CACHING IN LARGE-SCALE DISTRIBUTED FILE SYSTEMS

Matthew Addison Blaze

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1993

Abstract

This thesis examines the problem of cache organization for very large-scale distributed file systems (DFSs). Conventional DFSs, based on the client–server model, suffer from bottlenecks when the total client load exceeds the server's capacity. Previous work has suggested that hierarchical client organizations can ameliorate the problem somewhat, but at the expense of a substantial increase in client latency.

An analysis of existing DFS workloads reveals that there is considerable regularity in client file access patterns and that widely shared files lend themselves especially well to caching techniques. In particular, a large proportion of "cache miss" traffic is for files that are already copied in another client's cache. If clients can share these cached files, the server's load can be reduced by a potentially large margin, making larger-scale systems possible.

We introduce the notion of *dynamic hierarchical caching*, in which adaptive client hierarchies are constructed on a file - by - file basis. Trace - driven simulation and workload - driven runs of a prototype file system suggest that dynamic hierarchies can reduce server load substantially without the client performance penalties associated with more static schemes.

Acknowledgments

In doing research, one eventually must come to terms with the idea that absolute perfection is impossible and that "good enough" is just that. There are always more experiments to be done, more cases to check, more references to search, but eventually, one has to write something down and live with it, knowing that you could have done just a little bit better. This, too, is true of the process of acknowledging those who made the work possible. There is always someone left to acknowledge, some more eloquent way to express thanks, but eventually, you write something down, knowing you risk omitting a great deal. Let me apologize in advance to those who I have no doubt slighted.

My advisor, Rafael Alonso, is owed an enormous debt of gratitude for his seemingly boundless support, patience, and time, extending well beyond his tenure at Princeton. My readers, Hector Garcia-Molina and Kai Li, made careful and insightful (and, most importantly, helpful) suggestions that shaped the work described here in many ways. Rafael and Hector were also instrumental in my coming to Princeton to continue graduate school, a move I have not once had cause to regret.

Peter Honeyman of the University of Michigan made many truly valuable comments, in his unique style, over the course of this work. Several aspects of the design of the prototype implementation are due in part to discussions with John Ioannidis of Columbia University. I have benefited greatly from discussions of general file system performance issues with Carl Staelin, now of HP-Labs.

Many of the results in this thesis are based on file system trace data collected at DEC's Systems Research Center. I am extremely grateful to Andy Hisgen at DEC-SRC for making the data available to me. The following people at DEC-SRC

contributed to the file system tracing facility: Susan Owicki, B. Kumar, Jim Gettys, Deborah Hwang, and Andy Hisgen. The actual trace data was gathered by Andy Hisgen.

My friends managed to conspire to make my years at Princeton interesting and enjoyable, and although they are clearly too numerous to enumerate, I will risk naming a few. Adam Buchsbaum, Michael Cox, Michael Golan, Mark Greenstreet, Craig Kolb, Sally McKee, Tom Reingold, Frank Russo, Jennifer Wilson, and Jenny Zhao are among the good friends who have always managed to be there.

In my undergraduate years, Daniel I.A. Cohen of Hunter College provided the encouragement that ultimately led me to graduate school, for which I will remain eternally grateful.

Getting a PhD involves a staggering amount of money, and I would like to acknowledge the generous support of several organizations. An IBM Research Initiation Grant, a Princeton University Dean's Fellowship, and a Graduate Scholarship from the USENIX Association all provided partial support for this work. I am also grateful to Bellcore and Matsushita Information Technology Labs for providing interesting summer environments.

Finally, I would like to thank my parents, Frank and Marilyn Blaze, for their boundless and unwavering love and encouragement. Words do not begin to express my appreciation and love for them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction: Toward Massive Scale Distributed File Systems

Distributed File Systems (DFSs), such as those described in [27] and [14] have gained wide acceptance as a convenient and efficient paradigm for sharing data in local area computer networks. As computing becomes more pervasive, these local networks are becoming much larger and better inter-connected, with disparate groups of thousands of administratively autonomous machines rapidly becoming connected to one another. With this growth of internetworked computing, the DFS paradigm has grown beyond the local network and is emerging as a useful model for large-scale data sharing as well as the local area applications for which it was originally envisioned. Unfortunately, many of the assumptions on which the implementations of current systems are based break down when applied to these emerging massive-scale applications.

This thesis is concerned with data caching in massive-scale DFSs. In this chapter, we discuss the history of the DFS model, and identify some of the assumptions on which current systems are based. In Chapter 2 we describe the instrumentation of a file system for non-intrusively obtaining user workload traces. In Chapter 3 we discuss file access patterns observed in a number of different computing environments and how these patterns affect caching policies. Chapter 4 describes trace driven simulations of conventional "flat" caching schemes and discusses the inherent bottlenecks

when these schemes are applied to large-scale systems. In Chapter 5, "hierarchical" caching schemes which attempt to break these bottlenecks are introduced, and we explore the new problems and inherent tradeoffs in the use of these approaches. Chapter 5 also introduces "dynamic hierarchical" caching schemes, which construct adaptive hierarchies based on usage patterns and address some of the problems with static schemes. In Chapter 6, we describe a prototype distributed file service based on dynamic hierarchies, and discuss issues in the design of a massive-scale system. Finally, in Chapter 7, we conclude with directions for future work in this area.

Some of the results in this thesis have been published previously. The monitoring tools in Chapter 2 were described in [4]. Parts of the results in Chapters 3 and 4 appeared in [4] and [5]. The dynamic hierarchies of Chapter 5, as well as some preliminary trace results, were first outlined in [6].

1.1 Motivation and Background

The first computing systems were completely *centralized*; that is, a single computer served its users autonomously and from a single location. Users went to the computer, and the model of interaction was static and lent itself to such tasks as financial data processing and performing complex scientific calculations. 'Jobs" had to be "submitted" to the computer via such media as punched cards or paper tape and only after a program's turn came up and the run completed could the user see any results of the computation. With the development of inexpensive teletype (and, later, CRT) terminals and timesharing operating systems, however, computing shifted toward a more interactive, less centralized model. Users were no longer required to physically go to the computer for service, instead needing only hook up a remote terminal at their workplace to the central computer. Interactive computing in the workplace led to applications not possible with batch processing, including word processing, on-line databases, and electronic mail services. Since the same information could be made available at terminals located in different places, these systems could be said to be less centralized and more *distributed*, at least from the point of view of the user. The distribution, however, was limited to the number of terminals that could be served

by a single computer.

As computing systems became less expensive and more pervasive, the *workstation* model of computing became a practical alternative to the centralized terminal server [33]. Rather than terminals, users interact directly with their own computers, sharing information over a local area network such as *Ethernet* [19].

Distributed file systems were first developed to provide a convenient and efficient mechanism for the sharing of data in local networks. In a distributed file system, a central *file server* stores the shared files. Machines on the network connect to the file server to obtain these files as they are needed, and usually present a reasonably transparent interface to the user that makes files on the server appear as if they really reside on a local disk.

The number of machines in a local network is inherently rather small, limited by the number of machines that can be connected within a single building or area. Large scale in the first distributed file systems was simply not an important consideration.

As *internetworking* has become more widespread, the desire to share files often spans across individual local networks. For example, it is now common practice to share some files among hundreds or thousands of machines on university campuses and within large corporate entities. With internetworking becoming more global, the desire to share files often extends to many thousands of machines located thousands of miles from one another. Issues of *scale* make the problems of distributed file system design for such global networks somewhat different from systems that support only the local network.

Scale requires the absence of service bottlenecks; that is, no single component of a large scale system should be required to provide service proportional to the size of the system [3]. If a component must provide service proportional to the size of the system, it will eventually bottleneck when the size of the system places a demand that is beyond what we can construct a component to handle. In a distributed file system, the most obvious bottlenecks are the file server and the network to which it is connected.

Scale is a multi-faceted concept, and a rather subjective one at that. Whether a system scales well depends on many factors, not the least of which is what, exactly,

we mean by "scales well". We identify and distinguish among several kinds of scale in this thesis: "population," "traffic," "administrative," and "geographic." Each is concerned with a particular aspect of size, and ability to scale in one way does not guarantee scalability in another. In fact, improving one kind of scalability can sometimes degrade another.

Population scale is the ability of the system to withstand growth in the total number of potential clients, irrespective of the actual usage patterns of these clients. For example, if the file server must maintain persistent state information about each client (for example, about open or cached files), eventually the number of clients will exceed the capacity of the server to maintain their state data. Designing *stateless* servers is one way to improve population scale, although, as we shall see in subsequent chapters, this is can have a negative impact on other aspects of scale.

Traffic scale is the ability of the system to handle the total actual workload of all its clients. A system will not scale well for traffic if all client activity requires server intervention, since eventually the traffic generated by the clients will exceed the server's (or the network's) capacity. The usual technique for improving traffic scale involves having each client maintain a *cache* of previously read files or file blocks. Again, this is not without cost, since the issue of cache consistency for shared files makes it difficult to design stateless servers, and in any case, clients eventually require some server interaction for files not previously read.

The last two aspects of scale, administrative and geographic, concern the ability of the system to span autonomous entities and cover large physical distance. Administrative scale is made difficult by issues of authentication, charging, and system management. Geographic scale raises questions about the ability of the system to operate under failure, partition, and under degraded network conditions.

This thesis is concerned with caching in large scale systems, particularly with respect to population and traffic scale.

1.2 Related Work

There is much current research in distributed file systems, and some of this work has matured to the point of becoming commercial products. It is beyond the scope of this section to conduct and in-depth discussion of the state of distributed file system research. For an excellent survey, the reader is referred to [16].

For the purposes of this thesis, however, it is useful to discuss two projects that have evolved into widely-used commercial systems. In some sense these two systems represent the extremes of a continuum of design decisions for client-server file systems, and are the "state of the art" for their respective approaches in many ways.

The Sun Network File System (NFS) [27] was the first widely accepted distributed file system, and can be viewed as the classic example of a stateless server design. It was originally developed to support local area network workstations, each of which may or may not also have a local disk. Each client can maintain (usually in-memory) a cache of previously read file blocks (the "buffer cache"), but must validate with the server that that file has not changed before each cache use (at file open time). The NFS protocol is discussed in Chapter 2 in more detail. NFS does not scale very well for traffic, since the server must either send a file or verify its status for all client activity, and server load is therefore proportional to the total client workload. It does scale well for population, however, since there is no required state information maintained at the server.

The Andrew File System (AFS) [14] was designed for larger (campus wide) scale. AFS makes extensive use of client caches, which are expected to be larger and possibly disk-based. Server load is greatly reduced (compared with NFS) through the *callback* mechanism, which requires the server to maintain state information about the files cached by its clients. While AFS does do better than NFS in terms of traffic scale, the server still sees a load proportional to the client activity not handled by the caches, which is still proportional to total client activity (it just bottlenecks later). The callback state information also has implications for AFS's ability to scale for large populations. AFS is now a commercial product, and it is regarded as among the most scalable file systems available.

In Chapter 4, we discuss the inherent limits of the scale of these systems in detail.

Chapter 2

Data Collection: Non-Intrusive File System Instrumentation

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 file system activity that is being studied. In the case of file system activity, obtaining useful traces is particularly challenging. File system workload patterns may span long time periods, often entailing weeks or even months of continuously collected trace data. Modification of the file system to collect trace data is often difficult, and may introduce unacceptable runtime overhead. Distributed file systems exacerbate these difficulties, especially when the network is composed of a large number of heterogeneous machines. As a result, only a relatively small number of traces of real file system workloads have been conducted, primarily in computing research environments. Yet much file system research, including this thesis, is based on analysis of and simulation using real trace data.

This chapter describes our portable toolkit for obtaining workload traces by observing client and server network traffic generated by the NFS [27] file system. The non intrusive nature of the toolkit makes it possible to collect long-term traces in environments where other approaches are infeasible or unacceptable.

2.1 Tracing Methodologies

2.1.1 Trace Abstractions

There are a number of different approaches to file system tracing, each yielding a different kind of trace data. Whether a particular methodology is suitable depends, of course, primarily on what is being studied.

The various tracing methodologies can be distinguished by how abstractly they view the behavior of the file system and by whether the trace is static or dynamic in nature.

Static traces are the simplest to conduct; they consist of statistical measurement of a "snapshot" of a working file system at a fixed point in time. The file system may be viewed at either a high level (as a set of files) or a lower level (as a set of records, blocks, disk cylinders, etc). Static analysis is useful to answer questions about such properties as the distributions of file owners, file sizes, access times (if that is recorded in the snapshot), block fragmentation and so forth. Obviously, this approach tells us little about the behavior of the file system over time. If snapshots are taken at regular intervals, however, some of this dynamic information can be traced. See [29] for an example of a study that used static analysis to good advantage.

