The following paper was originally published in the
Proceedings of the USENIX 1997 Annual Technical Conference
Anaheim, California, January 6-10, 1997

# A Revisitation of Kernel Synchronization Schemes

Christopher Small and Stephen Manley
Harvard University

# A Revisitation of Kernel Synchronization Schemes

Christopher Small and Stephen Manley
*Harvard University*
{chris,manley}@eecs.harvard.edu

## Abstract

In an operating system kernel, critical sections of code must be protected from interruption. This is traditionally accomplished by masking the set of interrupts whose handlers interfere with the correct operation of the critical section. Because it can be expensive to communicate with an off-chip interrupt controller, more complex optimistic techniques for masking interrupts have been proposed.

In this paper we present measurements of the behavior of the NetBSD 1.2 kernel, and use the measurements to explore the space of kernel synchronization schemes. We show that (a) most critical sections are very short, (b) very few are ever interrupted, (c) using the traditional synchronization technique, the synchronization cost is often higher than the time spent in the body of the critical section, and (d) under heavy load NetBSD 1.2 can spend 9% to 12% of its time in synchronization primitives.

The simplest scheme we examined, disabling all interrupts while in a critical section or interrupt handler, can lead to loss of data under heavy load. A more complex optimistic scheme functions correctly under the heavy workloads we tested and has very low overhead (at most 0.3%). Based on our measurements, we present a new model that offers the simplicity of the traditional scheme with the performance of the optimistic schemes.

Given the relative CPU, memory, and device performance of today's hardware, the newer techniques we examined have a much lower synchronization cost than the traditional technique. Under heavy load, such as that incurred by a web server, a system using these newer techniques will have noticeably better performance.

## 1 Introduction

Although the BSD kernel is traditionally single-threaded, it needs to perform synchronization in the face of asynchronous interrupts from devices. In addition, while processing an interrupt from a device, the kernel needs to block delivery of new interrupts from that device. This is traditionally accomplished by communicating with an off-chip interrupt controller and masking interrupts. Historically, the routines that perform this service are named spl*intr*, where *intr* is the name of the interrupt to disable. A special routine, *splhigh*, is used to disable all interrupts.

Most, if not all, conventional hardware platforms allow device interrupts to be prioritized and assigned levels; a pending interrupt from a lower-level device can not interrupt the processing of an interrupt from a higher-level device. This method protects critical sections (by disabling interrupts from all devices) and device drivers (by disabling interrupts from the same device). Interrupts can then be prioritized by their frequency and the time required to handle them.

At Harvard we are in the process of developing a new operating system, the VINO kernel [Selt96]. In order to explore the space of kernel synchronization methods, we constructed analytic models of four basic interrupt handling schemes. We then used the NetBSD 1.2 kernel to analyze the performance of each scheme, by measuring:

- how often NetBSD enters critical sections,
- how long critical sections last,
- how often they are interrupted,
- the frequency of different types of interrupts, and
- the frequency with which these interrupts are delivered.

We found that critical sections are generally very short (usually less than four microseconds), that little total time is spent in critical sections, and that the synchronization overhead of three of the four basic schemes we examined is nominal. We also found that the overhead of the traditional scheme can be as large as 12% of CPU time.

On many system architectures interrupt processing is handled by an off-chip programmable interrupt controller. On a Pentium PC the time required to access the off-chip interrupt controller is quite high, approximately three hundred cycles (2.5µs on a 120MHz Pentium).

In addition to the off-chip interrupt controller, some CPUs include an on-chip instruction to mask all interrupts. On the Pentium this operation can be performed in five to ten cycles, two orders of magnitude

This research was supported by Sun Microsystems Laboratories.

more quickly than communicating with the interrupt controller.

If a critical section is short, the cost of going off-chip to disable and re-enable interrupts can be much greater than the time spent in the body of the critical section. For example, if the critical section adds an element to a linked list, it may only run for twenty or thirty cycles; if the kernel needs to go off-chip to synchronize, the time spent in the synchronization code is an order of magnitude more than the time spent in the body of the critical section. If we use the on-chip instructions to disable and re-enable interrupts, the synchronization cost is lowered to the point where it is less than that of the critical section code.

Optimistic synchronization techniques are particularly well-suited to short-duration critical sections that are infrequently interrupted. On entry to a critical section, instead of disabling interrupts, a global flag is set. If an interrupt is delivered while the flag is set, interrupts are only then disabled, and the interrupt is postponed until the flag is cleared. If no interrupt is delivered while the flag is set, the cost of synchronization is just the cost of setting and clearing the flag.

In this paper we model four schemes, based on two variables: where to mask interrupts (on-chip or off-chip) and how to mask them (pessimistically or optimistically).

We note that the techniques that use on-chip interrupt masking for synchronization may not be usable on multiprocessor systems. In symmetric multiprocessor systems, any processor may enter a critical section. Hence, any processor must be able to disable interrupts delivered to all processors. On this type of hardware the synchronization scheme must communicate either with the other processors (directly or through memory), or with an off-processor interrupt controller; in either case, it needs to go off-chip. An asymmetric multiprocessor (where only one processor runs operating system kernel code, and hence only one processor receives interrupts and enters critical sections) would be able to take advantage of the on-chip techniques we propose.

In the remainder of the paper we examine our four schemes in detail and compare their costs. We use our results to derive a fifth scheme which synthesizes the benefits of the others.

