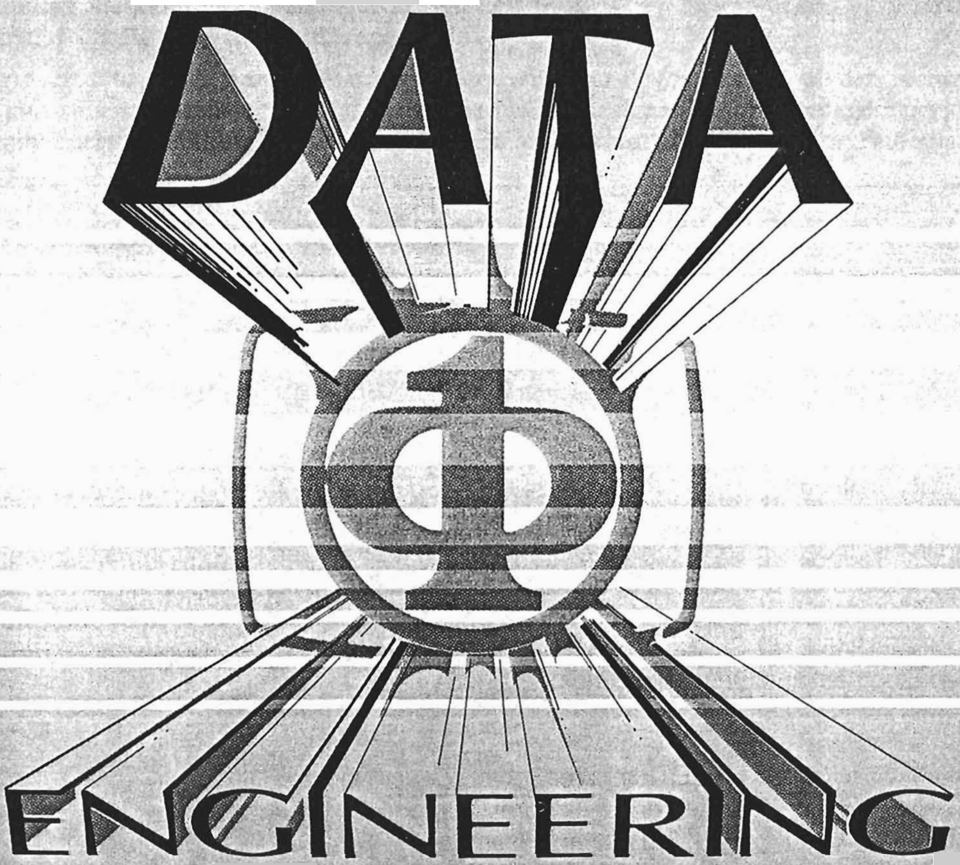
Proceedings

Third International Conference on DATA ENGINEERING



February 3–5, 1987
Pacifica Hotel
Los Angeles, California, USA

IEEE Computer Society Order Number 762
Library of Congress Number 86–46344
IEEE Catalog Number 87CH2407–5
ISBN 0–8186–0762–9

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY PRESS

File Sessions: A Technique and its Application to the UNIX File System

John H. Maloney and Andrew P. Black

University of Washington

Abstract – **This paper describes a new technique for analyzing dynamic file usage patterns based upon classification of file sessions. A *file session* is defined to be the set of operations on a given file from the moment it is opened until the moment it is closed. If file system measurement data is organized into sessions, each session may then be classified by the pattern of file use that it demonstrates. Examination of the overall pattern of file use revealed by this classification leads to valuable insights for file system designers. The technique is illustrated by applying it to data collected from a UNIX® file system by John Ousterhout at Berkeley [6]. One surprising result was a high incidence of "lock files".**

I. Introduction

The recent proliferation of networks of personal computers has created a demand for file servers, and has caused renewed interest in file system design. One way for file system designers to obtain information about file systems is to study the daily patterns of use in an established file system. Such information helps the designer establish the feasibility of proposed designs, provides a basis for making engineering tradeoffs within a given design, and can help identify patterns of use which might be separated out as special cases to improve performance.

The emphasis of this paper is on the analysis of *general purpose* file systems, such as are commonly used to store programs and documents of various kinds. A file system may also support large databases with specialized programs for accessing, locking, and updating records in the database. However, this paper does not consider issues related to database support. Moreover, the computers from which we obtained our data were not used for serious database work.

