# 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 activity in trace files and writing programs to analyze the traces. 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 short, are open a short time, and are accessed sequentially. Most new information is deleted or overwritten within a few minutes of its creation. We 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 by about 50%, but larger caches (several megabytes) can achieve much greater reductions, eliminating 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. 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 many workstations can one network support?
- What are typical file access patterns (and what protocols will support those patterns best)?
- How should disk block caches be organized and managed?
- Current memory prices make it practical to use very large caches for disk blocks on file servers. How much of a performance advantage do such caches provide?

To answer these questions we instrumented the 4.2 BSD system to collect information about file accesses and save the information in trace files. Section 3 describes how and what trace data was gathered. In order to reduce the size of the trace files and the impact of the tracing on the running system, we did not record individual read and write requests. From the information that we did collect we can deduce the exact range of bytes accessed, although the access times are less precise than they would have been if we had logged reads and writes.

We wrote two kinds of programs to process the trace files; Table I summarizes the most important results. The first set of analysis programs gathered general data about reference patterns. Section 4 discusses these results. Some of the conclusions

| |
|---|
| On average, about 300-600 bytes/second of file data are read or written by each active system 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 (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 a 4-Mbyte cache, a 16-kbyte block size is optimal. |

**Table I.** Selected results.

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 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).

The second set of analysis programs performed simulations of various disk block caching strategies, using the trace data to drive the simulations. This analysis is presented in Section 5. 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 by about a factor of two. But much larger caches of several megabytes perform even better, reducing disk traffic by more than 90%. With large caches and the delayed-write policy described in Section 5, 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 even greater reductions in disk I/O.

Even for relatively small caches, large block sizes are effective in reducing disk I/O.

## 2. Previous Work

There have been a number of previous projects to analyze file system usage, but all of them were more limited in scope than the results reported here. Furthermore, many of the measurements concerned older systems without good interactive facilities or hierarchical directory structures, so the results are not necessarily comparable to ours. For example, Smith has studied the file access behavior of IBM mainframes in order to predict the effects of automatic file migration [10]. The study was based on files used by a particular interactive editor, which were mostly program source files. The data were gathered as a series of daily samples 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 included less than 10% of all files accessed. Satyanarayanan analyzed file sizes and lifetimes on a PDP-10 system [9], but once again the study was made statically by scanning the contents of disk storage.

More recently, Smith used trace data from IBM mainframes to predict the performance of disk caches [11]; his conclusions are similar to ours even though his data was different (physical disk addresses, no information about transfer sizes or reading versus writing). Two other recent studies contain UNIX measurements comparable to ours: Lazowska et al. report on the disk I/O required per user [2], and Leffler et al. report on the effectiveness of Unix disk caches [3]. See later sections of this paper for comparisons between their results and ours.

## 3. Gathering the Data

The main difficulty in gathering file system trace information is the volume of data generated. The systems we traced are heavily used and have a great deal of file system activity. If we had attempted to record all file-system-related 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 [11]. The work involved in writing such a trace file would consume a substantial fraction of the CPU. It might potentially 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 disk space for the traces.

For this study we wished to gather data over several days to prevent temporary unusual activity from biasing the results. We also wanted to collect the traces without affecting the system's responsiveness to its users. To do this, we had to reduce the volume of data.

Our approach was 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 and the information that was recorded for each event. "Logical" level means that information was recorded about 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. Once we decided to gather information at a logical level, we could take advantage of the fact that file reading and

| System Call | Information Recorded |
|---|---|
| open and create | time, open id, file id, user id, file size |
| close | time, open id, 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 call; it is used to avoid confusion between concurrent accesses to the same file. *File id* is unique to each file and is used to correlate different operations on the same 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).

writing in UNIX are implicitly sequential (a special system call must be used to change the current 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.

Our trace analyses consider both user- and system-initiated file access, but they examine only the actual bytes contained in files. We did not consider overhead I/O activity to interpret pathnames or to read and write file descriptors. We also ignored paging activity. Section 6 discusses how these other factors might impact our results.

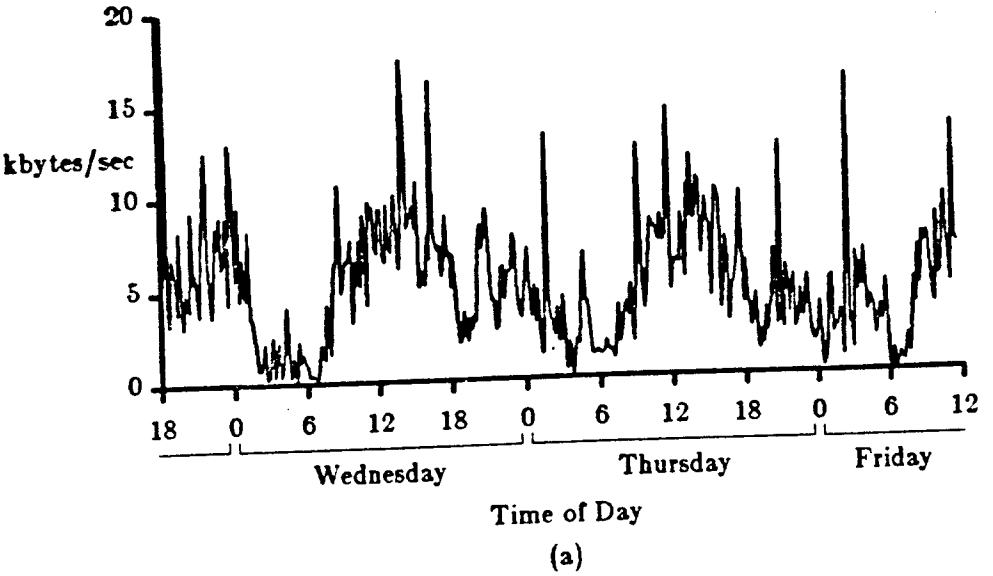
| Trace | A5 | | E3 | | C4 | |
|---|---|---|---|---|---|---|
| Duration (hours) | 79.4 | | 65.7 | | 72.5 | |
| Number of trace records | 1,017,464 | | 921,526 | | 733,403 | |
| Size of trace file (Mbytes) | 26 | | 23 | | 18 | |
| Total data transferred to/from files (Mbytes) | 1220 | | 1196 | | 1030 | |
| 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.

