to appear, Proc. 1984 USENIX Summer Conference, Portland, Oregon, (Hosted by University of Oregon), June 12-14, 1985

A File System Tracing Package for Berkeley UNIX*†

Songnian Zhou, Hervé Da Costa, and Alan Jay Smith

Computer Science Division
EECS Department
University of California
Berkeley, California 94720

Abstract

A tracing package for the UNIX file system has been implemented and statistics have been gathered from a heavily and widely used DEC VAX 11/780 running UNIX 4.2BSD. This tracing package is unusual in the comprehensiveness of the data gathered, the clean and usable format in which the final trace appears, and the use of a post processing step to assemble information in trace records that is not easily (or at all) available at trace time. Trace records are gathered for file opens, file creates, file closes, reads and writes, renames, file deletes, executes, forks and exit calls. Some preliminary analyses of the trace data are presented. We found that the I/O activities are very bursty, that very few read and write operations are performed in most of the open-close sessions, and that the process lifetime distribution is highly skewed, with many short lived processes and a few long term ones. The extensive data gathered using the package is valuable for the studies of disk caching and file migration algorithms, distributed file system performance, and load balancing strategies.

*The material presented here is based on research supported in part by the National Science Foundation under grant DCR-8202591 and by the Defense Advanced Research Projects Agency under contract N00039-82-C-0235. Partial support has also been provided by the Digital Equipment Corporation Eastern Research Laboratory.

†Unix is a trademark of Bell Laboratories

1. INTRODUCTION

Over the past years, Berkeley UNIX has undergone extensive modifications and enhancements in its functionality and performance. In particular, the file system was largely reimplemented in 4.2BSD to provide features such as larger file blocks and contiguous block allocation [Mcku84], yielding considerably better performance. With the increasing complexity of the system, it has become more and more difficult to fully understand its behavior, and to identify performance deficiencies and bugs by reading the source code. A comprehensive trace-driven analysis of input/output activity can not only reveal many aspects of the system and user file access behavior, thus indicating ways to improve system performance, but also will provide invaluable data for the design of and research into disk caches, file migration, network file systems, and load balancing strategies.

In this paper, we describe a logical file system tracing package designed and implemented for the Berkeley UNIX system. We also present some preliminary results from analyses of the trace data generated using this package. Several goals and requirements for the package were set up to guide its design and implementation, and they are discussed below.

1) **Comprehensiveness**. The tracing package should provide a clear and detailed picture of the file system activities. For this purpose, we need to trace all the relevant file operations, and collect all the useful information. Efforts must be made to generate complete and accurate information in order to avoid as much guess work as possible in the analyses of the trace. The trace data should be useful for a wide range of analyses and simulation studies, many of which were unforeseen at the time the package was developed.

2) **Flexibility**. The package should be flexible enough to tailor to different tracing needs. Besides tracing the complete system, it should be possible, by specifying a number of parameters when activating the trace, to trace a single user, a group of users, a single process, such as a system daemon, or a group of processes. One should be able to activate and deactivate the trace dynamically without disruption of system services. The package should also be able to trace long term system behavior, as well as short term behavior.

3) **Minimum performance impact**. While a trace must by its nature generate some overhead, such a penalty should be kept to a minimum. When the trace is not activated, there should be almost no extra system overhead caused by the tracing package, and when activated, the tracing should not cause noticeable degradation to the system performance. There are two aspects to this: First, the extra amount of computing for extracting data from the system should be kept low. Second, the amount of trace data generated should not be so large as to constitute a significant portion of the system file activity. Both aspects are important if the trace is to run for an extended

period, say several days, in order to eliminate the measurement error caused by temporary variations in system and user behavior.

4) **Minimum change to the system**. It is important to minimize the amount of code required for the tracer and to also minimize the degree of modification to the system. This not only makes the package easy to maintain and reduces the probability of the package introducing errors, but tends to reduce the performance impact on the system.

5) **Convenience for Analysis**. The trace data, in its final format, should be usable; i.e. it should be possible to analyze it easily and simply, with only simple computations required for simple outputs. Complex cross correlation of trace records should be done as part of the trace preparation, and should seldom be needed for trace analysis.

It is recognized that the above requirements often conflict with each other. However, our experience indicates that, by careful design and implementation, such conflict can be reduced and a reasonable compromise achieved. In the following sections, we will discuss how these requirements are met. The design of the package, including its general structure and trace data content, is described in Section Two. Section Three is a detailed discussion of the various issues involved in the implementation. We present some analyses results of the trace data in Section Four; potential uses of the package are also discussed there. The impact of the tracing package on system performance is quantitatively studied in Section Five. Section Six concludes this paper with an evaluation of the package.

2. DESIGN ISSUES

2.1. Design Considerations

The design of the tracing package follows the principle of achieving the functionalities required with minimum possible complexity. We first discuss the level at which the tracer runs and the reasoning behind our decision. This is followed by a general description of the package structure.