In Section 2, we discuss related work on synchronization. In Section 3, we propose our four schemes for handling interrupts. In Section 4 we discuss our experimental setup, and then in Section 5 measure the overhead of the four schemes. Section 6 examines the behavioral correctness of the on-chip schemes we propose, measuring the frequency of, and amount of time spent in, interrupt handlers. In Section 7 we discuss our results and propose our new scheme, and conclude in Section 8.

## 2 Related Work

The interrupt handlers of real-time systems often run with interrupts disabled [Stan88], as in the V kernel [Berg86]. In this paper we explore similar techniques, using atomicity for synchronization, although we make no assumptions about real-time behavior of the system as a whole.

Synchronization does not require locking; other techniques have been proposed. For example, non-blocking synchronization [Herl90, Mass91, Green96], lock-free data structures [Wing92], and restartable atomic sequences [Bers92] are novel techniques proposed for synchronization. Some of these schemes require special hardware support, such as a *compare-and-swap* instruction, which is not available on all processors (e.g., the MIPS R3000 and the Intel i386). These techniques are designed to work in environments where it is expensive (or impossible) to disable interrupts, and work best in environments where contention is minimal. The cost and complexity of these schemes is higher than the on-chip schemes we propose, and our analysis shows that under typical loads, such complexity is unnecessary.

Similarly, Stodolsky et al. [Stod93] proposed decreasing the cost of synchronization by taking an *optimistic* approach. When the kernel makes a request to raise the global priority level, instead of communicating with the interrupt controller, the kernel saves the desired priority level in a variable. The critical section is then run, and on completion, the former priority level is restored. If no interrupts occurred during the critical section, the kernel was able to save the (high) cost of communicating with the interrupt controller, paying instead the (lower) cost of saving and restoring the logical priority level.

If an interrupt is delivered while the kernel is in a critical section, the interrupt handler compares the logical priority level with the interrupt priority level. If the interrupt is higher priority, it is handled immediately; if it is lower priority, the system queues the interrupt and falls back to the pessimistic scheme, setting the actual priority level to match the logical level. The cost of setting the global flag is quite low compared to the cost of communicating with the interrupt controller, so if the kernel is rarely interrupted while in a critical section the performance of the optimistic technique is superior to that of the standard pessimistic technique. In the case of a null RPC microbenchmark, they saw a 14% performance improvement.

Our work begins with the ideas developed by Stodolsky and adds a second dimension of comparison, the performance difference seen when masking interrupts on-chip and off-chip.

Mogul and Ramakrishnan [Mogul96] have explored a related issue. Interrupt processing can be split across multiple priority levels, a high-priority level that responds to the device interrupt and a low-priority level that processes the interrupt. When this is the case, it is possible for a flood of high priority device interrupts to cause the starvation of the low-priority processing. Mogul and Ramakrishnan refer to this state as *receiver livelock*. When the system reaches this state, their strategy is to disable interrupts and poll for new requests as the old ones are handled.

Their techniques are most appropriate in the face of *unsolicited I/O* from network and serial devices. The kernel has little or no control over how quickly data is sent to this type of device. Unlike a network device, a disk device only generates interrupts in response to a request from the kernel. If the kernel is receiving disk interrupts more quickly than it can process them, the kernel can reduce the load by queueing fewer disk requests. A natural feedback loop is present; if the kernel is bogged down handling old work, little new work will be started.

Mogul and Ramakrishnan's strategy is related to ours. They find that, at times, it is more efficient to ignore new interrupts in order to process the ones that have already arrived. However, they propose dynamically switching between two schemes based on the current system load; we propose choosing a single scheme for handling all interrupts. Their work is compatible with ours; none of the synchronization schemes we analyze preclude use of their techniques.

## 3 Synchronization Strategies

The first axis of our strategy space is a comparison of *optimistic* and *pessimistic* schemes, as studied by Stodolsky et al. The second axis of our space is a comparison of *off-chip* and *on-chip* interrupt management. This gives us four strategies to explore, as seen in Table 1.

In the following sections we describe each scheme in detail, sketch its synchronization functions, and discuss its costs and benefits.

| | **Pessimistic** | **Optimistic** |
|---|---|---|
| **Off-chip** | *spl-pessim* | *spl-optim* |
| **On-chip** | *cli-pessim* | *cli-optim* |

**Table 1: Synchronization Strategies Explored.** On the x86, NetBSD 1.2 uses *spl-optim*, the Linux 1.3.0 kernel uses *cli-pessim*, and BSD/OS 2.1 uses *spl-pessim.*

### 3.1 Spl-Pessim

The *spl-pessim* scheme is named after the kernel "set priority level" function *(spl)*, which was named after the PDP-11 instruction of the same name. When a critical section is entered, a subset of interrupts are disabled by communicating with the off-chip programmable interrupt controller (PIC).

```
crit_sec_enter()
    saved_mask = cur_mask
    PIC_mask(all)

crit_sec_leave()
    PIC_mask(saved_mask)

interrupt_handler(int intr_level)
    saved_mask = cur_mask
    PIC_mask(intr <= intr_level)
    handle interrupt
    PIC_mask(saved_mask)
```

The benefit of this scheme is fine-grained control of which interrupts are disabled. The cost is high per-critical section overhead.

