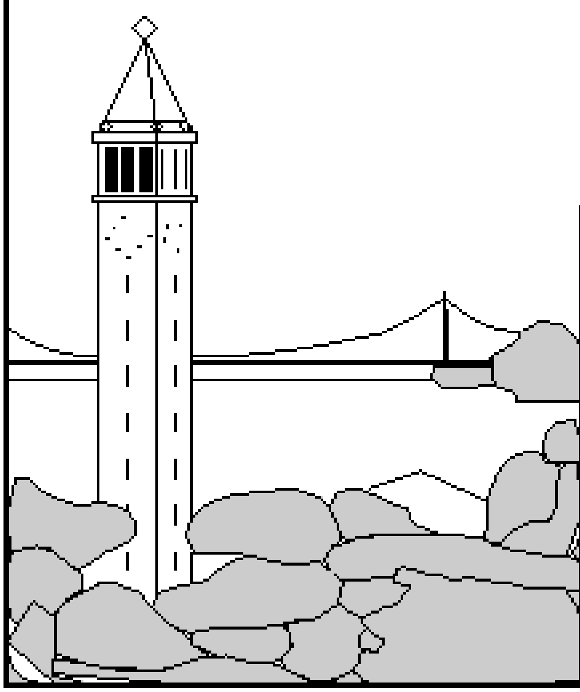
A Comparison of PC Operating Systems for Storage Support

Nisha Talagala, Satoshi Asami, David Patterson



Report No. UCB/CSD-98-1018

Jan 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Comparison of PC Operating Systems for Storage Support

Nisha Talagala, Satoshi Asami, David Patterson
{nisha, asami, pattrsn} @cs.berkeley.edu
University of California at Berkeley
January 1997

Abstract

In the past 10 years, the cost/megabyte of magnetic disks has been decreasing by almost a factor of two. This decrease makes large scale disk-based storage systems attractive. However, such systems usually come in the form of RAID arrays that have a much higher cost/megabyte than the underlying disks. To maintain the low cost ideal, COTS (commercial, off the shelf) components must be used. A large storage system, built from off the shelf hardware, could have disks hosted by PCs connected through a high performance LAN. The high bandwidth PCI bus and large number of expansion slots make PCs a good building block for such a system. However, most PCs (and PC operating systems) are used in desktop environments, typically with only 1 or 2 disks.

In this report, we test how well PC operating system support large amounts of storage. We compare three operating systems, Windows NT, FreeBSD and Solaris x86. A set of experiments are conducted on each OS to test support for (i) large numbers of disks and disk controllers, (ii) PCI expansion boxes, and (iii) Shared SCSI buses. The results show that all of the operating systems compared had some subset of the required features. We were also able to add the missing features to FreeBSD, and show that it is feasible to use PCs to host large amounts of storage. In addition, we present some measurements of multiple disk I/O on each operating system.

This research is funded by the DARPA Roboline Grant # N00600-93-K-2481, donations of machines from Intel, and the California State Micro Program.

1.0 Introduction

In the past 10 years, the cost performance gap between secondary and tertiary storage has been widening. The cost per megabyte of disk drives has been falling at a factor of 2 per year, compared to 1.5 per year for tape drives and libraries. Disk areal densities have been increasing at 60% per year [1]. These trends change the possibilities in large scale storage systems. If they continue, large storage systems composed of disks will have significant cost/performance advantages over tape libraries of similar capacity. The currently available solution for large disk storage is RAID arrays. These disk arrays have drawbacks in terms of cost/performance, availability, and scalability. Since they use custom hardware, the cost per megabyte of disk arrays increases with system capacity, unlike raw disks and tape systems. Also, a disk array needs to be connected to a host computer. This connection becomes a bottleneck for both performance and availability. Finally, scalability is limited by the number of disks that can be supported by the infrastructure. When applications reach the capacity limit of their disk array, another array must be added. Adding independent disk arrays lowers the reliability of the total system and complicates storage management.

A storage systems built with commercial, off the shelf, hardware could avoid some of the disadvantages of custom RAID systems. The low cost and good performance of PCs make them a good candidate for hosting disks in a COTS based system. However, most PCs are used in office environments, typically running Windows and MS-DOS. Not much is known about the ability of PC operating systems to support large amounts of storage. Also, there are many choices for PC operating systems, ranging from Microsoft Windows and Windows NT to UNIX compatible operating systems such as FreeBSD, Solaris and Linux.

In this work, we compare how well PC operating systems support large amounts of storage. We compared the three operating systems Windows NT, FreeBSD and Solaris x86. We were unable to study Linux as it did not support our SCSI controller. We installed each operating system on our testbed ran a series of experiments to see whether the operating system could support configurations of disks that could exist in a large storage system. In particular, we tested each operating system for support of large numbers of disks and SCSI controllers, support for PCI-PCI expansion bridges, and support of shared SCSI buses. Finally we also measured multiple disk I/O performance for each OS.

Our experiments revealed that each operating system had some subset of the required features, but no OS had all of them. Of the three operating systems we tested, FreeBSD was the only one with freely available source code. Because of this, we adopted a black box approach to the initial experiments. Later, we made several modifications to FreeBSD to enhance its support for the tested features. Our performance experiments also revealed that the I/O performance of each OS was far below the capabilities of the disk and SCSI subsystem. In FreeBSD, we discovered that this low performance was partly due to slow memory to memory copies. Other studies have also made this observation [2]. By replacing the memory copy routines with higher performance handcoded routines, we were able to increase FreeBSD's I/O performance by about 22%.