The most difficult problem in conducting a static trace is obtaining a consistent snapshot without affecting the operation of the file system. If the trace examines all the data in the file system, it may be necessary to suspend user operations while the trace is being taken in order to ensure that an internally consistent copy is made. Obviously, this may not always be practical, especially if the time required to copy all the file system data is significant. In some cases, it may be possible perform static analysis on offline backup copies, since these are not ordinarily needed for user operation. Analysis of backups is only useful, however, if the backup actually contains the data of interest (the precise location of blocks on the disk, for example, is not recorded in many backups schemes).

For many studies, we are more interested in the dynamic behavior of the file system and must look beyond static analysis. A dynamic trace entails logging the sequence of operations performed on or by a file system during a given period. The

Abstraction	*Unix Interface*	*Example Operation*
Disk	Disk Device Driver	`seek`
File System	Vnode Interface	`getattr`
System Call	System Call	`open`
User	Standard I/O Library	`fopen`

Table 1: Levels of Abstraction in Dynamic File System Traces

operations can be logged at several different levels of abstraction depending on the nature of the trace being taken. For convenience, we identify four abstractions that may be logged by a dynamic trace: disk level, file system level, system call level, and user level. The lowest level trace, the disk trace, is be concerned with the operations on the physical disk device (seek, read data, write data, etc). A file system level trace is concerned with the operations performed on the file system by the operating system. A system call trace logs the file system related system calls issued by user processes. The highest level trace, the user trace, logs the file system activity as seen by the user, who usually interacts with the file system through a high-level interface built on top of system calls.

In the Unix operating system, these four levels of abstraction map naturally into four interfaces within that operating system and programming environment. See Table 1. Looked at in this way, a trace at a particular level of abstraction is a log of the operations performed on a particular interface.

Clearly, each level of abstraction exposes and hides different kinds of information, and the volume and granularity of data generated varies with each. Depending on the particular system being traced and the kind of data collected, it may or may not be possible to derive the activity at one level of abstraction from a trace of another. For example, the entries in a file system level trace may not reflect all system call level activity, since the operating system's internal caches and buffers may serve some system calls without file system intervention. If files can be pre-fetched, there may also be file system activity that does not map directly to any system calls.

For the study and simulation file system caching algorithms, we are concerned

primarily with system call level traces. Disk level and file system level traces reflect activity that is highly dependent on the internal architecture and configuration of the system being traced. User level traces, on the other hand, reflect very high level activity that may never, in practice, be passed on to the file system. A system call level trace gives us the actual workload sent to the operating system, and is therefore readily useful for simulating various cache sizes, replacement rules, and cache organizations, as well as direct analysis of the user file access patterns that influence caching performance.

2.1.2 Practical Considerations

To trace at the disk, file system, or system call level generally requires modification of the operating system kernel. Tracing at the user level generally involves modification of the user I/O library (such as Unix's `libc.a`). Regardless of the level being traced, the most difficult problems often concern the management of the volume of data generated. Since the data will generally be logged to a disk, the designer of a tracing system must be careful that the trace can keep up with the workload data. In general, the lower level the trace, the more data will be generated. A parallel issue is performance; it is usually important that the trace software be sufficiently transparent to the user that it is not noticed. This is particularly important if the trace is to be taken over a long time period.

It is beyond the scope of this thesis to discuss the many details of operating system modification for file system tracing. For a detailed example of these issues, the reader is referred to [17] and [2].

2.1.3 Distributed File System Tracing

Distributed File Systems, in which the user (the "client") and the disk (the "server") are on different computers, introduce new problems as well as new opportunities for file system tracing. If we are interested in a system call trace, the problem is much harder: we must modify each client machine on which the system calls run to collect trace data. If we are interested in temporal relationships for files accessed from

more than one machine, the individual client traces must be reconciled and merged together. If the clients are not identically configured machines, the problems become even harder.

File system level tracing, however, may actually be easier in a distributed environment. Since the file system interface is transmitted between client and server over a network, it may be possible to capture this traffic and reconstruct a trace at a useful level of abstraction. This approach has the advantage of being completely independent of the client and server machines; there can be no performance impact if the network can be monitored in a completely passive manner.

Ethernet [19] based networks lend themselves to this approach particularly well, since traffic is broadcast to all machines connected to a given subnetwork. A number of commercially available general-purpose network monitoring tools are available that "promiscuously" listen to the Ethernet to which they are connected; Sun's `etherfind` [10] is an example of such a tool. The `nfswatch` package [9] uses similar techniques to report specific information on NFS packet headers. 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 file system traces. File system 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 [13], the results of a study are reported in which Ethernet traffic was monitored and statistics gathered on file system activity. While useful for understanding traffic patterns and developing models of NFS load patterns, 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 aspects of the individual packet sources, destinations, and types.

In the next section, we describe our toolkit for deriving file system and system call level traces from NFS [27] activity.

2.2 NFS Tracing by Passive Network Monitoring

This section describes a toolkit we constructed 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-initiated 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` [23] or the *Packetfilter* [20]) 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 file system, 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.

2.2.1 The NFS Protocol

It is well beyond the scope of this thesis to describe the protocols used by NFS in detail; for a detailed description of NFS protocols, the reader is referred to [26], [35], [22]. What follows is a very brief overview of how NFS activity translates into Ethernet packets.

An NFS network consists of *servers,* to which file systems are physically connected, and *clients,* which perform operations on remote server file systems as if the disks were locally connected. A particular machine can be a client or a server or both. Clients mount remote server file systems in their local hierarchy just as they do local

file systems; from the user's perspective, files on NFS and local file systems are (for the most part) indistinguishable, and can be manipulated with the usual file system 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 [24]. 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.

