Virtual Memory vs. The File System

Michael N. Nelson

Digital Equipment Corporation
Western Research Laboratory
Palo Alto, CA 94301

Abstract

This paper examines the behavior of mechanisms for providing variable-size file data caches. It presents the results of running virtual-memory- and file-intensive benchmarks on the Sprite operating system [OCD88]; the benchmarks are designed to simulate real-life applications that represent the worst case for variable-size cache mechanisms. The results indicate that variable-size cache mechanisms work well when virtual-memory- and file-intensive programs are run in sequence; the cache is able to change in size in order to provide overall performance no worse than that provided by a small fixed-size cache. However, when interactive programs are run concurrently with file-intensive programs, variable-size cache mechanisms perform very poorly if file pages and virtual-memory pages are treated equally. In order to guarantee good interactive response, virtual memory pages must be given preference over file pages.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00039-84-C-0107

Virtual Memory vs. The File System

1. Introduction

File data caches have been used in many operating systems to improve file system performance. In a distributed system the use of caches can reduce both network and disk traffic. A study of the use of caches on diskless workstations [NWO88] showed that the use of large caches can reduce the execution time of application programs by up to 1/3. Unfortunately, if file data caches are allowed to become too large, then they will conflict with the needs of the virtual memory system. In particular, if there is insufficient memory to run application programs, then the programs may slow down by factors of 10 to 100 because of excessive paging activity. Thus, if a cache is allowed to become too large, the improvement in file system performance may be more than offset by a degradation in virtual memory performance.

In order to provide both good file system performance and good virtual memory performance, several operating systems [BBM72, DaD68, Lea83, RaF86, Ras87] have implemented variable-size cache mechanisms. In these operating systems the portion of memory used for file data and virtual memory varies in response to the file and virtual-memory needs of the application programs being executed. These mechanisms will obviously work well when there is little or no contention for memory between file and virtual-memory pages.

This paper examines the behavior of variable-size cache mechanisms in the worst case; that is, the case when there is a serious amount of memory contention. It looks at both sequential and concurrent contention. Sequential contention corresponds to a user alternatively running programs that are either virtual memory or file intensive; an example of this type of contention is an edit-compile-debug loop where the editing and debugging are virtual-memory intensive and the compile is file intensive. Concurrent contention is when a user is running file intensive and virtual-memory intensive programs at the same time; an example of this type of sharing is a user interacting with a window system while a file intensive program is running.

I evaluated the behavior of variable-size cache mechanisms by running benchmarks that simulate both sequential and concurrent memory contention. The benchmark results indicate that variable-size cache mechanisms work well for sequential contention; the cache is able to change in size in order to provide overall performance no worse than that provided by a small fixed-size cache. However, when interactive programs are run concurrently with file intensive programs, variable-size cache mechanisms perform very poorly. The file intensive programs steal memory away from the interactive programs causing the interactive programs to exhibit poor response time because of extra page faults. In order to guarantee good interactive performance, virtual-memory pages should receive preferential treatment: a file page should not replace a virtual-memory page unless the virtual-memory page has been idle for a substantial period (e.g. 5 to 10 minutes).

The rest of the paper is organized as follows: Section 2 looks at mechanisms for providing variable-size caches including the mechanism provided in Sprite; Section 3 describes the benchmarks that were used to evaluate the variable-size caching schemes; Section 4 looks at the performance of fixed-size cache schemes; Section 5 sees how well the variable-size cache schemes compare to fixed-size schemes; Section 6 examines the effect of giving virtual memory data preference over file data; and Section 7 offers some conclusions.

2. Providing Variable-size Caches

The approach that has been commonly used to provide variable-size file data caches is to combine the virtual memory and file systems together; this is generally called the *mapped-file* approach. To access a file, it is first mapped into a process's virtual address space and then read and written just like virtual memory. This approach eliminates the file cache entirely; the standard page replacement mechanisms automatically balance physical memory usage between file and program information. Mapped files were first used in Multics [BCD72, DaD68] and TENEX [BBM72, Mur72]. More recently they have been implemented in Pilot [Red80], Accent [RaR81, RaF86], Apollo [LLH85, Lea83] and Mach [Ras87].

The Sprite approach to providing variable-size caches is quite different from the mapped-file approach. In Sprite, the file system and virtual memory system are separate. Users invoke system calls such as *read* and *write* to access file data. These system calls copy data between the file cache and the virtual address spaces of user processes. Variable-size caches are provided by having the virtual memory system and file system modules negotiate over physical memory usage. In this paper I will only give an overview of the Sprite mechanism; see [NWO88] or [Nel88] for more details.

In the Sprite mechanism, the file system module and the virtual memory module each manage a separate pool of physical memory pages. Virtual memory keeps its pages in approximate LRU order through a version of the clock algorithm [Nel86]. The file system keeps its cache blocks in perfect LRU order since all block accesses are made through the *read* and *write* system calls. Each module keeps a time-of-last-access for each page or block. Whenever either module needs additional memory (because of a page fault or a miss in the file cache), it compares the age of its oldest page with the age of the oldest page from the other module. If the other module has the oldest page, then it is forced to give up that page; otherwise the module recycles its own oldest page.

