Measurements of a Distributed File System

Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

We analyzed the user-level file access patterns and caching behavior of the Sprite distributed file system. The first part of our analysis repeated a study done in 1985 of the BSD UNIX file system. We found that file throughput has increased by a factor of 20 to an average of 8 Kbytes per second per active user over 10-minute intervals, and that the use of process migration for load sharing increased burst rates by another factor of six. Also, many more very large (multi-megabyte) files are in use today than in 1985. The second part of our analysis measured the behavior of Sprite's main-memory file caches. Client-level caches average about 7 Mbytes in size (about one-quarter to one-third of main memory) and filter out about 50% of the traffic between clients and servers. 35% of the remaining server traffic is caused by paging, even on workstations with large memories. We found that client cache consistency is needed to prevent stale data errors, but that it is not invoked often enough to degrade overall system performance.

1. Introduction

In 1985 a group of researchers at the University of California at Berkeley performed a trace-driven analysis of the UNIX 4.2 BSD file system [11]. That study, which we call ''the BSD study,'' showed that average file access rates were only a few hundred bytes per second per user for engineering and office applications, and that many files had lifetimes of only a few seconds. It also reinforced commonly-held beliefs that file accesses tend to be sequential, and that most file accesses are to short files but the majority of bytes transferred belong to long files. Lastly, it used simulations to predict that main-memory file caches of a few megabytes could substantially reduce disk I/O (and server traffic in a networked environment). The results of this study have been used to justify several network file system designs over the last six years.

In this paper we repeat the analysis of the BSD study and report additional measurements of file caching in a distributed file system. Two factors motivated us to make the new measurements. First, computing environments have changed dramatically over the last six years, from relatively slow time-shared machines (VAX-11/780s in the BSD study) to today's much faster personal workstations. Second, several network-oriented operating systems and file systems have been developed during the last decade, e.g. AFS [4], Amoeba [7], Echo [3], Locus [14], NFS [16], Sprite [9], and V [1]; they provide transparent network file systems and, in some cases, the ability for a single user to harness many workstations to work on a single task. Given these changes in computers and the way they are used, we hoped to learn how file system access patterns have changed, and what the important factors are in designing file systems for the future.

We made our measurements on a collection of about 40 10-MIPS workstations all running the Sprite operating system [9, 12]. Four of the workstations served as file servers, and the rest were diskless clients. Our results are presented in two groups. The first group of results parallels the analysis of the BSD study. We found that file throughput per user has increased substantially (by at least a factor of 20) and has also become more bursty. Our measurements agree with the BSD study that the vast majority of file accesses are to small files; however, large files have become an order of magnitude larger, so that they account for an increasing fraction of bytes transferred. Many of the changes in our measurements can be explained by these large files. In most other respects our measurements match those of the BSD study: file accesses are largely sequential, files are typically open for only a fraction of a second, and file lifetimes are short.

Our second set of results analyzes the main-memory file caches in the Sprite system. Sprite's file caches change size dynamically in response to the needs of the file and virtual memory systems; we found substantial cache size variations over time on clients that had an average cache size of about 7 Mbytes out of an average of 24 Mbytes of main memory. About 60% of all data bytes read by

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591, and an IBM Graduate Fellowship Award.

This paper will appear in the *Proceedings of the 13th ACM Symposium on Operating Systems Principles*.

applications are retrieved from client caches without contacting file servers. Sprite's 30-second delayed-write policy allows about 10% of newly-written bytes to be deleted or overwritten without being written back from the client cache to the server.

Sprite guarantees the consistency of data cached on different clients. We found that many users would be affected if Sprite's consistency guarantees were weaker, but write-sharing occurs infrequently enough that the overheads of implementing consistency have little impact on average system performance. We compared Sprite's consistency implementation with two other approaches and found that even the best approach, a token-based mechanism, does not significantly reduce the consistency overheads for our workload.

Sprite allows users to take advantage of many workstations simultaneously by migrating processes onto idle machines. Process migration increased the burst rates of file throughput by a factor of six in comparison to overall file throughput. Fortunately, we found that process migration does not reduce the effectiveness of file caches. Migrated processes actually had higher cache hit ratios than non-migrated processes, and process migration also had little impact on the cache consistency mechanism.

The rest of the paper is structured as follows: Section 2 describes the system that was measured and its workload, and Section 3 describes how we collected data. Section 4 contains the measurements that repeat the BSD study and Section 5 presents our measurements of Sprite caches. Section 6 is a summary.

2. The System Under Study

The system we measured is a cluster of about 40 SPARCstation 1, Sun 3, DECstation 3100, and DECstation 5000 workstations, all diskless and most with 24 to 32 Mbytes of memory. Each of these workstations has about ten times the compute power of the VAX timesharing systems used in the BSD study. The cluster is served by four file servers, with most traffic handled by a single Sun 4 file server with 128 Mbytes of main memory. About 30 users do all of their day-to-day computing on this cluster and another 40 people use the system occasionally.

The users fall into four groups of roughly the same size: operating system researchers, architecture researchers working on the design and simulation of new I/O subsystems, a group of students and faculty working on VLSI circuit design and parallel processing, and a collection of miscellaneous other people including administrators and graphics researchers. The makeup of this user community is similar to the user community measured in the BSD study. Common applications are interactive editors of various types, program development and debugging, electronic mail, document production, and simulation.

All of the workstations in the cluster run the Sprite network operating system. The Sprite kernel's facilities are almost identical to those of UNIX, and most of the applications running on the cluster are standard UNIX applications. Sprite has two features that made it a particularly interesting candidate for measurement: its network file system [17] and process migration [2]. Sprite's network file system provides a single-system image: there is a single shared file hierarchy with no local disks. The file system uses large file caches in the main memories of both clients and servers, and it ensures the consistency of data in the caches even in the event of concurrent write accesses. Thus the Sprite file system encourages users to share files.

The second interesting feature of Sprite is its process migration mechanism, which makes it easy for users to offload jobs to idle machines in the cluster [2]. The most common use of process migration is through *pmake*, a reimplementation of the *make* utility that uses migration to generate multiple targets in parallel. Pmake is used for all compilations in the cluster and also for simulations and other tasks. Long-term measurements show that 10% to 30% of all user-level cycles are executed by migrated processes.

Using Sprite had the disadvantage of limiting us to a single Sprite cluster for measurements; this makes our results vulnerable to anomalies in that cluster's workload. However, the danger of anomalies was reduced by the presence of different user groups in the cluster. Moreover, Sprite allowed us to examine some interesting features not present in the other heavily-used systems currently available to us for measurement. For example, Sprite permits much larger file caches than many commercial systems, and we were curious to see how they behaved. Sprite has ''perfect'' cache consistency; by measuring Sprite we could see how often users take advantage of this consistency. Lastly, Sprite's process migration mechanism makes it unusually easy for a single user to harness several different workstations to work on the same problem; we wished to see if this increased the frequency of file sharing or affected file system behavior in other ways.

3. Collecting the Data

To gather the measurements for this paper, we used two methods: tracing and kernel counters. To collect the traces we instrumented the Sprite kernels to log information about various events in the file system, such as opens, closes, and file deletions. The contents of the log were periodically passed to a user-level process that stored the data in a series of trace files. The results in Section 4 were produced by scanning the trace files.

The traces recorded file activity at the level of kernel calls, not at the level of network requests or disk operations. As in the BSD study, the traces contained enough information to indicate which portions of which files were read or written on behalf of which users, and at approximately what times. The traces did not record individual read and write kernel calls but they recorded the file access position before and after reposition operations (`lseek` kernel calls) and during opens and closes. This provided enough information to deduce the exact range of bytes

accessed, but it introduced a small amount of uncertainty in times (actual reads and writes could have occurred at any time between the surrounding open/close/reposition events). Ousterhout et al. have a more complete discussion of the tracing approach [11].