2.2.2 `rpcspy`

`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. That is, `rpcspy` produces a file system level trace for an NFS server. There are versions of `rpcspy` for a number of Berkeley Unix-derived systems, including ULTRIX (with the Packetfilter [20]), SunOS (with NIT [23]), 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:

```
2690529992.167140 | 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 below), 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.

2.2.3 rpcspy 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 file system 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.

2.2.4 nfstrace

Although `rpcspy` provides a trace of the low-level NFS commands, it is not, in and of itself, sufficient for obtaining system call-level file system traces. The low-level commands do not by themselves directly reflect system call-level activity. Furthermore, the volume of data that would need to be recorded is potentially enormous, on the order of megabytes per hour. We would prefer to record the system calls underlying the NFS activity.

`nfstrace` is a filter for `rpcspy` that produces a log of a plausible set of system calls 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 file system 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.

2.2.5 Practical Issues

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 [2], Baker, et al, describe a trace of a distributed file system which involved low-level modification of several different operating system kernels. In contrast, our entire file system 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 file system can be problematic.

Another parameter affecting the usefulness of the traces is the validity of the heuristics used to translate from RPC calls into system calls. This was tested with a workload generator 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.

2.3 Princeton Trace Collection

It was relatively easy to use `rpcspy` and `nfstrace` to conduct a week long trace of file system 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` file system; other file systems, 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. Figure 1 shows the configuration of the departmental network with the trace machine attached.

No attempt was made to "normalize" the workload during the trace period. Although users were notified that their 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.

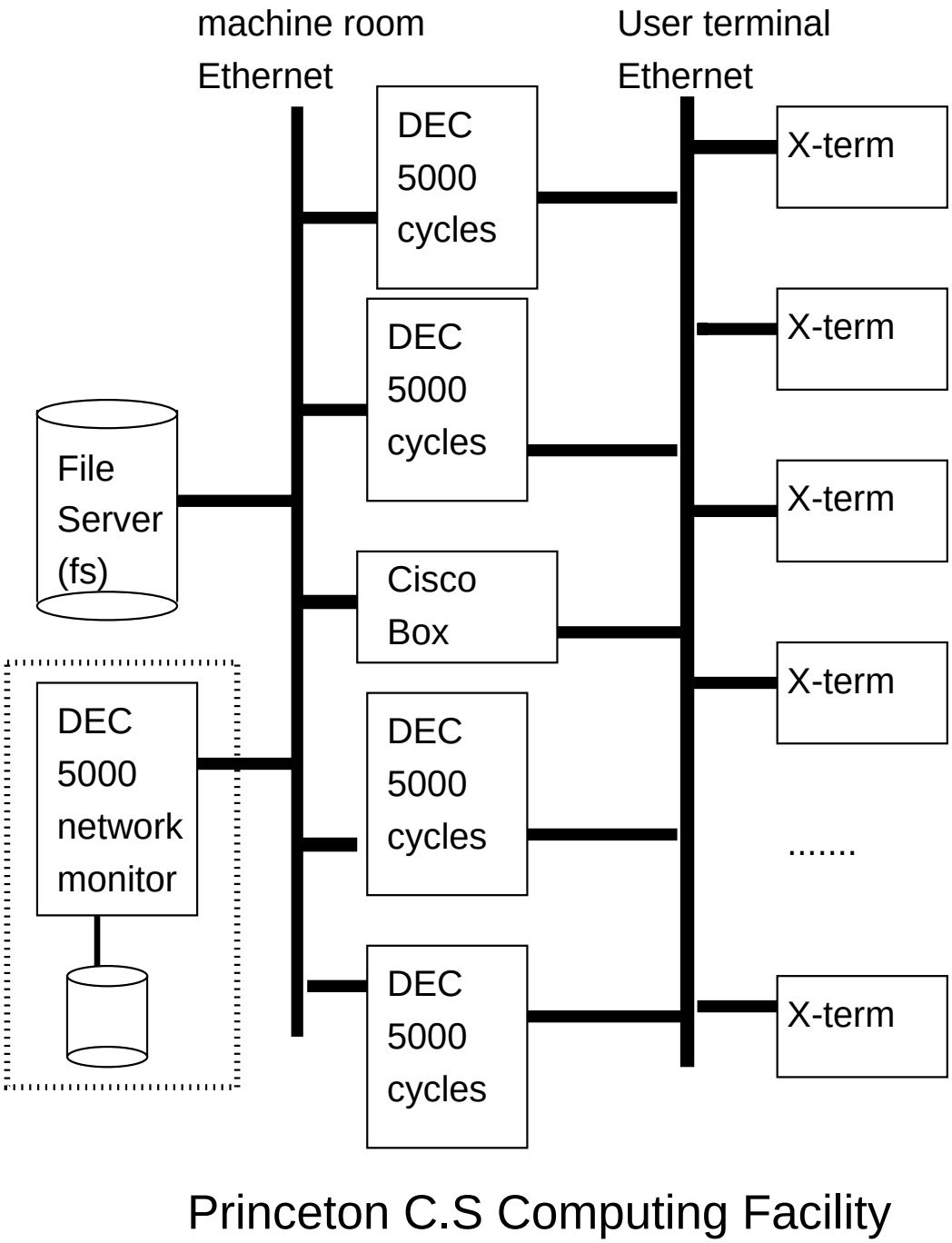

Figure 1: Princeton CS Department Network

2.3.1 Other Traces

In addition to the traces collected at Princeton and elsewhere using `nfstrace`, we were very fortunate to be provided with a trace from Digital Equipment Corporation's Systems Research Center (DEC-SRC). This trace was collected on 112 "Firefly" workstations over a period of four and one half days. Each client kernel (which used a Unix-like operating system written in Modula-2+) was modified to record all file system calls. The entire trace consisted of over three gigabytes of data, at the "system call" level.

Since the DEC-SRC trace will be used to produce results that will be compared with results from the Princeton traces, the data needed to be processed slightly to facilitate direct comparison. First, since the Princeton trace looked only at file opens, all but the file open operations were removed from the DEC-SRC data. This also had the advantage of reducing the volume of data to only about 25 megabytes, which permitted the entire trace to be stored online. The computing environment at DEC-SRC was also different from the Princeton environment in that the former assumed a single workstation per user while the latter relied on four cycle servers with X-terminals as the user interface. This difference can be reconciled by considering each user at Princeton to be a separate workstation for the purpose of simulating a distributed workstation environment like that of DEC-SRC.

The similarity in size and function of the Princeton environment and the DEC-SRC environment make this trace particularly valuable. If the two traces give similar results for a given set of file system metrics this is strong evidence that those metrics are relatively insensitive to slight changes in workload. Furthermore, similar results from traces collected with these two different methods would suggest that both methods are reasonably accurate ways to collect file system data.

2.4 Conclusions

File system workload traces are a valuable tool for many kinds of research and performance analysis. The difficulty of collecting trace data limits their use, however, and relatively few real workloads have been made available to file system researchers.

Traces of distributed file system activity can collected by passive monitoring of the network traffic to the file server. It is possible to use these traces to derive an approximate system call level trace. This chapter described a tool for collecting such traces from NFS systems.

In the following chapter, the traces described above are analyzed for access patterns that affect caching and scale.

Chapter 3

File Access Patterns and Caching

Scale and performance in distributed file systems are closely related to the effectiveness of the caching scheme used by the clients. In a caching scheme, clients place recently accessed data likely to be read in the future in a local storage "cache". When a read occurs, the cache is checked first, and reads are served without server intervention when the data are already there. Usually, data are inserted in to the cache when they are read ("demand caching"), and removed when either no longer valid ("cache invalidation") or when the cache is full and old data need to be discarded to make room for other data (the "replacement rule"). The success of a caching scheme is usually measured by the "hit rate" of the cache – the proportion of data accesses served by reading from the cache. A number of factors contribute to the hit rate, including the size of the client cache, the ability of the replacement rule to predict which data will be read again, and the particular workload presented to the cache.

Workload patterns affect file caching in several ways. Observe that in demand caching a "miss" will occur only under one of two circumstances. First, trivially, when a file is read for the first time it must miss in the cache, since there was no previous opportunity to insert the data into the cache. Second, even if the file has been read previously, it will miss if the replacement rule or invalidation policy caused it to be discarded already. Therefore, the hit rate depends on whether the workload includes multiple reads by the same clients, and whether the pattern of these reads is one that works well for the client replacement rule. Unfortunately, it is difficult to

make strong analytical statements about a cache without knowledge of the workload properties. At one extreme, the hit rate will be 100% if the same data are read over and over and are never thrown out of the cache. At the other extreme, the hit rate will be 0% if each read refers to data never before read by the client.

Even with knowledge of the workload properties, experimental techniques such as trace-driven simulation are generally used to predict the hit rate of a particular caching scheme. There are no known analytic models that can be used to predict the hit rate for a given cache size, replacement rule, and access workload pattern. Before such analytic models could be constructed, we must have a deep insight into how file access patterns interact with caches.

This chapter attempts to identify a number of access patterns that affect the cache hit rate. We do not attempt to construct an analytic model for cache performance nor even to quantify these patterns. Rather, we seek simply to identify in workload traces a number of common access properties that appear to influence cache hit rates. Understanding common access patterns is important for several reasons. It facilitates the construction of accurate *workload generators* [7] [30] that can be used, in the absence of real trace data, to drive simulators or measure the performance of real systems. In the longer term, a better understanding of workloads could lead to useful analytic cache performance models.

3.1 Previous Work

Most previous work on file access patterns has focused on either statistical data on static properties (e.g., the distribution of file sizes), or on relatively short term behavior (e.g., the percentage of files accesses that consist of reading all the bytes of a file in order). This is partially because of the difficulty of obtaining longer-term traces, as discussed in the previous chapter.

In spite of the relatively small number of studies of the dynamic behavior of files over time, a number of file access properties have been observed that are of interest to the cache designer. First, the distribution of files accesses tends to be skewed heavily so that a small percentage of files account for the majority of file I/O [31]

[8] [11] [34]. This is perhaps at the heart of why file system caching is successful at all; a sufficiently small proportion of the file system that can be stored at the client comprises the "working set" of file accesses. A caching or replication strategy should seek to maintain copies of these active files while ignoring the majority of dormant ones.

Most file accesses have been observed to be sequential and of the entire file [17] [2] [31], although the this is somewhat dependent on the attribute information associated with the file [11]. This observation may be partially explained by the limited facilities in the Unix operating system for non-sequential file I/O. If sequential access is known to be common, it may be simpler to cache entire files (or large chunks of files) rather than individual file blocks. Throughput may also be improved by pre-fetching later blocks of a sequentially accessed file when it is opened.

Files in Unix are very likely to be deleted or overwritten soon after they are created [17] [2]. This suggests that it is worth buffering files in the client cache for a short time before writing them back to the server, since the files are likely to be deleted anyway before they are written back.

3.2 Long-Term Access Patterns

A number of access patterns that influence cache performance were observed in the workload traces described in Chapter 2. This section describes these traces in more detail. Later chapters will use these traces to drive simulations of various caching strategies.

3.2.1 Workload Properties

Traffic vs. Time of Day

In both traces, the volume of read and write traffic (or, more precisely, the number of opens for read and write per hour) was highly dependent on the time of day. Figures 2 and 3 plot the hourly read and write traffic in the Princeton and DEC-SRC traces, respectively. Note that opens for read outnumber opens for write by about 3

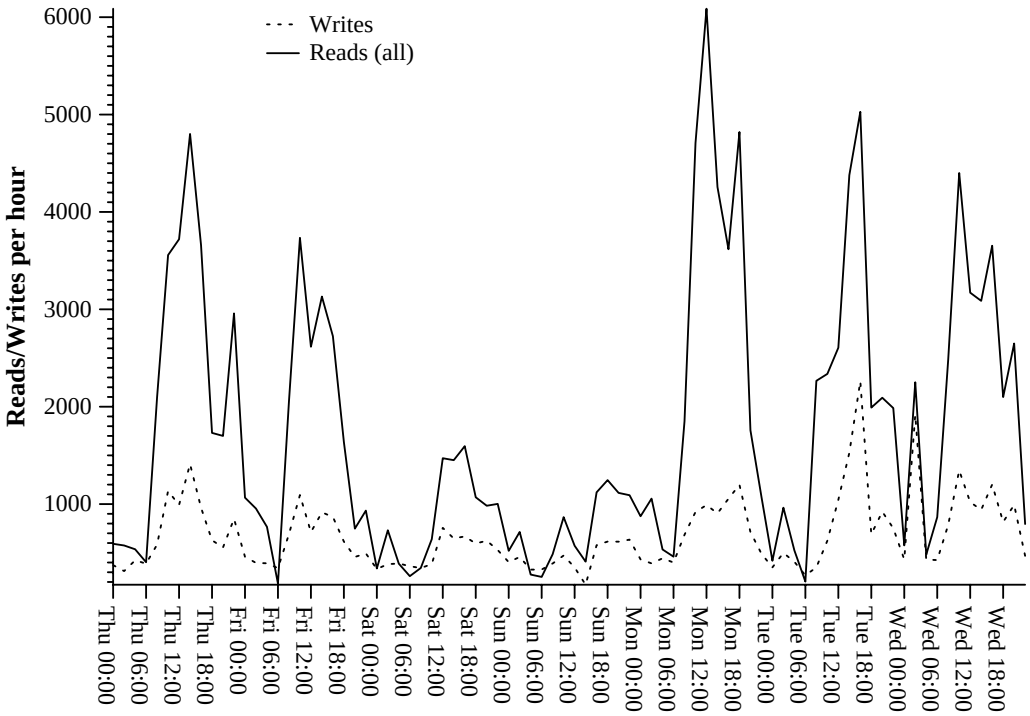

Figure 2: Read and Write Traffic in Princeton Trace

to 1 in the Princeton trace and by about 5 to 1 in the DEC-SRC trace.

Observed Cache Performance

An interesting statistic is the hit rate observed in the machines on which the trace was taken. This data could not be derived from the DEC-SRC trace but could be approximated in the Princeton Trace. Each of the four cycle servers allocated approximately 6 megabytes for the buffer cache. The hit rate observed was surprisingly low: under 23% over the trace week, and never exceeding 40% in any given hour. Figure 4 plots cache misses and cache hits over the period of the trace on the four cycle servers.

Past studies have predicted much higher workstation hit rates than those observed here. It is likely that the low hit rate can be attributed to the large number of users logged in to each cycle server and competing for space in the 6 Megabyte cache. This