### 3.2 Spl-Optim

When a critical section is entered a variable is set to the logical priority level; the variable is restored on exit from the critical section. When an interrupt is delivered, the logical priority level is checked; if the interrupt has been masked, the interrupt is queued. The hardware interrupt mask is then set to be the logical mask (by communicating with the interrupt controller).

```
crit_sec_enter()
    in_crit_sec = true

crit_sec_leave()
    if (any queued interrupts)
            handle queued interrupts
    in_crit_sec = false

interrupt_handler(int intr_level)
    if (in_crit_sec
        || intr_level < cur_level)
            queue interrupt
    else
            saved_mask = cur_mask
            PIC_mask(intr <= intr_level)
            handle interrupt
            PIC_mask(saved_mask)
            handle queued interrupts
```

The benefit of this scheme is low per-critical section overhead, and fine-grained control over which interrupts are disabled.

The cost of this scheme is communication with the off-chip interrupt controller if an interrupt is delivered when the kernel is in a critical section, and a higher code complexity than the pessimistic schemes.

### 3.3 Cli-Pessim

The *cli-pessim* scheme is named after the x86 instruction that clears the interrupt flag, *cli*. When a critical section is entered, all interrupts are disabled. This scheme is structurally similar to the *spl-pessim* scheme, but instead of communicating with the PIC, on-chip instructions are used to disable and enable interrupts.

```
crit_sec_enter()
    disable_all_interrupts()

crit_sec_leave()
    enable_all_interrupts()

interrupt_handler(int intr_level)
    crit_sec_enter()
    handle interrupt
    crit_sec_leave()
```

The benefit of this scheme is a low per-critical section overhead. The cost is increased risk of dropped interrupts (if multiple interrupts are delivered during critical sections) and possible delay of high-priority interrupts while processing a lower priority interrupt or while the kernel is in a critical section.

### 3.4 Cli-Optim

When a critical section is entered, a global flag is set; it is cleared on exit from the critical section. When an interrupt is delivered, interrupts are disabled while the interrupt is processed.

```
crit_sec_enter()
    in_crit_sec = true

crit_sec_leave(int level)
    if (any queued interrupts)
            handle queued interrupts
    in_crit_sec = false

interrupt_handler(int intr_level)
    if (in_crit_sec)
            queue interrupt
    else
            disable_all_interrupts()
            handle interrupt
            enable_all_interrupts()
```

This scheme is very similar to the *spl-optim* scheme, but, as with the *cli-pessim scheme,* instead of communicating with the PIC, interrupts are disabled and enabled through the use of on-chip instructions.

The benefit of this scheme is low per-critical section overhead. Its cost is increased risk of lost interrupts if multiple interrupts are delivered during critical sections, and possible delay of high-priority interrupts while processing a lower priority interrupt or critical section. It also has a slightly higher code complexity than the *cli-pessim* scheme.

### 3.5 Comparison of Schemes

While a scheme's performance is important, we must also verify its correctness. Devices are designed to queue a small number of pending interrupts for a short period of time; if interrupts are disabled for a long period of time, it is possible that multiple interrupts will be merged together or lost, and data buffers can overflow with an accompanying loss of data. In some cases the loss of an interrupt is a performance issue, not a correctness issue (e.g., TCP/IP packets are retransmitted if dropped). However, we need to be sure that the system as a whole behaves correctly in the face of heavy load.

Fortunately, it is not necessary to test all four schemes for correctness. The *cli-pessim* scheme is the least responsive to external interrupts; if it behaves correctly in the face of heavy load, the other systems can do no worse.

Note that use of a *cli* scheme does not preclude assigning priorities to interrupts. Although all interrupts are disabled while the system is in a critical section, pending interrupts can still be assigned priority levels, and higher-priority interrupts will take precedence over lower-priority interrupts when interrupts are once again enabled. (In our implementation of the *cli-pessim* kernel we assign the same priorities to interrupts as are used by standard NetBSD1.2, the *spl-optim* kernel to which we compare it.)

The costs and benefits of the four strategies depend on a number of components. First, the cost of going off-chip to set the priority level needs to be measured, as does the cost of disabling interrupts on-chip. Second, the frequency with which interrupts are received, and the frequency with which an interrupt arrives while the kernel is in a critical section, must be measured. We must also determine the time spent in critical sections in order to learn whether a *cli-pessim* kernel will ignore interrupts for too long a period of time, with too high a risk of losing interrupts.

| Scheme | Synchronization Overhead |
|---|---|
| *spl-pessim* | (number of critical sections entered) • (off-chip cost) |
| *spl-optim* | (number of critical sections entered) • (flag-setting cost) + (number of critical sections interrupted) • (off-chip cost) |
| *cli-pessim* | (number of critical sections entered) • (on-chip cost) |
| *cli-optim* | (number of critical sections entered) • (flag-setting cost) + (number of critical sections interrupted) • (on-chip cost) |

**Table 2: Synchronization Scheme Overheads.** The model describing the overhead associated with the four schemes discussed. The costs are functions of five variables: critical sections entered, critical sections interrupted, off-chip cost, on-chip cost, and flag-setting cost.

The overhead of synchronization for each strategy can be described by a simple analytic model. The model must take into account:

- *number of critical sections entered:* number of times a critical section was entered, per second. This factor is important when computing the overhead for any of the schemes.
- *number of interrupted critical sections:* number of times a critical section was interrupted, per second. This is only a factor for the *optimistic* schemes.
- *off-chip cost*: the time required to communicate with the interrupt controller, first raising the priority level, then lowering it. This is only relevant for the *spl* schemes.
- *on-chip cost*: the time required to disable interrupts and re-enable them. This cost only applies to the *cli* schemes.
- *flag setting cost*: the time required to set the variable holding the mask. This cost is only a factor when using an *optimistic* scheme.

The equations that describe the overhead for each scheme are found in Table 2. In the following section we discuss our experimental setup, and then in Section 5 we measure the component costs and derive a total cost for each scheme.

