A Performance Comparison of UNIX Operating Systems on the Pentium

Kevin Lai and Mary Baker
Stanford University

A Performance Comparison of UNIX Operating Systems on the Pentium

Kevin Lai and Mary Baker

Stanford University

Abstract

This paper evaluates the performance of three popular versions of the UNIX operating system on the x86 architecture: Linux, FreeBSD, and Solaris. We evaluate the systems using freely available micro- and application benchmarks to characterize the behavior of their operating system services. We evaluate the currently available major releases of the systems "as-is," without any performance tuning.

Our results show that the x86 operating systems and system libraries we tested fail to deliver the Pentium's full memory write performance to applications. On small-file workloads, Linux is an order of magnitude faster than the other systems. On networking software, FreeBSD provides two to three times higher bandwidth than Linux. In general, Solaris performance usually lies between that of the other two systems.

Although each operating system out-performs the others in some area, we conclude that no one system offers clearly better overall performance. Other factors, such as extra features, ease of installation, or freely available source code, are more convincing reasons for choosing a particular system.

1. Introduction

Many research, development, and product groups that have traditionally run on a UNIX workstation-based computing platform are now moving to a PC-based platform. Organizations can afford to purchase many more PCs than workstations on their equipment budgets. The x86 architecture's low cost, good performance, and expandability give it economies of scale that will reinforce, and be reinforced by, its popularity for at least a few more years.

As part of this transition, these groups must decide whether to move to a PC operating system such as Microsoft Windows, or whether to continue running a UNIX-compatible operating system. For many groups, including our own, dependence on the performance, features, and tools available in the UNIX environment make it sensible to run an x86 implementation of UNIX.

The next step is to choose between the available UNIX-compatible operating systems. In particular, our group is interested in free implementations of UNIX, because new ideas can be implemented without a non-disclosure agreement and the results can be freely distributed. We were concerned, though, by comments describing the free implementations as toy systems, unsupported and with poor performance and reliability. This argument has been used against Linux especially, since its source is not derived from as respected an ancestor as BSD UNIX 4.4. We decided to compare a few of the systems ourselves to determine the validity of these comments. This paper presents our results.

We benchmarked Linux, FreeBSD, and Solaris. Linux and FreeBSD are the most popular free implementations of UNIX that run on our hardware. Solaris is the least expensive commercial implementation known to support the hardware on our system. (Several other popular systems, such as NetBSD and BSDI's BSD/OS, do not currently support our SCSI controller.) We evaluated only the most recent major releases of the systems currently available, since these are the most relevant and accessible versions for most people.

Our benchmarks are by no means exhaustive; we only measure the performance of tasks and workloads that are important to us. The benchmarks test system call latency; context switch latency for varying numbers of processes; memory bandwidth; file system performance; network bandwidth for pipes, UDP and TCP; and NFS performance on a file system workload.

Our results show that:

- Linux has the best performance on file metadata operations because it updates metadata asynchronously;
- FreeBSD has the best network performance;
- Solaris' performance generally lies between that of the other two systems; and
- All three systems' library routines for setting and copying memory fail to deliver the full underlying Pentium memory bandwidth.

Given these mixed performance results, we believe overall performance is not a sufficient argument for choosing one of these operating systems over the others. Performance on specific tasks may make the difference for some users, but the systems are competitive overall, and particular performance problems are likely to improve in future releases of all three systems.

Other factors may be more important, including extra features, licensing arrangements, ease of installation, and available support. Solaris provides more sophisticated features, including multiprocessor support, than the current free versions of UNIX, and this will be sufficient argument in its favor for many users. The freely available source code and free licensing of Linux and FreeBSD motivate others to choose one of these systems. Ease of installation and the level of available support are also important, and we include a section in this paper on our relative experiences installing and configuring the systems on our hardware. A large user community and free access to software and other resources over the Internet combine to provide reasonable support for the free implementations.

We hope these results will contribute helpful information for those choosing a UNIX-compatible operating system for the PC. We also hope the results, where negative, will reveal areas for improvement in future versions of these systems.

The remaining sections of this paper describe our benchmarking platform and methodology, our results in more detail, and our experiences installing and using the three systems.

2. Benchmarking Platform

Our goal was to compare UNIX operating systems on identical PC hardware performing some standard tasks of interest to us. The relative performance of the systems on identical tasks is more important to us than the absolute best performance that could be achieved for any individual system through system-specific tuning. For comparison purposes, and because we only have source code available for two of the three systems, our benchmarking methodology is the “black box” approach. We usually attempt to explain curious results through external testing and benchmarking rather than investigations of kernel code or profiling.

2.1 Operating Systems

For operating systems, we chose UNIX systems that have a reasonably large user base and development group, run on our hardware, and cost less than $100 in the summer of 1995. Linux, FreeBSD and Solaris met these criteria.

Linux is a free version of UNIX distributed under the terms of the GNU General Public License. Roughly speaking, this means that works derived from the Linux kernel must be distributed with source code and a fee can only be charged for the transfer of a copy and not ownership of a copy or licensing of a copy. Linux was created by Linus Torvalds when he was a student at the University of Helsinki in Finland. Since then, many other developers have contributed to it. Its code is not derived from BSD or System V, but has features of both and is also generally compliant with Posix.1.

FreeBSD is a free version of UNIX distributed under the terms of a University of California license. This license requires that the copyright notice be included in both source and binary forms of any distribution and any advertising must mention that the product contains University of California, Berkeley code. FreeBSD is derived from the BSD 4.4-lite release by the Computer Systems Research Group at U.C. Berkeley. Like Linux, many developers have contributed to it. It is fully compatible with BSD-style API programs.

Solaris is a commercial version of UNIX developed by Sun Microsystems, Inc. As a commercial system, it costs money, but we purchased it on CD-ROM for $99; this made it cheaper than other available commercial versions of UNIX. Including program development utilities, the total cost was $244. No source code is included. Solaris is mainly a System-V-based UNIX, but includes BSD-compatibility header files and libraries. Solaris runs on both the x86 and Sparc architectures. It has a fully preemptive multi-threaded kernel and support for multi-processor systems.

Of these systems, we chose the most recent major release that was commonly available at our cut-off date of October 31, 1995. Consequently, we did not test unreleased or beta versions. For Linux, we used version 1.2.8 from the Slackware Distribution. For FreeBSD, we used version 2.0.5R. For Solaris, we used version 2.4. Of course, subsequent versions of any of these systems may perform very differently from the versions we tested.

