Cheap operating systems research and teaching with Linux

Victor Yodaiken
Department of Computer Science
New Mexico Tech

Abstract

The advent of freely distributed operating systems, compilers, and cross-compilers in combination with the radically lowered costs of hardware has made it possible to carry out quite ambitious research in operating systems with modest resources. These same factors make it possible to teach operating systems in a hands-on fashion that offers students more than a "conceptual" understanding of operating systems. This paper briefly describes two research projects in operating system design and a collection course changes, all making use of the Linux operating system. The first research project is aimed at developing a hard real-time executive to run over Linux. The second project is aimed at design of scientific workstation operating systems and is currently focused on IO and file systems.

1 Introduction

Low-cost personal computers and no-cost full-scale operating systems permit academic research and teaching in operating systems to be made more interesting and useful. In this paper, I will discuss how the Linux operating system and other freely distributable software has been incorporated into two research projects and into the systems curriculum at New Mexico Tech. The research projects include the development of a real-time operating system — aimed primarily at instrument control — and a long range project aimed at developing operating systems for scientific workstations. The curriculum changes effect both a semi-traditional introductory operating systems class and ongoing project classes that cover more advanced material. These classes include an effort to port Linux to the PowerPC architecture and projects on real-time operating systems. All these efforts are still in experimental stages. They illustrate, however, the liberating effects of the freely distributable operating systems and compilers combined with the rapid decline in costs of computing power. OS research that previously would have only been practical in a few well funded and staffed centers and "hands-on" operating systems education that would have been enormously expensive are now within the reach of even modestly funded academic institutions.

Credits. The real-time operating system described in section 2.1 has been taken from concept to a working system by Michael Barabanov and, in particular, the design of the soft `iret` is due to him. The data on file system operation in section 2.2 was collected as part of a Masters project by Wang Jun[Jun95]. The scientific workstation project is a collaborative effort with Professor Jeff Putnum of New Mexico Tech and with members of the scientific staff at the National Radio Astronomy Observatory in Socorro, New Mexico. The new version of the software for this project is being implemented by Craig Wu. Both projects have been partially funded by a grant from Sandia Labs contract AM4413, and my work on this project is partially funded by NSF Grant CCR-9409454.

2 Research

2.1 Real-time Linux

Research in real-time operating systems has been handicapped by the paucity of data — actual problems that have been addressed and solved. Nearby scientific laboratories give us access to a collection of interesting and manageable problems in the control of scientific instruments and robotics systems. These problems range from lightning detectors to rocket control to wafer processing stations. We'd like to use these applications to test some design methodologies, scheduling algorithms, and real-time system validation techniques. Proprietary real-time operating systems are too expensive and too rigid to serve this purpose. On the other hand, a from-scratch operating system will lack the graphical displays, network interfaces, and development tools needed for any but the smallest project.

One of the problems we are investigating is a controller for an instrument that measures electrical discharges in thunderstorms. We would like to be able to read data from the instrument periodically, buffer and then write the data to disk, generate a graphical display of the data either locally or over the network, and possibly accept data from other instruments over the network. Only the first of these tasks requires hard real-time, the remainder are standard programming tasks for which Linux is well suited. Another problem concerns the control of a liquid fueled rocket mounted on a test platform. Here we need to sample and display data on numerous channels, update a remote real-time display, accept emergency shutdown commands, and perform routine control operations. Again, most of the requirements are for vanilla operating systems services, but there are hard real-time components that need reasonably precise scheduling. For example, the shutdown sequence must be precisely timed and cannot be delayed by lower priority tasks without spectacular and unwelcome results.

At first glance, and at second and third glance too, Linux seems a very unlikely answer to the hard real time requirements[SR88]. The system is large and slow, and it suffers from the standard inability to preempt kernel mode processes. Redesign of the scheduling algorithm will not help because there are unpredicatable delays caused by the kernel preemption problem, the virtual memory paging system, and the demands of interrupt driven devices. One solution then, would be a complete internal redesign, but this would defeat the purpose. The correct solution is to make Linux run as a task under a real-time executive. Linux itself will not be rewritten. Instead a real-time executive will run Linux as its lowest priority task, preempting it when needed regardless of whether Linux is running in kernel or user mode. Of course, this does not completely solve the problem. As Koopman has illustrated [Koo93], the cache heavy design of modern computer systems can cause unpredicatable behavior on the hardware level. But problems with cache and similar problems with pipelines can be contained with careful programming and relatively lax deadlines. Unpredictable behavior by the OS is a more significant problem.