# 4 Experimental Setup

In this section we discuss the kernels that we constructed and measured, and the hardware platform used.

## 4.1 Kernels

The x86 version of NetBSD 1.2, a derivative of 4.4BSDLite, uses the *spl-optim* strategy for priority level management. We started with an off-the-shelf copy of NetBSD 1.2 for our *spl-optim* measurements. With this kernel we were able to measure the frequency with which critical sections are interrupted.

Starting with NetBSD 1.2 as a base, we developed a *cli-pessim* kernel. This kernel disables all interrupts any time a critical section or interrupt handler is entered, and enables all interrupts when the critical section or interrupt handler finishes. The *cli-pessim* kernel allowed us to accurately measure the length of time spent in critical sections, the number of critical sections, and the time required to handle each type of interrupt.

## 4.2 Hardware

We ran our tests on an x86 PC with a 120MHz Pentium processor and PCI bus. The memory system consisted of the on-chip cache (8KB i + 8KB d), a 512KB pipeline burst off-chip cache and 64MB of 60ns EDO RAM. There was one IDE hard drive attached, a 1080 MB, 5400 RPM Western Digital WDC31000, with a 64KB buffer and an average seek time of 10ms. The system included a BusLogic BT946C PCI SCSI controller; attached to it was a 1033 MB, 5400 RPM Fujitsu M2694ES disk, with a 10ms average seek time and a transfer rate of 5MB/second, and a Sony 4x SCSI CD-ROM drive (model CDU-76S). The Ethernet adapter was a 10Mbps Western Digital Elite 16 with 16KB of 100ns RAM. The serial ports were buffered (16550 UART).

Pentium processors include an on-chip cycle counter, which enable very precise timing measurements. We read the counter at the start and at the end of each experiment; the difference was multiplied by the processor cycle time (8 1/3 nanoseconds) to obtain the elapsed time. The cost of reading the cycle counter is roughly 10 cycles; where significant, the timing cost has been subtracted from our measurements. We include code to read the cycle counter in the Appendix.

# 5 Synchronization Overhead

As shown in Table 2, the synchronization overhead of each scheme is a function of five variables: the off-chip priority setting cost, the on-chip priority setting cost, the flag-setting cost, the number of times a critical section is entered, and the number of times an interrupt arrives while the system is in a critical section. In this section we measure each component to derive the synchronization overhead for the four schemes.

## 5.1 Critical Sections Per Second

We ran four tests to estimate the number of critical sections entered per second under heavy load. These tests supply the value used for the *number of critical sections entered* variable in the equations above. They measure the system under a mixed programming load, heavy web traffic, and high-speed serial traffic.

The first two tests are the result of running the Modified Andrew Benchmark [Ouster90] on a local file system (Andrew local) and an NFS-mounted file system (Andrew NFS). The Modified Andrew Benchmark consists of creating, copying, and compiling a hierarchy of files, and was designed to measure the scalability of the Andrew filesystem [How88].

| | **Mean Critical Sections per sec (std dev)** | **Max Critical Sections per sec** | **Mean Critical Section Duration** |
|---|---|---|---|
| **Andrew local** | 36048 (60%) | 72521 | 3.1µs |
| **Andrew NFS** | 25867 (68%) | 72940 | 4.6µs |
| **WebStone** | 34468 (31%) | 82653 | 3.3µs |
| **Serial 115.2** | 47762 (0.05%) | 47807 | 0.6µs |

**Table 3: Critical Sections Per Second.** The mean and maximum number of critical sections entered, per second, measured with the *cli-pessim* kernel. We also measured the mean critical section duration, which is seen to be very short, especially relative to the cost of off-chip synchronization.

The third test was running the WebStone 2.0 benchmark from Silicon Graphics [SGI96]. WebStone was designed to test the performance of web servers under varying load. We installed the Apache 1.1.1 server on our test machine, and ran the WebStone clients (which generate http requests of the server) from a SparcStation 20. The maximum number of critical sections per second occurred with 50 client processes; we report the results from this test.

The fourth test measured the kernel's behavior while receiving a 64KB file via tip(1) at 115,200 bps. This test measures the behavior of the kernel when it is presented with a high rate of short-duration interrupts.

We instrumented the kernel to measure the duration of each critical section, using the Pentium cycle counter. A user-level process polled the kernel once per second to retrieve the number of critical sections and the time spent in critical sections since the last poll. We use these results to derive the numbers in Table 4.

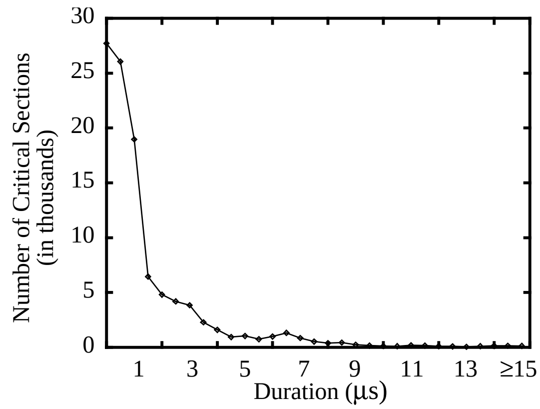


**Figure 1: Critical Section Duration Distribution.** Distribution of critical section durations for Andrew NFS; the other distributions had the same shape.