One of the most difficult tasks in tracing a network of workstations is coordinating the traces from many machines. Our task was greatly simplified because most of the information we wished to trace was available on the Sprite file servers. Several key file system operations, such as file opens, file closes, and directory accesses, require communication between the client and the server. Thus a system-wide trace of all such events can be made by gathering information on only the file servers. In the one case where information wasn't available on the server (repositions), we modified the client kernels to make an extra request to the server so that the operation could be logged. This meant that we only had to gather traces on four machines instead of forty. Each server's trace data was recorded in a different set of trace files, but the traces included enough timing information to merge the traces from the different servers into a single ordered list of records. The code that merged the traces also removed all records related to writing the trace files themselves and all records related to the nightly tape backup.

In this paper we present the results of tracing eight 24-hour periods. Some overall statistics about the traces are presented in Table 1. The traces were collected in pairs, i.e. we collected data for four 48-hour periods which were then divided into eight 24-hour periods. No attempt was made to ensure that the workloads were consistent across the traces. Users were free to run whatever programs they wished. During the third and fourth traces two users were working on class projects that used large input and output files. One user was using a simulator that read input files that averaged 20 Mbytes in size. The other user was running a cache simulation that produced a 10 Mbyte file that was subsequently postprocessed and deleted. Both users ran their simulators repeatedly throughout the two traces. As a result, the workloads for these two traces are very different from the rest of the traces, causing them to stand out in some of our measurements. They do, however, reinforce our assertion that the size of ''large'' files is increasing.

Our second method for collecting data was to gather statistics in the kernels of each of the workstations in the cluster. We used approximately 50 counters that recorded statistics about cache traffic, ages of blocks in the cache, the size of the cache, etc. A user-level process read the counters at regular intervals and recorded their values in a file. These counters were recorded 24 hours a day across a two-week period. The files for different clients were then post-processed to yield the statistics in Section 5.

4. BSD Study Revisited

We used our trace data to repeat all of the measurements of file system usage in the original BSD study. As in the BSD study, we report the results under three general categories: user activity (how much the file system is used), access patterns (sequentiality, dynamic file sizes, and open times), and file lifetimes. The results only include accesses to files and directories by user processes. Accesses to virtual memory swap files and I/O devices are excluded. Our general conclusions are that the basic access patterns are similar today to what they were six years ago,

Trace	1	2	3	4	5	6	7	8
Date	1/24/91	1/25/91	5/10/91	5/11/91	5/14/91	5/15/91	6/26/91	6/27/91
Trace duration (hours)	24	23.8	24	24	24	24	24	24
Different users	44	48	47	33	48	50	46	36
Users of migration	6	6	11	8	7	11	9	9
Mbytes read from files	1282	1608	13064	17754	822	1489	1292	2320
Mbytes written to files	493	614	4892	1383	476	610	506	626
Mbytes read from directories	30	67	25	18	15	17	14	15
Open events	149254	224102	149898	115929	124508	184863	133846	275140
Close events	151306	225590	151693	117536	126222	186631	136144	278388
Reposition events	122089	221372	127879	113796	176733	104579	103617	102114
Truncate events	5500	4883	6036	3501	6201	5860	4198	7604
Delete events	20278	30691	24111	16936	24495	28839	15762	20907
Shared Read events	21985	54351	39849	3244	832	2823	3456	9663
Shared Write events	443	1129	45043	3111	322	2499	1452	2224

Table 1. Overall trace statistics. Information about the traces is shown, such as dates, number of users, data rates, and number of events. *Mbytes read from directories* is the amount of data read from directories by user-level processes e.g. listing the contents of a directory. *Open* is the number of file or directory opens, and *Close* is the number of closes. *Reposition* events occur when the current file offset was changed via an `lseek` kernel call. These usually happen when the contents of a file are accessed randomly. *Truncate* events occur when a file is truncated to zero length. *Delete* events occur when a file or directory is removed. *Shared read* and *Shared write* events are read and write events on files undergoing write-sharing; these events were used for the simulations in Section 5.

		All Users	Users with Migrated Processes	BSD Study
10-minute intervals	Maximum number of active users	27	5	31
	Average number of active users	9.1 (5.1)	0.91 (0.98)	12.6 (6.7)
	Average throughput per active user (Kbytes/second)	8.0 (36)	50.7 (96)	0.40 (0.4)
	Peak throughput for a user in an interval (Kbytes/second)	458	458	NA
	Peak total throughput in an interval (Kbytes/second)	681	616	NA
10-second intervals	Maximum number of active users	12	4	NA
	Average number of active users	1.6 (1.5)	0.14 (0.4)	2.5 (1.5)
	Average throughput per active user (Kbytes/second)	47.0 (268)	316 (808)	1.5 (7.0)
	Peak throughput for a user in an interval (Kbytes/second)	9871	9871	NA
	Peak total throughput in an interval (Kbytes/second)	9977	9871	NA

Table 2. User activity. Each trace was divided into intervals of 10-minutes and 10-seconds, and the number of active users (those for whom a trace record appeared during the interval) and average file throughput were computed for each interval. The *All Users* column contains values for all active users; the *Users with Migrated Processes* column considers only users with active migrated processes during the interval. The *BSD Study* column reports the average numbers from [11] for comparison. Measurements that weren't available are labeled *NA*. The numbers in parentheses are standard deviations. For the average number of active users they are the standard deviations of each interval from the long-term average across all intervals. For the average throughput measurements they are the standard deviations of each user-interval from the long-term average across all user-intervals.

File Usage	Accesses (%)		Bytes (%)		Type of Transfer	Accesses (%)		Bytes (%)	
Read-only	88	(82-94)	80	(63-93)	Whole-file	78	(64-91)	89	(46-96)
					Other sequential	19	(7-33)	5	(2-29)
					Random	3	(1-5)	7	(2-37)
Write-only	11	(6-17)	19	(7-36)	Whole-file	67	(50-79)	69	(56-76)
					Other sequential	29	(18-47)	19	(4-27)
					Random	4	(2-8)	11	(4-41)
Read/write	1	(0-1)	1	(0-3)	Whole-file	0	(0-0)	0	(0-0)
					Other sequential	0	(0-0)	0	(0-0)
					Random	100	(100-100)	100	(100-100)

Table 3. File access patterns. The *Accesses* column indicates the percentages of all accesses that were read-only, write-only, or read/write. An access consists of opening a file, reading and/or writing it, and then closing the file. These access types reflect actual file usage, not the mode in which the file was opened. An access is considered read/write only if the file was both read and written during the access. The *Bytes* column shows the percentage of bytes transferred in each type of access. The last two columns divide each kind of access into three groups based on degree of sequentiality: *Whole-file* refers to accesses where the entire file was transferred sequentially from start to finish; *Other Sequential* refers to accesses that were not whole-file but where a single sequential run of data was transferred between open and close; and *Random* includes all other accesses. The numbers in parentheses are the minimum and maximum values measured for the individual traces. For example, the upper-right entry indicates that, of all bytes transferred in read-only accesses, 89% were transferred in whole-file reads; the minimum value measured for a trace was 46%, while the maximum was 96%.

but overall activity levels are substantially higher (with tremendous bursts of activity caused by process migration), and that the largest files in use are much bigger than before.

4.1. User Activity

Table 2 reports on overall file system activity during the trace. The traces are divided into 10-minute and 10-second intervals, and the number of active users and the throughput are averaged across those intervals. A user is considered active in an interval if there were any trace records for the user in the interval. The 10-minute averages represent steady-state behavior, while the 10-second averages show what can happen during bursts of activity. Table 2 shows that both of these throughput averages have increased since the BSD study. Average throughput over 10-minute intervals increased by a factor of 20, while average throughput over 10-second intervals increased by more than a factor of 30.

Although the increase in throughput is significant, it is substantially less than the increase in available processing power per user: in 1985, 20 to 50 users shared a single 1-MIPS timeshared VAX, but in 1991 each user had a personal 10-MIPS workstation. The net result is an increase in the computing power per user by a factor of 200 to 500, yet the throughput requirements only increased by about a factor of 20 to 30. On the other hand, the burstiness of the throughput requirements has increased, as indicated by the standard deviations. Users seem to have used their additional computing resources to decrease the response time to access data more than they have used it to increase the overall amount of data that they use.