® UNIX is a registered trademark of AT&T.

This work is supported in part by the National Science Foundation under Grant Number DCR-8420945, by the Xerox Corporation University Grants Program, and by the Digital Equipment Corporation External Research Program.

Files are often used in stylized ways. For example, some programs append data to files, while others create new files or overwrite existing files with new data. While all three of these styles may be equally efficient when using a particular file server, another server with a different design may favor one style over another. One might hope that an analysis of file usage would enable one to determine which server is best. However, since in all three cases the files may be opened in "write" mode, statistics based on file open modes would not help one to determine the relative importance of these styles of access.

This paper describes a technique for organizing file system trace data in a way that highlights the distinctions between different styles of file use. It then discusses the (sometimes surprising) results of applying this technique to a UNIX file system.

The technique revolves around the concept of a *file session*. A file session is defined as the entire set of operations on a given file between the open operation and the corresponding close operation. By analyzing this set of operations as a group, it is possible to discover patterns which are not otherwise obvious. For example, a given session might open a file for writing, position the write head at the current end of the file, write some data sequentially, and then close the file. This is logically an append operation, even if the file system does not have an append primitive. By classifying file sessions into categories and collecting statistics on each category as a whole, one may obtain a view of the file system activity which is oriented towards the functional needs of the user community. It might be discovered, for example, that append style access is very common in a certain file system, suggesting that improved performance might be obtained by implementing an optimized append operation.

The next section reviews recent work in the area of file system analysis with special emphasis on the technique used. None of this work has categorized the data according to style of access. Section III then describes the application of the file sessions technique to an analysis of dynamic file usage patterns in a UNIX 4.2 BSD file system. This analysis lead to the discovery of an unforeseen pattern of access (the use of "flag" or "lock" files for synchronization) which alters the interpretation of the results of two recent studies of file usage patterns in UNIX. The analysis also revealed that different styles of access have significantly different dynamic file size distributions.

II. Related Work

Satyanarayanan studied patterns of use in a TOPS-10 file system [7]. His technique was to analyze *static* snapshots of the file system, collecting data about the functional lifetimes, sizes, and types of both on-line and archived files. (The *functional lifetime* of a file is the length of time that the contents of the file are useful, that is, the length of time before the data is either forgotten or changed.) Among other things, he discovered that files in the TOPS-10 system tend to be small and have short functional lifetimes, that the size and lifetime of a file depends upon its type, and that larger files tend to have shorter functional lifetimes. Although these findings are relevant to the choice of a file migration strategy, they say nothing about the dynamics of file system use.

John Ousterhout and his students studied the dynamics of a UNIX 4.2 BSD file system [6]. Their technique was to log operations at the kernel interface to the file system and to perform a postmortem analysis of the log file. They collected information about data transfer rates, file sizes (weighted by the number of dynamic references), file lifetimes, file open times, file access patterns, and the effect of various caching strategies. Some of their more significant findings are: per-user data transfer rates tend to be low; files tend to be small; most information tends to be deleted soon after it is created; files tend to be open for only a very short time; most files are accessed sequentially in their entirety; and caching file blocks can significantly reduce the amount of disk traffic. Although these findings are valuable, Ousterhout's analysis of file access patterns was not sufficiently detailed to answer certain questions. In particular, accesses were categorized according to the mode of access (read, write, and read/write) rather than by the style of access.

In a very thorough study Rick Floyd also examined the dynamics of the UNIX file system [3]. His work corroborates many of Ousterhout's findings but goes much further towards an analysis of access patterns based upon the style of access. He broke the set of referenced files into three categories: log files, temporary files, and permanent files. This partitioning was done on the basis of the file name: *a priori* knowledge of the system was used to identify system log files, and temporary files were identified by either the directory in which they were created (*/tmp*) or by the syntax of the file name. The file identification process required some tedious trial and error and a deep understanding not only of the UNIX operating system, but also of many of its application programs. Furthermore, Floyd assumes that all references to files identified as either logs or temporary will have the same access pattern. The technique outlined in our paper depends upon neither an extensive knowledge of the system to be measured nor the assumption that all references to a given file have the same access pattern.

The next section describes the technique and its application to a UNIX file system. This work relies heavily upon the work of Ousterhout. In fact, since the analysis was performed upon file system data collected by Ousterhout, a direct comparison of these results with his results is possible.