Linux interrupt handling is strongly influenced by the x86 architecture. Kernel code disables all interrupts by executing a `cli` macro which executes a x86 `cli` instruction to clear the enable interrupt flag in the processor control word. Interrupts are enabled by a `sti` macro which sets the enable bit. Real-Time Linux interposes the executive between these commands and the hardware. Instead of changing a bit in the processor control word, Linux `sti` and `cli` commands set and clear a *soft* control bit in a memory variable. Instead of directly managing the hardware interrupt table, Linux manages a soft interrupt table. Hardware interrupts then are caught by the real-time executive which can pass them on to Linux or simply set a bit in an interrupt variable to indicate a pending interrupt. In particular, the clock cannot be disabled by Linux. When no real-time tasks are ready to run and the soft interrupt enable bit is set, the real-time executive will pass pending interrupts to Linux. Linux simply runs using whatever time is not needed by the real-time system.

A simplified version of the code implementing soft `sti` on the x86 architecture is reproduced in figure 1. The code essentially emulates the hardware interrupt controller. As soft interrupts are enabled, the highest priority pending interrupt takes control. When a Linux kernel process executes the `sti` macro it executes a soft `sti`. The soft `sti` first pushes data on the stack to emulate a trap so that a return from interrupt instruction will reach the label `Done` past the macro. The *soft* `iret` macro then acts to clear at least one pending interrupt as shown in figure 2.

The macro `S_IRET` begins by saving a few scratch registers. Then the interrupt bit is cleared in the hardware to hard disable interrupts. Now in the critical section the bit map of requested interrupts is ored with the bit map of enabled soft interrupts and the index of the highest order set bit is moved

```
          sti
          pushfl
          pushl $KERNEL_CS
          pushl $DoneS_STIf
          S_IRET
DoneS_STI:
```

Figure 1: Soft STI

```
        SAVE_LEAST
        cli
        movl (SFREQ),%edx
        andl (SFMASK),%edx
        bsrl %edx,%eax
        jz Endf
        S_CLI
        sti
        jmp  SFIDT(,%eax,4)
 End:   movl $1,SFIF
        sti
        RESTORE_LEAST
        iret
```

Figure 2: Soft iret

into the *a* register. If no pending interrupts were found, we simply unload the saved registers and execute an `iret` instruction, in this case, to return to `DoneS_STI`. Otherwise, we jump to the interrupt handler. In either case hard interrupts are enabled. The interrupt handler will terminate by executing `S_IRET`.

This code is not particularly efficient right now, but we still at an early stage and optimizations can come later. Both the redesign of low level Linux code that is currently taking place and our plans to port this design to the Alpha and PowerPC architectures make it inadvisable to spend too much time shaving microseconds from the execution of the non-real-time code.

Special lock-free data structures [Her91][MP89][MP91] may be used to allow real-time tasks to exchange data with Linux processes. Thus, a display program using X-Windows can run as a Linux process and display data gathered by a real-time task running under the real-time executive. In preliminary experiments on a Pentium/120, we have found that real time processes can be run on a 50 microsecond period while Linux is heavily loaded with network and disk transactions. An alternative test ran a single real-time task with a compute time of approximately 40ms and a scheduling period of 55ms while Linux was running a disk copy program and supporting a terminal display over the network. In this test, Linux continued to operate, although with vastly decreased response times — keyboard response on the remote window was about 1 second. But this is exactly the behavior we want. The non-real-time OS and applications take what compute time remains when the real time tasks are not busy.

Of course if a system contains time sensitive IO devices that run under Linux, it may fail if any real-time task is too long. But in that case, the control of that device should migrate into the real-time executive. One project here would be to move low level time-sensitive control code out of the Linux drivers entirely and centralize them in the RT executive. Currently, the timing interactions of low-level device code are discovered only when a system begins to fail. That is, standard operating systems contain a real-time component that is not designed as a real-time component.

The system is now at a stage where we expect to carry out alpha tests on the lightning instrument control and possibly on another similar project. An exec system call for real time tasks has just been completed and the fifos for data exchange are being made more sophisticated. Once the basic system is operational, we will turn to a loadable scheduler so that different scheduling algorithms may be tested. For many systems, a process that computes the rate-monotonic scheduling algorithm and loads this schedule before the real-time tasks are started could be quite useful. We hope to release a beta version of this system by March. Volunteers for beta site testing are needed.