The rest of the paper is organized as follows. Section 2 describes why PCs are a good building block for storage systems. Section 3 describes the operating systems that we tested. Section 4 describes some related work in comparing PC operating systems. Section 5 describes our testbed, experiments, and results. Section 6 describes some performance measurements of I/O on the three operating systems. Section 7 describes our experience in installing and using the three alternatives, and section 8 provides a summary and conclusions. Finally, the appendix in section 9 outlines the enhanced memory copy routines.

2.0 PCs as Building Blocks

PCs are a natural choice for hosting disks. The main system bus, PCI, has a peak bandwidth of 132 MB/s. Typical PCs have 4 PCI expansion slots on the main bus. If one slot is used for a network connection, 3 slots are available for disk controllers. In this manner, a large number of disks can be connected to a single PC. If each remaining slot holds a single channel wide SCSI controller, the PC can host up to 45 disks. If dual channel SCSI controllers are used, the number of disks doubles. Other competing machines, like the SPARCStation series from Sun Microsystems, usually have only 1-2 expansion slots.

It is also possible to extend the PCI bus using PCI-PCI bridges. The design of the PCI bus limits the number of slots to 7. However, if one of these slots is connected to a PCI-PCI bridge, six more slots are available. Such PCI expansion boxes are commercially available [3]. In this fashion, the number of disks connected to a single PC can be increased dramatically and many different configurations are possible.

PCs are also attractive from a cost perspective. The high volumes and fierce competition in the PC industry makes these machines cheaper than most UNIX based platforms.

3.0 Methodology

Since we were looking for PC operating systems that could function as storage nodes, we limited our choices to operating systems that were used in server environments. In particular, we did not include Windows in the study. We chose Windows NT version 3.5.1, FreeBSD version 2.1.5 and Solaris x86 version 2.4. Linux also meets these requirements. We had intended to include Linux, but there was no driver available for the SCSI controller that we were using. For each operating system, we chose the latest version available in mid 1996.

Solaris is a System V based version of UNIX from Sun Microsystems. Solaris is available for both SPARC and x86 archtectures. In early 1996, Solaris was available on CDROM for $99, source code not included. The Solaris kernel is fully preemptive multithreaded and has support for multiprocessor systems. More detailed information on Solaris can be found at [4].

FreeBSD is a freely available version of UNIX. It is derived from the BSD 4.4 Lite release from UC Berkeley. Its source code is freely available and a large number of people have contributed to it. The FreeBSD file system is based on Berkeley FFS. More information on FreeBSD can be found at [5 6].

Windows NT supports applications written for Windows, and also offers additional features available in UNIX. Two filesystems are available under Windows NT, the FAT file system used in DOS and NTFS. We used NTFS in our experiments. More details on Windows NT are available in [6,7].

4.0 Related Work

Two earlier studies have compared PC based operating systems. The first study, done by Chen et al., compared Windows for Workgroups, Windows NT and NetBSD [9]. The authors used hardware counters on the Pentium processor to gather data on a variety of processor events, like instruction counts and on chip cache misses. They also used microbenchmarks to measure performance of operating systems functions (syscalls, read/write calls, etc.) and application workloads (web server, Ghostscript, etc.) to measure end-to-end performance. The second study, by Lai et al. [2], compared Solaris x86, FreeBSD and Linux. These authors also used a mixture of microbenchmarks and application level workloads to evaluate each operating system.

These two studies had different goals. The first study compared three operating systems that are quite different in system functionality and user requirements. Windows for Workgroups is a version of Windows with integrated networking support. It does not support more advanced features like protected address spaces and preemptive multitasking, features that are available in Windows NT and UNIX. While the first study showed the choices available to PC users with a range of requirements, the second study compared UNIX operating system alternatives for the PC.

Since we are interested in using PCs to host disks and serve data, the information about Windows for Workgroups was not relevant to our work. The performance studies of the other five OSes are relevant. For instance Chen et al. [9] observed that Windows NT has much higher latency than NetBSD for disk I/O. For both the NTFS and FAT file systems, NT performance lags behind NetBSD by almost a factor of two. We do not attempt to summarize all the performance data in these two studies here. Some results are mentioned in Section 6, where we discuss the I/O performance of Solaris, FreeBSD and

Windows NT. However, both studies showed that each operating system is superior to the others in some way, but no operating system is clearly the winner. The choice depends on the intended application. Lee et al. [2] points out that other factors like cost, ease of installation and support may help a user to choose an operating system.

There are two main differences between these two studies and our PC-OS comparison. These two studies compared the strengths of each OS for a range of microbenchmarks. Our study focuses only the abilities of the operating systems to function as part of a storage node in a larger system. The two studies also do not test the hidden OS limits for hardware support. This in an important part of our study, as we are using PCs in a way that the they are not normally used.

5.0 TestBed and Experiments

In this section, we describe our testbed, the experiments we designed to test support for large storage configurations, and the results of these experiments.

5.1 Testbed

We installed each operating system, in turn, on a 2GB Quantum disk drive. All the experiments were done on a Pentium 133 machine with this 2GB internal disk. This machine had PCI 4 expansion slots. In the experiments described in the following sections, we changed the machine configuration by attaching SCSI controllers, disks and PCI expansion boxes. We used Adaptec 2940W and 3940W SCSI controllers, the latest available at the time. The 2940W model is a single channel Fast-Wide SCSI controller, while the 3940W model is the dual channel version. Our disks were 7200 RPM, 4GB, Seagate Barracuda drives. The PCI expansion boxes we used were manufactured by Bit3 corporation. For some experiments, we also used an ANCOT SCSI Bus Analyzer to monitor activity on the SCSI bus.

