NFS Tracing By Passive Network Monitoring

Matt Blaze

Department of Computer Science
Princeton University
mab@cs.princeton.edu

ABSTRACT

Traces of filesystem activity have proven to be useful for a wide variety of purposes, ranging from quantitative analysis of system behavior to trace-driven simulation of filesystem algorithms. Such traces can be difficult to obtain, however, usually entailing modification of the filesystems to be monitored and runtime overhead for the period of the trace. Largely because of these difficulties, a surprisingly small number of filesystem traces have been conducted, and few sample workloads are available to filesystem researchers.

This paper describes a portable toolkit for deriving approximate traces of NFS [1] activity by non-intrusively monitoring the Ethernet traffic to and from the file server. The toolkit uses a promiscuous Ethernet listener interface (such as the Packetfilter[2]) to read and reconstruct NFS-related RPC packets intended for the server. It produces traces of the NFS activity as well as a plausible set of corresponding client system calls. The tool is currently in use at Princeton and other sites, and is available via anonymous ftp.

1. Motivation

Traces of real workloads form an important part of virtually all analysis of computer system behavior, whether it is program hot spots, memory access patterns, or filesystem activity that is being studied. In the case of filesystem activity, obtaining useful traces is particularly challenging. Filesystem behavior can span long time periods, often making it necessary to collect huge traces over weeks or even months. Modification of the filesystem to collect trace data is often difficult, and may result in unacceptable runtime overhead. Distributed filesystems exacerbate these difficulties, especially when the network is composed of a large number of heterogeneous machines. As a result of these difficulties, only a relatively small number of traces of Unix filesystem workloads have been conducted, primarily in computing research environments. [3], [4] and [5] are examples of such traces.

Since distributed filesystems work by transmitting their activity over a network, it would seem reasonable to obtain traces of such systems by placing a "tap" on the network and collecting trace data based on the network traffic. Ethernet[6] based networks lend themselves to this approach particularly well, since traffic is broadcast to all machines connected to a given subnetwork. A number of general-purpose network monitoring tools are available that "promiscuously" listen to the Ethernet to which they are connected; Sun's `etherfind`[7] is an example of such a tool. While these tools are useful for observing (and collecting statistics on) specific types of packets, the information they provide is at too low a level to be useful for building filesystem traces. Filesystem operations may span several packets, and may be meaningful only in the context of other, previous operations.

Some work has been done on characterizing the impact of NFS traffic on network load. In [8], for example, the results of a study are reported in which Ethernet traffic was monitored and statistics gathered on NFS activity. While useful for understanding traffic patterns and developing a queueing model of NFS loads, these previous studies do not use the network traffic to analyze the *file access* traffic patterns of the system, focusing instead on developing a statistical model of the individual packet sources, destinations, and types.

This paper describes a toolkit for collecting traces of NFS file access activity by monitoring Ethernet traffic. A "spy" machine with a promiscuous Ethernet interface is connected to the same network as the file server. Each NFS-related packet is analyzed and a trace is produced at an appropriate level of detail. The tool can record the low level NFS calls themselves or an approximation of the user-level system calls *(open, close,* etc.) that triggered the activity.

We partition the problem of deriving NFS activity from raw network traffic into two fairly distinct subproblems: that of decoding the low-level NFS operations from the packets on the network, and that of translating these low-level commands back into user-level system calls. Hence, the toolkit consists of two basic parts, an "RPC decoder" (`rpcspy`) and the "NFS analyzer" (`nfstrace`). `rpcspy` communicates with a low-level network monitoring facility (such as Sun's *NIT* [9] or the *Packetfilter* [2]) to read and reconstruct the RPC transactions (call and reply) that make up each NFS command. `nfstrace` takes the output of `rpcspy` and reconstructs the system calls that occurred as well as other interesting data it can derive about the structure of the filesystem, such as the mappings between NFS file handles and Unix file names. Since there is not a clean one-to-one mapping between system calls and lower-level NFS commands, `nfstrace` uses some simple heuristics to guess a reasonable approximation of what really occurred.

1.1. A Spy's View of the NFS Protocols

It is well beyond the scope of this paper to describe the protocols used by NFS; for a detailed description of how NFS works, the reader is referred to [10], [11], and [12]. What follows is a very brief overview of how NFS activity translates into Ethernet packets.

An NFS network consists of *servers,* to which filesystems are physically connected, and *clients,* which perform operations on remote server filesystems as if the disks were locally connected. A particular machine can be a client or a server or both. Clients mount remote server filesystems in their local hierarchy just as they do local filesystems; from the user's perspective, files on NFS and local filesystems are (for the most part) indistinguishable, and can be manipulated with the usual filesystem calls.