2.2 System instrumentation

The second research project involves the design operating systems for scientific workstations. We are looking at a fairly typical application, the Astronomical Information Processing System (AIPS) developed by the National Radio Astronomy Observatory (NRAO). AIPS is an enormous collection of FORTRAN programs that are used to analyze and display radio telescope data. The size of the data sets is large and increasing — raw data from the NRAO Very Large Baseline Array telescope is measured in terabits and the partially processed data files are several gigabytes. It would not be surprising if IO turned out to be a limiting factor in performance. For obvious reasons, the NRAO is also very interested in central storage. So, the question of the practicality of networked IO over low bandwidth wires is also interesting.

There is almost no published research measuring the performance of file caching or other optimizations or even characterizing the load imposed on an operating system under scientific workstation computing. Much of the small literature on general file system loads is collected by programs similar to the the UNIX `trace` command which collect data at the user level. Ruemmler and Wilkes [RW93] offer one of the rare exceptions in their paper on data collected by instrumenting the HP-UX operating system (this paper also contains a good summary of the literature). Our first step, therefore, has been to instrument Linux so as to to find out what AIPS and other programs need from the operating system. Once we have collected sufficient data, we will be able to evaluate design options.

One interesting component of the problem is that AIPS is being rewritten in C++. We expect that this will radically change the IO demands of the system by putting more pressure on the virtual memory system and by making less use of temporary files. Again, we have found no published research on the effects of such transitions on the IO characteristics of scientific software.

To start, we have instrumented a version of Linux to trace all I/O system calls and the operation of the buffer cache. Every file operation — read, write, seek, *etc.* — and every buffer cache access is logged. For file operations, we currently log the start and end times in units of 100 microseconds, the inode, file position, and device. We also log the hit rate per a selectable number of buffer cache accesses.

It is our intention to make the trace system an easily installed patch to Linux so that it can be used to gather data from a wider set of applications. We hope to be able to use the net to collect a truly representative sampling. The first version of this project had a fixed log file that was written to by the kernel whenever internal buffers filled. The data file was placed on a second disk so that writing log information would minimally skew our data. A new version is currently being implemented which relies on a daemon program to periodically empty kernel log buffers. The rewrite was designed to minimize the amount of kernel code needed for logging and to permit remote collection of log data. The newer version also takes advantage of the internal timer on the Pentium for more precise measurement of time intervals.

Early work has been encouraging although it has not revealed any major surprises. Figure 4 shows the pattern of reads seen on a *make* of the Linux kernel. Figure 3 shows the pattern of reads seen on a run of the AIPS DDT benchmark which exercises several features in AIPS and is designed to provide some measure of how well a system runs AIPS. The DDT exercise used moderate size files in the several megabyte range and runs for about 20 minutes on a P90 with 24 megabytes of memory. The results shown in the two figures were typical of several hundred runs taken to eliminate artifacts. The preponderance of sequential file accesses in AIPS and the contrast with the kernel *make* is quite clear. One would expect from this data that aggressive read-ahead would be a good strategy, but much closer analysis is needed. One possible problem with aggressive read-ahead is that it might displace the code of frequently executed processes from the buffer cache.

A different perspective on the same two loads can be seen from figures 6 and 5 showing the hit rate on the buffer cache. As one might expect, the sequential accesses in AIPS results in uneven performance. LRU policies do not work well in this situation although they work very well for the kernel *make*. What is surprising about the AIPS results are the significant periods in which the cache satisfies all requests. These counteract the lows to produce a deceptively high cumulative hit rate of close to 90%.

More measurements will be ready soon and will be put on the web page. The kernel patch and analysis programs will be also made available over the web page when they are reliable and properly packaged.

3 Concrete Operating Systems

The mathematics textbook of Graham, Knuth, and Patashnik [GKP89] begins with a quotation from J. Hammersley that bears requoting.