The PCI expansion boxes we are using provide four PCI slots on a separate bus, which can be connected to the main bus using a bridge. Since one PCI bus can only offer 4 new PCI slots, and one of the slots of the master bus is taken by the expansion card, typical PCI expansion boxes have two PCI buses in the expansion unit. Also, because of electrical considerations, the cable to connect the host and the expansion unit is usually a separate PCI bus.

5.2 Experiments

Next, we describe the experiments we designed to test how well each operating system supported different disk/controller configurations. Since we did not have source code for two of the three operating systems, we used a black box approach in all the experiments described in this section. The features that we tested support of were:

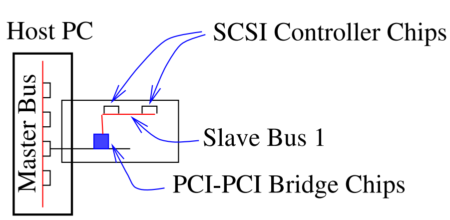


Figure 1(a)

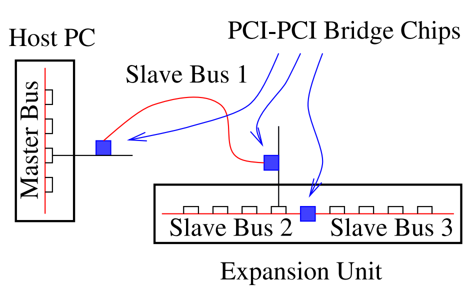


Figure 1(b)

Figures 1(a) and 1(b) show two PCI-PCI bridge configurations. Figure 1(a) shows a PCI-PCI bridge as used in a twin channel SCSI controller. Both SCSI controllers are on a separate PCI bus, connected to the main PCI bus through a bridge. Figure 1(b) shows the layout of a PCI expansion box. The cable connecting the expansion box to the machine is the first slave PCI bus. Three free slots are available in each of the second and third slave buses.

(i) Support for large numbers of disks and disk controllers:
If a PC were to function as a storage node in a large storage system, the operating system should be able to support and access a large number of disks. In this experiment, we connect up to 40 disks to our test machine. We then access as many as possible through the operating system. We were limited to testing upto 40 disks since we had only 40 disks to use for this experiment. However, the performance results in the next section show that even if the OS supports more disks, connecting more than 40 disks is probably not advisable in general for performance reasons.

(ii) Support for PCI-PCI bridges:
PCI-PCI bridges can be used to expand beyond a single PCI bus. Figure 1 shows two configurations using PCI-PCI bridges. Figure 1(a) shows a single bridge integrated into an expansion card. A dual channel SCSI controller uses this scheme. Both SCSI buses are on the new PCI bus, connected to the main bus through the bridge. Figure 1(b) shows the slightly more complicated configuration used in a PCI-PCI expansion box. In this case, the cable that connects the expansion box to the Pentium machine is itself a PCI bus. The slots on the left hand side on the expansion box are separated from the main PCI bus by two levels of bridging. The slots on the right side of the expansion box are one bridge level deeper.

We used the dual channel SCSI controllers to test support for a single level of bridging. We connected disks to each channel of the dual channel controller and attempted to access the disks through each operating system. For multiple levels, we used the PCI expansion boxes. In this case, we connected disks to single channel controller cards in the left hand and right hand slots of the expansion box. We then attempted to access these disks through each OS. Note: In this case, we used only single channel SCSI controllers to attach the disks, since the dual channel controllers would have added another level of bridging.

(iii) Support for shared SCSI buses
One obvious problem with such an approach of connecting many disks to a single machine is availability. A single PC failure will make a large number of disks unreachable from the rest of the network. One solution to this problem is to connect disks to two hosts at the same time. However, even though dual ported disks exist, they are expensive and not widely available. Since our goal is to use commodity hardware when ever possible, we experimented with shared SCSI buses. Figure 2 shows the structure of a shared SCSI bus. The two controllers are on either sides of the bus, with the disks in between. The SCSI string is terminated at both ends using external terminators. Since passive termination is used, it is theoretically possible to power down one of the host PCs and disconnect it form the bus without upsetting the other host PC. Although SCSI buses and commands work with multiple initiators (controllers), some operating system support is required to make shared SCSI buses work.

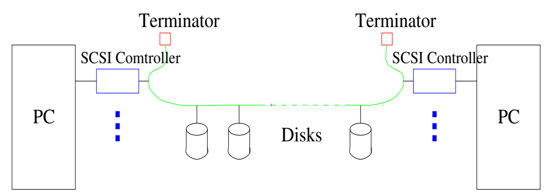


Figure 2: A shared SCSI bus. The SCSI controllers are connected at opposite ends of the bus, with the disks in between. Passive termination is used (the termination available in the SCSI controllers is disabled).

First, the operating system must support variable SCSI ID for SCSI controllers. Although the SCSI-2 specification allows initiators to have any SCSI ID between 0 and 15, most SCSI controllers are assigned an ID of 7. The reason is that SCSI ID 7 has the highest priority (priorities in decreasing order are 7-0, 8-15).