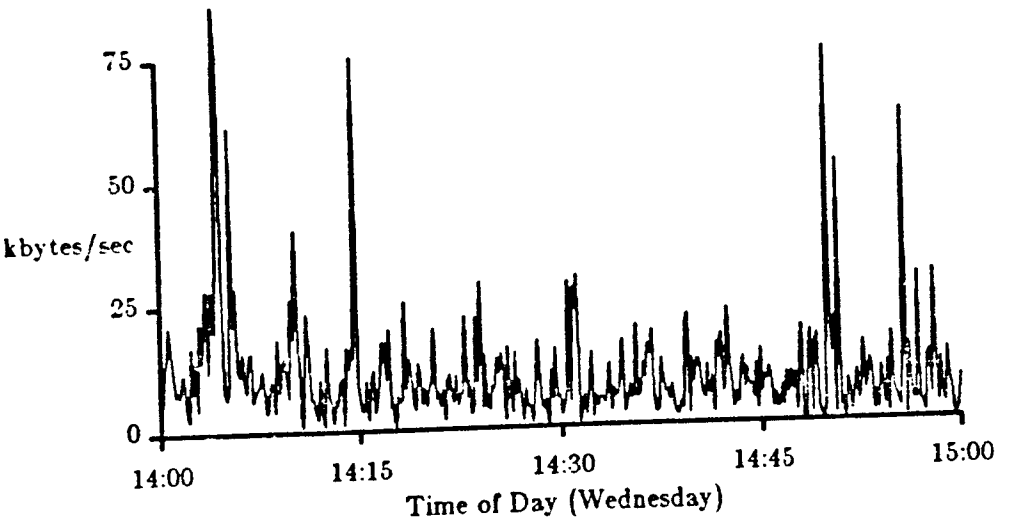
We used three different traces for the results reported in this paper. Two of them, "A5" and "E3", were recorded on VAX-11/780s used primarily for program development and document formatting. The third trace, "C4", was recorded on a VAX-11/780 used primarily for computer-aided design of integrated circuits. Each of these machines is a multi-user timesharing computer used by students and staff in a university environment. The traces were all gathered during the busiest part of the work week. For the A5 and E3 traces, the Unix load factor was typically 5-10 during the afternoon. For the C4 trace the load factor rarely exceeded 2 or 3. Table III gives summary information about the traces. The About 5000-6000 bytes of trace data per minute were collected, on average. Although the worst-case rate was higher than this, there was no noticeable degradation in the performance of the systems while the traces were being gathered.

## 4. 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 examines the effectiveness of disk block caches; Section 5 presents those results.




(a)




(b)

**Figure 1.** Chronological plot of the total rate at which bytes were transferred to/from files in trace E3. In (a) the throughput was averaged over ten-minute intervals. In (b) the throughput was averaged over ten-second intervals to illustrate the burstiness of the system.

### 4.1. System Activity

The first set of measurements concerns overall system activity in terms of users, active files, and bytes transferred. Figure 1 shows how the I/O demands on the file

system varied over time during the E3 trace. The average file system throughput during the day was about 5-10 kbytes per second, but Figure 1(b) shows that transfers were bursty.

Table IV gives several other measures of system activity. The most interesting figure 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. Averaged over ten-minute intervals, active users tend to transfer only a few hundred bytes of file data per second. On the other hand, if only ten-second intervals are considered, active users tend to have much higher transfer rates (a few kilobytes per second per user). Furthermore, the short-term transfer rates tend to be very bursty: in some intervals

| | A5 | E3 | C4 |
|---|---|---|---|
| Average throughput (bytes/sec. over life of trace) | 4200 | 5080 | 3940 |
| Total number of different users | 137 | 331 | 169 |
| Greatest number of active users in a 10 minute interval | 29 | 44 | 20 |
| Average number of different users (over 10 minute intervals) | 11.7 (± 5.8) | 18.7 (± 10.1) | 7.4 (±4.1) |
| Average number of different 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 minute intervals) | 370 (± 290) | 280 (± 190) | 570 (±760) |
| 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 or file is active in an interval if there are any trace events for that user or file 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 I/O per second during that interval, on average.

the rate exceeded 100 kbytes/sec/user. In [2] Lazowska et al. reported about 4 kbytes of I/O per second per active user. This is substantially higher than our figure, but their measurement includes additonal overhead not present in our analysis, such as paging I/O.

