A Trace-Driven Analysis of the UNIX 4.2 BSD File System

John K. Ousterhout, Hervé Da Costa, David Harrison,
John A. Kunze, Mike Kupfer, and James G. Thompson

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

Abstract

We analyzed the UNIX 4.2 BSD file system by recording user-level activity in trace files and writing programs to analyze the traces. The tracer did not record individual read and write operations, yet still provided tight bounds on what information was accessed and when. The trace analysis shows that the average file system bandwidth needed per user is low (a few hundred bytes per second). Most of the files accessed are open only a short time and are accessed sequentially. Most new information is deleted or overwritten within a few minutes of its creation. We also wrote a simulator that uses the traces to predict the performance of caches for disk blocks. The moderate-sized caches used in UNIX reduce disk traffic for file blocks by about 50%, but larger caches (several megabytes) can eliminate 90% or more of all disk traffic. With those large caches, large block sizes (16 kbytes or more) result in the fewest disk accesses.

1. Introduction

This paper describes a series of measurements made on the UNIX 4.2 BSD file system [5,8]. Most of the work was done in a series of term projects for a graduate course in operating systems at the University of California at Berkeley. Our goal was to collect information that would be useful in designing a shared file system for a network of personal workstations. We were interested in such questions as:

- How much network bandwidth is needed to support a diskless workstation?
- What are typical file access patterns (and what protocols will support those patterns best)?
- How should disk block caches be organized and managed?
- How much of a performance advantage do such caches provide?

We were unable to find answers to these questions in the literature, so we decided to instrument the 4.2 BSD system to collect information about file accesses. In order to reduce the size of the trace files and the impact of the tracer on its host systems, we did not record individual read and write requests. The information that we did collect allowed us to deduce the exact ranges of bytes accessed, although the access times were less precise than they would have been if we had logged reads and writes. Section 3 of this paper discusses the tracing technique and Section 4 describes the three systems we traced.

We wrote two programs to process the trace files: a reference pattern analyzer and a block cache simulator. Table I summarizes the most important results. Section 5 discusses the reference pattern analysis. Some of the conclusions are: individual users make only occasional (though bursty) use of the file system, and they need very little bandwidth on average (only a few hundred bytes per second per active user); files are usually open only a short time, and they tend to be read or written sequentially in their entirety; non-sequential access is rare; most of the files that are accessed are short; and most new files have short lifetimes (only a few minutes).

Section 6 describes the second part of the analysis, a series of disk-block cache simulations based on the trace data. The main conclusions are that even moderate-sized disk block caches such as those used in UNIX (a few hundred kilobytes) can reduce disk traffic for file blocks by about a factor of two. But larger caches of several megabytes perform much better,

On average, about 300-600 bytes/second of file data are read or written by each active user.
About 70% of all file accesses are whole-file transfers, and about 50% of all bytes are transferred in whole-file transfers.
75% of all files are open less than .5 second, and 90% are open less than 10 seconds.
About 20-30% of all newly-written information is deleted within 30 seconds, and about 50% is deleted within 5 minutes.
A 4-Mbyte cache of disk blocks eliminates between 65% and 90% of all disk accesses for file data (depending on the write policy).
For a 400-kbyte disk cache, a block size of 8 kbytes results in the fewest number of disk accesses for file data. For a 4-Mbyte cache, a 16-kbyte block size is optimal.

Table I. Selected results.

reducing disk traffic by as much as 90%. With large caches and the delayed-write policy described in Section 6, many files will not be written to disk at all: they will be deleted or overwritten while still in the cache. Large block sizes (8 or 16 kbytes) combined with large caches result in the greatest reductions in disk I/O. Even for relatively small caches, large block sizes are effective in reducing disk I/O.

2. Previous Work

There has been very little empirical data published on file system usage or performance. This is probably due to the difficulty of obtaining trace data, and also to the large volume of data that is likely to result. The published studies are limited in scope, and most deal with older operating systems. As a consequence, the results may not be applicable in planning future systems.

For example, Smith studied the file access behavior of IBM mainframes in order to predict the effects of automatic file migration [11]. He only considered files used by a particular interactive editor, which were mostly program source files. The data were gathered as a series of daily scans of the disk, so they do not include files whose lifetimes were less than a day. In another study, Porcar analyzed dynamic trace data for files in an

IBM batch environment [7]. He considered only shared files, which accounted for less than 10% of all the files accessed in his system. Satyanarayanan analyzed file sizes and lifetimes on a PDP-10 system [10], but the study was made statically by scanning the contents of disk storage at a fixed point in time.

More recently, Smith used trace data from IBM mainframes to predict the performance of disk caches [12]; his conclusions are similar to ours although he used different trace information (physical disk addresses, no information about files, transfer sizes or reading versus writing). Two other recent studies contain UNIX measurements that partially overlap ours: Lazowska et al. analyzed block size tradeoffs and reported on the disk I/O required per user [2], and Leffler et al. reported on the effectiveness of current UNIX disk caches [4]. Sections 5 and 6 of this paper compare their results and ours.

3. Gathering the Data

Our main concern in gathering file system trace information was the volume of data. We wished to gather data over several days to prevent temporary unusual activity from biasing the results. If we had attempted to record all file system activity, an enormous amount of data would have been produced. For example, the traces for Smith's cache study contained 1.5 gigabytes or more per day [12]. We feared that the work involved in writing such a trace file would have consumed a substantial fraction of the CPU. It might have perturbed our results, and it certainly would have made us unpopular with the systems' users. In addition, the volume of data would have been so great that we could only have traced a few hours of activity before running out of space for the trace files.

