. A processor must hold this lock before modifying the contents of the object. For example, each block in the Sprite file cache has a lock, and that lock must be held when accessing the contents of the block.

In addition to ensuring mutual exclusion between threads, Sprite must also provide a means for threads to wait for interesting conditions to occur. *Condition variables* are used for this purpose. If a process waits on a condition variable while holding a lock it will release the lock and be put to sleep. At a later time another process will signal the condition variable, causing all processes waiting on the variable to awake and try to reacquire the lock. This signaling mechanism has the same semantics as Mesa's *broadcast* facility.

Major Locks in the Sprite Kernel		
Name	Type	Description
sched_Mutex	Master	controls access to context switch code and run queue
handleTableLock	Monitor (code)	controls access to table of all known file handles
vmMonitorLock	Monitor (code)	lock around all virtual memory functions
perPCBLock	Monitor (data)	must be held when accessing contents of a process control block
pdevLock	Monitor (data)	controls access to individual pseudo-device handles
exitLock	Monitor (code)	provides exclusive access to code for destroying a process

Figure 1. Some of the major locks in the Sprite kernel. Monitor lock types are qualified by a designation in parentheses. *Code* indicates that there is a single lock protecting a critical section of code. *Data* designates those locks for which there is one lock per object. Handles are Sprite's equivalent to Unix's inodes. Pseudo-devices are explained later in this paper.

Figure 1 is a list of the major locks in the Sprite kernel. There are many other locks in the kernel, such as locks associated with the file system cache, system timers, and the RPC system, but none of these have a significant impact on system performance.

3. The SPUR Hardware

This section outlines some of the features of the hardware used in the study. Sprite does not require any special hardware support for running on a shared-memory multiprocessor, other than an atomic test-and-set operation and coherent processor caches. Details of the hardware are only provided to allow comparisons to be made to more familiar machines.

At the time of this study the only multiprocessor that Sprite supported was the SPUR multiprocessor [4]. SPUR is a RISC microprocessor developed at Berkeley as part of a project to study the impact of adding symbolic processing support and multiprocessing to RISC architectures. Individual SPUR processor boards can be combined to form a shared-memory multiprocessor. The machine used in this study has 32 Mb of shared memory and up to five processors. Each SPUR processor board has a 128-kbyte data cache that is kept consistent by hardware.

The SPUR prototype used in this study is not a particularly fast machine. The processor cycle time is 140 ns. Due to a design error the on-chip instruction buffer is not functional, causing instruction fetches to require several cycles. As a result, a SPUR processor can execute about 1.5 native MIPS. Furthermore, a lack of compiler optimization leads to inefficient code. We estimate that the resulting performance is equivalent to about 0.5 VAX MIPS.

4. Instrumentation

Contention for kernel resources was measured by collecting information on lock behavior. We added fields to each lock to record the number of successful lock acquisitions (referred to as *hits*), and the number of unsuccessful attempts to acquire the lock (*misses*). For some types of locks, such as the locks associated with process control blocks, the statistics for an individual lock aren't as useful as those for the type as a whole. For this reason the individual lock counts were consolidated and recorded on a per-type basis. Recording the information on a per-type basis also makes it easy to handle locks that are created and destroyed dynamically. The lock counts for these transient locks are added to the total for the type when the lock is destroyed.

The definition of what constitutes a lock miss is different for monitor locks and master locks. For master locks there is at most one miss per hit. If a process misses on a lock it spins until the lock is free, then locks it. This sequence is counted as one miss, followed by a hit. On the other hand, it is possible for monitor locks to have more misses than hits. If multiple processes are waiting when a lock is released then they will all be awakened and will attempt to grab the lock. Only one will succeed and the rest will go back to sleep. If a process is awakened in this fashion and does not get the lock a miss is recorded. This makes it possible for the number of misses on a monitor lock to be greater than the number of hits.

In addition to instrumenting the locks we also created new system calls to clear the lock information and to copy the information to user-level. These two calls were used to reset the lock information at the beginning of a test and gather the information when it completed.

