USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Why Aren't Operating Systems Getting Faster As Fast as Hardware?

John K. Ousterhout – University of California at Berkeley

ABSTRACT

This paper evaluates several hardware platforms and operating systems using a set of benchmarks that stress kernel entry/exit, file systems, and other things related to operating systems. The overall conclusion is that operating system performance is not improving at the same rate as the base speed of the underlying hardware. The most obvious ways to remedy this situation are to improve memory bandwidth and reduce operating systems' tendency to wait for disk operations to complete.

1. Introduction

In the summer and fall of 1989 I assembled a collection of operating system benchmarks. My original intent was to compare the performance of Sprite, a UNIX-compatible research operating system developed at the University of California at Berkeley [4,5], with vendor-supported versions of UNIX running on similar hardware. After running the benchmarks on several configurations I noticed that the ''fast'' machines didn't seem to be running the benchmarks as quickly as I would have guessed from what I knew of the machines' processor speeds. In order to test whether this was a fluke or a general trend, I ran the benchmarks on a large number of hardware and software configurations at DEC's Western Research Laboratory, U.C. Berkeley, and Carnegie-Mellon University. This paper presents the results from the benchmarks.

Figure 1 summarizes the final results, which are consistent with my early observations: as raw CPU power increases, the speed of operating system functions (kernel calls, I/O operations, data moving) does not seem to be keeping pace. I found this to be true across a range of hardware platforms and operating systems. Only one ''fast'' machine, the VAX 8800, was able to execute a variety of benchmarks at speeds nearly commensurate with its CPU power, but the 8800 is not a particularly fast machine by today's standards and even it did not provide consistently good performance. Other machines ran from 10% to 10x more slowly (depending on the benchmark) than CPU power would suggest.

The benchmarks suggest at least two possible factors for the disappointing operating system performance: memory bandwidth and disk I/O. RISC architectures have allowed CPU speed to scale much faster than memory bandwidth, with the result that memory-intensive benchmarks do not receive the full benefit of faster CPUs. The second problem is in file systems. UNIX file systems require disk writes before several key operations can complete. As a result, the performance of those operations, and the performance of the file systems in general, are closely tied to disk speed and do not improve much with faster CPUs.

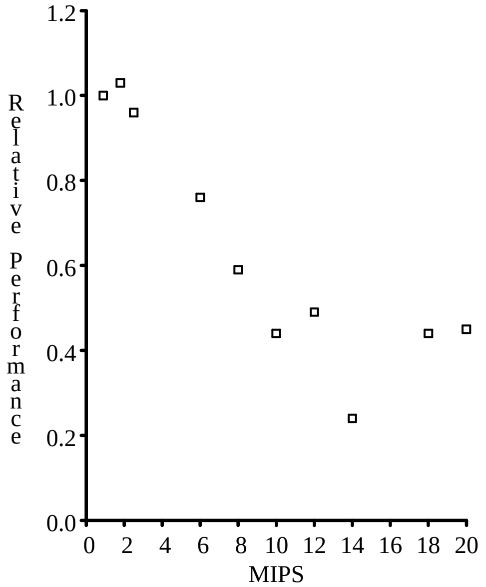

Figure 1. Summary of the results: operating system speed is not scaling at the same rate as basic hardware speed. Each point is the geometric mean of all the MIPS-relative performance numbers for all benchmarks on all operating systems on a particular machine. A value of 1.0 means that the operating system speed was commensurate with the machine's basic integer speed. The faster machines all have values much less than 1.0, which means that operating system functions ran more slowly than the machines' basic speeds would indicate.

The benchmarks that I ran are mostly ''micro-benchmarks'': they measure specific features of the hardware or operating system (such as memory-to-memory copy speed or kernel entry-exit). Micro-benchmarks make it easy to identify particular strengths and weaknesses of systems, but they do not usually provide a good overall indication of system performance. I also ran one ''macro-benchmark'' which exercises a variety of operating system features; this benchmark gives a better idea of overall system speed but does not provide much information about why some platforms are better than others. I make no guarantees that the collection of benchmarks discussed here is complete; it is possible that a different set of benchmarks might yield different results.

The rest of the paper is organized as follows. Section 2 gives a brief description of the hardware platforms and Section 3 introduces the operating systems that ran on those platforms. Sections 4-11 describe the eight benchmarks and the results of running them on various hardware-OS combinations. Section 12 concludes with some possible considerations for hardware and software designers.

2. Hardware

Table 1 lists the ten hardware configurations used for the benchmarks. It also includes an abbreviation for each configuration, which is used in the rest of the paper, an indication of whether the machine is based on a RISC processor or a CISC processor, and an approximate MIPS rating. The MIPS ratings are my own estimates; they are intended to give a rough idea of the integer performance provided by each platform. The main use of the MIPS ratings is to establish an expectation level for benchmarks. For example, if operating system performance scales with base system performance, then a DS3100 should run the various benchmarks about 1.5 times as fast as a Sun4 and about seven times as fast as a Sun3.