III. A Study of the UNIX File System

Motivation

Our primary interest in patterns of UNIX file usage stems from our involvement in the design of a common file storage service for use in our own department and in similarly heterogeneous environments elsewhere [1]. The file system design is novel in a number of respects, but for the purposes of this presentation two aspects of the design are particularly significant.

- Multiple versions of a file can be stored; the file system includes version naming and efficient storage for a large number of versions using a novel difference representation [2].
- Immutability of versions: once created, a version is treated as read-only. A file cannot be modified in place.
- A new version of a file can be created only by writing it sequentially from the beginning to the end.

Although immutability has the virtue of simplicity, its implications on the performance and utility of the resulting file service must be considered. We assumed a model in which the user's workstation has access to a local disk for temporary storage. A file on the file server is "updated" by fetching the most recent version to the local disk, modifying it as required, and storing the new version back on the file server. If the modification is small (for example, inserting a line in a text file) and the file is large, considerably more work is required to create a new version of the file using the fetch-modify-restore scenario than would be necessary if the file server allowed the file to be modified in place.

Fortunately, there is some empirical evidence that makes the fetch/store design defensible. At the Xerox Palo Alto Research Center, a file system in which versions must be written and read in their entirety has been in use for some years [9]. In the UNIX environment, many application programs deal with entire files [3, 6]; for example, if the insertion of the single line in the above scenario were accomplished with a UNIX text editor, the entire file would be read into main store and then re-written to the disk. We wished to collect more detailed information on usage patterns before committing to a file server design that disallowed in-place modification. In particular, Ousterhout's results were not detailed enough to distinguish between usage patterns corresponding to the creation of a new file, appending to a file, or arbitrary modification of a file.

Method

Analysis was performed on three traces of UNIX system activity collected by Ousterhout and his students at Berkeley in the spring of 1985 [6]. The traces were produced by logging every system call which affected the file system. The following log entries were of interest to us.

- File *opens* and *creates*, which mark session beginnings. There are also two operations which may be performed as side effects when the file is opened: the file may be truncated to zero length (truncate mode) or the write head may be positioned to the end of the file (append mode).
- File *closes*, which mark session ends. The read/write head position is recorded when the file is closed.
- *Seek* operations, which explicitly change the position of the file's read/write head; both the old and new head positions are recorded.
- Read and write *data transfer* operations were not recorded on the log, but could be inferred by examining changes in the file's read/write head position.
- *Truncate* operations on open files, which change the file's length.

Unlike Ousterhout, we were not interested in the file reads corresponding to program loading (*execs*), since we assumed that frequently executed programs would be resident on the workstation's disk. More detailed information about the data maintained in the trace log may be found in Ousterhout's paper.

The three machines from which the traces were taken were used primarily for document preparation and program development (the machines named A5 and E3) and computer-aided design (machine C4). Analysis was performed on the same three traces analyzed by Ousterhout to allow easy comparison of our results with his. Each trace covers approximately three weekdays and contains between 733 000 and just over a million event records which constitute between 233 000 and 358 000 complete file sessions. Because the system was not quiescent when tracing was started and stopped, each trace also contains a tiny number of incomplete sessions, which were ignored.

Before analysis is begun, our technique requires that one postulate a set of access-style categories, using intuition and observations. It may be necessary to repeat this process several times to develop an appropriate set of categories. The final set of categories used for this analysis was as follows.

ReadOnly	The file is not modified.
NewData	The file is created from scratch or by completely overwriting an existing file. The latter may occur if the file is written sequentially from the beginning past its previous end-of-file, or if it is truncated to zero length and before data is written into it. Either way, none of the old contents of the file is retained.
Modified	The file is modified in some arbitrary way. Database updates would fall into this category. All sessions in which the file is read as well as written were placed in this category.
Flag	No data is written. The file starts empty, ends empty, and is empty in between.
Append	New data is added to the end of the file. The old contents of the file remain untouched.
DeleteBody	The file is truncated to zero length and left empty.
Temp	The file starts empty and ends empty but some data resides in the file in between. (No sessions of this type were encountered.)