2.2 Hardware

We chose the most cost-effective high performance hardware that was available to us in May, 1995. Our benchmarking platform is `tnt.stanford.edu`, an Intel Pentium P54C-100MHz system with 32 megabytes of main memory and two 2-gigabyte disks. It has a standard 10-Megabit/second Ethernet card (3Com Etherlink III 3c509). The motherboard is an Intel Plato. The disk controller is an NCR 53c810 PCI

SCSI card, which has no on-board cache. One disk is a Quantum Empire 2100 SCSI disk, and the other is an HP 3725 SCSI disk.

On the first disk we installed the various operating systems, each in its own partition. The partitioning of the disk is shown in Table 1. We made each partition 200 megabytes more than the minimum recommended for each OS. Since we installed Linux last, its partition received the remainder of the disk and is larger than it needs to be. (Otherwise, its partition would be the same size as FreeBSD's.)

OS	Version	Size (megabytes)
DOS/Windows	6.2/3.1	250
Solaris	2.4	700
FreeBSD	2.0.5R	400
Linux	1.2.8	600

TABLE 1. Disk Partitioning: This table lists the versions of the operating systems benchmarked and shows how `tnt.stanford.edu`'s disk is partitioned.

We used the second disk to ensure that the unequal partitioning of the first disk does not affect our file system performance results. All benchmarks that manipulate files refer to files on this second disk. We create a fresh 200-megabyte file system on this second disk between different benchmarks, but use the same file system for different iterations of the same benchmark. In this way, each of the systems has the benefit of a fresh file system for its use, but any problems it suffers from its management of that file system during the benchmark will remain.

3. Benchmark Overview

We took our benchmarks from a variety of sources. The system call, context switch, and file create/delete microbenchmarks are derived from those John Ousterhout used in [Ousterhout 90] to compare the effect of RISC and CISC architectures on operating system performance. The Modified Andrew Benchmark, developed from the Andrew Benchmark written by M. Satyanarayanan at CMU [Howard 88], was also used in Ousterhout's experiments. To get a more complete picture of context switch and memory system performance, we rewrote Ousterhout's benchmarks in those areas, and we modified the Modified Andrew Benchmark for better portability.

To test file system bandwidth and seek performance, we used Tim Bray's `bonnie` benchmark [Bray 90].

Our network benchmarks are a combination of some of the network benchmarks from Larry McVoy's `lmbench` package [McVoy 95] and the `ttcp` TCP/IP benchmarking program.

We tied all of these benchmarks together with Tcl scripts [Tcl 90] and ran each benchmark program twenty times. Most of the benchmark programs themselves also loop several times over their respective routines, and we report the average result for the total number of iterations. All benchmarks were executed in single-user mode. When run in multi-user mode, the benchmarks exhibited slightly higher variance.

4. System Call

We measure system call performance since the system call is one of the basic mechanisms by which the operating system provides functionality to applications. Our results show that Linux has the fastest basic system call, followed by FreeBSD and then Solaris.

We estimate system call time by calling `getpid()` in a loop. We then divide the total time by the number of calls. This is an optimistic estimate of the time to make a system call, because the loop allows successive `getpid()` calls to benefit from data and instructions cached the first time through the loop. Furthermore, `getpid()` does so little work in the kernel that all of the application data and code can remain in the processor cache. Given the few instructions executed in the loop and the small amount of data accessed, the entire loop could execute in an eight-kilobyte instruction and eight-kilobyte data processor cache. This estimate, although optimistic, is fine for our purposes, because we want to measure the relative performance of the systems on the same hardware.

OS	Time (μseconds)	Std Dev	Norm.
Linux	2.31	0.10%	1.00
FreeBSD	2.62	0.08%	0.88
Solaris	3.52	2.95%	0.66

TABLE 2. System Call: This table lists the time to make the `getpid()` system call, averaged over twenty runs of 100,000 iterations each. Lower times are better. The Norm. column lists the speed of the benchmark, normalized to the best time among the systems. This gives a proportional ranking for the systems in which higher numbers are better.

Table 2 shows the results. Examination of the source code for performing a system call reveals that Linux has slightly more optimized assembly instructions than FreeBSD. Solaris' extra features and multi-threaded fully-preemptive kernel contribute to its longer system call time [McVoy 95].

5. Context Switching

Context switch time is important for file and database servers and is increasingly important for Internet servers that must sometimes service hundreds of simultaneous connections. We determined that Linux has the best context switch time of the three systems with fewer than 20 processes, while FreeBSD is faster with more processes. Solaris context switches more slowly in all cases.

We used our own context switching benchmark, `ctx`, based on ideas from the original Ousterhout context switching benchmark, `cswitch`, and Larry McVoy's `lmbench` suite. `Ctx` estimates the context switch time by measuring the time to `write()` a byte to another process and then `read()` the one-byte reply. For more than one process, the byte is passed around in a round-robin fashion through a ring of processes. The overhead of the pipe operations is included in our results.

As with the `getpid` benchmark, most, if not all of the code and data in the loop could be cached in the first-level CPU cache, since the Pentium architecture has a physically addressed first- and second-level cache [Anderson 93] that does not need to be flushed during context switch. In addition, the context switch benchmark is written as one program that forks into the required number of processes. Code-sharing

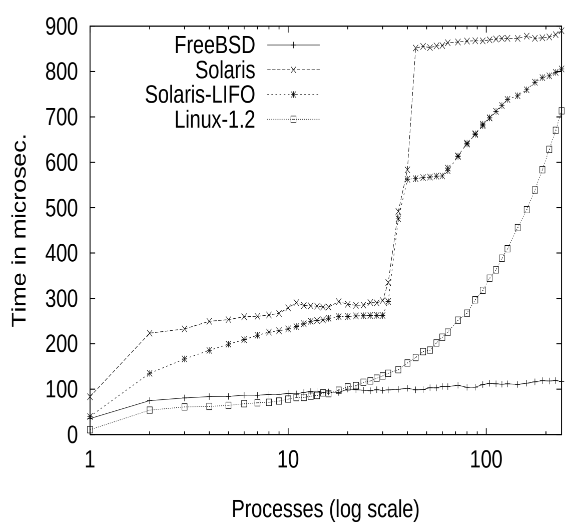

FIGURE 1. Context Switch: This figure shows the time in microseconds to make a context switch as a function of the number of active processes in the system. The results include the overhead of the pipe operations used in the benchmark. Solaris-LIFO passes a byte back and forth through a chain of processes. The other benchmarks pass the byte around a ring of processes. Data points are the average values over twenty runs with 50,000 context switches each. At two processes, the standard deviations were 3% for Linux, 4% for FreeBSD, and 9% for Solaris.

between the processes increases the probability that the entire loop fits into the cache. As with the system call benchmark, this lower-bound estimate is fine for our purposes, since we want to compare the systems.