2.1.1. Level of tracing

Selecting the level of tracing is the first step. We want to trace the file operations performed by the users and system processes; this can be accomplished on top of the UNIX kernel. There are, however, several ways a user can perform a file operation: he can make a system call directly, or can use the system utility routines. This implies that the hooks for catching the operations must be placed in a number of places in order to capture all operations of the same type. Even so, there is still the risk of missing some of them. Rather than performing the tracing at the user level, we decided to do it inside the kernel. UNIX has a small

set of well defined system calls for handling all file operations, such as open, close, read, write, and rename. These calls provide a clean interface used by every file operation, thus enabling us to record all the operations by placing exactly one hook for each type of relevant system call.

There are two disadvantages to tracing at the system call level, however. First, by operating inside the kernel, the package has no way to tell where the file operation comes from. We remedy this by tracing the command that caused this operation, and matching the two records together later. Another problem is that the kernel has to be changed and debugging the kernel is generally much harder than debugging a user program; an error in the tracing package can potentially crash the system. We attempted to ease this problem by minimizing the changes to the kernel and by careful implementation. (This would have been easier had we been the only people attempting to modify the kernel at the same time.) Our results show that placing the tracer in the kernel is not a serious problem compared to the benefits achieved by an in-kernel tracing package.

Our interest in the trace data is primarily toward studying user behavior and related research issues such as disk caching, file migration, etc. Conversely, we are not strongly interested in "tuning" the system. Thus, our data collection is directed toward the collection of "logical" I/O information rather than "physical", and we want the logical read/write requests and not the physical block transfers to and from disk. Because of the complexities of tracing physical activity, such complexities including the use of a disk block cache and write behind, we have chosen to trace only at the system call level, ignoring some of the implementation details of the file system.

2.1.2. Structure of the tracing package

The package collects data about the file operations and records them in trace files. The package is composed of four parts. The first part provides the mechanism to activate and deactivate the tracing and to switch tracing files when they grow too large. Because of the wide range of system calls traced and the comprehensive set of information recorded, a large amount of data is generated in an actively used system; the limited amount of system disk space requires that we "ping-pong" between output files and dump them to tape as they become full. The second part of the package includes all the hooks placed in the relevant system call handling routines, and the corresponding routines to generate the trace records. The third part is the buffer management, synchronization, and file dumping routines. In order to reduce overhead, the individual records are not written to the trace file right after they are generated; rather, they are first accumulated in a large buffer in memory and then dumped. Since it is possible that several processes are trying to access a buffer at the same time, synchronization has to be provided to enforce strict time order of events and to avoid overwriting

other trace records.

The last part of the trace package is not used during the generation of the trace, but rather at the *post processing* stage. Certain pieces of data are impossible to get during the trace, or are too expensive, in terms of processing time and additional data structures to be set up. Instead, we process the raw trace off-line by making several passes on it. The final trace generated from the raw trace and other information of the system is much easier to use by analysis programs. The details of the package implementation will be discussed in the next section.

2.2. Trace Data Content

2.2.1. What files are traced?

UNIX has a very general notion of files. Besides disk files, devices, such as terminals (*character special*) and disks (*block special*), are also treated as files and managed by the same naming mechanism. The inclusion of the IPC mechanism in 4.2BSD further extended the concept of files to include communication end points called *sockets*. Since we are primarily interested in the file system proper, all these special "files" are ignored by the trace package.

2.2.2. What activities are traced?

We trace the file **open**, file **close**, **read**, **write**, file **rename**, and file **delete** calls. In addition, three process control calls, **fork** (**vfork**), **exec**, and **exit** are also traced. Besides ordinary data files, there are also a few other types of files and file system internal data blocks stored on disk, including directories, symbolic links, and the file descriptors, called *inodes*. An inode contains information about a file, such as its owner, its size, its disk block addresses, and a unique number called *i-number*. In order to operate on a file, its inode must first be brought into the memory.

The files in UNIX are organized in a hierarchical structure. When a user makes an **open** call, s/he specifies the name of the file. Directories must be searched to find its inode. Because the way the directories are used is very different from ordinary data files and is not through the same set of system calls, tracing the directories would increase the complexity of the package by a substantial extent. Also, many directory accesses can be inferred, given full path names of nondirectory files accessed. Based on these two considerations, we decided not to trace directory activities. Similar considerations led to the exclusion of the I/O's for file inodes and indirect disk pointer blocks of files. Another type of special files is symbolic links. They simply contain names of other files so that multiple directory entries in different file systems can share the same file descriptor. We decided to ignore the symbolic links. Executable files in UNIX are accessed

by the virtual memory system. The I/O operations on them are initiated internally by demand paging, rather than by explicit user requests. Although we trace the **exec** calls, which specify the executable files to be used, we do not trace the paging activities. (Paging activity can also, as an approximation, be inferred.)

In favor of simplicity, we ignored a number of I/O activities. This implies that a portion of the disk I/O is missing from the trace. Although this affects the accuracy of our data as a characterization of the system and user file operations, we feel that the most important and largest fraction of the I/O operations is still retained; much of the rest can be inferred with little loss in accuracy.