Table 2 also shows the impact of process migration on throughput. In the second column of the table, a user is considered to be active in an interval only if a migrated process invoked file activity on the user's behalf during the interval, and only file activity from migrated processes is included in the throughput calculations. Migration produced activity levels about six or seven times higher than the overall average; this is partly because migration is used for activities that are particularly I/O-intensive (such as compilations), and partly because migration marshals the resources of several workstations in the network on a single user's behalf.

The increasing burstiness of file I/O makes it particularly important to perform client-level caching in order to offload the networks and file servers. For example, in one 10-second interval a single user averaged more than 9.6 Mbytes/second of file throughput; without client-level caching this would not have been possible, since the data rate exceeds the raw bandwidth of our Ethernet network by a factor of ten.

4.2. File Access Patterns

The BSD study concluded that most accesses are sequential; our measurements, as shown in Table 3, reaffirm this conclusion. By sequential access, we mean that a file was opened, a series of bytes was transferred in order, then the file was closed. An access is still considered sequential even if the entire file was not transferred because the beginning or end was skipped. If anything, the sequentiality of access has increased over the last six years: our traces show that about 78% of all read-only accesses were sequential whole-file transfers, versus about 70% in the BSD study, and that more than 90% of all data was transferred sequentially, versus less than 70% in the BSD study.

A *sequential run* is a portion of a file read or written sequentially (i.e. a series of data transfers bounded at the start by an open or reposition operation and at the end by a close or another reposition operation). The top graph in Figure 1 shows that most sequential runs are short (about 80% transfer less than 10 Kbytes each). This is necessarily true since most accessed files are short: there simply aren't very many opportunities for long sequential runs. Long sequential runs should not be ignored, however, because they transfer most of the bytes, as shown in the bottom graph of Figure 1. Large transfers play a much greater role in our system than in the original BSD study. For example, in the BSD study, only 10% of all bytes were transferred in sequential runs longer than 100 Kbytes; in our traces at least 10% of all bytes were transferred in runs longer than one Mbyte. In other words, the longest runs have increased in size by at least a factor of ten. There is a wide variation in run length between traces, however. For most of the traces between 55% and 80% of the data was transferred in runs up to one Mbyte in length. For one trace these runs represented 90% of the bytes, and in two traces these runs represented less than 10%. During the last two traces a large number of simulations were run on input files that were about 20 Mbytes in size. This caused the run length distribution to be heavily skewed. Clearly the system workload has a significant impact on run length.

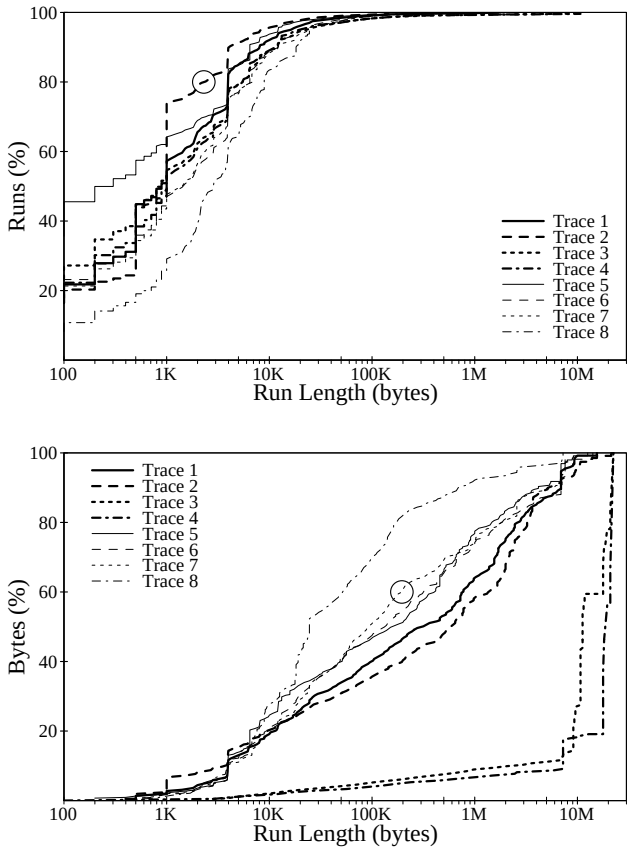

Figure 1. Sequential run length. Cumulative distributions of the lengths of sequential runs. The length of a sequential run is the number of bytes transferred before closing the file or repositioning. The top graph is weighted by the number of runs; for example, the circle indicates that for Trace 2 80% of all sequential runs are less than 2300 bytes in length. The bottom graph is weighted by the number of bytes transferred; for example, the circle indicates that for Trace 7 60% of all bytes were transferred in sequential runs of 191 Kbytes or less.

We were initially concerned that the trend towards larger files might be an artifact of our particular environment. In particular, the kernel development group commonly works with kernel binaries ranging in size from two to ten Mbytes. To measure the effect of the kernel binaries, we reprocessed the traces while ignoring all accesses from the kernel development group. The results were very similar to those obtained with the kernel group included. For example, the parallel processing researchers used simulation data files ranging up to 20 Mbytes in size. Our conclusion is that the increase in file size is not an artifact of our particular environment but a general trend that follows from increases in computing power and storage space.

The existence of very long sequential runs implies the existence of very large files. Figure 2 shows that this implication is correct. Most of the accesses are to short files, but most bytes are transferred to or from large files. Once again, the large files in our measurements are much bigger than the large files in the BSD study: the top 20% of all files (by bytes transferred) are more than an order of magnitude larger than the top 20% in the BSD study. The computational resources now available to users allow them to store and manipulate much larger files than they could previously, and they use these files often enough to have a significant effect on system behavior.

The last measurement of access patterns is displayed in Figure 3; it shows that about 75% of the files are open less than one-quarter second. In comparison, the BSD study found that 75% of files were open less than one-half second. This shortening of open times is easy to explain: machines have gotten much faster while most files remain short, so it doesn't take as long to process most files as it used to. The speedup isn't as great as might be expected, however, since machines have gotten about ten times faster while open times have only decreased by a factor of two. This discrepancy may be caused by the use of a local file system in the BSD study, and the use of a network file system in the current study. Previous studies have shown that open/close times on a client using a network file system are a factor of four or five slower than those on a machine with a local file system [13]. This difference in open/close times could account for the relatively small change in file open times despite the large improvement in processor speed.

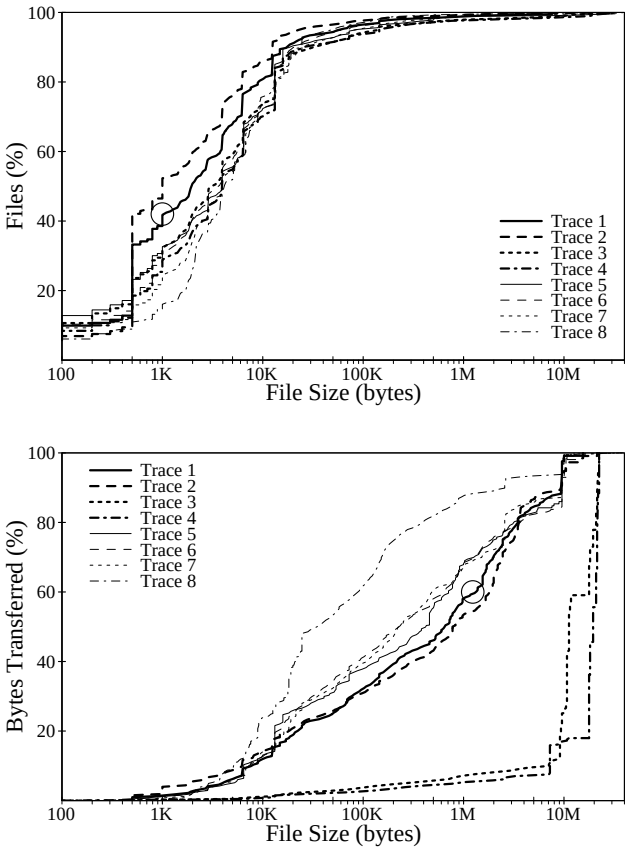

Figure 2. File size. Dynamic distribution of file sizes, measured when files were closed. The top graph is a cumulative distribution weighted by number of files, and the bottom graph is weighted by the number of bytes transferred to or from the file. For example, the circles indicate that for Trace 1 42% of all accesses were to files less than one Kbyte long, but that during the same trace 40% of all bytes were transferred to or from files of at least one Mbyte in size.