> *... what we should ask of educated mathematicians is not what they can speechify about, nor even what they know about the existing corpus of mathematical knowledge, but rather what they now do with their learning and whether they can actually solve mathematical problems. In short we look for deeds not words.* [Ham68]

The same standard should be applied to computer scientists. But the traditional operating systems class does little to prepare students to "solve problems". Instead emphasis is placed on general concepts and on a high-level perspective that ignores the critical implementation details. The reason for this emphasis is that teaching students how a particular operating system works is less important than teaching them the fundamental principles behind all operating systems. It can be argued that most students will never need to program operating systems internals and thus are better served by a high level course. This seems to me to be a philosophy suited to producing managers and coders, not computer scientists capable of creative work. Just as one would not attempt to teach the principles of the differential calculus without solving problems, one should not try to teach operating systems without solving problems. The details are needed in order to animate the principles.

Students who have taken an introduction to operating systems class often have only the most idealized view of synchronization and are completely unable to construct a semaphore from a test-and-set primitive or to analyze the possible deadlocks in a realistic sleep/wakeup system. Time-sharing and virtual memory, in particular, remain mysteries until the students get their hands dirty actually working with code and bare machines. This problem is worse for advanced courses where, for example, understanding distributed shared memory depends on a detailed understanding of the mechanisms of memory management, network device drivers, and disk IO.

To give students "hands on" experience, we have incorporated Linux into the junior level operating systems course and several advanced classes. We are now considering how to move some of the simpler OS projects, such as writing system calls, into lower level classes as well.

3.1 You *are* expected to understand this

The primary equipment for the current version of our junior level operating systems class consists of a collection of 486/Pentium workstations running both Linux and Novell Netware. Because we have to share these workstations with other undergraduate classes and even other departments it is impossible to allow students to experiment with the operating system code or to gain access to the raw file system structures or to the raw device interfaces. Since this is exactly what we would like students to do for their OS projects, we have equipped the workstations with rack mounted drives. The drives can be unlocked and swapped out in a few seconds by any of the user consultants employed by the computer center. Students in the OS class are divided into small development teams of between 2 and 4 members. Each team is given a bootable Linux disk for the semester. Copies of the disk are kept on the network so that when project teams destroy their own file systems or otherwise render the disk unbootable they can get a fresh start.

