

Dartmouth College Computer Science Technical Report TR2004-500
Scheduling Pipelined, Multi-Threaded Programs in Linux

Brunn W. Roysden III
Dartmouth College
Department of Computer Science
Brunn.W.Roysden.III.04@alum.dartmouth.org

Advisor: Thomas H. Cormen

June 4, 2004

Abstract

A process causes latency when it performs I/O or communication. Pipelined processes mitigate latency by concurrently executing multiple threads—sequences of operations—and overlapping computation, communication, and I/O. When more than one thread is ready to run, the scheduler determines which thread in fact runs. This paper presents techniques for scheduling pipelines, with the following three findings.

First, using Linux kernel version 2.6 and the NPTL threads package, we observe a 3-6% performance improvement over kernel version 2.4 and the LinuxThreads package.

Second, we test techniques that both take advantage of prior knowledge about whether a program is I/O-bound or compute-bound and raise and lower priorities before the pipeline begins working. These techniques, referred to as fixed scheduling, further improve performance by 5% in the case of the compute-bound columnsort algorithm. In the I/O-bound algorithm, fixed scheduling failed to yield better performance than the default scheduling.

Third, we test simple, adjusting methods that do not take advantage of prior knowledge about whether a program is compute-bound or I/O-bound but rather adjust scheduling as the pipeline progresses. These techniques, called adjusting scheduling, fail to yield better performance than the default scheduling in any of our test cases.

In addition, we suggest new scheduling calls and other operating-system improvements.

1 Introduction

Latency is a significant but surmountable problem for high-performance computing. *Latency* is idle time on the CPU while it waits for data to arrive. Pipeline-structured programs mitigate latency by overlapping high-latency and CPU-intensive operations. FG is a framework generator that reduces time to completion and facilitates experimentation. We shall introduce priority to extend FG’s capabilities. Scheduling the priority of operations to maximize system-resource usage further improves performance.

Pipelines mitigate latency by overlapping operations that use different system resources. To set up a pipeline, the programmer breaks up the code into *stages*, where each stage predominantly has one of computation, communication, and I/O operations. He also breaks up the data into *blocks*, so that a stage can work on a single block without needing outside data. A *round* in a pipeline is the time during which every stage begins and completes work on its buffer. At the end of the round, each stage conveys its buffer to its successor and accepts its next buffer from its predecessor; the next round subsequently begins. While each block passes through the stages sequentially, all stages have data at a given time. Rather than using only one resource at a time, therefore, a pipelined program can use all resources at the same time. The lower bound of a pipelined program is not the sum of computation, communication, and I/O time, but only the maximum of the three times—the running time of the *limiting resource*.

If the limiting resource is running at maximum throughput, then scheduling will not reduce the running time of the program; scheduling can improve overall running time in a pipeline to the extent that it can reduce the actual running time per round of the limiting resource. FG, a framework generator for pipelined programs not only simplifies creating pipelined programs but also allows scheduling changes by changing the structure of the pipeline. This paper expands on those controls by introducing *priority* to FG. Priority has to do with whether one process or thread should preempt another.

This paper investigates scheduling in the two following cases: I/O-bound and compute-bound programs. The I/O case works as follows. Assume that a program is I/O-bound and that the I/O device is not running at maximum throughput. The scheduler can give I/O operations higher priority to the CPU. The increased priority means that the I/O thread can preempt other threads and send a new read request back to the I/O device more quickly, reducing idle time on the I/O device. The scheduler can also try to reorder I/O operations. Alternatively, assume that a program is compute-bound. In this case, the scheduler can give computation operations higher priority. The higher priority means other threads will not be able to preempt the computation threads, resulting in fewer context switches and preventing cache misses. One computation stage could take considerably longer than another. In an I/O-bound pipeline or single processor node the imbalance does not matter once the pipeline is in full swing. In an SMP node running a compute-bound pipeline this will reduce performance. Unfortunately, Linux controls are not fine enough to handle individual-thread time quanta.

It is difficult to determine the limiting resource directly, however, especially “on the fly.” We must use techniques to guess which is the limiting resource in order to adjust priority and improve performance. A programmer can raise the priority of any stage that has multiple buffers in its input queue, or lower the priority of a stage with multiple buffers in its output queue—signs that the stage is moving slower or faster than its neighbor respectively.