The low average throughput per user suggests that a network-based file system using a 10 Mbit/second network can support several hundred active users without saturating the network. For communities of this size it shouldn't be necessary to resort to multi-network systems with their more complex internetwork protocols.

| | 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 is 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. The only category where there was a substantial amount of non-sequential access was files opened for read-write access.

### 4.2. File Access Patterns

In doing this study, we were particularly interested in measuring the sequentiality of file access. It seems to be generally accepted that most file access is sequential. For a network file system, this suggests a pipelined protocol between clients and servers, with extensive prefetching.

Table V and Figure 2 summarize our measurements of sequentiality. Table V shows that more than 90% of all files are processed sequentially, and that 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.

Figure 2 measures the lengths of sequential runs in two ways. Figure 2a 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 2b shows that long sequential runs account for much of the data transferred: 30% of all bytes are read or written in sequential runs of 20 kbytes or more. If a file system uses large disk blocks (e.g. 4 kbytes), it appears that only a very small number of files will benefit from additional file system mechanisms to support sequential access, such as contiguous block allocation and prefetching. On the other hand, the benefit to those few files may be substantial enough to justify the mechanisms.

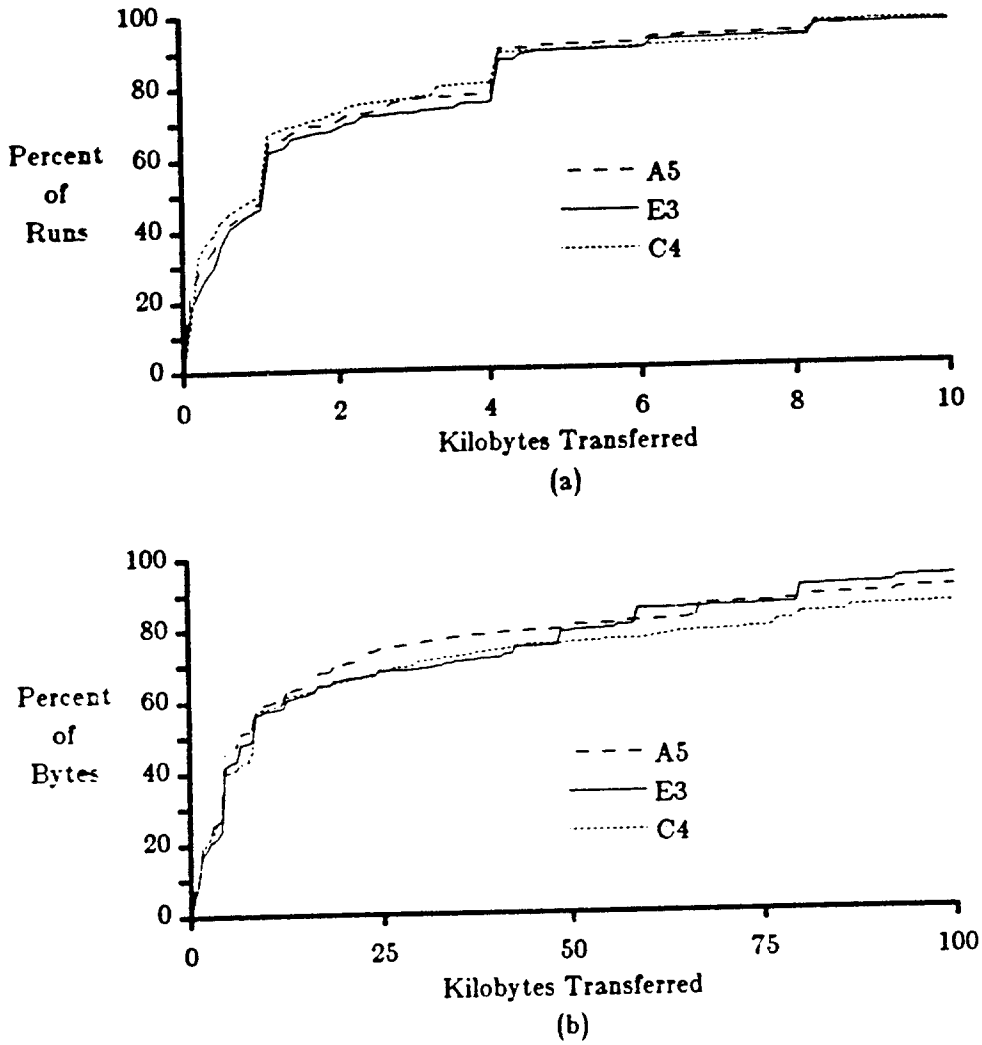



**Figure 2.** Cumulative distributions of the lengths of sequential runs (number of bytes transferred before repositioning or closing the file). Figure (a) is weighted by number of runs: about 70-75% of all sequential runs were less than 4000 bytes in length. Jumps occur at 1024 bytes and 4096 bytes 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.

Figure 3 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.