Flag sessions result from the use of the file system for synchronization. Older versions of UNIX did not provide file locks as a primitive, and some applications use the existence of a file with a certain name as a lock. (No attempt is ever made to read the flag file itself; it exists solely to cause certain operations on the directory to fail.) Although we were familiar with this locking convention, we did not expect to find that it was used so frequently. We made flag sessions a separate category when we discovered a large number of files with zero length in the Modified category. The opposite occurred with Temp sessions: sessions matching this pattern were expected but not observed. It is probable that applications which create a temporary file close the file with data still in it and then delete the file. Since no Temp sessions were encountered they will not be mentioned again.

Since the purpose of this analysis was to investigate issues relating to a network file server, the classification of file sessions ignored manipulation of newly created data; workstation software would presumably perform this manipulation locally before transferring the file to the server. For example, consider a session which opens a file for writing, positions the write head at the current end of the file, writes 100 bytes of data, moves the write head back to the original end of the file and re-writes 30 bytes of data, and finally truncates the file to 10 bytes greater than its original length. This session would be placed in the Append category, since the overall effect is that the file

Category	A5 count, % of sessions		E3 count, % of sessions		C4 count, % of sessions		Composite count, % of sessions	
ReadOnly	240274	67%	208149	66%	131728	56%	580151	64%
New Data	69616	19%	58080	18%	58040	25%	185736	21%
Modified	15954	5%	20053	6%	21263	9%	57270	6%
Flag	17727	5%	19224	6%	11536	5%	48487	5%
Append	11559	3%	10078	3%	9229	4%	30866	3%
DeleteBody	3049	1%	2116	1%	1269	1%	6434	1%
Total	358179	100%	317700	100%	233065	100%	908944	100%

Table 1: Distribution of Sessions by category.

has 10 new bytes of data appended to it. (Note that no operation involved the data which constituted the original body of the file.)

Results

Table 1 shows how the file sessions were distributed between the various categories. The "Composite" column combines the columns for the individual machines. ReadOnly sessions constitute about two-thirds of all sessions. Among the write sessions, the NewData category is the most significant, accounting for over half of the write sessions. Flag sessions form an unexpectedly large fraction of the write sessions.

There is a discrepancy between our analysis and the results of Ousterhout and Floyd. Ousterhout observed that 81 to 85 per cent. of write-only file accesses wrote data sequentially from beginning to end, while Floyd observed that, overall, 78 per cent. of write-only or read/write file accesses wrote to the entire file. (Floyd does report that log files are a significant exception.) Yet, in the current work, nearly all of the whole-file, sequential writes occurred in the NewData category, which constitutes only 58 per cent. of all writes. What accounts for the difference? It seems that both Ousterhout's and Floyd's results were skewed by the large number of Flag sessions, which both considered to be whole-file accesses even though no data is actually written. This is clearly a case where knowing the styles of access leads to a greater understanding of the data. Since Flag files are an artifact of the UNIX environment, the Flag session class should probably be considered irrelevant to the design of network file servers.

Our analysis verified the finding of Ousterhout and Floyd that about two-thirds of the ReadOnly sessions access the entire file. This fact, and the fact that many write sessions touch the entire file (even after eliminating Flag sessions from consideration), agrees with the Xerox PARC experience that a file server that provides only whole-file transfers is not unreasonable. Yet what about the sessions that do *not* touch the entire file? One argument for not making special provision for these sessions has been put forth by the designers of the ITC file server [5]: files are small enough that the cost of transferring the entire file from one place to another is not significantly greater than the cost of transferring only part of the file. To test this hypothesis, size distribution statistics were collected for each session category. Graphs of the file size distributions for the NewData, ReadOnly, Modified, and Append categories are presented in Figure 1. (These graphs show the combined file size distributions of all three traces.)

The file size distributions for these categories are markedly different from each other. First, consider the graph for NewData sessions. Newly created files tend to be small, with a median size between 50 and 500 bytes. Ninety per cent. of the NewData sessions created fewer than 5 000 bytes of data. ReadOnly sessions also have a tendency to access small files; seventy per cent. of these sessions access files of 5 000 bytes or fewer. However, a noticeable fraction of the ReadOnly sessions accessed files in the 500 000 to 5 000 000 byte range. This leads us to question file server designs that require clients to read the whole file. Random access read is easy to implement and may significantly reduce the overhead involved in reading small portions of large files.