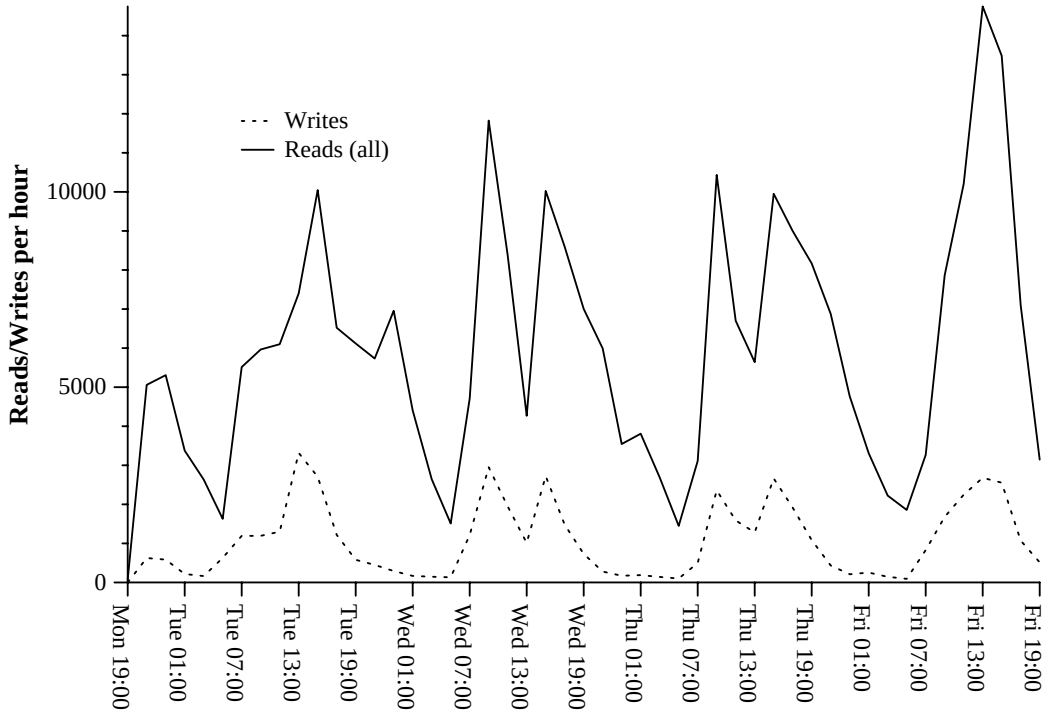

Figure 3: Read and Write Traffic in DEC-SRC Trace

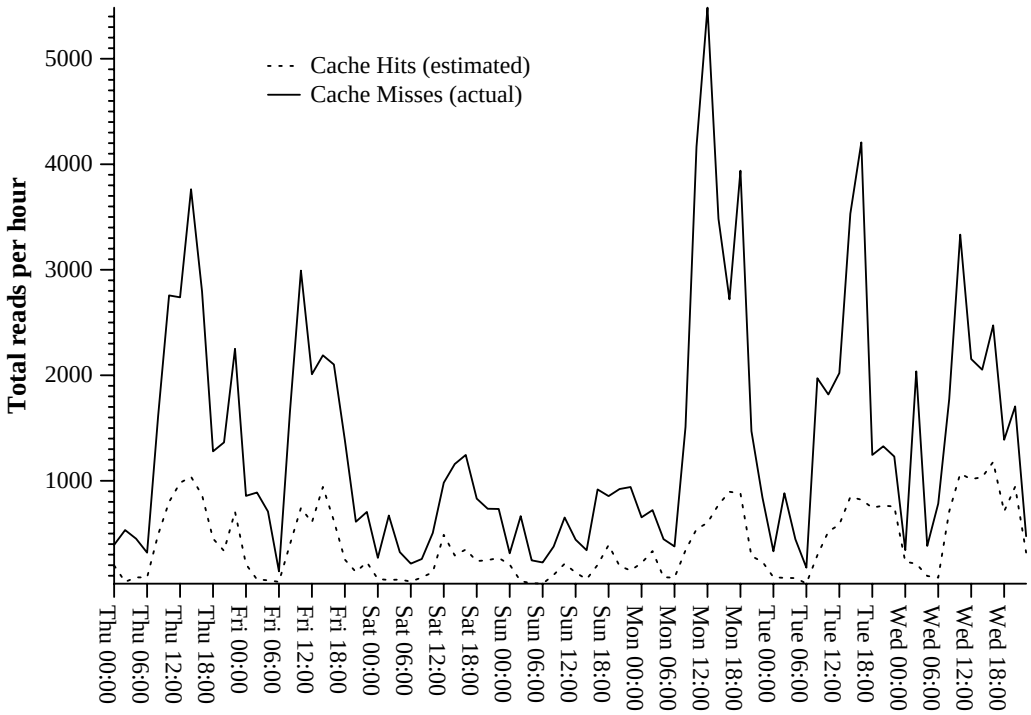

Figure 4: Cache Hits and Misses in Princeton Cycle Servers

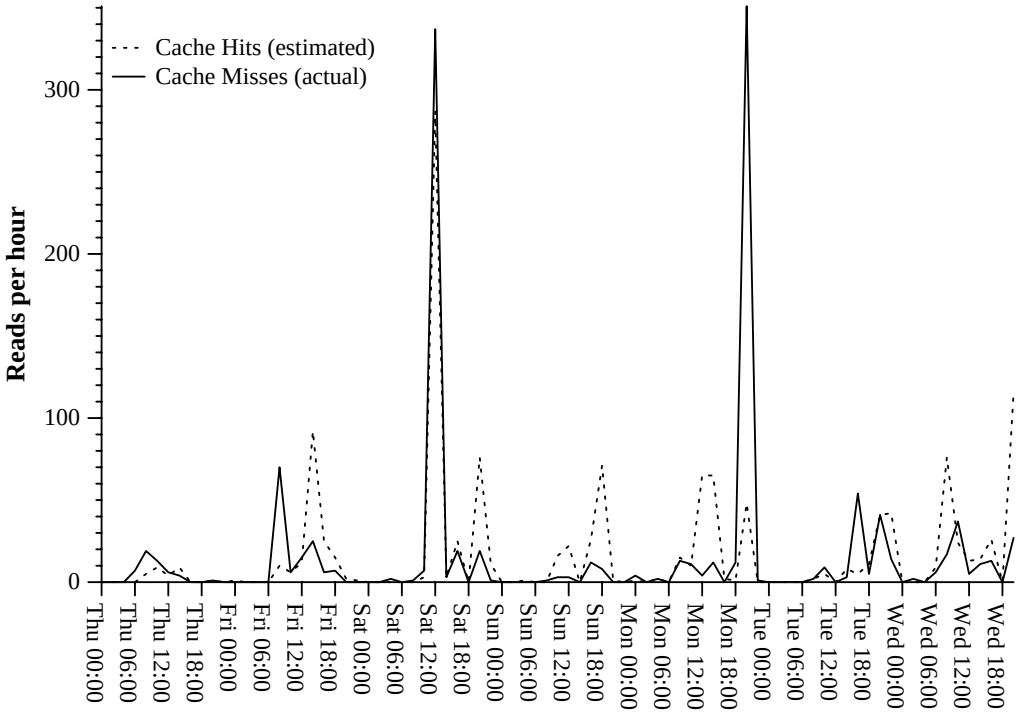

Figure 5: Cache Hits and Misses in Private Workstation

hypothesis is supported by the much higher hit rates observed on similarly configured private workstations on which only one user was logged in. Figure 5 plots the hits and misses on a typical private workstation in the Princeton trace. On this workstation, the hit rate was more than double that of the cycle servers, averaging over 52% over the week.

3.2.2 File "Inertia"

Files in both traces were usually opened in the same mode (read or write) and by the same user (or workstation) as the previous open on that file. That is, the most recent operation on a particular file is a good predictor of the next operation on that file. We can consider each file open to represent a transition between read and write states for the same user or a different user. Tables 2 and 3 show the observed probability of switching between states on file opens in the Princeton and DEC-SRC traces, respectively. Observe that in both traces, between 51% and 70% of reads and

Previous Operation	Next Operation on File			
	Read by Same User	Read by New User	Write by Same User	Write by New User
Read	51.7%	36.3%	10.1%	2.0%
Write	18.7%	10.4%	70.0%	0.9%

Table 2: Next Operation vs Previous Operation, Princeton Trace

Previous Operation	Next Operation on File			
	Read by Same User	Read by New User	Write by Same User	Write by New User
Read	66.2%	28.7%	4.9%	0.2%
Write	30.3%	1.4%	64.0%	4.2%

Table 3: Next Operation vs. Previous Operation, DEC-SRC Trace

writes are followed by the same operation by the same user. This suggests designs optimized for sequences of the same kind of operation by the same user.

3.2.3 Temporal Locality

Intuition suggests that recently read files make up a large proportion of read traffic. In both traces, most opens for read were for files that had already been read by the same user (or workstation) in the very recent past. This implies that caches that can store all files read with a relatively short history will be very effective. Figures 6 and 7 show the percentage of opens for reading by time since the file was last read at that workstation for the Princeton and DEC-SRC traces, respectively. Note that these graphs are assuming a "cold start" in which files read for the first time in the trace are counted even though they may have been read just prior to the beginning of the trace period. The curves indicate the percentage of opens for reading that follow an open for reading on the same machine within t hours. Observe that only two hours of cache data will yield a hit rate over 66% in both traces.

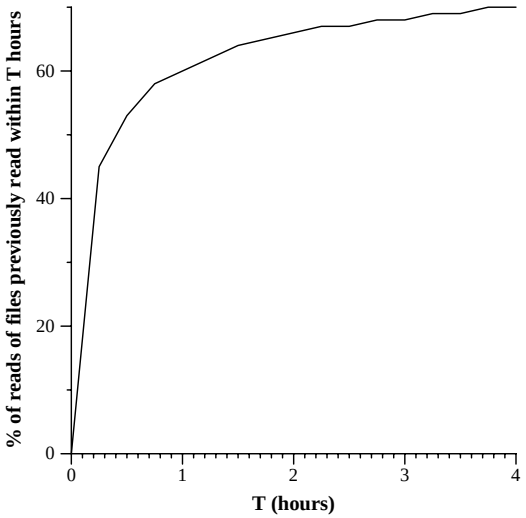

Figure 6: % of Reads of Files Last Read Within T Hours, Princeton Trace

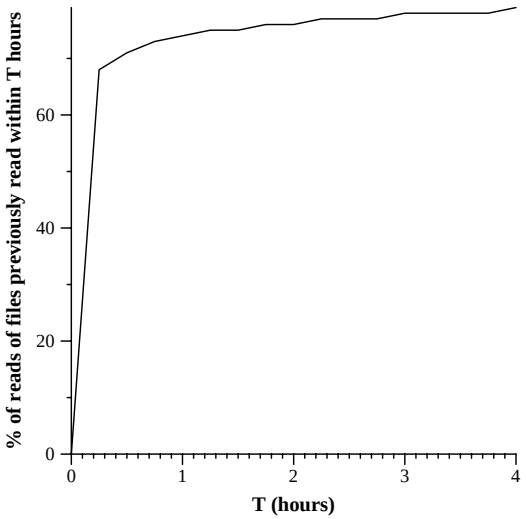

Figure 7: % of Reads of Files Last Read Within T Hours, DEC-SRC Trace

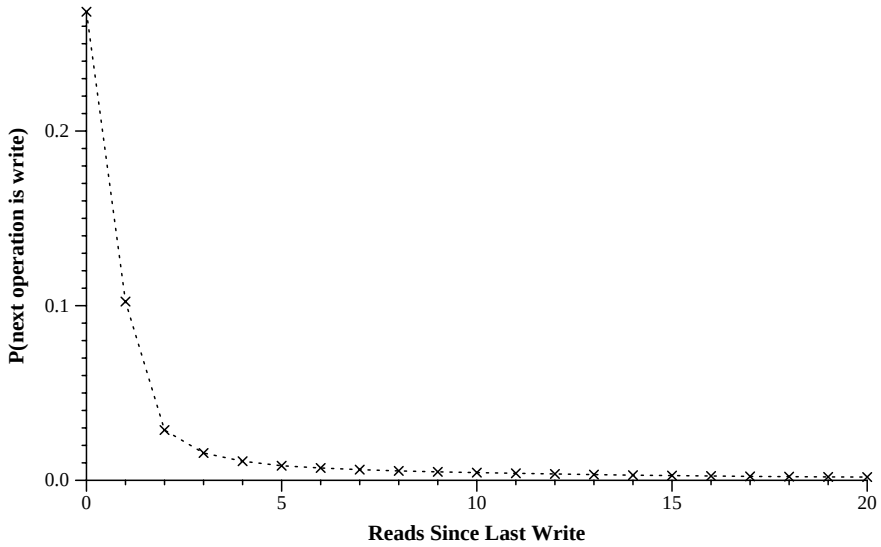

Figure 8: File Entropy, Princeton Trace

3.2.4 File "Entropy"

The likelihood that a file will be overwritten decreases sharply after it has been opened for reading several times. In other words, as a file is read repeatedly, it becomes increasingly likely that the next operation on that file will be an open for reading rather than an open for write or unlink. We call this property "file entropy," since files seem to "move toward" a state of being read-only over time. File entropy was very strong in the Princeton and DEC-SRC traces; in both traces, files that had been opened for reading at least four times were overwritten less than 10% as often as files that had been opened for reading at least once. The vast majority of opens for write are for files that had been read zero or one times since they were created or last written. Figures 8 and 9 show the percentage of opens for writing by the number of previous opens for reading in the Princeton and DEC-SRC traces, respectively.

File entropy allows a reasonable prediction of the "volatility" of a file to be made based on a simple file-by-file counter of opens for reading since last write. This could be useful in a system where, for example, it is possible to make the cost of reading low in exchange for increased write cost for a particular file; entropy suggests that this could be profitable after only a few reads.

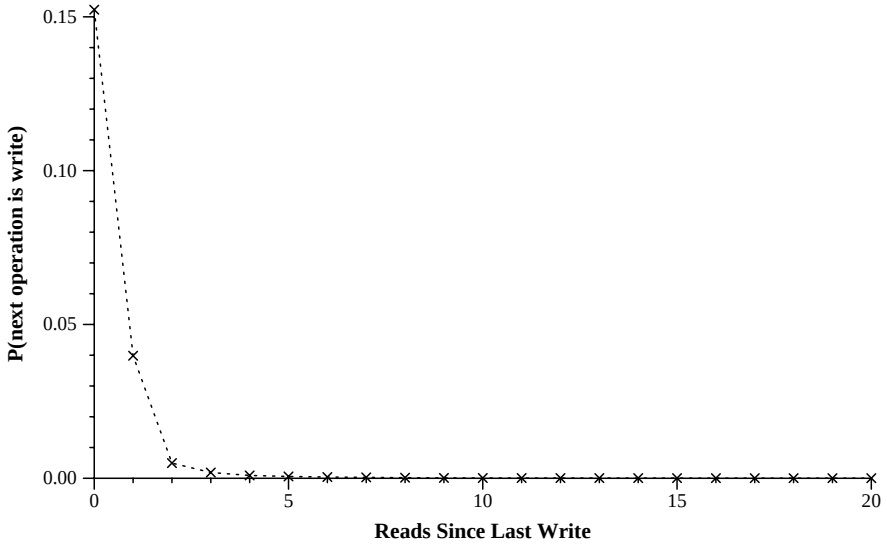

Figure 9: File Entropy, DEC-SRC Trace

3.2.5 Write Behavior of Shared Files

Files that are widely shared by many machines are good candidates for replication or caching only if they are not expected to change often. The difficulty of maintaining consistency among multiple copies is related to how often the copies are expected to change and under what circumstances. Past studies [18] have examined "micro level" questions such as the frequency with which files still open at other machines are updated. It is also interesting to look at the higher-level question of how often files that have ever been used by multiple users get updated.

In both traces, files used by more than one user or workstation made up a large proportion of read traffic but a small proportion of write traffic. In fact, writes of shared files drop off exponentially with the number of previous readers. Figures 10 and 11 show the proportion of opens for read and write of files previously opened by n clients.

This property suggests that a caching or replication scheme in which the cost of writing is proportional to the number of previous readers will do very well. Observe that caching schemes in which clients get a copy the first time they they read a file and are notified by the server should the file change follow exactly this cost model.

The large proportion (over 60%) of read traffic for files already read by several machines is worth noting. This suggests that clients may be able to share cache or

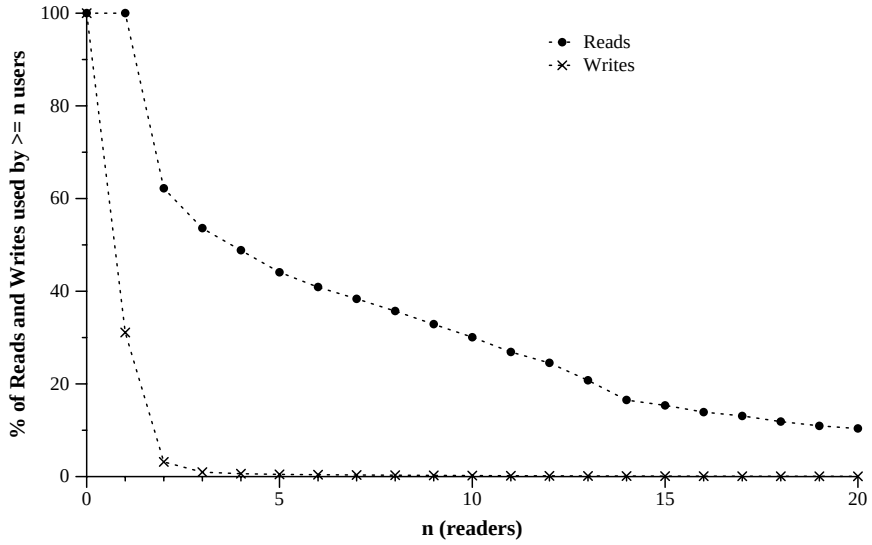

Figure 10: Read and Write Sharing, Princeton Trace

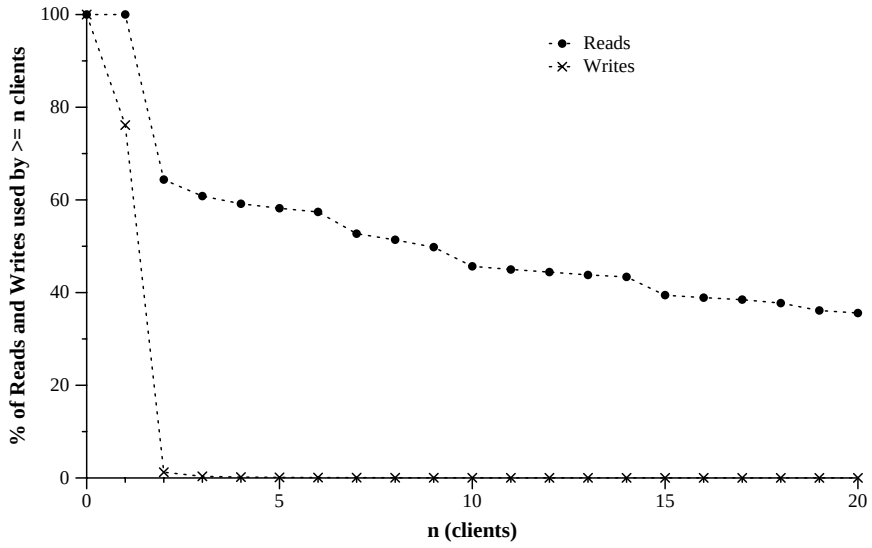

Figure 11: Read and Write Sharing, DEC-SRC Trace

replication data without going to the server as often. We discuss caching schemes that exploit this property in subsequent chapters.

3.3 Conclusions

Understanding real file system workload properties is useful for several purposes. First, common access patterns provide insight for the designers of file system caching and replication policies. For example, the history of a file may be used to predict its future write or sharing behavior, which could influence decisions about whether to cache the file and how to handle consistency. Second, real workload patterns can be incorporated into synthetic workload generators to produce more realistic simulations and performance studies in the absence of real trace data. Finally, better understanding of access patterns may lead to improved analytic models of file system performance, which may replace time-consuming and speculative simulations altogether.

This chapter examined two workload traces taken at research computing environments. Both sites served similar user populations and were similar in scale and scope, but had considerably different computing environments. Several properties, Inertia, Entropy, Temporal Locality and Limited Write Sharing, were common to both environments. The reader is cautioned not to over-generalize from this data, since it is impossible to be certain that these properties apply to other environments or even to these environments at other times. Still, the similarities between the two traces do suggest that the properties observed in the traces may be rather fundamental access patterns, at least among Unix-like systems in research environments. It would be desirable to study sites other than those engaged in computing research to see whether these properties generalize even further.

The following chapter examines how well suited traditional caching schemes are to these workload patterns, and examines the limits of cache performance under the client-server model.

Chapter 4

Flat Caching: Limits of the Client-Server Model

Distributed File Systems implement the "client-server" model; "client" machines direct their file system requests to a "server" machine to which a file system is physically connected. In general, file system semantics require that there be a single server associated with each file system, or at least that it appear this way to the user. In other words, all clients should have a consistent view of the file system, and there should be no doubt as to what constitutes the "valid" version of a file.

The actual implementation of a DFS need not strictly follow the client-server model, however, as long as it presents these semantics to the user. In fact, all current DFSs deviate from a strict client-server implementation somewhat in that clients may maintain a cache of recently-read files which may be used in lieu of the server. If the file is not in the local cache, the server is contacted. Although this complicates the implementation (especially with regard to maintaining consistency), client performance is improved when the cost of reading from the cache is lower than the cost of fetching a file from the server. Note that both read and write operations may be cached, although this thesis focuses only on read caches.

This chapter examines the scalability of systems in which all requests are served either by the clients' cache or by a single server. We call these "flat" systems, since all clients are connected to the one server in a flat, non-hierarchical fashion.

4.1 File System Cost Models

Caching is traditionally thought of as a technique to improve client performance; it can be argued that this is the ultimate goal of system design once functionality is established. The usual metrics of client performance are throughput (the number of operations that can be processed in a given time period) and response time (the average or worst-case time for an operation). There are many factors that influence client performance. Some are mostly fixed and static and therefore rather simple to model coarsely, such as the number of instructions processed per second, the overhead of a system call, the time required to serve a request out of the cache, the cache size, and so on. Others are workload dependent, such as the expected hit rate of the cache and the overhead of maintaining cache consistency, but these can be modeled using relatively simple simulations. In a small scale distributed system, where the server is always able to keep up with the client requests, the cost of a cache miss is more or less fixed and usually low. In these small scale systems, the client hit rate is a probably the most important factor of concern to the DFS designer.

However, in a large-scale system the costs of cache miss traffic and validation overhead are not static. Since the server serves many clients, it has the potential to become a bottleneck when client traffic exceeds server capacity. The client throughput and response time depend on global server traffic. The scale of the system is limited by the server's ability to process those requests not hidden from it by the client caches. In these large-scale systems, we become less concerned with *local* client cache hit rates and more concerned with the *global* client miss rate, which represents the load presented to the server. At first glance these figures seem equivalent, since the hit and miss rates are complementary, yet they represent very different magnitudes. With sufficiently large caches, client hit rates can reach 90% or more [2]. An increase in the hit rate to 95% represents a local performance improvement of less than 6% to the client but a reduction of 50% in the server load. With these high hit rates, even a modest change in the client's ability to cache data can have a large impact on the overall scalability of the system.