2.2.3. What information is recorded?

Table 1 is a list of the system calls traced and the data fields for each of them. Some of the data fields, such as the file close time in the open record, the complete file names, and the number of bytes transferred while a file is open, are impossible to derive during the tracing, at least at the proper time to insert them in the appropriate record. An important feature of the tracing package is the extensive amount of *post processing* performed on the raw trace to generate the richer and reformated final trace data. (A similar procedure was used to gather the data presented in [Smit85].) In the next section, we describe in detail the meaning of the data fields and the way they are derived, as well as the post processing.

3. IMPLEMENTATION ISSUES

We discussed the design of the tracing package in the last section, including the level of tracing, the package structure, and the data content. In this section, we consider a number of implementation issues in detail.

3.1. Synchronization and Buffer Management

Our scheme for buffer management is similar to the one used in an earlier file tracing package developed by Tibor Lukac [Luca83], and to the accounting mechanism available in 4.2BSD. As mentioned in the previous section, trace records are collected into a big buffer and dumped to the trace file in large blocks. When a system call of interest is invoked, the process enters a trace collection routine which allocates a section of memory from the trace buffer. If there is not enough space in the current buffer, a new buffer is requested from the system, while the old one is dumped and returned to the system buffer pool. Synchronization on the buffer must be provided, however, because the same buffer is shared by all the processes in the system. This turns out to be fairly easy because a process allocating space in the buffer is executing in the kernel mode, and therefore may only be interrupted by totally irrelevant system activities, such as the device

RECORD TYPE	DATA FIELDS
open/create	*record type, record length, real time, process time, file id, open id, process id, user id, open mode, device number, last modify time, last access time, user name, file size when open, file size when close, file type, close time, number of I/O, bytes transferred while open, reference count, file name*
close	*record type, record length, real time, process time, file id, open id, process id, user id, open mode, device number, file size, file type, number of I/O, bytes transferred while open, reference count*
read/write	*record type, record length, real time, process time, duration, file id, open id, process id, user id, offset, bytes transferred*
rename	*record type, record length, real time, file id, process id, user id, device number, last modify time, last access time, user name, file size, old file type, new file type, old file name, new file name*
delete	*record type, record length, real time, file id, process id, user id, device number, last modify time, last access time, user name, file size, file type, file name*
execute	*record type, record length, real time, process time, file id, process id, user id, device number, user name, file size, file name*
fork	*record type, record length, real time, process time, vfork flag, parent process id, child process id, user id*
exit	*record type, record length, real time, process time, process id, user id*

Table 1. Trace Record Types and Their Data Fields (after post processing).

driver. Since the space in the buffer is allocated in strict order of time, chronological ordering of event records is enforced.

3.2. File ID and Open-close Sessions

It is important for any meaningful analysis and understanding of the data that files have unique names. The path names of the files are not easy to use and are difficult to get during the trace. The i-number stored in a file's inode uniquely identifies the file within a file system; unfortunately, the i-number of a file is reused after the file is deleted, so it is only unique at a given time, and may not be unique for the duration of the trace. We decided to generate and assign unique

file ID's to files during the trace. A file ID (identifier) is a number that is never assigned to more than one file for the entire duration of the trace. When a file is seen for the first time by an **open**, **rename** or **exec** call, we mark it by setting a flag in one of the unused locations in its inode. We also give it a file ID and record the ID in its inode so that next time this file is accessed, we can use its ID to identify it. These file ID's are just consecutively assigned integers, and function as unique, easy-to-use internal names of files.

Operations on a file generally follow the basic pattern of open/create - read/write - close. We call this cycle an *open-close session*. Tying together the data for a session provides insight into file access patterns; we use an *open ID* for this purpose. When a file is opened, an open ID is generated and stored in its inode so that all the operations on this file can be tagged with this ID. When the file is closed, the open ID is erased and never reused; next time this file is opened, a different open ID will be generated. These open ID's are also just consecutive assigned integers. There are two complications, though. First, the same file might be opened by several processes concurrently. Clearly, they should be considered as separate sessions. However, because of the constraint of available space in the inode, this scheme requires that they share the same open ID. In this case, the open ID alone is not sufficient to specify a session; the process ID supplied by the kernel must be used as well. Secondly, the same process might open the same file several times before closing it, in which case even the process ID is not sufficient. We choose to regard the multiple operations as belonging to the same session, so the pair {process id, open id} can be used to provide a real unique session id. A field in the open record, the reference count, remembers how many opens have been performed on this file, thus giving an indication of the level of file sharing.

3.3. Fork, Exec and Exit

These three system calls do not describe file activity directly, but tracing them is necessary to correctly interpret the remaining data. For example, in UNIX, new processes are created by a **fork** or **vfork** call. All of the open files of the parent are inherited by the child, which means that it is possible for a process to open a file, do some I/O, and then fork a child process, which itself has a new process id. The child may do some more I/O and finally close the file. This sequence of file operations should be considered as a single open-close session, involving multiple processes. In order to recognize this sequence, the creation and destruction of processes must be recorded.