The interface between client and server is defined in terms of 17 remote procedure call (RPC) operations. Remote files (and directories) are referred to by a *file handle* that uniquely identifies the file to the server. There are operations to read and write bytes of a file *(read, write),* obtain a file's attributes *(getattr),* obtain the contents of directories *(lookup, readdir),* create files *(create),* and so forth. While most of these operations are direct analogs of Unix system calls, notably absent are *open* and *close* operations; no client state information is maintained at the server, so there is no need to inform the server explicitly when a file is in use. Clients can maintain buffer cache entries for NFS files, but must verify that the blocks are still valid (by checking the last write time with the *getattr* operation) before using the cached data.

An RPC transaction consists of a call message (with arguments) from the client to the server and a reply message (with return data) from the server to the client. NFS RPC calls are transmitted using the UDP/IP connectionless unreliable datagram protocol[13]. The call message contains a unique transaction identifier which is included in the reply message to enable the client to match the reply with its call. The data in both messages is encoded in an "external data representation" (XDR), which provides a machine-independent standard for byte order, etc.

Note that the NFS server maintains no state information about its clients, and knows nothing about the context of each operation outside of the arguments to the operation itself.

2. The `rpcspy` Program

`rpcspy` is the interface to the system-dependent Ethernet monitoring facility; it produces a trace of the RPC calls issued between a given set of clients and servers. At present, there are versions of `rpcspy` for a number of BSD-derived systems, including ULTRIX (with the Packetfilter[2]), SunOS (with NIT[9]), and the IBM RT running AOS (with the Stanford enet filter).

For each RPC transaction monitored, `rpcspy` produces an ASCII record containing a timestamp, the name of the server, the client, the length of time the command took to execute, the name of the RPC command executed, and the command- specific arguments and return data. Currently, `rpcspy` understands and can decode the 17 NFS RPC commands, and there are hooks to allow other RPC services (for example, NIS) to be added reasonably easily.

The output may be read directly or piped into another program (such as nfstrace) for further analysis; the format is designed to be reasonably friendly to both the human reader and other programs (such as nfstrace or awk).

Since each RPC transaction consists of two messages, a call and a reply, rpcspy waits until it receives both these components and emits a single record for the entire transaction. The basic output format is 8 vertical-bar-separated fields:

timestamp | *execution-time* | *server* | *client* | *command-name* | *arguments* | *reply-data*

where *timestamp* is the time the reply message was received, *execution-time* is the time (in microseconds) that elapsed between the call and reply, *server* is the name (or IP address) of the server, *client* is the name (or IP address) of the client followed by the userid that issued the command, *command-name* is the name of the particular program invoked *(read, write, getattr*, etc.), and *arguments* and *reply-data* are the command dependent arguments and return values passed to and from the RPC program, respectively.

The exact format of the argument and reply data is dependent on the specific command issued and the level of detail the user wants logged. For example, a typical NFS command is recorded as follows:

```
690529992.167140 | 11717 | paramount | merckx.321 | read | {"7b1f00000000083c", 0, 8192} | ok, 1871
```

In this example, uid 321 at client "merckx" issued an NFS *read* command to server "paramount". The reply was issued at (Unix time) 690529992.167140 seconds; the call command occurred 11717 microseconds earlier. Three arguments are logged for the read call: the file handle from which to read (represented as a hexadecimal string), the offset from the beginning of the file, and the number of bytes to read. In this example, 8192 bytes are requested starting at the beginning (byte 0) of the file whose handle is "7b1f00000000083c". The command completed successfully (status "ok"), and 1871 bytes were returned. Of course, the reply message also included the 1871 bytes of data from the file, but that field of the reply is not logged by rpcspy.

rpcspy has a number of configuration options to control which hosts and RPC commands are traced, which call and reply fields are printed, which Ethernet interfaces are tapped, how long to wait for reply messages, how long to run, etc. While its primary function is to provide input for the nfstrace program (see Section 3), judicious use of these options (as well as such programs as grep, awk, etc.) permit its use as a simple NFS diagnostic and performance monitoring tool. A few screens of output give a surprisingly informative snapshot of current NFS activity; we have identified quickly using the program several problems that were otherwise difficult to pinpoint. Similarly, a short awk script can provide a breakdown of the most active clients, servers, and hosts over a sampled time period.

2.1. Implementation Issues

The basic function of rpcspy is to monitor the network, extract those packets containing NFS data, and print the data in a useful format. Since each RPC transaction consists of a call and a reply, rpcspy maintains a table of pending call packets that are removed and emitted when the matching reply arrives. In normal operation on a reasonably fast workstation, this rarely requires more than about two megabytes of memory, even on a busy network with unusually slow file servers. Should a server go down, however, the queue of pending call messages (which are never matched with a reply) can quickly become a memory hog; the user can specify a maximum size the table is allowed to reach before these "orphaned" calls are searched out and reclaimed.