Second, the SCSI driver of one host PC should be capable of handling SCSI commands issued by the SCSI controller on the other host PC. For example, if one machine reboots, several BUS_RESET commands will be issued of the SCSI bus. The effect of the reset is that all activity on the SCSI bus is aborted and both controllers must renegotiate transfer parameters with the disks. The operating system must support this renegotiation.

If disks are not simultaneously accessed by both machines,

Feature	Experiments	Operating System		
		NT/NTFS	FreeBSD	Solaris x86
Large number of disks	Upto 40 disks and 3 dual channel SCSI controllers accessed through each operating system	Up to 31 allowed by Disk Administrator	Up to 32 allowed	Up to 6 allowed per SCSI bus
PCI-PCI Bridge Support	One level of bridging - Dual channel SCSI controller	Y	Y	Y
	Multiple bridge levels - PCI-PCI expansion boxes	Y	N	N
Shared SCSI Buses	Variable SCSI Controller ID	N	Y	Y
	Renegotiate on BUS-RESET	Y	N	Y

Table 1: OS comparison. This table summarizes the experiments and their results. Windows NT, FreeBSD and Solaris x86 are tested for support of many disks, shared SCSI buses and PCI-PCI bridges. A 'Y' indicates that the OS passed our test, and an 'N' indicates that it did not.As the table shows, each OS supported some subset of the features. None of the three OSes got Y's in all experiments.

the features mentioned above are the minimum required to support shared SCSI. We tested the three OSes for support of variable SCSI IDs for controllers and for renegotiation on BUS-RESETs. We changed the SCSI controller's ID to something other than 7, to see if each operating system could recognize the new ID and use the controller. To test the response to BUS-RESETs, we set up a shared SCSI string between two machines. Then, we rebooted one machine and attempted to access the disks through the other machine. By inserting a SCSI Bus Analyser into the string, we were able to verify that the OS SCSI driver responded correctly to the BUS-RESET event.

5.3 Results

Table 1 summarizes the experiments and results. We got different results for all three operating systems when we tried to connect 40 disks at the same time. In Windows NT, we were not able to access more than 31 disks through the Disk Administrator GUI. (this utility is used to format disks and set up stripe groups). In FreeBSD, we were able to access 32 disks through the OS. In Solaris x86, we were able to access all 40 disks through the OS, as long as only 7 disks were attached to a single SCSI bus. The experiment uncovered a bug that would cause the Solaris operating system to crash if a disk with SCSI ID greater than 7 was connected. Therefore, to get to 40 disks in Solaris, we had to use 6 SCSI controllers, or 2 single channel and 2 dual channel controllers.

Next, we tested PCI-PCI bridge support. All three operating systems were able to access disks through the dual-channel controllers, indicating that one level of bridging was supported. Windows NT was able to access devices connected to PCI expansion boxes. FreeBSD was not able to recognize devices below more than one bridge level and neither was Solaris x86.

Next, we tested for support of variable controller SCSI IDs. We were not able to change the controller SCSI ID in Windows NT, it appeared to be hardwired to 7. In FreeBSD we were able to change the controller ID to any value in the 0-15 range. In Solaris x86, we were also able to change the controller SCSI ID, as long as we remained within the range 0-7.

Both Windows NT and Solaris reacted correctly when a BUS-RESET appeared on a SCSI string. The SCSI drivers immediately renegotiated transfer parameters with all disks. The SCSI driver in FreeBSD did not react in this manner, so we were not able to access the disks in the string after a BUS-RESET event.

These results show that none of the operating systems we studied was initially able to meet all the requirements. Since we had source code for FreeBSD, we were able to place fixes for FreeBSD in all cases where the operating system failed our experiments. For instance, we found that FreeBSD was only able to recognize 32 disks because of the size of a data structure that held the disk minor number (this field was set at 5 bits). By expanding the size of this field, we were able to access a lot more disks. The second problem was that FreeBSD was not able to recognize devices that were two PCI bridge levels away from the main PCI bus. We fixed this problem by adding code to do a depth first search down all PCI-PCI bridges that were encountered in the initial device search at boot time. After this fix was in place, we tested the operating system with configurations of 5-6 PCI bridge levels. In all cases, we were able to access the disks. The third problem, the response to BUS-RESETs, was fixed by adding code to the SCSI driver to renegotiate transfer parameters with the disks after a BUS-RESET message was detected.

We found that with several minor source code changes, we were able to fix one operating system to meet all the requirements we tested. This suggests that if necessary, fixes may also be possible for the other operating systems.

6.0 Performance Measurements

In this section, we describe some performance measurements of the three operating systems. The two prior studies, [9] and [2], measured the performance of a variety of system services. However, we focus only on I/O performance. In particular, we measure each operating system's I/O performance with multiple disks. Chen et al.[9] and Lee et al. [2] only measured file system performance with a single disk. We did two types of multi-disk performance experiments. First we used the striping drivers that were available with each operating system. These drivers enabled us to create virtual disks by grouping disks into a stripe set. We also did multi-disk measurements by issuing parallel I/O requests to several disks. Finally, we studied the effect of improving memory copy bandwidth on I/O performance. We also present some raw disk performance measurements of dedicated and shared SCSI buses.