5. Benchmarks

All of the benchmarks are intended to stress the operating system. Compute-bound applications were avoided for that reason. The benchmarks can be divided into two classes. Realistic workloads are represented by the *macro-benchmarks*. These benchmarks are comprised of real programs that represent the Sprite development environment. The behavior of the kernel while running the macro-benchmarks is indicative of its behavior under real workloads, and allows projections to be made of maximum system size.

Individual parts of the kernel were stressed by running a series of *micro-benchmarks*. For example, a micro-benchmark may consist of repeated forking of children, or repeated message passing between processes. Such repetitive behavior is rarely seen in real programs, but it does allow the behavior of different parts of the kernel to be isolated and measured.

5.1. Macro-benchmarks

5.1.1. PmakeInd

The *pmakeInd* benchmark is intended to be representative of a multi-user environment with multiple independent compilations taking place. *PmakeInd* recompiles Csh from its sources using the *Pmake* program. Pmake is similar to the UNIX† "make" utility, except that it runs the compilations in parallel whenever possible [2]. The *pmakeInd* benchmark runs a separate instance of Pmake on each processor and each instance uses separate copies of the sources. This eliminates the Pmake program and contention for the source files as causes of performance degradation. Each instance of Pmake runs two compilations concurrently, ensuring that each processor remains highly utilized.

5.1.2. Pmake

The *pmake* benchmark is also a compilation of the Csh sources, except that only one instance of Pmake is run, rather than one per processor. Pmake attempts to use all of the processors in the system by running two compilations per processor. The purpose in running this benchmark is to see how well a single parallel application can harness the processing power available in a multiprocessor. Ideally the application will realize linear speedup as the number of processors is increased. A speedup that is less than linear is due to either contention in the kernel or lack of concurrency in the application. Since we are primarily interested in the former effect rather than the latter, the final link of the Csh binary was eliminated from the benchmark. This increased the concurrency inherent in the computation and increased contention for kernel resources.

† UNIX is a trademark of Bell Laboratories.

5.1.3. Troff

The *Troff* benchmark consists of running the text formatting program Troff on the man page for Csh. The resulting output is sent to /dev/null. Each processor runs a different instance of Troff.

5.2. Micro-benchmarks

Twelve micro-benchmarks were run, of which only six are discussed in this paper. The other six pertained to stressing the file system, by reading files, opening files, etc. The results obtained were not significantly different from the first six and are therefore omitted in the interest of brevity.

5.2.1. Fork

The *fork* benchmark consists of a process that repeatedly forks off a child process using the *fork()* system call. The parent waits for the child to die before forking another child. The child process exits immediately. This benchmark should stress the process creation and destruction components, and the kernel virtual memory system. The context-switch code will also be heavily utilized.

5.2.2. Fexec

The *fexec* benchmark is similar to the *fork* benchmark. A parent process repeatedly forks off children, waiting for each one to die before forking the next. The child process uses the *exec()* routine to create another instance of itself instead of exiting. This new instance then exits, allowing the parent to continue. The *fexec* benchmark should stress the same components as the *fork* benchmarks. Any differences in behavior are due to the call to *exec()*.

5.2.3. Mem

Mem is a benchmark that stresses the virtual memory system. The process repeatedly increases the size of its heap using the *sbrk()* system call. The new heap pages are not touched. Once the process reaches a maximum size it execs itself and starts over. *Mem* is targeted at the virtual memory system, particularly the component that is responsible for maintaining process page tables.

5.2.4. Cswitch

The *cswitch* benchmark consists of two processes that pass a byte back and forth using a pair of pipes. If this pair of processes is run on a single processor, then two context switches are required per round trip and the context switch code should receive heavy utilization. The context switches can be avoided on a multiprocessor if the processes are run on different processors. In order to ensure that the context switches occur one pair of processes is run per processor in the system.

5.2.5. Pdevtest

Pdevtest stresses the pseudo-device implementation. Pseudo-devices are like devices except that they are implemented by user-level server processes [9]. They provide a mechanism for allowing trusted services, such as display servers and network communication protocols, to be implemented at user-level rather than in the kernel. A client process accesses a pseudo-device in the same manner as a real device. The kernel then forwards the request to the user-level server, instead of to a kernel device driver. Pseudo-devices are similar to the UNIX "pty" mechanism, the only difference being that all operations on a pseudo-device, such as open, close, and ioctls, are passed through to the server process.

