Long Term Distributed File Reference Tracing: Implementation and Experience

L. Mummert and M. Satyanarayanan

November 1994
CMU-CS-94-213

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

DFSTrace is a system to collect and analyze long-term file reference data in a distributed UNIX workstation environment. The design of DFSTrace is unique in that it pays particular attention to efficiency, extensibility, and the logistics of long-term trace data collection in a distributed environment. The components of DFSTrace are a set of kernel hooks, a kernel buffer mechanism, a data extraction agent, a set of collection servers, and post-processing tools.

Our experience with DFSTrace has been highly positive. Tracing has been virtually unnoticeable, degrading performance 3-7%, depending on the level of detail of tracing. We have collected file reference traces from approximately 30 workstations continuously for over two years. We have implemented a post-processing library to provide a convenient programmer interface to the traces, and have created an on-line database of results from a suite of analysis programs to aid trace selection.

Our data has been used for a wide variety of purposes, including file system studies, performance measurement and tuning, and debugging. Extensions of DFSTrace have enabled its use in applications such as field reliability testing and determining disk geometry. This paper presents the design, implementation, and evaluation of DFSTrace and associated tools, and describes how they have been used.

This research has been supported by the National Science Foundation under Grant ECD-8907068, and the Air Force Materiel Command (AFMC) and the Advanced Research Projects Agency (ARPA) under Contract F19628-93-C-0193. Support also came from the Digital Equipment Corporation and the IBM Corporation. The U.S. government is authorized to reproduce and distribute reprints for government purposes, notwithstanding any copyright notation thereon.

Keywords: file reference tracing, distributed file systems, Coda file system, Andrew file system, file access patterns, measurement, evaluation

1 Introduction

Empirical data from file systems has been used in many phases of the development of data storage systems. For example, such data has been used to study file caching[1, 2], placement[3], and migration[4, 5, 6, 7]. In this paper, we describe the design and implementation of a system called *DFSTrace* to collect long-term file reference data in a distributed workstation environment. The challenges involved in collecting such data are in engineering rather than concept. Hence this paper focuses on the design and implementation of DFSTrace rather than on the results of using the traces.

The need for detailed file reference traces arose in 1989 during the development of the Coda file system[8, 9], an experimental distributed file system that provides high availability. The trace data had to have several properties that distinguish our work from other file reference tracing efforts. First, the data had to be *long-term* – weeks or months. Second, it had to contain information on a *broad class of file system operations*. Third, it had to be from a *distributed workstation environment*. None of the existing sets of file reference data from UNIX* environments at the time[10, 11, 2] satisfied all of these requirements. Even now, five years later, only our data meets these requirements.

DFSTrace meets these requirements. We have used this system to collect data continuously from approximately 30 workstations for over two years. We have obtained over 150 GB of data containing references to the Andrew File System[12] (AFS†), NFS[13], Coda, and the local UNIX file system[14]. We have developed a versatile post-processing library and tools to analyze the data, and an on-line database of results from a suite of analysis programs to aid in selecting traces for study.

The rest of this paper is organized as follows. Section 2 describes the design of DFSTrace. The instrumentation and collection machinery are described in Sections 3 and 4, respectively. The post-processing library, summary suite, and on-line database are described in Section 5. In Sections 6 and 7 we evaluate DFSTrace qualitatively and quantitatively. Section 8 summarizes the ways in which researchers have used and extended DFSTrace. We close with a discussion of related work and conclusions.

2 Design Rationale

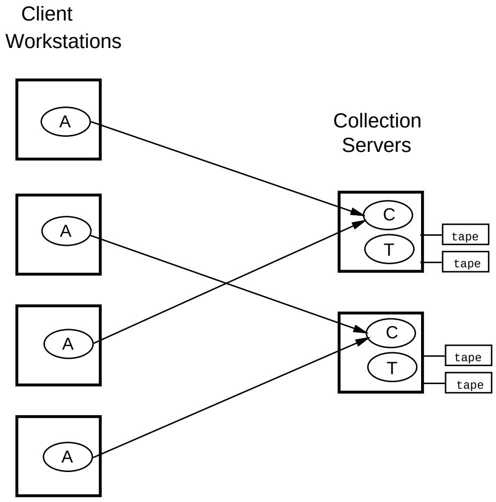
In this section we describe how our data requirements influenced the design of DFSTrace. We then present the architecture of the collection system, and discuss how it addresses the design requirements. Last, we describe the format and content of the data collected by DFSTrace.

2.1 Requirements

Long term data collection imposes several requirements on a tracing system. The most important requirement is that tracing must be unobtrusive, otherwise users may alter their behavior or refuse to be traced. This requirement is critical in view of our desire for detailed traces, because clients are likely to generate a large amount of data. The system must be efficient both in terms of client workstation performance and client resources used, and it should be application-transparent (i.e., users should not have to run special versions of their application software to generate trace data). The desire for efficiency and application-transparency suggests data should be gathered in the operating system kernel. Because the information needed to construct trace records resides in kernel data structures, gathering data in the kernel minimizes crossings of the user-kernel boundary and is hence more efficient than gathering data at user level. Again, to keep tracing overhead low, data should not be processed during collection. To minimize client resource use, data should reside on the client only temporarily; it should then be shipped to a collection site in the background.

Tracing a distributed workstation environment imposes the following additional system requirements. Distribution introduces multiple points of failure. The system should be robust enough to detect and tolerate failures. Buffering on the client can mask short failures, but may not suffice for prolonged outages. In the long term, failures resulting in data loss are inevitable. The system must be able to record the occurrence of data losses so they may be detected later.

*UNIX is a registered trademark of UNIX Systems Laboratories, Inc.
†AFS is a registered trademark of the Transarc Corporation.



This figure shows the overall structure of DFSTrace. Each client runs an agent daemon (A), which extracts trace data from the kernel and ships it to a collection site. Each collection site runs a collector (C), which receives the data and buffers it in disk files. An optional tape daemon (T) writes these files to tape in the background.

Figure 1: Top-level view of DFSTrace

Distributed environments are often heterogeneous, and the architectures used tend to change over time. Therefore the system should be reasonably portable to new architectures. In a long-term collection effort, changes are inevitable in the data being collected as well as in the system collecting it. The system must be flexible enough to detect and cope with such changes gracefully. A versioning scheme should be used that allows components of the system to be treated as interchangeable. This implies that the collection mechanism should not depend on the content of the data being collected. Finally, the system should be easy to administer. The logistics of gathering the data should be automated wherever possible.

2.2 Architecture

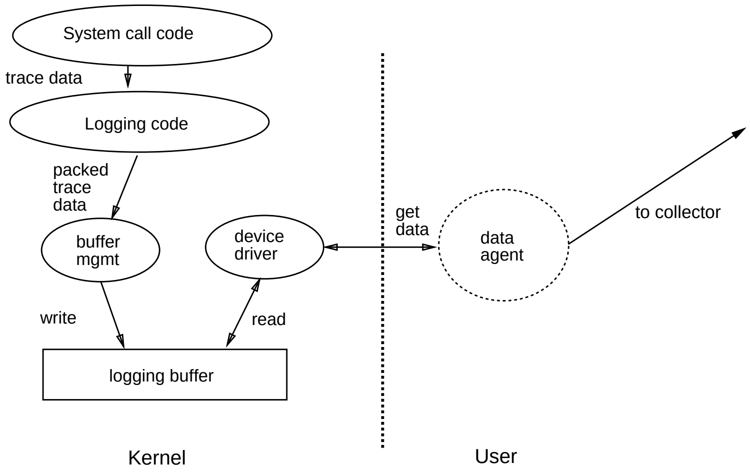
Figure 1 presents a high-level view of DFSTrace, excluding post-processing software. Trace data is generated by client workstations running kernels instrumented at the system call level. The data is extracted by a user-level process, or *agent*, buffered locally in memory, and then sent to one of a small number of data collection servers, or *collectors*. A collector buffers the data on disk; in the background an optional *tape daemon* moves the data to tape. The data is post-processed later to obtain a usable set of traces for analysis. Multiple servers may be used to balance load and maintain availability.

The agent and collector do not interpret the data, thus their operation is independent from the data being collected. The kernel, agent, and collector may be changed independently. The agent and collector employ version information in their communication interface to allow incompatible releases of code to be detected at runtime. If an agent is incompatible with the collector, the agent exits with an advisory message.

More detail on client operation is provided in Figure 2. We have instrumented system call code to gather data on file system activity. Relevant data is passed to a logging routine which packs a trace record and writes it into an circular memory buffer. The agent extracts blocks of data from the buffer through a simple device driver interface. The agent buffers data in memory rather than in files to minimize its impact on the data being collected.

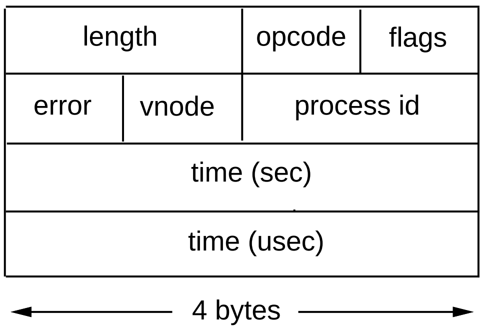
2.3 Data Format and Content

The performance of client workstations is affected directly by the amount of data they generate. We wanted to collect detailed data on file system operations within the limit of reasonable client performance. Needless to say, it took a few iterations before the data generated was complete, and struck a good balance between detail and performance. In this



This figure shows the components of DFSTrace on a client workstation. The system call code in the operating system kernel is instrumented with hooks that gather data on file system operations. Trace data is packed into records by the logging module, and written into a circular memory buffer. The buffer is exported as a device; the user-level agent process reads trace data from the buffer through the device driver interface.

Figure 2: Tracing on a Client Workstation



This figure shows the raw trace record header. The first field contains the length of the record in bytes. The return code of the call is in the "error" field. Each half of the "vnode" field indicates the file system in which objects in the record reside. For records referencing more than two objects (such as link and rename), a separate word is provided for this purpose. The "flags" field is reserved for internal errors; flags are set if data required for the record (such as pathnames) could not be obtained. The rest of the record consists of the system time, in seconds and microseconds. The trace library, described in Section 5.2, uses the length and vnode fields internally. The header it presents to analysis programs omits these fields.