The global miss rate does not tell the entire story, however. Cache consistency

validation also affects server load, and can potentially comprise the bulk of the server's work. Validation overhead can manifest itself at the server in two ways: in additional communication traffic between the server and clients (initiated by either) and in state information that must be maintained by the server on behalf the clients.

The sections that follow examine the factors that influence global miss rate and validation overhead, discuss the characteristics of existing systems, and analyze various caching schemes through trace driven simulation. The focus of the analysis is on server load, since that is the bottleneck in large scale systems; no attempt is made to model client response time directly. Because the trace data used to drive the simulations is limited to a log of files opened for reading and writing, server load is measured according to a very simple cost model. Files transferred between server and client cost f units of server time; overhead messages that do not involve file transfer are charged a cost of m. Clients can cache a fixed number of files $0 \leq n \leq \infty$. Note that this model is not useful for predicting the exact load on any real system since the simplifying assumptions hide the difference in cost between transferring small and large files and the cases where less than the entire file is actually used at the client. Nor are any real values associated with the costs; in fact, no assumption is made except that the cost of an overhead message is substantially lower than the cost of a file transfer. The model is intended only to make rough comparisons between approaches with large differences in performance. It is not suitable for fine-grained comparisons between similar algorithms.

4.2 Global Miss Rate: Client Cache Size and Replacement Rule

In a flat caching system, the global miss rate represents the amortized file transfer load presented to the file server. The miss rate is governed by three client parameters: the size of the cache, the placement policy, and the replacement policy. Cache size is usually a fixed number of disk blocks (or some other transfer unit); as noted above, we model this as a fixed number of files for simplicity. The placement policy is the

algorithm the client uses to decide when to put something in the cache. We consider only *demand caching* here, where files are put in the cache only when they are actually read. To improve response time, clients might *pre-fetch* data that are likely to be read in the near future, but this can never achieve a lower miss rate than demand caching. When data are pre-fetched, either they will be used (in which case the same number of transfers as the demand case occurs) or they will not (in which case the the extra transfers are wasted).

The replacement policy is the rule used to select a cache entry for removal when the cache is full and another item is to be added. In a demand caching scheme where all misses carry the same cost it has been shown that the optimal replacement rule (OPT) is one that always replaces the item that will be used the furthest in the future [1]. This policy can only be implemented in an "off-line" model, where the entire sequence of operations is known in advance. Since the future file access pattern is not known, the best that can be done in practice is to use the history of previous file accesses to guess which item is least likely to be used again soon.

It is worth noting that the replacement rule is important only when the cache size is non-zero and finite. When cache size is zero, the miss rate is always 100% and when cache size is infinite, nothing is ever replaced and so the replacement rule is never triggered. Therefore, an interesting statistic about a replacement rule is how large the cache must be under the rule in order to begin to converge on the performance of OPT.

The replacement rules employed in practical systems are generally very simple and are derived from those in virtual memory systems. The most common rule is to replace the least recently used (LRU) item. LRU is easy to implement in software and is believed to perform well in practice.

In trace driven simulations of the LRU and optimal off-line replacement rules, surprisingly small (per-user) caches are able to achieve low miss rates. Figures 12 and 13 show the simulation results in the Princeton and DEC-SRC traces for optimal replacement using cache sizes of between 0 and ∞. These graphs reflect one simulation day of cache "warm up". Note that caching only 256 files per user yields a miss rate of under 16% in both cases, and that the LRU and OPT rules are close to converging

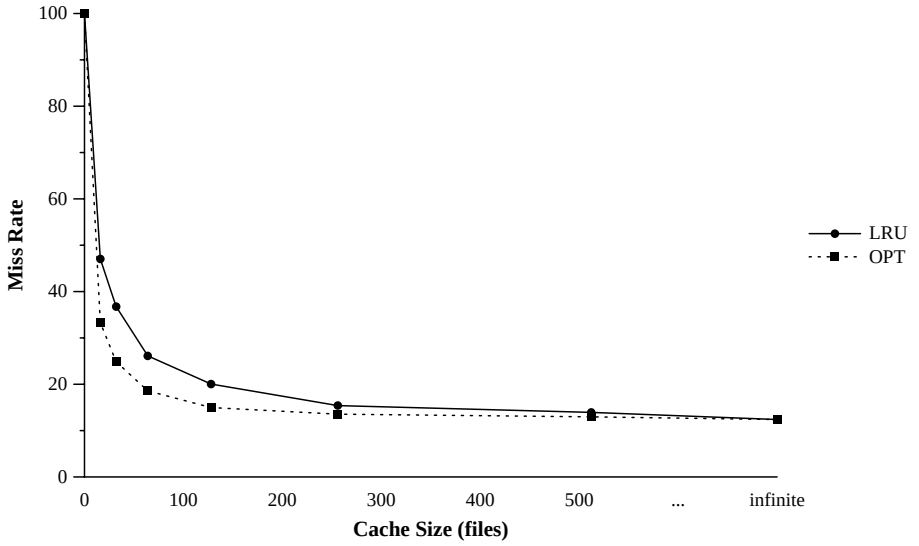

Figure 12: LRU and OPT Miss Rate vs. Cache Size, Princeton Trace

at a cache size of only 512 files.

Files accessed in both traces were small (mean of about 21KB), so a per-user LRU cache of only about 10MB can be expected to perform nearly as well as an optimal infinite cache.

More sophisticated replacement rules have been proposed that use *frequency-based replacement*, favoring frequently used files. These algorithms have been shown to perform up to 20 to 50 percent better (compared against optimal) than LRU for small caches [25]. Given the very good performance of LRU with moderate-size caches, however, any improvement in a frequency-based replacement may be offset by the additional computational and code complexity of the implementation.

The miss rate is theoretically affected slightly by the cache validation scheme in use (see below). However, here was no significant difference in miss rates with different validation schemes in the simulations described above. (The graphs shown reflect a client-driven invalidation scheme; the server-driven invalidate miss rates were within .5% in both simulations).

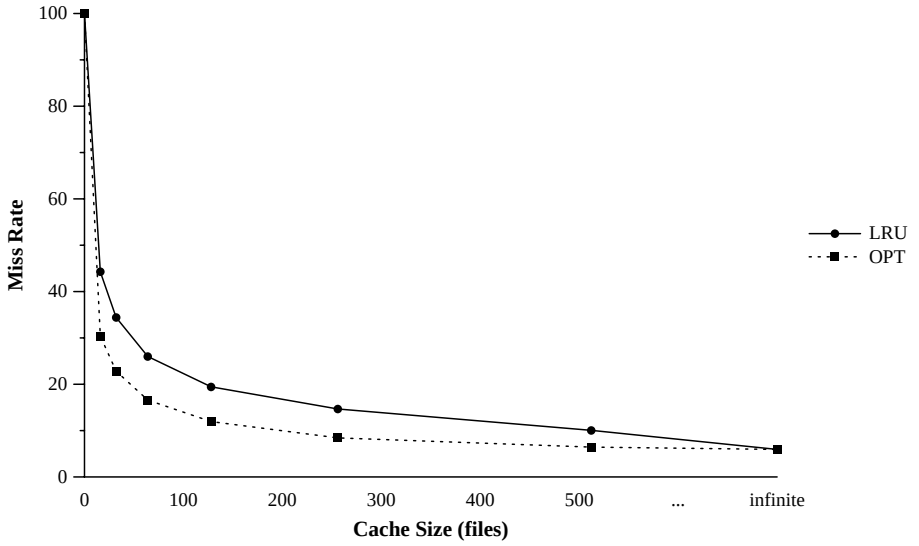

Figure 13: LRU and OPT Miss Rate vs. Cache Size, DEC-SRC Trace

4.3 Overhead: Cache Consistency & Server State

The semantics of most file systems require that all reads reflect the latest version of the data being read. On a non-distributed system or a strictly client-server one in which the same server always handles all client operations, these semantics occur rather naturally. When clients in a distributed system cache files that are not known to be immutable (read-only), however, some protocol is required to ensure that the reads from the cache return the current data. In this section, we examine the communications overhead of various cache validation strategies.

Cache validation can be initiated either by the client or the server. In fact, validation can be viewed as a continuum from strictly client-driven to strictly server-driven, with some algorithms falling somewhere in between.

4.3.1 "Stateless" Servers: Client-Driven Invalidation

The simplest schemes are those in which the client verifies the validity of the cache entries as they are used. That is, for each cache hit, the client verifies that the data has not changed or otherwise become invalid since the time at which it was cached. This requires either that the server maintain a list of cached copies, or, more

commonly, that the server include a "last updated" timestamp with any data it sends to a client. The client can verify the consistency of a copy by asking the server for the current timestamp for the data and comparing it with the cached copy. The server operates in a completely *reactive* mode to client requests and need maintain no state information about its clients. Such servers are said to be *stateless*.

Caching with client-driven invalidation is worthwhile for reducing server load and improving client performance only when the cost of the validation transaction is significantly lower than the cost of fetching the data from the server. The probability that the cached data will, in fact, be valid must also be high, since invalid data must be transferred to the client again anyway.

Observe that the amount of overhead traffic in a client-driven invalidation scheme is directly proportional to the client hit rate.

Client-driven invalidation can be made more attractive by relaxing the consistency requirements slightly. For example, clients could be guaranteed that cached data will never be returned that is more than some time-bound out of date, or that the data were valid at file open time but not necessarily at the time the data are actually read.

NFS [27] is a good example of a relaxed-consistency client-driven invalidation system. Clients query the server for the last update time of a file when a cached file is opened; directory information is verified if the cached data was obtained more than a short time (usually 30 seconds) in the past.

Client-invalidate schemes and stateless server systems have a number of useful properties. The statelessness of the server implies that clients need not "synchronize" with the server; if a server fails, the client can simply wait until the server becomes available again and continue with the next operation. At the same time, the server need not keep track of any information on behalf of its clients (beyond verifying their authority to perform operations on a particular file system). Adding clients is a relatively simple matter. Both client and server code are simple, since all transactions are initiated by the client. Fairly strong statements can be made with regard to consistency, since the client verifies the cache contents at well-defined times.

4.3.2 "Stateful" Servers: Server-Driven Invalidation

Despite the attractive qualities of stateless client-invalidate systems, the large amount of traffic they generate and the heavy load this places on the server can make them impractical for larger-scale systems. If widely shared files are expected to change very infrequently (as observed in the previous chapter), less traffic would be generated if the server assumes responsibility for invalidating the client caches at write time. That is, the invalidation can be server-driven, rather than client-driver.

Server-driven invalidation is not without cost, however. If the network connection between client and server fails, a client could miss an invalidation message, making consistency potentially unreliable and ill-defined. Write performance suffers when many clients must be invalidated, in proportion to the number of previous readers. Write response time can be particularly poor when the server must wait for time outs caused by network or machine failure of one of the clients. Server- invalidation is also much more complex. The server must also maintain a list of clients with cached copies of each file, which imposes potentially large storage requirements, bounded only by the number of potential readers. The clients must be able to handle messages initiated by the server, and the server must have a recovery procedure for when it cannot reach a client with an invalidation message. Similarly, clients require a protocol to verify that they have not lost a message from the server, particularly after a crash.