Hardware	Abbrev.	Type	MIPS
MIPS M2000	M2000	RISC	20
DECstation 5000	DS5000	RISC	18
H-P 9000-835CHX	HP835	RISC	14
DECstation 3100	DS3100	RISC	12
SPARCstation-1	SS1	RISC	10
Sun-4/280	Sun4	RISC	8
VAX 8800	8800	CISC	6
IBM RT-APC	RT-APC	RISC	2.5
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1. Hardware platforms on which the benchmarks were run. ‘‘MIPS’’ is an estimate of integer CPU performance, where a VAX-11/780 is approximately 1.0.

All of the machines were generously endowed with memory. No significant paging occurred in any of the benchmarks. In the file-related benchmarks, the relevant files all fit in the main-memory buffer caches maintained by the operating systems. The machines varied somewhat in their disk configurations, so small differences in I/O-intensive benchmarks should not be considered significant. However, all measurements for a particular machine used the same disk systems, except that the Mach DS3100 measurements used different (slightly faster) disks than the Sprite and Ultrix DS3100 measurements.

The set of machines in Table 1 reflects the resources available to me at the time I ran the benchmarks. It is not intended to be a complete list of all interesting machines, but it does include a fairly broad range of manufacturers and architectures.

3. Operating Systems

I used six operating systems for the benchmarks: Ultrix, SunOS, RISC/os, HP-UX, Mach, and Sprite. Ultrix and SunOS are the DEC and Sun derivatives of Berkeley’s 4.3 BSD UNIX, and are similar in many respects. RISC/os is MIPS Computer Systems’ operating system for the M2000 machine. It appears to be a derivative of System V with some BSD features added. HP-UX is Hewlett-Packard’s UNIX product and contains a combination of System V and BSD features. Mach is a new UNIX-like operating system developed at Carnegie Mellon University [1]. It is compatible with UNIX, and much of the kernel (the file system in particular) is derived from BSD UNIX. However, many parts of the kernel, including the virtual memory system and interprocess communication, have been re-written from scratch.

Sprite is an experimental operating system developed at U.C. Berkeley [4,5]; although it provides the same user interface as BSD UNIX, the kernel implementation is completely different. In particular, Sprite’s file system is radically different from that of Ultrix and SunOS, both in the ways it handles the network and in the ways it handles disks. Some of the differences are visible in the benchmark results.

The version of SunOS used for Sun4 measurements was 4.0.3, whereas version 3.5 was used for Sun3 measurements. SunOS 4.0.3 incorporates a major restructuring of the virtual memory system and file system; for example, it maps files into the virtual address space rather than keeping them in a separate buffer cache. This difference is reflected in some of the benchmark results.

4. Kernel Entry-Exit

The first benchmark measures the cost of entering and leaving the operating system kernel. It does this by repeatedly invoking the *getpid* kernel call and taking the average time per invocation. *Getpid* does nothing but return the caller’s process identifier. Table 2 shows the average time for this call on different platforms and operating systems.

The third column in the table is labeled ‘‘MIPS-Relative Speed’’. This column indicates how well the machine performed on the benchmark, relative to its MIPS rating in Table 1 and to the MVAX2 time in Table 2. Each entry in the third column was computed by taking the ratio of the MVAX2 time to the particular machine’s time, and dividing that by the ratio of the machine’s MIPS rating to the MVAX2’s MIPS rating. For example, the DS5000 time for *getpid* was 11 microseconds, and its MIPS rating is approximately 18. Thus its MIPS-relative speed is (207/11)/(18/0.9) = 0.94. A MIPS-relative speed of 1.0 means that the given machine ran the benchmark at just the speed that would be expected based on the MVAX2 time and the MIPS ratings from Table 1. A MIPS-relative speed less than 1.0 means that the machine ran this benchmark more slowly than would be expected from its MIPS rating, and a figure larger than 1.0 means the machine performed better than

Configuration	Time (μsec)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	11	0.94
M2000 RISC/os 4.0	18	0.52
DS3100 Mach 2.5	23	0.68
DS3100 Ultrix 3.1	25	0.62
DS3100 Sprite	26	0.60
8800 Ultrix 3.0	28	1.1
SS1 SunOS 4.0.3	31	0.60
SS1 Sprite	32	0.58
Sun4 Mach 2.5	32	0.73
Sun4 SunOS 4.0.3	32	0.73
Sun4 Sprite	32	0.73
HP835 HP-UX	45	0.30
Sun3 Sprite	92	1.1
Sun3 SunOS 3.5	108	0.96
RT-APC Mach 2.5	148	0.50
MVAX2 Ultrix 3.0	207	1.0

Table 2. Time for the *getpid* kernel call. ''MIPS-Relative Speed'' normalizes the benchmark speed to the MIPS ratings in Table 1: a MIPS-relative speed of .5 means the machine ran the benchmark only half as fast as would be expected by comparing the machine's MIPS rating to the MVAX2. The HP835 measurement is for *gethostid* instead of *getpid*: *getpid* appears to cache the process id in user space, thereby avoiding kernel calls on repeated invocations (the time for *getpid* was 4.3 microseconds).

might be expected.