Append sessions showed a marked tendency to access larger files than either NewData or ReadOnly sessions. About seventy-five per cent. of these files are over 5 000 bytes and fifty-five per cent. are over 50 000 bytes. This is not surprising; Append sessions are used by UNIX to add entries to log files maintained for accounting purposes. Since our size statistics are weighted by the number of

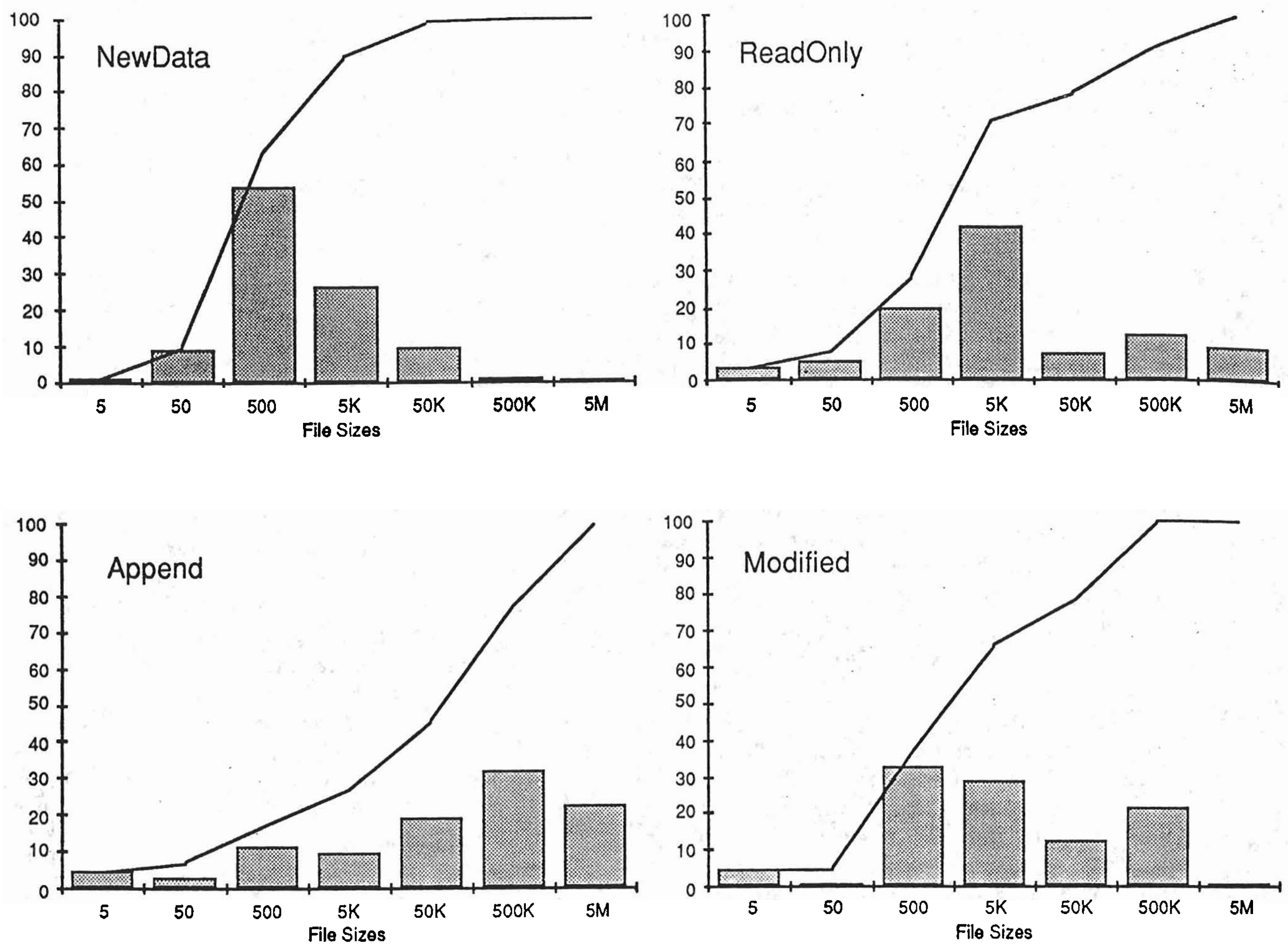


Figure 1: Dynamic file size distribution for various session types. In each case, the columns indicate the percentage of sessions that used a file of the indicated size. The lines plot the same data cumulatively, i.e. the percentage of sessions using files less than the given size.

accesses, the frequent use of append mode by the system (whose log files are large) could be weighting the file size distribution towards larger files.