Figure 3: Trace Record Header

Record	Items recorded (with header)
`open`	flags, mode, file descriptor, index, user ID, old size, size, file type, fid, directory fid, path
`close`	file descriptor, index, # reads, # writes, # seeks, bytes read, bytes written, size, fid, file type, open count, flags, caller, mode
`stat, lstat`	fid, file type, path
`seek`	file descriptor, index, # reads, # writes, bytes read, bytes written, offset
`chdir, chroot, readlink`	fid, path
`execve`	size, fid, owner, path
`access, chmod`	fid, mode, file type, path
`creat`	fid, directory fid, old size, file descriptor, index, mode, path
`mkdir`	fid, directory fid, mode, path
`chown`	owner, group, fid, file type, path
`rename`	from fid, from directory fid, to fid, to directory fid, size, file type, # links, from path, to path
`link`	from fid, from directory fid, to directory fid, file type, from path, to path
`symlink`	directory fid, fid, target path, link path
`rmdir, unlink`	fid, directory fid, size, file type, # links, path
`truncate`	old size, new size, fid, path
`utimes`	access time, modify time, fid, file type, path
`mknod`	device, fid, directory fid, mode, path
`mount`	fid, read/write flag, path
`unmount`	fid, path
`fork`	child pid, user ID
`exit, settimeofday`	(header only)
`read, write`	file descriptor, index, amount
`lookup`	component fid, parent fid, file type, component path
`getsymlink`	fid, component path, link path
`root`	component fid, target fid, path
`dump`	system call counts
`note`	annotation

This table shows the contents of trace records for each operation. Records corresponding to UNIX system calls are shown in the upper portion of the table. The `lookup`, `root`, and `getsymlink` records are generated during name resolution. The `note` record allows users to embed notes in a trace, for example to indicate a particular point in the execution of an application.

Figure 4: Contents of Trace Records

section, we give the history behind the data we decided to collect, and discuss some of the surprises along the way. Then we discuss the content of the data we currently collect.

2.3.1 Evolution

We wanted to collect data on all system calls relating to the file system, and any other calls that would aid in post-processing (such as `fork` and `exit`)[‡]. We were not certain that tracing `read` and `write` calls would be feasible, because of the large amount of data that was likely to result. We began by estimating the amount of data a client workstation would generate in a day. We instrumented Mach[16] kernels running on IBM PC/RTs to count the system calls of interest. Based on the information we expected to record for each system call, we estimated that each workstation would generate 6.2 MB per day without reads and writes, and 31 MB per day with reads and writes. We decided not to record reads, writes, or seeks, but only to record summary information on those operations when the file was closed.

A prototype implementation yielded only half the data volume we expected. We added tracing of `seek` calls, thinking that it would not increase data volume significantly because non-sequential access was uncommon in our environment. We were very surprised when data volume increased dramatically from several workstations. The culprit was a monitoring program that displays the status of a variety of workstation resources, such as disk and CPU utilization. The program obtains its information by reading from `/dev/kmem`, a special file that allows random locations in kernel virtual memory to be accessed[15]. Unfortunately, the program has to seek to each location in memory containing data of interest. One could argue that having to read kernel memory to obtain information on resource utilization represents a deficiency in the UNIX system call interface. Given that, and the fact that we were not keenly interested in accesses to special files, we disabled the reporting of individual seeks on `/dev/kmem`. The number of seeks is contained in the close record, so our data still shows that large numbers of non-sequential accesses are performed on `/dev/kmem`.

We also implemented collection of read and write data as an option. Our workstations do not normally enable it because it is not critical to our studies. We can obtain reasonably detailed information about access patterns from summary statistics recorded in close and seek records, including the number of reads and writes and the amount of data read and written.

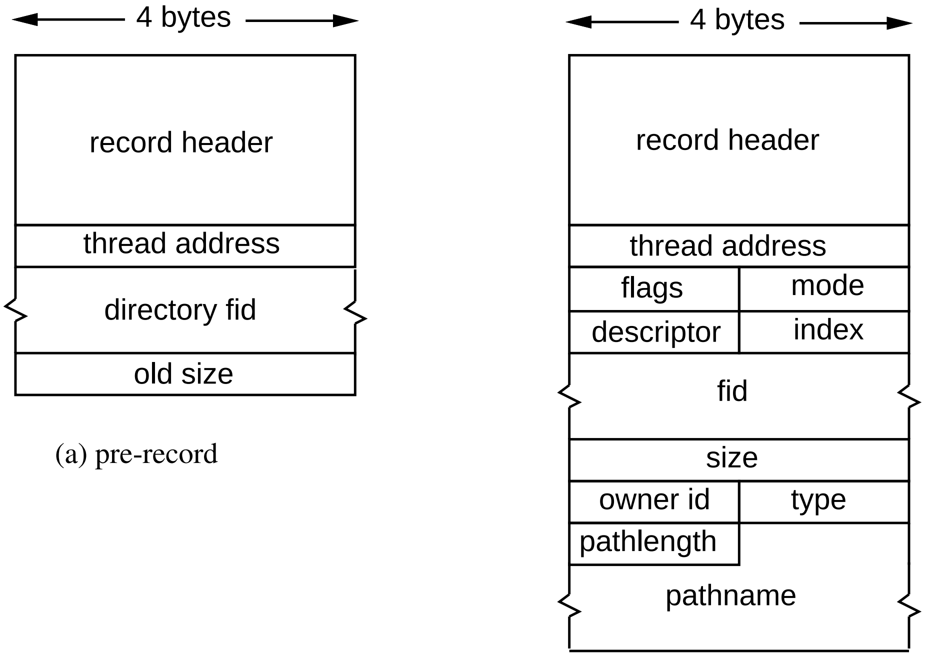
We discovered a critical omission in the data after using it as input to a simulator for the Coda file cache manager. The cache manager receives requests not as system calls, but as *Vnode operations*[17]. The mapping between system calls and Vnode operations is reasonably direct, except for *name resolution*. Name resolution is the mapping of a path name to a fixed-length low-level identifier. It involves traversing the path name by component, and is performed beneath the system call interface. Although it is possible to simulate name resolution if a snapshot of the file system exists[18], snapshots are not feasible in our environment because workstations access large distributed file systems such as AFS. Hence we added support for tracing name resolution operations.

2.3.2 Trace Records

Figure 4 lists the data we now collect. All records begin with a fixed-length header that includes the length of the record, opcode, process ID, error code, and time. The raw form of the header is depicted in Figure 3. The upper section of Figure 4 lists the contents of records corresponding to UNIX system calls. In general, these records contain the arguments and return values for the call, and internal information on the objects involved in the call.

Trace records are variable in length. Most records contain a path name and one or more low-level file identifiers, or *fids*. The format and length of the fid depends on the file system in which the object resides, and can vary from eight to sixteen bytes in length. The file system containing each object referenced in the record is also recorded, to allow comparisons of local and distributed file system usage, and to identify references to the same object through different workstations or pathnames. We record the fids of all objects that could be affected by an operation. For example, an

‡We assume familiarity with the UNIX system call interface. For more information, see Section 2 of the *UNIX Programmer's Manual*[15].



(a) pre-record

(b) post-record

This figure shows an example of a split record, in this case the `open` record. Variable-length fields are bordered with broken lines. Fields from the "fid" or "directory fid" onwards may not be present if errors occur while obtaining the data. If such an error occurs, a flag is set in the flags field of the record header.

Split records consist of a pre-record and a post-record. The pre-record, shown on the left, is written if data of interest will be destroyed during the system call. This record may or may not be present. In this example, a pre-record is written if the file exists and it is being recreated or truncated at open time. The "old size" field of the open pre-record indicates the size of the file at open time if it already exists. The post record, shown on the right, is always present. It contains data that is available at the end of the system call.

Figure 5: Split Record

`open` might create a new file, so we record information on the parent directory of the file. A `rename` of a file to a different directory where the new name already exists involves four different objects.

Several of the system calls we record involve *file descriptors*, which are used by processes to perform I/O. A file descriptor is a result of a successful `open` system call. It is used by the kernel as an index into a table of open files for the process. Each entry in the process open file table points to an entry in the system open file table, which contains information about the file represented by the descriptor. New descriptors for an open file may be created for a process using the `dup` system call. If a process creates a child process, all of the parent's descriptors are inherited by the child. To avoid recording calls like `dup` and keeping track of aliasing, we record the file's index in the system open file table along with the descriptor.

The lower section of the table corresponds to auxiliary or internal operations. The `note` record allows programs to deposit additional information into the trace. Users have found this facility convenient for annotating experiments. The rest of the operations in the lower part of the table occur during name resolution.

3 Kernel Instrumentation

Our goal in instrumenting the kernel was to modify as little of the existing code as possible. We added two modules to the kernel – one containing code for packing trace records, and another for managing the circular buffer. The kernel instrumentation consists of three layers, as illustrated in Figure 2.

The topmost layer of instrumentation is in the system call code, which contains hooks to the packing code. For many system calls, a single one-line hook at the end of the call is sufficient to capture the data of interest. The hook appears at the end of the call to record the return code and any output parameters.

Unfortunately, not all system calls are structured in a way that allows all the desired data to be obtained with one hook. Some system calls destroy data. The obvious ones are `unlink` and `rmdir`. Less obvious examples include `rename`, which may remove the target if it exists, and `open`, which will remove a pre-existing file if the "create" flag is set. For these cases, there is a hook to record information on the data about to be destroyed, in addition to the hook at the end of the call. These "split" records are reassembled by the post-processing library and presented as single records to the user. An example of a split record is shown in Figure 5.

For other system calls, the data of interest is scattered throughout several modules. Examples of this are `mkdir` and `open`. When a file or directory is created, the parent directory changes. Information on the parent directory is most conveniently obtained in a routine called by the system call. We use split records in these cases to record information that is not available in the system call itself.