The *pdevtest* benchmark creates one pseudo-device that is written to by multiple clients. Each client repeatedly writes one byte to the server.

5.2.6. PdevtestInd

This variation on the *pdevtest* benchmark creates multiple pseudo-devices. One server and one client is created for each instance of the benchmark that is run (i.e for each processor in the system). The intent is to see if there are any performance bottlenecks in the pseudo-device implementation that occur when there is a single server vs. multiple servers.

6. Results

The results are divided up into sections on elapsed time for running the benchmarks, calculations of the incremental throughput for each additional processor, monitor lock behavior, and master lock behavior. These measurements indicate that the micro-benchmarks suffer heavily from saturation of critical resources. The macro-benchmarks also experience performance degradation, but to a lesser degree.

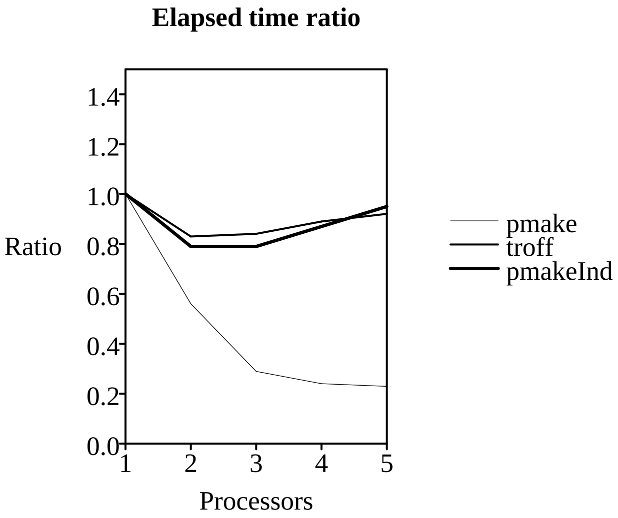

Figure 2. A graph of the ratio of the multiprocessor elapsed time to run the macro-benchmarks to the uniprocessor elapsed time. The workload was scaled with the system size (except for the *pmake* benchmark), causing the ratio for an ideal system to be a horizontal line at 1.0. A ratio that is greater than 1.0 indicates that the increase in system throughput is less than linear. The *pmake* benchmark has a fixed workload, hence its ideal curve should be $1/n$, where n is the number of processors.

6.1. Elapsed time

The elapsed time ratio for the macro-benchmarks is shown in Figure 2. The ratio is calculated by dividing the time it takes a system with n processors to finish n benchmarks by the time it takes a uniprocessor to finish one benchmark. For all but the *pmake* benchmark the workload is scaled with the number of processors, so an ideal system would have a constant ratio of 1.0. The curve for the *pmake* benchmark looks different because the workload is not scaled with the number of processors. The compilation of the Csh sources always runs faster with more processors. The ideal elapsed time ratio for the *pmake* benchmark is $1/n$, where n is the number of processors in the system. The measured ratio for the *pmake* benchmark is very close to the ideal, falling behind only when there are five processors in the system.

The *troff* and *pmakeInd* benchmarks have elapsed time ratios that are actually better (i.e. less) than the ideal value of 1.0. This is because there is a certain amount of background processing that must be done that is independent of system size. This extra work is due to other processes running on the system and interrupts. As the number of processors in the system is increased, the per-processor load induced by background processing is decreased, allowing each processor to allocate more cycles to the benchmark. While running the *troff* and *pmakeInd* benchmarks the slowdown from kernel contention was more than offset by the amortization of the background processing.

Figure 3 shows the elapsed time ratio for the micro-benchmarks. Contention causes the elapsed time of all the benchmarks to increase over the range of system sizes tested. For two of the benchmarks the elapsed time initially decreases due to amortization of background processing costs, but this benefit is eventually outweighed by contention for kernel resources.