Although the RISC machines were generally faster than the CISC machines on an absolute scale, they were not as fast as their MIPS ratings would suggest: their MIPS-relative speeds were typically in the range 0.5 to 0.8. This indicates that the cost of entering and exiting the kernel in the RISC machines has not improved as much as their basic computation speed.

5. Context Switching

The second benchmark is called *cswitch*. It measures the cost of context switching plus the time for processing small pipe reads and writes. The benchmark operates by forking a child process and then repeatedly passing one byte back and forth between parent and child using two pipes (one for each direction). Table 3 lists the average time for each round-trip between the processes, which includes one *read* and one *write* kernel call in each process, plus two context switches. As with the *getpid* benchmark, MIPS-relative speeds were computed by scaling from the MVAX2 times and the MIPS ratings in Table 1.

Once again, the RISC machines were generally faster than the CISC machines, but their MIPS-relative speeds were only in the range of 0.3 to 0.5. The only exceptions occurred with Ultrix on the DS3100, which had a MIPS-relative speed of about 0.81, and with Ultrix on the DS5000, which had a MIPS-relative speed of 1.02.

Configuration	Time (ms)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	0.18	1.02
M2000 RISC/os 4.0	0.30	0.55
DS3100 Ultrix 3.1	0.34	0.81
DS3100 Mach 2.5	0.50	0.55
DS3100 Sprite	0.51	0.54
8800 Ultrix 3.0	0.70	0.78
Sun4 Mach 2.5	0.82	0.50
Sun4 SunOS 4.0.3	1.02	0.40
SS1 SunOS 4.0.3	1.06	0.32
HP835 HP-UX	1.12	0.21
Sun4 Sprite	1.17	0.35
SS1 Sprite	1.19	0.28
Sun3 SunOS 3.5	2.36	0.78
Sun3 Sprite	2.41	0.76
RT-APC Mach 2.5	3.52	0.37
MVAX2 Ultrix 3.0	3.66	1.0

Table 3. Context switching costs, measured as the time to echo one byte back and forth between two processes using pipes.

6. Select

The third benchmark exercises the *select* kernel call. It creates a number of pipes, places data in some of those pipes, and then repeatedly calls *select* to determine how many of the pipes are readable. A zero timeout is used in each *select* call so that the kernel call never waits. Table 4 shows how long each *select* call took, in microseconds, for three configurations. Most of the commercial operating systems (SunOS, Ultrix, and RISC/os) gave performance generally in line with the machines' MIPS ratings, but HP-UX and the two research operating systems (Sprite and Mach) were somewhat slower than the others.

The M2000 numbers in Table 4 are surprisingly high for pipes that were empty, but quite low as long as at least one of the pipes contained data. I suspect that RISC/os's emulation of the *select* kernel call is faulty and causes the process to wait for 10 ms even if the calling program requests immediate timeout.

7. Block Copy

The fourth benchmark uses the *bcopy* procedure to transfer large blocks of data from one area of memory to another. It doesn't exercise the operating system at all, but different operating systems differ for the same hardware because their libraries contain different *bcopy* procedures. The main differences, however, are due to the cache organizations and memory bandwidths of the different machines.

The results are given in Table 5. For each configuration I ran the benchmark with two different block sizes. In the first case, I used blocks large enough (and aligned properly) to use *bcopy* in the most efficient way possible. At the same time the blocks were small enough that both the source and destination block fit in the cache (if any). In the second case I used a transfer

Configuration	1 pipe (μsec)	10 empty (μsec)	10 full (μsec)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	44	91	90	1.01
M2000 RISC/os 4.0	10000	10000	108	0.76
DS3100 Sprite	76	240	226	0.60
DS3100 Ultrix 3.1	81	153	151	0.90
DS3100 Mach 2.5	95	178	166	0.82
Sun4 SunOS 4.0.3	103	232	213	0.96
SS1 SunOS 4.0.3	110	221	204	0.80
8800 Ultrix 3.0	120	265	310	0.88
HP835 HP-UX	122	227	213	0.55
Sun4 Sprite	126	396	356	0.58
SS1 Sprite	138	372	344	0.48
Sun4 Mach 2.5	150	300	266	0.77
Sun3 Sprite	413	1840	1700	0.54
Sun3 SunOS 3.5	448	1190	1012	0.90
RT-APC Mach 2.5	701	1270	1270	0.52
MVAX2 Ultrix 3.0	740	1610	1820	1.0

Table 4. Time to execute a *select* kernel call to check readability of one or more pipes. In the first column a single empty pipe was checked. In the second column ten empty pipes were checked, and in the third column the ten pipes each contained data. The last column contains MIPS-relative speeds computed from the ‘‘10 full’’ case.