6.1 Single disk performance

Table 2 lists single disk bandwidth for each operating system. These measurements were taken through the file system. The workload was sequential reads and writes in 64KB blocks. By inspecting SCSI traces, we determined that Windows NT used tagged queuing, while Solaris x86 did not. Tagged queuing allows the SCSI driver to have multiple outstanding requests to a single disk. The disk can order the commands for maximum performance. FreeBSD could be configured with or without tagged queuing.

	NT/NTFS	FreeBSD	Solaris x86
Read BW (MB/s)	3.71	7.99	7.69
Write BW (MB/s)	2.95	6.49	3.21

Table 2: Sequential read and write performance for a single disk. These measurements were taken through the file system using 64KB requests. Both Windows NT and FreeBSD are using tagged queuing, while Solaris x86 is not.

Windows NT with NTFS got almost 4 MB/s on reads and 3 MB/s on reads. FreeBSD had a peak bandwidth of 8 MB/s on reads and 6.5MB/s on writes with tagged queuing. Without tagged queuing, FreeBSD still got almost 8MB/s on reads, but the write performance dipped to 3.6 MB/s. Solaris x86 had peak bandwidths of 7.7 MB/s on reads and 3.2 MB/s on writes. [2] reports similar single disk performance for Solaris x86. Their results for FreeBSD agree with our results for FreeBSD without tagged queuing.

6.2 Multiple disk performance

For our measurements of I/O on multiple disks, we used both striping and parallel requests. The first set of measurements were done using the striping software available with each operating system. For Windows NT, we used the striping software available through the Disk Administrator utility. On FreeBSD we used the CCD (concatenated disk) driver. On Solaris x86 we used the DiskSuite software that is packaged with the operating system. Although both the Windows NT and Solaris software provides RAID 1 and RAID 5, we only created RAID 0 (no redundancy) stripe sets for our experiments.

Figure 3 shows the performance of the striping software on the three operating systems. In each case, the measurements are done on a stripe set accessed through the file system. Figure 3 shows the read bandwidth as the number of disks is increased. This data is based on the maximum achieved by each operating system over a range of controller/disk configurations. We varied the number of SCSI adapters in the machine and the number of disks per SCSI string. For Windows NT, the read performance did not improve with multiple disks, and remained close to the performance of a single disk. The read bandwidth of Solaris x86 peaked at 12 MB/s. FreeBSD with the concatenated disk driver achieved 21MB/s.

A direct comparison between the three striped drivers is not possible, as their structure and functions are quite different. We measured their performance because this is software that the OS vendor intended to help uses manage large numbers of disks. To compare multi-disk performance on a more even platform, we measured the peak read bandwidth possible with parallel disk accesses. Windows NT performance did not improve. Even with parallel disk requests, the peak bandwidth was close to that of a single disk. The performance on FreeBSD remained similar to that under the CCD stripe set. In Solaris, running parallel reads to all disks gave us a peak bandwidth of 14 MB/s, slightly more than its result with the DiskSuite software. This suggests that the DiskSuite software may be the bottleneck in the results of Figure 3.

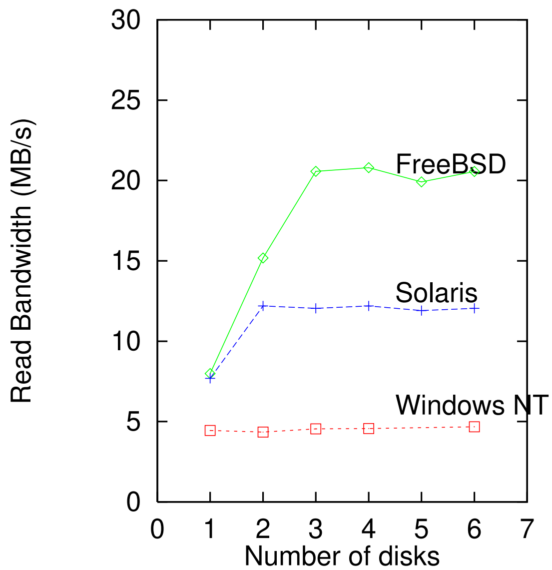
Although the multi-disk read performance of FreeBSD was well above that of the other two operating systems, it was still well below the capabilities of the disks and the SCSI bus. Since each disk is capable of delivering almost 8 MB/s on reads, 6 disks could deliver upto 48 MB/s. The configuration that delivered 21 MB/s had three SCSI controllers, so the performance is not SCSI limited either. Three Fast-Wide SCSI buses can have a peak bandwidth of 60 MB/s.

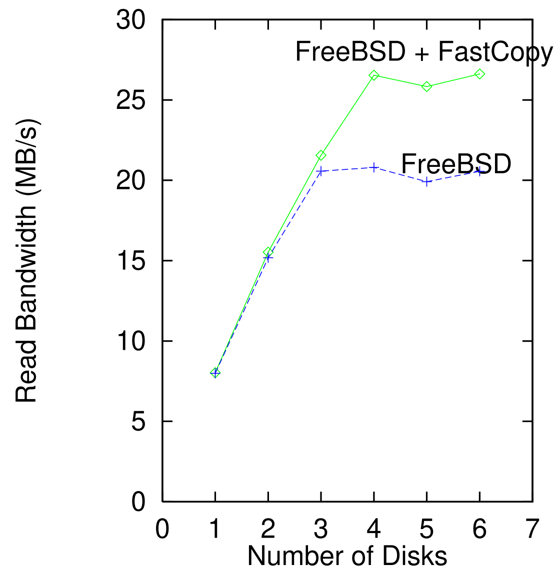
Lee et al. [2] observed that the memory write performance in both Solaris and FreeBSD was well below the capabilities of the hardware. We suspected that this may be limiting our read performance in FreeBSD to 21 MB/s. To test this hypothesis, we replaced the memory copy routines in FreeBSD with several hand-coded assembler routines. The Pentium has a write allocate cache, so the performance of memory write routines can be improved by prefetching cache lines. Using this prefetching technique, we implemented a set of custom memory copy routines. The copy routines we used are described in the Appendix. Our hand coded routines delivered copy bandwidths of 80 MB/s, compared to the 40MB/s possible without prefetching. We incorporated these prefetching routines into FreeBSD and repeated our I/O performance measurements. Figure 4 shows the read bandwidth on a striped array with and without the specialized copy routines. We were able to improve read performance on striped arrays by about 22%, to 27 MB/s.