The *cswitch* benchmark has a worse than linear slowdown: a system with five processors actually takes longer to complete the work than a uniprocessor should. None of the other benchmarks do this badly, but neither do they come close to the ideal ratio. All of the benchmarks show a steady increase in the elapsed time ratio once there are three processors in the system. This suggests that the micro-benchmarks are severely affected by contention for kernel resources and the system throughput does not increase

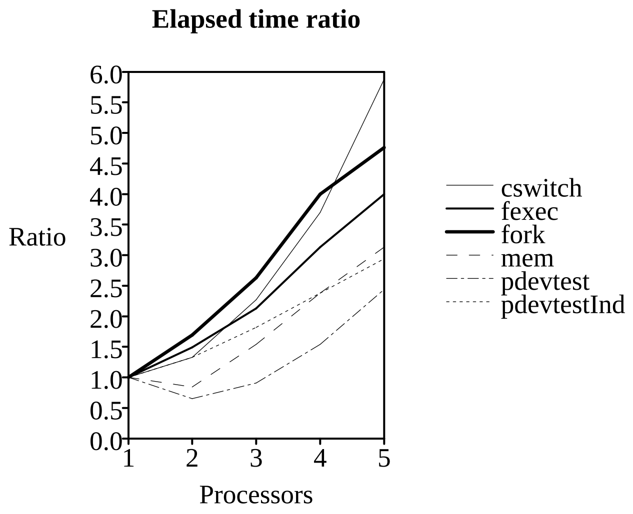

Figure 3. This is a plot of the ratio of the multiprocessor elapsed time to run the micro-benchmarks to the uniprocessor elapsed time. As in Figure 2 an ideal system would have a ratio of 1.0. All of the benchmarks exhibit an increase in the elapsed time as the system size increases.

significantly if the system is scaled beyond three processors.

6.2. Incremental throughput per additional processor

The incremental throughput of an additional processor is the net amount of work per unit time that the processor adds to the system. Throughput is measured in *processor units*, which is the amount of work per unit time done by a uniprocessor. Incremental throughput is derived by taking the throughput with n processors in the system, and subtracting the throughput with n - 1 processors. The result is then normalized to the throughput of a uniprocessor to obtain the incremental throughput in processor units.

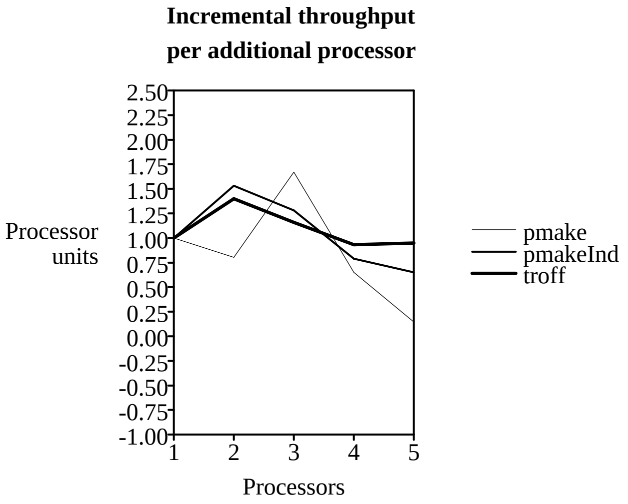

Figure 4. Incremental throughput per additional processor while running the macro-benchmarks. Two of the benchmarks have almost constant incremental throughput, indicating that the total system throughput is proportional to the size of the system.

Figure 4 shows the incremental throughput per processor for the macro-benchmarks. The independent compilation of Csh (*pmakeInd*) and the troff of the Csh man page (*troff*) curves both display a gain of more than one processor unit when the second processor is added. As mentioned previously, this is due to amortization of background processing. The *troff* benchmark does the best of all, maintaining an incremental gain of almost one processor unit even for the fifth processor.

The *pmake* benchmark shows a gain of more than one processor unit of throughput for the third processor, followed by decreasing gains for subsequent processors. Some of this effect is probably due to kernel contention, although the lock miss ratios aren't high enough to account for all of it. The rest is probably due to a lack of concurrency in the *Pmake* program. This is interesting because it suggests that although there are five processors in the system, *Pmake* cannot make effective use of them.