Configuration	Cached (Mbytes/second)	Uncached (Mbytes/second)	Mbytes/MIPS
DS5000 Ultrix 3.1D	40	12.6	0.70
M2000 RISC/os 4.0	39	20	1.00
8800 Ultrix 3.0	22	16	2.7
HP835 HP-UX	17.4	6.2	0.44
Sun4 Sprite	11.1	5.0	0.55
SS1 Sprite	10.4	6.9	0.69
DS3100 Sprite	10.2	5.4	0.43
DS3100 Mach 2.5	10.2	5.1	0.39
DS3100 Ultrix 3.1	10.2	5.1	0.39
Sun4 SunOS 4.0.3	8.2	4.7	0.52
Sun4 Mach 2.5	8.1	4.6	0.56
SS1 SunOS 4.0.3	7.6	5.6	0.56
RT-APC Mach 2.5	5.9	5.9	2.4
Sun3 Sprite	5.6	5.5	3.1
MVAX2 Ultrix 3.0	3.5	3.3	3.7

Table 5. Throughput of the *bcopy* procedure when copying large blocks of data. In the first column the data all fit in the processor's cache (if there was one) and the times represent a ‘‘warm start’’ condition. In the second column the block being transferred was much larger than the cache.

size larger than the cache size, so that cache misses occurred. In each case several transfers were made between the same source and destination, and the average bandwidth of copying is shown in Table 5.

The last column in Table 5 is a relative figure showing how well each configuration can move large uncached blocks of memory relative to how fast it executes normal instructions. I computed this figure by taking the number from the second column (‘‘Uncached’’) and dividing it by the MIPS rating from Table 1. Thus, for the 8800 the value is (16/6) = 2.7.

The most interesting thing to notice is that the CISC machines (8800, Sun3, and MVAX2) have normalized ratings of 2.7-3.7, whereas all of the RISC machines except the RT-APC have ratings of 1.0 or less. Memory-intensive applications are not likely to scale well on these RISC machines. In fact, the relative performance of memory copying drops almost monotonically with faster processors, both for RISC and CISC machines.

The relatively poor memory performance of the RISC machines is just another way of saying that the RISC approach permits much faster CPUs for a given memory system. An inevitable result of this is that memory-intensive applications will not benefit as much from RISC architectures as non-memory-intensive applications.

8. Read from File Cache

The *read* benchmark opens a large file and reads the file repeatedly in 16-kbyte blocks. For each configuration I chose a file size that would fit in the main-memory file cache. Thus the benchmark measures the cost of entering the kernel and copying data from the kernel's file cache back to a buffer in the benchmark's address space. The file was large enough that the data to be copied in each kernel call was not resident in any hardware cache. However, the same buffer was re-used to receive the data from each call; in machines with caches, the receiving buffer was likely to stay in the cache. Table 6 lists the overall bandwidth of data transfer, averaged across a large number of kernel calls.

Configuration	Mbytes/sec.	MIPS-Relative Speed
M2000 RISC/os 4.0	15.6	0.31
8800 Ultrix 3.0	10.5	0.68
Sun4 SunOS 4.0.3	8.9	0.44
Sun4 Mach 2.5	6.8	0.33
Sun4 Sprite	6.8	0.33
SS1 SunOS 4.0.3	6.3	0.25
DS5000 Ultrix 3.1D	6.1	0.13
SS1 Sprite	5.9	0.23
HP835 HP-UX	5.8	0.16
DS3100 Mach 2.5	4.8	0.16
DS3100 Ultrix 3.1	4.8	0.16
DS3100 Sprite	4.4	0.14
RT-APC Mach 2.5	3.7	0.58
Sun3 Sprite	3.7	0.80
Sun3 SunOS 3.5	3.1	0.67
MVAX2 Ultrix 3.0	2.3	1.0

Table 6. Throughput of the *read* kernel call when reading large blocks of data from a file small enough to fit in the main-memory file cache.

The numbers in Table 6 reflect fairly closely the memory bandwidths from Table 5. The only noticeable differences are for the Sun4 and the DS5000. The Sun4 does relatively better in this benchmark due to its write-back cache. Since the receiving buffer always stays in the cache, its contents get overwritten without ever being flushed to memory. The other machines all had write-through caches, which caused information in the buffer to be flushed immediately to memory. The second difference from Table 5 is for the DS5000, which was much slower than expected; I do not have an explanation for this discrepancy.

9. Modified Andrew Benchmark

The only large-scale benchmark in my test suite is a modified version of the Andrew benchmark developed by M. Satyanarayanan for measuring the performance of the Andrew file system [3]. The benchmark operates by copying a directory hierarchy containing the source code for a program, *stat*-ing every file in the new hierarchy, reading the contents of every copied file, and finally compiling the code in the copied hierarchy.

Satyanarayanan's original benchmark used whichever C compiler was present on the host machine, which made it impossible to compare running times between machines with different architectures or different compilers. In order to make the results comparable between different machines, I modified the benchmark so that it always uses the same compiler, which is the GNU C compiler generating code for an experimental machine called SPUR [2].