The advantage that the Sprite approach has over mapped-file approaches is that it makes it easy to discriminate between file and virtual-memory pages. This makes Sprite a good vehicle for running experiments in variable-size cache behavior. Of course, Sprite has the disadvantage that it requires more copies than mapped-file schemes. However, measurements presented in [Nel88] demonstrate that the extra copies have an insignificant impact on performance.

3. Benchmarks

In order to measure the performance of variable-size cache mechanisms, I developed two benchmarks and ran them on the Sprite operating system. The results obtained on Sprite should be similar to results obtained with the same benchmarks on systems with mapped-files.

The first benchmark that I used is an edit-compile-debug (ECD) benchmark that runs under the X11 window system on Sprite (see Table 1). This benchmark represents work that is commonly done on Sprite, and is both VM and FS intensive. Each program in the benchmark is run in sequence; no two programs are ever run concurrently. I used this benchmark to determine how well variable-size cache mechanisms and fixed-size cache mechanisms perform with sequential memory contention.

Phase	Description	FS I/O	VM Image Size
Edit	Run window-based editor on 2500 line file.	70 Kbytes	560 Kbytes
Compile	Compile VM Module	800 Kbytes	1 Mbyte
Link	Link the kernel	8 Mbytes	3 Mbytes
Debug	Run kernel debugger	4 Mbytes	8.5 Mbytes
Environment	The X window system plus several typescript windows and tools.	--	5 Mbytes

Table 1. The phases of the edit-compile-debug benchmark. The first two columns describe the phase of the benchmark. The third column gives the number of bytes read and written by each phase. The last column gives the size of the largest virtual memory image of the phase. The last row is not a phase in the benchmark but rather shows the total amount of memory required by the basic environment in which the benchmark is running.

The ECD benchmark was run on a Sun-3/75 workstation with from 10 to 16 Mbytes of physical memory; if less than 10 Mbytes was used then the benchmark did not finish in a reasonable amount of time because of virtual-memory thrashing. The workstation's files were stored remotely on a Sun-3/180 file server with 16 Mbytes of memory†. Although the benchmark was executed on a diskless workstation, the results should be similar to results obtained by running the benchmark on a machine with a local disk. Each benchmark consisted of two runs through the edit-compile-link-debug loop. Each data point was taken from the average of three runs of the benchmark.

The other benchmark that I used runs a virtual-memory intensive program and a file-intensive program concurrently. This benchmark is used to determine the impact of variable-size cache mechanisms on interactive performance; I will refer to this benchmark as the IFS (Interactive-File-System) benchmark. The virtual-memory-intensive program is a program which periodically touches many pages in its virtual address space, dirtying some of them‡. This simulates a user who is interacting with a program. When a user interacts with a program, the program must have its code, heap and stack pages memory-resident in order to give good interactive response. In fact, if the user is interacting with a program under a window system such as X11, then several programs have to be memory-resident in order for the user to get good interactive response.

The file system component of the IFS benchmark is the UNIX sort program run on a 1-Mbyte file. The UNIX sort program is an external merge sort which uses many

† In this paper the term *client* will be used to refer to the diskless workstation that the benchmarks were run on and the term *server* will be used to refer to the workstation that stored the client's files.

‡ The fraction of memory that the interactive program dirties each time it touches the memory in its address space may impact the performance of the IFS benchmark. Measurements of 5 workstations running Sprite showed that between 40 and 60 percent of the memory that was being used by user processes was dirty. For this reason the virtual-memory intensive program dirties half of the pages that it touches.

temporary files. The *sort* program is run concurrently with the interactive program to simulate a file system program that attempts to grow its cache by stealing memory from an interactive program.

The IFS benchmark was run on a diskless 8-Mbyte Sun-3/75. The file server was a Sun-3/180 with 16 Mbytes of memory. 1.3 Mbytes of the 8 Mbytes were used by the kernel, which left 6.7 Mbytes for user processes. The interactive program used 5.7 Mbytes of memory and left at most 1 Mbyte for *sort* and the file system cache. This is small enough that *sort* will contend with the virtual memory system for memory.

4. Why Not Just Use Fixed-Size Caches?

The results from previous measurements of file data caches [NWO88] suggest that a large fixed-size cache will provide the best performance for file-intensive programs. However, for the two benchmarks used here, a small fixed-size cache is best. Figure 1 gives the elapsed time and server utilization for the ECD benchmark as a function of the amount of physical memory available on the client and the size of its file cache. A cache of 0.5 Mbytes provides the lowest elapsed time, and a cache from 0.5 Mbytes to 1 Mbyte gives the lowest server utilization for the benchmark; note that this benchmark is so virtual-memory intensive that even with the largest physical memory the smallest file cache is best.