There are sets of system calls that are similar enough that their code is a veneer over a common routine. Examples of this are `open` and `creat`, `mknod` and `mkdir`, `stat` and `lstat`, and the attribute-setting variants `chmod, chown, utimes`, and `truncate`. In these cases the best location for the hook is in the common routine, but it is not always obvious from that routine which operation is the caller. For the cases that are not easily deduced, we have added a parameter to the common routine that indicates what the calling operation is.

Another complication is early return points. We have instrumented certain early return points because they generate file system activity. For example, a common early return point in system calls that take pathnames as arguments is when there is no file corresponding to the pathname. Even though the system call fails, we still record the call because the system must perform name resolution to discover the error, generating file system activity.

It is important to be able to match file opens and closes in a trace. Files are closed in several places other than the `close` system call. For example, files are closed when a process exits. They are also closed in a variant of `dup` which allows the new file descriptor to be specified. If there is already a file open with that descriptor, the system will close it first. Under certain conditions, files are closed in `execve` as well. Each of these locations must be instrumented to capture file close events completely.

All of the hooks are above or within the vnode interface, which is a layer in the kernel that allows a variety of local, remote, or even non-Unix file systems to be incorporated in a single system. Since the vnode layer is file system independent, the hooks capture references to any file system hooked into the kernel. There is only one piece of file system dependent tracing code, namely, a routine that packs fids into trace records.

Trace records are packed in the left middle layer of Figure 2. The routines in this layer gather any additional data that may be needed for the records, such as file attributes and fids. Packed records are placed in a circular memory buffer, in the bottom left layer of Figure 2. The interface to this buffer is that of a simple device driver supporting `read`, `select`, and `ioctl` system calls. If the buffer wraps around, the `read` call returns an error and advances the "bytes read" counter by the amount of the read. Through the `ioctl` call, tracing may be turned off or on dynamically, and tracing of various classes of operations (such as reads and writes or name resolution) may be enabled or disabled.

4 Collection Machinery

As described in Section 2.2, the collection machinery consists of the agent daemons running on client workstations, and collectors running on a small number of servers. An optional tape daemon may be used at collection sites to spool data to tape.

One of the challenges of long-term data collection is coping with changes in tracing software and the format and content of the traces. It is desirable to structure the system so that older traces are still usable, even though they may not be compatible with newer ones. We have incorporated version information into each component of DFSTrace, and the system embeds this information in the header of each trace. Thus the traces are in some sense self-documenting. The library is structured to accept any of the various formats, and determines which it is by reading the version information in the trace header.

4.1 Agent

The goal of the agent is to extract trace data from the kernel without consuming excessive resources on the host machine. The agent is implemented as a multi-threaded user-level process, with one thread reading data from the kernel through the tracing device described in Section 3, and another sending data to the collector via remote procedure call. We used the LWP threads package, which provides non-preemptive (co-routine) threads, and the RPC2 remote procedure call package[19]. The agent reads blocks of data from the kernel and buffers them in memory. It uses two fixed-size buffers, one for each thread, consuming roughly 1 MB of memory by default. Users can specify a different memory limit using a command line argument. The agent is typically started at boot time.

The agent's *kernel thread* is responsible for reading blocks of trace data out of the kernel before the data is overwritten. If data has been overwritten, an error is returned to the agent on its next read. The agent reports the amount of data lost before a successful read, if any, in a header that is prepended to the trace data block. In addition, the thread records whether or not there were problems communicating with the collector before the read. This gives some indication why the data was lost. We describe that further below. The header also contains a block sequence number. This is useful in post-processing the traces, also described below.

The *network thread* takes a buffer filled with trace data blocks and headers and sends it to a collector. If communication fails, the network thread records the failure and attempts to resend the data. It backs off exponentially if subsequent resends fail. Note that while this is happening the kernel thread may run out of space to put new trace data and fall behind. We decided to record the server failure in the agent header to see why data gets lost. We have found that most data losses occur because of contention for resources on the client, and not because of network failures.

The agent responds to several UNIX signals that allow users to tell the agent to flush data or shut down. Users may also specify at what level operations are to be traced using a command line switch. The operations are grouped into the following independent categories – basic system calls (open, close, etc.), read and write system calls, and name resolution. Most of our clients traced the basic system calls and name resolution.

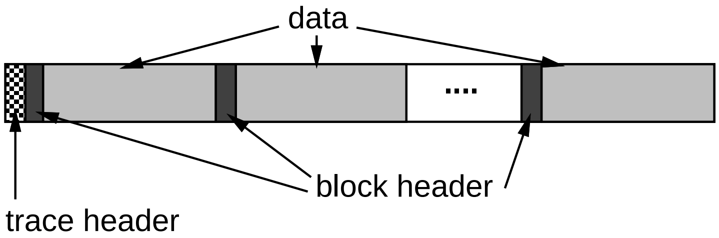
4.2 Collector

The collector is a multi-threaded server that receives trace data from potentially many hosts. Data is stored on disk in *staging files*; traces from different hosts are stored in different staging files. After a staging file reaches a certain size (about 5 MB), the collector starts a new staging file for that host, and the filled file may be archived to tape. The collector prepends a header to each staging file containing version information for the tracing kernel, agent, and collector – together these define the format of the trace. The header also contains the client's network address and boot time, and the start time of the agent. The format of raw trace data is shown in Figure 6.

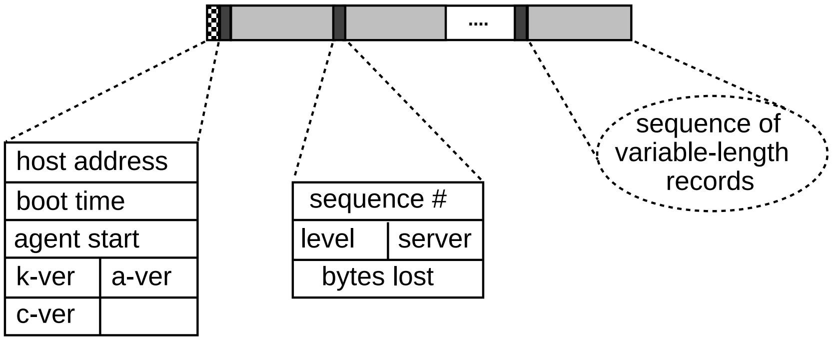
Periodically, the collector prints summary statistics on the clients from which it is receiving data. The default period for the summary report is one hour. A sample summary report is given in Figure 7. It is easy to see from this summary which hosts have not connected recently, and which hosts are active. There is a longer form of the summary that also includes the client birth time and the versions of client software.

4.3 Tape Daemon

The tape daemon is an optional component of DFSTrace that automatically archives filled staging files to tape. It can scan multiple data partitions, and switch between multiple tape drives. The tape daemon responds to a signal to scan for new data to archive. The collector uses this signal to notify the tape daemon when a staging file is ready to be archived.



(a) Trace Structure



(b) Contents of Fields within a Block

This figure shows the format of raw trace data. The agent reads trace data from the kernel in fixed-size blocks, and prepends a header to each block containing the block sequence number, the level of tracing, the number of bytes lost before the block (if any), and a flag indicating the status of the agent's connection to the collector.

The collector receives a sequence of block headers and blocks, and prepends a trace header containing the traced host's network address, time of the last system and agent restart, and the versions of the tracing kernel, agent, and the collector.

Figure 6: Format of Trace Data

```
Host            last transfer   # bytes (transfers)     conn open
128.2.209.204   Jan  7 17:20:24       2037568 (4)       Jan  5 22:05:52
128.2.222.111   Jan  7 17:25:47        509392 (1)       Jan  5 22:03:44
128.2.209.213   Jan  7 16:34:12        509392 (1)       Jan  5 22:03:44
128.2.209.215   Jan  7 17:05:49       2546960 (5)       Jan  5 21:59:28
128.2.209.217        *****                  0 (0)            *****
128.2.206.77    Jan  7 17:00:04        509392 (1)       Jan  5 22:08:00
```

Figure 7: Collector Summary Report

5 Post-processing

So far, we have discussed how trace data is generated. In this section, we discuss how to use trace data. Once the trace data is generated, it must pass through a post processing step that assembles the longest possible trace subject to a set of conditions. This is discussed in Section 5.1. In Section 5.2, we discuss the *trace library*, which simplifies trace analysis by hiding the underlying structure of a trace beneath a convenient programming interface. Then in the last part of this section, we discuss the *summary suite*, which is a set of analysis programs that generates summary statistics for a trace. We run this suite on every trace and place the results in a database to aid users in identifying and selecting traces for analysis.

5.1 Maximizing Trace Length

We would like to guarantee that the traces are *complete*, namely, that they contain every event that occurred on the client in the interval covered by the trace. To do this, a post-processing step is necessary to transform staging files into complete traces. This post-processing step assembles the longest trace from staging files, subject to several termination conditions. These conditions correspond to machine restarts, agent restarts, and data losses. Data are recorded by the agent in the data block header. When a loss is detected, the trace is split at that point. Machine reboots and agent restarts cause new staging files to be created. The new staging files have different trace headers than their predecessors.

The length of post-processed traces varies. Our traces range from approximately five minutes to weeks in length, and approximately 1 MB to 800 MB. A few traces were broken at 800 MB even though none of the trace ending conditions applied, because they were limited by the size of the disk partition on which they were constructed.

5.2 Trace Analysis Library

The goals of the trace analysis library are to provide a convenient programmer's interface to the traces and to implement common operations. The underlying structure of the trace is hidden behind a simple interface, shown in Figure 8. The library is structured to accommodate traces of various formats, including those of other researchers, while maintaining a consistent interface to the programmer.

The operations for initialization and termination are shown in Figure 8(a). The `Trace_Open` call opens the trace file and determines the format of the trace by reading the *preamble* at the beginning of the file.

The library calls for obtaining records are shown in Figure 8(b). The central call is `Trace_GetRecord`. The library unpacks the raw, structured trace, and presents it to the application as a sequence of records through this call. The call returns the next record, subject to a filter specification, if any, as a pointer to a record structure. The library allocates the storage necessary for the record and any pathnames included in the record. To free the storage, programs call `Trace_FreeRecord`. The `Trace_CopyRecord` copies a record, allocating new storage for both the record and any pathnames it references.