Table 7 contains the raw Andrew results. The table lists separate times for two different phases of the benchmark. The ''copy'' phase consists of everything except the compilation (all of the file copying and scanning), and the ''compile'' phase consists of just the compilation. The copy phase is much more I/O-intensive than the compile phase, and it also makes heavy use of the mechanisms for process creation (for example, a separate shell command is used to copy each file). The compile phase is CPU-bound on the slower machines, but spends a significant fraction of time in I/O on the faster machines.

I ran the benchmark in both local and remote configurations. ''Local'' means that all the files accessed by the benchmark were stored on a disk attached to the machine running the benchmark. ''Diskless'' refers to Sprite configurations where the machine running the benchmark had no local disk; all files, including the program binaries, the files in the directory tree being copied and compiled, and temporary files were accessed over the network using the Sprite file system protocol [4]. ''NFS'' means that the NFS protocol [7] was used to access remote files. For the SunOS NFS measurements the machine running the benchmark was diskless. For the Ultrix and Mach measurements the machine running the benchmark had a local disk that was used for temporary files, but all other files were accessed remotely. ''AFS'' means that the Andrew file system protocol was used for accessing remote files [3]; this configuration was similar to NFS under Ultrix in that the machine running the benchmark had a local disk, and temporary files were stored on that local disk while other files were accessed remotely. In each case the server was the same kind of machine as the client.

Table 8 gives additional ''relative'' numbers: the MIPS-relative speed for the local case, the MIPS-relative speed for the remote case, and the percentage slow-down experienced when the benchmark ran with a remote disk instead of a local one.

There are several interesting results in Tables 7 and 8. First of all, the faster machines generally have smaller MIPS-relative speeds than the slower machines. This is easiest to see by comparing different machines running the same operating system, such as Sprite on the Sun3, Sun4 and DS3100 (1.7, 0.93, and 0.89 respectively for the local case) or SunOS on the Sun3, Sun4, and SS1 (1.4, 0.85, 0.66 respectively for the local case) or Ultrix on the MVAX2, 8800, and DS3100 (1.0, 0.93, and 0.50 respectively for the local case).

The second overall result from Tables 7 and 8 is that Sprite is faster than other operating systems in every case except in comparison to Mach on the Sun4. For the local case Sprite was generally 10-20% faster, but in the remote case Sprite was typically 30-70% faster. On Sun-4's and DS3100's, Sprite was 60-70% faster than either Ultrix, SunOS, or Mach for the remote case. In fact, the Sprite-DS3100 combination ran the benchmark remotely 45%

Configuration	Copy (seconds)	Compile (seconds)	Total (seconds)
M2000 RISC/os 4.0 Local	13	59	72
DS3100 Sprite Local	22	98	120
DS5000 Ultrix 3.1D Local	48	76	124
DS3100 Sprite Diskless	34	93	127
DS3100 Mach 2.5 Local	29	107	136
Sun4 Mach 2.5 Local	37	122	159
Sun4 Sprite Local	44	128	172
Sun4 Sprite Diskless	56	128	184
DS5000 Ultrix 3.1D NFS	68	118	186
Sun4 SunOS 4.0.3 Local	54	133	187
SS1 SunOS 4.0.3 Local	54	139	193
DS3100 Mach 2.5 NFS	58	147	205
DS3100 Ultrix 3.1 Local	80	133	213
8800 Ultrix 3.0 Local	48	181	229
SS1 SunOS 4.0.3 NFS	76	168	244
DS3100 Ultrix 3.1 NFS	115	154	269
Sun4 SunOS 4.0.3 NFS	92	213	305
Sun3 Sprite Local	52	375	427
RT-APC Mach 2.5 Local	89	344	433
Sun3 Sprite Diskless	75	364	439
Sun3 SunOS 3.5 Local	69	406	475
RT-APC Mach 2.5 AFS	128	397	525
Sun3 SunOS 3.5 NFS	157	478	635
MVAX2 Ultrix 3.0 Local	214	1202	1416
MVAX2 Ultrix 3.0 NFS	298	1409	1707

Table 7. Elapsed time to execute a modified version of M. Satyanarayanan's Andrew benchmark [3]. The first column gives the total time for all of the benchmark phases except the compilation phase. The second column gives the elapsed time for compilation, and the third column gives the total time for the benchmark. The entries in the table are ordered by total execution time.

Configuration	MIPS-Relative Speed (Local)	MIPS-Relative Speed (Remote)	Remote Penalty (%)
M2000 RISC/os 4.O Local	0.88	--	--
8800 Ultrix	0.93	--	--
Sun-4 Mach 2.5	1.0	--	--
Sun3 Sprite	1.7	1.9	3
DS3100 Sprite	0.89	1.01	6
Sun4 Sprite	0.93	1.04	7
MVAX2 Ultrix 3.0 NFS	1.0	1.0	21
RT-APC Mach 2.5 AFS	1.2	1.2	21
DS3100 Ultrix 3.1 NFS	0.50	0.48	26
SS1 SunOS 4.0.3 NFS	0.66	0.63	26
Sun3 SunOS 3.5 NFS	1.4	1.3	34
DS3100 Mach 2.5 NFS	0.78	0.62	50
DS5000 Ultrix 3.1D NFS	0.57	0.46	50
Sun4 SunOS 4.0.3 NFS	0.85	0.63	63