AFS [14] is perhaps the best-known example of a server invalidate system. Servers maintain a record for each client with a cached copy and issue a *callback* when the file is overwritten. AFS avoids a number of the pitfalls of strict server-driven invalidation, however. Clients may be guaranteed a consistent copy for only a fixed time, after which they must verify the data upon use. If too many clients have copies of a particular file, the server may issue callbacks to reduce the amount of state it must maintain. This makes AFS something of a hybrid client- and server- invalidation system. AFS performs a number of other optimizations based on known file access patterns, such as caching large (64KB) "chunks" of files rather than individual disk blocks, since most programs use entire files anyway.

A number of other systems, including [18] and [32], also attempt to balance simplicity with performance by striking a balance between strict stateless and strict

statefulness. The fundamental tradeoffs, however, remain the same.

4.3.3 Limiting Server State

One of the problems with server-invalidated systems is that the amount of state information the server must maintain is potentially very large. One way to limit the server state is to put a bound on the number of client-file pairs the server is willing to store. This could be on a file by file or global basis. When the state capacity of a file or file system is reached and a new client wants to cache a file, the server selects another client and first "breaks" its promise to invalidate the file for it. Note that this can only increase the server's load; the total number of messages it processes is equal or more than the number in a boundless state system. In fact, the total number of messages in the worst-case is double that of a boundless state system, since each read the server processes could entail invalidating another client. However, the load is spread out more evenly, and write response time is bounded by the maximum number of clients to be invalidated.

Another way to limit server state does not put a direct bound on the number of clients invalidated per file but limits the time that the server guarantees it will provide invalidation messages. The server provides a *lease* [12] on the file when a client makes a cached copy during which time it guarantees that it will "break the lease" if a write occurs. Leases are typically very short (measured in seconds or minutes), and are used primarily in practice to improve the performance of otherwise stateless schemes. Recovery from crashes or unreachable clients is very simple in a short-term leased system, since all the server need do is block writes until any unbreakable leases have expired.

Finally, consider an optimal-off line scheme in which no server state need be maintained and no validation overhead is sent. Such as scheme could be modeled by a lease scheme in which the lease term always expires just before the file is actually overwritten. Obviously such a scheme could not be implemented in practice, although history information could be used to attempt to guess when the file will be overwritten. A "predictive lease" scheme must balance a simple tradeoff: if the lease term is set too long, server state builds up and more invalidation messages must be sent at write

time; if the term is too short, the system reverts to client-invalidation. In [5], a simple predictive lease algorithm outperformed either client- or server- driven invalidation by a small margin.

4.3.4 Validation Overhead Traffic

Since the spectrum of validation schemes runs from strictly client- to strictly server-driven, it is worthwhile to compare the two extremes. Since we are concerned with scale, we model issues of server traffic rather than client response time.

In trace driven simulation of both the Princeton and DEC-SRC traces, overhead traffic was found to be quite large, as expected, for client-driven invalidation with all but the smallest caches. The overhead of server-driven invalidation, on the other hand, is insensitive to client cache size and was very small. The overhead traffic with server-driven invalidation was insignificant compared with the file transfer (miss) traffic even with the largest client caches.

Figures 14 and 15 show the number of overhead messages sent in the Princeton and DEC-SRC simulations for pure client- and pure- server driven schemes with LRU and OPT replacement (after one simulation day of cache warm-up). An overhead message is defined as any message exchange (regardless of direction) not involving an actual file transfer.

If files are known to be immutable, validation can be ignored completely, since cached files are can never be out of date. This approach is often taken for the wide distribution operating system files (which change rarely) among machines in a local network, for example.

4.4 Flat Caching and Scale

A computing system can be said to be scalable only to the extent that that there are no bottlenecks that prevent growth. In the case of client-server distributed file systems, the server (with the network to which it is connected) has the potential to be such a bottleneck. The server must process any request that cannot be handled by

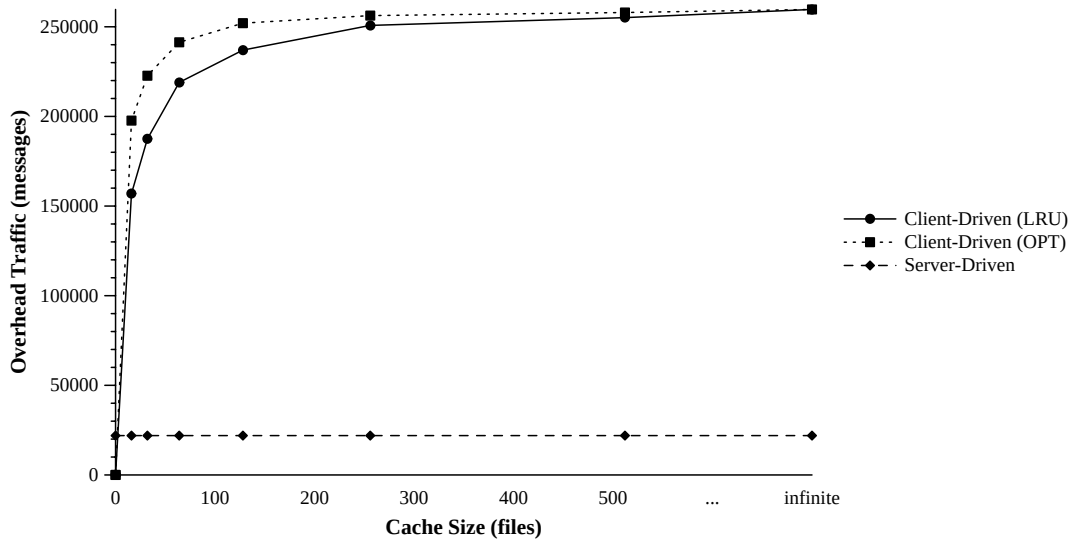

Figure 14: Validation Overhead Traffic, Princeton Trace

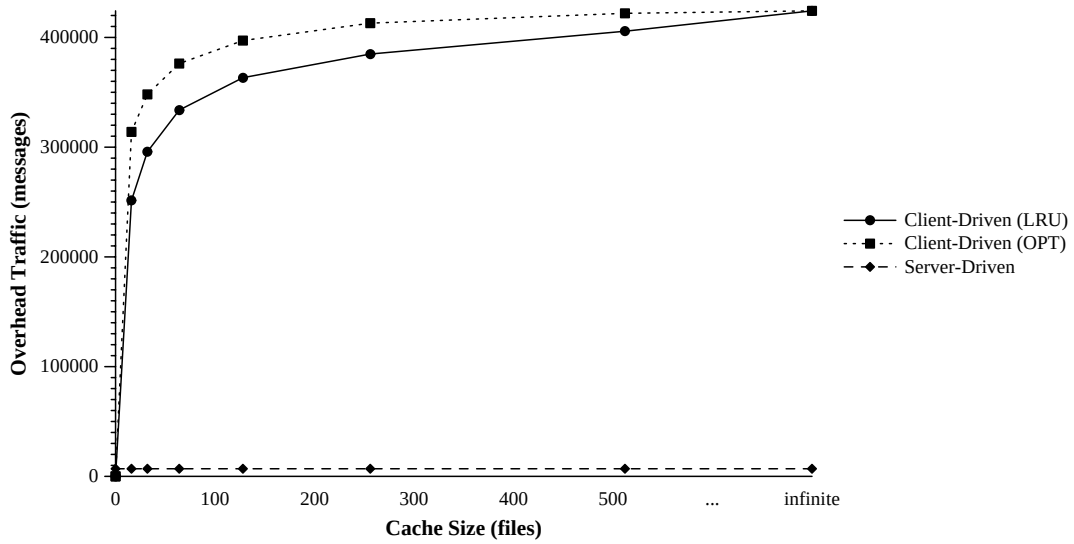

Figure 15: Validation Overhead Traffic, DEC-SRC Trace

the client caches; the scale of the system is limited by the server's ability to handle the client traffic. When the server becomes a bottleneck, either the server's hardware must be improved or its load reduced to a more manageable level. Technological and economic limitations often preclude the former approach, and file server hardware is generally among the fastest available. In general, load reduction is the only practical approach toward larger scale. In a flat system, the server load can be reduced by increasing the size of the client caches, by improving the replacement rule used by the caches, and by employing low-overhead cache validation protocols.

Even with large client caches, inexpensive protocols, and good replacement rules, the load on the server is high in flat systems. Figures 16 and 17 show the results of a simulation of total server load in a typical configuration using the Princeton and DEC-SRC trace data. Simulations of LRU replacement with client- and server-invalidation are shown, as well as a zero-overhead optimal offline scheme for reference. In these simulations, it was assumed that a packet can hold a maximum transfer unit of up to 8192 bytes, and that all packets incur the same cost regardless of actual size; invalidation messages occupy one packet each and file transfers occupy $\lceil n/8192 \rceil$ packets for n byte files. These parameters represent those used in a typical Ethernet-based [19] system. Other common packet sizes result in substantially similar graphs, although they are not shown here. The simulations approximate server load with the number of packets processed by the server compared with a completely uncached system in which the server handles all client I/O. (As before, these simulations reflect data collected after one day of cache warm-up).

These simulation results suggest that server- invalidation imposes about half the load of client- invalidation and very close to the performance of the optimal system as cache size $\rightarrow \infty$. That is, a modern scale-oriented system such as AFS with moderate size disk-based client caches will perform almost as well (in terms of server load) as the theoretical optimum.

Unfortunately, the theoretical optimum for these flat systems is not very encouraging for scale. Observe that even in the best case – no validation overhead and infinite client cases – the server handles 8 to 12% of the traffic that it would handle if the clients were doing no caching at all. While this may be a reasonable margin

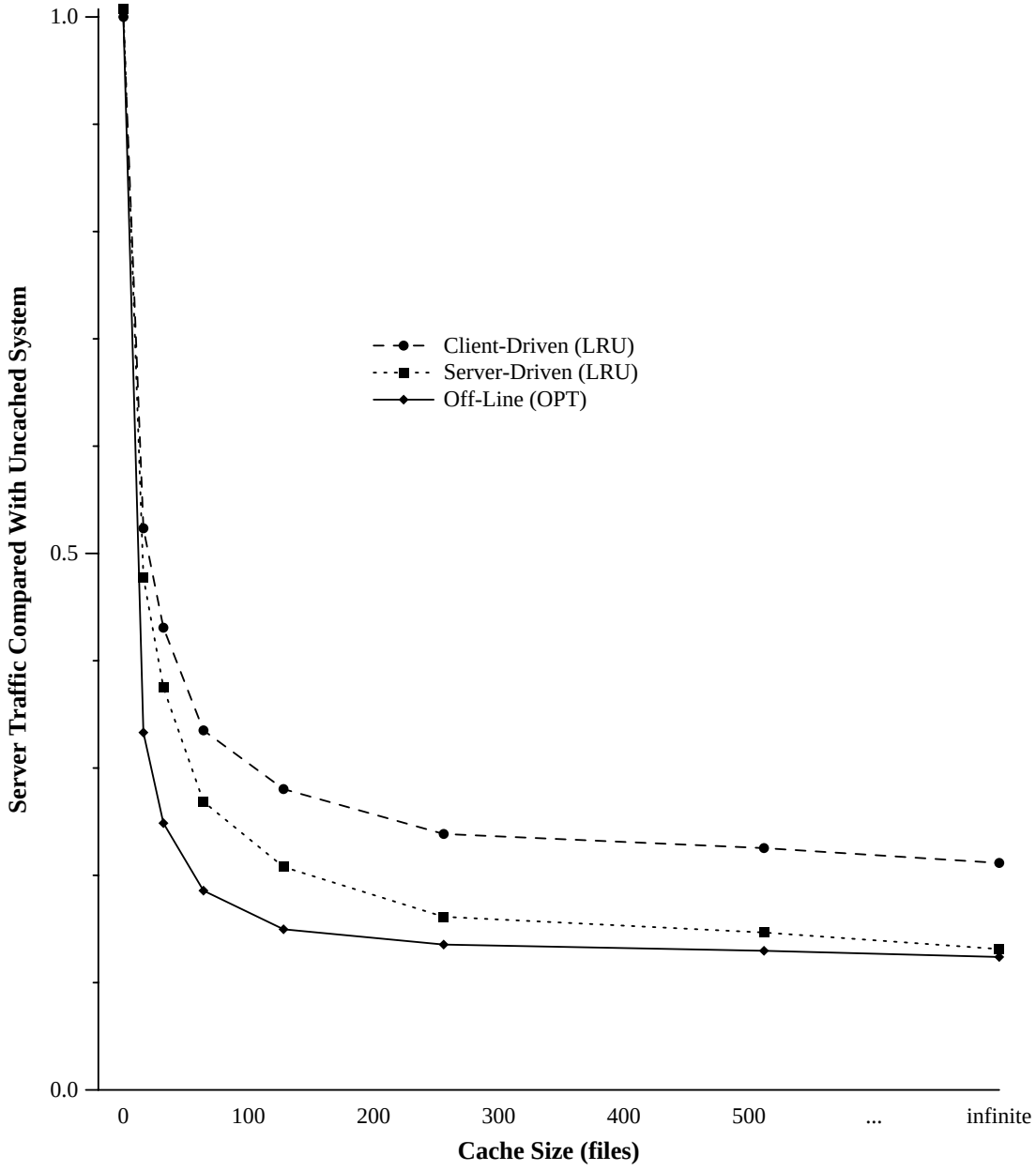

Figure 16: Total Server Load, Princeton Simulations

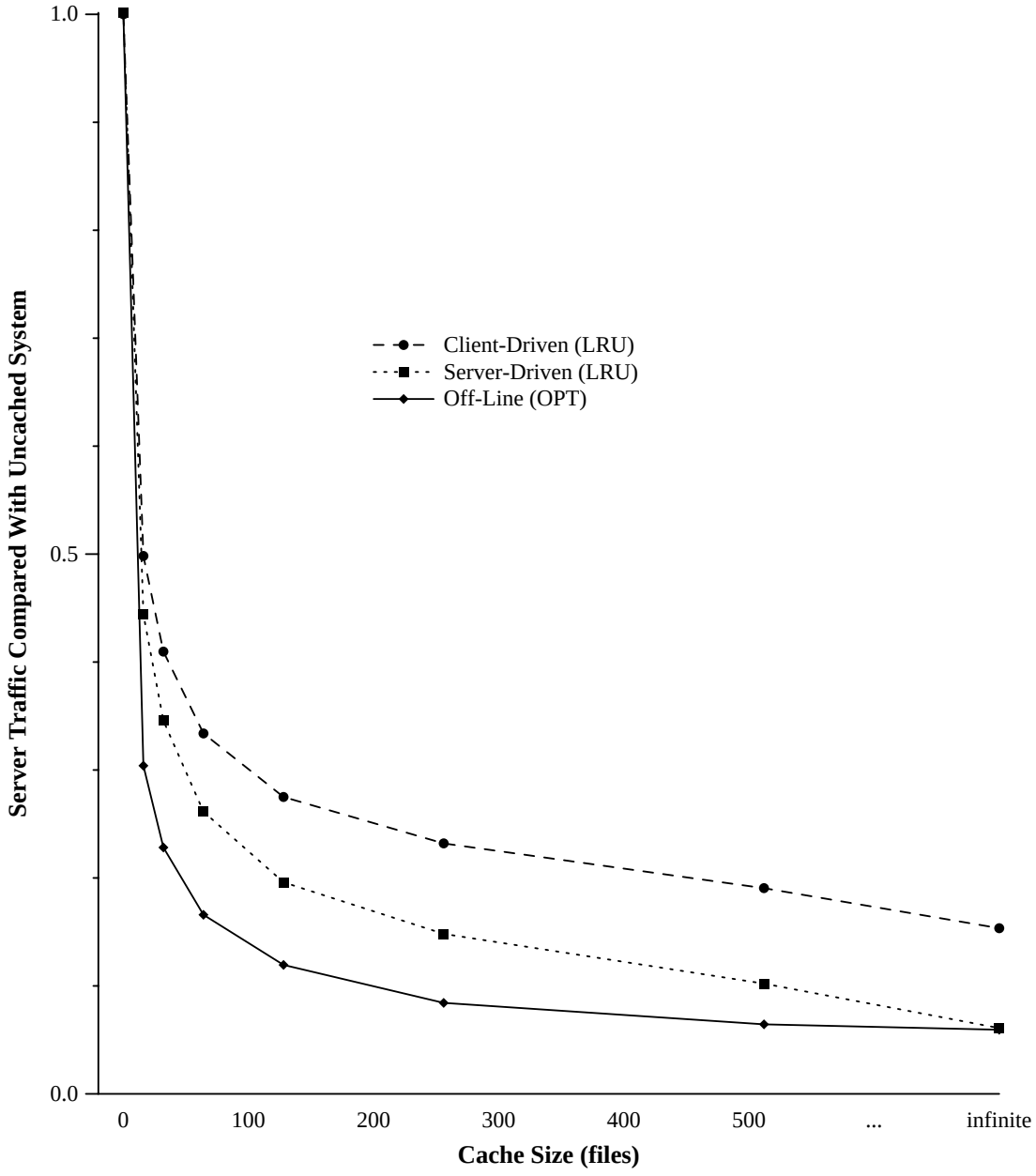

Figure 17: Total Server Load, DEC-SRC Simulations

for a system with several hundred clients, it can become a serious limitation as the number of clients increases.

4.5 Conclusions

Client caches have a large impact on the server load, which is why all practical systems employ client caching to some extent. Caching only a few hundred files and using very straightforward replacement rules results, in trace driven simulation, in cache hit rates that approach the theoretical optimum. As memory becomes less expensive and as local disks become more commonplace on client machines, miss rates of below 90% can be expected, given current workload patterns. Cache validation, with reasonable consistency, can be performed with very low overhead by making the server responsible for notifying clients when their cached files have changed. Several existing commercial DFS architectures make use of these strategies.

Yet the limits of the flat client-server model appear to render truly large scale impossible. In the next chapter, we explore alternatives to flat systems that allow client miss traffic to be handled by sources other than the file server. These architectures have the potential to reduce server load by a large factor, and may provide a framework for building much larger systems.

Chapter 5

Hierarchical Caching

Very large scale in distributed file systems requires that enough client activity be hidden from the server that it does not become a bottleneck. Flat client-server systems, although a convenient abstraction for the user, fail to scale beyond the point where the server can handle the total cache miss traffic of all the clients.

In the previous chapter, we saw that (under at least some workloads) even moderate size client caches and very simple replacement rules approach the theoretical optimum cache miss rate. At first glance, this would seem to imply that there very little room for additional scale; that is simply not possible to construct large-scale distributed file systems that have global miss traffic beyond what a single server can handle. Even when the the server can handle the total client miss traffic, there is the matter of client cache consistency, which places a further load on the server under a large number of clients. (This additional load takes the form of either overhead message traffic for proportional to the client cache hit rate or server state proportional to the number of clients). Clearly, the server becomes a bottleneck as the number of clients scales up. If server data are available from more than one source, however, this single-server bottleneck might be broken. This chapter explores DFS architectures that maintain the client-server abstraction but do not route all client miss traffic to the central server.

5.1 Hierarchical Server Organizations

5.1.1 Hierarchical Replication

A simple way to divide the work of the server is to replicate the server files among several *secondary* servers, with each client communicating only with a single secondary server and updates propagating from the primary to the secondaries. If each secondary server has adequate capacity for its clients, and the primary server has adequate capacity for its secondaries, such an organization will do well. This model could be extended to a multi-level hierarchy of secondary servers in an obvious manner.

In fact, it could be argued that this is precisely the model under which large scale software distribution takes place today, with the replication of the files handled outside the file system by distributing magnetic tapes or manually transferring the files over a network.