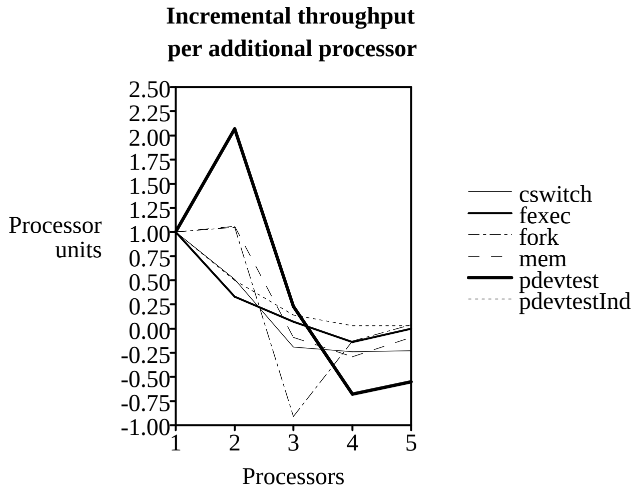

Figure 5. The incremental throughput added to the system by each processor, in processor units. These measurements were taken while running the micro-benchmarks. All benchmarks show very little throughput gained by adding additional processors once there are three processors in the system.

The micro-benchmarks all suffer from severe degradation in elapsed time as the number of processors is increased, therefore we would expect the incremental throughput per processor to diminish as well. This result is seen in Figure 5. The throughput gained by adding additional processors drops off rapidly so that the third processor does not add much, if any, processing power to the system. For a few of the benchmarks the third processor actually produces negative processor units of throughput, in effect slowing the system down. At this point the scheduler lock has become saturated, preventing additional processors from doing anything useful.

6.3. Monitor lock measurements

The graphs of the macro-benchmark monitor lock miss ratios are shown in Figure 6. The miss ratio of a lock is the number of misses on that lock divided by the number of hits. Contention for monitor locks does not appear to be a major performance bottleneck for a system with five or fewer processors. The maximum miss ratio for any benchmark is less than 18%. Extrapolation of the curves indicates that monitor lock contention should not reach saturation levels until the system is scaled by a factor of three or four.

The highest monitor lock miss ratio occurs on the same lock for all the benchmarks. The kernel has a single monitor lock, vmMonitorLock, that surrounds the entire virtual memory system. Any routines that modify the virtual memory state must hold this monitor lock. When the virtual memory system was written the emphasis was on correctness, rather than concurrency, hence vmMonitorLock's monolithic nature. However, the benchmarks suggest that system performance on large multiprocessors could be improved by replacing the single vmMonitorLock with several locks on individual data structures.

The monitor lock miss ratios while running the micro-benchmarks are shown in Figure 7. Most of the curves have small slopes and small maximum values. Exceptions are the *pdevtest* and *cswitch* benchmarks. The causes of the contention can be found by examining the kernel code. The *pdevtest* benchmark spends a lot of its time copying the data from the client's address space to the server's. A monitor lock associated with the server process (perPCBLock) is held during the copy. If there are multiple clients then this lock is a critical resource. With more than three clients the lock is held all of the time, causing the miss ratio to reach almost two hundred percent. Each time a process releases the lock more than one process is awakened causing more total misses than hits.

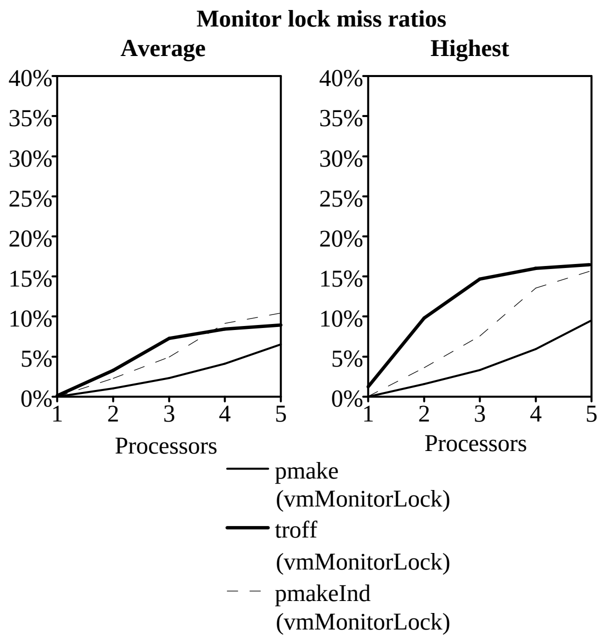

Figure 6. Graphs of the monitor lock miss ratios while running the macro-benchmarks. The graph on the left is the miss ratio averaged across all monitor locks. The graph on the right is the miss ratio of the lock with the highest miss ratio. The name of the lock with the highest miss ratio is displayed in parentheses under the name of the benchmark.