We plotted the frequency of duration of critical sections for each test, and found that the distribution roughly follows the shape of an exponential distribution (see Figure 1 as an example; the other distributions had the same shape). The standard deviation of an exponential distribution is equal to the mean of the distribution, which in all measured cases is less than 5µs. If we use the mean as an estimate of the standard deviation, we find that most critical sections take less than 10µs; in the case of the Serial 115.2, we expect most to be less than 1.2µs.

## 5.2 Interrupted Critical Sections

In order to compute the overhead of the optimistic techniques we need to determine the percentage of critical sections that are interrupted. We reran the tests from Table 4 with the standard NetBSD 1.2 kernel (which uses the *spl-optim* scheme) and measured the number of critical sections interrupted per second. These results are shown in Table 4, on page 7.

For comparison with the results of Table 4, we include the Mean Critical Sections Per Second measured using this kernel; these results are very similar to the values seen in Table 4. As we expected (based on the short duration of critical sections), we saw that a very small number of critical sections are interrupted.

## 5.3 Synchronization Primitive Cost

The synchronization overhead parameters were measured by performing each operation pair (set and

| | Mean Critical Sections per second (std dev) | Interrupted Critical Sections per second (% intr) |
|---|---|---|
| **Andrew local** | 31904 (69%) | 15 (0.05%) |
| **Andrew NFS** | 25172 (67%) | 50 (0.2%) |
| **WebStone** | 33890 (33%) | 300 (0.9%) |
| **Serial 115.2** | 45269 (8%) | 140 (0.3%) |

**Table 4: Interrupted Critical Sections.** We measured the number of critical sections entered per second on the *spl-optim* kernel, and the number of times per second critical sections were interrupted (which causes the optimistic kernel to revert to pessimistic synchronization behavior). We include the percentage of critical sections interrupted.

unset) in a tight loop 100,000 times and computing the mean and standard deviation. In all cases the standard deviation was less than 1% of the mean. The results of these measurements are given in Table 5.

Because these tests were run in a tight loop, a minimal number of cache misses and other side-effects are seen in these results, hence the synchronization times we use are somewhat optimistic. Cache misses could have a large percentage impact on the on-chip and flag-setting measurements, because these instruction sequences are very short. However, *because* they are so short, even if all of the instructions and data referenced by the code cause cache misses, the total absolute additional cost (in cycles) would be very small.

| Synchronization Primitive | Time |
|---|---|
| off-chip *(spl-pessim)* | 2.5µs (304 cycles) |
| on-chip *(cli-pessim)* | 0.18µs (21 cycles) |
| set flag (*spl-optim, cli-optim)* | 0.05µs (6 cycles) |

**Table 5: Synchronization Primitive Cost:** Time required to set priority level off-chip, on-chip, and to set a global flag saving the logical priority level. The Pentium has a write buffer, so the flag-setting cost does not include time to write to main memory.

The synchronization cost of each of the four schemes is directly driven by the costs shown in Table 5. The *off-chip* cost is the synchronization cost of the *spl-pessim* kernel (304 cycles, 2.5µs), *on-chip* gives the synchronization cost the *cli-pessim* kernel (21 cycles, 0.18µs), and the flag setting cost is the synchronization cost of the optimistic kernels (6 cycles, 0.05µs). As we immediately see by comparing the costs in Table 5 with the critical section durations seen in Table 4, the mean critical section duration seen is less than twice the off-chip synchronization cost, and more than three times the mean Serial 115.2 critical section cost.

## 5.4 Synchronization Overhead

Given the results in Table 3, Table 4, and Table 5, the synchronization cost and overhead for each of the four schemes can be computed. These costs are given in Table 6.

| Scheme | Synchronization Overhead | | | |
|---|---|---|---|---|
| | **Andrew local** | **Andrew NFS** | **Web Stone** | **Serial 115.2** |
| *spl-pessim* | 9.0% | 6.5% | 8.6% | 11.9% |
| *spl-optim* | 0.2% | 0.1% | 0.3% | 0.3% |
| *cli-pessim* | 0.7% | 0.5% | 0.6% | 0.9% |
| *cli-optim* | 0.2% | 0.1% | 0.2% | 0.2% |

**Table 6: Synchronization Scheme Overhead.** Percentage of time spent synchronizing critical sections using the described techniques. Computed using equations in Table 2 and measurements from Table 3, Table 4, and Table 5.

We see that with the large number of critical sections per second seen under the Serial 115.2 test, the *spl-pessim* kernel can spend nearly 12% of its time in synchronization primitives. Even under the lightest measured load, the system spends more than 6% of its time in synchronization primitives.

Another thing to note is that the other three techniques all have an overhead that is an order of magnitude less than that of the *spl-pessim* technique. Even though the *spl-optim* technique goes off-chip for synchronization, because so few critical sections are interrupted (from 0.05% to 0.9%) the difference is negligible (between 0.2%-0.7%).

# 6 Correctness

With the knowledge that the absolute overhead of the three proposed schemes is very low, we must be sure that none of the schemes allow interrupts to be lost or overly delayed.

The optimistic schemes do not delay high-priority interrupts, nor does the *spl-pessim* scheme; the *cli-pessim* scheme is the only one that masks all interrupts while in a critical section or interrupt handler. If the *cli-pessim* scheme can operate correctly, the other schemes will also operate correctly.