The results are as follows. First, using Linux kernel version 2.6 and the NPTL threads package, we observe a 3-6% performance improvement over kernel version 2.4 and the LinuxThreads package. Second, we test techniques that both take advantage of prior knowledge about whether a program is I/O-bound or compute-bound and raise and lower priorities before the pipeline begins working. These techniques, referred to as fixed scheduling, further improve performance by 5% in the case of the compute-bound column sort algorithm. In the I/O-bound algorithm, fixed scheduling failed to yield better performance than the default scheduling. Third, we test simple, adjusting methods that do not take advantage of prior knowledge about whether a program is compute-bound or I/O-bound but rather adjust scheduling as the pipeline progresses. These techniques, called adjusting scheduling, fail to yield better performance than the default scheduling

in any of our test cases.

The remainder of this paper is organized as follows. Section 2 describes threading and FG. It describes how the design of FG relates to the problem of scheduling threads. In Section 3, we survey scheduling in Linux and the system calls for altering the default scheduling available to Linux users. Section 4 discusses experimental results. Section 5 concludes the paper by suggesting further areas of research and system calls that would be helpful for pipeline scheduling in an ideal operating system.

2 Threading and FG

Pipelined programs can use threads to overlap computation, communication, and I/O, thereby reducing latency. This section describes threads, how FG simplifies working with threads, and how FG allows easy experimentation with pipeline structure. We extend FG by introducing priority.

Threading

The performance of threads and processes in Linux forms the bases of how we handle concurrency and what controls we have on scheduling. The operating system switches between active threads when running a multithreaded program. It puts threads on a waiting queue when they need access to data currently locked or when they perform a blocking operation, such as a disk read or communication.

Whereas processes are separate programs with separate memory spaces, threads are concurrent paths of execution in the same program. One process can have many threads. The Linux kernel represents threads and processes identically, however, with a *process descriptor*, which is a kernel-level data structure.¹ A flag in the process descriptor indicates whether it is a process or a thread. This model is called the *one-on-one model* because the operating system schedules each thread directly rather than scheduling processes and letting the user divide up the process's time among its threads.²

This implementation, when combined with hardware and kernel improvements, allows for efficient thread use. The *Native Posix Threading Library* has replaced the old implementation of threads, Linux-Threads, in the standard runtime environment. The new library uses the same low-level system call for creation and exiting of threads as is used for processes. This system call results not only in constant-time creation and exiting of threads, but also in each action taking approximately one eighth as long as in Linux-Threads, the old implementation. The number of threads the library allows has increased from 2^{13} to 2^{31} . In tests run by the creators of NPTL, creating and deleting 100,000 threads took 15 minutes using Linux-Threads and only 2 seconds using NPTL. The developers also made other improvements.³ In Section 4, we shall see that the NPTL implementation, when combined with scheduler improvements in kernel version 2.6 versus version 2.4, results in better performance. Although threads are efficient, they are also hard to work with. Creation, synchronization, and exiting of pipelines is tricky. Also, changing the configuration of a pipeline is cumbersome.

¹Source code for the process descriptor and scheduler appears in `/usr/src/kernel/sched.c`.

²The alternative model is the *M-on-N model*, where the number of kernel data structures is fewer than the number of threads. The operating system allows the user to schedule threads directly, or to pass information about which thread to schedule through a message-passing scheme such as scheduler activations.

³One important change is that all threads within a process now have the same *process ID*. One can test which version of threading is running by creating a few threads and seeing if they all have the same PID. The second improvement concerns signal handling. The kernel now handles signals much more efficiently for threads. There are still shortcomings preventing full POSIX compliance. For example, although the `nice` system call is supposed to be a process-wide command, in Section 4, we shall see that we can `nice` a single thread. We will use `nice` to test different scheduling arrangements because it does not require superuser privileges.

FG

FG is a framework generator for pipelined programs. It reduces time to completion and facilitates experimentation. A programmer takes a program that runs sequentially on a single node or with communication on a cluster and breaks it into smaller pieces. The programmer places each piece of code in a separate function and associates each function with an FG stage. FG creates a thread for each stage and handles synchronization between stages. Each stage works on buffers of data. FG handles creation, recycling, and swapping of buffers. FG also notifies each stage when the last buffer in a pipeline passes through, as some algorithms require stages to treat the last buffer differently. The stages run concurrently. By overlapping computation, communication, and I/O, the system mitigates the effect of latency in the multithreaded implementation. After the completion of the pipeline, FG automatically shuts it down. FG provides additional functionality, described in the next paragraph, that allows programmers to test different configurations of pipelines.