The **exec** call starts the execution of a new program. We would like to know which system command or user program initiated a particular open-close session, so that we can study the file I/O requirements and the CPU use of the various commands. This can be accomplished by tracing the **exec** calls.

Including these three calls in the trace greatly enhances its usefulness, since by looking at the trace, we know (almost) exactly what happened in the system. The CPU overhead caused by the generation of these call records is small and the amount of additional tracing code minor. For example, in a trace that we ran on a production system for a whole (working) weekday, the records of these three types constitute only 7.6% of the total number of trace records and 5.0% of the volume (bytes) of trace data generated.

The UNIX mechanisms of the inheritance of open files by child processes, the sharing of files by multiple processes, and the permissible multiple opens within one process make the recognition of an open-close session fairly complicated. By keeping track of process fork and exit calls, and by the use of open ID's, however, we have been able to identify all the sessions, thus making it possible to generate statistics such as the distributions of the number of I/O operations and the amount of data transferred in a session.

3.4. Complete File Names and File Types

We mentioned above that the user usually only provides partial file names; i.e. the user makes references relative to a working directory. Complete file names, however, are very useful; they often indicate the type and function of a file. For example, files in "/bin" are system provided utility programs, while files in "/tmp" are usually temporary files generated by system utilities such as the editors. File suffixes are also often informative. C language programs usually have the suffix ".c", while object files have ".o"; fortran files are identified by ".f". We have defined a number of file types based on file name prefixes and suffixes. File type information is generated during post processing and stored in the file open records of the final trace.

Complete file names are not easy to derive during the trace; instead, we decided to find them during the post processing. We record the parent directory information and the last component of the file name during the trace. Before and after the trace, we obtain a static picture of the file system. This information enables us to construct the complete names of most of the files by appending the last component to the name of the parent directory. The names of the files under directories that are created and destroyed during the tracing period cannot be constructed using this method. Since the directories are orders of magnitude more stable than data files, however, we do not believe we lose many file names. It would have been possible to construct all the complete path names had we traced the directory creation and deletion operations.

3.5. Times

There are two types of times in the trace records: the real time is the time since the trace started, and the process time is the total virtual time this process has consumed. All the times included in the trace are in milliseconds, but since the local VAX UNIX systems have a clock resolution of ten milliseconds, the times in the data collected are only approximate. The process time suffers from a larger error margin because it is derived by a sampling process—which ever process is caught executing at the end of a ten millisecond interval is charged with ten milliseconds of CPU time, no matter how long it actually spent running during the interval. Despite this error margin, both types of times are very valuable for studying file operations and process behavior. For example, the real time durations of the user read and write calls are recorded, and can be used to evaluate the effectiveness of the read ahead and delayed write strategies in 4.2BSD file system. Lifetime distributions of processes, both real and virtual, are useful information for load balancing studies.

3.6. Post Processing

Post processing is an indispensable part of the tracing package. Although the raw trace contains (somewhere) almost all of the information that appears in the final trace, laborious processing would be required in any (and every) analysis to understand and interpret the data, since as explained earlier, the raw trace was generated with minimal overhead and consequently minimal user friendliness. We have chosen to write a post processing program which does that laborious processing once and for all. Post process consists of two phases: First, the raw trace is parsed to generate *session records* for each open-close session. A session record contains statistics such as the number of reads and writes performed during this session, and the number of bytes transferred. The close time and the size of the file at its close are also recorded. Static pictures of the complete file system taken before and after the trace are also processed to extract all the directories and their device numbers and i-numbers. In addition, a table is set up relating user ID's to user login names. Using the above pieces of information, the second phase of the post processing converts the raw trace into the finished format. Specifically, complete file names, file types, user names, and session statistics are all included in the finished trace data.

4. TRACE ANALYSES

In the above sections, we discussed the design and implementation of the tracing package in some detail. We are using the package to trace several machines in the EECS department at UC Berkeley, and in this section, we present some preliminary analysis results from one such (heavily used) machine; further data collection and analysis is continuing. An earlier study of this same machine

appears in [Oust85], but relies on less complete and comprehensive trace data. Some of the measurements presented here include the distribution of the file I/O length and duration, the I/O transfer rate, the number of I/O operations within an open-close session, the process lifetime distribution and the distribution of the number of active processes.

4.1. General Statistics

The general statistics for the one-day tracing session are shown in Table 2. The machine was moderately to heavily loaded, with the load average (average number of ready processes) between 3 to 8, and sometimes as high as 12. The number of users ranged from 20 to 50. It should be pointed out that the data gathered are dependent on the system load characteristics, and will vary with the number of users and the type of work that they are doing; one cannot generalize from our data to different environments.

The distribution of the number of I/O operations per open-close session is highly skewed. For the files opened for read, 97% of them have less than 4 read operations. Eighty seven percent of files opened for write have only zero or one write operation. Roughly, two reads and one write are performed on the average during each session. The durations of open-close sessions are usually very short, on the order of one to a few hundred milliseconds. (70% of the sessions last less than 150 milliseconds, and only 17% of them last longer than 350 milliseconds.) The number of concurrently opened files is a few hundred under normal system load. The level of file sharing is fairly low. Of all the files opened, only about 8% are shared.