In Table 3 we saw that the mean critical section duration is between 0.6 and 4.6µs. We are concerned that handling a device interrupt could take much longer, especially with programmed I/O devices, such as the

| Test | Duration of Test (seconds) | Total Interrupts | Interrupts per second | Mean time to handle interrupt (std dev) | Min time to handle interrupt | Median time to handle interrupt | Max time to handle interrupt |
|---|---|---|---|---|---|---|---|
| *ide disk* | 3.2s | 1027 | 321 | 1519µs (7%) | 15µs | 1545µs | 2003µs |
| *scsi disk* | 8.8s | 553 | 61 | 65µs (15%) | 16µs | 64µs | 86µs |
| *scsi cd-rom* | 13.4s | 3931 | 293 | 27µs (7%) | 20µs | 26µs | 40µs |
| *floppy* | 64.2s | 506 | 8 | 169µs (57%) | 4µs | 200µs | 271µs |
| *serial-300 (8K)* | 274s | 8221 | 30 | 9µs (5%) | 9µs | 9µs | 12µs |
| *serial-38.4 (64K)* | 17.4s | 8195 | 471 | 27µs (2%) | 19µs | 27µs | 42µs |
| *serial-115 (64K)* | 5.8s | 8176 | 1410 | 27µs (4%) | 11µs | 27µs | 54µs |
| *ethernet-small* | 2.4s | 1003 | 418 | 78µs (25%) | 44µs | 78µs | 370µs |
| *ethernet-large* | 3.4s | 1703 | 501 | 487µs (8%) | 45µs | 491µs | 1194µs |

**Table 7: Behavior of *spl-optim* kernel during device tests.** Frequency of device interrupts and time to handle an interrupt from each device was measured. Although the spread between minimum and maximum is large, the median is in all cases other than *floppy* is within 2% of the mean. Each test measured the reception of several thousand interrupts.

IDE disk, which copy data from the device to the kernel one word at a time.

We are concerned that high-frequency interrupts are not overly delayed while in long-duration interrupt handlers. We gathered data on both long-duration handlers (IDE disk interrupts) and high-frequency interrupts (serial lines at 115.2Kbps).

We measured the time required to process interrupts from a variety of devices attached to our test system. Our tests involve reading from or writing to each device and measuring the amount of time required to perform the test and the number of interrupts delivered while performing the test. We ran the following tests:

- *ide disk:* write 8MB to IDE disk.
- *scsi disk:* write 8MB to SCSI disk.
- *scsi cd-rom:* read 8MB from SCSI CD-ROM.
- *serial-300:* read 8KB (via tip) at 300bps.
- *serial-38.4:* read 64KB (via tip) at 38.4Kbps.
- *serial-115.2:* read 64KB (via tip) at 115.2Kbps.
- *floppy:* read 1MB from raw floppy disk.
- *ether-small:* flood ping of 1000 64 byte packets.
- *ether-large:* flood ping of 1000 1024 byte packets.

The results of these tests are shown in Table 7. The mean time to handle an interrupt from a particular device is useful, although we found that there can be considerable variation. For each test we also include the minimum, maximum, and median interrupt handling time.

## 6.1 IDE, SCSI, and Floppy Interrupts

As mentioned above, the *ide disk* test performs programmed I/O (PIO), i.e., data is copied by the CPU one byte or word at a time between the disk controller and main memory. This is very CPU intensive, and can be quite slow. By comparison, the SCSI disk controller uses direct memory access (DMA) to copy data directly from the controller to main memory without the intervention of the CPU. Each *ide disk* interrupt took substantially longer than a *scsi disk* interrupt (1519µs vs. 65µs).

Surprisingly, although the specifications of the IDE and SCSI disks is very similar, it took nearly three times as long to transfer 8MB to the SCSI disk as it did to transfer it to the IDE disk. We believe that this is an artifact of the NetBSD 1.2 disk drivers; we saw similar performance with the generic NetBSD 1.2 kernel and our *cli-pessim* kernel. We do not believe that this reflects the underlying performance of the disk or the controller; under BSD/OS 2.1 the performance of the two disks was much closer (in fact, the SCSI disk was about 10% faster than the IDE disk).

The mean time required to process an IDE interrupt (1.5ms) is substantially greater than any other interrupt processing time we measured. We discuss the possibility of timing conflicts between IDE interrupts and other interrupts in Section 6.4.

The standard deviation of *floppy* interrupt processing time is quite large (57%). It is caused by the several different types of interrupts generated by the floppy device, from seek completion notification (4µs), to transfer completion notification (25µs), to data transfer (250µs). The maximum time seen, 271µs, is short enough that we are not concerned about floppy interrupts.

## 6.2 Serial Interrupts

We see that a *serial-300* interrupt takes one third as long as a *serial-38.4* interrupt; we attribute this to the fact that multiple characters are transferred on each *serial-38.4* interrupt, as evidenced by the fact that 1/8 as many interrupts were generated per kilobyte by the latter. (Please note that the *serial-300* test transferred 8KB, where the other two tests transferred 64KB.)

When we tested higher-rate serial connection (at 115.2 bps) we saw roughly the same number of interrupts as we saw in the *serial-38.4* test (8195 vs. 8176) and the same mean interrupt handling time (27µs), implying the same number of characters processed per interrupt, but the total test took one third as long. The high rate of interrupts during the *serial-115.2* test may conflict with the large amount of time required to process an *ide-disk* interrupt; we discuss this possibility in Section 6.4.

## 6.3 Network Interrupts

As stated above, our network tests were taken using a 10Mbps Ethernet board. We ran three tests: flood (continuous) ping, UDP, and TCP. We saw no significant variation in interrupt processing time as a function of protocol (ping/IP, UDP, and TCP), and hence include only the ping/IP results.