Figure 2 clearly shows why the smallest cache is best for the ECD benchmark. As the cache grows in size, the number of file system bytes transferred drops. However, larger file caches leave less memory for virtual memory, so the number of page faults increases, resulting in more network traffic to fetch VM pages. This causes an increase in the total number of network bytes transferred and a corresponding increase in client degradation and server utilization.

A small fixed-size cache is also best for the IFS benchmark. This benchmark was designed so that interactive performance would degrade if more than 1 Mbyte were used for the file system cache. In addition, measurements in [NWO88] show that even with a small cache the *sort* benchmark will only execute at most 25 percent more slowly than with a large cache. Thus a small fixed-size cache will give instantaneous interactive response (no page faults required) while only slightly degrading file system performance.

The results with fixed-size caches demonstrate that different cache sizes are needed for different types of programs. The results in [NWO88] show that when purely file-intensive programs are run, a large cache is best. However, when a mix of file- and virtual-memory-intensive programs are run, then a small cache is best.

5. Variable-Size Cache Performance

It is clear from the previous section that different file cache sizes are required for different types of programs. Variable-size cache mechanisms attempt to provide the ability to adjust the size of the cache based on the types of programs that are being run. This section examines how the Sprite variable-size cache mechanism affects the performance of the ECD and IFS benchmarks. Measurements of other variable-size cache mechanisms should yield similar results.

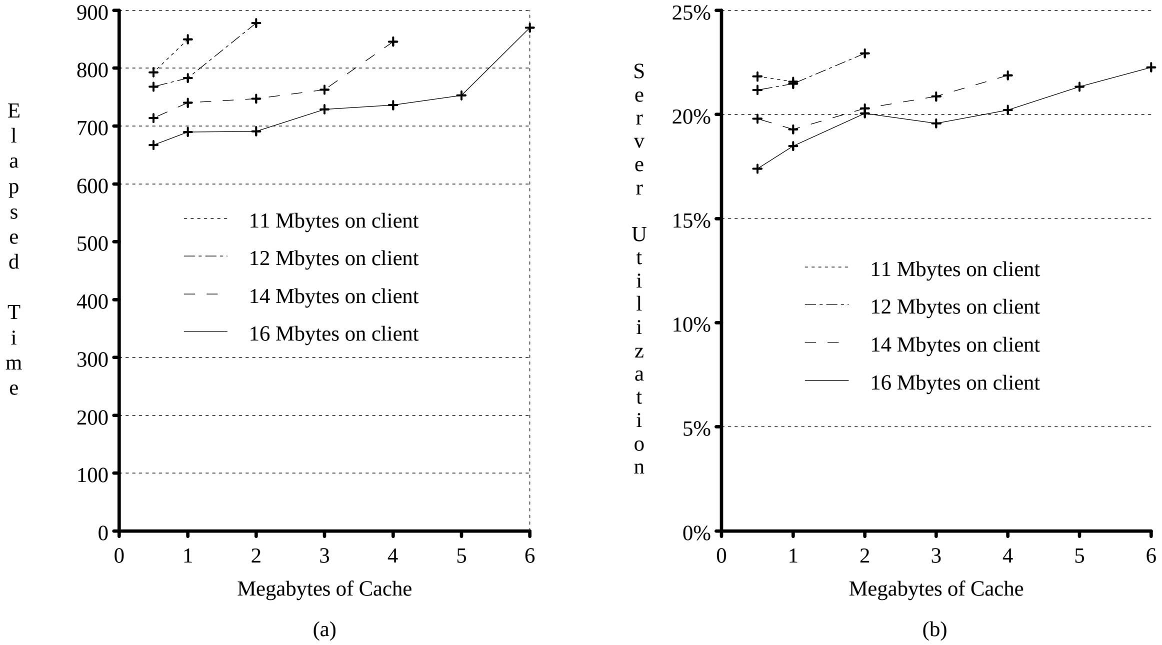

Figure 1. Elapsed time and server utilization for the edit-compile-debug benchmark with fixed-size caches as a function of client physical memory size. In both graphs the X-axis is the size of the client's file cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark. The system thrashed whenever the amount of physical memory left for the virtual memory system dropped below 10 Mbytes. I did not run the benchmark for points beyond where thrashing occurred (since elapsed time more than doubles), which explains why some curves have fewer data points than others.