File handles pose special problems. While all NFS file handles are a fixed size, the number of significant bits varies from implementation to implementation; even within a vendor, two different releases of the same operating system might use a completely different internal handle format. In most Unix implementations, the handle contains a filesystem identifier and the inode number of the file; this is sometimes augmented by additional information, such as a version number. Since programs using rpcspy output generally will use the handle as a unique file identifier, it is important that there not appear to be more than one handle for the same file. Unfortunately, it is not sufficient to simply consider the handle as a bitstring of the maximum handle size, since many operating systems do not zero out the unused extra bits before assigning the handle. Fortunately, most servers are at least consistent in the sizes of the handles they assign. rpcspy allows the user to specify (on the command line or in a startup file) the handle size for each host to be monitored. The handles from that server are emitted as hexadecimal strings truncated at that length. If no size is specified, a guess is made based on a few common formats of a reasonable size.

It is usually desirable to emit IP addresses of clients and servers as their symbolic host names. An early version of the software simply did a nameserver lookup each time this was necessary; this quickly flooded the network with a nameserver request for each NFS transaction. The current version maintains a cache of host names; this requires a only a modest amount of memory for typical networks of less than a few hundred hosts. For very large networks or those where NFS service is provided to a large number of remote hosts, this could still be a potential problem, but as a last resort remote name resolution could be disabled or `rpcspy` configured to not translate IP addresses.

UDP/IP datagrams may be fragmented among several packets if the datagram is larger than the maximum size of a single Ethernet frame. `rpcspy` looks only at the first fragment; in practice, fragmentation occurs only for the data fields of NFS *read* and *write* transactions, which are ignored anyway.

3. `nfstrace`: The Filesystem Tracing Package

Although `rpcspy` provides a trace of the low-level NFS commands, it is not, in and of itself, sufficient for obtaining useful filesystem traces. The low-level commands do not by themselves reveal user-level activity. Furthermore, the volume of data that would need to be recorded is potentially enormous, on the order of megabytes per hour. More useful would be an abstraction of the user-level system calls underlying the NFS activity.

`nfstrace` is a filter for `rpcspy` that produces a log of a plausible set of user level filesystem commands that could have triggered the monitored activity. A record is produced each time a file is opened, giving a summary of what occurred. This summary is detailed enough for analysis or for use as input to a filesystem simulator.

The output format of `nfstrace` consists of 7 fields:

timestamp | *command-time* | *direction* | *file-id* | *client* | *transferred* | *size*

where *timestamp* is the time the open occurred, *command-time* is the length of time between open and close, *direction* is either `read` or `write` (mkdir and readdir count as write and read, respectively). *file-id* identifies the server and the file handle, *client* is the client and user that performed the open, *transferred* is the number of bytes of the file actually read or written (cache hits have a 0 in this field), and *size* is the size of the file (in bytes).

An example record might be as follows:

```
690691919.593442 | 17734 | read | basso:7b1f00000000400f | frejus.321 | 0 | 24576
```

Here, userid 321 at client `frejus` read file `7b1f00000000400f` on server `basso`. The file is 24576 bytes long and was able to be read from the client cache. The command started at Unix time 690691919.593442 and took 17734 microseconds at the server to execute.

Since it is sometimes useful to know the name corresponding to the handle and the mode information for each file, `nfstrace` optionally produces a map of file handles to file names and modes. When enough information (from *lookup* and *readdir* commands) is received, new names are added. Names can change over time (as files are deleted and renamed), so the times each mapping can be considered valid is recorded as well. The mapping information may not always be complete, however, depending on how much activity has already been observed. Also, hard links can confuse the name mapping, and it is not always possible to determine which of several possible names a file was opened under.

What `nfstrace` produces is only an approximation of the underlying user activity. Since there are no NFS *open* or *close* commands, the program must guess when these system calls occur. It does this by taking advantage of the observation that NFS is fairly consistent in what it does when a file is opened. If the file is in the local buffer cache, a *getattr* call is made on the file to verify that it has not changed since the file was cached. Otherwise, the actual bytes of the file are fetched as they are read by the user. (It is possible that part of the file is in the cache and part is not, in which case the *getattr* is performed and only the missing pieces are fetched. This occurs most often when a demand-paged executable is loaded). `nfstrace` assumes that any sequence of NFS *read* calls on the same file issued by the same user at the same client is part of a single open for read. The close is assumed to have taken place when the last read in the sequence completes. The end of a read sequence is detected when the same client reads the beginning of the file again or when a timeout with no reading has elapsed. Writes are handled in a similar manner.