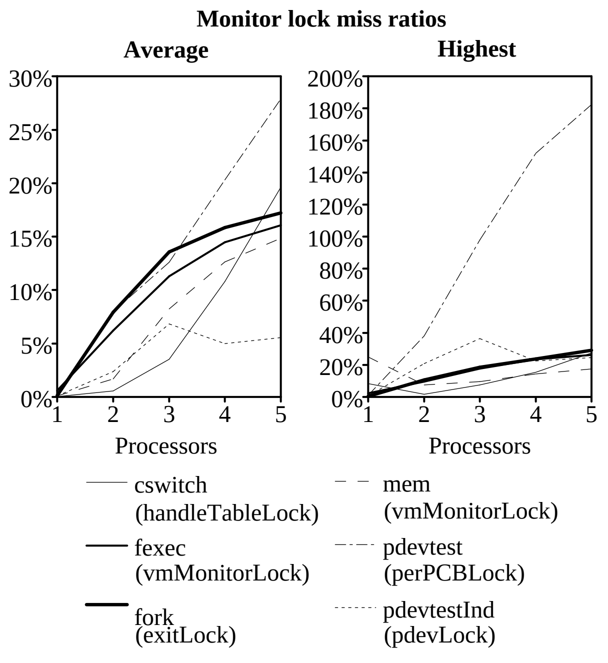

Figure 7. The average and highest monitor lock miss ratios while running the micro-benchmarks. Note that the vertical scales on the two graphs are not the same. Benchmarks whose curves have a steep slope will not scale well to larger systems due to saturation of a monitor lock. The initially high values for some benchmarks on a one processor system are due to all of the benchmark processes exiting at once, causing a high miss ratio although the absolute number of hits and misses is very low.

The *cswitch* benchmark uses pipes to pass a byte between processes. Each time a pipe is accessed a monitor lock around the file handle table (handleTableLock) in the kernel is grabbed. Although different pairs of processes use different pipes, they all need to grab the handle table monitor lock, causing a bottleneck. This is a surprising result, since the *cswitch* benchmark was intended to stress the context switch code. It is not always easy to predict what effect a particular kernel lock will have on a benchmark's performance.

Two other locks show up in the graph of the highest monitor lock miss ratio. ExitLock and pdevLock have the highest miss ratio of any monitor lock while running the *fork* and *pdevtestInd* benchmarks, respectively. The miss ratios are not high enough, however, to warrant replacing each of these locks with multiple locks unless there are many more processors in the system.

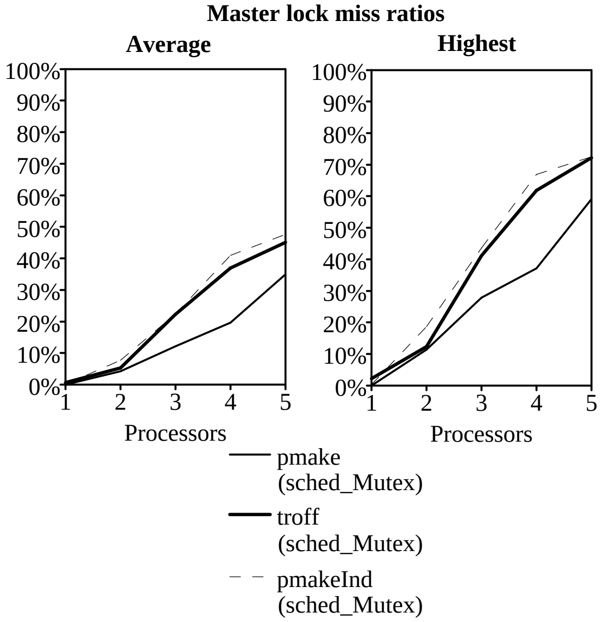

Figure 8. Graphs of the master lock miss ratios while running the macro-benchmarks. All of the curves have a steep slope due to contention for the scheduler lock.