The library maintains a good deal of bookkeeping on the trace, such as keeping track of open files, gluing split records together, and building and tracking process trees, so that groups of processes may be studied in aggregate (e.g. `make`). Because of this, the records that the library presents to the programmer are often more detailed than shown in Figure 4. For example, the library simulates the system open file table for each trace it processes. This allows it to provide data from the open record for file descriptor based operations (e.g., seek and close), such as pathnames.

Certain fields are common amongst a set of records, such as pathnames and fids. In Figure 8(c), we show routines that obtain those fields from records, allowing the fields to be treated generically. The call `Trace_GetUser` obtains the user ID (uid) that generated the record. The uid is not present in all records, only the fork record. The library keeps track of process activity through fork and exit records, and thus is able to determine which user generated a record in most cases.

In Figure 8(d), we show generic printing routines for records and the file preamble, which may differ in traces of different versions. Miscellaneous calls are shown in Figure 8(e), such as for obtaining statistics on a trace.

```
Trace_Open(filename)
Trace_SetFilter(filep, filter_file_name)
Trace_Close(filep)
```

(a) Initialization and Termination

```
Trace_GetRecord(filep)
Trace_CopyRecord(sourcep, destpp)
Trace_FreeRecord(filep, recordp)
```

(b) Record Manipulation

```
Trace_FidsEqual(fid1p, fid2p)
Trace_GetFid(recordp, fidplist, nump)
Trace_GetFileIndex(recordp)
Trace_GetFileType(recordp)
Trace_GetPath(recordp, pathplist, nump)
Trace_GetRefCount(recordp)
Trace_GetUser(filep, pid, uidp)
```

(c) Field Retrieval

```
Trace_PrintPreamble(filep)
Trace_PrintRecord(recordp)
Trace_DumpRecord(recordp)
Trace_OpcodeToStr(opcode)
Trace_NodeIdToStr(addr)
Trace_OpenFlagsToStr(flags)
Trace_RecTimeToStr(recordp)
Trace_FileTypeToStr(type)
Trace_InodeTypeToStr(type)
Trace_FlagsToStr(flags)
Trace_FidPtrToStr(fidp)
```

(d) Output and Formatting

```
Trace_Stats(filep, statp)
```

(e) Miscellaneous

Figure 8: Library Interface

It is common to want to include or exclude various types of records from a trace, such as by uid or opcode. The library supports *filtering* of various kinds, such as by start and end time, opcode, uid, and path name. The library is a natural place to implement filtering because it is such a common operation, and because certain types of filtering require data structures the library already maintains, such as the open file table for matching opens and closes. Filtering fits neatly beneath the `Trace_GetRecord` call. Once a filter is applied to the trace, the library returns only those records that satisfy the filter specification.

Filters are specified in a filter file, which is applied to a trace using `Trace_SetFilter`. Filter specifications take the form `<attribute> [<modifier>] <value> <value> ...`, where an attribute is the opcode, for example. To keep specifications short, an optional modifier can be used to specify values to be included or excluded from the trace. Figure 9 gives an example of a filter.

5.3 Summary Suite

As the body of data we collected grew larger, summary information of various kinds for each trace became necessary, so that a user confronted with 150GB of this data has some idea where to begin. We have built an on-line database for the traces that contains, for each trace, summary information including composition by system calls, access characteristics, and activity levels. The summary information is the output of a suite of analysis programs run on each trace before being archived on tape.

The output is placed in an on-line collection of summary results to assist in finding appropriate traces for study. The suite is comprised of the programs `tstat`, `users`, `sessions`, and `patterns`. Each of these programs is run on the trace without filtering, then `tstat`, `sessions`, and `patterns` are run for each active user found in the trace. The programs are described below, along with sample output from each.

The `users` program classifies trace records by user ID where detectable. The user ID is found in the `fork` record for the process or any child processes it creates. If the process was created before the trace starts, and creates no child processes, it falls into the "Unknown" category. Output for `users` is shown in Figure 10. Uids 0, 7, 9, 11, and 4035 are system IDs. User 2336 is the primary user of the workstation from which the trace was collected.

The `tstat` program prints a variety of statistics on a trace, including a breakdown of trace records by opcode and

```
opcode open close stat lstat chdir chroot creat mkdir access chmod readlink
getsymlink chown utimes truncate rename link symlink unlink rmdir lookup root
type directory regular link
refcount 1
error 0
matchfds
start 21-Feb-91,12:00:00
end 22-Feb-91,00:00:00
pid exclude 326 2961 3640 4369
path exclude /dev/null
```

This figure shows an example of a trace filter. The opcode attribute specifies a subset of operations to be included in the filter. The user could listed the opcodes to exclude in this example. In addition, the objects referenced in the records must be either directories, files or symbolic links (no device or special files). The "refcount" filter says for operations that record a reference count (e.g., close, unlink), only return those records with a reference count of 1. The "matchfds" filter says only return close, read, and write records that have matching open records. The pid filter in this example excludes certain long-running system daemons. The path filter is a simple example; the library supports regular expressions for pathname matching.

Figure 9: Example Filter

uid	processes	records	(%)
2336	1574	643397	(69.6)
0	975	260272	(28.2)
Unknown	3	15936	(1.7)
7	46	3326	(0.4)
1516	15	672	(0.0)
9	6	388	(0.0)
4840	9	315	(0.0)
4035	2	96	(0.0)
11	1	58	(0.0)

Figure 10: Output of `users`.

file systems referenced. Figure 11 shows the output of `tstat`. The percentage is by number of records, not by volume. The "fail" column is the number of operations that failed. Name lookup usually has a high percentage of failing operations because of shell pathname searches. The difference between "records" and "raw records" reflects the presence of split records. The difference between "records" and "records returned" reflects the presence of a filter.

The `patterns` program summarizes the file reference patterns based on `close` records in the trace. The summary includes the number of read-only, write-only, and read-write accesses to files, as well as bytes transferred for each access type. Each access type is further divided into whole-file transfer, other sequential access, and random access. An access is a whole-file transfer if the amount of data read (or written) is equal to the size of the file, and there were no seeks. If there are no seeks, but the amount of data is not equal to the size of the file, then the access falls into the "other sequential" category. If seeks occur, the access is considered random.

The summary information in `close` records (number of bytes read, written, etc.) is cumulative. For example, if a file is open and the descriptor is `dup`'ed, and then the file is manipulated by both agents, the statistics reported will be the sum of their accesses, and there is no way to determine from the final `close` who did which accesses. If one of the agents only reads the file, and the other only writes, the complete session, from data in the last `close` record, will be reported as a read-write access. Figure 12 summarizes the file reference patterns in the trace for processes owned by user 2336. The format is reminiscent of that used in the study by Baker[1].

Given a trace over some length of time, how does one decide which periods to analyze? For example, one may be interested in only those periods during which a user is active. Activity can be defined in terms of the number of operations performed during a unit of time. Given that definition, an active period would consist of some number of intervals in which the activity (number of operations) exceeds some threshold. One may want to include intervals in which the number of operations falls below the threshold, as long as the decrease in activity is transient. We call the resulting period a *session*, illustrated in Figure 13.

The `sessions` program finds sessions in a trace, given the interval length, minimum session length, activity threshold, and transient length as parameters. Defaults were chosen ad-hoc as follows: an interval length of 15 minutes, session length of 16 intervals (4 hours), activity level of 16 operations per interval, and a transient length of 4 intervals. The default settings locate long stretches of fairly low activity. The summary suite uses three settings – low activity (session length of one interval, other parameters at default values), medium activity (session length of one interval, activity level of 180 operations per interval), and high activity (session length of one interval, activity level of 900 operations per interval). Figure 14 shows intervals of high activity for the primary user of a workstation.

5.4 Replaying Traces

One of the principal advantages of trace-based workloads is realism. The most direct way to subject a file system to such a workload is to *replay* a trace on it. To replay a trace, one must first construct a skeleton of the file system over which the traced operations will execute. Then commands representing operations in the trace are replayed on this skeleton.

We have developed an "untrace" facility that allows a trace to be replayed in a subtree of the name space. Untrace takes a trace as input, and produces command files for constructing the skeleton and replaying the trace. Sample output for the skeleton and replay command files is shown in Figure 15. Untrace is useful for creating realistic, repeatable workloads.

6 Status and Experience

DFSTrace runs on DECstations, Sun 4s, SPARCstations, IBM RTs, and i386s running Mach 2.6. We have traced up to 36 machines in various projects for up to two years. In the remainder of this section, we present qualitative observations about DFSTrace, paying particular attention to the requirements set forth in Section 2.1.