Interrupt processing time is independent of protocol because the higher-level protocol processing (where we would see differences between TCP, UDP, and ping) does not take place when the packet arrives; instead, the packet is queued for processing by a higher-level “soft” interrupt handler which runs once the hardware interrupt has completed.

We ran additional tests with varying packet sizes. We saw the expected relationship between packet size and processing time (which more or less scaled linearly with packet size).

Faster network technologies (e.g., 100Mbps Ethernet and ATM) would generate interrupts much more frequently than our 10Mbps Ethernet interface. However, as we see below, the rate at which the serial device generates interrupts is sufficient to preclude use of the *cli* schemes. Based on this, the interrupt generation rate of ATM is a moot point.

## 6.4 Summary

When examining the data gathered for each interrupt, we found that although there was often a significant difference between the minimum and maximum, in most cases the standard deviation was relatively small, and the median was close to the mean.

Our results show that in our environment it takes little time to handle an interrupt from any of our devices. No interrupt seen took more than 2ms to service, and in most cases the interrupt processing time was several orders of magnitude less than that.

However, we see from Table 7 that interrupts can occur very frequently. For example, the *serial-115.2* interrupts arrived at 1400Hz (every 700µs). We must examine the consequences of delaying the processing of these interrupts.

As was discussed in Section 2, we can partition interrupts into two classes: *solicited* and *unsolicited*. *Solicited interrupts* come in response to requests from the CPU (e.g., in response to a disk request). *Unsolicited interrupts* are generated externally, and are not under the control of the system (e.g., serial and network traffic). The rate of solicited interrupts are under the control of the system; when a solicited interrupt is received, the system spends its time processing the interrupt rather than generating more requests. The system can thus control the rate at which solicited interrupts are generated.

The system does not control the rate at which unsolicited interrupts are generated. However, the system can indirectly impact the rate at which they are generated, either in software (by not sending acknowledgments to network packets that are dropped or processed in time), or in hardware (by use of serial-line flow control).

To block serial interrupts it is necessary for the system to communicate with the device, and to be aware that it is necessary to slow the interrupt rate. If serial interrupts are overly delayed, the system will not be aware of the possibility of overflow, and hence will not be able to stem the tide in time.

From our measurements, the longest that an interrupt will be delayed is 2ms (the longest time spent handling an *ide-disk* interrupt). The *serial-115.2* device receives characters every 70µs (115.2 bits per sec = 14,400 bytes per sec = one byte every 70µs). This means that up to 28 characters (2ms / 70µs = 28) could arrive while an Ethernet interrupt is being processed. Although the serial port is buffered, the buffer only holds 16 characters. At this rate, characters could easily be lost on the serial line while processing an IDE interrupt.

If we could eliminate programmed I/O devices, the *cli* schemes would work. However, newer devices with a high interrupt rate (e.g., ATM network controllers), combined with the possibility of slow interrupt handlers shift the balance against the *cli* techniques. Although, in some hardware combinations, the *cli* schemes would work, many common configurations could easily lead to loss of data.

The failure of the strict *cli* schemes leads us to propose a new, hybrid scheme, *cli-spl-pessim*, which is described in Section 7. The *cli-spl-pessim* scheme combines the low cost of the *cli* schemes for critical section synchronization with the interrupt prioritization of the *spl* schemes.

# 7 Analysis and Proposal

Our results show that under the benchmark workloads the absolute cost of synchronization using the optimistic techniques is low, less than 0.4%. The traditional *spl-pessim* scheme has a much higher cost, over 6% in all tested cases.

The performance improvement we see is less than that seen by Stodolsky et al. with their *spl-optim scheme* (14%). We attribute this in part to the differences between our benchmarks and theirs; their test consisted of a highly optimized null RPC, which has a single critical section. Because the null RPC took very little time (2140 cycles), an improvement in critical section synchronization had a large performance impact. Under the more varied loads that we measured, the kernel spends a much lower percentage of its time in synchronization code.

For the *cli* schemes to work, critical sections and interrupt handlers must complete their work quickly. These schemes completely disable interrupts during a critical section, so a long-running interrupt handler can delay the delivery of interrupts for a long period of time, increasing the likelihood of data loss. The combination of devices whose interrupt handlers require a long time to service (e.g., IDE disks with programmed I/O) with low latency requirements (e.g., very fast serial ports) is a worst-case scenario for the *cli* scheme.

## 7.1 Proposal

We find the simplicity of the *pessim* schemes and the low cost of the *cli* schemes appealing. This leads us to propose a fifth scheme, *cli-spl-pessim*, which combines the *cli* technique for critical sections with the prioritization of the *spl* technique for interrupt handlers. In this scheme, the *cli* and *sti* instructions are used to mask interrupts while the kernel is in a critical section; because critical sections are quite short, this will not overly delay interrupt delivery. When handling an interrupt, the kernel communicates with the off-chip interrupt controller, disabling the delivery of equal and lower-priority interrupts. Because

- interrupts are delivered much less frequently than critical sections are entered,
- interrupt handlers take much longer than critical sections, and
- on our hardware platform, when an interrupt is delivered it is necessary to communicate with the off-chip controller,

the additional performance impact of going off-chip will be minimal and more than made up for by the increase in system robustness. In the format of Section 3, we include the following pseudo-code for *cli-spl-pessim*:

```
void crit_sec_enter(int level)
    disable all interrupts

void crit_sec_leave(int level)
    enable all interrupts

void interrupt_handler
    prev_level = cur_interrupt_level
    PIC_mask(interrupt_level)
    handle_interrupt
    PIC_mask(prev_level)
```

The benefit of this scheme is the low cost of synchronization in critical sections. In addition, because we use an *spl* scheme for synchronizing handlers, a low-priority handler will be interrupted by the arrival of a high-priority interrupt.