FG facilitates trying many different pipeline configurations; these features constitute FG's built-in scheduling controls. There are three such features. The first is *multistage threads*. If a programmer wants two stages to run as a unit, then he can map both stages to a single thread. For example, a pipeline might contain two communication stages on a single thread since both cannot run in parallel. As a result, the pipeline requires fewer threads, but the downside is that the built-in scheduling of the operating system—including that found in the operating system's communication and I/O layers—must schedule the stages in the order specified by the programmer. As described in Section 3, scheduling is optimized in Linux. Second, FG also provides *multistage repeat*. For example, a pipeline could have a multistage read and write thread run two reads and then two writes. Multistage repeat also departs from the built-in scheduling of the operating system. Third, the programmer can easily separate stages. The programmer can break computational stages into smaller pieces so that the operating system can balance them in an SMP node. More stages means more synchronization overhead and could reduce performance. A fourth concept, not yet implemented in FG, is *forking* and *joining*. Multiple incarnations of a stage could run in parallel and feed back into the pipeline. Each of these techniques relates to controlling the number of threads or the behavior of stages within threads. There are advantages and disadvantages to each. Pipeline designs are best compared by testing them against one another. FG facilitates quickly trying different pipeline structures in order to improve performance.

Introducing the notion of priority into FG allows more control over scheduling and the possibility of increasing performance even further. The stage data structure, the building block of pipelines in FG, easily facilitates adding the notion of priority. FG represents each stage as a C++ object with methods to accept buffers from the stage's predecessor and convey buffers to stage's successor. In Section 4, we add adjusting-scheduling code to these methods. This code will change the priority of the calling stage under certain circumstances. During accept and convey calls, stages use the synchronization primitives that control the interstage queues. A stage can check on the status of the queues by making a system call to the primitives. The addition of this call requires introducing locking, however, and might diminish performance. Each FG stage can also have programmer-defined init and cleanup functions, which are called at the beginning and end of the pipeline respectively. These functions are logical places to put fixed-scheduling code. Section 4 discusses the results of the tests of fixed and adjusting scheduling.

3 Linux scheduling and system calls for scheduling

This section describes default scheduling in Linux and what changes a programmer can make to scheduling behavior.

Linux scheduling

The implementation of the scheduler in Linux not only determines the default scheduling behavior, but also the options for customizing scheduling behavior. Recent improvements in kernel version 2.6, particularly the load balancer and preemptable kernel, have increased performance for I/O-bound or compute-bound applications.

The scheduler is designed around the one-to-one model and the notion of priority. Linux 2.6 introduced the $O(1)$ -time scheduler, which has two identical data structures. The data structure is an array of ready queues with one entry for each possible priority. One array holds ready queues of active processes, and the other holds queues of expired processes. An expired process is one that has used up its timeslice on the CPU but is otherwise ready to run. After all the processes of highest priority have used their timeslices, the scheduler gives every process a full timeslice and designates the expired array as the ready array and the ready array as the expired. The switch is the start of a new *epoch*, which will last until another switch. The design of the scheduler shows that its basic behavior is to run the highest priority threads round robin.

Two important functions run in the background: the load balancer and the timeslice-adjustment function. The load balancer ensures that in an SMP node, each processor has the same number of processes. An equal number of threads does not ensure an equal load, since some processes can take longer than others. The timeslice-adjustment function keeps track of which processes use up their quanta and lowers their priorities. In testing, I did not observe the effects of this function.

The 2.6 kernel is preemptable. Preemption helps I/O-bound and communication-bound pipelined programs. High-priority threads can become ready after interrupts and send new requests to devices faster in a preemptable kernel. The 2.6 kernel has other improvements also.⁴ All of these improvements lead to performance gains.

System calls for scheduling

Although the POSIX standard defines ways to control which of the ready threads the operating system chooses to run—modifying the default scheduler behavior—POSIX leaves considerable leeway to vendors in implementing controls. Linux provides minimal implementation of scheduling controls, and in some regards it fails to meet the POSIX requirements.⁵ The lack of controls constrains what actions the user can take to control scheduling.