The most important requirement of DFSTrace was that it be unobtrusive. We have found that the performance degradation caused by tracing is virtually unnoticeable to users. The system requires very little user intervention,

```
Trace of host 128.2.209.215, versions 3.1, 3.1, 3.2
Host booted Mon Mar 30 12:40:29 1992, agent started Mon Mar 30 13:19:00 1992
Trace starts Thu Apr  2 10:15:07 1992, ends Sat Apr  4 06:30:38 1992
19356916 bytes, 448405 raw records (2/sec), 418887 records, 418887 returned
```

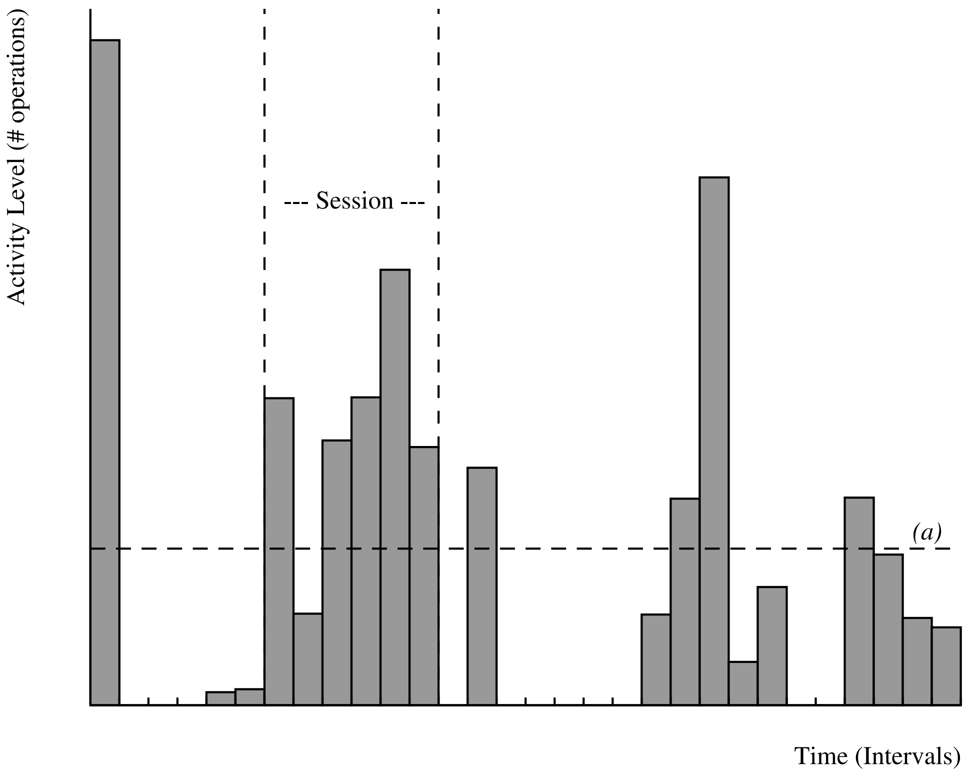
Opcode	num	%	fail	ufs	afs	cfs	nfs
OPEN	29521	6	1350	27185	1179	121	0
CLOSE	45913	10	0	44124	1567	222	0
STAT	13214	2	392	11478	992	352	0
LSTAT	30436	6	385	29705	258	88	0
SEEK	30623	6	0	28762	1857	4	0
EXECVE	4130	0	2449	1583	96	2	0
EXIT	1689	0	0	0	0	0	0
FORK	1689	0	0	0	0	0	0
CHDIR	3376	0	2	3296	59	21	0
UNLINK	620	0	243	692	16	46	0
ACCESS	1042	0	627	111	161	143	0
READLINK	18	0	15	1	6	11	0
CREAT	670	0	49	1014	26	202	0
CHMOD	75	0	3	65	5	5	0
SETREUID	539	0	0	0	0	0	0
RENAME	120	0	15	364	15	15	0
RMDIR	92	0	90	184	0	0	0
LINK	73	0	0	219	0	0	0
CHOWN	80	0	1	80	0	0	0
MKDIR	6	0	1	4	0	6	0
SYMLINK	0	0	0	0	0	0	0
SETTIMEOFDAY	0	0	0	0	0	0	0
MOUNT	0	0	0	0	0	0	0
UNMOUNT	0	0	0	0	0	0	0
TRUNCATE	88	0	0	88	0	0	0
CHROOT	0	0	0	0	0	0	0
MKNOD	0	0	0	0	0	0	0
UTIMES	52	0	8	38	0	6	0
READ	0	0	0	0	0	0	0
WRITE	0	0	0	0	0	0	0
LOOKUP	210856	47	5606	359193	48489	8424	0
GETSYMLINK	21211	4	0	17730	3429	52	0
ROOT	20142	4	0	35776	3713	795	0

This figure shows the output of `tstat`. The % column shows the the number of records of a given opcode as a percentage of the number of raw records. Dump records, containing system call counts, are not reported by `tstat`, although they are reflected in the record counts. The number of objects referenced depends on the operation and whether or not there was a failure; it will not necessarily be the same as the number of records.

Figure 11: Output of `tstat`

Access Type	Accesses (%)	Bytes (%)	Transfer Type	Accesses (%)	Bytes (%)
			Whole-file	3022 (57.1)	14173603 (53.5)
Read-only	5289 (80.4)	26470408 (70.3)	Other Seq	1154 (21.8)	1513723 (5.7)
			Random	1113 (21.0)	10783082 (40.7)
			Whole-file	990 (80.0)	5909950 (55.5)
Write-only	1237 (18.8)	10651214 (28.3)	Other Seq	0 (0.0)	0 (0.0)
			Random	247 (20.0)	4741264 (44.5)
			Whole-file	0 (0.0)	0 (0.0)
Read-write	49 (0.7)	543203 (1.4)	Other Seq	7 (14.3)	346760 (63.8)
			Random	42 (85.7)	196443 (36.2)
Total	6575	37664825			

Figure 12: Output of `patterns` for user 2336



This figure illustrates an active session in a trace. A session is composed of a series of time intervals for which activity (in number of operations) exceeds some threshold a, possibly including some number of transient intervals below the threshold. The figure shows the activity levels for a short trace. If the minimum number of intervals in a session is four, and one interval below the threshold is allowed, then there is one active session in the trace as shown above.

Figure 13: Example of a Session

```
Trace starts Wed May  6 06:30:42 1992

Begin Wed May  6 08:10:40 1992, end Wed May  6 09:40:40 1992 (1.49 hours)
        ActiveIntervals = 1, Activity = 3821m+n (34m, 3787n)
Begin Wed May  6 09:40:40 1992, end Wed May  6 12:55:40 1992 (3.25 hours)
        ActiveIntervals = 6, Activity = 10405m+n (98m, 10307n)
Begin Wed May  6 13:10:40 1992, end Wed May  6 15:53:40 1992 (2.72 hours)
        ActiveIntervals = 4, Activity = 7423m+n (4m, 7419n)

Trace ends Wed May  6 15:55:28 1992
```

This figure shows intervals of high activity for the primary user of a workstation. High activity is defined as 900 operations per interval, and a session length of one interval. Each active session is reported, along with its length and the amount of activity in mutating and non-mutating operations.

Figure 14: Output of `sessions` for user 2336, high activity

```
mkdir root
mkdir root/ufs.700.800
mkdir root/ufs.700.800/.LOCALROOT
mkdir root/ufs.700.800/.LOCALROOT/usr2
mkdir root/ufs.700.800/.LOCALROOT/usr2/lily
mkdir root/ufs.700.800/.LOCALROOT/usr2/lily/src
mkdir root/ufs.700.800/.LOCALROOT/usr2/lily/src/xmines
open root/ufs.700.800/.LOCALROOT/usr2/lily/src/xmines.c 2562 -1
open root/ufs.700.800/.LOCALROOT/usr2/lily/src/Makefile 2562 -1
mkdir root/ufs.700.800/.LOCALROOT/sys0
mkdir root/ufs.700.800/.LOCALROOT/sys0/cs
mkdir root/ufs.700.800/.LOCALROOT/sys0/cs/include
mkdir root/ufs.700.800/.LOCALROOT/sys0/cs/include/sys
open root/ufs.700.800/.LOCALROOT/sys0/cs/include/sys/types.h 2562 -1
open root/ufs.700.800/.LOCALROOT/sys0/cs/include/stdio.h 2562 -1
```

(a) Skeleton commands

```
stat root/ufs.700.800/.LOCALROOT/usr2/lily/src/Makefile
open root/ufs.700.800/.LOCALROOT/usr2/lily/src/Makefile 0 296
close 296 -1
open root/ufs.700.800/.LOCALROOT/usr2/lily/src 0 298
stat root/ufs.700.800/.LOCALROOT/usr2/lily/src/xmines.c
open root/ufs.700.800/.LOCALROOT/sys0/tmp/cc.131914 2562 301
stat root/ufs.700.800/.LOCALROOT/sys0/cs/include/sys
open root/ufs.700.800/.LOCALROOT/sys0/cs/include/sys/types.h 0 302
close 302 -1
stat root/ufs.700.800/.LOCALROOT/sys0/cs/include
open root/ufs.700.800/.LOCALROOT/sys0/cs/include/stdio.h 0 305
close 305 -1
```

(b) Replay file

This figure presents sample output from the skeleton and replay files generated by untrace. The trace was taken during a compile of the game `xmines`; we show the beginning of the compile. The commands in (a) show the construction of the file system skeleton, starting at "root". The open calls in the skeleton file create the named files. Commands in (b) are operations derived from the trace. The arguments to the open calls are the flags with which the file is to be opened, and an index which, if nonnegative, is used to refer to the open file.

Figure 15: Sample untrace output

usually just at installation time. DFSTrace is prevented from consuming excessive client resources by using fixed-length buffers in both the kernel and the agent. If the buffer capacity is exceeded, data is lost.

Data losses occur for two reasons. First, a failure may occur, such as a server crash or a network outage, for which the buffering on the client is not sufficient. Such failures are a fact of life in a distributed environment. The second source of data loss is improper tuning of the kernel and agent buffer size on the client. Clearly, one cannot trace system events in unlimited detail, and expect the client to keep up with a fixed amount of resources. Ideally, one should choose buffer sizes that balance the amount of data being generated with the resources available on the client. If the traced workload generates a large amount of data, either the buffer sizes must be increased, or the losses must be accepted.

Data losses can yield information about the clients and the system in general. Persistent losses can be an indication of improper tuning or of system or hardware failures. This class of losses generally merits investigation. For example, one of our clients had a faulty Ethernet card that caused it to lose more data than it sent. Another group of machines was separated from the collector by a gateway that was faulty, so those clients tended to lose data more than clients on the same side of the gateway as the server. As an example of improper tuning of the agent buffers, we found that some clients running a certain text processing tool lost data. When they started the tool, it read a large number of font files and generated data faster than the agent could read it from the kernel.

Our use of extensive version information has paid off. The system has gone through three major revisions and many minor revisions, and the transitions were painless. In addition to compatibility checks, versioning is useful for detecting buggy versions of traces. We have had one buggy release of the tracing kernel that generated unusable traces. Using version information we were able to find and discard traces generated by that release of the kernel.

The separation between data collection and interpretation is critical not only for good performance but also for extensibility. Although extensibility was not one of our original goals, this separation allowed others to extend DFSTrace to record other classes of events, and still take advantage of the existing collection machinery. The extensions are discussed more in section 8.