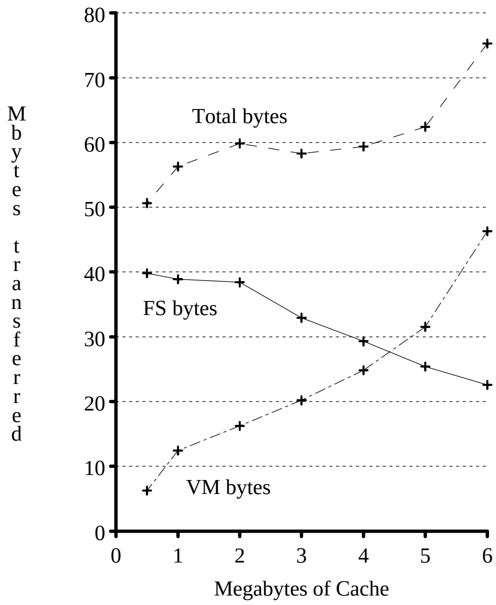

Figure 2. This graph gives the number of Mbytes transferred across the network for the edit-compile-debug benchmark with fixed-size caches and 16 Mbytes of memory on the client. Graphs of network bytes transferred for the other four memory sizes yield similar results. The X-axis is the size of the cache and the Y-axis is the number of Mbytes transferred. The ''FS bytes'' line is the amount of file system data transferred across the network, the ''VM bytes'' line is the amount of virtual memory data transferred across the network, and the ''Total bytes'' line is the total amount of network bytes transferred which includes file and virtual memory data as well as packet headers and control packets.

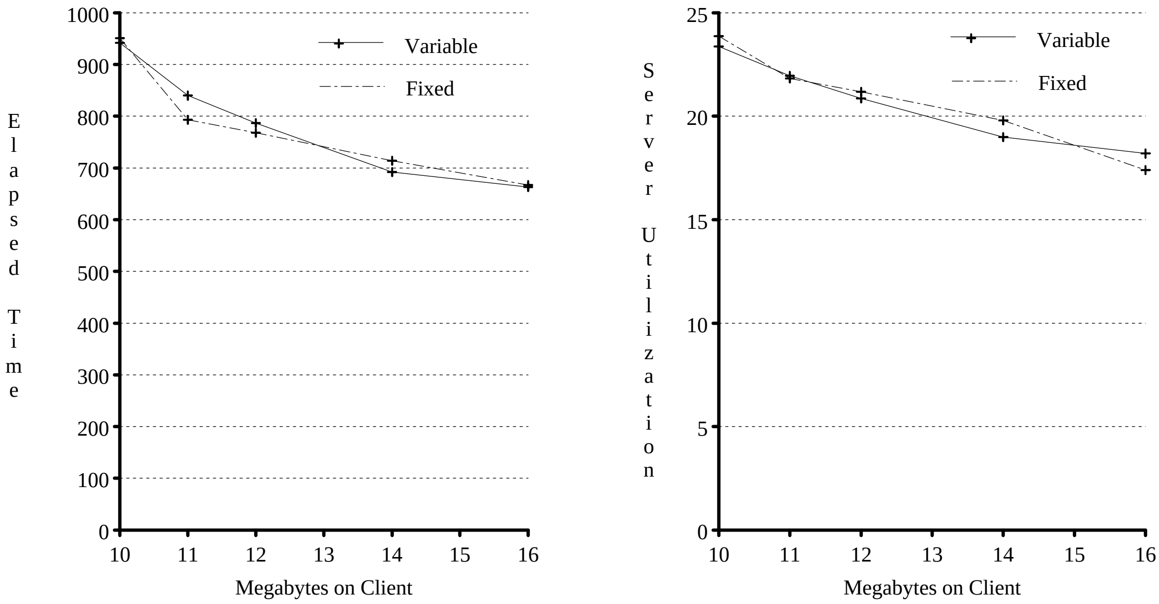

Figure 3. Elapsed time and server utilization for the edit-compile-debug benchmark with a variable-sized cache and with the smallest fixed-size cache as a function of physical memory size. In both graphs the X-axis is the amount of cache. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

5.1. The ECD Benchmark

For the ECD benchmark, variable-size cache mechanisms work quite well. Figure 3 shows that, in terms of elapsed time and server utilization, the variable-size and fixed-size cache mechanisms provide nearly identical performance. The reason why the performance is similar is demonstrated in Figure 4, which gives the amount of network traffic. The variable-size cache gives consistently fewer file system bytes transferred than a fixed-size cache, and the fixed-size cache gives fewer virtual memory bytes transferred. However, in terms of total bytes transferred, the variable-size cache is slightly better than the best fixed-size cache. Thus, the poorer virtual memory performance for the variable-size cache is more than offset by the much better file system performance.

The edit-compile-debug benchmark shifts between file- and virtual-memory-intensive programs. This requires that there be constant shifts in the allocation of physical memory between the file system and the virtual memory system (see Table 2). The minimum and maximum cache size columns from Table 2 show that the file cache varied widely in size during the life of the benchmark, going from the minimum possible size (0.25 Mbytes) up to over half the amount of physical memory available. As the amount of physical memory increased, the maximum size of the cache increased as well; the variable-size cache mechanism allowed the file system to take advantage of the extra physical memory.