6.4. Master lock measurements

The graphs in Figure 8 are plots of the master lock miss ratios for the macro-benchmarks. All of the curves show increasing amounts of contention as the size of the system is increased. All of this contention is due to sched_Mutex, the master lock around the scheduler. Sched_Mutex has a higher miss ratio than any other lock in the kernel for almost all of the benchmarks. This is surprising, since our first intuition was that various monitor locks would saturate first. It turns out that sched_Mutex is used in many different places in the kernel. Its main function is to provide mutually exclusive access to the run queue, but it is also used for other purposes, including synchronization when a miss occurs on a monitor lock (see Figure 9). As a result, an increase in the miss rate for monitor locks will increase the contention for sched_Mutex.

The graphs in Figure 10 are plots of the master lock miss ratio for the micro-benchmarks. The slopes of the curves are quite steep -- all benchmarks have an average miss ratio on a five processor system that is between fifty and eighty-five percent. The highest miss ratio for any lock was greater than eighty percent for all benchmarks, and in some cases was higher than ninety percent. Once again this contention is for sched_Mutex. Although the micro-benchmarks were targeted at an array of kernel monitor and master locks, they all piled up on the scheduler lock. Clearly the importance of this single lock must be reduced.

```
Miss on monitor lock.
Lock sched_Mutex.
Put process on wait queue for monitor lock.
Remove first process from ready queue.
Context switch to the ready process.
Release sched_Mutex.
```

Figure 9. This sequence of events occurs when a process misses on a monitor lock. Note that sched_Mutex is used to synchronize access to the queue of processes waiting for the monitor lock, as well as the ready queue. This unnecessarily increases the total length of time that sched_Mutex is held. The addition of individual master locks to synchronize access to the wait queue for each monitor lock would reduce the length of the critical section protected by sched_Mutex and reduce its utilization.

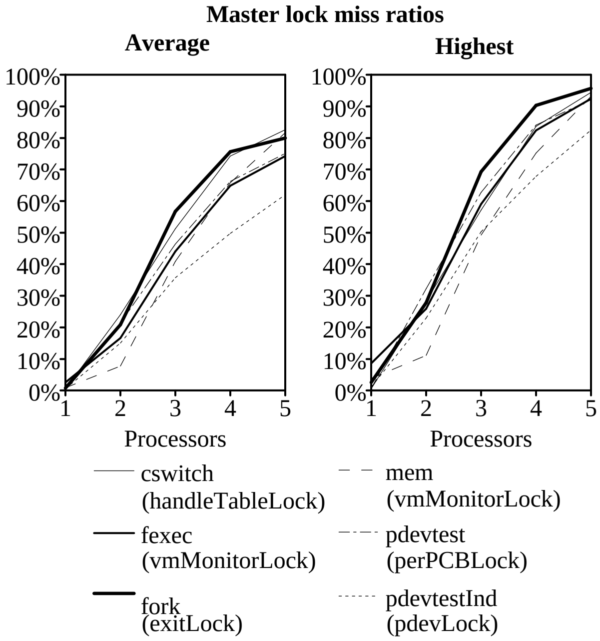

Figure 10. Graphs of the master lock miss ratio while running the micro-benchmarks. The steep slopes of the curves are due to a heavy contention for the master lock around the scheduler.

7. Comments

A running Sprite kernel contains anywhere from five hundred to a thousand locks. Of these locks, two repeatedly suffered more contention than the others. These two locks, sched_Mutex and vmMonitorLock, account for most of the contention for kernel locks and represent serious obstacles to better system performance. Both of these locks are code locks that protect large regions of the kernel code. In order to reduce their impact they must be replaced by several data locks with a finer granularity. It is not necessary to have single locks around the virtual memory system and the context switch mechanism. Replacing these locks with data locks on the various kernel structures will increase the concurrency and system performance.