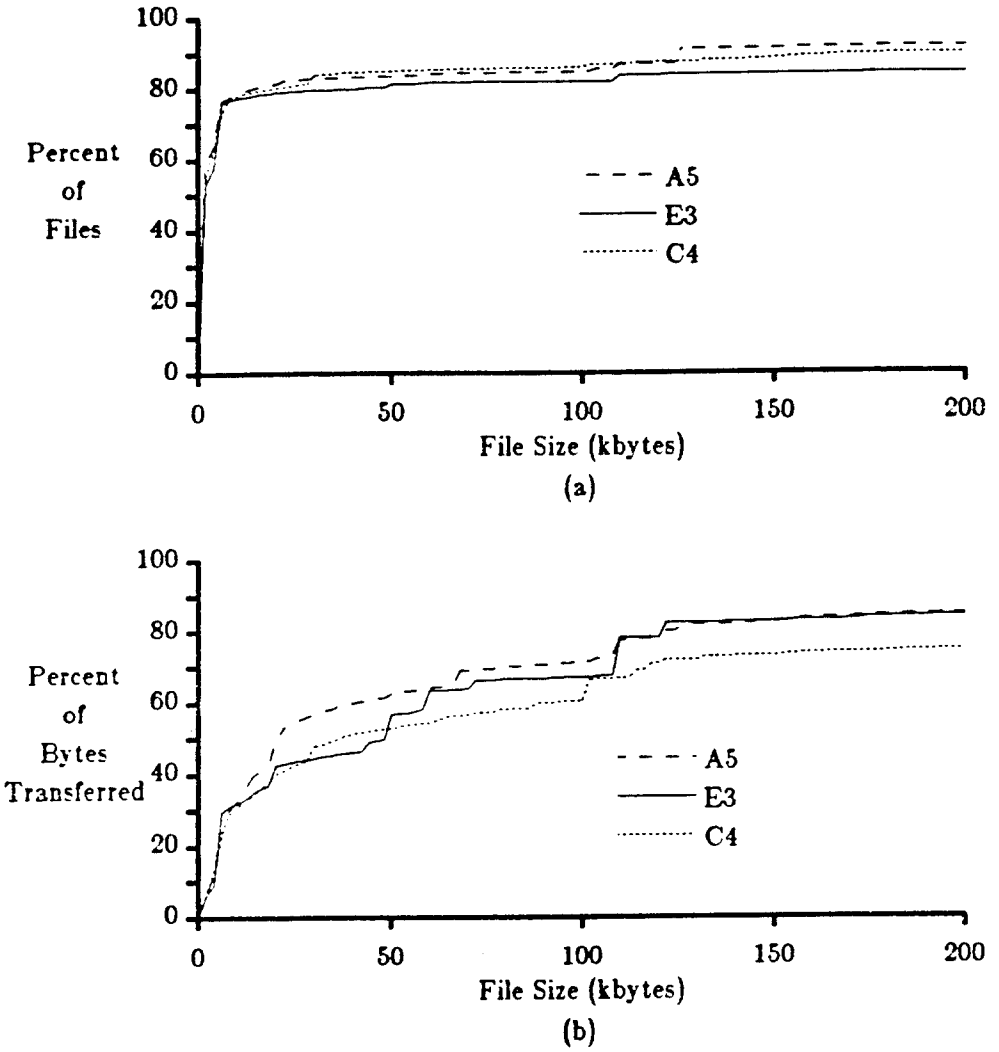



**Figure 3.** Dynamic distribution of file sizes, measured when files were closed. Figure (a) is a cumulative distribution weighted by number of files. 80% of all file accesses were to files less than 10 kbytes long; most of the remaining 20% were to a few very large administrative files. Figure (b) 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).

The file sizes shown in Figure 3 are much shorter than those measured for IBM systems in [7] and [10]. This difference is probably 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 measurement was a static one and his system did not have hierarchical directories [9].

Our last measurement of access patterns is displayed in Figure 4. 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.

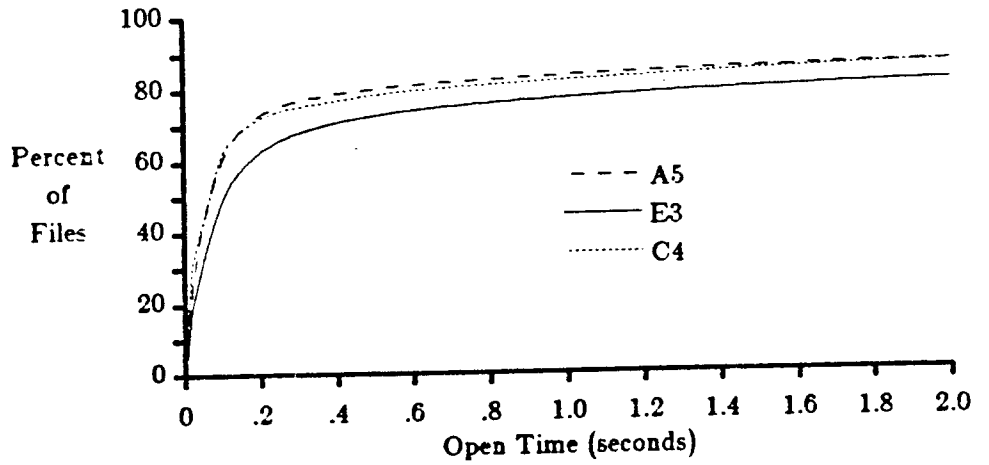



**Figure 4.** Distribution of times that files were open. This is a cumulative distribution. For example, about 70-80% of all files were open less than .5 second.