An important lesson we have learned is that is critical to use the data as soon as possible to ensure that it is complete and sufficient for its intended purpose. We went through several iterations of collecting and then attempting to use data before we arrived at the final set and content of records. The traces were validated using comparison to known workloads, such as the Andrew benchmark[20], and comparison to kernel data structures.

The library has proven to be effective in simplifying development of analysis programs. It allows the user to concentrate on the analysis of the trace rather than on manipulating the trace itself. For example, Kumar was able to read the library documentation, then write and debug the analysis program for his study[21] in about one hour.

7 Evaluation

7.1 Tracing Overhead

This section presents the performance of various levels of tracing for the Andrew benchmark. The benchmark was run in three file systems – the local Unix file system, the Andrew file system, and the Coda file system. In each file system, tracing was run at four levels. The default tracing level records all of the operations listed in Figure 4 except for read and write system calls and name resolution. We then added read and write calls and name resolution separately. Tracing all activity records all of the operations listed in Figure 4.

Figure 16 shows the elapsed time of the Andrew benchmark for each level of tracing. The overhead ranges between 3–7%, depending on the events traced. Tracing a large compile had lower overhead than the Andrew benchmark, ranging from 3–5%. Figure 17 shows the amount of trace data generated per run of the Andrew benchmark, again at four tracing levels and in three file systems. Background system activity accounts for the variability in the amount of data generated. The benchmark under AFS generated more data because of longer pathnames and longer fids. The benchmark under Coda generated more data than under AFS because the Coda cache manager operates at user level as opposed to within the kernel. Thus its activity is captured in the trace.

Tracing Level	**Elapsed time (min:sec)**					
	UFS	%	**AFS**	%	**Coda**	%
off	2:20 (6)	0%	3:18 (6)	0%	3:40 (5)	0%
default	2:26 (4)	4.2%	3:22 (4)	2.0%	3:47 (3)	3.1%
default, read/write	2:25 (0)	3.5%	3:19 (1)	0.5%	3:52 (6)	5.4%
default, name res	2:29 (1)	6.4%	3:26 (3)	4.0%	3:54 (5)	6.3%
all	2:27 (1)	5.0%	3:25 (3)	3.5%	3:55 (9)	6.8%

This table presents the elapsed time of the Andrew benchmark for four different tracing levels and on three different file systems. The mean of three runs is given in minutes and seconds. The standard deviation in seconds is given in parentheses. UFS is the local Unix file system. AFS is the Andrew file system. The percent slowdown for each trace level is calculated as $100 \times (t_{\text{traced}} - t_{\text{off}})/t_{\text{off}}$. The benchmark was run on a DECstation 3100. File caches were warm for the AFS and Coda results.

Figure 16: Tracing Overhead for the Andrew Benchmark

Tracing Level	**Data (bytes)**		
	UFS	**AFS**	**Coda**
default	512813 (3281)	561356 (3782)	614924 (509)
default, read/write	765381 (159)	818880 (1473)	1099680 (5745)
default, name res	1329764 (1265)	1556908 (1360)	1578433 (1272)
all	1622268 (24020)	1848928 (11727)	2087077 (2598)

This table shows the amount of data generated in bytes for four different tracing levels on three different file systems during the Andrew benchmark. Each entry is the mean of three runs. Standard deviations are given in parentheses.

Figure 17: Volume of Trace Data Generated during the Andrew Benchmark

7.2 Importance of Kernel Implementation

In section 2.1, we stressed the importance of good performance in a long-term tracing system. We used this requirement along with application transparency to justify a kernel implementation of DFSTrace. But is a kernel implementation strictly necessary to satisfy this requirement?

To answer this question, DFSTrace was reimplemented using a toolkit for interposing code between applications and the UNIX system call interface as part of Jones' Ph.D. thesis[22, 23]. The toolkit allows the kernel code and agent to be replaced by an out-of-kernel *interposition agent* and *log merge server*. Instances of the toolkit agent run as part of user programs. Each toolkit agent constructs trace records and sends them to the log merge server, which creates a single trace for the host and sends it to the collector. Because tracing is performed in user space, the interposition agent and the log merge server must synthesize information that is normally obtained from kernel data structures, such as the system time, file identifiers, file attributes, and process information. The interposition agents make additional system calls, such as `gettimeofday` and `getuid` to obtain this information. In addition, since name resolution is transparent to user-level processes, the toolkit implementation must traverse the name space explicitly using `lstat` to produce name resolution records.

The two implementations were compared along several dimensions, including code size and modularity, implementation time, and performance using the Andrew benchmark. The two implementations were comparable in code size. The toolkit implementation was considerably more modular, requiring changes to only 60% as many files as DFSTrace. The toolkit implementation required no changes to existing kernel files. Implementation time using the toolkit was an order of magnitude less than DFSTrace, primarily because the final content of the records had already been determined, and the latter involved building, debugging, and maintaining kernels.

Performance of the toolkit implementation was an order of magnitude worse than DFSTrace, ranging from 64–138% slowdown, compared to the 3–7% in the original. Most of the slowdown in the toolkit implementation is attributable to additional system calls the toolkit agent must make to construct equivalent log records. These results reaffirm our decision to gather data in the kernel, avoiding the performance penalty of repeated crossings of the system interface boundary.

8 Applications

DFSTrace has proven to be invaluable for a variety of purposes. Our original goal was to answer questions about the Coda file system. Since then, DFSTrace has been applied to a number of other areas. In this section, we discuss the uses of DFSTrace in four areas – in trace-driven simulation, as a diagnostic tool, as an instrument for exploration, and as the basis of extensions for understanding low-level system behavior.

8.1 Simulation Studies

Trace-driven simulation has been used to evaluate many aspects of computer systems, such as paging and CPU scheduling algorithms. The virtues of trace-driven simulation, in particular credibility and reproducibility of results, are well known[24]. In this section, we present some of the simulation studies conducted using traces generated by DFSTrace.

8.1.1 Cache Size for Disconnected Operation

The first serious use of DFSTrace was for a simulation of the file cache manager in the Coda file system. One of the questions that arose during the development of Coda was how large a file cache would be needed to support disconnected operation for a day[25]. An analysis based on traces from five active Coda workstations calculated a high-water mark of disk usage for the file cache of approximately 30 MB. Thus a portable computer with a 50-60 MB disk would be adequate for operating disconnected for a twelve hour day. The analysis was later extended to cover a five-day work week[26]. Ten of the most active traces were selected from over 1700 for which on-line summaries were available at the time. The maximum cache space usage for the full week traces was less than 100 MB, and the median was less than 50 MB.

8.1.2 Log Space Requirements for Directory Resolution

Information on long-term file reference behavior was needed during the design of the Coda resolution subsystem[21]. Coda supports replication, and uses an optimistic replica control strategy that allows updates in any network partition. The resolution subsystem is responsible for detecting and classifying partitioned updates to directories, and merging them if they do not conflict. A log-based strategy to support resolution was being considered, in which each server would maintain a history of directory updates it performed during a partition. A concern was whether or not the logs would consume excessive space on the servers. Since a log grows linearly with work done during the partition, any realistic estimate of log size had to be derived from empirical data. A feasibility study was conducted to determine average and peak log growth. A total of 44 AFS and Coda volumes were studied in traces from 20 workstations over a 10 week period. Long-term log growth was only 94 bytes per hour per volume on average, and peak hourly growth rates were less than 10KB for over 99.5% of the data points. Thus a 20KB log would be sufficient for most hour-long partitions. This estimate was confirmed by data gathered from the implementation in actual use, which showed that 99% of the logs grew less than 240KB per day[27].

8.1.3 Improvements Due to Prefetching

Traces were used to estimate the performance improvements possible for TIP[28, 29], a system which exploits application-supplied hints about future I/O activity to reduce file read latency. Experiments were conducted with several applications, including a `make` of an X windows calculator tool. The make program was augmented with a prefetching process, which read exactly the files needed. By using traces, perfect accuracy of future file access could be achieved to estimate the maximum performance gain.

8.1.4 Reintegration Latency

The `untrace` facility described in Section 5.4 was used for evaluating reintegration latency in Coda. The experiments replayed traces of the Andrew benchmark and a large compile. The results of these experiments suggested that typical one-day disconnections would take about one minute to reintegrate, and typical work-week disconnections would take about five minutes[26].

8.2 DFSTrace as a Diagnostic Tool

In complex system software, performance problems often mask bugs. In this section, we describe how the tracing system was useful as a diagnostic tool for discovering problems with systems and software.

8.2.1 Performance Tuning

Tracing of read and write system calls has been useful for profiling the I/O activity of RVM[30], a package providing persistent virtual memory. RVM manages recoverable storage in unstructured *segments*, which are backed by files or disk partitions. Tracing helped uncover a serious performance problem in mapping of large segments into memory; the read buffers being used were too large and were causing the system to thrash. Tracing has also been useful as a diagnostic tool for understanding the I/O behavior of incremental log truncation in RVM.

8.2.2 Mobile Client Configuration

Tracing has been used for more mundane tasks, such as determining which programs should be installed on the local disk of portable machines (as opposed to fetched into a file cache), and discovering problems with tracing clients. If a client generated large amounts of data (over 50 MB/day) it was almost invariably because something was wrong. For example, one new client generated over 400 MB of data in a single weekend. An examination of a few of the traces showed that the machine had a large mail backlog, which the mailer was attempting to rectify with enthusiasm. The primary user of the machine maintained a mailing list, but he had not noticed that some of the addresses were no longer valid.

8.2.3 Application Debugging

The traces have also been useful debugging aids. For example, we have discovered several applications that do not close all of the files they open. Because of per-process limits on the number of open files, this bug eventually rendered the application unusable. In another case, we found a bug causing our file servers to crash because of a piece of code that depended on the system time to be non-decreasing. Although this seems like a reasonable assumption, the traces showed otherwise. Another process on the machine, a daemon running the Network Time Protocol[31], periodically adjusts the system time. Occasionally, these adjustments decrease the time.