For this and other reasons, it would be nice to know what portion of the sessions in the Append category correspond to system activities. Floyd found that programs run by users made relatively little use of user-owned log files. Recall, however, that his categorization of log file accesses was based upon an *a priori* identification of the "log files" in the file system rather than on analysis of the actual patterns of access. When we filtered out sessions opened by programs running under user names 'root' and 'daemon', we discovered that Append sessions accounted for five per cent. of the remaining sessions. Surprisingly, this is significantly *more* than their fraction (three per cent.) of the unfiltered sessions. This discrepancy could be explained by user programs accessing system- and network-owned files, since Floyd examined only user programs accessing *user-*owned files. It might also be explained as an artifact of Floyd's static categorization method: there could be significant numbers of appends to files which Floyd did not identify as log files. It is not possible to establish the truth of either hypothesis without using both techniques to analyze identical data.

We were extremely interested in the file size distribution for the Modified category, since it is these sessions which could cause the greatest unnecessary performance penalty for a file server that disallows in-place file modification. We were relieved to discover that most of the files accessed by Modified sessions were fairly small. Two-thirds of them were less than 5 000 bytes and only twenty per cent. were in the 50 000 to 500 000 byte range.

Relevance to File Server Design

The results of our analysis allow us to answer several critical file server design questions including:

- Should the file server support only "whole-file transfer" for file reading?
- Should the file server make special provision for append operations?
- Will disallowing modification of files in place preclude the use of the file server for the usual activities supported by UNIX?

Ousterhout and Morris have made strong arguments for supporting only whole-file transfer [5, 6], noting that the majority of read sessions access the file in its entirety. Indeed, we found that sixty-nine per cent. of the ReadOnly sessions accessed entire files. It certainly is possible that fewer ReadOnly sessions access *large* files in their entirety than similar sessions on smaller files. (One could conduct an experiment to test this hypothesis by splitting the ReadOnly category into two categories, ReadOnlyLarge and ReadOnlySmall, with separate whole-file transfer statistics for each category.) However, even assuming that the whole-file transfer characteristic is independent of file size, we must still be concerned about the thirty-one per cent. of the ReadOnly sessions which do *not* access the entire file.

Fetching the entirety of a large file to the user's workstation takes a lot of time and consumes a large portion of the workstation's local disk. Because we cannot assume that the program will always *need* the entire file, it seems best to choose a transfer increment which is large†, but not necessarily as large as the entire file. For example, if files were transferred in 8k byte chunks, small files (seventy per cent. of the accesses) would be transferred in their entirety in a single chunk but larger files could be read incrementally.

Append sessions constitute only three per cent. of all sessions. However, since fifty-five per cent. of Append sessions involve files larger than 50k bytes, we conclude that special provisions might reasonably be made for them. In a typical append operation, the amount of data appended is only a very small fraction of the existing data. Note that supporting an append operation is not inconsistent with the notion of immutable file versions: an append operation creates a new file version similar to the old one except that some additional data appears at the end. It is possible to avoid storing multiple copies of the same data by integrating the append operation with the file version storage mechanism; only the new data and a pointer to the previous file version need be stored.

The final potential problem concerns Modified sessions. What does it cost to support this usage pattern without in-place file modification? We may assume that workstations cache files, as they do in the Vice/Virtue system [8]. Thus, a sequence of changes to a file within a short period of time would share the cost of a single fetch and store. Once the file has been fetched to the workstation, the cost of file modification, as perceived by the user, would be determined by the speed of the workstation's file cache. Storing the new version back on the server may be done in the background. What cannot be hidden from the user is the cost of the initial fetch operation. Ignoring communication and processing overhead, and assuming a file transfer rate of 10 kbytes/second, we see that the files touched by eighty per cent. of the Modified sessions could be fetched in five seconds or less. The worst case is only fifty seconds. These delays are acceptable, especially given the very small fraction of sessions in the Modified category. Furthermore, it is possible that many of these sessions reflect database usage, which would be absorbed by a database server rather than the file server.

† Large transfers are to be preferred because the server overhead is amortized over more bytes; see reference 4 for details.

One further area of interest is the CPU load on the proposed file server. CPU load has been cited as a significant influence on file server response time [4]. Our proposed design, by insisting on transferring files in their entirely (when creating new files or reading small files) or in large chunks (when reading large files) keeps the file server's processing overhead low. Disallowing in-place file modification forces much of the cost of file updates to be assumed by the workstations. The consequent reduction in server CPU load should improve average case performance.