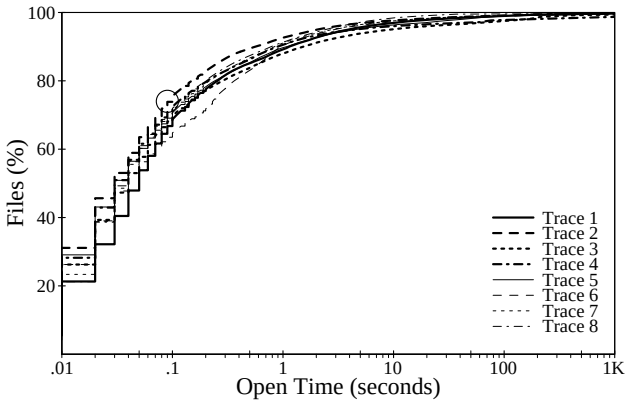

Figure 3. File open times. Cumulative distribution of the length of time files were open. For example, the circle indicates that for Trace 2 about 74% of all opens lasted less than 0.1 second.

4.3. File Lifetimes

In Figure 4 we present our measurements of file lifetimes (the interval between when a file is created and when it is deleted or truncated to zero length). File lifetimes have an important impact on the writing policy of a file cache. If most lifetimes are short then it is advantageous to buffer data in the file cache for a while before writing it to the file server; data deleted during this period will never need to be written to the server. Our measurements show that most files have very short lifetimes (between 65% and 80% live less than 30 seconds, which is the default time that Sprite delays before writing data to the server). However, the short-lived files tend to be short: only about 4 to 27% of all

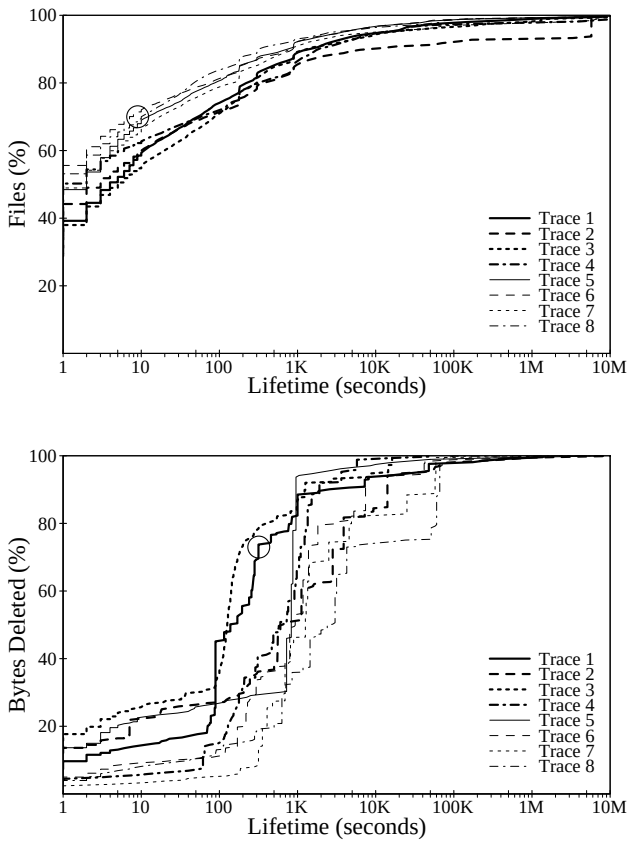

Figure 4. File lifetimes. Cumulative distribution of file lifetimes, measured when files were deleted (files that were truncated to zero length are considered deleted). The top graph is weighted by the number of files deleted, while the bottom graph is weighted by the number of bytes deleted from the file since its creation. Lifetimes are estimated from the ages of the oldest and newest bytes in the file. In the top figure, the file lifetime is the average age of the oldest and newest bytes. In the bottom figure, it is assumed that the file was written sequentially, so that the age of each byte is proportional to its offset in the file. For example, the top circle indicates that for Trace 6 70% of all files were deleted within 9 seconds of their creation, while the bottom circle indicates that for Trace 1 73% of all bytes were deleted within about 6 minutes.

new bytes are deleted or overwritten within 30 seconds. Our per-file lifetimes are similar to those of the original BSD study, but our per-byte lifetimes are a bit longer reflecting the increased importance of larger longer-lived files.

5. File Cache Measurements

One of the most important features of the Sprite system is its mechanism for caching file data in main memory [9]. The mechanism was motivated in large part by the measurements of the BSD study, and has the following features:

- File data is cached on both client and server machines on a block-by-block basis (4-Kbyte blocks).
- Servers cache both naming information and file data, whereas clients only cache file data; all naming operations, such as opens, closes, and deletes, are passed through to servers.
- Caches vary in size depending on the needs of the file system and the virtual memory system. As described in [10], the virtual memory system receives preference: a physical page used for virtual memory cannot be converted to a file cache page unless it has been unreferenced for at least 20 minutes. This 20 minute value was chosen after examining the results from several benchmarks.
- Writes are normally delayed 30 seconds. Every 5 seconds, a daemon process checks the cache for dirty data that is 30 seconds old. This data is then written to the server's cache (if it hasn't been deleted), and an additional 30 seconds later it is written to disk. Application programs may bypass the delays and request synchronous write-through with the `fsync` kernel call.
- File servers use three mechanisms to maintain consistency of the data in client caches: timestamps, recall, and cache disabling. When a client opens a file it receives the file's current timestamp from the server; the client uses this to flush any stale data from its cache. The server keeps track of the last writer for each file and *recalls* the dirty data from the writer if some other client opens the file before the writer has written the new data back to the server. If two or more client machines have a file open at a time, and at least one of them has the file open for writing (*concurrent write-sharing*), then the server disables client caching for the shared file: until the file has been completely closed again by all clients, all read and write requests are passed through to the server. The result of these three techniques is that every read operation is guaranteed to return the most up-to-date data for the file.

In this section of the paper we present measurements of the Sprite client caches in order to answer the following questions:

- How big are Sprite file caches, and how much do their sizes vary over time?
- How much of the read and write traffic is absorbed by the caches?
- How long does data live in a cache, and what causes it to leave the cache?
- How often does write-sharing occur? Is it important to guarantee consistency among file caches?
- If consistency is to be guaranteed, what is the best algorithm for doing so?

We answered the above questions by analyzing a collection of statistics gathered on the machines in the Sprite cluster over a period of two weeks. For the last two questions we also used the trace data gathered on the Sprite servers; we wrote simulators to estimate the importance of consistency and to evaluate the overheads of different approaches to consistency.

5.1. File Cache Sizes

Our first measurements recorded information about the sizes of client-level file caches. As shown in Table 4, Sprite's file caches tend to grow quite large, even though the virtual memory system receives preferential use of main memory. On file servers, the caches automatically adjust themselves to fill nearly all of memory. On the clients, the ''natural'' size of file caches appears to be about one-fourth to one-third of physical memory, or about 7 Mbytes out of an average of 24 Mbytes of physical memory on each client workstation. This is a significantly larger fraction of memory than the 10% that is allocated by many UNIX kernels. The increased use of large files that we found in Section 4.2 suggests that file caches will need to be large to be effective.

Table 4 also shows significant variations in cache size, both among machines and at different times on the same machine. On client workstations, cache sizes often varied by several hundred Kbytes over intervals of a few minutes. Although we cannot estimate the performance impact of the trading mechanism between the file caches and virtual memory, we can say that it was used frequently.