Reads that are entirely from the client cache are a bit harder; not every *getattr* command is caused by a cache read, and a few cache reads take place without a *getattr.* A user level *stat* system call can sometimes trigger a *getattr,* as can an `ls -l` command. Fortunately, the attribute caching used by most implementations of NFS seems to eliminate many of these extraneous *getattrs,* and `ls` commands appear to trigger a *lookup* command most of the time. `nfstrace` assumes that a *getattr* on any file that the client has read within the past few hours represents a cache read, otherwise it is ignored. This simple heuristic seems to be fairly accurate in practice. Note also that a *getattr* might not be performed if a read occurs very soon after the last read, but the time threshold is generally short enough that this is rarely a problem. Still, the cached reads that `nfstrace` reports are, at best, an estimate (generally erring on the side of over-reporting). There is no way to determine the number of bytes actually read for cache hits.

The output of `nfstrace` is necessarily produced out of chronological order, but may be sorted easily by a post-processor.

`nfstrace` has a host of options to control the level of detail of the trace, the lengths of the timeouts, and so on. To facilitate the production of very long traces, the output can be flushed and checkpointed at a specified interval, and can be automatically compressed.

4. Using `rpcspy` and `nfstrace` for Filesystem Tracing

Clearly, `nfstrace` is not suitable for producing highly accurate traces; cache hits are only estimated, the timing information is imprecise, and data from lost (and duplicated) network packets are not accounted for. When such a highly accurate trace is required, other approaches, such as modification of the client and server kernels, must be employed.

The main virtue of the passive-monitoring approach lies in its simplicity. In [5], Baker, et al, describe a trace of a distributed filesystem which involved low-level modification of several different operating system kernels. In contrast, our entire filesystem trace package consists of less than 5000 lines of code written by a single programmer in a few weeks, involves no kernel modifications, and can be installed to monitor multiple heterogeneous servers and clients with no knowledge of even what operating systems they are running.

The most important parameter affecting the accuracy of the traces is the ability of the machine on which `rpcspy` is running to keep up with the network traffic. Although most modern RISC workstations with reasonable Ethernet interfaces are able to keep up with typical network loads, it is important to determine how much information was lost due to packet buffer overruns before relying upon the trace data. It is also important that the trace be, indeed, non-intrusive. It quickly became obvious, for example, that logging the traffic to an NFS filesystem can be problematic.

Another parameter affecting the usefulness of the traces is the validity of the heuristics used to translate from RPC calls into user-level system calls. To test this, a shell script was written that performed `ls -l, touch, cp` and `wc` commands randomly in a small directory hierarchy, keeping a record of which files were touched and read and at what time. After several hours, `nfstrace` was able to detect 100% of the writes, 100% of the uncached reads, and 99.4% of the cached reads. Cached reads were over-reported by 11%, even though `ls` commands (which cause the "phantom" reads) made up 50% of the test activity. While this test provides encouraging evidence of the accuracy of the traces, it is not by itself conclusive, since the particular workload being monitored may fool `nfstrace` in unanticipated ways.

As in any research where data are collected about the behavior of human subjects, the privacy of the individuals observed is a concern. Although the contents of files are not logged by the toolkit, it is still possible to learn something about individual users from examining what files they read and write. At a minimum, the users of a monitored system should be informed of the nature of the trace and the uses to which it will be put. In some cases, it may be necessary to disable the name translation from `nfstrace` when the data are being provided to others. Commercial sites where filenames might reveal something about proprietary projects can be particularly sensitive to such concerns.

5. A Trace of Filesystem Activity in the Princeton C.S. Department

A previous paper[14] analyzed a five-day long trace of filesystem activity conducted on 112 research workstations at DEC-SRC. The paper identified a number of file access properties that affect filesystem caching performance; it is difficult, however, to know whether these properties were unique artifacts of that particular environment or are more generally applicable. To help answer that question, it is necessary to look at similar traces from other computing environments.

It was relatively easy to use `rpcspy` and `nfstrace` to conduct a week long trace of filesystem activity in the Princeton University Computer Science Department. The departmental computing facility serves a community of approximately 250 users, of which about 65% are researchers (faculty, graduate students, undergraduate researchers, postdoctoral staff, etc), 5% office staff, 2% systems staff, and the rest guests and other "external" users. About 115 of the users work full-time in the building and use the system heavily for electronic mail, netnews, and other such communication services as well as other computer science research oriented tasks (editing, compiling, and executing programs, formatting documents, etc).