Table 2 also quantifies the negotiation between the virtual memory system and the file system. As the amount of physical memory increased, the number of times that the

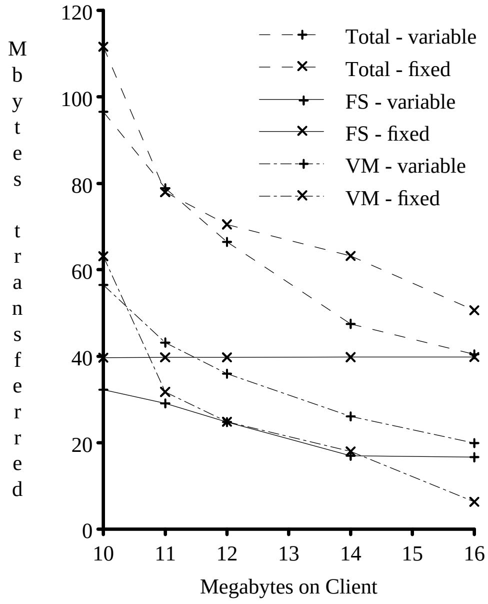

Figure 4. This graph gives the number of Mbytes transferred across the network with variable-size and smallest-fixed-size caches and 16 Mbytes of memory on the client. The X-axis is the amount of cache and the Y-axis is the number of Mbytes transferred. There are two lines for file, virtual memory and total bytes transferred: one for variable-size caches and one for fixed-size caches. The total bytes transferred lines includes file and virtual memory data bytes plus bytes from packet headers and control packets.

Client Mem (Mbytes)	Min Cache Size (Mbytes)	Max Cache Size (Mbytes)	FS Asks VM		VM Asks FS	
			Num	Satisfied	Num	Satisfied
10	0.25	5.6	8125	1810	2942	1846
11	0.25	6.4	7105	1889	2610	1967
12	0.25	6.9	5840	1964	2555	2075
14	0.25	8.7	4012	1957	2669	2162
16	0.34	8.8	3652	1937	2629	2229

Table 2. Traffic between the virtual memory system and the file system. The first column gives the amount of physical memory available on the client. The second and third columns give the minimum and maximum file cache sizes during the benchmark. The fourth and fifth columns are the number of times that the file system asked the virtual memory system for the access time of its oldest page and the number of times that it was able to get a page from the virtual memory system. The sixth and seventh columns are the same as the previous two, except that they are the number of times the virtual memory system asked the file system for memory.

file system attempted to get memory from the virtual memory system dropped dramatically; however, the number of times that the file system was successful in stealing a page from the virtual memory system remained fairly constant across all memory sizes. In contrast, the number of requests for memory made by the virtual memory system to the file system remained reasonably constant for all memory sizes, but the virtual memory system was more successful in taking pages from the file system as the amount of memory increased.

Table 2 suggests that the virtual memory system is much less elastic in its needs than the file system, at least for this benchmark; I hypothesize that this is true in general. The low success rate that the file system has when asking the virtual memory system for memory implies that the pages in the virtual memory system are being more actively used than those in the file system. Thus, the virtual memory system has fairly strict memory needs regardless of the physical memory size, and it actively uses the pages that it has. On the other hand, the file system caches files after they are no longer being used so it will grow to fill the available memory. Since the file system does not actively use many of its cached pages, its pages are the best candidates for recycling.

5.2. The IFS Benchmark

Variable-size caches work poorly for the IFS benchmark (see Table 3). The performance is dependent on the length of the interactive program's sleep interval. Short sleep intervals correspond, for example, to temporary pauses in an editing session. Long sleep intervals correspond, for example, to windows that have been idle because the user was working in a different window.

Table 3 shows that the interactive response time has a high variance: sometimes it is instantaneous and other times it takes up to 22 seconds. This corresponds to a user typing a key stroke and waiting 22 seconds for a response from the program. The response time gets worse as the sleep interval is increased. Longer sleep intervals allow the sort program to steal more memory which causes the interactive program to wait for pages to get faulted in from the file server.

In addition to producing poor interactive response, the use of variable-size caches also degrades the performance of the *sort* benchmark. The benchmark takes up to 72% longer to execute than the standalone case, and 50% longer than when a small fixed-size cache is used. The performance degrades because the CPU is busy trying to fault in pages for the interactive benchmark; if the interactive benchmark is memory resident,

Sleep	Response Time			Sort Time		Page	Page	Cache Size	
Interval	Min	Max	Avg	Time	Deg	Ins	Outs	Min	Max
1	0.0	4.7	0.1	79.6	33%	173	456	152	784
5	0.0	4.5	0.8	83.8	40%	437	509	157	842
10	1.9	13.3	5.9	103.4	72%	1605	1177	146	1226
30	12.5	22.0	15.8	96.3	61%	1250	983	141	2533

Table 3. Results for the IFS benchmark with variable-size caches. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 4 give the minimum, maximum and average number of seconds it took the interactive benchmark to touch all of its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds it took to execute the *sort* benchmark, and the amount of degradation relative to the standalone case which took 60 seconds. Column 7 is the number of pages read in from swap files and Column 8 is the number of pages written to swap files. Columns 9 and 10 give the minimum and maximum amount of memory in the cache in Kbytes.

then it utilizes very little of the CPU.