Cache size	Minimum	195 Kbytes
	Maximum	23820 Kbytes
	Average	7110 Kbytes
	Standard deviation over 15-minute intervals	5556 Kbytes
Cache size changes over 15-minute intervals	Maximum	21904 Kbytes
	Average	493 Kbytes
	Standard deviation	1037 Kbytes
Cache size changes over 60-minute intervals	Maximum	22924 Kbytes
	Average	1049 Kbytes
	Standard deviation	1716 Kbytes

Table 4. Client cache sizes. This table presents measurements of client cache sizes and how the cache sizes varied over time. The *size change* for an interval was computed as the difference between a cache's maximum and minimum sizes during the interval. We included only intervals in which either a user was active at the console or there was at least one second of user-level or migrated process CPU time. We screened out the effects of machine reboots, since a rebooted machine's cache starts at the minimum size and almost always grows immediately. The standard deviations are computed from the individual machine-intervals in comparison to the long-term averages across all intervals.

5.2. The Impact of Caching on File Traffic

The BSD study predicted that large caches would be highly effective at absorbing read traffic. For example, it predicted a miss ratio of only 10% for a 4-Mbyte cache size. In reality the Sprite caches are reasonably effective, but they did not reduce traffic to the server as much as the BSD study predicted. This section examines file traffic in more detail to understand why there is still so much server traffic.

To examine the effects of the client caches, we measured I/O traffic in three different places in the system, as illustrated in Figure 5: the traffic presented by applications to the client operating system (''raw traffic''), the behavior of the client caches, and the traffic from clients to the file server. The rest of this section discusses each of these sets of measurements in turn. Table 5 lists the relative amounts of the file and paging traffic presented by applications to the client operating system. Some of this traffic cannot be cached (''uncacheable traffic'') and is sent directly to the server. The cacheable I/O requests are given first to the client cache and only sent to the server if the cache access fails. Table 6 presents measurements of client cache effectiveness in terms of miss ratios and traffic ratios in and out of the client cache. The final table of measurements, Table 7, shows the types and relative amounts of traffic that the clients present to the file server across the network. Some of this traffic is generated by the client caches in order to satisfy cache requests, and the rest of the traffic consists of the uncacheable client application I/O requests. Overall, the ratio of the bytes transferred to the server divided by the raw traffic to the client operating system is 50%; the client caches filter out 50% of the raw traffic. These numbers do not reflect the effects of caching on the file server.

The measurements of application traffic in Table 5 show that only 20% of raw file and paging traffic is uncacheable, and most of this is paging. Other sources of non-cacheable file traffic are files undergoing write-sharing and directory reads (Sprite doesn't cache directories on clients in order to avoid consistency problems). Traffic caused by write-sharing is very low, less than 1%.

Table 6 measures the effectiveness of the file caches for the traffic that is cacheable. Read requests miss in the cache about 40% of the time. In terms of bytes, about 40% of the file bytes requested by applications must be retrieved from a file server. Although Sprite's caches perform better than the (much smaller) caches used at the time of the BSD study, the miss ratios are about four times higher than the BSD study predicted for caches as large as Sprite's. We suspect that the increased use of very large files accounts for this difference in performance. Read miss ratios vary significantly from machine to machine depending upon the workload; machines processing large files show miss ratios as high as 97%. Sprite does not prefetch cache blocks; prefetching could reduce the read miss ratio, and hence I/O latencies, but it would not reduce the read-related server

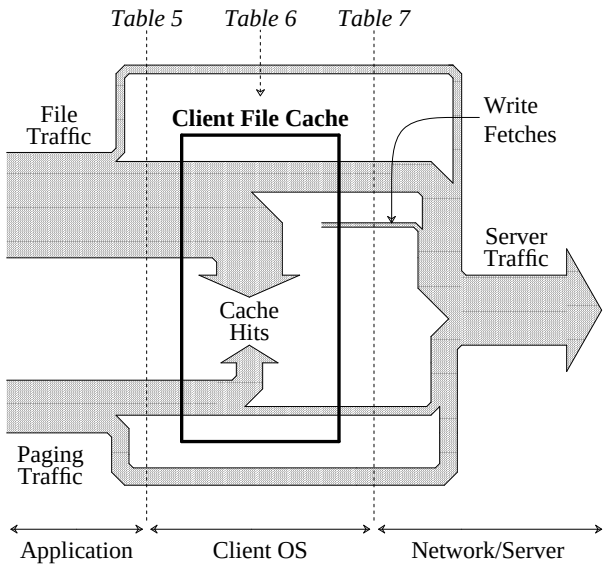

Figure 5. Data traffic flow. This figure diagrams the flow of file-related requests in Sprite clients and shows the points at which data was collected for Tables 5 to 7. Read and write traffic is combined, and the width of each path indicates the relative amount of data transferred via that path. The vertical lines indicate the measurement points for Tables 5 (raw traffic presented to the client operating system), 6 (client cache effectiveness), and 7 (traffic sent from clients to the server). *Write fetches* are read requests made by the client cache to satisfy a partial-block cache write.

Traffic Type		Bytes Read (%)		Bytes Written (%)		Total (%)
Cached	Files	52.4	(15.2)	10.5	(8.7)	62.9
	Paging	16.9	(10.8)	0.0	(0.0)	16.9
Uncached	Paging	11.3	(12.8)	6.7	(8.3)	18.0
	Shared files	0.3	(1.7)	0.2	(2.3)	0.5
	Directories	0.5	(1.0)	0.0	(0.0)	0.5
	Other	0.3	(1.6)	0.9	(5.9)	1.2
Total (%)		81.7		18.3		100.0

Table 5. Traffic sources. This table lists the sources and types of raw file traffic on Sprite clients. Each entry is a percent of all raw file traffic (*raw* means that the traffic has not yet been reduced by any caches). The right-most column gives the total for both reads and writes of each type of traffic listed. The numbers in parentheses are the standard deviations of the daily averages for individual machines relative to the overall long-term average across all machines and days. See Section 5.3 for an explanation of the paging traffic.

traffic. A possible solution is to use the file cache for small files and a separate mechanism for large sequentially-read files.

For write traffic Table 6 provides two different measures of overhead: *writeback traffic* and *write fetches*. The writeback traffic gives the percent of bytes written to the cache that were eventually written to the server. The measured traffic ratio of nearly 90% means that about one-tenth of all new data is overwritten or deleted before it can be passed on to the server; this number is comparable to that predicted in the BSD study and also agrees with the byte lifetimes measured in Section 4.3. The high standard deviation for writeback traffic results from applications that append to the ends of files. While the application may append only a few bytes to the file, the data written back to the server includes the portion from the beginning of the cache block to the end of the appended data. The second source of overhead, write fetches, occurs if an existing file block is partially overwritten. If the block isn't already in the client's cache, then the client must fetch the block from the server as part of the write. Our measurements show that write fetches rarely happen.

Ratio	Client Total (%)		Client Migrated (%)	
File read misses	41.4	(26.9)	22.2	(20.4)
File read miss traffic	37.1	(27.8)	31.7	(22.3)
Writeback traffic	88.4	(455.4)	NA	NA
Write fetches	1.2	(6.8)	1.6	(1.9)
Paging read misses	28.7	(23.6)	8.8	(40.3)

Table 6. Client cache effectiveness. This table gives measurements of the amount of file traffic that client caches do not absorb. Larger values indicate less effective caches. We screened out the effects of processing the large counter files from the counters that were still being recorded. *Read misses* is the percent of cache read operations that were not satisfied in the cache. *Read miss traffic* is the ratio of bytes read from the server to bytes read by applications. *Writeback traffic* is the percent of bytes written to the cache that were eventually written to the server. *Write fetches* is the percent of cache write operations that required a block to be fetched from the server. (This occurs during a partial write of a non-resident file block.) The column *Client Migrated* considers only read and write accesses made on behalf of migrated processes. The numbers in parentheses are the standard deviations of the daily averages for individual machines relative to the overall long-term average across all machines and days.