3.1. No Reads and Writes

In order to reduce the volume of data, we decided to record file-system-related events at a logical level rather than a physical level, and *not* to record individual read and write requests. Table II shows the events that were logged. ‘‘Logical’’ level means that information was recorded about files and ranges of bytes within files, not about physical disk blocks. There is no information in the traces about the locations of blocks on disk or the timing of actual disk I/Os. Furthermore, the traces do not contain any information about disk accesses for paging, file name lookup, or file descriptors (see Section 3.2 below).

System Call	Information Recorded
open and create	time, open id, file id, user id, file size
close	time, open id, final position
seek (reposition within file)	time, open id, previous position, new position
unlink (delete file)	time, file id
truncate (shorten file)	time, file id, new length
execve (load program)	time, file id, user id, file size

Table II. The events recorded by the trace package. *Time* is accurate to approximately 10 milliseconds. *Open id* is a unique identifier assigned to each ‘‘open’’ system call. It is used to avoid confusion between concurrent accesses to the same file. *File id* is unique to each file. *User id* identifies the account under which the operation was invoked. *Position* is the current access position in the file (i.e. the byte offset to/from which data will be transferred next).

Once we decided to gather information at a logical level, we could take advantage of the fact that file reading and writing in UNIX are implicitly sequential (a special system call must be used to change the access position within the file). This means that read and write events need not be logged to determine which data were accessed. We recorded the current access position in the file when it was opened and closed, and also before and after each repositioning operation. This information completely identifies the areas of files that were read or written.

The drawback of the no-read-write approach is that it reduces the accuracy of times in the system: the open, close, and reposition events provide bounds on when bytes were actually transferred, but these may be loose bounds if open files are idle for long periods. In all of our analyses, we ‘‘billed’’ each transfer at the time of the next close or reposition event for the file. When analyzing concurrent accesses to different files, the order in which we processed the data transfers may not be the same as the order in which reads and writes occurred.

We had two hypotheses about usage patterns that led us to adopt the no-read-write approach in spite of its potential inaccuracy. First, we thought that most file system activity would be sequential, so that the no-read-write approach would reduce the volume of trace data substantially. Our experiences bear out this hypothesis. Second, we thought that most files would only be open a short time, so that the *open* and *close* events would provide tight bounds on the access times. This hypothesis is also supported by the data in Section 5.

After collecting the trace data we measured the intervals between successive trace events for the same open file. These bound the times when data transfers actually occurred. 75% of the intervals were less than 0.5 second, 90% were less than 10 seconds, and 99% were less than 30 seconds. The measurements in Sections 5 and 6 were averaged over intervals of at least 10 seconds and often longer, so we do not believe that the time imprecision biased our results very much. A later study [13] suggests that no-read-write approach exaggerates slightly the burstiness of the system. This makes our performance numbers slightly pessimistic. For example, [13] concludes that actual cache miss ratios will be 2-3% lower than predicted by Section 6.

3.2. Missing Data

Our trace analyses consider both user- and system-initiated file access, but they examine only the actual bytes contained in files. We did not include paging activity, nor did we include the overhead I/O activity needed to interpret pathnames or to read and write file descriptors. The paragraphs below discuss these other factors individually. It appears that the other factors could result in as much disk activity as the logical file accesses that we measured in detail. Fortunately, the results presented in this paper are independent of the other factors, with the exception of the block cache simulations of Section 6.

The first ‘‘other factor’’ is paging activity, which consists primarily of loading programs on demand from disk files into main memory. Paging to and from swapping store can also result in I/O activity but is rare in 4.2 BSD systems (see [2] and [6]). We estimated the effects of paging by logging *execve* system calls and recording the sizes of the files that were executed. The total number of bytes in such files ranged from 1.2 to 2 times the total number of bytes of logical file I/O, depending on the system measured. However, the actual paging I/O was probably less than this, for three reasons. First, UNIX provides shared code segments and will not re-read code pages if they are already in use by another process. Second, program files may contain large amounts of debugging information, which is never paged in. Third, files are paged in on demand, which means some pages may never be read. See Section 6 for an estimate of the effect of program page-in on disk block caches.

The second additional source of disk I/O consists of file descriptors (i-nodes), which map logical file blocks to disk blocks. UNIX maintains a main-memory cache for the i-nodes of all open files and many recently-used ones. We were not able to measure the effectiveness of this cache. In the (unlikely) worst case, i-node transfers could result in more disk I/O than the actual file blocks (for example, access to a small file might consist of reading the i-node on file open, reading or writing one file block, then writing the i-node on file close).

The third additional source of disk I/O is the directories that must be examined when opening files. This results in a minimum of two block accesses for each element in a file's pathname (one for the directory's descriptor and one for the contents of the directory). However, 4.2 BSD contains a directory cache to hold recently-used entries. Leffler et al. report that the directory cache achieves an 85% hit ratio [4].

4. The Traced Systems

We collected trace data on three different systems, all timeshared VAX-11/780s in the Department of Electrical Engineering and Computer Sciences at U.C. Berkeley. The machines' names are ‘‘Ucbarpa’’, ‘‘Ucbernie’’, and ‘‘Ucbcad’’, and the traces we used for analysis are called ‘‘A5’’, ‘‘E3’’, and ‘‘C4’’, respectively. Ucbarpa and Ucbernie are both used primarily by graduate students and staff for program development