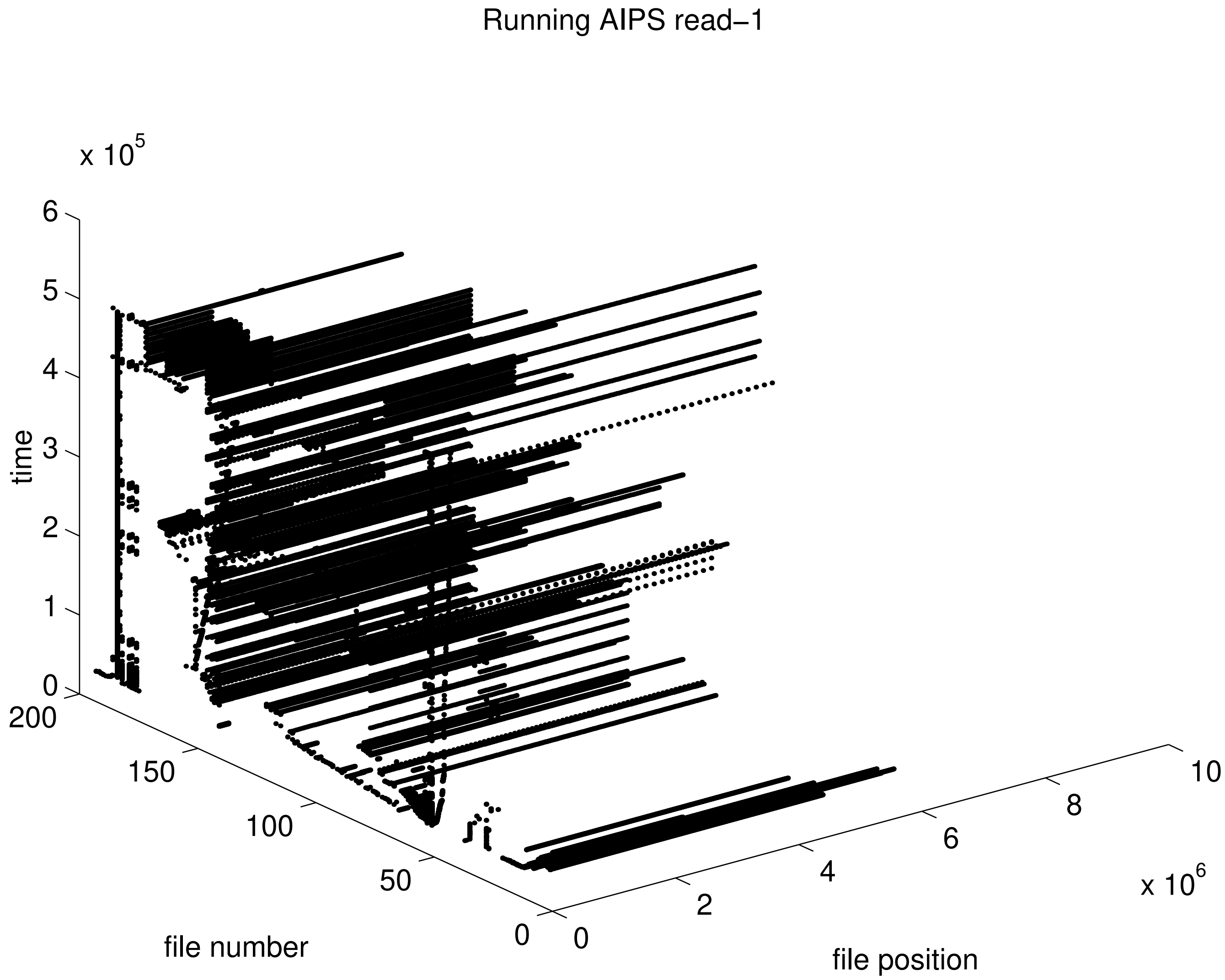

Figure 3: Reads for AIPS

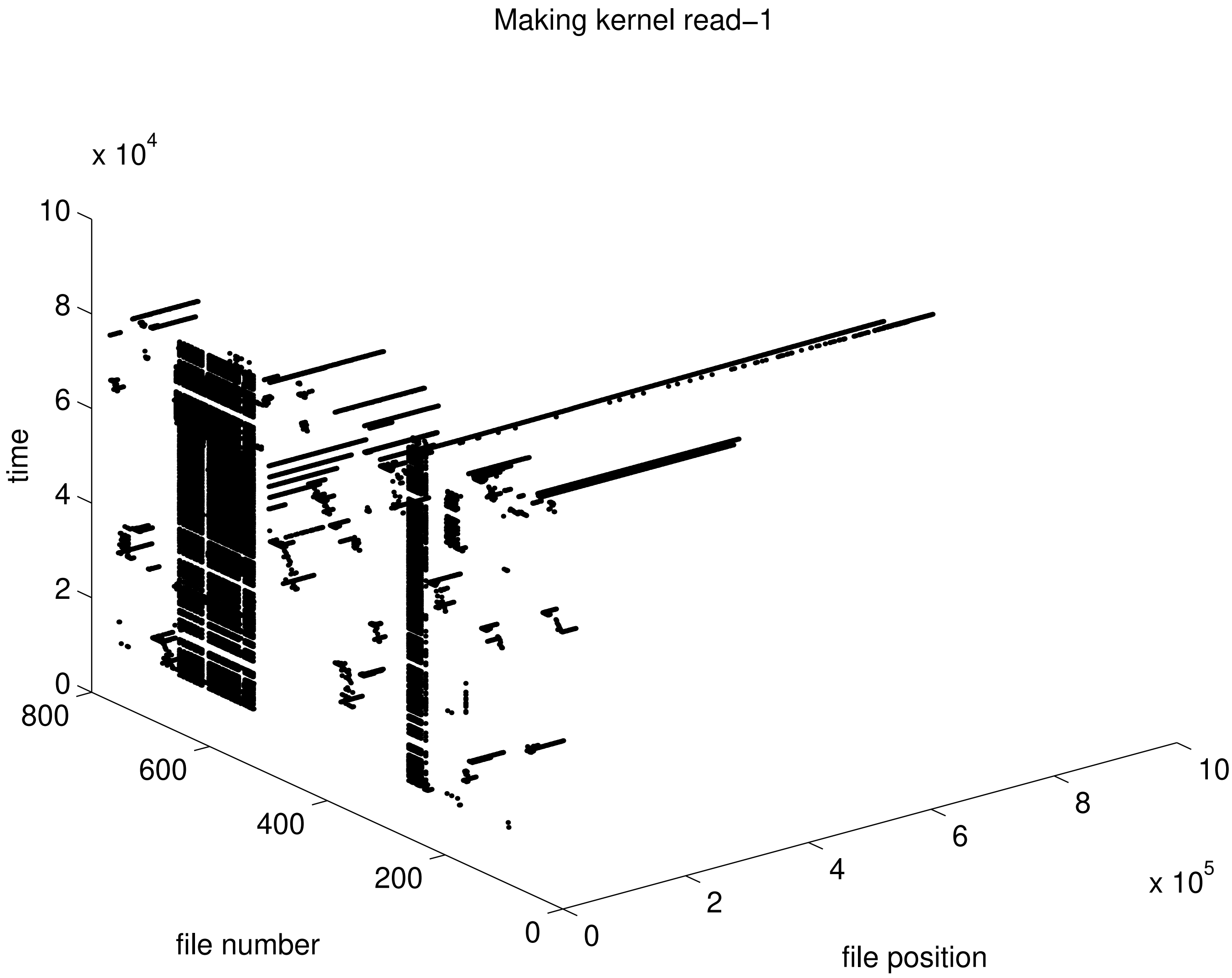

Figure 4: Reads for kernel make

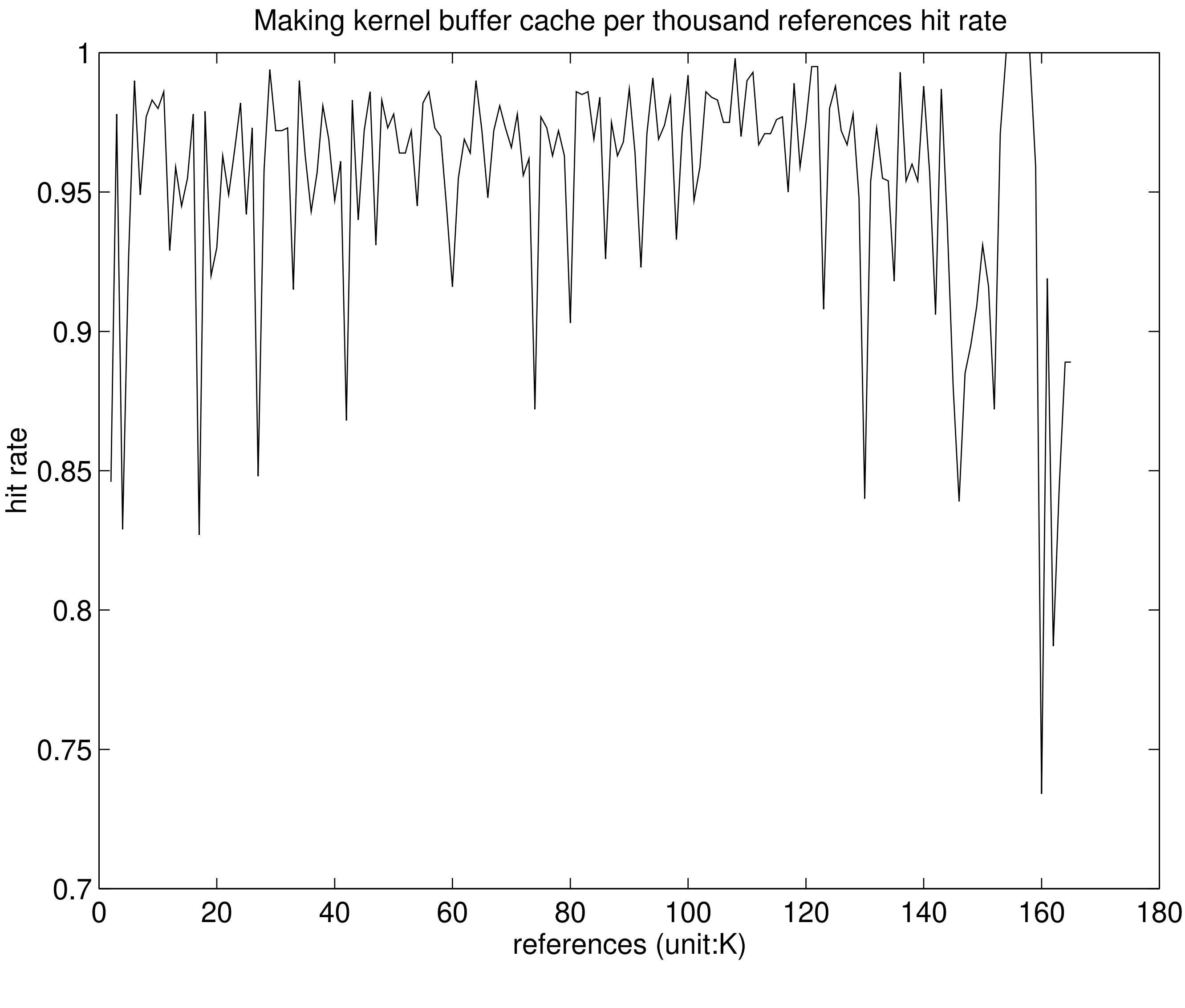

Figure 5: Buffer hit rate for kernel make

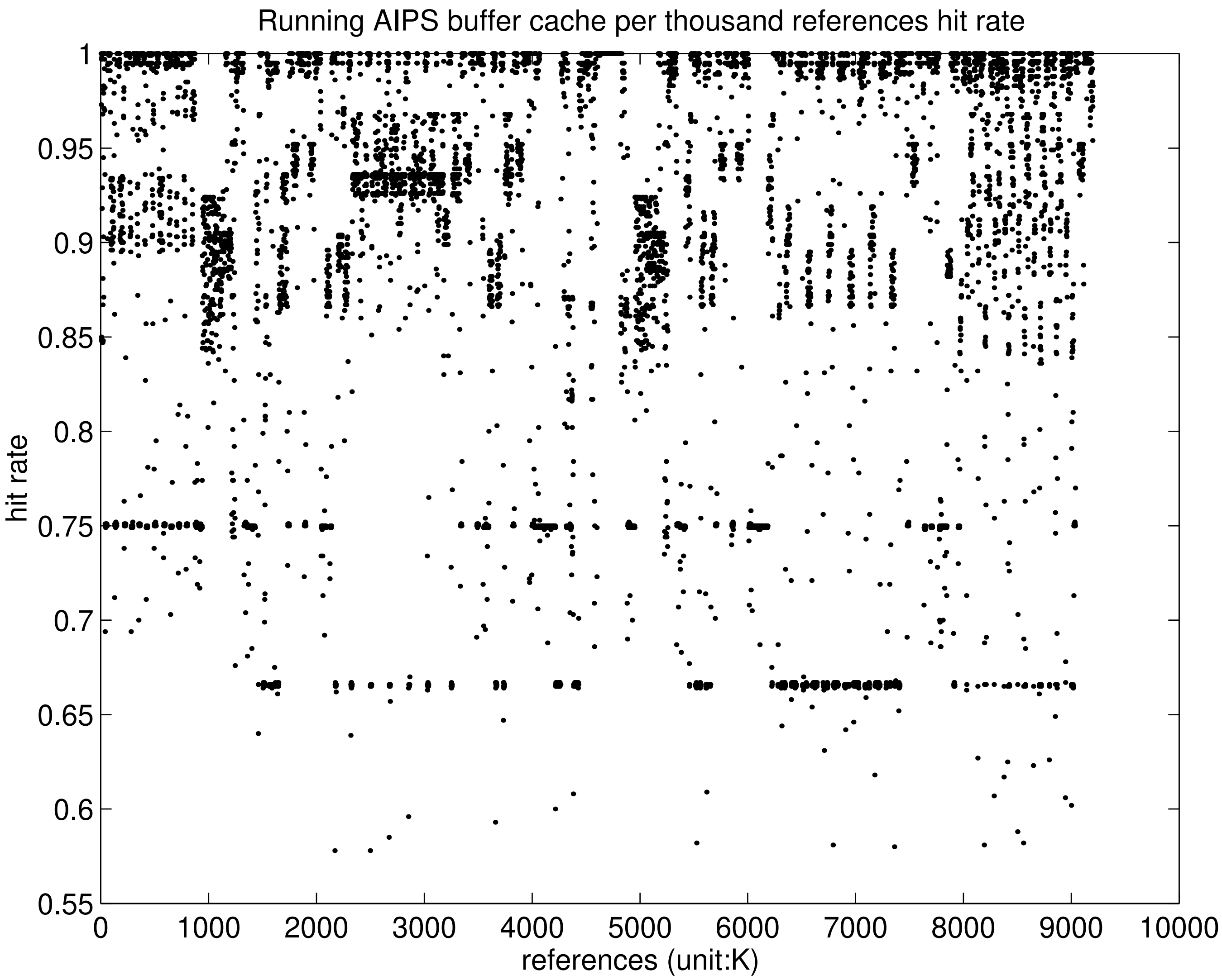

Figure 6: Buffer hit rate for AIPS

The issue of security is dealt via the usual technical means and by warnings that anyone found exploiting the security holes will find themselves in serious academic and legal difficulties. As more and more students have Linux capable machines of their own (70% of our freshman class this year arrived with computers) we expect to be able to reduce the number of disks reserved for the class.

The OS course provides students with a series of projects of increasing difficulty. We begin with some warm up exercises, making sure that students know how to recompile the kernel and add simple system calls. Later projects range from copying data between kernel and user space (reimplementing Linux utilities) to controlling devices and switching tasks. An example of a late course project is to implement system calls that can read and write the floppy disk — without using any of the Linux components that do this in the ordinary course of events. One of the projects scheduled for this year's course is