6. Biasing Against the File System

The results of the IFS and ECD benchmarks demonstrate that virtual memory performance is the most important factor in determining overall system performance. This was shown with the ECD benchmark where performance degraded as the size of fixed-size file cache was increased; fortunately the variable-size cache mechanism worked well in this case and was able to adjust the size of the cache effectively. Unfortunately, the variable-size cache mechanism did not work well for the IFS benchmark. These results indicate that it may not be practical to treat virtual memory and file data equally in a variable-size cache mechanism. This section evaluates the effect of giving virtual-memory pages preference over file pages.

6.1. Implementation

Since the Sprite mechanism treats virtual memory and file data separately it is quite easy to implement a scheme that biases against the file system. The method used in Sprite involves adding a fixed number of seconds to the reference time of each virtual memory page. This makes each virtual memory page appear to have been referenced more recently than it actually was. For example, if 5 minutes is added to the reference time of each virtual memory page, then the file system will not be able to take any page from the virtual memory system that has been referenced within 5 minutes of the oldest file system page.

6.2. ECD Benchmark

Penalizing the file system has little or no effect on the performance of the ECD benchmark. Figure 5 shows that, regardless of the penalty, the elapsed time and server utilization are about the same. Figure 6 shows why the penalty has no effect. As the penalty is made larger, the virtual memory performance gets better and the file system performance worse. The result is that overall performance is about the same regardless of the penalty.

6.3. IFS Benchmark

Penalizing the file system is very effective in improving the performance of the IFS benchmark. Table 4 shows that the interactive response is excellent when the file system is penalized. The 120-second penalty prevents the file system from taking any memory away from the virtual memory system. Thus the response time is the same regardless of the amount of time that the interactive program pauses between successive touching of its memory.

Surprisingly, the file system penalty actually improves the execution time of the *sort* benchmark relative to without the penalty (see Tables 3 and 4). When the file system is penalized, *sort* takes only 25% longer than the best case. This degradation is nearly identical to the degradation shown in [NWO88] when *sort* was run using only a small cache.

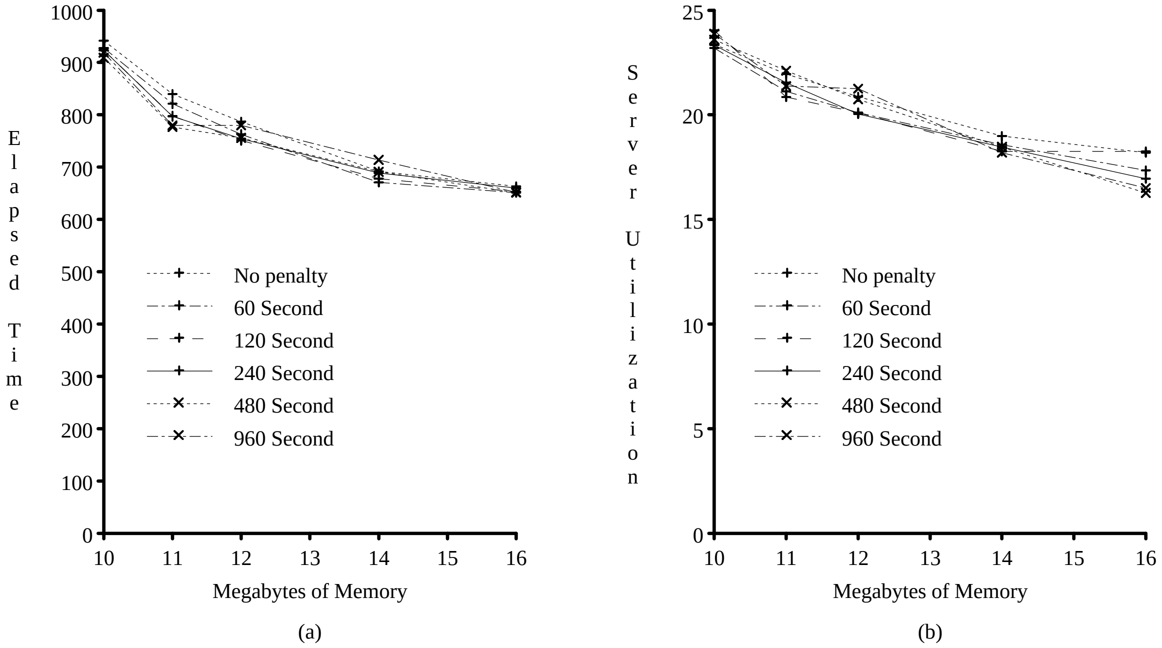

Figure 5. Elapsed time and server utilization with various penalties as a function of client physical memory size. In both graphs the X-axis is client memory size. In graph (a) the Y-axis is the number of seconds to execute the benchmark and in graph (b) the Y-axis is the percent of the server's CPU that was utilized while the client was executing the benchmark.