Table 6 provides a second set of cache measurements that consider only file traffic from migrated processes. Process migration spreads a user's workload out over several machines; because of this, and because processes from one user may migrate to different machines at different times, we hypothesized that file caches would be less effective for migrated processes. The actual behavior is just the opposite: migrated processes actually have a lower read miss ratio than the average for all processes. The most likely explanation is that there is an unusually high degree of locality in the tasks that are most often migrated, since the same user may repeatedly use migration in a short time period. The policy used to select hosts for migration tends to reuse the same hosts over and over again, which may

allow some reuse of data in the caches [2]. It is encouraging to see evidence that a load-sharing scheme such as process migration does not negate the benefits of file caching.

Our measurements support the BSD study's predictions that caches will be more successful at absorbing reads than writes. Table 7 shows a breakdown of file traffic between clients and servers. Of the non-paging traffic, the ratio of reads to writes is about 2:1; in comparison, the raw file traffic generated by applications (Table 5) favors reads by a ratio of 4:1. The cache on the server would further reduce the ratio of read traffic seen by the server's disk.

5.3. The Impact of Paging Traffic

One of the goals of this study was to measure the relationship between paging and file traffic. The paging system and file system are closely related in Sprite: they trade physical memory pages back and forth as described previously, and backing storage for the virtual memory system is implemented with ordinary files so all paging traffic appears as file traffic at some level in the system. We were curious whether one system dominates the other in terms of traffic and whether the relationship is changing over time.

Sprite divides each process's pages up into four groups: code pages, which are read-only, initialized data pages that have not yet been modified, modified data pages, and stack pages. Code and unmodified data pages are paged from the program's executable file, while modified data pages and stack pages are paged to or from backing files. We found that about 50% of paging traffic is to or from backing files, 40% is for code, and 10% is for unmodified data pages.

Sprite uses several techniques to reduce the amount of paging traffic between clients and servers. First, Sprite keeps code pages in memory even after processes exit; the pages tend to remain in memory for many minutes before being replaced, so they can be reused by new invocations of the same program. Second, Sprite favors the virtual memory system when trading pages between the file cache and virtual memory. Third, initialized data pages are cached in the file system. When a process references an initialized data page for the first time, it is copied from the file cache to the virtual memory system. Processes typically dirty all of their data pages, so the data pages must be discarded from virtual memory when processes exit, but clean versions are available from the file cache if the same program is re-run. Sprite does not intentionally cache code pages in the file cache, since the virtual memory system retains code pages, but Sprite still checks the file cache on code faults, since recompilations or other accesses to the executable file can cause new versions of code pages to exist in the file cache. If a code page is found in the file cache, the file cache block is marked for replacement after its contents are copied to the virtual memory system. Table 6 shows that cache accesses for code and initialized data pages have a relatively high hit rate. Pages of backing files are never present in the file caches of clients, but the backing files for clients are cached on the file servers.

Traffic Type		Bytes Read (%)		Bytes Written (%)		Total (%)
Cached	File reads	38.4	(26.3)	0.0	(0.0)	38.4
	Paging	2.7	(5.6)	0.0	(0.0)	2.7
	Writeback	0.0	(0.0)	18.3	(17.8)	18.3
	Write fetch	0.7	(2.4)	0.0	(0.0)	0.7
Uncached	Paging	22.3	(20.8)	13.2	(12.5)	35.5
	Shared files	0.6	(3.1)	0.4	(11.7)	1.0
	Directories	1.0	(2.9)	0.0	(0.0)	1.0
	Other	0.6	(5.2)	1.8	(12.4)	2.4
Total (%)		66.3		33.7		100.0

Table 7. Server traffic. This table gives a breakdown of file traffic between clients and servers. While Table 5 shows the percentages of different types of traffic before the client caches have filtered some of it, this table shows the traffic presented to the server, after the filtering effect of the client caches. The numbers in parentheses are the standard deviations of the daily averages for individual machines relative to the overall long-term average across all machines and days.

Our measurements show a fairly even division between paging and non-paging traffic. Both in terms of raw traffic (Table 5) and server traffic (Table 7), paging represents about 35% of bytes transferred. This division of traffic is similar to what existed at the time of the BSD study. Although the BSD study did not explicitly measure paging activity, Nelson and Duffy measured paging traffic on one of the same machines used for the BSD study at about the same time and found an average paging rate of about 3 Kbytes per second [8]. The BSD study measured about 4 Kbytes of raw file traffic per second, so paging accounted for about 43% of all I/O traffic at that time. This comparison suggests that increasing memory sizes benefit file caches and virtual memory systems about equally, so that neither is likely to dominate the other in the future.

The absolute volume of paging traffic is not very high: during the middle of the work-day each workstation transfers only about one 4-Kbyte page every three to four seconds. However, paging traffic appears to be bursty, and the amount of paging traffic varies significantly across different machines, depending on their memory sizes and the locality of their workloads. We suspect that much of the paging traffic occurs during major changes of activity, such as when a user returns to a workstation that has been used only by migrated processes for several hours. The migrated processes are evicted and the user's activities determine the new workload.

Local disks are often proposed for reducing paging traffic, but we disagree. If backing files were kept on local disks in Sprite, it would only reduce the server traffic by about 20%; this does not dramatically increase the scalability of the overall system. In terms of latency, a local paging disk may actually decrease performance: in Sprite it currently takes about 6 to 7 ms for a client to fetch a 4-Kbyte page from a server's file cache over an Ethernet.

This time is already substantially less than typical disk access times (20 to 30 ms), and it will improve with faster CPUs and networks. In environments where servers have large file caches, it will be much faster to page over the network than from a local disk. Network saturation is not a problem either: 40 Sprite workstations collectively generate only about 42 Kbytes/second of paging traffic, or about four percent of the bandwidth of an Ethernet. With properly-designed virtual memory and file systems, we think it makes more sense to spend money on additional memory than local disks.

We were also concerned that process migration might induce extra paging traffic when a migrated process starts up on a new machine. Fortunately, this appears not to be the case. Most process migrations occur at the same time that a new process is created, so no virtual memory needs to be transferred with the process. Overall, the paging rate for migrated processes was no higher than for other processes.

5.4. Cache Block Replacement and Writeback

This section examines the fate of data once it has been placed in a client's cache. To make room for new data in the cache, old data must be removed. The caches use a least-recently-used mechanism to choose blocks for replacement. Table 8 displays information about block replacement. Most of the time replacement occurs in order to free up a block for other file data, but about one-fifth of the time a file block is ejected so that its page can be given to the virtual memory system. On average, blocks have been unreferenced for almost an hour before they get replaced. With such long cache lifetimes, usually only clean blocks are replaced; dirty blocks have almost always been written back to the server long before they could be replaced.

Dirty cache blocks can be written to the server (''cleaned'') for several reasons; Table 9 indicates how often each of these reasons occurs. The most common reason is Sprite's delayed-write policy, which forces blocks to be written to the server after they have been dirty for 30 seconds. All dirty blocks for a file are written to the server if any block in the file has been dirty for 30 seconds. About three-fourths of all block cleanings occur because the 30-second delay has elapsed. Of the remaining block cleanings, about one-half occur because an application program explicitly requested that the block be written to disk, and the rest occur because the file is about to be read on a different workstation and the server needs the most up-to-date copy to supply to that workstation.

New Contents	Blocks Replaced (%)		Age (minutes)	
Another file block	79.4	(58.5)	47.6	(1301.6)
Virtual memory page	20.6	(36.2)	201.0	(1163.5)

Table 8. Cache block replacement. For a cache block to be replaced it must be the least-recently-used. The *Blocks Replaced* column indicates the percentage of blocks that were replaced with data for another file block and the percentage that were passed to the virtual memory system. The *Age* column indicates the average number of elapsed minutes between the last reference to a block and the time it was replaced. The numbers in parentheses are the standard deviations of the daily averages for individual machines relative to the overall long-term average across all machines and days.