Trace	A5		E3		C4	
Duration (hours)	79.4		65.7		72.5	
Number of trace records	1,017,464		921,526		733,403	
Size of trace file (Mbytes)	26		23		18	
Total data transferred to/from files (Mbytes)	1220		1196		1030	
create events	38,142	(3.8%)	37,172	(4.1%)	29,462	(4.1%)
open events	320,065	(31.9%)	280,579	(30.9%)	203,613	(28.2%)
close events	358,191	(35.7%)	317,763	(35.0%)	233,078	(32.3%)
seek events	185,709	(18.5%)	169,714	(18.7%)	189,245	(26.2%)
unlink events	37,780	(3.8%)	36,517	(4.0%)	28,373	(3.9%)
truncate events	1,485	(0.1%)	2,070	(0.2%)	1,115	(0.1%)
execve	60,712	(6.1%)	64,732	(7.1%)	37,704	(5.2%)

Table III. Overall statistics for the three traces. The percentages are expressed as fractions of all events in that trace.

and document formatting. Ucbernie supports a substantial amount of secretarial and administrative work. Ucbarpa has 4 Mbytes of primary memory and Ucbernie has 8 Mbytes. The third machine, Ucbcad, is used primarily by electrical engineering graduate students to run computer-aided design tools for integrated circuits. Circuit simulators, layout editors, design-rule checkers, and circuit extractors are commonly-used programs on this machine. Ucbcad has 16 Mbytes of primary memory. We included Ucbcad in the analysis to see if CAD programs would show different file system behavior from program development and word-processing programs. The results in Sections 5 and 6 show little difference between the three machines.

Table III gives summary information about the three traces. Each was gathered over a period of 2-3 days during the busiest part of the work week. During the peak hours of the day, about 2-3 files were opened per second, on average. For the A5 and E3 traces, the UNIX load average was typically 5-10 during the afternoon, with a few dozen users active at any given time. For the C4 trace the load average rarely exceeded 2 or 3, with around ten active users at a time. About 5000-6000 bytes of trace data per minute were collected, on average. Although the worst-case rate was somewhat higher than this, there was no noticeable degradation in the performance of the systems while the traces were being gathered.

5. How the File System is Used

Our trace analysis was divided up into two parts. The first part contains measurements of current UNIX file system usage. They are presented in this section under three general categories: system activity (how much the file system is used), access patterns (sequentiality, dynamic file sizes, and open times), and file lifetimes. The second part of the analysis, examining the effectiveness of disk block caches, is presented in Section 6.

5.1. System Activity

The first set of measurements concerns overall system activity in terms of users, active files, and bytes transferred; see Table IV. The most interesting measurement for us is the throughput per active user. We consider a user to be active if he or she has any file system activity in a ten-minute interval.

	A5	E3	C4
Average throughput (bytes/sec. over life of trace)	4200	5080	3940
Total number of different users over life of trace	137	331	169
Greatest number of active users in a 10 minute interval	29	44	20
Average number of active users (over 10 minute intervals)	11.7 (± 5.8)	18.7 (± 10.1)	7.4 (±4.1)
Average throughput per active user (bytes/sec. over 10 minute intervals)	370 (± 290)	280 (± 190)	570 (±760)
Average number of active users (over 10 second intervals)	2.5 (± 1.5)	3.3 (± 2.0)	1.7 (±1.1)
Average throughput per active user (bytes/sec. over 10 second intervals)	1490 (± 10000)	1380 (± 4100)	1790 (± 7400)

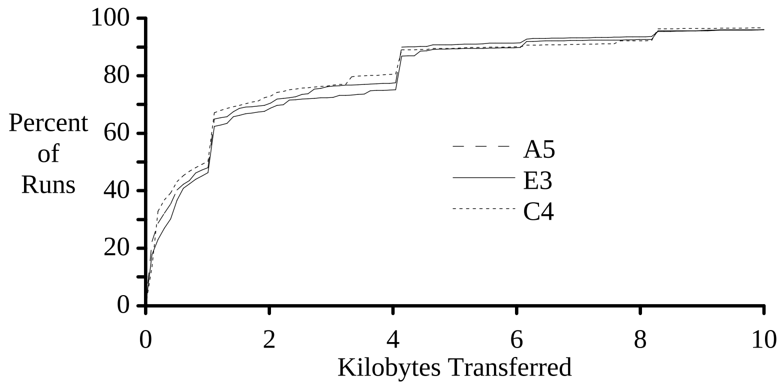
Table IV. Some measurements of system activity. The numbers in parentheses are standard deviations. A user is active in an interval if there are any trace events for that user in the interval. For example, the lower-right entry in the table means that if a user was active in a 10-second interval, he/she requested 1790 bytes of file data per second during that interval, on average.

Averaged over ten-minute intervals, active users tend to transfer only a few hundred bytes of file data per second. If only ten-second intervals are considered, users active in these intervals tend to have much higher transfer rates (a few kilobytes per second per user) but there are fewer active users. In [2] Lazowska et al. reported about 4 kbytes of I/O per second per active user. This is somewhat higher than our figure, but their measurement includes additional overhead not present in our analysis, such as paging I/O and directory searches, and was measured for a single user at a time of heavy usage.

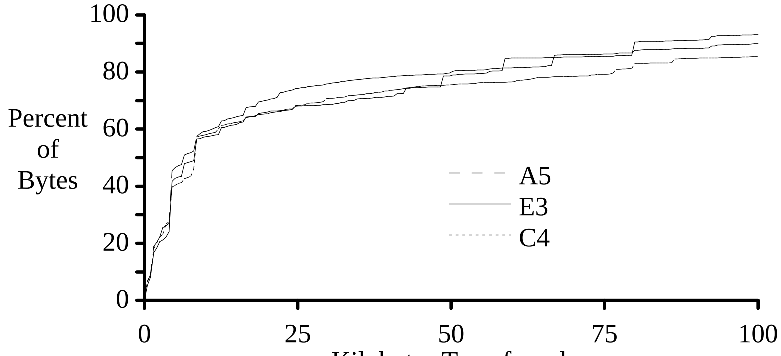
The low average throughput per user suggests that a network-based file system using a single 10 Mbit/second network can support many hundreds of users without overloading the network. Transfer rates tended to be relatively bursty in our measurements, with rates as high as 100 kbytes/sec recorded for some users in some intervals, but even so a 10 Mbit/second network could support several such bursts simultaneously without difficulty.