POSIX scheduling controls are called real-time scheduling, which suggests the philosophy behind them. Real-time scheduling occurs when some task requires immediate system attention because of concerns external to the computer. An example, cited in [3], is the control thread for a robotic arm in a factory. This thread needs to be able to react to sensor input in real time. The user can give the control thread privileges to preempt the other threads, by increasing its priority. Priority is all or nothing. The scheduler always chooses a ready thread with the highest priority to run. If a high-priority thread's timeslice expires and no threads of equal priority are ready to run, a new epoch begins and the high-priority threads continue to run, starving low-priority threads.

⁴The virtual file system has been redesigned to have a simpler data structure for a memory block. The simplicity improves kernel performance and should speed up I/O requests. The operating system has a new I/O layer, and the separate I/O scheduler has a more efficient algorithm that groups reads and writes together. The virtual memory system can directly access high memory. If the SCSI driver is not already making these optimizations, the I/O layer will improve performance. Future versions of the Linux kernel will have an improved SCSI layer. In another vein, the new Intel compiler has the capability to use *hyperthreading* built into the Xeon processor. Hyperthreading is fast context switching between processes at the hardware level to increase processor throughput. Intel claims performance improvements of 25%.

⁵The function `sysconf(_SC_THREAD_PRIORITY_SCHEDULING)` returns 1 if real time support is provided and 0 otherwise. The `sysconf` source appears in `/usr/include/unistd.h`. Although the `sysconf` call returns 1 for NPTL, it should return 0. Thread priority scheduling is actually undefined in NPTL; different systems may experience different behavior.

System Call	Result	Type
<code>sched_setscheduler</code> , <code>sched_getscheduler</code>	Sets and gets static priority, values 0–99 (highest)	Static
<code>nice</code>	Raises the dynamic priority by the number provided, values 20 (highest)–19, <code>nice(0)</code> returns current priority	Dynamic
<code>setpriority</code> , <code>gepriority</code>	More modern version of <code>nice</code>	Dynamic
<code>sched_yield</code>	Causes caller to yield the processor and go to the back of the priority queue	-

Table 1: Summary of scheduling system calls.

In testing, I observed that the magnitude of priority differences does not matter (i.e., priority 2 versus priority 1 has the same behavior as priority 20 versus priority 1). I observed that there are two different priority metrics, static and dynamic, although both had the same effect—the higher priority thread always runs. Confusingly, a higher priority number does not always mean higher priority. For dynamic priority, a lower number is a higher priority. For static priority, a higher number is higher priority.

The operating system grants access to the CPU for threads of equal priority based on the scheduling policy. POSIX defines first-in first-out (FIFO) and round robin (RR) policies. As the name implies, a FIFO policy means that the thread given access to the CPU runs until it blocks or a higher-priority thread is ready. An RR policy means that threads are given time quanta and when the time expires, another thread with the same priority will run. If a process with higher priority becomes ready, it preempts the running process in both policies, as in the default policy.

Table 1 provides a summary of scheduling commands and their behavior.

4 Experimental results

This section describes the results of performance tests run on an out-of-core implementation of the column-sort algorithm. The algorithm has the following five stages: read, compute, communicate, compute, and write. Two versions of the column-sort algorithm are used for each test. The first is compute-bound. The computation stages in the second have been optimized, and the algorithm is I/O-bound. In the tests, the I/O-bound version of column-sort is sorting twice as much data as the compute-bound version.

Since all scheduling system calls have the same observed effect—causing the thread with the highest priority to run whenever it is ready—we use the `nice` system call because the user can call `nice` (with a positive value) without needing superuser privileges.⁶ Other scheduling calls, specifically the thread calls, require superuser privileges.

Testing environment

I tested timing and performance using a Beowulf cluster of 32 dual 2.8-GHz Intel Xeon nodes. Each node uses Fedora Core 1 running Linux kernel 2.6.5 #1 SMP. The Native POSIX Threading Library (NPTL) is the thread package. I used GCC version 3.3.3-03 to compile FG and the algorithms. I used the time command from tcsh shell version 6.12.00.

Each node has 4 GB of RAM and an Ultra-320 36-GB SCSI hard drive running at 15,000 RPM. I used the C stdio interface for disk I/O. The nodes run the EXT3 File System, which has journaling built on top of EXT2. For future research, it is more efficient to use plain EXT2 or a raw disk. The nodes are con-

⁶Although the user can raise the `nice` value—lower the priority—of a thread, he cannot lower the `nice` value of the thread without superuser privileges, even back to its original value. The `nice` call is an irreversible action for a non superuser, therefore.