Because our model includes only the cost of synchronization of critical sections, and disregards the cost of the synchronization overhead in interrupt handlers, using our model the cost of *cli-spl-pessim* is identical to that of the *cli-pessim* scheme, with (as shown in Table 6) sub-1% synchronization overhead for the system loads that we studied.

With the performance characteristics and simplicity of the *cli-pessim* scheme, without its incorrect behavior under heavy load, we plan to use the *cli-spl-pessim* scheme in the implementation of our new operating system kernel.

## 7.2 Related Schemes

Although we have described NetBSD as using the *spl-optim* technique, it also supports a *cli-optim*-style technique for short duration interrupt handlers.

The Linux[1] operating system uses a scheme similar to *cli-spl-pessim*. The *cli* and *sti* instructions are used to synchronize critical sections (with the concomitant high performance), and, like NetBSD 1.2, short-duration

1. We examined the source code of version 1.3.0.

interrupt handlers are run with all interrupts disabled. Long-duration handlers are either run with no protection from interrupts (which increases their complexity) or wake a kernel task to handle the processing of the interrupt, and then return. Although this scheme efficiently synchronizes critical sections, we believe that the increased latency and complexity of waking a separate task argue against use of this technique.

## 8 Conclusions

In this paper we have shown that it is worthwhile to rethink how interrupts and synchronization are handled on modern hardware[2].

Our results have driven the design of our new kernel, where we use the proposed *cli-spl-pessim* scheme. This scheme provides us with the simplicity of the pessimistic schemes we describe, with the low overhead of the optimistic schemes.

Comparing our scheme to those used in earlier systems, we are reminded of how quickly hardware changes: the CPU and I/O bus of today is substantially faster than the one available in 1993 [Stod93], and much, much faster than the one available on the VAX or the PDP-11 [Leff89]. Nevertheless, developers of newer systems [Cust93] find themselves re-using old, complex techniques to solve a problem that may no longer exist.

## Status and Availability

All code discussed in this paper (benchmark tests and patch files for a *cli-pessim* NetBSD 1.2) is available at http://www.eecs.harvard.edu/~chris.

## Acknowledgments

Our thanks to our advisor, Margo Seltzer, for her invaluable feedback, wordsmithing, and grasp of the subtleties of “that” and “which.” We thank our reviewers, especially Mike Karels, our shepherd, for providing very helpful feedback. And thanks to Brad Chen, whose previous work on fast interrupt priority management helped inspire this work.

## Bibliography

[Berg86] Berglund, E., “An Introduction to the V-System”, *IEEE Micro 10*, 8, 35–52 (August 1986).

[Bers92] Bershad, B., Redell, D., Ellis, J., “Fast Mutual Exclusion for Uniprocessors,” *ASPLOS V,* 223–233, Boston, MA (1992)

[Cust93] Custer, H., *Inside Windows NT*, Microsoft Press, Redmond, WA, 218–221 (1993).

[Green96] Greenwald, M., Cheriton, D., “The Synergy Between Non-blocking Synchronization and Operating System Structure,” *Second Symposium on Operating Systems Design and Implementation*, 123-136, Seattle, WA (1996).

[Herl90] Herlihy, M. “A Methodology for Implementing Highly Concurrent Data Structures,” *Second ACM Symposium on Principles and Practice of Parallel Programming*, 197–206 (1990).

[How88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M., “Scale and Performance in a Distributed File System,” *ACM Transactions on Computer Systems 6*, 1 51–81 (1988).

[Intel95] *Pentium Processor Family Developer’s Manual, Volume 3: Architecture and Programming Manual*, Intel Corporation, Mt. Prospect, IL (1995).

[Leff89] Leffler, S., McKusick, M. K., Karels, M., Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley Publishing, Reading, MA (1989).

[Mass91] Massalin, H., Pu, C., “A Lock-Free Multiprocessor OS Kernel,” Technical Report CUCS-005-91, Department of Computer Science, Columbia University (1991)

[Mogul96] Mogul, J., Ramakrishnan, K. K., “Eliminating Receiver Livelock in an Interrupt-Driven Kernel,” *USENIX Technical Conference,* 99–111, San Diego, CA (1996).

[Ouster90] Ousterhout, John, “Why Aren’t Operating Systems Getting Faster as Fast as Hardware,” *USENIX Summer Conference*, 247–256, Anaheim, CA (1990)

[Selt96] Seltzer, M., Endo, Y., Small, C., Smith, K., “Dealing With Disaster: Surviving Misbehaving Kernel Extensions,” *Proceedings of the Second OSDI*, Seattle, WA (1996).

[SGI96] Silicon Graphics Inc., “Webstone: World-Wide Web Server Benchmarking,” http://www.sgi.com/Products/WebFORCE.

[Stan88] Stankovic, J., Ramamritham, K., “A New Hard Real-Time Kernel”, *Hard Real-Time Systems*, IEEE, 361–370 (1988).

[Stod93] Stodolsky, D., Chen, J. B., Bershad, B., “Fast Interrupt Priority Management in Operating System Kernels,” *USENIX Symposium on Microkernels and Other Kernel Architectures*, 105–110, San Diego, CA (1993).

[Wing92] Wing, J, Gong, C, “A Library of Concurrent Objects,” CMU CS TR 90-151, School of Computer Science, Carnegie Mellon University (1992).

2. At least, modern by comparison with the PDP-11.