The computing facility consists of a central Auspex file server (`fs`) (to which users do not ordinarily log in directly), four DEC 5000/200s (`elan, hart, atomic` and `dynamic`) used as shared cycle servers, and an assortment of dedicated workstations (NeXT machines, Sun workstations, IBM-RTs, Iris workstations, etc.) in individual offices and laboratories. Most users log in to one of the four cycle servers via X window terminals located in offices; the terminals are divided evenly among the four servers. There are a number of Ethernets throughout the building. The central file server is connected to a "machine room network" to which no user terminals are directly connected; traffic to the file server from outside the machine room is gatewayed via a Cisco router. Each of the four cycle servers has a local `/, /bin` and `/tmp` filesystem; other filesystems, including `/usr, /usr/local,` and users' home directories are NFS mounted from `fs`. Mail sent from local machines is delivered locally to the (shared) `fs:/usr/spool/mail`; mail from outside is delivered directly on `fs`.

The trace was conducted by connecting a dedicated DEC 5000/200 with a local disk to the machine room network. This network carries NFS traffic for all home directory access and access to all non-local cycle-server files (including the most of the actively-used programs). On a typical weekday, about 8 million packets are transmitted over this network. `nfstrace` was configured to record opens for read and write (but not directory accesses or individual reads or writes). After one week (wednesday to wednesday), 342,530 opens for read and 125,542 opens for write were recorded, occupying 8 MB of (compressed) disk space. Most of this traffic was from the four cycle servers.

No attempt was made to "normalize" the workload during the trace period. Although users were notified that file accesses were being recorded, and provided an opportunity to ask to be excluded from the data collection, most users seemed to simply continue with their normal work. Similarly, no correction is made for any anomalous user activity that may have occurred during the trace.

5.1. The Workload Over Time

Intuitively, the volume of traffic can be expected to vary with the time of day. Figure 1 shows the number of reads and writes per hour over the seven days of the trace; in particular, the volume of write traffic seems to mirror the general level of departmental activity fairly closely.

An important metric of NFS performance is the client buffer cache hit rate. Each of the four cycle servers allocates approximately 6MB of memory for the buffer cache. The (estimated) aggregate hit rate (percentage of reads served by client caches) as seen at the file server was surprisingly low: 22.2% over the entire week. In any given hour, the hit rate never exceeded 40%. Figure 2 plots (actual) server reads and (estimated) cache hits per hour over the trace week; observe that the hit rate is at its worst during periods of the heaviest read activity.

Past studies have predicted much higher hit rates than the aggregate observed here. It is probable that since most of the traffic is generated by the shared cycle servers, the low hit rate can be attributed to the large number of users competing for cache space. In fact, the hit rate was observed to be much higher on the single-user workstations monitored in the study, averaging above 52% overall. This suggests, somewhat counter-intuitively, that if more computers were added to the network (such that each user had a private workstation), the server load would decrease considerably. Figure 3 shows the actual cache misses and estimated cache hits for a typical private workstation in the study.