Kernel	Observations	Mean	Std. Err.
Compute: Kernel v. 2.4, LinuxThreads	125	13.38	0.040
Compute: Kernel v. 2.6, NPTL	251	12.61	0.039
I/O: Kernel v. 2.4, LinuxThreads	41	43.52	0.58
I/O: Kernel v. 2.6, NPTL	43	42.09	0.26

Table 2: Performance in new and old kernel and thread libraries.

Scheduling	Observations	User Time	System Time	Clock Time	Procc. Use
Compute: Default	251	23.50	1.05	12.61	194.6
Compute: Nice Down I/O	227	23.05	1.05	12.41	194.1
Compute: Nice Up I/O	202	22.24	1.05	12.05	193.1
I/O: Default	173	23.88	11.07	24.26	143.6
I/O: Nice Down I/O	112	23.82	11.10	24.36	142.9
I/O: Nice Up I/O	83	23.99	11.07	24.21	144.32

Table 3: Performance in fixed scheduling algorithms.

nected with a 2-GB/sec Myrinet network. Communication occurs via MPI calls. I used the ChaMPIon/Pro implementation of MPI.

The tests were run on a single node using an out-of-core dataset located on the local hard drive in the /tmp directory. To prevent file caching from affecting the test results of the I/O-bound version of columnsort, we generate two datasets and work on the first. The second is garbage and is the size of the node’s main memory to ensure that data from the dataset we are using does not start in memory.

Kernel improvements

These are the results from the columnsort algorithm run on the 2.4 kernel with LinuxThreads and the 2.6 kernel with NPTL. Table 2 shows the new kernel and threads package are 6% faster than the old implementations for the compute-bound thread and 3% faster for the I/O-bound thread. The code run on the two systems has only minor differences.⁷ The I/O-bound columnsort code used for this test was slightly different than that used in the other tests in this paper, which is why the running time is longer and the processor usage higher.

Fixed scheduling

The first step in fixed scheduling is determining the limiting resource and raising the priority of the threads that use it. We see from Table 2 that one implementation has a processor usage of 194.6% out of 200%. The other has 143.6%. It is safe conclude that they are computation-bound and I/O-bound respectively. We can also run tests where we double the number of I/O operations and see the effect on clock time. If the clock time doubles, then the pipeline is I/O-bound. A similar test can be used if the programmer thinks the pipeline is communication-bound. Having determined that the algorithms are compute-bound and I/O bound, the next step is to increase the priority of the threads that contain operations for the limiting resource.

⁷The implementations used different communication calls in two tests—cnpic versus mpic in the new versus old respectively. Since the tests were run on single nodes, this probably does not even affect the results.

Scheduling	Observations	User Time	System Time	Clock Time	Procc. Use
Compute: Default	251	23.50	1.05	12.61	194.6
Compute: Dynamic up	143	23.08	1.05	12.42	194.0
Compute: Dynamic up, down	121	23.06	1.05	12.44	193.7
I/O: Default	173	23.88	11.07	24.26	143.6
I/O: Dynamic up	83	23.80	11.10	24.44	142.4
I/O: Dynamic up, down	137	24.05	11.10	24.56	142.8

Table 4: Performance in adjusting scheduling algorithms.

Variable	Compute: Coefficient	I/O: Coefficient
User time	0.512	0.64
	(1072.86)**	(18.76)**
System time	0.501	0.70
	(19.64)**	(15.15)**
Percent used	-0.066	-.174
	(213.71)**	(90.89)**
t stats.	in parens.	
* sig. at 5%	** sig. at 1%	

Table 5: Clock Time and components regression.

Table 3 shows the results from the tests of fixed scheduling. We run the test 225 times for the compute-bound algorithm and 173 times for the I/O-bound algorithm. The value of 12.61 for clock time for the default scheduling of the compute-bound thread means the wall clock time for the default scheduling is 12.68 seconds. Similarly, the value of 12.05 for the nice up I/O scheduling means that the average running time for the program with the I/O threads calling nice with a positive parameter (having lower priority) is 5% faster than the default. The results from a simple t-test for whether the two means are equal (i.e. whether the running times are statistically different) shows that in addition to having a difference in running times of 5%, the probability that the running times are different by chance is less than 0.005%. There are no other significant improvements in performance over the default scheduling in Linux.