Reason for Write	Blocks Written (%)		Age (seconds)	
30-second delay	71.1	(29.6)	27.2	(6.0)
Write-through requested by application	16.2	(22.7)	1.3	(4.6)
Server recall	12.6	(17.8)	19.0	(9.1)
Virtual memory page	0.1	(1.3)	11.9	(9.8)

Table 9. Dirty block cleaning. A block can be cleaned for any of four reasons: (a) it has been dirty for 30 seconds; (b) an application program invoked the `fsync` kernel call to force the block to the server's disk; (c) the server needs the block to be written (*recalled*) because another client is about to access the file; or (d) the page is given to the virtual memory system. The *Blocks Written* column indicates the relative frequencies of these reasons. The *Age* column gives the average elapsed time in seconds between when the block was last written and when it was written back to the server. The numbers in parentheses are the standard deviations of the daily averages for individual machines relative to the overall long-term average across all machines and days.

Table 9 shows that data integrity, and not cache size, is the reason why dirty bytes leave the cache. Dirty blocks almost never leave the cache to make room for other blocks; they are usually written out to make new data permanent, either by the delayed-write mechanism or by request from the application. For this reason, increasing the size of the file cache will not reduce its writeback traffic. The write traffic can only be reduced by increasing the writeback delay or reducing the number of synchronous writes requested by applications. This would leave new data more vulnerable to client crashes.

5.5. The Importance of Cache Consistency

Sprite's caching mechanism provides ''perfect'' consistency: it guarantees that each read operation will return the most recently written data for the file, even if the file is being read and written simultaneously on more than one machine. Most network file systems, such as AFS [4] and NFS [16], don't provide such strong consistency guarantees. In this section we attempt to evaluate the importance of cache consistency, first by measuring how often Sprite takes special actions to guarantee consistency, and second by estimating how many errors might occur under a weaker

Type of Action	File Opens (%)	
Concurrent write-sharing	0.34	(0.18-0.56)
Server recall	1.7	(0.79-3.35)

Table 10. Consistency action frequency. This table gives the frequency of various consistency actions, measured as a percent of all file (excluding directory) opens. *Concurrent write-sharing* refers to opens that result in a file being open on multiple machines and being open for writing on at least one of the machines. *Server recall* refers to opens for which the file's current data resides on another client and the server must retrieve it. The numbers in parentheses give the minimum and maximum values among the 8 traces.

---

approach to consistency.

Table 10 measures the frequency of various consistency actions in Sprite. About one in every 300 opens causes a file to be opened for reading and writing on more than one machine. In addition, for about one in every 60 opens the server recalls dirty data from another client's cache in order to obtain the most recent version of a file. This is an upper bound on the required number of server recalls, because the Sprite server does not keep track of whether the client has already finished flushing its dirty data via the delayed write mechanism.

To estimate the negative impact of a weaker cache consistency scheme, we used our trace data to simulate a cache consistency mechanism similar to that used in some NFS implementations [16]. In the simulated mechanism, a client considers data in its cache to be valid for a fixed interval of time; on the next access to the file after the expiration of the interval, the client checks with the file's server and updates its cache if necessary. New data is written through to the server almost immediately in order to make it available to other clients. However, if one workstation has cached data for a file when another workstation modifies the file, the first workstation may continue to use its stale cache data. The actual NFS approach is an adaptive one in which the interval varies between 3 and 60 seconds, depending on how recently the file has been modified; for our simulations we used fixed intervals of 3 and 60 seconds.

Table 11 presents the results of the simulations. A 60-second refresh interval would have resulted in many uses of stale data each hour; one-half of all users would have accessed stale data over a 24-hour period. Although a 3-second refresh interval reduces errors, about 3 out of 52 users would still have received stale data at some point in an average 24-hour period. This is a large number of potential file access errors, particularly when compared to other sources of error such as undetected network or disk errors, and we can eliminate them with a cache consistency mechanism.

We had hypothesized that migrated processes would experience more stale-data errors than normal processes, since migration could cause files to be opened on more than one machine as a process migrates. (This assumes that no special measures are used to flush dirty data before a process migrates.) Our hypothesis appears to be false, since our results show that migrated processes are no more likely to cause errors than non-migrated processes. This may be because many processes are migrated before they begin executing (they are ''remotely executed''); these processes open most of their files after they migrate.

Our overall conclusion from Tables 10 and 11 is that shared access to modified data occurs often enough that a system should provide cache consistency. Otherwise, it appears that many users could be inconvenienced by stale data or have to change their working habits to avoid potentially dangerous situations. An examination of the user IDs of the processes in the trace showed that users accustomed to working in NFS environments are just as likely to suffer stale data errors as users accustomed to working on Sprite. This suggests that users have not adjusted their access patterns to NFS's weak consistency and that errors probably happen frequently under some implementations of NFS.

5.6. Algorithms for Concurrent Write-Sharing

When a file is open on several workstations and at least one of them has the file open for writing, the file is said to undergo *concurrent write-sharing*. As previously described, Sprite handles this situation by disabling client caching for the file. In our measurements only about 1% of server traffic is caused by write-shared files (see Table 7), so the current approach is not a major performance problem. Nonetheless, we were curious about the potential benefits of other consistency mechanisms. Sprite's approach is simple, but it isn't necessarily the most efficient. In this section we use the trace data to compare Sprite's algorithm for concurrent write-sharing with two alternatives.

The first alternative approach is a slight modification of the Sprite scheme. In the current Sprite implementation, an uncacheable file remains uncacheable until it has been closed by all clients. The modified scheme makes a file cacheable again as soon as it has been closed by enough clients to eliminate the concurrent write-sharing.

The second alternative is a token-based scheme similar to that implemented in the Locus [14], Echo [6], and DEcorum [5] file systems. In this approach a file is always cacheable on at least one client. In order to access a file, a client must obtain a read-only or read-write token from the server; once a client has obtained the appropriate token it is free to cache the file and access it. The server guarantees that at any given time there is either a single read-write token or any number of read-only tokens outstanding for each file. Concurrent write-sharing takes the form of conflicting token requests. When this occurs, the server recalls one or more of the existing tokens in order to grant the new request. When a write token is recalled from a client, the client must flush any dirty data for that file back to the server. If the new token granted is a write token, all

Measurement	60-second		3-second	
Average errors per hour	18	(8-53)	0.59	(0.12-1.8)
Percent of users affected during 24 hours	48	(38-54)	7.1	(4.5-12)
Percent of users affected over all traces	63	NA	20	NA
Percent of file opens with error	0.34	(0.21-0.93)	0.011	(0.0001-0.032)
Percent of migrated file opens with error	0.33	(0.05-2.8)	< 0.0l	(0.0-0.055)

Table 11. Stale data errors. This table lists our simulations of a weaker cache consistency mechanism based on polling: clients refresh their caches by checking the server for newer data at intervals of 60 seconds or 3 seconds. Errors are defined as potential uses of stale cache data. Affected users are those whose processes suffered errors. The numbers in parentheses give the minimum and maximum values among the 8 traces.

---

the clients that give up tokens for that file must also invalidate their cache blocks for the file.

The token-based approach allows a file to be cached in many cases for which the Sprite scheme, or even the modified Sprite scheme, would cause the file to become uncacheable. For example, if two workstations both have a file continuously open but they don't actually access it at the same time, then Sprite will make the file uncacheable but the token approach will allow the file to be cached. On the other hand, the token approach is worse than the Sprite approach in the extreme case where a file is undergoing concurrent read-write sharing on a very fine grain. In this case, the token approach will generate just as much server traffic as Sprite: the caches will be flushed continuously by token recalls, whole cache blocks will be reread many times, and there will be additional overhead for issuing and recalling tokens.