Figure 1 shows that FreeBSD context switches at almost the same speed no matter how many active processes there are. In contrast, Linux context switching time increases linearly with the number of active processes, suggesting that the Linux scheduler must search an O(number of processes) data structure during a context switch. Aside from the linear time required to traverse this data structure, Linux has very little overhead, so it context switches faster than FreeBSD for fewer than 20 active processes.

Solaris context switches more slowly in all cases. This is in part due to slower pipe performance (as described in Section 9.1). We measured the overhead of sending a byte from a process, through a pipe, and back to the same process. This took 80 microseconds. The time for Solaris to context switch between two active processes is 220 microseconds. Therefore, without the pipe overhead, the estimated time to context switch would be about 140 microseconds. For the same number of processes, FreeBSD and Linux context switch in 80 and 55 microseconds, respectively. The additional overhead is largely due to the extra work that Solaris' multi-threaded fully preemptive kernel scheduler must perform [McVoy 95].

Another interesting result for Solaris is the large increase in context switch time at about 32 processes. We hypothesized that a system resource overflows at that point. In order to test this, we changed the `ctx` benchmark so that its processes pass the token in LIFO order, back and forth through a chain of processes. We expected that this would take advantage of a system table with a limited number of elements and show a gradual increase in context switch time per process for more than 32 processes, instead of a steep one. As shown in Figure 1, this is only true for more than 64 processes. We still see a sharp increase at 32 processes. This behavior does not occur for Solaris running on other architectures [Bonwick 95], so it is not caused by the machine-independent portion of the Solaris scheduler.

6. Memory Bandwidth

As CPUs become faster without a matching speedup in memory, the time to access memory may dominate the execution time of non-I/O-bound programs, including the operating system. We therefore wanted to know which of these systems best exposes the underlying Pentium memory performance. To do this, we compared the performance of the systems' `libc`

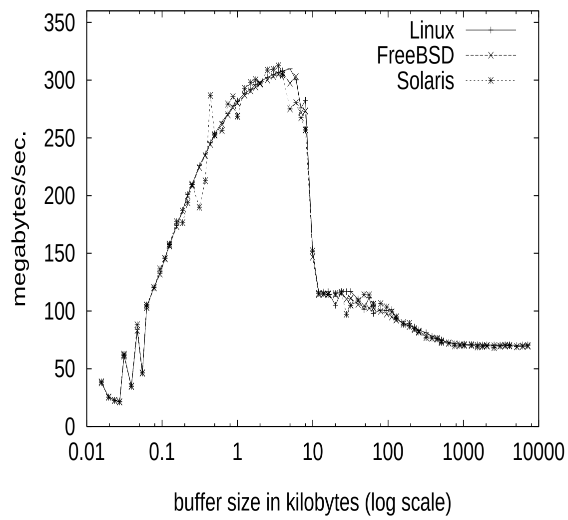

FIGURE 2. Custom Read: This figure shows memory read bandwidth as a function of the size of the buffer read. The humps at 8 kilobytes and 256 kilobytes reveal cache effects. The results are averaged over twenty runs.

`memcpy()` and `memset()` routines. Essentially the same routines are also used by the operating systems themselves. We also wrote our own easily-modifiable custom routines for reading/writing/copying data to help us better understand the behavior of the system library routines.

For all the benchmarks, one or two buffers of varying sizes are used to read, write, or copy data. The same buffers are used over and over again until eight megabytes of data have been transferred, since this gives us direct information about the effects of the hardware caches.

Our results show that none of the systems adequately delivers the Pentium's memory write performance. For example, the Pentium can copy data at over 160 megabytes/second using a prefetching copy routine, yet none of the systems we tested have implemented such a routine. As described below, the prefetching routines address the fact that the Pentium does not have a write-allocate cache. Without this optimization, the same routines copy data at about 40 megabytes/second.

6.1 Memory Read

As shown in Figure 2, the Pentium can read at a peak bandwidth of slightly over 300 megabytes/second from its first-level cache, i.e., it is reading approximately one word every 13ns or four words every 50ns. Since our Pentium runs at 100Mhz, 50ns corresponds to five clock ticks. Given the Pentium architecture's dual issue pipeline, this is a reasonable result.

For buffer sizes larger than 8 kilobytes, the Pentium's performance drops off significantly because that is the size of its first-level data cache.

The next plateau is from approximately 10 kilobytes to 256 kilobytes, where the bandwidth is 110 megabytes/second. This is due to the second-level cache. Finally, read performance levels out at approximately 75 megabytes/second.

6.2 Memory Write

Given the good read performance of the systems, we were initially surprised by the poor `memset()` write bandwidth, which did not reach even 50 megabytes/second (Figure 3).

This poor write performance is due to the lack of a write-allocate cache on the Pentium [Intel 94]. In a write-allocate cache, when a write is done to a line that is not in the cache, that line is brought into the cache while the write is being done, so that later writes to the same line will hit in the cache. We speculated that prefetching the cache lines in software could improve performance on a chip without a write-allocate cache.

In order to test this hypothesis, we coded two versions of a custom memory writing routine, one to do a normal copy and the other to prefetch cache lines as the write is taking place. The results of our non-prefetching custom write benchmark are shown in Figure 4 and are very similar to the system `memset()` results. In comparison, the prefetching version improved the Pentium's performance dramatically, as shown in Figure 5. The peak write bandwidth improved to 310 megabytes/second.

6.3 Memory Copy

As with the `memset()` function, the `memcpy()` routine on the x86 systems has not been optimized to prefetch, so the results for `memcpy()` in Figure 6 resemble those for a custom copy routine without prefetching (Figure 7).

As with the custom write routine, we re-coded the custom copy routine to do prefetching and achieved a peak of over 160 megabytes/second in copy bandwidth, as shown in Figure 8. This is equivalent to 320 megabytes/second in total bandwidth, which approaches the peak set by the custom read routine.