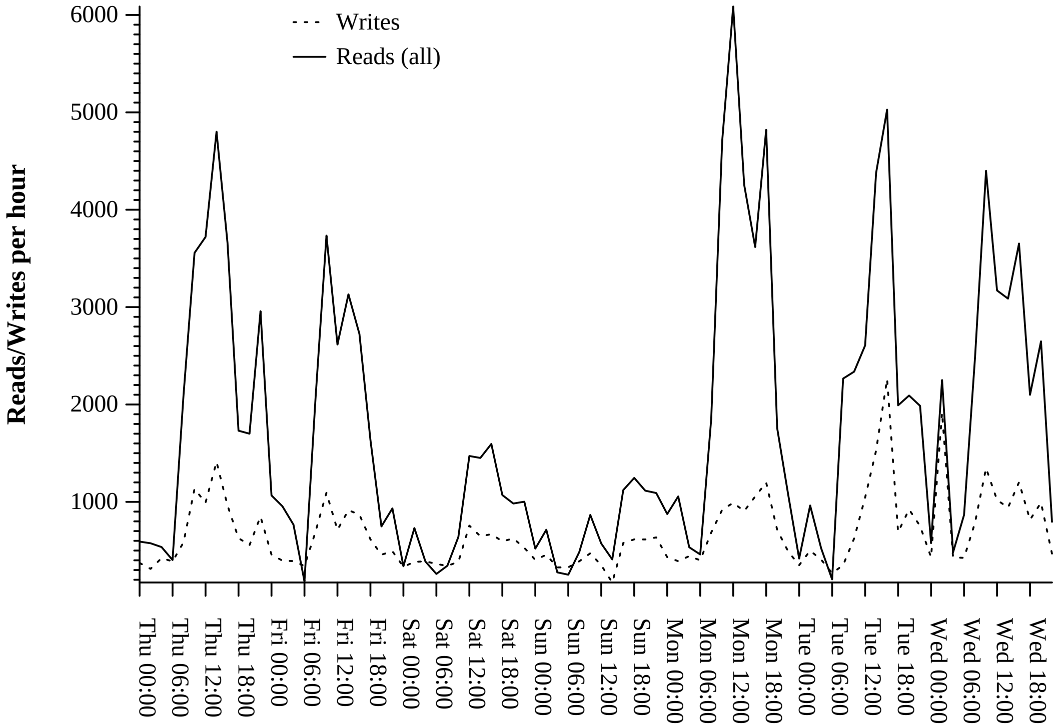

Figure 1 - Read and Write Traffic Over Time

5.2. File Sharing

One property observed in the DEC-SRC trace is the tendency of files that are used by multiple workstations to make up a significant proportion of read traffic but a very small proportion of write traffic. This has important implications for a caching strategy, since, when it is true, files that are cached at many places very rarely need to be invalidated. Although the Princeton computing facility does not have a single workstation per user, a similar metric is the degree to which files read by more than one user are read and written. In this respect, the Princeton trace is very similar to the DEC-SRC trace. Files read by more than one user make up more than 60% of read traffic, but less than 2% of write traffic. Files shared by more than ten users make up less than .2% of write traffic but still more than 30% of read traffic. Figure 3 plots the number of users who have previously read each file against the number of reads and writes.

5.3. File "Entropy"

Files in the DEC-SRC trace demonstrated a strong tendency to "become" read-only as they were read more and more often. That is, the probability that the next operation on a given file will overwrite the file drops off sharply in proportion to the number of times it has been read in the past. Like the sharing property, this has implications for a caching strategy, since the probability that cached data is valid influences the choice of a validation scheme. Again, we find this property to be very strong in the Princeton trace. For any file access in the trace, the probability that it is a write is about 27%. If the file has already been read at least once since it was last written to, the write probability drops to 10%. Once the file has been read at least five times, the write probability drops below 1%. Figure 4 plots the observed write probability against the number of reads since the last write.

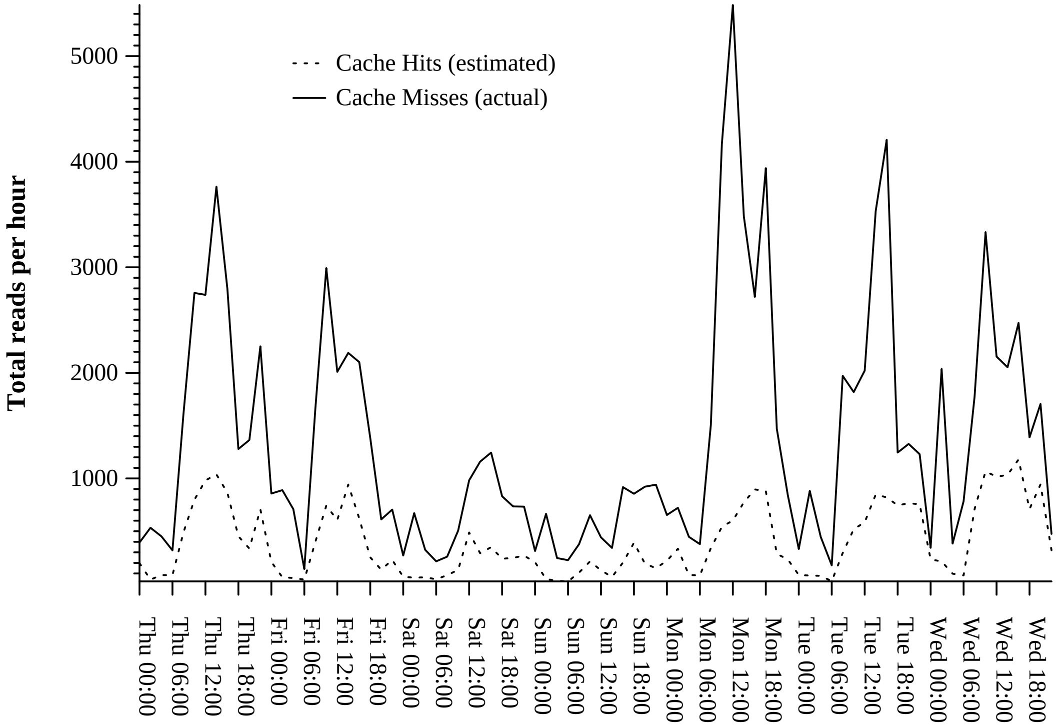

Figure 2 - Cache Hits and Misses Over Time

6. Conclusions

Although filesystem traces are a useful tool for the analysis of current and proposed systems, the difficulty of collecting meaningful trace data makes such traces difficult to obtain. The performance degradation introduced by the trace software and the volume of raw data generated makes traces over long time periods and outside of computing research facilities particularly hard to conduct.