Table 8. Relative performance of the Andrew benchmark. The first two columns give the MIPS-relative speed, both local and remote. The first column is computed relative to the MVAX2 Ultrix 3.0 Local time and the second column is computed relative to MVAX2 Ultrix 3.0 NFS. The third column gives the remote penalty, which is the additional time required to execute the benchmark remotely, as a percentage of the time to execute it locally. The entries in the table are ordered by remote penalty.

faster than Ultrix-DS5000, even though the Ultrix-DS5000 combination had about 50% more CPU power to work with. Table 8 shows that Sprite ran the benchmark almost as fast remotely as locally, whereas the other systems slowed down by 20-60% when running remotely with NFS or AFS. It appears that the penalty for using NFS is increasing as machine speeds increase: the MVAX2 had a remote penalty of 21%, the Sun3 34%, and the Sun4 63%.

The third interesting result of this benchmark is that the DS3100-Ultrix-Local combination is slower than I had expected: it is about 24% slower than DS3100-Mach-Local and 78% slower than DS3100-Sprite-Local. The DS3100-Ultrix combination did not experience as great a remote penalty as other configurations, but this is because the local time is unusually slow.

Configuration	‘‘foo’’ (ms)	‘‘a/b/c/foo’’ (ms)	MIPS-Relative Speed
DS5000 Ultrix 3.1D Local	0.16	0.31	0.91
DS3100 Mach 2.5 Local	0.19	0.33	1.1
Sun4 SunOS 4.0.3 Local	0.25	0.38	1.3
DS3100 Ultrix 3.1 Local	0.27	0.41	0.81
Sun4 Mach 2.5 Local	0.30	0.40	1.1
SS1 SunOS 4.0.3 Local	0.31	0.44	0.84
M2000 RISC/os 4.0 Local	0.32	0.83	0.41
HP835 HP-UX Local	0.38	0.61	0.49
8800 Ultrix 3.0 Local	0.45	0.68	0.97
DS3100 Sprite Local	0.82	0.97	0.27
RT-APC Mach 2.5 Local	0.95	1.6	1.1
Sun3 SunOS 3.5 Local	1.1	2.2	1.3
Sun4 Sprite Local	1.2	1.4	0.27
RT-APC Mach 2.5 AFS	1.7	3.5	7.6
DS5000 Ultrix 3.1D NFS	2.4	2.4	0.75
MVAX2 Ultrix 3.0 Local	2.9	4.7	1.0
SS1 SunOS 4.0.3 NFS	3.4	3.5	0.95
Sun4 SunOS 4.0.3 NFS	3.5	3.7	1.2
DS3100 Mach 2.5 NFS	3.6	3.9	0.75
DS3100 Ultrix 3.1 NFS	3.8	3.9	0.71
DS3100 Sprite Diskless	4.3	4.4	0.63
Sun3 Sprite Local	4.3	5.2	0.34
Sun4 Sprite Diskless	6.1	6.4	0.67
HP835 HP-UX NFS	7.1	7.3	0.33
Sun3 SunOS 3.5 NFS	10.4	11.4	1.7
Sun3 Sprite Diskless	12.8	16.3	1.4
MVAX2 Ultrix 3.0 NFS	36.0	36.9	1.0

Table 9. Elapsed time to open a file and then close it again, using the *open* and *close* kernel calls. The MIPS-relative speeds are for the ‘‘foo’’ case, scaled relative to MVAX2 Ultrix 3.0 Local for local configurations and relative to MVAX2 Ultrix 3.0 NFS for remote configurations.

10. Open-Close

The modified Andrew benchmark suggests that the Sprite file system is faster than the other file systems, particularly for remote access, but it doesn’t identify which file system features are responsible. I ran two other benchmarks in an attempt to pinpoint the differences. The first benchmark is open-close, which repeatedly opens and closes a single file. Table 9 displays the cost of an open-close pair for two cases: a name with only a single element, and one with 4 elements. In both the local and remote cases the UNIX derivatives are consistently faster than Sprite. The remote times are dominated by the costs of server communication: Sprite communicates with the server on every open or close, NFS occasionally communicates with the server (to check the consistency of its cached naming information), and AFS virtually never checks with the server (the server must notify the client if any cached information becomes invalid). Because of its ability to avoid all server interactions on repeated access to the same file, AFS was by far the fastest remote file system for this benchmark.

Although this benchmark shows dramatic differences in open-close costs, it does not seem to explain the performance differences in Table 8. The MIPS-relative speeds vary more from operating system to operating system than from machine to machine. For example, all the Mach and Ultrix MIPS-relative speeds were in the range 0.8 to 1.1 for the local case, whereas all the Sprite MIPS-relative speeds were in the range 0.27 to 0.34 for the local case.