6.4 Memory Anomalies

The spikes at the low end of the figures for all of the custom memory benchmarks are a consequence of the way the memory benchmarks are written. The mem-

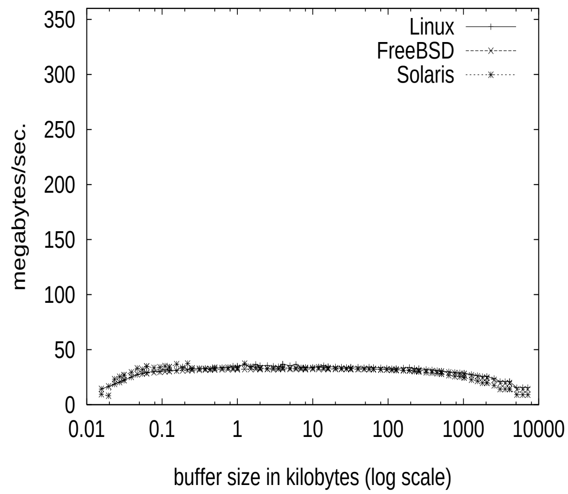

FIGURE 3. Memset: This figure shows memory write bandwidth using `memset()` as a function of buffer size. the results are averaged over twenty runs.

ory benchmark's inner loop actually consists of two loops. One loop performs the appropriate operation on 16 bytes of data per iteration and iterates

$$\left\lfloor \frac{totalBytes}{16} \right\rfloor$$

times. The other loop performs the same operation to the remaining 0-15 bytes at one iteration per byte. When the buffer size is such that 15 bytes have to be processed in the second loop, the memory bandwidth dips, since the second loop is so much more inefficient than the first.

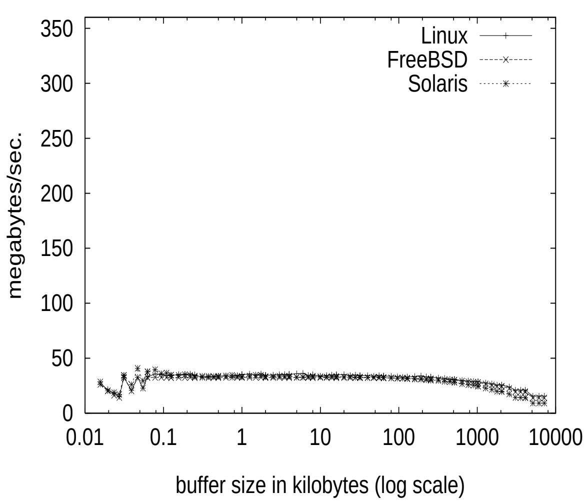

FIGURE 4. Naive Custom Write: This figure shows memory write bandwidth using a custom memory writing routine as a function of buffer size. The results are averaged over twenty runs.

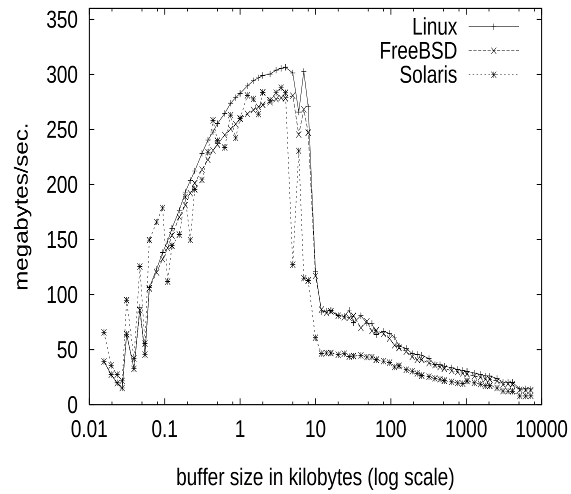

FIGURE 5. Prefetching Custom Write: This figure shows memory write bandwidth with prefetching as a function of buffer size. The results are averaged over twenty runs.

6.5 Summary

Applications programmers rely on the system to shield them from the unnecessary details of the machine while delivering its performance. In this duty, the x86 operating system libraries that we tested fall short in exposing the full memory write bandwidth of the Pentium.

Adding prefetch to memory routines in software used across all the processors in the x86 family is not necessarily appropriate, since some members of the x86 family have a write-allocate cache. Therefore, statically-linking applications with prefetching memory routines might cause these applications to perform worse on some CPUs. However, adding prefetching memory routines to dynamically-linked libraries would allow maximum performance on each machine, because the decision about which library to link with is made at run time. Similarly, adding prefetching memory routines to the kernel allows maximum performance, since the kernel can be compiled separately for individual machines.

7. File System Performance

We benchmarked the ability of the operating systems to satisfy the needs of two types of I/O-intensive workloads. One workload, which includes applications such as video playback and editing and large databases, accesses large files and therefore needs high raw bandwidth and fast seeking. The other workload includes program compilation and accessing, creating and deleting many small files. It therefore stresses the file system's ability to manipulate the file

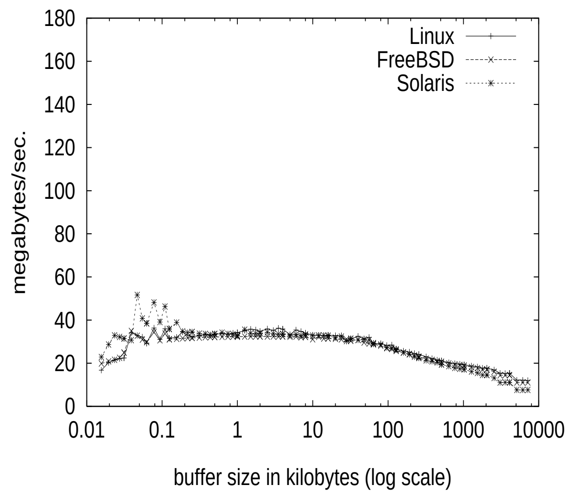

FIGURE 6. Memcpy: This figure shows memory copy bandwidth using `memcpy()` as a function of buffer size. The results are averaged over twenty runs.

In order to isolate the differences between operating systems, we used two disks to do the file system benchmarking. The Quantum 2100S contains the operating systems themselves and the code for the benchmarks. We used the HP 3725 as the actual benchmarking disk. We used the same partition for each system and benchmark. After each benchmark (`bonnie`, `crtdel`, `MAB`), we re-made the file system on that partition to ensure that the previous benchmark could not affect the allocation of blocks during the current benchmark.

All of the systems have a dynamically sized buffer cache that trades physical pages for buffer cache

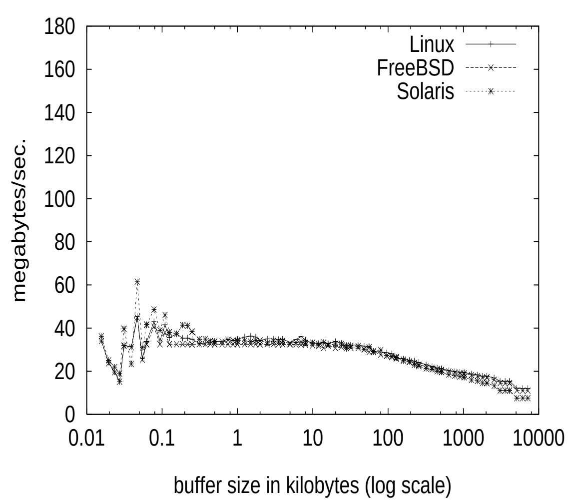

FIGURE 7. Naive Custom Copy: This figure shows memory copy bandwidth using a custom copy routine as a function of buffer size. The results are averaged over twenty runs.