Although not as accurate as direct, kernel-based tracing, a passive network monitor such as the one described in this paper can permit tracing of distributed systems relatively easily. The ability to limit the data collected to a high-level log of only the data required can make it practical to conduct traces over several months. Such a long-term trace is presently being conducted at Princeton as part of the author's research on filesystem caching. The non-intrusive nature of the data collection makes traces possible at facilities where kernel modification is impractical or unacceptable.

It is the author's hope that other sites (particularly those not doing computing research) will make use of this toolkit and will make the traces available to filesystem researchers.

7. Availability

The toolkit, consisting of `rpcspy`, `nfstrace`, and several support scripts, currently runs under several BSD-derived platforms, including ULTRIX 4.x, SunOS 4.x, and IBM-RT/AOS. It is available for anonymous ftp over the Internet from `samadams.princeton.edu`, in the compressed tar file `nfstrace/nfstrace.tar.Z`.

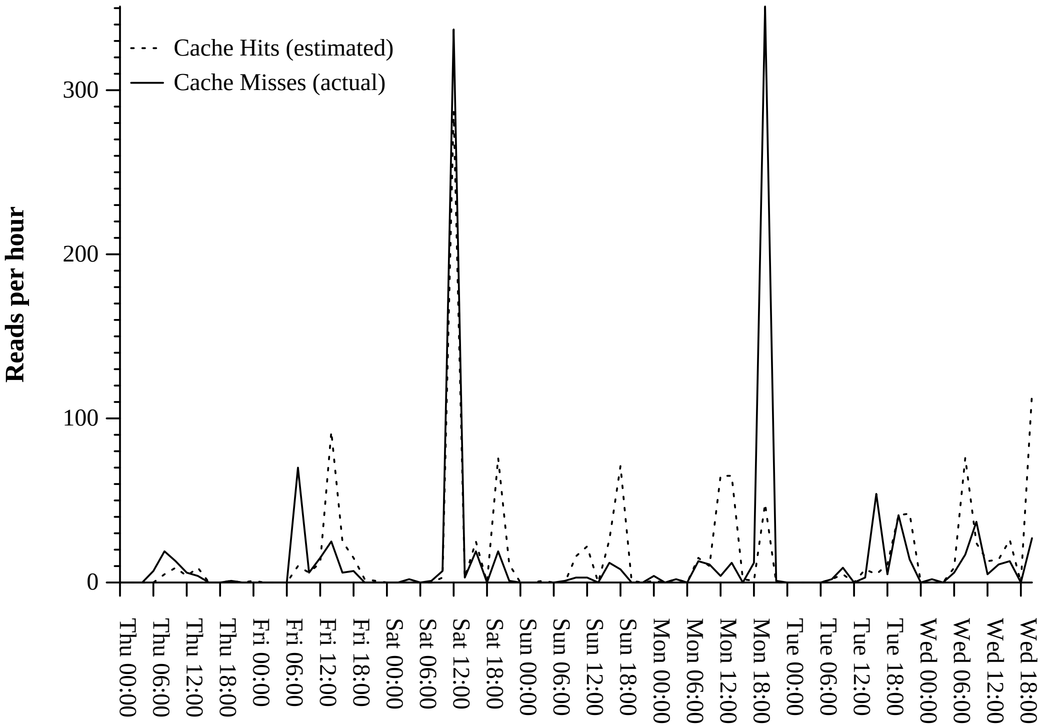

Figure 3 - Cache Hits and Misses Over Time - Private Workstation

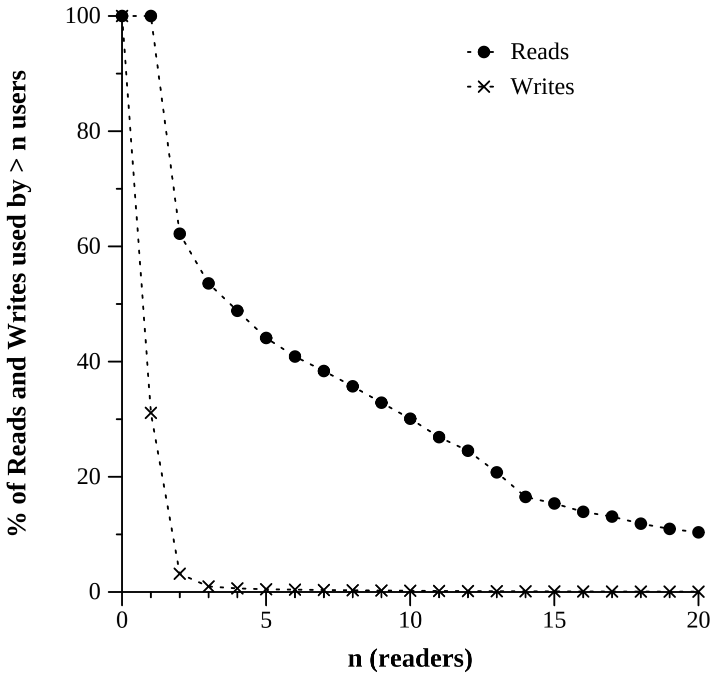

Figure 4 - Degree of Sharing for Reads and Writes

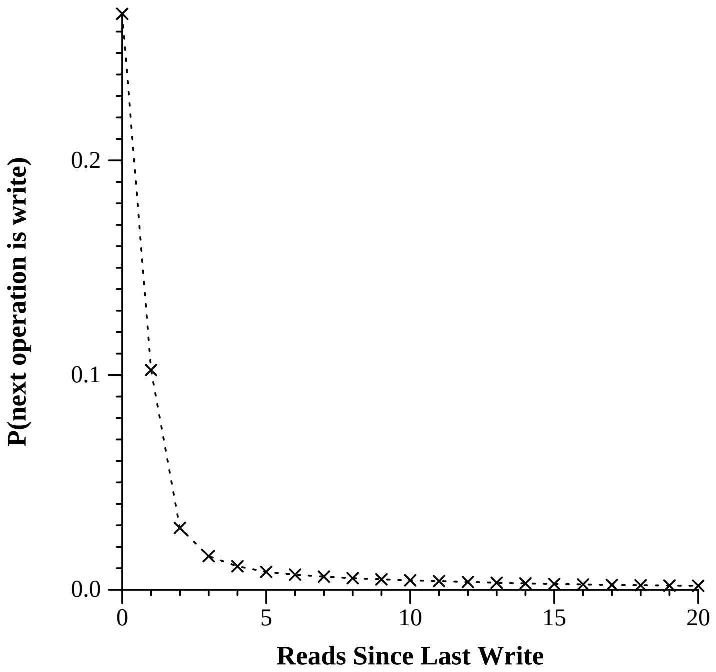

Figure 5 - Probability of Write Given >= n Previous Reads

8. Acknowledgments

The author would like to gratefully acknowledge Jim Roberts and Steve Beck for their help in getting the trace machine up and running, Rafael Alonso for his helpful comments and direction, and the members of the program committee for their valuable suggestions. Jim Plank deserves special thanks for writing `jgraph`, the software which produced the figures in this paper.

9. References

[1] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX,* Summer, 1985.

[2] Mogul, J., Rashid, R., & Accetta, M. "The Packet Filter: An Efficient Mechanism for User-Level Network Code." *Proc. 11th ACM Symp. on Operating Systems Principles,* 1987.