	A5	E3	C4
Whole-file read transfers (% of all read-only accesses)	168,127 (69%)	131,408 (63%)	93,469 (70%)
Whole-file write transfers (% of all write-only accesses)	78,542 (82%)	67,340 (81%)	60,363 (85%)
Data transferred in whole-file transfers (Mbytes)	664 (54%)	592 (49%)	547 (53%)
Sequential read-only accesses (% of all read-only accesses)	221,136 (92%)	189,734 (91%)	122,557 (93%)
Sequential write-only accesses (% of all write-only accesses)	92,954 (97%)	79,847 (96%)	76,425 (98%)
Sequential read-write accesses (% of all read-write accesses)	4215 (19%)	5459 (21%)	8163 (35%)
Data transferred sequentially (Mbytes)	801 (66%)	804 (67%)	703 (68%)

Table V. Data tends to be transferred sequentially. Whole-file transfers were those where the file was read or written sequentially from beginning to end. Sequential accesses include whole-file transfers plus those where there was an initial reposition operation before any bytes were transferred. Only files opened for read-write access showed significant non-sequential use.



(a)



(b)

Figure 1. Cumulative distributions of the lengths of sequential runs (number of bytes transferred before repositioning or closing the file). Figure (a) is weighted by number of runs: about 70-75% of all sequential runs were less than 4000 bytes in length. Jumps occur at 1024 bytes and 4096 bytes because user-level I/O routines round up transfers to these sizes. Figure (b) is weighted by the number of bytes transferred: about 30-40% of all bytes were transferred in runs longer than 25000 bytes.

5.2. File Access Patterns

Table V contains our measurements of sequentiality, which confirm the widely-held belief that file access is highly sequential. More than 90% of all files are processed sequentially, and more than two thirds of file accesses are whole-file transfers. Of those accesses that are not whole-file transfers, most consist of a single reposition to a particular position in the file, followed by a transfer of data to or from that position without any additional repositioning. This mode of operation is used, for example, to append new messages onto existing mailbox files.

Figure 1 measures the lengths of sequential runs in two ways. Figure 1(a) shows that most sequential runs are short, rarely more than a few kbytes in length. This is because most files are short (see below); there simply isn't much data to transfer. On the other hand, Figure 1(b) shows that long sequential runs account for much of the data transferred: 30% of all bytes are read or written in sequential runs of 25 kbytes or more.

Figure 2 shows the dynamic distribution of file accesses by size at close. Most of the files accessed are short. Short files are used extensively in UNIX for directories, command files, memos, circuit description decks, C definition files, etc. The figure also shows that a few very large administrative files account for almost 20% of all file accesses. These files are each around 1 Mbyte in size and are used for network tables, a log of all logins, and other information. They are typically accessed by positioning within the file and then reading or writing a small amount of data.

The file sizes shown in Figure 2 are much smaller than those measured for IBM systems in [7] and [11]. We believe that this difference is due to the better support provided in UNIX for short files, including hierarchical directories and block-based disk allocation instead of track-based allocation. Satyanarayanan's file-size measurements are roughly comparable to ours (about 50% of all his files were less than 2500 bytes), even though his measurements were made statically and his system did not support hierarchical directories [10]. The measurements of Lazowska et. al. are also very similar to ours [2].

Our last measurement of access patterns is displayed in Figure 3. It shows that most files are open only a short time: programs tend to open files, read or write their contents, then close the files again very quickly. This measurement is consistent with our previous observations: if most files are short, and most are accessed as whole-file transfers, then it shouldn't take very long to complete most of the accesses. On the other hand, there are a few files that stay open for long periods of time, such as temporary files used by the text editor.

5.3. File Lifetimes

Both Satyanarayanan [10] and Smith [11] have published measurements of file lifetimes (the intervals between when files