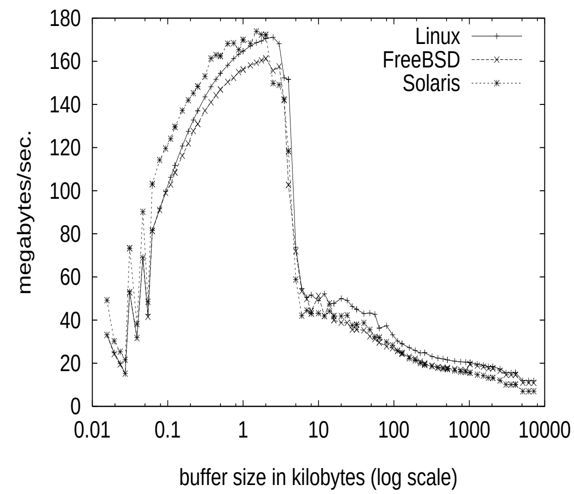

FIGURE 8. Prefetching Custom Copy: This figure shows memory copy bandwidth with prefetching as a function of buffer size. The results are averaged over twenty runs.

pages during intensive disk accesses; as a result, they generally do well when the data set accessed is small enough to fit in main memory. Our results show that FreeBSD and Solaris perform well for large files. For small file workloads, characterized by a high percentage of metadata operations, Linux is an order of magnitude faster than the other systems, because it performs file metadata updates asynchronously.

7.1 Large-file Benchmarks

We wanted to test three aspects of large file performance: 1) sequential read bandwidth, 2) sequential write bandwidth, and 3) time to seek to a random block in a file and perform an I/O operation on it. To do this, we used the `bonnie` benchmark, written by Tim Bray. `Bonnie` creates and writes to a file of the user-specified size, reads from it sequentially, and then seeks randomly within it. We ran `bonnie` with file sizes from two to 100 megabytes to test performance for files that do and do not fit in the buffer cache. Unlike some of the other benchmarks we used, `bonnie` performs each of its operations only once per invocation. We invoke it 20 times per file size.

As shown in Figure 9, all three systems cache the file for sizes up to 20 megabytes out of 32 megabytes total on the machine. This is because all three systems allow a trade-off between memory pages and the file cache, so the file cache can grow to accommodate large files.

For files in the buffer cache, FreeBSD reads between 5% and 15% faster than both Linux and Solaris. For files outside of the buffer cache, Solaris has the best read bandwidth. Large sequential

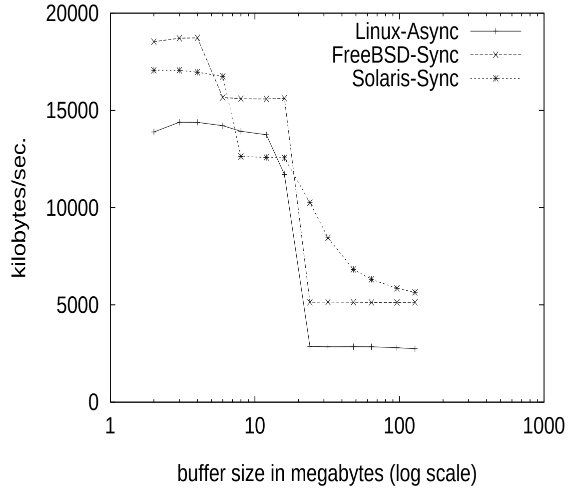

FIGURE 9. Bonnie Read: This figure shows file system sequential read bandwidth in Megabytes/second as a function of file size. The results are averaged over twenty runs.

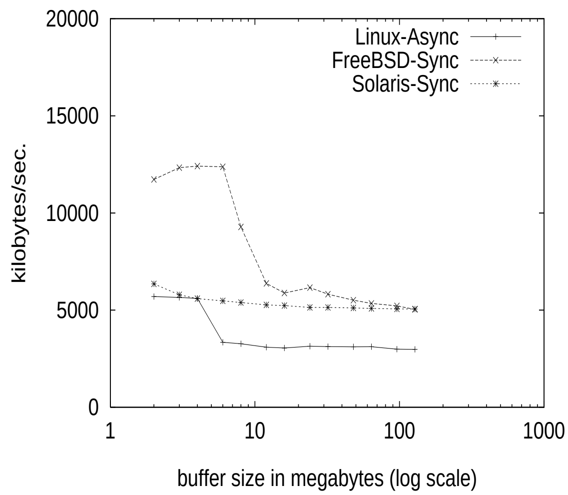

FIGURE 10. Bonnie Write: This figure shows file system sequential write bandwidth as a function of file size. The results are averaged over twenty runs.

accesses negate the benefits of an LRU file cache, and Solaris compensates for this better than the other systems. Linux has the worst read bandwidth for files larger than the buffer cache.

The effects of FreeBSD's efficient file cache and Linux's poor large file performance are also apparent in Figure 10. FreeBSD writes files of size less than eight megabytes 50% faster than Solaris or Linux. Linux maintains less than half the write bandwidth of FreeBSD or Solaris for almost all file sizes.

In contrast, Linux and Solaris can perform approximately 50% more random seeks and I/O operations per second than FreeBSD for files inside the file

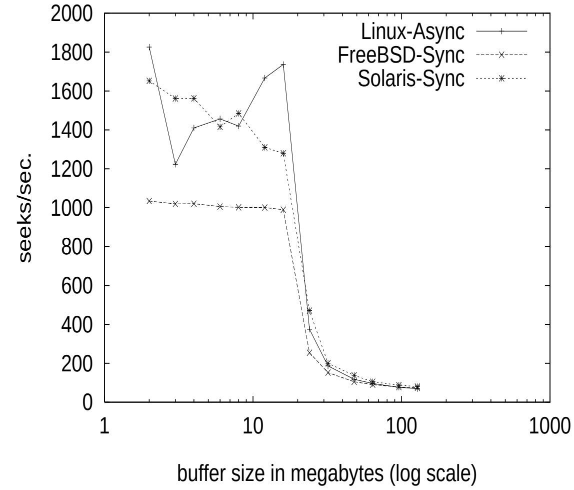

FIGURE 11. Bonnie Seek: This figure shows the number of random seeks per second as a function of file size. The results are averaged over twenty runs.

cache, as shown in Figure 11. The `bonnie` seek benchmark seeks to a random block in a file, reads the 8-kilobyte block and then writes it out. All three systems converge to 14ms for random seeks to blocks on disk.