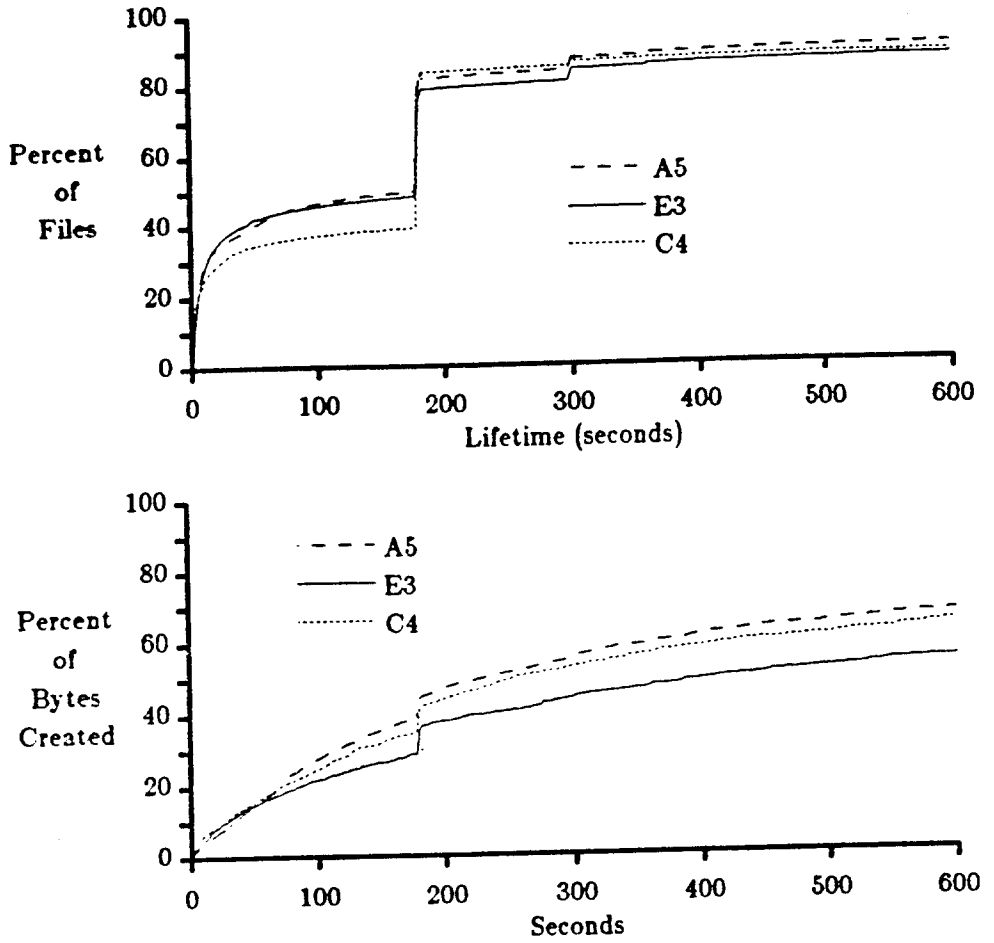



**Figure 5.** Cumulative distributions of file lifetimes. Figure (a) is weighted by number of files (about 80% of all new files had lifetimes less than 200 seconds). Figure (b) is weighted by the size of the file (about 40% of all bytes written to new files had lifetimes less than 200 seconds). The large jumps at 180 seconds are due to network monitoring programs: a separate status file for each host on the network is updated every three minutes.

### 4.3. File Lifetimes

Both Satyanarayan and Smith have published measurements of file lifetimes (the intervals between when files are written and they are overwritten or deleted) [9,10]. 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 5 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. 40-50% of all new files have lifetimes between 179 and 181 seconds. This concentration is due primarily to network status daemons that update information files every three minutes. One file is used for each of about 20 hosts on the net.

Figure 5 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 a existing file. Section 5 contains another lifetime measurement that is more inclusive but reaches about the same conclusion.

## 5. Block Cache Simulations

In considering various design alternatives for a network filing system, one of the most interesting possible areas of change is the cache of disk blocks. The UNIX file system keeps in memory 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. 4.2 BSD systems typically use about 10% of main memory (200-400 kbytes) for the disk cache.

For a network filing system with dedicated file servers it seems reasonable to use almost all of the servers' memories for disk caches; this could result in caches of four megabytes or more with today's memory technology, and perhaps 32 or 64 megabytes in a few years. The cost of such a cache is modest in comparison to the cost of disk storage. In order to understand the effects of larger caches on the performance of a file system, we wrote a program to simulate the behavior of various kinds of caches, using the trace data to drive the simulations. For all of the measurements below the three traces produced 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. All of the results of this section are based on blocks, not bytes. 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 number of disk accesses required. The fewer disk accesses, 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.

For some of the results we measured the *miss ratio* instead of the number of disk accesses. We define the miss ratio as the ratio of disk I/O operations to block accesses. 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, many programs make smaller requests than these, which result in lower miss ratios than we have reported (there will be many more block accesses but about the same number of disk I/Os).