to reimplement the core process switch code without using the x86 task switch instructions. In addition to the projects, Linux is brought directly into the course lectures to illustrate such concepts as semaphores and virtual memory management.

Linux is not the most elegantly designed or coded operating system. In fact, the code quality is uneven and the design shows the stresses of rapid growth. For our purposes, this is not a disadvantage. First, students see the operating system internals as something written partially by other students and not as a mysterious object produced whole by higher powers. Second, the results of prior design decisions, the effects of peculiar hardware, and changes in the design are visible in the code. During the course, we discuss how the system design could be improved, why some parts of the code are so complex, and the relationship between OS design and computer architecture. Finally, the very opaqueness of some of the code requires deeper reading. One cannot simply look at the surface structure and gain a superficial understanding. Students are required to really study the code and think about what it should be doing. And this is the purpose of the course.

3.2 Advanced courses.

Advanced operating systems courses here are taught as seminars. The goal here is to give students some experience in research and development in place of lecture. Students read current papers and books and take part in projects. The simpler projects that are appropriate for the introductory course are not appropriate for higher level courses. Currently our primary project is a port of Linux to the PowerPC architecture. We are also setting up projects involving alpha testing of the real-time OS. Within the port project we have been able to provide students with mini-projects tailored to their interests. Several of the students have worked on low level kernel design and debugging, a small team has worked on a redesign of the memory management system, and other students have worked on developing and porting tools. The challenge here is to keep the project small enough so that individual students can take responsibility for complex pieces and, not coincidentally, so that management requirements are minimized.

References

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[Ham68] J. M. Hammersley. On the enfeeblement of mathematical skills by 'modern mathematics' and by similar soft intellectual trash. *Bulletin of the Institute for Mathematics and its Applications*, 4(4):68–65, October 1968.

[Her91] M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages*, 13:124–149, 1991.