7.2 Small-file and Metadata Benchmarks

To benchmark the ability of these operating systems to deal with many small files, we used `crtdel` from the Ousterhout microbenchmarks. `Crtdel` opens a file, writes some data to it, closes it, opens it again, reads data from it, and deletes it. It mimics the use of a temporary file by a compiler. It stresses the updating of file system metadata such as the inode, directory block and directory inode. We ran it using various file sizes to get a view of metadata overhead versus file data overhead (Figure 12).

Given that we measured the average non-cached seek time of these systems to be 14ms (Figure 11), Linux clearly is not accessing the disk during this benchmark. This is because the Linux file system, ext2fs, uses an asynchronous metadata update policy, unlike the FreeBSD and Solaris file systems. While this gives Linux a performance advantage, it could result in losing more data after a system crash. Some of the synchronous updates in the BSD-derived file systems are intended to help preserve file system consistency in the event of such failures.

FreeBSD does worse on this benchmark than can be explained by its use of synchronous metadata writes. Since both the FreeBSD and Solaris 2.4 file systems are derived from the BSD FFS [McKusick 84], they both use synchronous metadata writes. However,

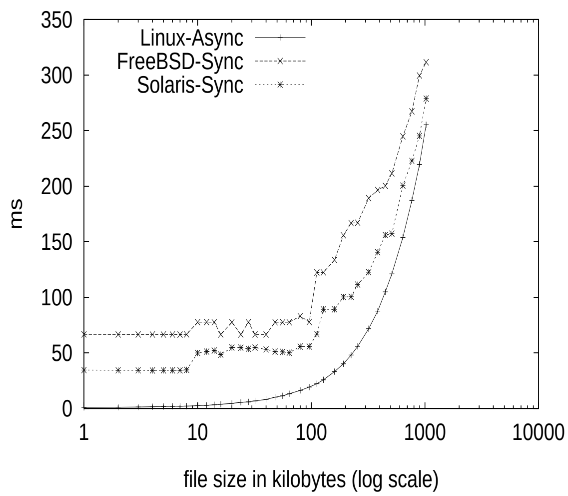

FIGURE 12. File Create/Delete: This figure shows the number of milliseconds to create and delete files as a function of file size. The results are averaged over twenty runs.

Solaris executes `crtdel` in only 34ms (Figure 12), compared to FreeBSD's almost 66ms. The magnitude of FreeBSD's overhead compared to Solaris suggests that it accesses the disk more than is necessary or seeks further. Furthermore, as the amount of data written increases from one kilobyte to one megabyte, the difference between the Solaris `crtdel` time and the FreeBSD `crtdel` time remains almost constant at about 32ms.

We also tested the FreeBSD file system using its optional asynchronous update policy. However, this option appears not to be implemented yet in version 2.5R, since our results for the synchronous and asynchronous modes were identical within the range of experimental error.

8. Modified Andrew Benchmark (MAB)

So far, we have reported the results of microbenchmarks. Microbenchmarks measure particular aspects of a system, but they may not reflect overall system performance under any realistic workload.

As a step towards comparing the operating systems under a typical software engineering workload, we used the Modified Andrew Benchmark (MAB). It consists of five parts: directory creation, file copying, directory stats, file reading, and compilation. To achieve portability and to eliminate the differences in the compilation speed of different compilers, the original MAB includes the source for an early version of `gcc`. This early version of `gcc` is used to compile for the SPUR architecture during the compilation phase. The code generated during the benchmark is never executed, so the choice of architecture does not matter.

We found we had to make further modifications to MAB, so our results are no longer directly comparable to previously-reported MAB results. The problem with the original MAB is that its version of `ranlib` relies on the system's `ar`, and binary file formats have changed enough that this scheme no longer works. Furthermore, the version of `gcc` in the original MAB is not portable to Linux or System V OSs (such as Solaris). To maintain the spirit of the original MAB, we modified MAB to use a recent version of `gcc` and included a compatible version of GNU's `binutils`, which includes portable versions of `ar`, `ld` and `ranlib`. We configured `gcc` and the `binutils` to generate code for the x86 architecture under Linux since the SPUR architecture is no longer supported as a compilation target.

In this section we report MAB results for a local file system. We report MAB results for accessing remote file systems over NFS in Section 10.

8.1 Local File System

The results of running MAB on a local disk are summarized in Table 3. Linux's first place finish is not surprising, given its performance on the file and disk micro-benchmarks. Linux's asynchronous file metadata updates and its good read performance for the small files (<1 megabyte) used in MAB indicate that it should do well on MAB.

OS	Time (seconds)	Std Dev	Norm.
Linux	43.12	4.10%	1.00
FreeBSD	47.45	1.02%	0.91
Solaris 2.4	54.31	1.93%	0.80

TABLE 3. MAB Local: This table shows the total time to execute MAB on the local file system as averaged over twenty runs.

What is more surprising is FreeBSD's good performance on MAB, given its poor performance in manipulating file meta-data and in reading small files. FreeBSD is competitive with Solaris in each of the benchmark phases, except `stat`ing directories, where it exceeds even Linux's performance. FreeBSD keeps a separate attribute cache for the directory information, which is filled in the first phase (directory creation) and is thus accessed in the third phase (directory stats). Linux does not have such a separate attribute cache, so its attribute information is knocked out of the file data cache during the second (file copying) phase.

9. Network Benchmarks

Faster network technology such as 100 Megabit/second Ethernet is becoming more affordable, while CPUs are becoming memory speed bound. As a result, the limiting factor for network performance is the efficiency of the network protocol implementation. We discovered that none of the x86 systems can fully utilize a 100 Megabit/second Ethernet link, with Linux being two to three times slower than FreeBSD and Solaris.

In most of our network benchmarks we used the loopback interface rather than an actual Ethernet interface. Although this ignores the effect of collisions and other real world effects, we wanted to measure the best possible performance in order to predict these operating systems' performance on a future network.

To isolate possible contributors and detriments to network performance, we tested network performance using three protocols: pipes, UDP, and TCP.

9.1 Pipes

Although pipes are not a network protocol, they require much of the same functionality as a network protocol, such as system calls, context switches and data copying. We measured pipe bandwidth as an upper bound on what network protocols could achieve if there were no other overhead. The pipe benchmark, `bw_pipe`, comes from Larry McVoy's `lmbench` benchmark package. It forks off a child and transfers 50 megabytes in 64-kilobyte chunks between itself and the child.