8.3 DFSTrace as an Exploratory Tool

In designing system software, it is important to know which operations are the common ones. Therefore, understanding user behavior is critical to designing reliable, high performance systems. In this section, we describe how traces were useful in providing realistic examples of user behavior for evaluating file systems.

Traces inspired the *micromodels* used by SynRGen[32], a synthetic file reference generator. A micromodel is a program that captures the file reference activity exhibited by an application. For example, a general reference pattern for a C compiler is reading a .c file, reading some number of .h files, and creating a .o file. One can create a parameterized micromodel of a C compiler that takes as input the number of .h file referenced, and the names of the .c, .h, and .o files. By combining micromodels, one can create a synthetic user that can be used as a benchmark for comparing systems, or as a test program. New releases of the Coda file system are tested in this manner. The quality of the references generated by SynRGen depends on the accuracy of its micromodels. Using traces allows the modeler to obtain a respectable degree of realism while still retaining the flexibility of a parameterized model. Traces were used to develop SynRGen micromodels for activities in an edit/debug cycle. These models, when compared to the activity generated by real users, came within 20% of the mean values for most system variables.

8.4 Study of Low-Level I/O Behavior

This section describes extensions to DFSTrace for recording low-level system events. Although this work was not part of our original implementation, it demonstrates that DFSTrace is relatively easy to extend, and is adaptable to the needs of other researchers.

8.4.1 Unix Buffer Cache Diagnosis

DFSTrace has been used as a diagnostic tool in understanding UNIX I/O behavior during the development of TIP. A key component of UNIX I/O is the kernel buffer cache, which contains copies of recently used disk blocks[33]. DFSTrace was extended to record buffer cache activity in addition to file reference data. The file reference data is used to identify and separate sources of low-level activity (e.g., user vs. system activity). The buffer cache traces contain records for read hits and misses, read ahead hits and misses, buffer releases, and prefetches by TIP.

8.4.2 Disk Geometry

Tsao extended DFSTrace to record SCSI disk I/O operations for his work in determining disk geometry[34]. Because SCSI exports a linear block address space, one cannot always determine the location of a disk block based on its address. Tsao gathered traces of I/O operations from a known workload, and developed a tool to analyze timing patterns between operations in the trace of the workload. Based on these patterns, his tool infers a variety of information about the disk, such as the disk cache size, number of heads, rotational period, number and location of spare sectors, and track and cylinder skew. This kind of tool is valuable for measurement studies that employ disks because it allows the performance of a disk to be diagnosed independent of the application and operating system.

8.4.3 Field Reliability Test

The SCSI extensions to DFSTrace have enabled its use in a two-year field reliability test of Seagate disks in an AT&T 6299 disk array[35]. Every I/O to the disk array controller is recorded as an enqueue and completion event. If a disk fails, the data will be sent along with the disk back to the manufacturer. It is important that there are no data losses in this application. The authors hope to achieve this in two ways. First, only SCSI events are being recorded, and the records are small (approximately 24 bytes). Second, an additional level of buffering is used at the client, allowing up to 10 MB to be stored at the client on disk.

8.4.4 Isolation

One of the disadvantages of traces is lack of flexibility[36]. In particular, the effects of multiprogramming are embedded in traces and are often difficult to remove. One might want to extract the records for a particular process or set of processes, and use them as if they were the only processes running on a machine. Patterson extended DFSTrace to record context switches and process times[37]. This allows an *extraction* of a trace, such as the records for a specific process, to be used as a workload with accurate timing between events.

9 Related Work

The value of empirical file usage data was recognized long ago. Data on file references has been collected and used for many aspects of file system design over the last two decades[59]. Broadly, there are two methodologies for collecting trace data.

Early file reference data was collected *statically*, by taking one or more snapshots of the file system. The principle advantage of static collection is that it does not require modifications to the file system or operating system. If the system software is proprietary, this approach may be the only feasible one. Often, data can be obtained using existing tools such as accounting or backup programs. A disadvantage of static collection is that there is no way to determine how many times a file has been accessed between snapshots. The bodies of statically collected data are summarized in Figure 18(a). Strange's data is the only set collected from a distributed environment; earlier data was collected in timesharing or batch environments. Irlam's data was obtained through an Internet survey in which he supplied a script that snapshots local file systems and gathers statistics on file sizes.

Most recent data is collected *dynamically*, using continuous monitoring. Numerous bodies of data have been collected on individual machines under a variety of operating systems. Some collections include snapshots to eliminate edge effects during analysis. Most of the data, listed in Figure 18(b), was collected from timesharing environments. There are a few bodies of dynamically collected data from distributed workstation environments; they are listed in Figure 18(c). Hisgen's data was collected at DEC SRC from Firefly[60] workstations running Taos, which provides an Ultrix emulation interface. Baker collected traces only on four Sprite[61] file servers, however, she also collected two weeks of summary data from clients. In contrast, DFSTrace has enabled collection of much longer term data (two years) in a distributed environment.

Year	Collector	System	Duration	Notes	References
1993	Irlam	100 ws, 650 fsys		survey	38
1992	Strange	6 Sprite fsys, 1 SunOS/NFS fsys	76-84 sn	distributed	39
1982	Lawrie	1 CDC NOS	233 sn		7
1981	Satyanarayanan	1 PDP-10, TOPS-10	1 sn		40
1977	Stritter	2 IBM 360, 370, MVS	˜13 months of sn		4
1975	Revelle	2 IBM 360, MVS	144 sn		41

(a) Static collections

Year	Collector	System	Duration	Notes	References
1994	Griffioen, Appleton	2 SunOS ws	2-4 weeks	rw	42
1992	Miller	1 UNICOS, others	2 years	used sys logs	43
1991	Bozman	2 CMS	analyzed 1 day/user	sn, rw	44
	Jensen	2 UNICOS, others	3 years	used sys logs	45
	Muller, Pasquale	1 4.3 BSD Unix	9 75-minute periods	rw, other	46
	Schilit	1 SunOS	3 traces, 33-86 hours	used audit trail	47
1990	Biswas, et. al.	1 VMS	9-12 hour periods	sn, rw	48, 49, 50
	Korner	1 4.2 BSD Unix	not specified		51
	Staelin	2 IBM MVS	1 week, 3 days	used SMF	52
1988	Burrows	1 4.2 BSD Unix	3 work days	+2 weeks sn	53
1986	Floyd	1 4.2 BSD Unix	1 week	sn	10, 54
	Majumdar, Bunt	1 4.1 BSD Unix	1 month, 1984		11
1985	Ousterhout	3 4.2 BSD Unix	2-3 days		2
	Zhou	1 4.2 BSD Unix	9 hours	rw	55
1982	Porcar	2 IBM OS, TSO	9, 13 days	used SMF	3

(b) Local, dynamic collections

Year	Collector	System	Duration	Notes	References
1994	Kuenning	1-10 fs, DOS/Unix	7-10 weeks		56
	Dahlin, et. al.	1 NFS fs, 237 ws	7 days	used net monitor	57
1993	Mummert, Satyanarayanan	˜ 30 Mach ws, Coda fs	over 2 years, 1991–1993	ns, some rw	
1992	Blaze	1 NFS fs, many ws	1 week	used net monitor	58
1991	Baker, et. al.	˜ 40 Sprite ws, fs	8 24-hour periods	+2 weeks summary	1
1990	Hisgen	˜ 100 Taos ws	4 days, Feb. 1990		

(c) Distributed, dynamic collections

This table summarizes sources of file reference data. We use the following abbreviations: sn (snapshot), ws (workstation), fs (file server), fsys (file system) rw (includes read/write operations), ns (includes name resolution operations). Static collections, derived from periodic snapshots, are shown in part (a). Snapshots are taken daily. Part (b) shows dynamically-collected data from single machines, while part (c) shows data from distributed systems.

Figure 18: Sources of File Reference Data

The sets of file system operations recorded varied between studies. For example, most studies did not record individual read and write system calls, because the data would be too voluminous. Exceptions are DFSTrace, Bozman, Biswas et. al., Zhou, and Muller and Pasquale. The latter also recorded other low-level events, as does an extended version of DFSTrace. Other than DFSTrace, only Burrows recorded name resolution, and that was in one set of data that is no longer available[62].

We list several sets of data that were not recorded at the system call interface, but still represent empirical data on file usage. Miller's and Jensen's data from supercomputing environments consists of activity to archival or mass storage systems, gleaned from existing system logs. Blaze's system, NFSTrace, is one of several packages that monitors the network for NFS traffic, and then generates a plausible series of file system events that would result in the observed traffic. The resulting trace is an approximation of file system activity. Dahlin, et. al. also used NFSTrace to collect their traces.

Figure 18 shows that most file reference data was collected in academic and research environments. Exceptions are Biswas, et. al., who collected traces from seven different commercial sites including a large newspaper company and a machine parts distribution company; Bozman, who collected data from an IBM programming center; Staelin, who collected data from two Amdahl customer sites; Porcar, one of whose data sets was collected from an installation at Hughes Aircraft; and Kuenning, whose data is the only set we know of that captures a DOS workload. Unfortunately, little data from other environments is publicly available.

Most of the dynamic studies cited provide few details on the tools used to collect the data. A few used existing monitoring tools, such as SMF[63], audit trail facilities, system logs, or network monitors. The remainder of the efforts involved instrumenting the operating system. This is a feasible approach particularly in Unix environments because of the availability of source code.

Performance is an issue in dynamic collection efforts because tracing runs continuously. This issue is critical in long-term collections. Of course, if existing system logs or off-site monitors are used, there is little or no overhead incurred by gathering the data. Only a fraction of the studies report information on performance. Burrows reported an increase of CPU utilization of less than 2%, Biswas, et. al. reported less than 1%, and Muller and Pasquale reported less than 5%. Appleton estimates the CPU overhead of his package at 2%[64]. A more meaningful expression of tracing overhead is slowdown. Korner, who used a package by Simonetti[65] for her study, reported a 50% system slowdown. Zhou reported slowdowns of 7.7–10% for I/O intensive programs, and 2–4% for CPU intensive programs. In contrast, DFSTrace incurred a 3–7% slowdown for a file system intensive benchmark. In practice, the performance degradation was unnoticeable.