In a simple hierarchy of servers it is necessarily to replicate a complete and consistent copy of the server data on each secondary server. Obviously, this entails considerable overhead of disk space, since multiple copies are maintained even of files that are rarely or never shared. Consistency becomes difficult to maintain as the number of secondary servers increases. For example, when a file changes, the new version must propagate to each secondary copy; this can make writing prohibitively expensive in a large network. Full replication at secondary servers is attractive primarily for those files of known wide interest that are known in advance to change rarely, such as large read-only databases and software distributions. Under full hierarchical replication, changing a file is potentially very expensive.

5.1.2 Multi-Level Caching – Static Cache Hierarchies

When more conventional file system semantics are desired for files shared by a potentially large number of clients, a hierarchical model based on caching rather than replication may be a reasonable alternative. Under caching, a replica is made only when a local client has actually read the file, and the the copy does not have an infinite lifetime. Update propagation is simpler, involving simple validation messages

rather than the transfer of entire files. The Deceit File System [28] allows groups of local clients to be organized into *cells* in which remote files are operated on using the ordinary NFS protocols from a local cell server, which provides caching and consistency services. If the file is not in the local cell cache, it is fetched from the remote server.

Although a hierarchical scheme would seem to solve the problem of limiting server state while also limiting server traffic, there is still an important tradeoff. If a client low in the hierarchy accesses a file not in the local cache, it must wait as the read is propagated up to a cache that has the data. Only after each cache between it and the server is searched does the request actually reach the server. While a hierarchy has the potential to lower server traffic considerably for widely shared files likely to be in many caches below the server in the hierarchy, it also has the potential to introduce considerable delay for access to files with a lower degree of sharing.

The simplest form of cache hierarchy can be modeled by having all client requests go through an intermediary which maintains its own cache. If a file is already in the intermediate cache, the server never sees the request; when a write occurs, it is sent directly to the server, which sends an invalidation message to the intermediate server. The intermediate server propagates the message down to the its clients.

The benefits of an intermediate cache that handles all file requests are surprisingly small, according to one experiment. Muntz and Honeyman [21] used the DEC-SRC trace data to simulate a two level cache hierarchy. All 112 clients interacted with an intermediate cache server instead of the actual server. The intermediate server acted as a "proxy" for the clients, maintaining an infinite-size cache of files previously read and written. While this did cause a reduction in server traffic, from the client's perspective the intermediate cache was rarely used, even when the client cache was small. Depending on the size of the client caches, the hit rate at the intermediate cache was between 7% and 70%, falling off very rapidly (to about 10%-20%) for even small client cache sizes. So although there was a modest benefit to the server, the intermediate cache actually introduced a substantial delay for the clients, since most requests not satisfied by the client's own cache were not in the intermediate cache either, and had to be fetched from the file server anyway. We found similar results

when restricting ourselves to the open and unlink operations from the same data. (A more encouraging result is obtained by eliminating home directory accesses from the intermediate cache, yielding an intermediate server hit rate of above 45%). Based on these results, a multi-layer hierarchy of finite caches used for all client file accesses could be expected to yield an even smaller reduction in server load at an even higher cost to the clients.

It is also worth considering the asymptotic behavior of a single intermediate cache, however. Observe that if the intermediate cache is infinite and actually handles all client operations, the only time it will miss is when a file is read for the first time by some client. Once all the files have been accessed at least once by any client, the intermediate cache will always hit (until the file is invalidated). So as time goes to ∞, the intermediate cache miss rate goes to 0. In fact, [21] observed the intermediate hit rate go up continuously throughout the simulation; clearly, the reason it did not reach a much higher rate is due only to the limited time span of the trace data. If the intermediate cache had been finite, however, and if there had been a multi-layer hierarchy of intermediate servers, we would see the asymptotic intermediate hit rate more quickly. However, the experiment does demonstrate an important pitfall of hierarchical caching.

A multilevel cache hierarchy illustrates a tradeoff between client response time and server load. To minimize server load, each client should contact the server only as a last resort, after first querying other client caches for the file. This extra step increases response time for that client, potentially by a large amount. (It also puts a heavy load on the network, assuming inexpensive broadcasting is not available). To minimize response time, each client is better off being served by either its own cache or directly by the server. This places a heavy burden on the server, however, which indirectly increases overall client response time. It is something of a "free-rider paradox" where the local optimization strategy can degrade global performance. The solution, then, is to find a strategy that causes a significant reduction in server load with only a marginal direct increase in client response time.

5.2 Dynamic Hierarchical Caching

5.2.1 Sharing in Cache Miss Traffic

Although the general use of multi-level caching can lead to inefficiencies as described above, Unix file access patterns suggest that there may still be considerable advantage to be gained from some kind of cache hierarchy.

A measure of the potential for a multi-level caching scheme is the proportion of client cache misses that are for files that exist in another client's cache. This can be seen in a simulation of a simple flat caching scheme with the DEC-SRC and Princeton traces. Each client maintained a fixed sized (in terms of the number of files) cache, using server-driven invalidation and an LRU replacement policy. Each client cached any file it read or wrote. (Caching written as well as read files is an important optimization, cutting the miss rate in half). Access to non-shared (or not widely shared) hierarchies, such as /tmp and user's home directories, were considered to be on a local disk and not counted, though such files did occupy space in the client cache. With warm client caches (simulated by eliminating the first trace-day from the statistics), about 60% (in the DEC-SRC trace) or 80% (in the Princeton trace) of client cache misses were of files that existed in another cache. Surprisingly, this was fairly insensitive to the actual size of the client cache (we simulated client caches that held between 16 and ∞ files). Figure 18 shows the percentage of these "redundant" cache misses by cache size in the two traces. Note that even when home directories were included, the shared miss traffic remained above 30%.

This simulation, while crude, is encouraging. It suggests that if clients can share cached files without excessive communication cost to locate the data, traffic to the shared-file server can be reduced by a factor of two to five.

5.2.2 Dynamic Client Hierarchies

The problem, of course, is that although there may be a high probability that a file exists in another cache, the client must determine where it is. Clearly, broadcasting

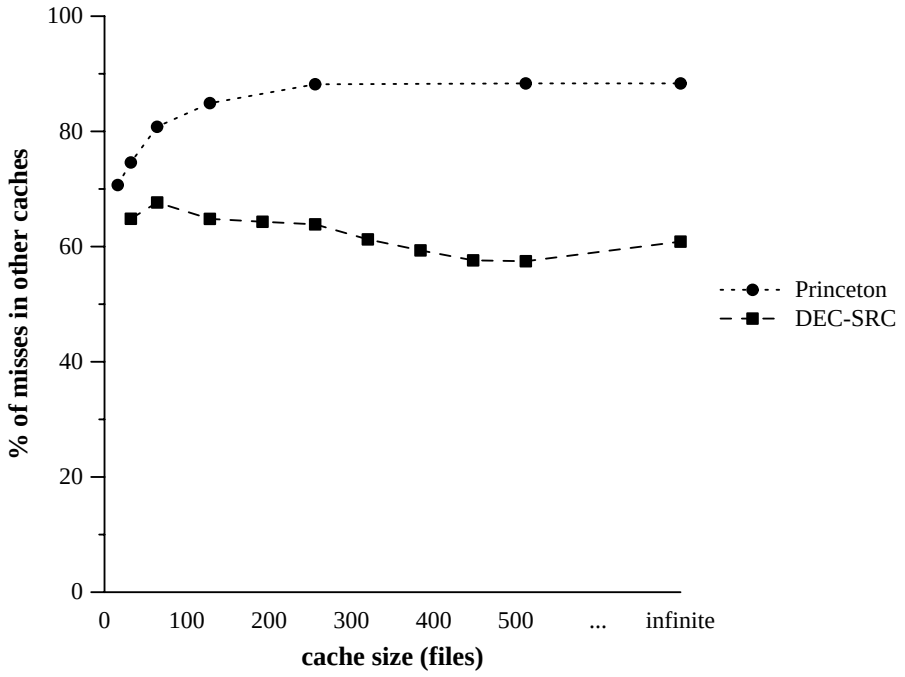

Figure 18: Percentage of Cache Misses of Files Already in Another Client Cache

a request to all other clients does not scale, especially when the clients are geographically distant from one another. Organizing the clients into a static hierarchy, with clients "nearest" the server either relaying requests to the server or serving them out of their own caches, will suffer the same (or worse) performance problems as the infinite-intermediate server discussed in the previous section, and the question of how to best organize such a hierarchy is not at all obvious.

An ideal solution would be one that allows those files used by only a few clients to be cached in the conventional way, but with more widely used files going through a hierarchy. A dynamic hierarchy, different for each file depending on its access patterns, would solve the problem of identifying *a priori* the widely shared files and the best place for the intermediate caches.

The server can assist the clients in locating likely sources of relevant cache data. Consider a simple scheme in which the server serves the early readers of a file in the conventional way, but "sheds" its load to the other clients when a file starts to become more "popular". Each client maintains a conventional fixed-size LRU *file cache* of files

it reads or writes. In addition, it maintains a separate *name cache* indexed by file name. The name cache contains a record for each file read giving the source of the file (the *parent,* which may be another client or the actual file server) and a list of up to Δ *children* that must be notified if the file is changed. The name cache must be large enough to accommodate at least the files in the file cache. The server also maintains a record of children with copies of each file that it must notify should the file change. When client reads a file, it looks for the file in the file cache; if the file is present, it is read in the usual manner (we assume whole-file caching). If the file is not in the file cache, the name cache is checked; if there is a record for the file in the name cache, the file is requested from the parent. If no record is found in the name cache, the file is requested from the server. The reply consists of either the file requested or a list of clients that already have cached copies. In the latter case, one is selected from the list (either randomly or according to some criteria such as proximity) and the request is repeated to that machine, which also will respond with either the file or a list of its children. The procedure is repeated until the file is finally received, at which time the client creates a name cache entry for the file with the address of the parent, if none existed already. The process is guaranteed to eventually terminate when the "bottom" of the tree is reached with a client serving fewer than Δ other clients. The algorithm for fetching a file (without error correction) is formalized in Figure 19.

Each client (as well as the file server) must run a *server daemon* to process requests from other clients. When a client (or server) receives a request for a file, it first checks whether the requester is already in the child list for that file. If not, and the number of children already on the list is equal to the pre-determined parameter Δ, the list of children is sent to the requester instead of the file contents. Otherwise, the file is sent and the requester is added to the child list for that file. If the requester was already on the child list or was just added to it, the file just sent to it and no further action is taken. Note that when a file needs to be sent from one client to another, it is possible that the file is not in the "parent's" cache anymore, in which case it is fetched from the parent's parent first, using the same algorithm described in the previous paragraph. The server daemon algorithm is formalized in Figure 20.