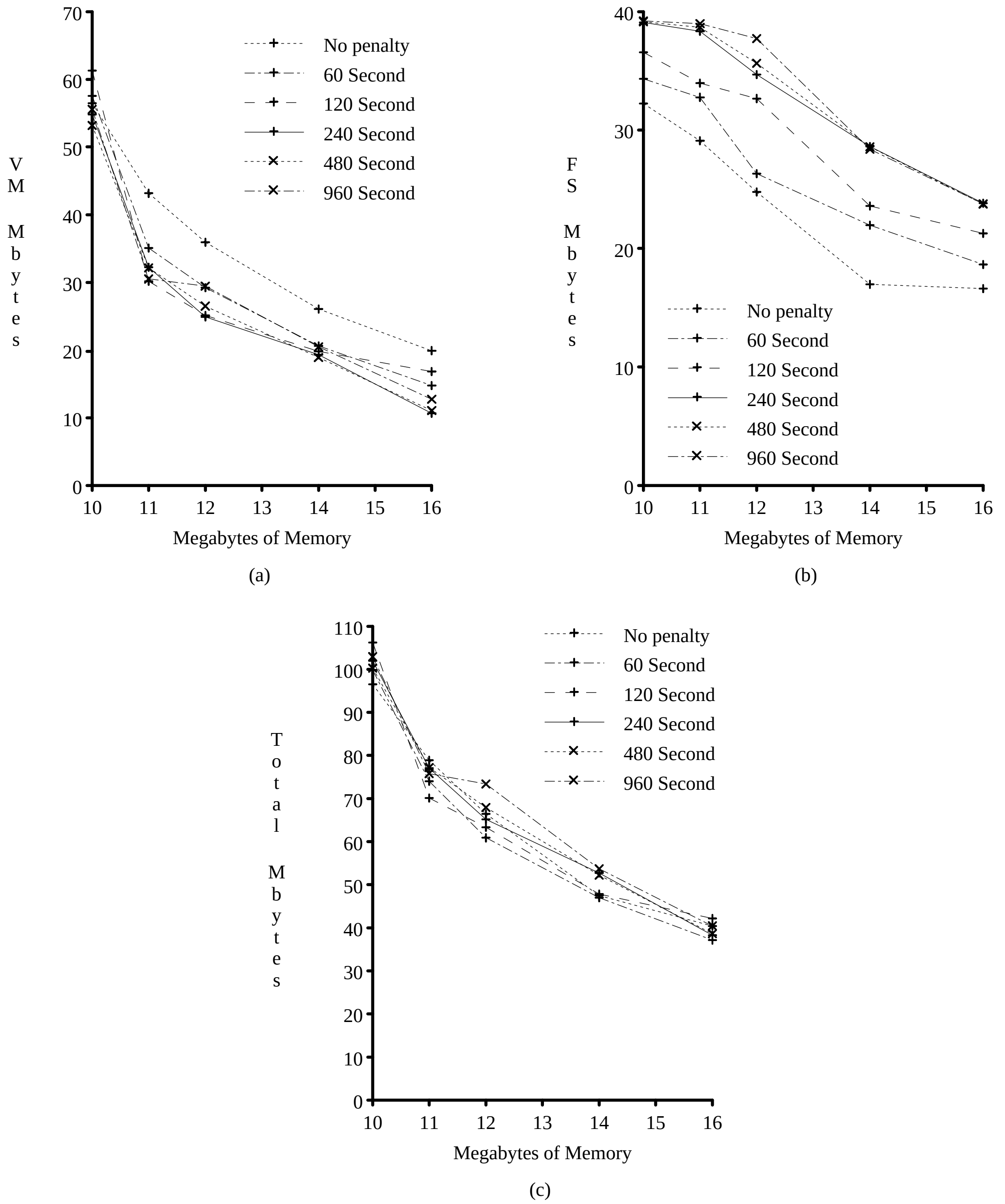

Figure 6. These graphs gives the number of Mbytes transferred across the network with various penalties as a function of client memory size. In all three graphs the X-axis is the total amount of client memory and the Y-axis is the total number of Mbytes transferred during the benchmark. Graph (a) is virtual memory traffic, (b) is file system traffic and (c) is total network traffic which includes virtual memory traffic, file system traffic and packet headers and control packets.

Sleep Interval	Response Time Min	Response Time Max	Response Time Avg	Sort Time Time	Sort Time Deg	Page Ins	Page Outs	Cache Size Min	Cache Size Max
1	0.0	0.4	0.03	74.8	25%	1	4	64	178
5	0.0	0.1	0.02	72.8	21%	0	0	64	168
10	0.0	0.1	0.01	72.1	20%	0	0	64	168
30	0.0	0.1	0.03	74.0	23%	0	0	64	168

Table 4. Results for the IFS benchmark with variable-size caches when the file system is penalized 120 seconds. Each data point is the average of the results from three runs of the benchmark. The first column gives the number of seconds that the interactive benchmark slept before touching all of its memory. Columns 2 through 4 give the minimum, maximum and average number of seconds it took the interactive benchmark to touch all of its memory when it awoke from its sleep. Columns 5 and 6 give the total number of seconds it took to execute the *sort* benchmark, and the amount of degradation relative to the standalone case which took 60 seconds. Column 7 is the number of pages read in from swap files and Column 8 is the number of pages written to swap files. Columns 9 and 10 give the minimum and maximum amount of memory in the cache in Kbytes.