OS	Bandwidth (megabits/second)	Std Dev	Norm.
Linux	119.36	1.60%	1.00
FreeBSD	98.03	2.79%	0.82
Solaris 2.4	65.38	1.56%	0.55

TABLE 4. Pipe Bandwidth: This table shows the bandwidth of a pipe as averaged over twenty runs.

From Table 4, we see that Linux and FreeBSD could theoretically keep up with a 100-Megabit/second Ethernet, if the TCP/IP protocols added no additional overhead. Solaris, however, could not keep up. Solaris' slower system calls and context switches do not explain this poor performance. The extra overhead

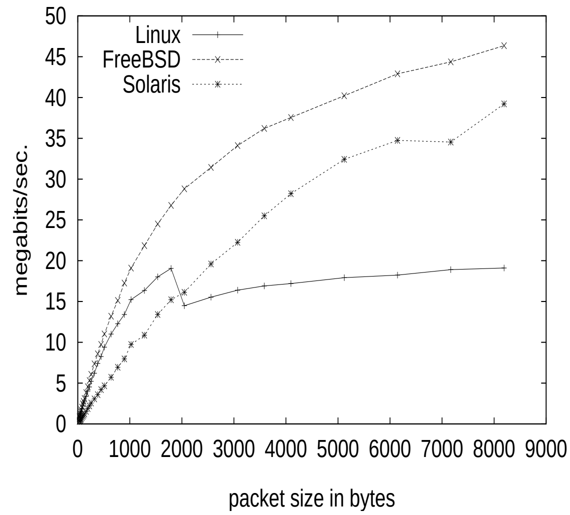

FIGURE 13. UDP: This figure shows UDP bandwidth as a function of packet size when averaged over twenty benchmark runs.

for Solaris pipes is largely due to their implementation on top of System V streams [Kottapurath 95].

9.2 UDP

The UDP protocol is a slightly higher-level protocol than pipes, in that UDP forms packets but does not use time-outs, sequence numbers, and retransmission as in TCP. In order to test UDP bandwidth, we ran `ttcp` using a variety of packet sizes, transferring 4 megabytes every iteration. When run as the sender, `ttcp` reads data from `stdin`, breaks it up into packets, and sends the packets to the receiver. When run as the receiver, `ttcp` reads packets and writes the data to `stdout`. We redirected the output to `/dev/null`.

From Figure 13, we see that FreeBSD achieves a bandwidth of almost 50 megabits/second, meaning that its UDP runs at only 50% of the bandwidth of pipes. Solaris is worse. It achieves a peak bandwidth of 32 megabits/second; just as with FreeBSD, this is 50% of the pipe bandwidth. Linux has the most surprising result. Although it has the best pipe bandwidth, it has the worst UDP performance. Its UDP performance of 16 Megabits/second is only 14% of its pipe bandwidth. Its UDP implementation has a high amount of overhead due to unnecessary copies and inefficient buffer allocation.

9.3 TCP

TCP is one of the most widely used protocols today, forming the basis for many reliable protocols, such as ftp. In order to benchmark TCP, we use `bw_tcp`, which comes from Larry McVoy's `lmbench` benchmark package. `Bw_tcp` transfers 3 megabytes from one process to another during each iteration using a 48K buffer.

OS	Bandwidth (megabits/ second)	Std Dev	Norm.
FreeBSD	65.95	2.36%	1.00
Solaris 2.4	60.11	16.34%	0.91
Linux	25.03	5.45%	0.38

TABLE 5. TCP Bandwidth: This table shows the bandwidth of a TCP connection.

As shown in Table 5, Solaris's TCP performance is not hindered by its poor UDP performance. On the other hand, Linux's TCP implementation is just as slow as its UDP implementation. Our investigations indicate that version 1.2.8 of Linux has a TCP window of only one packet. This severely limits its TCP bandwidth, as our results show.

10. MAB across NFS

To measure network file system performance for the three systems, we ran MAB over NFS, using the three systems as clients. We ran these tests using a Linux 1.2.8 file server and a SunOS 4.1.4 file server. We did not test FreeBSD or Solaris as servers, since we do not have the extra equipment available.

OS	Time (seconds)	Std Dev	Norm.
FreeBSD	53.24	0.87%	1.00
Linux	57.73	2.20%	0.92
Solaris 2.4	58.38	1.36%	0.91

TABLE 6. MAB NFS with Linux Server: This table shows the total time to execute MAB across NFS to a Linux server.

Using a Linux server, the FreeBSD client was the top performer due to its good networking performance. Linux comes in second place with Solaris coming in third.

Overall, the benchmark ran more slowly when accessing the SunOS server rather than the Linux server. The SunOS file server uses a synchronous update policy, as required by the NFS specifications. The Linux file server continues using its asynchronous update policy, and we hypothesize that this explains the difference in performance.

With the SunOS file server, we see somewhat different relative results between the three clients. FreeBSD's good networking performance again serves it well when connected to a SunOS NFS server. Solaris performs relatively poorly when using the SunOS server instead of the Linux server. Linux's networking code is apparently tuned to work with other Linux hosts and performs miserably when connected to other types of servers.

OS	Time (seconds)	Std Dev	Norm.
FreeBSD	67.60	1.41%	1.00
Solaris 2.4	87.94	3.17%	0.77
Linux	115.06	1.54%	0.59

TABLE 7. MAB NFS w/SunOS Server: This table shows the total time to execute MAB across NFS to a SunOS Server.

11. Other Comments

In testing the performance of these systems, we encountered other differences that may be of interest to those choosing which system to run. Although some of these differences may disappear in later releases, some are a consequence of the policies of the system developers or vendors and therefore may not change in future releases. These differences include installation difficulties, porting differences, and system bugs found while running the benchmarks. All of these areas help indicate the level of support one can expect when using the systems.

In general, the availability of free system source code combined with a large user community seems to have a positive effect on these problems. If the user community contributes drivers and other system software, then that system will work sooner on a wider range of hardware than is possible for a system with only a few developers. Even if no one has contributed a desired feature yet, we can implement and even distribute it ourselves at a minimum of cost. Almost by definition, systems research requires new hardware and software that has not yet made it to the commercial sector. Additionally, a large user community with access to source code will provide support for a system outside of a vendor's or a developer's support, increasing the probability that bugs will be found and fixed quickly and questions answered.

Our installation experiences with the three systems were very different, with Linux being the easiest and Solaris being the most difficult.