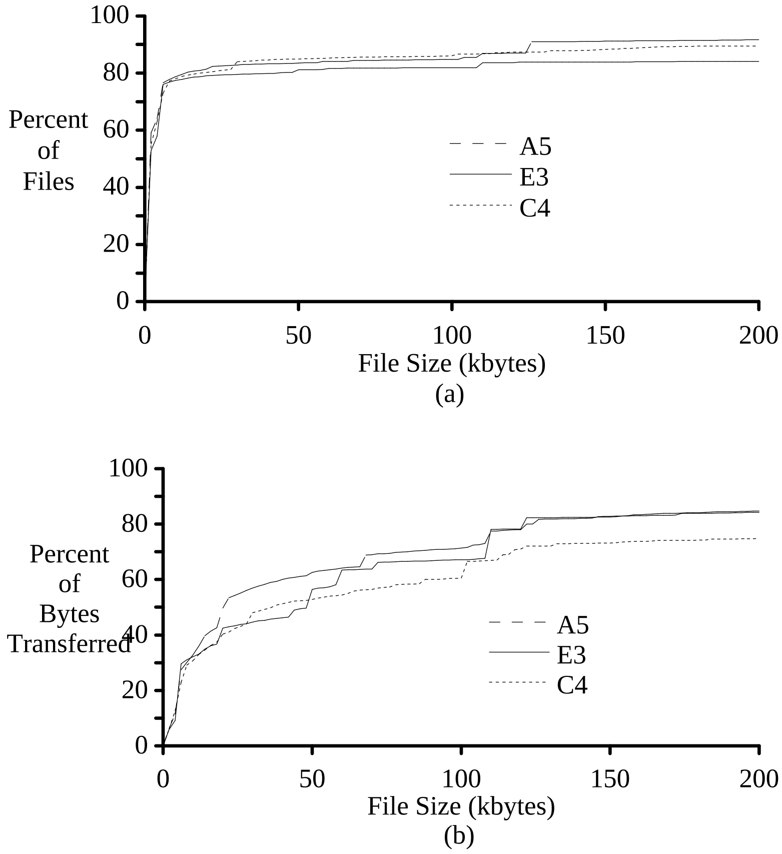


Figure 2. Dynamic distribution of file sizes, measured when files were closed. Figure (a) is a cumulative distribution weighted by number of files. 80% of all file accesses were to files less than 10 kbytes long; most of the remaining 20% were to a few very large administrative files. Figure (b) is also cumulative but is weighted by number of bytes transferred (only about 30% of all bytes were transferred to or from files less than 10 kbytes long).

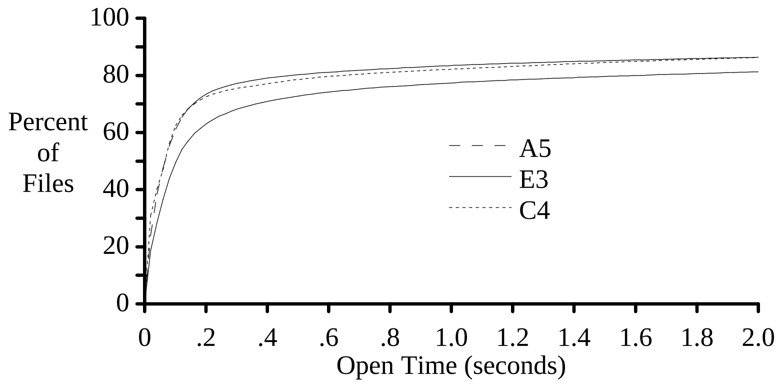


Figure 3. Distribution of times that files were open. This is a cumulative distribution. For example, about 70-80% of all files were open less than .5 second.

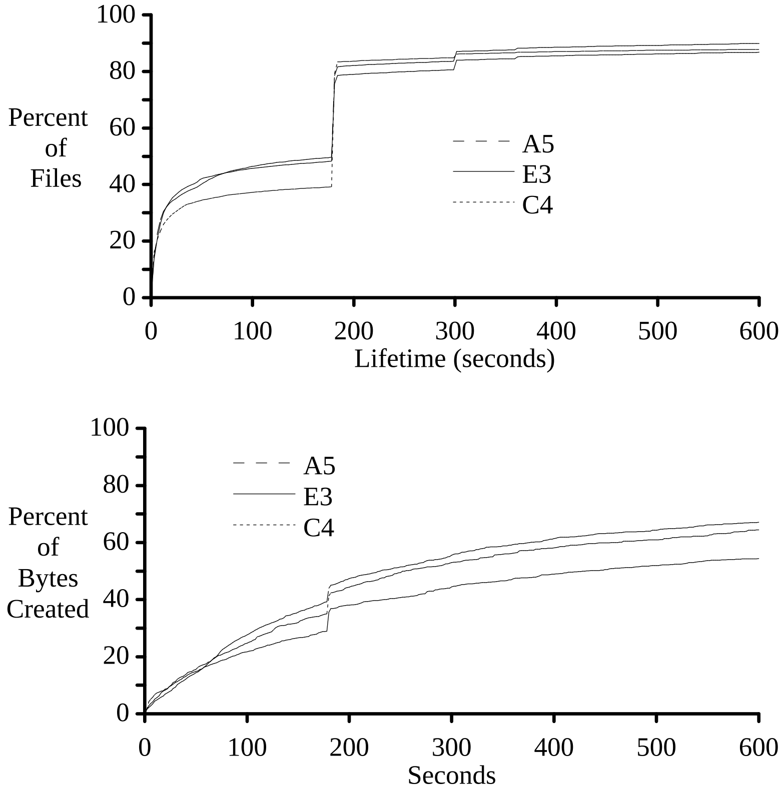


Figure 4. Cumulative distributions of file lifetimes. Figure (a) is weighted by number of files (about 80% of all new files were deleted or completely overwritten in less than 200 seconds). Figure (b) is weighted by the size of the file (files deleted or overwritten in less than 200 seconds accounted for about 40% of all data written to new files). The large jumps at 180 seconds are due to network status daemons.

are written and they are overwritten or deleted; this is actually the lifetime of the file's data, not necessarily the lifetime of the file). In both cases the measurements were made by sampling the ''last-modified'' and ''last-examined'' times of files on a disk, so they describe only long-term behavior (a few days or months). We used our trace data to study file lifetimes over much shorter intervals.

Figure 4 shows the results, which are surprising in two respects. First of all, most file lifetimes are very short: 80% of all new files are deleted or overwritten within about 3 minutes of creation. The second unusual characteristic of the data is the large concentration of lifetimes around 3 minutes. 30-40% of all new files have lifetimes between 179 and 181 seconds. This concentration is due to network daemons that update each of about 20 host status files every three minutes. This feature is peculiar to 4.2 BSD. However, even disregarding the files with lifetimes around 3 minutes, 50-60% of the remaining files have lifetimes less than 3 minutes and 30-40% of all new information (counted by bytes) is overwritten within 3 minutes.

The results in Figure 4 were quite surprising to us, but can be accounted for by temporary files. For example, in program development the compiler generates an assembler file which is deleted as soon as it has been translated to machine code. In a CAD environment, a circuit simulator generates output listings that are examined and then deleted before the next simulation run. In a word-processing environment, printer spool files can account for some of the short lifetimes.