[3] Ousterhout J., et al. "A Trace-Driven Analysis of the Unix 4.2 BSD File System." *Proc. 10th ACM Symp. on Operating Systems Principles,* 1985.

[4] Floyd, R. "Short-Term File Reference Patterns in a UNIX Environment," *TR-177* Dept. Comp. Sci, U. of Rochester, 1986.

[5] Baker, M. et al. "Measurements of a Distributed File System," *Proc. 13th ACM Symp. on Operating Systems Principles,* 1991.

[6] Metcalfe, R. & Boggs, D. "Ethernet: Distributed Packet Switching for Local Computer Networks," *CACM* July, 1976.

[7] "Etherfind(8) Manual Page," *SunOS Reference Manual,* Sun Microsystems, 1988.

[8] Gusella, R. "Analysis of Diskless Workstation Traffic on an Ethernet," *TR-UCB/CSD-87/379,* University Of California, Berkeley, 1987.

[9] "NIT(4) Manual Page," *SunOS Reference Manual,* Sun Microsystems, 1988.

[10] "XDR Protocol Specification," *Networking on the Sun Workstation,* Sun Microsystems, 1986.

[11] "RPC Protocol Specification," *Networking on the Sun Workstation,* Sun Microsystems, 1986.

[12] "NFS Protocol Specification," *Networking on the Sun Workstation,* Sun Microsystems, 1986.

[13] Postel, J. "User Datagram Protocol," *RFC 768,* Network Information Center, 1980.

[14] Blaze, M., and Alonso, R., "Long-Term Caching Strategies for Very Large Distributed File Systems," *Proc. Summer 1991 USENIX,* 1991.

Matt Blaze is a Ph.D. candidate in Computer Science at Princeton University, where he expects to receive his degree in the Spring of 1992. His research interests include distributed systems, operating systems, databases, and programming environments. His current research focuses on caching in very large distributed filesystems. In 1988 he received an M.S. in Computer Science from Columbia University and in 1986 a B.S. from Hunter College. He can be reached via email at mab@cs.princeton.edu or via US mail at Dept. of Computer Science, Princeton University, 35 Olden Street, Princeton NJ 08544.