6.3 Shared SCSI Bus Performance

All of the prior measurements were done using dedicated SCSI buses. Next we examined the performance implications of sharing a SCSI bus. We set up the machine/SCSI configuration shown in Figure 2. Next, we ran read benchmarks on disks on the shared bus, first from one machine and then simultaneously from both machines. These experiments used the raw disk interface. Figure 5 shows the read performance on one machine while the other machine is idle. Figure 6 shows the read performance on one machine while both machines are executing the same benchmark on the disks. Both figures show the results for FreeBSD; we also ran the experiment for Solaris x86, with similar results. The workload is read requests of varying sizes.

We can make two main observations from the data in Figures 5 and 6. First, for the smaller requests (8KB) both hosts get the same performance with shared SCSI as a sin-





Figures 3 and 4: Figure 3 shows performance through striped drivers with multiple disks. In each case the disks were configured into a striped array. This graph shows read performance for sequential 64KB requests. FreeBSD had the best performance, but none of the OSes came close to the theoretical peak performance supported by the disks and the SCSI hardware. Figure 4 shows the effects of memory copy performance on large I/O performance. This figure compares the read performance of a striped disk array in FreeBSD using two memory copy routines. The first is the standard copy routine offered by libc. The second is a handcoded copy routine. As user level programs, the standard libc routine peaks at 40MB/s, the handcoded routine at 80MB/s. In the file system, the second routine improves read performance by about 22%.

gle host did with dedicated SCSI. This is because the SCSI bus is not a bottleneck for smaller request sizes. For the larger request sizes, the SCSI bus does become a bottleneck. When two hosts share the SCSI bus, each gets half the read bandwidth.

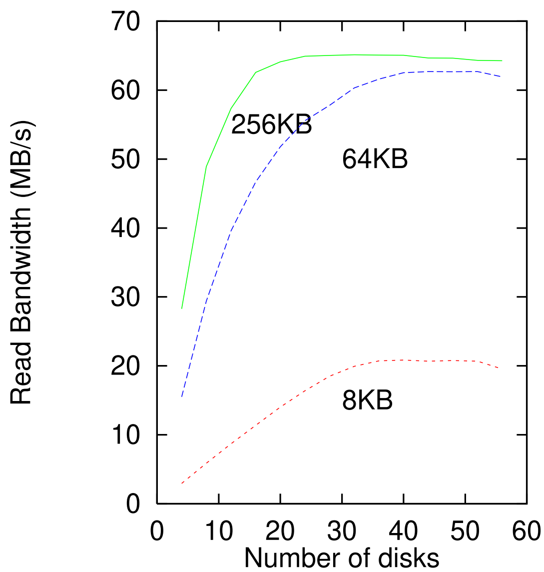
7.0 Observations

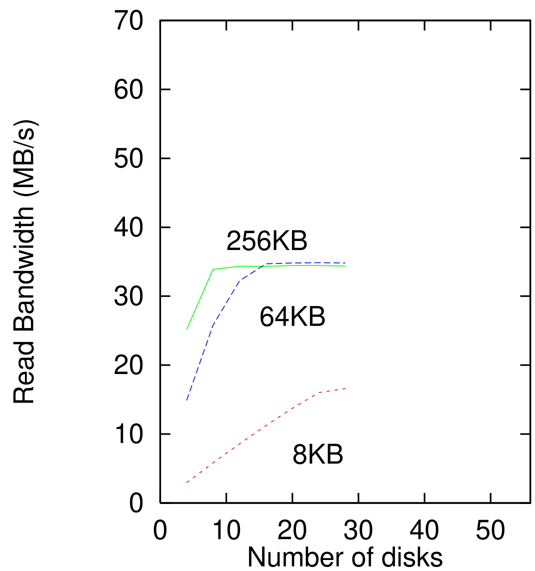
While running these experiments, we made the following general observations about the three operating systems;

Windows NT: Installation was the easiest of the three operating systems. However, detailed information is harder to find. References such as [7,8] only give very high level information. It was difficult to perform low-level measurements, of raw disk access, for instance, as the options needed to do this were not described in any of the Windows NT documentation we checked.

FreeBSD: Installation is more involved than for Windows NT. However, there is documentation available online, as well as many precompiled utilities. Mailing lists and newsgroups are also available.

Solaris: There is a bug which causes the operating system to crash during installation if more than 7 SCSI disks are placed on the same string and an IDE disk is also present in the system. If this is not the case, the installation process is similar to FreeBSD. Periodic driver updates are available from an ftp site. However, since Solaris x86 does not appear to have a large user community, not much information is available on-line. Also, the operating system does not come with a compiler.