The simulations varied in three respects: cache size, write policy, and block size. Figure 6 shows the effect of varying the cache size and write policy with a block size of 4096 bytes (the most common size in 4.2 BSD UNIX systems). We tried four different write policies in the simulations. One possible 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 an attractive policy because it ensures that the disk always

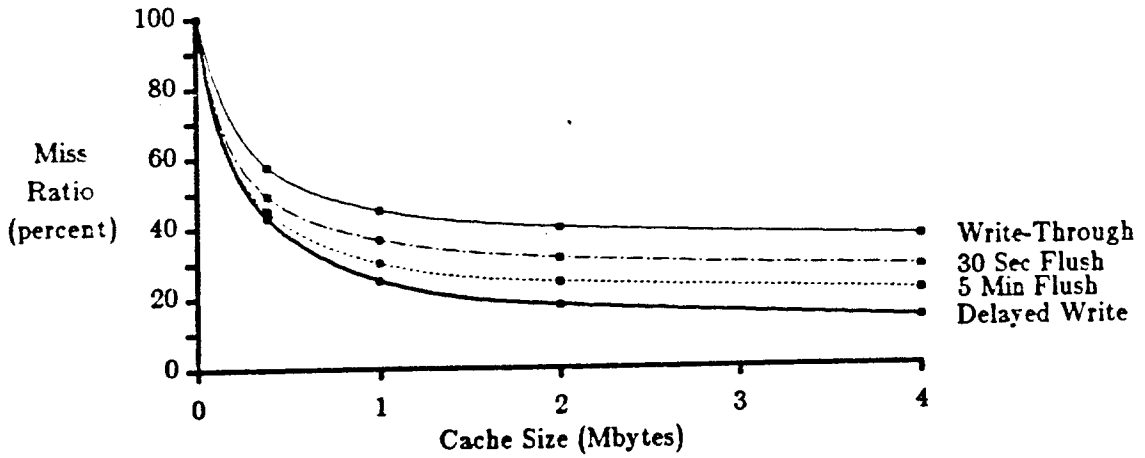



**Figure 6.** Cache miss ratios as a function of cache size and write policy, using the A5 trace with a cache block size of 4096 bytes.

| 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 6 (miss ratio as a function of cache size and write policy for the A5 trace).

contains an up-to-date copy of each block. Unfortunately, it has the worst performance of all the write policies since it requires one disk access for every write to a block. In our traces, 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 dramatically better performance for large caches. With a cache size of several megabytes, miss ratios as low as 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 (see Figure 7). 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

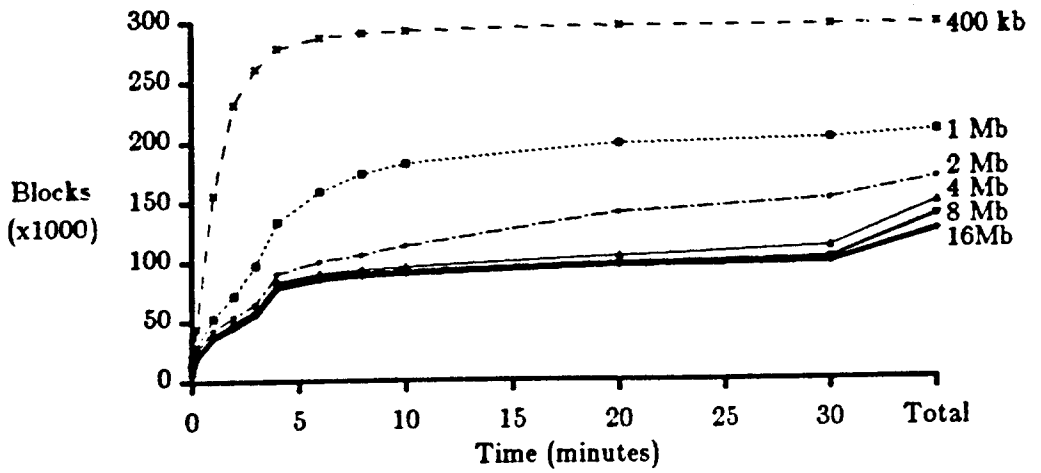



**Figure 7.** Cache lifetimes from the A5 trace using 4096-byte blocks and a delayed-write policy with different cache sizes. A block's cache lifetime is measured from when it enters the cache until it is ejected from the cache or the block is deleted from its file. The plot is a cumulative distribution: for example, with a cache size of 1 Mbyte about 150,000 blocks remained in the cache less than 6 minutes. The column labelled "Total" shows the total number of blocks deleted or ejected from the cache. For small cache sizes, most blocks are ejected within a few minutes to make room for other blocks; with larger cache sizes, blocks tend to remain until deleted. For the large caches, about 20% of all blocks had lifetimes greater than 30 minutes.

call these *flush-back* policies. With a flush-back policy the cache is scanned at regular intervals and 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 6 shows two different flush-back intervals: 30 seconds and 5 minutes. For large caches, a 30-second flush-back policy reduces the number of writes by about 25% and a 5-minute flush-back policy reduces the number of writes 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 very similar to the results of Figure 5.

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% [3]. There are two explanations for the discrepancy. First, there are several programs that make I/O requests in small units instead of the cache block size; this inflates the number of logical I/Os and reduces the miss ratio. Second, the measurements in [3] include block accesses for additional information that we did not consider, such as file descriptors (see Section 6). The overhead accesses may have greater locality than the accesses to file data.