Some of the good installation features:

- Installation across the Internet (Linux, FreeBSD)
- WWW installation documentation (Linux, FreeBSD)

Among the problems we encountered:

- Didn't support the (very common) Panasonic/Creative Labs CD-ROM drive (FreeBSD, Solaris)
- Crashed during installation due to a driver incompatibility (FreeBSD, Solaris)
- Obliterated existing boot loader and disk partitions (Solaris)
- Inaccessible or missing system administration documentation (Solaris)

Our experiences porting the benchmarks to the three systems were somewhat more pleasant, with Linux again being the easiest system and Solaris the most difficult. In general, Solaris was the most difficult because there is no Internet repository of Solaris binaries, and there isn't yet a large enough Solaris x86 user community to provide the level of support found for the other systems.

Some of the good porting features:

- BSD and System V compatibility (Linux)
- Automatic installation of commonly used free software like `gcc`, `emacs`, and `tcsh` (Linux, FreeBSD)
- Internet repository of pre-compiled binaries (Linux, FreeBSD)

Some of the porting difficulties:

- No installed compiler (Solaris)
- Only an old and buggy pre-compiled `gcc` available on the Internet (Solaris)

All of the systems had problems running the benchmarks, with the most irritating problem being that the Linux 1.2.8 NFS server requires that clients connect on a privileged port. FreeBSD 2.0.5 clients do not do this by default.

12. Conclusions

No one system dominates our benchmarks. Linux does well on system calls, context switching, and pipe bandwidth. Its performance on small-file workloads with intensive metadata manipulation is an order of magnitude faster than the other systems. Linux also does well when communicating with a Linux NFS server. However, Linux has poor overall networking performance and poor NFS performance when connected to a SunOS NFS server.

FreeBSD has better networking and NFS performance than the other systems. It performs well on large files but not on small files. It does well on the Modified Andrew Benchmark both remotely and locally.

Solaris has poor system call, context switch and pipe performance. It reads large files efficiently but does poorly when the Modified Andrew Benchmark is run locally.

An inherent disadvantage of our "black box" benchmarking approach is that it cannot conclusively explain all of the performance differences in these systems. In many cases, it merely exposes the differences. In addition, using microbenchmarks isolates the areas of both good and bad performance, but microbenchmarks cannot predict overall application performance. Despite the differences on the microbenchmarks, the systems' overall performance on the MAB workload is much closer.

13. Future Work

Benchmarking operating systems that are under active development is always a work in progress. As we write this paper, new versions of all of these systems are about to be released with several changes in their performance. The latest development version of the Linux kernel (1.3.40) is a good example. It has very fast context switching (10 microseconds for two active processes with very little slowdown as the number of active processes increases). Its NFS performance has also improved. The next release version of FreeBSD (2.1) will offer ordered asynchronous metadata updates to improve small-file performance while helping maintain file system consistency during a crash. The next version of Solaris (2.5) will have faster context switching and better performance in general.

Architectural support for counting operating system events such as TLB misses [Chen 95] can reveal more about the workings of an operating system than using timers alone. We plan to apply some of those techniques to the systems that interest us.

14. Benchmark Source Code Availability

Our benchmarking package is available at http://plastique.stanford.edu/bench.html.

15. Acknowledgments

We thank Larry McVoy for his many helpful comments. We thank Jeff Bonwick, Sherif Kottapurath, Dean Long, and Behfar Razavi for their answers to questions about Solaris. We thank Stuart Cheshire, Elliot Poger, Mendel Rosenblum, Jonathan Stone, Diane Tang, and especially Darrell Long for their comments on the paper. We thank the authors of all the benchmarks we used. This work was supported by a grant from the Reid and Polly Anderson Faculty Scholar Fund at Stanford University.

16. References

[Anderson 93] Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, MindShare Press, 1993.

[Anderson 91] Thomas Anderson, Henry Levy, Brian Bershad, and Edward Lazowska, "The Interaction of Architecture and Operating System Design." *ASPLOS-IV*, April 1991.

[Bonwick 95] Jeff Bonwick, personal communication, November 1995.

[Bray 90] Tim Bray, Bonnie source code, 1990.

[Chen 93] J. Bradley Chen and Brian N. Bershad, "The Impact of Operating System Structure on Memory System Performance," *Proceedings of the Fourteenth International Symposium on Operating Systems Principles,* pp. 120-133, December 1993.

[Chen 95] J. Bradley Chen, Yasuhiro Endo, Kee Chan, David Mazieres, Antonio Dias, Margo Seltzer, and Michael D. Smith, "The Measured Performance of Personal Computer Operating System." To appear in the *Proceedings of the Fifteenth International Symposium on Operating Systems Principles*, December 1995.

[FreeBSD 95] Various Authors, FreeBSD Home Page, `http://www.freebsd.org/`, 1995.

[Howard 88] J. Howard, et al. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.

[Intel 94] Intel Corporation, "The Pentium Family User's Manual, Volume 3: Architecture and Programming Manual." Intel Literature Sales, P.O. Box 7641, Mt. Prospect, IL 60056-7641, 1994.

[Kottapurath 95] Sherif Kottapurath, personal communication, November 1995.

[LDP 95] Various Authors, Linux Documentation Project, `http://sunsite.unc.edu/mdw/welcome.html`, 1995.

[McKusick 84] Marshall K. McKusick, "A Fast File System for Unix," *ACM Transactions on Computer Systems* 2(3) pp. 181-197, 1984.

[McVoy 95] Larry McVoy and Carl Staelin, "lmbench: Portable tools for performance analysis," To appear in *Proceedings for the 1996 Usenix Technical Conference*, January 1996.

[Ousterhout 90] John K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast As Hardware?" *Proceedings of the 1990 Summer Usenix Conference*, pp. 247-256, June 1990.

[Rashid 88] Richard F. Rashid, et al., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers,* Vol. 37 No. 8, pp. 896-908, August 1988.

[Tcl 90] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, Reading, Massachusetts, 1994.

[Welsh 94] Matt Welsh, *The Linux Bible*, Yggdrasil Computing Incorporated, 1994

17. Biographical Information

Kevin Lai is a Master's student at Stanford University. He received his B.A. in Computer Science in 1992 from U.C. Berkeley. His interests include performance measurement, operating systems, and operating system support for mobile computing.

Mary Baker is an assistant professor in the Departments of Computer Science and Electrical Engineering at Stanford University. Her interests include operating systems, distributed systems, and software fault tolerance. She received her Ph.D. in computer science in 1994 from U.C. Berkeley.

The authors' email addresses are `{laik, mgbaker}@cs.stanford.edu`.