Figures 5 and 6: Figure 5 shows raw disk read performance for multiple disks in FreeBSD. Raw disk performance in Solaris x86 is similar. For smaller requests, the disk is the bottleneck, and performance is limited by seek/rotational delays. For larger requests performance is limited by SCSI bandwidth. Figure 6 shows Shared SCSI performance. This figure shows raw read performance for shared SCSI in FreeBSD. Small request performance is not affected by shared SCSI (as the disk seek/rotational delays are the bottleneck). The performance of large requests is halved, each host getting half of the SCSI bandwidth.

Lai et al. [2] also made similar observations about FreeBSD and Solaris in their study.

8.0 Summary and Conclusions

We tested each operating system for support of large numbers of disks and SCSI controllers, support of PCI-PCI bridges, and support for shared SCSI buses. The results showed that each operating system had some subset of the required features, but none had support all the features that we tested. However, we were able to add this support to FreeBSD, the one operating system for which we had source code. This suggests that the missing features can be added for the other two operating systems as well.

The performance measurements showed that there is a wide variance in the I/O performance of the three operating systems. In our experiments, FreeBSD had consistently the best performance. However, even FreeBSD was not able to deliver close to the peak bandwidth of the underlying disk and SCSI subsystem. We also showed that optimizing the memory copy routines can greatly improve the I/O bandwidth. Finally we examined the performance implications of sharing SCSI buses. The results were quite intuitive; there was no performance penalty when the SCSI bus was not the bottleneck and each of the two machines had half the dedicated bus performance when the SCSI bus was the bottleneck.

9.0 Appendix: Memory Copy Routines

In this appendix, we describe our experiments with fast memory copies on Pentium processors. All these experiments were done under FreeBSD v2.2 on our testbed machine, a 133 Mhz Pentium with Triton chipset, 60ns EDO RAM and 256KB pipeline burst cache. All performance measurements were run as user level programs.

We provide examples of three copy routines and compare their performance. The first copy routine uses the **rep/movsl** instructions, the second uses integer registers and the third uses floating point registers. We also discuss the effectiveness of loop unrolling and prefetching when integer and floating point registers are used.

Standard libc:

We begin with the assembly code used in the libc memory copy routine. Figure A1 shows sample code for this experiment. Similar code is used for memory copies within the FreeBSD kernel.

```
movb    %cl,%al
/* copy longword-wise */
shrl    $2,%ecx
cld
rep
movsl
movb    %al,%cl
/* copy remaining bytes */
andb    $3,%cl
rep
movsb
```

Figure A1: libc memory copy routine.

This routine uses the **rep/movsl** instruction pair. The rep instruction repeatedly executes the following instruction as many times as specified in the **%ecx** register. The **movs** instruction moves data from an area pointed to by the %esi ("source index") register to that pointed to by %edi ("destination index"). Since **movsl** is used, items are moved in 32-bit longword units until there are at most 3 bytes remaining. Then the last 3 bytes are moved with **rep/movsb**.

On our testbed, this routine gets a peak performance of about 40MB/s when run as a user level program.

Using integer registers:

Figure A2 shows our next copy routine. Here we use the integer registers as temporay storage. The sample code in Figure A2 copies 64 bytes per iteration. It also attempts to prefetch the next cache line by touching the src+32 and src+64'th bytes. (We are assuming the cache line size is 32 bytes.)

```
cmpl $63,%ecx
jbe unrolled_tail

.align 2,0x90
unrolled_loop:
/* prefetch next cache line */
movl 32(%esi),%eax
cmpl $67,%ecx
/* and one more if we have */
/*   >= 68 bytes to move */
jbe unrolled_tmp
movl 64(%esi),%eax

.align 2,0x90
unrolled_tmp:
movl 0(%esi),%eax               /*
load in pairs */
movl 4(%esi),%edx
movl %eax,0(%edi)               /*
store in pairs */
movl %edx,4(%edi)
movl 8(%esi),%eax
movl 12(%esi),%edx
movl %eax,8(%edi)
movl %edx,12(%edi)
movl 16(%esi),%eax
movl 20(%esi),%edx
movl %eax,16(%edi)
movl %edx,20(%edi)
movl 24(%esi),%eax
movl 28(%esi),%edx
movl %eax,24(%edi)
movl %edx,28(%edi)
movl 32(%esi),%eax
movl 36(%esi),%edx
movl %eax,32(%edi)
movl %edx,36(%edi)
movl 40(%esi),%eax
movl 44(%esi),%edx
movl %eax,40(%edi)
movl %edx,44(%edi)
movl 48(%esi),%eax
movl 52(%esi),%edx
```

```
movl %eax,48(%edi)
movl %edx,52(%edi)
movl 56(%esi),%eax
movl 60(%esi),%edx
movl %eax,56(%edi)
movl %edx,60(%edi)
addl $-64,%ecx
addl $64,%esi
addl $64,%edi
cmpl $63,%ecx
ja unrolled_loop


unrolled_tail:
  /* this part same as libc */
movl %ecx,%eax
shrl $2,%ecx
cld
rep
movsl
movl %eax,%ecx
andl $3,%ecx
rep
movsb
```

Figure A2: Move using integer registers.

This version has a peak performance of 60MB/s on our test machine. This is a 50% speedup over the standard libc routine.

Using floating-point registers:

The last version (shown in Figure A3) uses floating point registers for temporary storage instead of integer registers. This routine also uses loop unrolling.