The observations given in the paragraph above illustrate a very important point: *the behavior of the system at a detailed and low level, such as that given by an I/O trace, is VERY hard to predict simply from a knowledge of what a typical user appears to do on the system.* It has been observed previously [Smit85] that the bulk of the I/O is generated by the system (only indirectly in response to user activity) and similar loose coupling between user behavior and system activity seems to also be present under Unix.

4.2. File I/O Transfer Sizes and Durations

4.2.1. File I/O transfer size and utility program behavior

Table 3 lists a few I/O sizes around which large numbers of the I/O operations concentrate; it is not a complete size distribution. Seventy-eight percent of reads and eighty-one percent of writes have sizes less than 40 bytes, and around 512 bytes, 1 KB, 4 KB, 6 KB and 8 KB. The percentage of reads with 512 bytes

Machine: ucbarpa, VAX-11/780, 4.2BSD
Friday Apr 26 8:43 to 6:29 pm (9 hrs, 46 mns)
amount of data read: 385,646,844 bytes
amount of data written: 204,970,844 bytes
amount of trace data generated: 21,821,604 bytes
number of shared files: 7,977 (~8%)
maximum number of active processes: 189

Record Type	Count	Record Type	Count
number of records	538,588 (100%)	delete records	6,337 (1.2%)
create records	6,533 (1.2%)	rename records	168(3 in 10,000)
open records	87,664 (16.3%)	fork records	7,633 (1.4%)
close records	94,153 (17.5%)	vfork records	7,023 (1.3%)
read records	220,579 (41.0%)	exec records	11,641 (2.2%)
write records	82,266 (15.3%)	exit records	14,591(2.7%)

Table 2. General Statistics of a Tracing Session.

is unexpectedly high; so is the number of writes with 6 KB. Studies of the frequently used commands show that the text processing command "troff" reads source files in 512 byte blocks, and writes to output files in 6 KB blocks. We also noticed that the percentage of I/O's with 1 KB size is much higher than those with 4 KB and 8 KB. Since the 4.2BSD file system attempts to improve the performance by increasing the block size to 4 KB, and by doing read ahead, this indicates that many system utility programs are not optimized for the reimplemented file system. We examined a number of them by turning our tracer on, running the commands, turning the tracer off, and then looking at the trace output. Below are some of our observations.

- the text editor "vi" reads and writes in 1 KB blocks;
- the C language compiler "cc" reads in 1 KB blocks;
- the pattern searching program "grep" reads in 1 KB blocks;
- the remote copy (across machine boundaries) command "rcp" reads and writes in 1 KB blocks;
- the redirection mechanism (e.g., head file1 > file2) is not buffered, but rather sends data line by line.

We identified these block sizes without reading the usually complex utility programs, and the information is useful for the updating and tuning of system programs. (Some of these, such as the small block size for rcp, are not performance

bugs; the ethernet board buffer is too small to accomodate much larger blocks [Cabr85].)

I/O size range (bytes)	number of reads	percent	number of writes	percent
0-40	35,551	16.1%	18,191	22.1%
510-520	47,069	21.3%	1,076	1.3%
1020-1030	42,847	19.1%	20,946	25.5%
4090-4100	18,401	8.3%	3,332	4.1%
6140-6150	22,876	10.4%	22,979	27.9%
8190-8200	4,462	2.0%	1,756	2.1%
0-oo	220,579	100%	82,266	100%

Table 3. I/O Size Distribution.

4.2.2. File I/O duration

Figure 1 shows the distributions of the durations of the read and write operations in real time. The file system performance improvements in 4.2BSD seem to be quite effective; most of the I/O operations return within 20 milliseconds.

4.3. I/O Transfer Rates

The read and write transfer rate of the file system at a resolution of two minutes is plotted in Figure 2. We can see that the read and write rates are very much synchronized: when read activity is high, the write activity is high also. As might be expected, more data is read than written; the ratio of reads to writes is 2.68 and the ratio of bytes read to written is 1.88. We also observe that the I/O transfer pattern is quite bursty; the peak rate is around 3.7 megabytes/sec. for reads, and 3.3 megabytes/sec. for writes.

4.4. Process Lifetimes and Number of Active Processes

We observed earlier that the distribution of an open-close session duration is highly skewed. A similar pattern occurs, though less severe, in the distribution of process lifetimes measured in real time. Figure 3 shows the lifetime distribution of processes existing less than three seconds, which include 56.7% of all the processes observed during the entire trace. Thirty-seven percent of the processes last less than one second. Conversely, 26.7% of the processes exist longer than 10 seconds, 12.7% of them longer than 1 minute, and 2.86% of them longer than 10