Figure 2 summarizes the previous discussion. Notice that NewData sessions always require the transfer of the entire new contents of the file, so they would cost the same independent of whether the file server allows the modification of files in place. Flag and DeleteBody sessions reflect UNIX operations that would not have analogues in our proposed file server. The two possible performance problems, Modify and Append sessions to large files, are indicated.

Validity

Two points of caution about the validity of these results are in order. First, it is clear that the UNIX environment is different from the workstation-based environment in which file servers operate. Furthermore, UNIX's data stream abstraction encourages a certain programming style. Many UNIX utility programs are *filters* which always process their entire input stream (or file) in sequential order. Lessons taken from an analysis of UNIX may not transfer well to systems with significantly different programming "cultures". Second, it should be noted that because these traces are based on usage patterns averaged over a period of days, we are likely to be counting system activities more heavily than user activities. Not only are our statistics affected by large system administration jobs scheduled to run in the small hours of the morning, but we are also accumulating the effects of ongoing system activities (network table

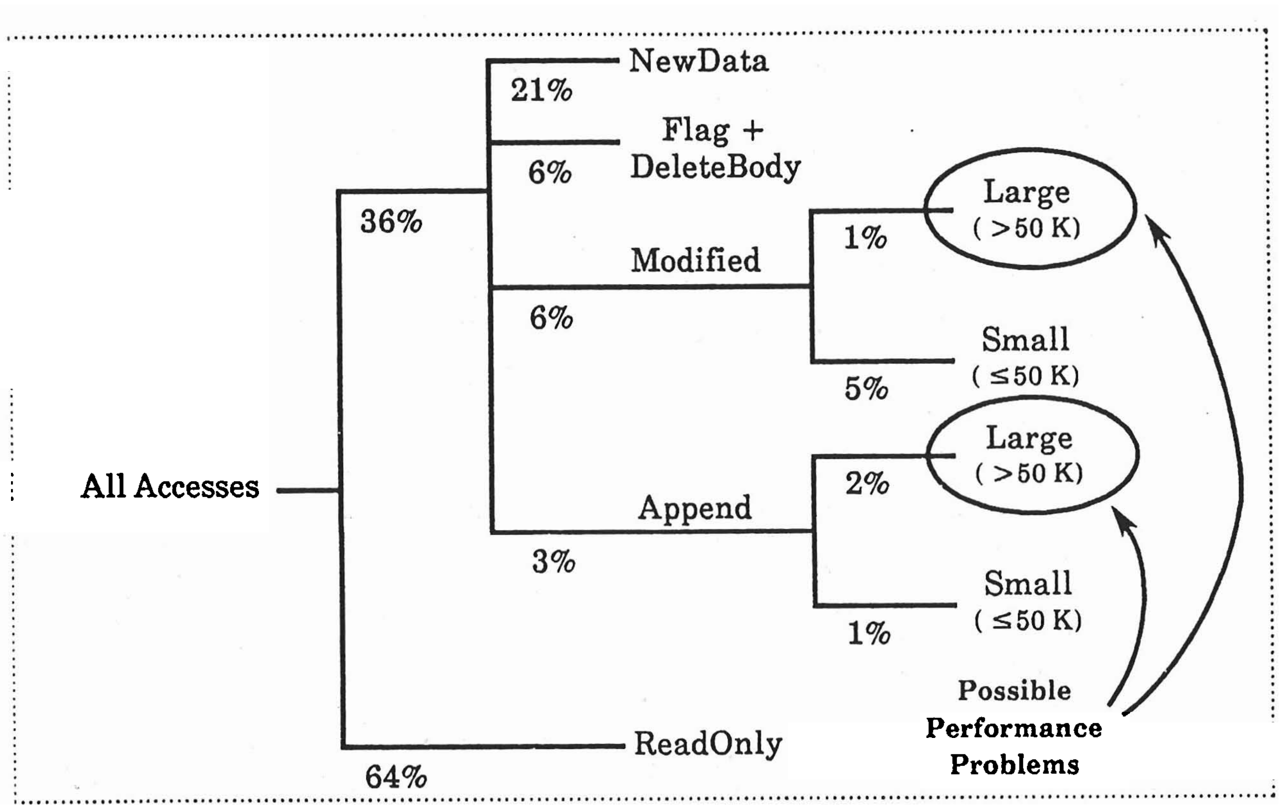


Figure 2: Summary of results relevant to server with immutable files

maintenance, mail, etc.) around the clock, whereas the user community is only active for part of that time. In a workstation-based environment, much of this "system" activity might be carried out by specialized servers (e.g., mail delivery servers) which would operate independently of the file system. Floyd, who also collected data around the clock, discovered that the system itself initiated about seventy-three per cent. of the file system activity [3].