6.4. How Much of a Penalty?

The results of the benchmarks in this section show that penalizing the file system can improve interactive response without degrading overall system performance. In some cases, it can even make the performance of both file- and virtual-memory intensive programs better. However, it is not clear what the optimal penalty should be. The penalty should be large enough so that idle user programs that will be used in the near future will not be removed from memory, but not so large that the performance of the file system is degraded unnecessarily. The best value for the penalty will depend on the behavior of the users of the system. In Sprite we normally set the penalty to 20 minutes. This means that an interactive program's pages will not be reclaimed by the file cache until the program has been idle for 20 minutes.

7. Conclusions

Different size file caches are required for different program mixes. As a result variable-size cache mechanisms are required to provide good performance for all types of programs. Unfortunately, standard variable-size cache mechanisms that treat virtual-memory and file data equally are not good enough. In order to ensure good interactive response to users, virtual memory data accessed in the last several minutes must be kept memory resident if possible - even if this requires removing more recently accessed file data. Basically, the file cache should be limited to those pages that are not required by the virtual memory system.

Fortunately, the Sprite variable-size cache mechanism has the ability to favor virtual memory pages. This is easy in Sprite because the virtual-memory system and the file system are kept separate. We use the Sprite file system penalty mechanism as we do our day to day work on Sprite. Since we began penalizing the file system, we have noticed that a file intensive program is no longer capable of ruining interactive response. It is my advice to implementors of other variable-size caching mechanisms (e.g. mapped files) that they include in their implementation the ability to favor virtual-memory pages. This will make interactive users happier and may even improve the performance of file intensive programs.

8. Acknowledgements

I want to thank the other Sprite developers: John Ousterhout, Brent Welch, Fred Douglis, and Andrew Cherenson. Without their efforts Sprite would not exist. John Ousterhout, Jeff Mogul, and Anita Borg provided numerous helpful comments that improved the presentation of this paper.

The work described here was done as part of my PhD research at the University of California at Berkeley. It was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00039-84-C-0107.

9. Bibliography

[BCD72] A. Bensoussan, C. T. Clingen and R. C. Daley, ‘‘The MULTICS Virtual Memory: Concepts and Design’’, *Comm. of the ACM 15*, 5 (May 1972).

[BBM72] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy and R. S. Tomlinson, ‘‘TENEX, a Paged Time Sharing System for the PDP-10’’, *Comm. of the ACM 15*, 3 (Mar. 1972), 1135-143.

[DaD68] R. C. Daley and J. B. Dennis, ‘‘Virtual Memory, Processes and Sharing in MULTICS’’, *Comm. of the ACM 11*, 5 (May 1968), 306-312.

[LLH85] P. Leach, P. Levine, J. Hamilton and B. Stumpf, ‘‘The File System of an Integrated Local Network’’, *Proc. of the 1985 ACM Computer Science Conference*, Mar. 1985, 309-324.

[Lea83] P. J. Leach, et al., ‘‘The Architecture of an Integrated Local Network’’, *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (Nov. 1983), 842-857.

[Mur72] D. L. Murphy, ‘‘Storage organization and management in TENEX’’, *Proceedings AFIPS Fall Joint Computer Conference 15*, 3 (1972), 23-32.

[Nel86] M. N. Nelson, ‘‘The Sprite Virtual Memory System’’, Technical Report UCB/Computer Science Dpt. 86/301, University of California, Berkeley, June 1986.

[Nel88] M. N. Nelson, *Physical Memory Management in a Network Operating System*, Phd Thesis, University of California at Berkeley, 1988.

[NWO88] M. N. Nelson, B. B. Welch and J. K. Ousterhout, ‘‘Caching in the Sprite Network File System’’, *Trans. Computer Systems 6*, 1 (Feb. 1988), 134-154.

[OCD88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson and B. B. Welch, ‘‘The Sprite Network Operating System’’, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.

[RaR81] R. F. Rashid and G. G. Robertson, ‘‘Accent: A communication oriented network operating system kernel’’, *Proceedings of the 8th Symposium on Operating Systems Principles*, 1981, 164-175.

[RaF86] R. F. Rashid and R. Fitzgerald, ‘‘The Integration of Virtual Memory Management and Interprocess Communication in Accent’’, *Trans. Computer Systems 4*, 2 (May 1986), 147-177.

[Ras87] R. Rashid, et al., ‘‘Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures’’, *Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Oct. 1987, 31-39.

[Red80] D. D. Redell, et al., ‘‘Pilot: An Operating System for a Personal Computer’’, *Communications of the ACM 23*, 2 (Feb. 1980), 81-92.