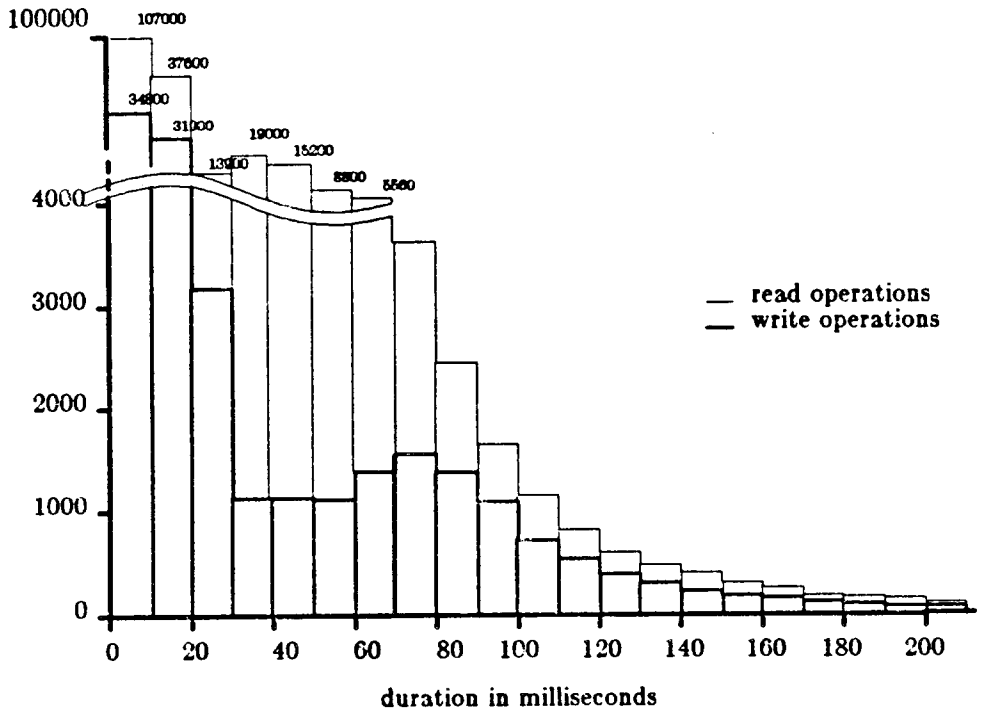

Figure 1. Distribution of File I/O Durations.

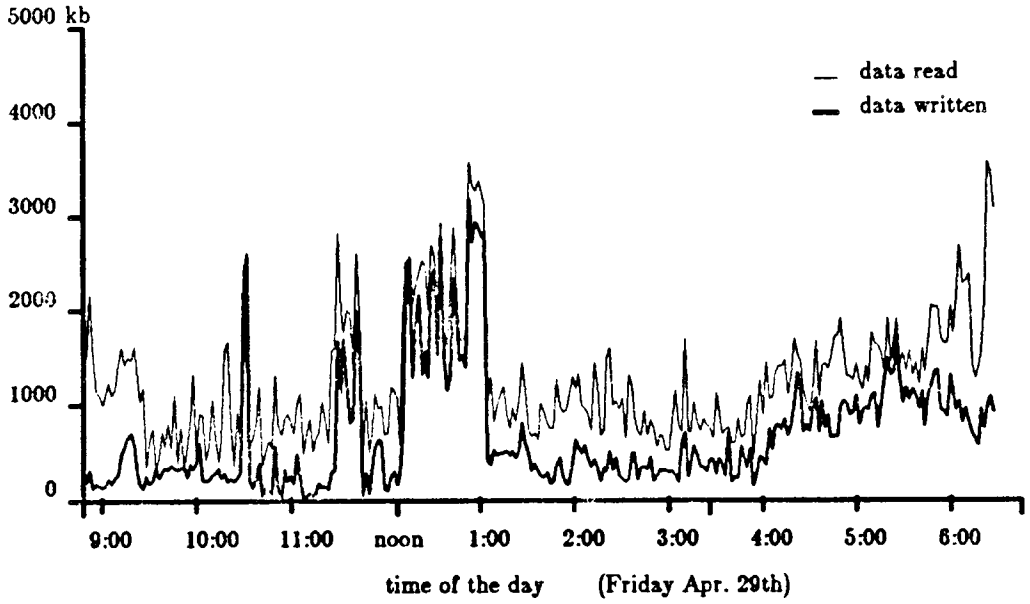

Figure 2. I/O Transfer Rate during the Day Measured in 2 Minute Intervals.

minutes.