IV. Conclusions

The categorization of file sessions by access pattern is a powerful tool for analyzing file system dynamics. When we applied this technique to a UNIX file system, we discovered two cases in which previous file system analysis techniques have generated misleading conclusions. First, the large number of Flag sessions in UNIX led both Ousterhout and Floyd to over-estimate the proportion of write accesses which display the whole-file transfer phenomenon. (Although Floyd was aware of the existence of Flag sessions, his technique made it difficult to discard these sessions in his analysis of whole-file transfer.)

Second, Floyd's static classification of file types led him to under-record the number of append-style file accesses, particularly those performed by user programs. Our technique allows us to detect append-style file accesses by examining actual patterns of access; there is no need to pre-identify log files and no danger of overlooking any append-style accesses.

We also discovered that the dynamic distribution of file sizes is correlated with the style of access. For example, newly created file versions tend to be small while files accessed in the append style tend to be large. An analysis of the file access traffic based upon a combination of access style and file size lead us to conclude that files should be read in large chunks (but not necessarily in their entirety), that provisions should be made for append-style accesses, and that a file server that disallows in-place file modification is feasible.

This is not to say that in such a design all access patterns will be equally efficient. On the contrary: we have seen that small changes to large files will involve reading and writing the whole file over the network, which is potentially very costly. However, our data does say that such usage is rare; the considerable extra effort necessary to design, implement and maintain a file server with an in-place modification protocol would yield a benefit in only one per cent. of the sessions. Even if one has manpower to spare, it is probably a mistake to provide this facility: one would be better advised to devote one's energy to optimizing the performance of the other ninety-nine per cent. of sessions. This is the reduced instruction set principle applied to software systems.

The value of the file sessions technique (and the study of the UNIX file system) that we have presented is that it enables system designers to optimize the "instruction set" of their products to deal with real rather than imagined problems.

Acknowledgement

We thank John Ousterhout for making his trace data available to us.

References

[1] Black, A. P. and Lazowska, E. D. "Interconnecting Heterogeneous Computer Systems". *Proc. European UNIX systems User Group Autumn '86 Conf.*, Manchester, UK, September 1986, pp43-52.

[2] Burris, C. H. "Selection Matrices: An Algebraic System for Representing File Versions ". MS Thesis (in preparation), University of Washington, Computer Science Dept.

[3] Floyd, R. "Short-Term File Reference Patterns in a UNIX Environment". Tech. Rep. 177, Dept of Computer Science, Univ. of Rochester, Rochester, NY 14627, March 1986.

[4] Lazowska, E. D., Zahorjan, J., Cheriton, D. R. and Zwaenepoel, W. "File Access Performance of Diskless Workstations". *Trans. Computer Systems 3, Nr 3* (August 1986), pp238-268.

[5] Morris, J. H., Satyanarayanan, M., Conner, M., Howard, J., Rosenthal, D. and Smith, F. D. "Andrew: A Distributed Personal Computing Environment". *Comm. of the ACM 28, Nr 3* (March 1986), pp184-201.

[6] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, K. and Thompson, J. G. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System". *Proc. 10th ACM Symp. on Operating System Prin.*, December 1985, pp15-24.

[7] Satyanarayanan, M. "A Study of File Sizes and Lifetimes". CMU-CS-81-114, Computer Science Dept, Carnegie-Mellon University, Pittsburgh, PA, April 1981.

[8] Satyanarayanan, M., Howard, J. H., Nichols, D. A., Sidebotham, R. N., Spector, A. Z. and West, M. J. "The ITC Distributed File System: Principles and Design". *Proc. 10th ACM Symp. on Operating System Prin.*, December 1985, pp35-50.

[9] Schroeder, M. D., Gifford, D. K. and Needham, R. M. "A Caching File System for a Programmer's Workstation". *Proc. 10th ACM Symp. on Operating System Prin.*, December 1985, pp25-34.