Figure 4 includes only data written to new files: files that did not exist before or that were truncated to zero length after being opened. Although this includes most of the data written (refer back to Table V), it does not include information written to the middle or end of an existing file. Section 6 contains another lifetime measurement that is more inclusive but reaches about the same conclusion.

6. Block Cache Simulations

In considering various designs for a network filing system, one of the most interesting possible areas of change is the cache of disk blocks. The UNIX file system uses about 10% of main memory (200-400 kbytes) for a cache of recently-used disk blocks. This cache is maintained in a least-recently-used fashion and results in a substantial reduction in the number of disk operations.

For a network filing system with dedicated file servers it seems reasonable to use almost all of the servers' memory for disk caches; this could result in caches of eight megabytes or more with today's memory technology, and perhaps 32 or 64 megabytes in a few years. Although the general benefits of block caches are already well-known, there were a number of questions we wished to answer:

- How do the benefits scale with the size of the cache?
- How should the cache be organized to maximize its effectiveness?
- Can large block caches be used without risking large information losses on server crashes?

6.1. The Cache Simulator

In order to answer these questions we wrote a program to simulate the behavior of various kinds of caches, using the trace data to drive the simulations. As mentioned in Section 3, the trace data contains only approximate timing information, which could conceivably have biased the results of a simulation. Fortunately, the inaccuracy in the trace times (a few seconds) is small in comparison to typical cache lifetimes (a few minutes to a few hours), so we doubt that it had much affect on the results. For the measurements below the three traces produced nearly indistinguishable results; only the results from the A5 trace are reported.

In each of the simulations, the disk cache consisted of a number of fixed-size blocks used to hold portions of files. We used a least-recently-used algorithm for cache replacement. When the trace indicated that a range of bytes in a file was read or written, the range was first divided up into one or more block accesses. For each block access, the simulator checked to see if the block was in the cache. If so, it was used from the cache. If not, then the block was added to the cache, replacing the block that had not been accessed for the longest time.

In evaluating the different caches, our principal metric was the *miss ratio*, which is the ratio of disk I/O operations to logical block accesses. The smaller the miss ratio, the better. Disk accesses occurred in two ways in the simulations. First, a disk access was necessary each time a block was referenced that wasn't in the cache, unless the block was about to be overwritten in its entirety. Second, disk accesses were necessary to write modified blocks back from the cache to disk. We experimented with several different write policies, which are discussed below.

In computing block accesses, we assumed that programs made requests in units of the cache block size, rather than as several smaller requests. In practice, though, some programs make smaller requests than these, resulting in lower miss ratios than we have reported (there will be many more block accesses for the same amount of data, but about the same number of disk I/Os).

6.2. Cache Size and Write Policy

The simulations varied in three respects: cache size, write policy, and block size. Figure 5 and Table VI show the effect of varying the cache size and write policy with a block size of

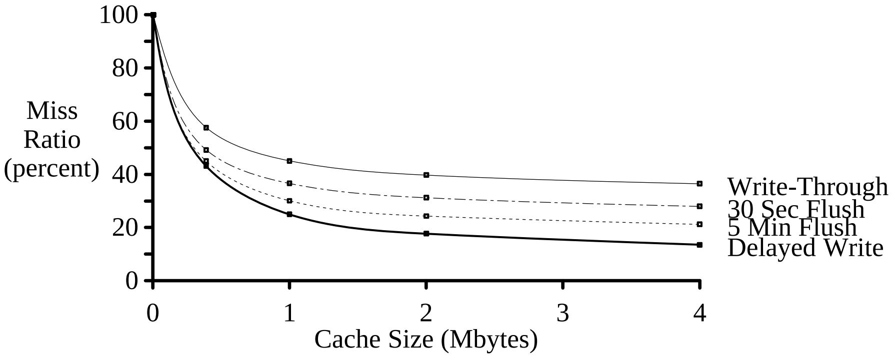


Figure 5. Cache miss ratio as a function of cache size and write policy, using the A5 trace with a cache block size of 4096 bytes.

Cache Size	Write-Through	30 sec Flush	5 min Flush	Delayed Write
390 kbytes (UNIX)	57.6%	49.2%	45.0%	43.1%
1 Mbyte	45.1%	36.6%	30.1%	25.0%
2 Mbytes	39.7%	31.2%	24.3%	17.7%
4 Mbytes	36.5%	28.0%	21.2%	13.5%
8 Mbytes	34.7%	26.2%	19.3%	11.2%
16 Mbytes	33.5%	25.0%	18.1%	9.6%

Table VI. A tabular representation of the data from Figure 5 (miss ratio as a function of cache size and write policy for the A5 trace with 4096-byte cache blocks).

4096 bytes (the most common size in 4.2 BSD UNIX systems). We tried four different write policies in the simulations. The first write policy is *write-through*: each time a block is modified in the cache, a disk access is used to write the block through to disk. Write-through is attractive because it ensures that the disk always contains an up-to-date copy of each block. Unfortunately, about one third of all block accesses were writes, so the miss ratio was never lower than about 30%.

The caches were most effective with the policy we call *delayed-write* (this policy is sometimes referred to as ‘‘copy-back’’ or ‘‘write-back’’). The delayed-write policy waits to write a block to disk until the block is about to be ejected from the cache. This resulted in much better performance for large caches. With a cache size of 16 megabytes, miss ratios less than 10% occurred. The improvement occurred because about 75% of the newly-written blocks were overwritten or their files were deleted before the blocks were ejected from the cache; these blocks were never written to disk at all.