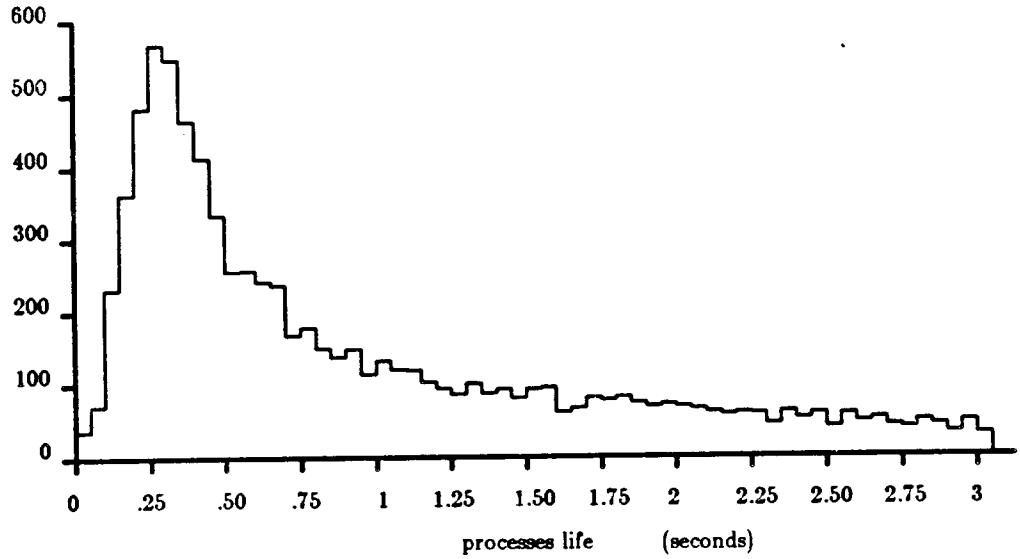

Figure 3. Process Life Time Distribution.

Although processes are born and die rather frequently, the number of *active* processes in the system (those that performed some file operations in an interval of a given length) does not fluctuate excessively. Figure 4 is a plot of active processes seen in one minute intervals throughout the whole day. We can regard this curve as an approximate pattern of system load. When the trace was started in the morning, mainly only secretaries were logged on, doing some clerical work; the peak load occurs around noon time, and then slowly decreases. (There is usually a pre-lunch peak, as everyone submits jobs before going to eat; there is often a pre-5pm peak as the clerical staff leaves, although it isn't evident in figure 4.) The load remains high through the evening, as one might expect at a university, although it isn't shown in this figure. The maximum number of active processes observed is only 189.

5. PERFORMANCE EVALUATION

In order to assess the impact of the tracing package on system performance, and to decide if it is suitable for extended tracing periods, we performed a few measurement experiments. Several factors contribute to the overhead incurred by the package. First, hooks are placed in relevant system calls and information gathering routines are executed every time these calls are made. Second, tracing information is written to trace files using the standard file system facility, thus competing with other activities in the system for I/O channel and disk bandwidth, and disk space. Finally, trace records are accumulated in large buffers before

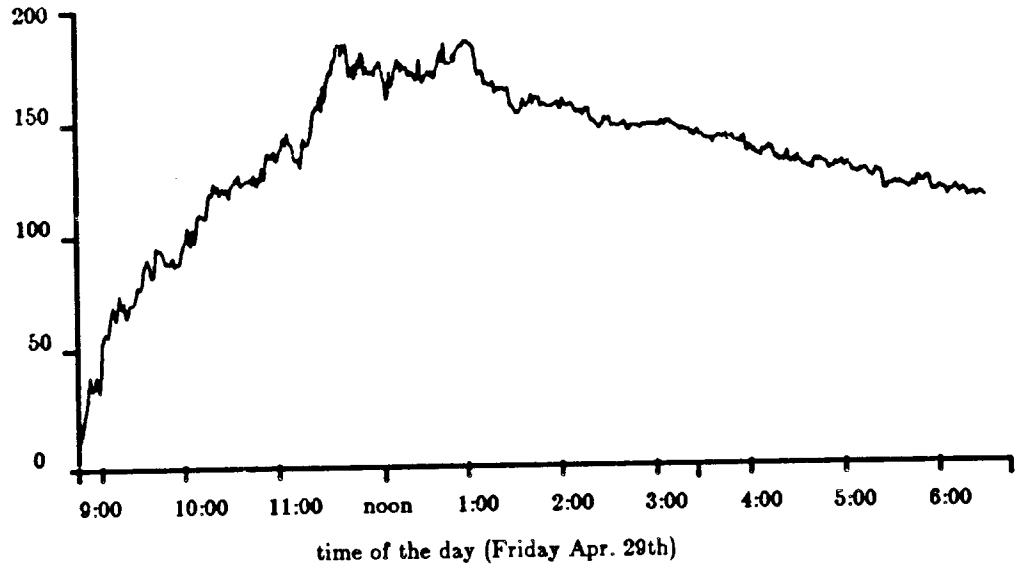

Figure 4. Number of Active Processes.

being dumped to the files, and new buffers are allocated on demand from the system buffer pool, hence adding to the contention for the buffers.

5.1. CPU Overhead

We wrote a test program that does nothing but a large number of file system calls in proportion to the distribution of the calls observed in the trace. The program was run on a VAX-11/750 a number of times with the trace turned on and off. There was no other user on the system; this way, we eliminated the error introduced by CPU and device resource contention, and by the fact that measured times under Unix are very sensitive to overall system load. Timing results of the program executions show that the test program consumes 7.7% more system time when running with the trace turned on than with the trace off. We can regard this increase as a measure of the overhead of the tracing on the file system calls. Since programs do other things besides making file system calls, it is reasonable to expect that the overall overhead of the tracing should be much lower than this number on a typical system. Although by running alone on a system, we avoided the error caused by interactions with other users, the resource contention overhead is also not measured in the above experiment. Hence, we performed the same set of experiments on the production machine, ucbarpa, on which the trace data were collected. The resulting overhead figures are slightly higher, but still below 10%. We also ran a few other I/O intensive test programs, and the relative increases in their execution times are similar to the numbers cited above. Another experiment with a CPU intensive program suggests a 2% to 4% increase in

measured CPU time when the tracer is in use, although these numbers are very approximate due to load average differences and fluctuations between the trace/on and trace/off measurements.