Our final cache measurement evaluates the effectiveness of different block sizes. The original UNIX system used 512-byte blocks, but the block size has grown since then to 1024 in AT&T's System V [1], and 4096 in most 4.2 BSD systems. Figure 8 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. For smaller caches, larger block sizes are less beneficial since they result in fewer blocks in the cache and hence more frequent re-use of those blocks. As the block size gets larger, much of the cache

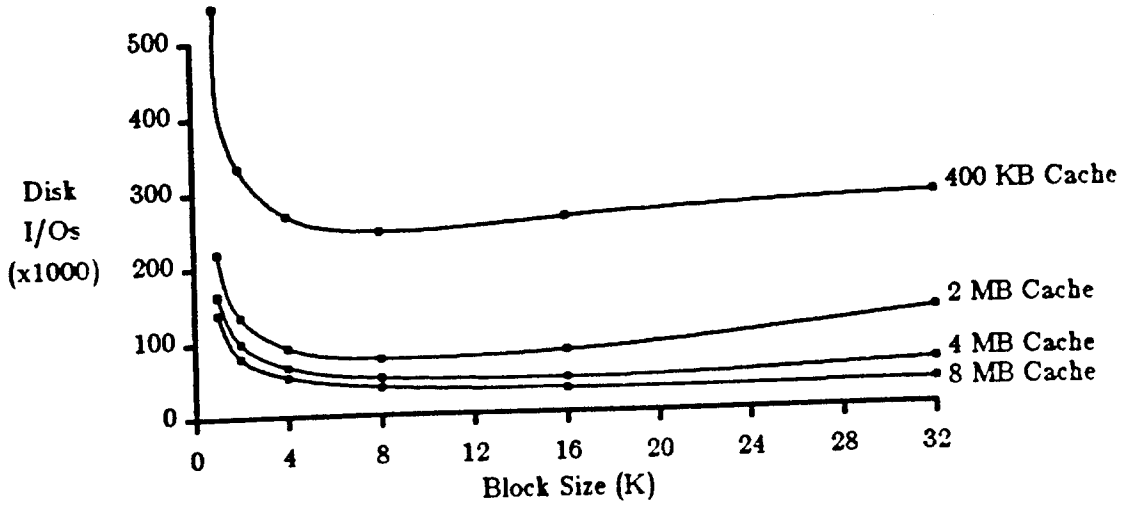



**Figure 8.** Disk traffic as a function of block size and cache size, for the A5 trace using the delayed-write policy. Large block sizes work well for small caches, but they work even better for large caches. For very large block sizes, the curves turn up because the cache has too few blocks to function effectively as a cache.

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

**Table VII.** A tabular representation of the data from Figure 8 (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.

space ends up being wasted (short files only occupy the first portion of their blocks).

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

## 6. Validity of the Approach

There are four potential problems with our trace-based analysis, all having to do with missing or incomplete information in the traces. The first potential problem, which was mentioned in Section 3, is the no-read-write approach. In addition, the traces provided only partial information about file accesses used to page-in programs, and they provided no information about disk accesses for directory lookups or file descriptors. These issues are discussed in the following subsections.

### 6.1. No Reads and Writes

The no-read-write approach means that the traces contain imprecise information about when bytes were transferred to and from files. For all of our analyses we billed each data transfer at the time of the file's next close or reposition event following the transfer. We measured the potential inaccuracy by examining the intervals between successive trace events for the same open file. The intervals provide bounds on when data transfers actually occurred. 75% of the intervals were less than .5 second, 90% were less than 10 seconds, and 99% were less than 30 seconds. Our measurements were averaged over periods much longer than most inter-event intervals. For example, ten seconds was the smallest interval used in Section 4. In the cache simulations, the cache lifetimes of blocks were typically several minutes or more, so once again the time imprecision should not have biased the results.

## 6.2. Paging Activity

Paging activity in UNIX consists of reading in a program file from the file system when a process begins execution, and swapping pages of the process between main memory and backing store while the process executes. The backing store is handled with a totally different mechanism than the file system, so none of our measurements concern it. Both [2] and [6] report that I/O to and from the backing store is infrequent in comparison to I/O to and from files.

When a program is run in 4.2 BSD UNIX, only the first block of the program file is read immediately; the other blocks are paged in from the file when they are first referenced. Our trace data logged the *execve* operation but not the individual page-ins, so it was difficult to determine what effect the page-in activity might have on the file system performance. All of the measurements reported until now have excluded paging accesses.

Table VIII shows that the amount of file system activity would be more than doubled if every executed file were read in its entirety from disk. To get some idea of the effects of paging activity on cache performance, we re-ran the cache simulations

| | A5 | E3 | C4 |
|---|---|---|---|
| File I/O (Mbytes) | 1,220 | 1,196 | 1,030 |
| File I/O (4k-byte blocks) | 450,480 | 392,913 | 516,845 |
| Exec I/O (Mbytes) | 1,879 | 2,382 | 1,231 |
| Exec I/O (4k-byte blocks) | 494,554 | 618,978 | 323,458 |

**Table VIII.** If every executed file must be read from disk in its entirety, the page-in activity will account for more I/O activity than regular file I/O. The "Exec I/O" lines make this assumption. In practice, however, sharing and demand paging will reduce these figures.