It is important to limit local resource use by tracing for several reasons. First, use of local resources such as disk files may perturb the data, because the activities of the tracing system are recorded in the trace itself. Second, users may be unwilling to sacrifice significant amounts of local resources to store the data, especially in the long term. Third, in long-term collections it is impossible to store all of the data likely to be generated locally. For these reasons, we chose to buffer trace data in a fixed amount of memory, and ship it to a remote collection site. To our knowledge, none of the studies cited in Figure 18 except DFSTrace placed limits on local resource use. Only DFSTrace, Muller and Pasquale, and Griffoen and Appleton[64] used remote collection sites.

In summary, DFSTrace is the only tracing system that has enabled long-term collection of detailed file reference trace data in a distributed workstation environment. Its low overhead both in terms of performance and local resource use were critical for successful long-term data collection. Our emphasis on long-term data collection has made DFSTrace unique in several other respects. Versioning of both data and software, and interchangeability of components simplify the logistics of collecting and handling long-term data. Detection and recording of data losses was necessary because of limits on local resource use and distribution of the collection mechanism. Both of these constraints were consequences of the desire for long-term data.

10 Conclusion

DFSTrace is a system that has proven its worth over the last several years. Its design pays careful attention to efficiency, extensibility, and the logistics of long-term data collection in a distributed workstation environment. The need for long-term data from a distributed environment influenced many aspects of the design of DFSTrace. Low overhead and limits on local resource use are critical in long-term data collection. The separation of data gathering from interpretation is key for good performance and extensibility. Practical considerations such as versioning of data and software and interchangeability of components simplify the logistics of collecting and handling long-term data.

The importance of long-term data cannot be understated. Much of the work mentioned in Section 8 would not have been possible without data of the detail and length that DFSTrace generates. DFSTrace is the only system that we know of that provides data that meets these requirements. We are confident that it will continue to be valuable for future research in data storage systems.

Acknowledgements

We owe special thanks to Jay Kistler for being our first user, along with all that entails, and for writing the initial versions of the `sessions`, `untrace`, and `creplay` programs. We are grateful to members of the Coda project for enduring early versions of DFSTrace and using the data in their research, and the many members of the Carnegie Mellon School of Computer Science for allowing us to trace their file references. We wish to thank Hugo Patterson for first extending DFSTrace to record I/O events, and Garth Gibson and his students for encouraging the use of DFSTrace for I/O related studies. Maria Ebling, Kathryn Porsche, and Mirjana Spasojevic were helpful in improving the presentation of this paper.

References

1. Mary G. Baker, John H. Hartmann, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurement of a Distributed File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 198–212, October 1991.
2. John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Knupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985.
3. Juan M. Porcar. *File Migration in Distributed Computer Systems*. PhD thesis, University of California, Berkeley, July 1982.
4. Edwin P. Stritter. *File Migration*. PhD thesis, Stanford University, March 1977.
5. Alan Jay Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, 7(4):403–417, July 1981.
6. Alan Jay Smith. Long Term File Migration: Development and Evaluation of Algorithms. *Communications of the ACM*, 24(8):521–532, August 1981.
7. D.H. Lawrie, J.M. Randal, and R.R. Barton. Experiments with Automatic File Migration. *IEEE Computer*, 15(7), July 1982.
8. M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39(4), April 1990.
9. M. Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access. *IEEE Computer*, 23(5), May 1990.
10. Rick Floyd. Short-Term File Reference Patterns in a UNIX Environment. Technical Report TR 177, Department of Computer Science, University of Rochester, March 1986.
11. Shikharesh Majumdar and Richard B. Bunt. Measurement and Analysis of Locality Phases in File Referencing Behaviour. In *Proceedings of Performance '86 and ACM SIGMETRICS 1986 Joint Conference on Computer Performance Modelling, Measurement and Evaluation*, May 1986.
12. M. Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC Distributed File System: Principles and Design. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 35–50, December 1-4 1985.
13. Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1985.
14. Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 2(3):181 – 197, August 1984.
15. Department of Electrical Engineering Computer Systems Research Group, Computer Science Division and Berkeley Computer Science, University of Californa. *Unix Programmer's Manual Reference Guide*. USENIX Association, April 1986.
16. Mike Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, Atlanta, GA, July 1986.
17. S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer Conference Proceedings*. USENIX Association, 1986.
18. Richard A. Floyd and Carla Schlatter Ellis. Directory Reference Patterns in Hierarchical File Systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), June 1989.
19. M. Satyanarayanan (editor). *RPC2 User Guide and Reference Manual*. School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, October 1991.
20. John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
21. Puneet Kumar and M. Satyanarayanan. Log-Based Directory Resolution in the Coda File System. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 202 – 213, January 1993. Also available as technical report CMU-CS-91-164, School of Computer Science, Carnegie Mellon University.
22. Michael Blair Jones. *Transparently Interposing User Code at the System Interface*. PhD thesis, Carnegie Mellon University, September 1992.
23. Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

24. S. W. Sherman and J. C. Browne. Trace Driven Modeling: Review and Overview. In *Proceedings of the ACM-SIGSIM Symposium on the Simulation of Computer Systems*, pages 201–207, June 1973.

25. James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

26. James J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, School of Computer Science, Carnegie Mellon University, April 1993.

27. Brian D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 138 – 149, Nashville, TN, May 1994.

28. R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *Operating Systems Review*, 27(2), April 1993.

29. R. Hugo Patterson and Garth A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, September 1994.

30. M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), February 1994.

31. D.L. Mills. Internet Time Synchronization: The Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–93, October 1991.

32. Maria R. Ebling and M. Satyanarayanan. SynRGen: An Extensible File Reference Generator. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 108 – 117, Nashville, TN, May 1994.

33. Samual J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

34. Steven Tsao. Disk Geometry and Performance Characteristic. Data Storage Systems Center, Research for Undergraduate Students Program, July 1992.

35. Mark Holland and Rachad Youssef. Personal communication, October 1994.

36. Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc., New York, NY, 1991.

37. R. Hugo Patterson. Personal communication, November 1993.

38. Gordon Irlam. A Static Analysis of Unix File Systems circa 1993. `ftp://cs.dartmouth.edu/pub/file-sizes/ufs93b.tar.gz`, October 1993.

39. Steven Strange. Analysis of Long-Term Unix File Access Patterns for Application to Automatic File Migration Strategies. Technical Report UCB/CSD 92/700, University of California, Berkeley, Computer Science Division, August 1992.

40. M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 96–108, December 1981.

41. Ron Revelle. An Empirical Study of File Reference Patterns. Technical Report RJ 1557, IBM, April 1975.

42. Jim Griffioen and Randy Appleton. Reducing File System Latency Using a Predictive Approach. In *USENIX Summer Conference Proceedings*, pages 197 – 207. USENIX Association, June 1994.

43. Ethan L. Miller and Randy H. Katz. An Analysis of File Migration in a Unix Supercomputing Environment. Technical Report UCB/CSD 92/712, University of California, Berkeley, Computer Science Division, November 1992.

44. G.P. Bozman, H.H. Ghannad, and E.D. Weinberger. A Trace-Driven Study of CMS File References. *IBM Journal of Research and Development*, 35(5–6), September–November 1991.

45. David W. Jensen and Daniel A. Reed. File Archive Activity in a Supercomputer Environment. Technical Report UIUCDCS-R-91-1672, University of Illinois at Urbana-Champaign, Department of Computer Science, April 1991.

46. Keith Muller and Joseph Pasquale. A High Performance Multi-Structured File System Design. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 56–67, October 1991.

47. Carl D. Tait and Dan Duchamp. Detection and Exploitation of File Working Sets. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 2–9, May 1991.

48. Prabuddha Biswas and K.K. Ramakrishan. File Access Characterization of VAX/VMS Environments. In *Proceedings of the 10th International Conference on Distributed Systems*, pages 227–234, May 1990.

49. K. K. Ramakrishnan, P. Biswas, and R. Karedla. Analysis of File I/O Traces in Commercial Computing Environments. In *Proceedings of the 1992 ACM SIGMETRICS and Performance '92 International Conference on Measurement and Modeling of Computer Systems*, pages 78–90, June 1992.

50. Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace Driven Analysis of Write Caching Policies for Disks. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–23, June 1993.

51. Kim Korner. Intelligent Caching for Remote File Service. In *Proceedings of the 10th International Conference on Distributed Systems*, pages 220–226, May 1990.

52. Carl Staelin and Hector Garcia-Molina. File System Design Using Large Memories. Technical Report CS-TR-246-90, Princeton University, Department of Computer Science, February 1990. Appeared in 5th Jerusalem Conference on Information Technology, Jerusalem Israel, Oct 1990.

53. Michael Burrows. *Efficient Data Sharing*. PhD thesis, University of Cambridge, December 1988.

54. Rick Floyd. Directory Reference Patterns in a UNIX Environment. Technical Report TR 179, Department of Computer Science, University of Rochester, August 1986.

55. Songnian Zhou, Hervé Da Costa, and Alan Jay Smith. A File System Tracing Package for Berkeley UNIX. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1985.

56. Geoffrey H. Kuenning, Gerald J. Popek, and Peter L. Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1994.

57. Michael Dahlin, Clifford Mather, Randolph Wang, Thomas Anderson, and David Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150 – 160, Nashville, TN, May 1994.

58. Matt Blaze. NFS Tracing by Passive Network Monitoring. In *USENIX Winter Conference Proceedings*, pages 333 – 343. USENIX Association, January 1992.

59. M. Satyanarayanan. *Distributed File Systems*, chapter 14, pages 353–383. Distributed Systems. Addison-Wesley, second edition, 1993.

60. C. Thacker, L. Stewart, and E. Satterthwaite. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

61. Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

62. Michael Burrows. Personal communication, July 1989.

63. IBM. OS/VS2 MVS System Programming Library: System Management Facilities (SMF). Technical Report GN28-2903, IBM, May 1978.

64. Randy Appleton. Personal communication, July 1994.

65. J.D. Simonetti. A System Call Trace Facility. Technical Report 85/13, State University of New York at Stony Brook, 1985.