5.2. I/O and Buffer Allocation Overhead

Besides the CPU overhead of the package, the above experiments also partly show the overhead on the I/O system and the buffer management. This overhead, however, can be directly assessed by looking at the amount of trace data written to the disk as a fraction of the total amount of data transferred by the I/O system. These numbers are included above in the general statistics of the tracing session. (The amount of data read and written in the statistics does not include the dumping of the trace data because the dumping is not recorded.) We see that the dumping constitutes about 3.6% of the total I/O calls (reads and writes), and about 9.6% of the volume of data written to the disks. These numbers are consistent with the CPU overhead given above. Since the trace data are written onto the disks in large blocks whose sizes are chosen for good file system performance, the actual I/O overhead should be somewhat lower than these figures. Considering the extensive amount of information we collect, we feel that such an overhead is acceptable. During the trace, we asked around and did not get any complaints from the users about the system performance.

6. CONCLUSIONS

In this paper, we described the design and implementation of a logical file system tracing package for Berkeley UNIX. We also presented some preliminary analysis results derived from data generated by the package. We believe that the package has met the design requirements specified in the introduction, except that more tracing options could have been incorporated to provide more flexibility. Although we are mainly interested in file system activity, the package sets up a framework for tracing any event inside the kernel; this framework includes the scheme to place hooks, the buffering and trace file switching mechanism, the data format specification method, and the notion of internally generated unique object ID's (file_id, open_id,...). Experience obtained so far indicates that the package is robust, easy to use, and relatively low in the overhead incurred. (For comparison, one of the authors of this paper found that tracing IBM systems using GTF caused CPU overhead upwards of 20%.) Since the size of the trace code is small, the tracer has the potential to be incorporated in the future as a feature of Berkeley UNIX. An embedded tracing package provides the system administrators the opportunity of turning the trace on whenever they want to examine the system operations and/or to detect performance problems, (or to collect data for research studies), without even bringing the system down in order to install a special instrumented version; the package can thus serve as a monitoring and

performance debugging tool for the system.

The data generated by the tracing package can be used in a number of ways to provide insights into various aspects of the UNIX file system and user file access behavior. Although only some very simple analyses of the trace data are included in this paper, that data is informative. For instance, we found that I/O activity is very bursty, that very few read and write operations are performed in most of the open-close sessions, and that most processes are short lived. The analyses also exposed some unexpected behavior and some performance deficiencies in a number of frequently used system utilities, hence suggesting possible performance improvements. Besides studies of system behavior, the trace data can also be used in driving simulation models for the study of disk caching and file migration algorithms [Smit81a,b], and distributed file system performance [Porc82]. By providing detailed information about file and process activity, the data can also be useful for load balancing studies. Extensive analyses and simulation experiments on the trace data are being planned and will be reported in separate papers. We believe that the techniques used in the design and implementation of the tracing package are applicable in the construction of tracing packages for other systems as well, and that the results of the analyses will provide significant additional understanding of file system operations, above and beyond those aspects particular to the UNIX system.

7. ACKNOWLEDGEMENTS

The work presented herein would not have been conducted so smoothly without a number of people's support and help. We would like to thank Mike Karels, Kirk McKusick and Jim Bloom for their patience in explaining the UNIX kernel and system operation procedures, and for making available to us a number of the research machines for our experiments. Joel Emer and Joe Falcone of the Eastern Research Laboratory of Digital Equipment Corporation provided some valuable suggestions during the development of the package. We are also indebted to many other friends for their help, including the plots in the paper.

8. REFERENCES

[Cabr85] Luis Cabrera, private communication.

[Luka83] Tibor Lukac, "A UNIX File System Logical Trace Package", Master Report, University of California, Berkeley, August 1983.

[Mcku84] Marshall McKusick, William Joy, Samuel Leffler and Robert Fabry, "A Fast File System for UNIX", ACM TOCS, 2, 3, August, 1984, pp. 181-197.

[Oust85] John K. Ousterhaut, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2BSD File System", submitted to the 10th Symposium on Operating Systems

Principles.

[Porc82] Juan Porcar, "File Migration in Distributed Systems", Ph.D. dissertation, UC Berkeley, EECS Dept., June, 1982.

[Smit81a] Alan Jay Smith, "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", IEEETSE, SE-7, 4, July, 1981, pp. 403-417.

[Smit81b] Alan Jay Smith, "Long Term File Migration: Development and Evaluation of Algorithms", CACM, 24, 8, August, 1981, pp. 521-532.

[Smit85] Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations", to appear, ACM Transactions on Computer Systems, 1985.