In order to compare these three approaches quantitatively, we recorded additional information in the traces on the file servers. For files undergoing concurrent write sharing, we logged the starting position, size, and time for every read or write request on the shared file (this was easy to do since all such uncacheable requests are passed through to the server in Sprite). We then wrote a cache simulator that scanned the trace files and computed the consistency overhead for each of the three algorithms. The simulator considered only the accesses that occurred during concurrent write-sharing, and it assumed infinitely large client caches, so blocks were never removed from the simulated caches except in response to consistency actions. The simulator included the effects of a 30-second delayed write scheme. In computing the overheads for the token-based mechanism, we assumed that remote procedure calls would be combined (piggy-backed) whenever possible (e.g. write-token recalls are combined with dirty-data recalls).

Table 12 shows the simulation results for the different consistency mechanisms both in terms of bytes transferred between the clients and server and in terms of remote procedure calls. We were surprised to discover that there were only small differences in overhead between the algorithms. Since Sprite leaves a file uncacheable for a larger amount of time than the other approaches, we expected that it would perform comparatively poorly. Instead, only the token approach shows an improvement, and only by 2% in terms of bytes and 20% in terms of remote procedure calls. The reason for this lack of improvement is small I/O requests made by some applications. While a file is uncacheable, reads and writes go through to the server. If these reads and writes are small (less than a block's worth of data), only a few bytes are transferred. In the consistency mechanisms that allow a file to become cacheable again (the modified Sprite and token mechanisms), these small reads and writes miss in the cache at first and cause whole cache blocks to be fetched from the server. During concurrent write-sharing, we see a significant number of these short I/O operations in our traces. The token mechanism is the most sensitive to application I/O behavior, as seen from the high variance in its performance. When files are shared at a fine grain, the token mechanism invalidates caches and rereads whole cache blocks frequently.

Our conclusion is that concurrent write-sharing is rare enough now that it is reasonable to pick the simplest consistency mechanism. In the future, if write-sharing increases, the I/O behavior of the applications causing it should be examined in order to choose the best mechanism.

Method	Shared Bytes Transferred (%)		RPCs per I/O Event	
Sprite	100	(100-100)	1.02	(1.00-1.09)
Modified Sprite	123	(86-156)	1.20	(0.89-1.44)
Token	98	(37-275)	0.85	(0.14-1.00)

Table 12. Cache consistency overhead. This table lists cache consistency overheads for three different consistency schemes: the Sprite approach, a modification of the Sprite approach in which files become cacheable again when concurrent write-sharing ends, and a token-based approach. The second column shows the ratio of bytes transferred by the consistency algorithm compared to the actual number of bytes requested by the applications to write-shared files. The current sprite mechanism transfers exactly these bytes. The third column gives the number of remote procedure calls for an algorithm divided by the actual number of read and write events requested by the applications to write-shared files. The numbers in parentheses give the minimum and maximum values among the 8 traces.

6. Summary

Our measurements of application-level file accesses show many of the same trends found by the BSD study six years ago. Average throughput per user is relatively low, most files are short and are opened for only brief periods of time, most accesses are sequential, and most bytes don't live more than a few minutes. We found two substantial changes, however. First, file throughput has increased by a factor of 20 overall and has become much more bursty. Second, typical large files used today are more than an order of magnitude larger than typical large files used in 1985. Large files account for much of the increase in throughput and burstiness, and they stress many parts of the system, such as the file caches.

In our measurements of the Sprite file caches, we found that increases in cache size have led to increases in read hit ratios, but the improvements have been much smaller than we expected. We suspect that the increasing size of large files accounts for this discrepancy. We found almost no improvement in the effectiveness of caches at reducing write traffic: about 90% of all new bytes eventually get written to the server in order to safeguard the data. If read hit ratios continue to improve, then writes will eventually dominate file system performance and new approaches, such as longer writeback intervals, non-volatile cache memories, and log-structured file systems [15], will become attractive.

We found that many users access file data in a way that assumes cache consistency among workstations, and that they will be inconvenienced on a daily basis if full consistency is not provided. Fortunately, we also found that the overheads for implementing consistency are very low, since write-sharing only occurs for about one percent of file accesses. Our simulations of cache consistency mechanisms showed no clear winner: the mechanisms had comparable overheads, and where there were differences they depended strongly on the application mix. Without specific information about application behavior, it seems wisest to choose a consistency mechanism based on the simplicity of its implementation.

Lastly, we found that process migration increases the burstiness of file traffic by an order of magnitude. For example, users with migrated processes generated file traffic at a short-term rate 40 times the medium-term average rate for all users. Fortunately, we found that migration does not seem to degrade the performance of file caches or increase cache consistency overheads. In fact, we found that file caches worked better for migrated processes than for processes in general.

7. Acknowledgements

We owe special thanks to David Bacon, Fred Douglis, Bruce Nelson, Mendel Rosenblum, Jim Mott-Smith, and the SOSP program committee reviewers for their many helpful comments on this paper. We'd also like to thank Bob Bruce for his help in the formative stages of this study. Finally, we thank all the patient users of Sprite who put up with us and the trouble we caused during the course of these measurements.

8. References

1. Cheriton, D. R., ''The V Kernel: A Software Base for Distributed Systems'', *IEEE Software 1*, 2 (April 1984), 19-43.

2. Douglis, F. and Ousterhout, J., ''Transparent Process Migration: Design Alternatives and the Sprite Implementation'', *Software—Practice & Experience 21*, 7 (July 1991).

3. Hisgen, A., Birrell, A., Mann, T., Schroeder, M. and Swart, G., ''Availability and Consistency Tradeoffs in the Echo Distributed File System'', *Proceedings of the Second Workshop on Workstation Operating Systems*, September 1989, 49-53.

4. Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N. and West, M. J., ''Scale and Performance in a Distributed File System'', *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51-81.

5. Kazar, M. L., Leverett, B. W., Anderson, O. T., Apostolides, V., Bottos, B. A., Chutani, S., Everhart, C. F., Mason, W. A., Tu, S. and Zayas, E. R., ''DEcorum File System Architectural Overview'', *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 11-15 1990, 151-164.

6. Mann, T., Hisgen, A. and Swart, G., *An Algorithm for Data Replication*, Digital Systems Research Center Tech. Rep. 46, June 1989.

7. Mullender, S., van Rossum, G., Tanenbaum, A., van Renesse, R. and van Staveren, H., ''Amoeba: A Distributed Operating System for the 1990s'', *IEEE Computer 23*, 5 (May 1990), 44-53.

8. Nelson, M. N. and Duffy, J. A., *Feasibility of Network Paging and a Page Server Design*, Term project, CS 262, Department of EECS, University of California, Berkeley, May 1984.

9. Nelson, M. N., Welch, B. B. and Ousterhout, J. K., ''Caching in the Sprite Network File System'', *ACM Transactions on Computer Systems 6*, 1 (February 1988), 134-154.

10. Nelson, M. N., *Physical Memory Management in a Network Operating System*, PhD Thesis, University of California, Berkeley, November 1988. Also available as Technical Report UCB/CSD 88/471.

11. Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M. and Thompson, J. G., ''A Trace-Driven Analysis of the UNIX 4.2 BSD File System'', *Proceedings of the 10th Symposium on Operating System Principles*, Orcas Island, WA, December 1985, 15-24.

12. Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M. and Welch, B., ‘‘The Sprite Network Operating System’’, *IEEE Computer 21*, 2 (February 1988), 23-36.

13. Ousterhout, J., ‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’, *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, June 11-15 1990, 247-256.

14. G. J. Popek and B. J. Walker, eds., *The LOCUS Distributed System Architecture*, MIT Press, Cambridge, MA, 1985.

15. Rosenblum, M. and Ousterhout, J. K., ‘‘The Design and Implementation of a Log-Structured File System’’, *Proceedings of the 13th Symposium on Operating System Principles*, Asilomar, CA, October 1991.

16. Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. and Lyon, B., ‘‘Design and Implementation of the Sun Network Filesystem’’, *Proceedings of the Summer 1985 USENIX Conference*, Portland, OR, June 1985, 119-130.

17. Welch, B. B., *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*, PhD Thesis, University of California, Berkeley, February 1990. Also available as Technical Report UCB/CSD 90/567.