The Intel x86 floating-point unit has eight 80-bit registers organized as a stack. The fildq (floating-point integer load quadword) instruction loads a 64-bit integer into a 80-bit register, converting it into floating point in the process. (Note there is no data loss since the 80-bit floating-point format has 64 bits for the significand.) The fistpq (floating-point integer store and pop quadword) does the opposite .

```
cmpl $63,%ecx
jbe unrolled_tail

4:
pushl %ecx
cmpl $1792,%ecx                          /*
prefetch up to 1792 bytes */
jbe 2f                                   /*
(1792 = 2048 - 256) */
movl $1792,%ecx
2:
subl %ecx,0(%esp)
cmpl $256,%ecx
jb 5f
pushl %esi
pushl %ecx
.align 4,0x90
3:
movl 0(%esi),%eax
movl 32(%esi),%eax
movl 64(%esi),%eax
movl 96(%esi),%eax
movl 128(%esi),%eax
movl 160(%esi),%eax
movl 192(%esi),%eax
movl 224(%esi),%eax
addl $256,%esi
subl $256,%ecx
cmpl $256,%ecx
jae 3b
popl %ecx
popl %esi
5:
 .align 2,0x90
 unrolled_loop:
fildq 0(%esi)
fildq 8(%esi)
fildq 16(%esi)
fildq 24(%esi)                           /*
load 8 quad (64-bit) words */
fildq 32(%esi)
fildq 40(%esi)
fildq 48(%esi)
fildq 56(%esi)
fistpq 56(%edi)
fistpq 48(%edi)
fistpq 40(%edi)
fistpq 32(%edi)                          /*
store them in reverse order */
fistpq 24(%edi)
```

```
fistpq 16(%edi)
fistpq 8(%edi)
fistpq 0(%edi)
addl $-64,%ecx
addl $64,%esi
addl $64,%edi
cmpl $63,%ecx
ja unrolled_loop
popl %eax
addl %eax,%ecx
cmpl $64,%ecx
jae 4b


unrolled_tail:
  /* this part same as libc */
  movl %ecx,%eax
 shrl $2,%ecx
 cld
 rep
 movsl
 movl %eax,%ecx
 andl $3,%ecx
 rep
 movsb
```

Figure A3: Copy routine using floating point registers.

This version can move up to 80MB/s on our test machine. This is a 100% speedup over standard libc and 50% speedup over using integer registers.

Figure A4 compares the performance of the three copy routines for copy sizes between 32 bytes and 8K bytes. As the graph shows, the libc routine (figure A1) starts out at about 30MB/s for 32 byte copies and increases to about 40 MB/s for 8KB copies. For second and third routines, figure A4 shows copy bandwidth for different levels of loop unrolling. The level of unrolling affects the number of bytes moved in one loop iteration. As more of the loop is unrolled, the separation point between the integer/floating point register copy performance and libc moves to the right. In other words, with more unrolling, it takes a larger copy size to get the benefits. The best compromise seems to be the floating point copy with 64 bytes moved per iteration.

More information about the memory copies and more measurements can be found at *http://now.cs.berkeley.edu/Td/mcopy.html.*

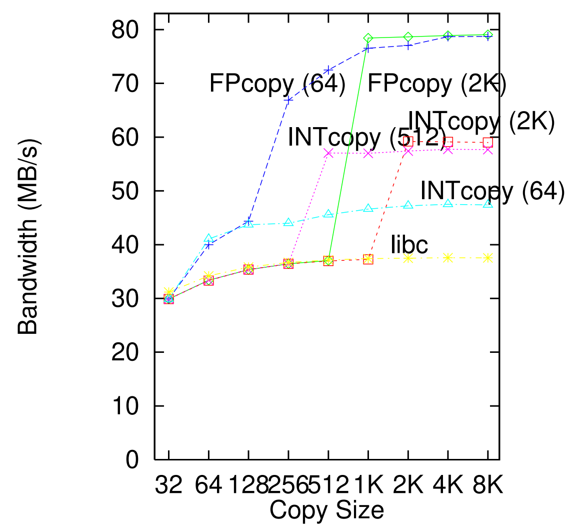


Figure A4: Performance comparison of memory copy routines using rep/movsl, integer registers and floating point registers.

10.0 References

[1] Growchowski, E.G, Hoyt, R.F, “Future Trends in Hard Disk Drives”, I*EEE Transactions on Magnetics* 32, 3 May 1996 pp1850-1854.

[2] Lai, K. Baker M. A Performance Comparison of UNIX Operating Systems on the Pentium. *Proceedings of the USENIX Technical Conference* January 1996.

[3] PCI-PCI Bridges. Bit3 Corporation. *http://www.bit3.com/*

[4] Graham, J. *Solaris 2.x: Internals and Architecture*. McGraw Hill, 1995.

[5] FreeBSD Group. *http://www.freebsd.org/*

[6] McKusik, M. *The Design and Implementation of the 4.3 BSD Operating System.* Addison Wesley Press.

[7] Helen Custer, *Inside Windows NT*, Microsoft Press,

Redmond, Washington 1993

[8] Helen Custer, *Inside the Windows NT File System*, Microsoft Press, Redmond, Washington 1994

[9] Chen, B. Endo, Y. Chan K. Mazieres D. Dias A. Seltzer M. Smith M. "The Measured Performance of Personal Computer Operating Systems" *Proceedings of SOSP* 1995.