Unfortunately, a delayed-write policy may not be practical because some blocks could reside in the cache a long time before they are written to disk. For example, we found that with a 4 Mbyte cache, about 20% of all blocks stay in the cache longer than 20 minutes. System crashes could cause large amounts of information to be lost. We tried two write policies that were intermediate between write-through and delayed-write. We call these *flush-back* policies. With a flush-back policy the cache is scanned at regular intervals: any blocks that have been modified since the last scan are written to disk. If the flush interval becomes very small then flush-back is equivalent to write-through; if the flush interval becomes very large then flush-back is equivalent to delayed-write.

Figure 5 shows two different flush-back intervals: 30 seconds and 5 minutes. For large caches, a 30-second flush-back policy reduces the number of I/Os by about 25% and a 5-minute flush-back policy reduces the number of I/Os by about 50%. This means that about 25% of newly-written blocks are overwritten or deleted within 30 seconds and about 50% are overwritten or deleted within 5 minutes. These data provide another measurement of the lifetime of information in files, and are similar to the results of Figure 4.

6.3. Block Size

We also evaluated the effectiveness of different block sizes. The original UNIX system used 512-byte blocks, but the block size has grown since then to 1024 bytes in AT&T's System V [1] and 4096 bytes in most 4.2 BSD systems. Figure 6 and Table VII show the results of varying the block size and cache size. For a 4-Mbyte cache, a block size of 16 kbytes reduces disk accesses by about 25% over a 4-kbyte block size and by a factor of 3 over 1-kbyte blocks. Even for a cache size of 400 kbytes, an 8-kbyte block size results in about 10% fewer disk I/Os than a 4-kbyte block size and 60% fewer I/Os than a 1-kbyte block size. This conclusion is similar to the one reached by Lazowska et. al. in [2]. For smaller caches, larger block sizes are less beneficial because they result in fewer blocks in the cache; most of the cache space is wasted since short files only occupy the first portions of their blocks.

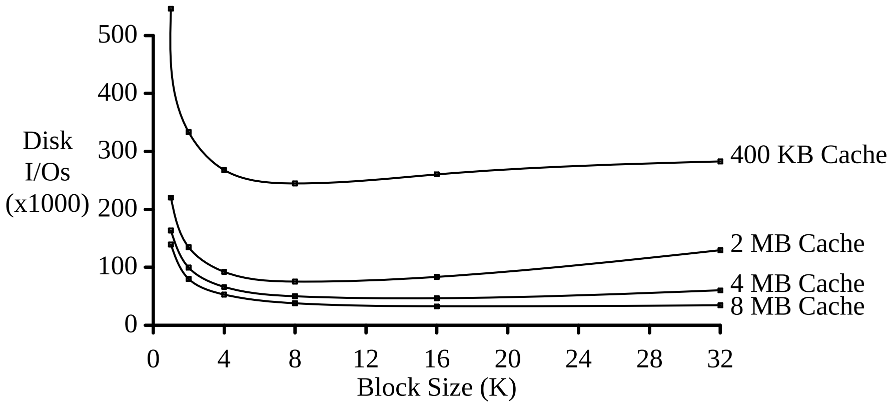


Figure 6. Disk traffic as a function of block size and cache size, for the A5 trace using the delayed-write policy. Large block sizes work well for small caches, but they work even better for large caches. For very large block sizes, the curves turn up because the cache has too few blocks to function effectively as a cache.

Block Size	No Cache	400 Kbyte Cache	2 Mbyte Cache	4 Mbyte Cache	8 Mbyte Cache
1 kbytes	1,432,179	562,492	280,056	227,299	194,724
2 kbytes	925,934	365,806	165,312	129,654	110,369
4 kbytes	623,573	268,864	110,182	84,164	69,651
8 kbytes	527,634	259,941	90,539	65,302	51,635
16 kbytes	481,052	280,068	103,223	63,330	47,626
32 kbytes	461,976	307,002	156,523	82,350	51,883

Table VII. A tabular representation of the data from Figure 6 (disk I/O's as a function of cache size and block size). The first column gives the total number of block accesses for each block size.

Although large blocks are attractive for a cache, they may result in wasted space on disk due to internal fragmentation. Fortunately a scheme like the one in 4.2 BSD, which uses multiple block sizes on disk to avoid wasted space for small files, works well in conjunction with a fixed-block-size cache.

6.4. Comparisons

Typical 4.2 BSD systems run with disk block caches containing about 100-200 blocks of different sizes, with a total cache size of about 400 kbytes. The *sync* system call is typically invoked every 30 seconds to flush the cache. According to our simulations, this combination of cache size and write policy should reduce disk accesses by about a factor of two.

However, Leffler et al. report a measured cache miss ratio of only about 15% [4]. There are two explanations for the discrepancy. First, there are many programs that make I/O requests in units smaller than the cache block size; this inflates the number of logical I/Os and reduces the miss ratio. Second, the measurements in [4] include block accesses for paging, directories, and file descriptors, which we did not consider.

We ran a crude test to verify the hypothesis that paging accesses also exhibit high locality. The trace data include information about which files were executed as programs; we simulated paging activity by forcing a whole-file read to each program file at the time the program was executed. We did not attempt to simulate page-out activity, since [2] and [6] indicate that it rarely happens. As Figure 7 shows, the simulated paging resulted in degraded performance for small cache sizes (the large program files increased the total working set of file information), but improved miss ratios for large cache sizes. This implies that the locality of program accesses is at least as great as that of file data.

From this evidence we think that our miss ratio estimates are likely to be upper bounds; the real benefits of caches should be even better than our figures suggest.