[Jun95] Wang Jun. A linux file system file i/o system calls and buffer cache trace package. Technical report, New Mexico Tech, july 1995.

[Koo93] P. Koopman. Perils of the pc cache. *Embedded Systems Programming*, 6(5), may 1993.

[MP89] H. Massalin and C. Pu. Threads and input/output in the synthesis kernel. In *Proc. Twelfth ACM Symp. on Operating Sys., Operating Systems Review*, page 191, December 1989. Published as Proc. Twelfth ACM Symp. on Operating Sys., Operating Systems Review, volume 23, number 5.

[MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[RW93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *USENIX Technical Conference Proceedings*, pages 405–420, San Diego, CA, Winter 1993. USENIX.

[SR88] John A. Stankovic and Krithi Ramamritham. *Hard Real-Time Systems*, volume 819 of *IEEE Tutorials*. IEEE, 1988.

About the author

Victor Yodaiken is an assistant professor of Computer Science at the New Mexico Institute of Mining and Technology (New Mexico Tech). **Mail:** Department of Computer Science. Speare 4. New Mexico Tech. Socorro, New Mexico 87801.

Email: `yodaiken@nmt.edu`

URL:
`http://www.cs.nmt.edu/~yodaiken`

URL for Real-time projects:
`http://www.nmt.edu/~realtime`

URL for Scientific OS project:
`http://www.cs..nmt.edu/~yodaiken/os/os.html`