Adjusting scheduling

A second set of tests calls nice on threads during running time based on a rule that does not take advantage of knowing whether the program is compute-bound or I/O-bound. In the first technique, a stage calls nice up—permanently lowers its priority—if it observes a two or more buffers in its output queue when it conveyed a completed buffer. The priority is lowered only once; further backups do not result in subsequent calls to nice. In the second technique, the thread calls nice up if it sees a backup and does not already have a lowered priority (as in the first). The thread also calls nice down—raises its priority—if it sees no other buffers when it conveys and if it has a lower priority. This technique requires superuser privileges.

Table 4 presents the results. Both of these techniques had similar running times. The improvement in clock time, however, is minor (approximately 1%). Still, the probability that these running times are the same as the base case is significant at 1%.

Table 5 is a measure of load balancing. It shows that in general, raising user or system time by 1 second raises clock time by 0.5 seconds in the compute-bound algorithm. A two processor SMP node would exhibit

this behavior if the load is evenly balanced. If the coefficient were different from 0.5, as is the case for the I/O-bound thread, then raising user or system time would have an effect substantially different from 0.5. The load balancer cannot balance the I/O-bound algorithm as well because it cannot control how long disk reads and writes take. A lower wall clock time correlates with a higher percent used of the CPU time. This result confirms that faster running times have higher CPU usage rates and lower latency on average.

5 Conclusion

The conclusion reviews the findings of the paper. In addition, it discusses new scheduling calls and other operating-system improvements.

The three main findings of the paper are as follows. First, using Linux kernel version 2.6 and the NPTL threads package, we observe a 3-6% performance improvement over kernel version 2.4 and the Linux-Threads package. Second, we test techniques that both take advantage of prior knowledge about whether a program is I/O-bound or compute-bound and raise and lower priorities before the pipeline begins working. These techniques, referred to as fixed scheduling, further improve performance by 5% in the case of the compute-bound columnsort algorithm. In the I/O-bound algorithm, fixed scheduling failed to yield better performance than the default scheduling. Third, we test simple, adjusting methods that do not take advantage of prior knowledge about whether a program is compute-bound or I/O-bound but rather adjust scheduling as the pipeline progresses. These techniques, called adjusting scheduling, fail to yield better performance than the default scheduling in any of our test cases.

The default algorithm for scheduling in Linux is generally efficient. The new threads package reduces the time to create and exit threads by two orders of magnitude over the old package and allows a virtually unlimited number of threads. The lack of control over scheduling prevents any custom scheduling other than raising thread and process priority, however. The ability to lengthen or shorten timeslices would allow the user to control better computational threads and to achieve uniform throughput. More control over load balancing would allow the user to ensure that a CPU is not idle part of the time in an SMP environment. These capabilities would make Linux scheduling even better.

Even if better scheduling controls are implemented, scheduling methods that require knowledge of the overall state of the pipeline may still not be practical. Such knowledge is difficult to obtain in a distributed system. One solution is to extend semaphore semantics to include atomic semaphore up and semaphore down commands that return the value of the semaphore after a change in its value. These commands would allow a stage to know the number of buffers in its incoming and outgoing queues without requiring a critical section and a lock.

FG facilitates experimentation with pipeline structure. Introducing priority in FG allows more control and more possibilities. The system calls described in this section would provide even more control. Because performance varies in test runs and it is hard to get complete theoretical understanding of the complexity of scheduling, experimentation may be the best way to improve pipeline performance.

Acknowledgements

I would like to thank Tom Cormen, my advisor, for suggesting the topic, enlightening me on research, and most of all teaching me how to write. I would also like to thank Laney Davidson, who developed FG and kindly helped me run tests on it. Tim Tregubov and Wane Cripps set up the Beowulf cluster and answered all questions. Geeta Chaudry wrote the I/O-bound columnsort program. Finally, I want to thank my family, who has supported me and made my Dartmouth experience possible.

References

- [1] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, second edition, 2003.
- [2] Thomas H. Cormen and Elena Riccio Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. To appear in PDCS-2004. Available at <http://www.cs.dartmouth.edu/FG>, 2004.
- [3] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, 1998.