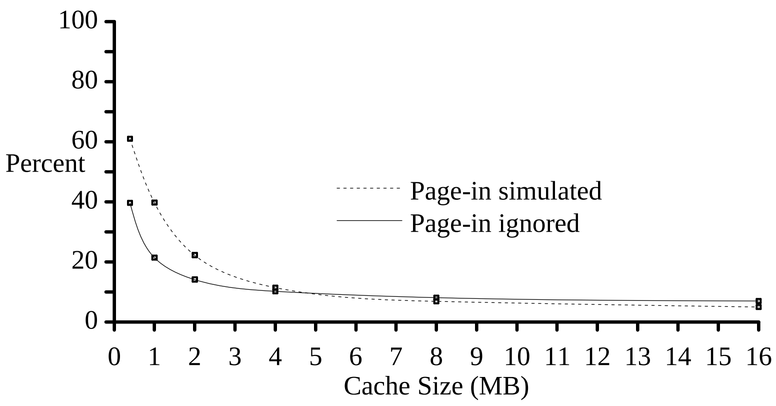


Figure 7. Miss ratios (4096-byte blocks, delayed write, trace A5) with paging behavior approximated by forcing a whole-file read of each program that is executed.

7. Do the Results Generalize?

A few of our results, such as the large number of files with lifetimes around 3 minutes, are peculiar to 4.2 BSD. However, we think that most of the conclusions will apply across a wide range of personal workstations and timesharing systems. We also think that the results will apply to operating systems other than UNIX. For example, Smith's disk cache study reaches conclusions similar to ours [12], even though his study used IBM mainframes and was based on physical disk blocks rather than logical file accesses. Rodriguez-Rosell determined in [9] that database systems also exhibit sequential access patterns.

The generality of our conclusions is also supported by the similarity of the results for the three different traces. The results are similar in all three traces, even though one of the traces (C3) was for a substantially different application domain than the other two (computer-aided design as opposed to program development).

8. Conclusions

Our trace analysis of the 4.2 BSD file system has three important overall results. First, it shows that individual users don't use very much file data on average. This suggests that network bandwidth will not be a limiting factor in building network filesystems. Second, the analysis shows that most file data is deleted or replaced within a few minutes of creation. This is a key reason for the success of large disk block caches. The third overall result is that very large disk caches (many megabytes) with very large blocks (16 kbytes) result in very large reductions in disk I/O, and that occasional flush-backs provide safety against crashes without destroying the benefits of the large caches. As memory sizes approach 100 megabytes, we think that disk caches will become so effective that the whole role of magnetic disks comes into question: could write-once optical disks provide the same level of backup protection for less cost?

Our results also confirm several suppositions of operating system folklore: most files accessed are short, though long files account for a large fraction of the data transferred; accesses tend to be highly sequential; and file system activity is bursty.

Our final conclusion is that as block sizes become larger and disk block caches become more and more effective, I/O for things other than file data (paging, directories, and file descriptors) begins to play a larger role in determining overall file system performance. It appears from our data that more than half of all disk block references could come from these ‘‘other’’ accesses. There are indications that the other accesses can also be handled efficiently by caching, but more work is needed to understand their importance and to evaluate mechanisms for dealing with them.

9. Acknowledgements

We owe special thanks to Bob Henry, Mike Karels, Brad Krebs, and Richard Newton for allowing us to gather the trace data on their machines and for assisting us in installing an instrumented version of the kernel. The kernel modifications were based on a Master’s project by Tibor Lukac [3]. Luis Felipe Cabrera, Alan Smith, and the SOSP program committee provided helpful comments on early drafts of the paper. This work was supported in part by the Defense Advanced Research Projects Agency under Contract No. N00039-85-R-0269 and in part by the National Science Foundation under grant ECS-8351961.

10. References

[1] Feder, J. ‘‘The Evolution of UNIX System Performance.’’ *Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pp. 1791-1814.

[2] Lazowska, E.D. et al. *File Access Performance of Diskless Workstations*. Technical Report 84-06-01, Department of Computer Science, University of Washington, June 1984.

[3] Lukac, T. ‘‘A UNIX File System Logical I/O Trace Package.’’ M.S. Report, U.C. Berkeley, 1984.

[4] McKusick, M.K., Karels, M., and Leffler, S. ‘‘Performance Improvements and Functional Enhancements in 4.3 BSD.’’ *Proceedings of the 1985 Usenix Summer Conference*, Portland, Oregon, June 1985, pp. 519-531.

[5] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S. ‘‘A Fast File System for UNIX.’’ *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 181-197.

[6] 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.

[7] Porcar, J.M. *File Migration in Distributed Computer Systems*. Ph.D. Dissertation, University of California, Berkeley, July 1982.

[8] Ritchie, D.M. and Thompson, K. ''The UNIX Time-Sharing System.'' *Communications of the ACM*, Vol. 17, No. 7, July 1974, pp. 365-375.

[9] Rodriguez-Rosell, J. ''Empirical Data Reference Behavior in Data Base Systems.'' *IEEE Computer* , November 1976, pp. 9-13.

[10] Satyanarayanan, M. ''A Study of File Sizes and Functional Lifetimes.'' *Proc 8th Symposium on Operating Systems Principles*, 1981, pp. 96-108.

[11] Smith, A.J. ''Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms.'' *IEEE Transactions on Software Engineering*. Vol. SE-7, No. 4, July, 1981, pp. 403-417.

[12] Smith, A.J. ''Disk Cache — Miss Ratio Analysis and Design Considerations.'' *ACM Transactions on Computer Systems*, August 1985, pp. 161-203.

[13] Thompson, J. ''File Deletion in The UNIX System: Its Impact of File System Design and Analysis.'' CS 266 term project, Department of EECS, University of California, Berkeley, April 1985.