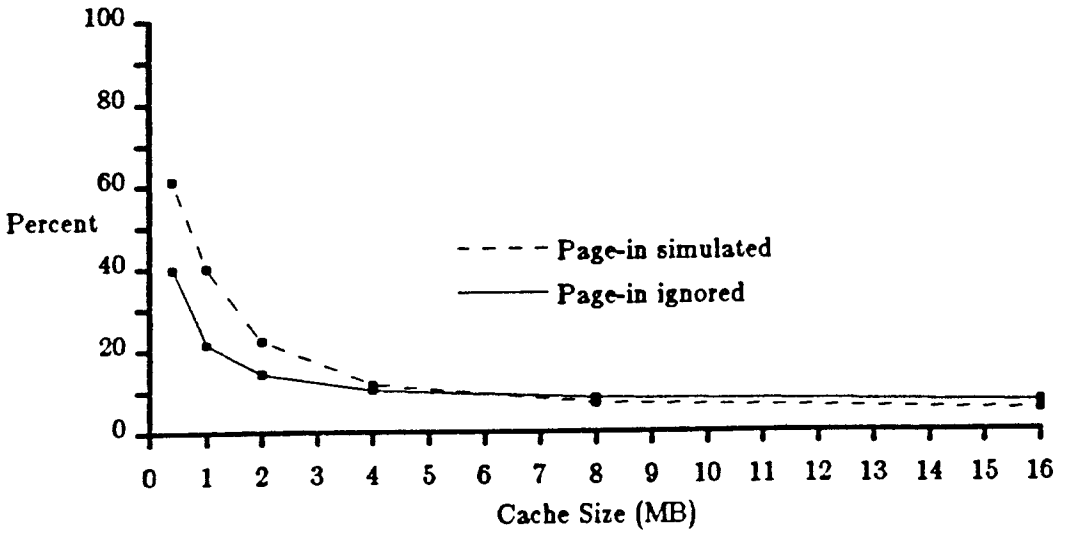



**Figure 9.** Miss ratios (4096-byte blocks, delayed write, trace A5) with paging behavior approximated by forcing a whole-file read of each program that is executed. The dashed line includes executed files, while the solid line does not.

and forced a whole-file read to each program at the time of its *execve*. Figure 9 shows the results. At large cache sizes the performance is slightly better (in terms of miss ratio) with the paging activity. At intermediate cache sizes the performance is somewhat worse with the simulated paging activity, due to a larger working set of disk information.

However, the above analysis is questionable due to UNIX's handling of program files. First, UNIX provides for shared code segments: if there is already one copy of a program in use then the system will re-use that copy for a new instance of the program and will not have to access disk. In a sense, all of main memory acts as a cache for programs. Second, program files may also contain large amounts of debugging symbol information, which will never be read. The debugging information can be many times larger than the code and data portions of the file. Third, program

files are paged in on demand, so pages that are not referenced will not be read. These factors all suggest that the effects of paging will be less than indicated above; more work is needed to verify this.

### 6.3. No Descriptor Blocks

We measured only the *logical* behavior of the file system, based on transfers of bytes to and from areas of files. Thus the measurements do not include disk activity for reading and writing file descriptors (*i-nodes* and *indirect blocks* in UNIX terminology). In UNIX, one file descriptor access may have to be made for each open and each close. In the worst case, this could more than double the number of block accesses (this can be inferred from Tables III and VII). An i-node cache is kept in main memory to reduce the number of disk accesses required for them; we do not have any measurements of its effectiveness.

Indirect blocks are used to hold the disk maps for large files; they are not needed for files less than 40 kbytes long. Figure 3 shows that only about 20% of all files accessed are longer than 40 kbytes. Only extremely large files need more than a single indirect block, and for sequential file access the indirect block(s) are very likely to remain in the cache. Thus, indirect blocks are unlikely to have a large impact on any of our measurements.

### 6.4. No Directory Lookups

When a file is looked up in UNIX, the file system reads the blocks of directories along the path to the file. In general, this involves one file descriptor block and one

directory file block for each name along the path, with a minimum of two blocks for each file looked up. However, 4.2 BSD now contains a directory cache to hold recently-used directory entries. Leffler et al. report that the directory cache achieves an 85% hit ratio [3]. Nonetheless, directory lookups appear to account for a substantial fraction of all file system activity.

## 7. Summary

In many ways, our results confirm operating system folklore and other file system studies. For example, we were not suprised that most file accesses are to short files, nor were we surprised that most accesses are sequential. Smith's study for disk caches agrees with our conclusion that multi-megabyte caches are effective [11], even though his study was based on physical disk blocks rather than logical file accesses.

However, a few of the results were surprising to us. First, the average throughput per user was lower than we had expected and suggests that industrial and university organizations of the size of a department or laboratory can be supported on a single network without using internetwork protocols for file access. Second, although the advantages of large block sizes have been known for some time, we were surprised at just how advantageous they are. The 1024-byte blocks used in AT&T's System V result in many more disk accesses than the 4096-byte blocks used in 4.2 BSD, and 8-kbyte or 16-kbyte block sizes look even better.

As block sizes become larger and disk-block caches become more and more effective, "overhead" accesses (to directories and file descriptors) 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 be overhead accesses. Program page-ins also appear to account for a significant amount of file traffic. More work is needed in both of these areas to understand their importance and to evaluate mechanisms for dealing with them.

## 8. 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 [4]. Luis Cabrera and Alan Jay Smith provided helpful comments on an early draft of the paper. This work was supported in part by the Defense Advanced Research Projects Agency under Contract No. N00039-85-R-0269.

## 9. 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] Leffler, S., Karels, M., and McKusick, M.K. *Measuring and Improving the Performance of 4.2 BSD.* Technical Report UCB/CSD 84/218, Department of EECS, University of California, Berkeley, December 1984.

[4] Lukac, T. "A UNIX File System Logical I/O Trace Package." M.S. Report, U.C. Berkeley, 1984.

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

[10] 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.

[11] Smith, A.J. "Disk Cache — Miss Ratio Analysis and Design Considerations." *ACM Transactions on Computer Systems*, to appear, 1985.