11. Create-Delete

The last benchmark was perhaps the most interesting in terms of identifying differences between operating systems. It also helps to explain the results in Tables 7 and 8. This benchmark simulates the creation, use, and deletion of a temporary file. It opens a file, writes some amount of data to the file, and closes the file. Then it opens the file for reading, reads the data, closes the file, and finally deletes the file. I tried three different amounts of data: none, 10 kbytes, and 100 kbytes. Table 10 gives the total time to create, use, and delete the file in each of several hardware/operating system configurations.

This benchmark highlights a basic difference between Sprite and the UNIX derivatives. In Sprite, short-lived files can be created, used, and deleted without any data ever being written to disk. Information only goes to disk after it has lived at least 30 seconds. Sprite requires only a single disk I/O for each iteration of the benchmark, to write out the file’s i-node after it has been deleted. Thus in the best case (DS3100’s) each iteration takes one disk rotation, or about 16 ms. Even this one I/O

Configuration	No data (ms)	10 kbytes (ms)	100 kbytes (ms)	MIPS-Relative Speed
DS3100 Sprite Local	17	34	69	0.44
Sun4 Sprite Local	18	33	67	0.63
DS3100 Sprite Remote	33	34	68	0.67
Sun3 Sprite Local	33	47	130	1.5
M2000 RISC/os 4.0 Local	33	51	116	0.12
Sun4 Sprite Remote	34	50	71	0.98
8800 Ultrix 3.0 Local	49	100	294	0.31
DS5000 Ultrix 3.1D Local	50	86	389	0.09
Sun4 Mach 2.5 Local	50	83	317	0.23
DS3100 Mach 2.5 Local	50	100	317	0.15
HP835 HP-UX Local	50	115	263	0.13
RT-APC Mach 2.5 Local	53	121	706	0.68
Sun3 Sprite Remote	61	73	129	2.42
SS1 SunOS 4.0.3 Local	65	824	503	0.14
Sun4 SunOS 4.0.3 Local	67	842	872	0.17
Sun3 SunOS 3.5 Local	67	105	413	0.75
DS3100 Ultrix 3.1 Local	80	146	548	0.09
SS1 SunOS 4.0.3 NFS	82	214	1102	0.32
DS5000 Ultrix 3.1D NFS	83	216	992	0.18
DS3100 Mach 2.5 NFS	89	233	1080	0.25
Sun4 SunOS 4.0.3 NFS	97	336	2260	0.34
MVAX2 Ultrix 3.0 Local	100	197	841	1.0
DS3100 Ultrix 3.1 NFS	116	370	3028	0.19
RT-APC Mach 2.5 AFS	120	303	1615	0.89
Sun3 SunOS 3.5 NFS	152	300	1270	0.97
HP835 HP-UX NFS	180	376	1050	0.11
MVAX2 Ultrix 3.0 NFS	295	634	2500	1.0

Table 10. Elapsed time to create a file, write some number of bytes to it, close the file, then re-open the file, read it, close it, and delete it. This benchmark simulates the use of a temporary file. The Mach-AFS combination showed great variability: times as high as 460ms/721ms/2400ms were as common as the times reported above (the times in the table were the lowest ones seen). MIPS-relative speeds were computed using the ‘‘No Data’’ times in comparison to the MVAX2 local or NFS time. The table is sorted in order of ‘‘No Data’’ times.

is an historical artifact that is no longer necessary; a newer version of the Sprite file system eliminates it, resulting in a benchmark time of only 4 ms for the DS3100-Local-No-Data case.

UNIX and its derivatives are all much more disk-bound than Sprite. When files are created and deleted, several pieces of information must be forced to disk and the operations cannot be completed until the I/O is complete. Even with no data in the file, the UNIX derivatives all required 35-100 ms to create and delete the file. This suggests that information like the file’s i-node, the entry in the containing directory, or the directory’s i-node is being forced to disk. In the NFS-based remote systems, newly-written data must be transferred over the network to the file server and then to disk before the file may be closed. Furthermore, NFS forces each block to disk independently, writing the file’s i-node and any dirty indirect blocks once for each block in the file. This results in up to 3 disk I/O’s for each block in the file. In AFS, modified data is returned to the file server as part of the close operation. The result of all these effects is that the performance of the create-delete benchmark under UNIX (and the performance of temporary files under UNIX) are determined more by the speed of the disk than by the speed of the CPU.

The create-delete benchmark helps to explain the poor performance of DS3100 Ultrix on the Andrew benchmark. The basic time for creating an empty file is 60% greater in DS3100-Ultrix-Local than in 8800-Ultrix-Local, even though the DS3100 CPU is twice as fast as the 8800 CPU. The time for a 100-kbyte file in DS3100-Ultrix-NFS is 45 times as long as for DS3100-Sprite-Diskless! The poor performance relative to the 8800 may be due in part to slower disks (RZ55’s on the DS3100’s). However, Ultrix’s poor remote performance is only partially due to NFS’s flush-on-close policy: DS3100-Ultrix-NFS achieves a write bandwidth of only about 30 kbytes/sec on 100 Kbyte files, which is almost twice as slow as I measured on the same hardware running an earlier version of Ultrix (3.0) and three times slower than Mach. I suspect that Ultrix could be tuned to provide substantially better file system performance.