There are a number of factors that cause system performance to depend heavily on contention for a few locks. The first is ease of design. It is much simpler to design a system with a few locks than it is to design one with many locks. Multiple locks may increase the concurrency, but they also increase the chance of introducing race conditions and deadlocks. A single lock should be replaced by multiple locks only if performance measurements indicate that the single lock is a bottleneck.

The second factor that causes performance-critical locks is that they develop during the evolution of the kernel. Race conditions and deadlocks tend to occur as new features are added to the kernel. When a kernel developer is faced with such an unwanted side-effect they typically rewrite their new feature to hold the most prominent lock they can find. In this manner prominent locks tend to become more important, until they are held in many places throughout the kernel for many different reasons. Such is the case with sched_Mutex. Its influence grew as the kernel was modified and synchronization problems arose.

The underlying cause of both the design and development problems is the lack of tools. The graph of lock dependencies in the Sprite kernel is fairly complex. Tools are needed to help understand the synchronization requirements of the various kernel resources and where to place locks to satisfy those requirements. Once the locks are in place, tools are needed to measure the performance of the locks in order to find the performance bottlenecks. Our measurements of kernel locks in Sprite found that it is not always obvious which locks will be bottlenecks and why.

8. Conclusion

The elapsed time and incremental throughput measurements for the macro-benchmarks indicate that the Sprite kernel gives acceptable performance on a machine with up to five processors. All of these benchmarks showed an almost linear speedup as the system was scaled, and two of them showed almost constant incremental throughput per processor. The performance degradation of the *pmake* benchmark was primarily due to sequential processing in the application, rather than kernel contention. This underscores the difficulty of writing a single application that can make full use of a multiprocessor's processing power.

Although the macro-benchmarks exhibited suitable performance increases for the system sizes that were measured, the sched_Mutex master lock approached saturation. With five processors in the system its miss ratio was close to seventy percent, indicating that the lock suffered very heavy contention. From this it would appear that without elimination of some of the more important bottlenecks the kernel will not support more than seven processors in the system efficiently, except for compute-bound applications.

Prior to undertaking this study we had assumed that the kernel monitor locks were the biggest performance bottlenecks. In particular, it was feared that the single monitor lock around the virtual memory system posed the biggest obstacle to scaling the number of processors in the system. It came a surprise that a master lock, sched_Mutex, was more heavily utilized than the virtual memory system lock. We were also surprised when micro-benchmarks stressed unintended locks. The behavior of kernel locks is difficult to predict, even to the people who designed the system. Clearly there is a need for tools to help system developers to better understand and modify the locking structures of multiprocessor operating systems.

9. Acknowledgements

We would like to thank Ken Lutz for putting together and maintaining the SPUR prototype, and Mendel Rosenblum for his efforts in getting both Sprite and SPUR to run reliably.

10. References

1. M. J. Bach and S. J. Buroff, Multiprocessor UNIX Operating Systems, *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1733-1749.

2. A. Boor, PMake -- A Tutorial, Unpublished, June 1, 1988.

3. E. W. Dijkstra,, Hierarchical Ordering of Sequential Processes., in *Operating Systems Techniques*, 1972, 72-93.

4. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, SPUR: A VLSI Multiprocessor Workstation, Computer Science Division Technical Report UCB/Computer Science Dpt. 86/273, December 1985.

5. C. A. R. Hoare, Monitors: An Operating System Structuring Concept, Vol. 17, October 1974.

6. B. W. Lampson and D. D. Redell, Experiences with Processes and Monitors in Mesa., *Communications of the ACM 23*, 2 (February, 1980), 105-117.

7. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

8. U. Sinkewicz, A Strategy for SMP Ultrix, *Usenix Conference Proceedings*, June 1988, 203-212.

9. B. B. Welch and J. K. Ousterhout, Pseudo-Devices: User-Level Extensions to the Sprite File System, *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.

John H. Hartman is a Ph.D candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He is currently a member of the Sprite network operating system project. His interests include operating systems, high-performance networks, and computer architecture. He received an Sc.B. in computer science from Brown University in 1987.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.