Lastly, Table 10 exposes some surprising behavior in SunOS 4.0.3. The benchmark time for a file with no data is 67 ms, but the time for 10 kbytes is 842 ms, which is almost an order of magnitude slower than SunOS 3.5 running on a Sun3! This was so surprising that I also tried data sizes of 2-9 kbytes at 1-kbyte intervals. The SunOS 4.0.3 time stayed in the 60-80ms range until the file size increased from 8 kbytes to 9 kbytes; at this point it jumped up to about 800 ms. This anomaly is not present in other UNIX derivatives, or even in earlier versions of SunOS. Since the jump occurs at a file size equal to the page size, I hypothesize that it is related to the implementation of mapped files in SunOS 4.0.3.

12. Conclusions

For almost every benchmark the faster machines ran more slowly than I would have guessed from raw processor speed. Although it is dangerous to draw far-reaching conclusions from a small set of benchmarks, I think that the benchmarks point out four potential problem areas, two for hardware designers and two for operating system developers.

12.1 Hardware Issues

The first hardware-related issue is memory bandwidth: the benchmarks suggest that it is not keeping up with CPU speed. Part of this is due to the 3-4x difference in CPU speed relative to memory bandwidth in newer RISC architectures versus older CISC architectures; this is a one-time-only effect that occurred in the shift from RISC to CISC. However, I believe it will be harder for system designers to improve memory performance as fast as they improve processor performance in the years to come. In particular, workstations impose severe cost constraints that may encourage designers to skimp on memory system performance. If memory bandwidth does not improve dramatically in future machines, some classes of applications may be limited by memory performance.

A second hardware-related issue is context switching. The *getpid* and *cswitch* benchmarks suggest that context switching, both for kernel calls and for process switches, is about 2x more expensive in RISC machines than in CISC machines. I don't have a good explanation for this result, since the extra registers in the RISC machines cannot account for the difference all by themselves. A 2x degradation may not be serious, as long as the relative performance of context switching doesn't degrade any more in future machines.

12.2 Software Issues

In my view, one of the greatest challenges for operating system developers is to decouple file system performance from disk performance. Operating systems derived from UNIX use caches to speed up reads, but they require synchronous disk I/O for operations that modify files. If this coupling isn't eliminated, a large class of file-intensive programs will receive little or no benefit from faster hardware. Of course, delaying disk writes may result in information loss during crashes; the challenge for operating system designers is to maintain an acceptable level of reliability while decoupling performance.

One approach that is gaining in popularity is to use non-volatile memory as a buffer between main memory and disk. Information can be written immediately (and efficiently) to the non-volatile memory so that it will survive crashes; long-lived information can eventually be written to disk. This approach involves extra complexity and overhead to move information first to non-volatile memory and then to disk, but it may result in better overall performance than writing immediately to disk. Another new approach is to use log-structured file systems, which decouple file system performance from disk performance and make disk I/O's more efficient. See [6] for details.

A final consideration is in the area of network protocols. In my opinion, the assumptions inherent in NFS (statelessness and write-through-on-close, in particular) represent a fundamental performance limitation. If users are to benefit from faster machines, either NFS must be scrapped (my first choice), or NFS must be changed to be less disk-intensive.

13. Code Availability

The source code for all of these benchmarks is available via public FTP from ucbvax.berkeley.edu. The file `pub/mab.tar.Z` contains the modified Andrew benchmark and `pub/bench.tar.Z` contains all of the other benchmarks.

14. Acknowledgments

M. Satyanarayanan developed the original Andrew benchmark and provided me with access to an IBM RT-APC running Mach. Jay Kistler resolved the incompatibilities that initially prevented the modified Andrew benchmark from running under Mach. Jeff Mogul and Paul Vixie helped me get access to DEC machines and kernels for testing, and explained the intricacies of configuring NFS. Rick Rashid provided me with benchmark results for Mach running on DS3100's and Sun4's. Joel McCormack and David Wall provided helpful comments on earlier drafts of this paper.

15. References

[1] Accetta, M., et al. ''Mach: A New Kernel Foundation for UNIX Development.'' *Proceedings of the USENIX 1986 Summer Conference*, July 1986, pp. 93-113.

[2] Hill, M., et al. ''Design Decisions in SPUR.'' *IEEE Computer*, Vol. 19, No. 11, November 1986, pp. 8-22.

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

[4] Nelson, M., Welch, B., and Ousterhout, J. ''Caching in the Sprite Network File System.'' *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134-154.

[5] Ousterhout, J., et al. ''The Sprite Network Operating System.'' *IEEE Computer*, Vol. 21, No. 2, February 1988, pp. 23-36.

[6] Rosenblum, M., and Ousterhout, J. ''The LFS

Storage Manager.'' *Proceedings of the USENIX 1990 Summer Conference*, June 1990.

[7] Sandberg, R., et al. ''Design and Implementation of the Sun Network Filesystem.'' *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.