```
if InFileCache(FileName)
  return CacheContents(FileName)
if InNameCache(FileName, server) then
  GetFile(FileName, server, FileBuf)
  AddToFileCache(FileName,FileBuf)
  return FileBuf
endif
server := file_server_address
top:
reply := ask_for_file(FileName, server,
                      SvrList, FileBuf)
if reply = REDIRECT_MSG then
  server := SelectServer(SvrList)
  goto top
endif
AddToFileCache(FileName,FileBuf)
AddToNameCache(FileName,server)
return FileBuf
```

Figure 19: Client File Fetch Algorithm

```
top:
  GetMessage(Client, FileName)
  if AlreadyAClient(Client, FileName) then
    GetCopy(FileName)
    SendFile(FileName, Client)
  else if ClientsServed(FileName) <= DELTA then
    AddToClientList(FileName, Client)
    GetCopy(FileName)
    SendFile(FileName, Client)
  else SendClientList(FileName, Client)
  goto top
```

note: GetCopy reads a copy of the file into the local cache if a client, and is a NoOp on the server.

Figure 20: Server and Client Daemon Process

Writes are always done directly through the server, which sends invalidation messages to each client on its child list, who in turn send invalidation messages to each of their children for the file. Note that since the list of children is kept in the name cache, invalidation messages must be sent out to each child whenever a name cache entry is replaced. (It is possible to use a different write policy which may be more desirable for performance purposes, but this thesis only considers the simple server-based write scheme).

This scheme forms a hierarchy for each file which can be viewed as a tree of maximum degree Δ. The first Δ readers of a file communicate directly with the server, in the conventional way. Other clients wanting to read more widely shared files (those with more than Δ readers) communicate with the earlier readers (or their children). Thus, the server need only keep track of (and serve requests from) the first Δ readers, plus the additional traffic caused by new clients who want to read the file for the first time and must be given the list of existing children.

5.2.3 Trace-Driven Simulation

This simple dynamic hierarchy was simulated using the DEC-SRC and Princeton traces. A single server was assumed for the shared hierarchies; each client used another server (not simulated) for more local files such as `/tmp` and home directories, although these files did occupy client cache space. We simulated a logically connected network in which all clients can communicate at equal cost, and clients used a uniformly distributed random function for selection of a parent when re-directed by the server. The performance of the system is influenced by two major parameters: the client file cache size n and the maximum degree Δ. We measured values of n from 64 files to infinite, and values of Δ from 2 to ∞. (Since there were at most 250 machines in the trace data, Δ values of 250 or more are for all practical purposes infinite and equivalent to a conventional, flat scheme.) A third parameter, the client name cache size, turned out to be very small in practice, and sizes above about 100k bytes per client were for all practical purposes infinite in the simulation. We measured a number of parameters, including the number of file transfers handled by the server and overall in the network, and the number of overhead messages (invalidation messages

and messages redirecting clients to children) sent by the server and in the overall network. We would expect decreasing Δ to decrease the traffic at the server but raise it in the overall network, since some client requests must be routed through more than one downstream client to be satisfied. Server traffic is a measure of server cost, while overall network traffic can be viewed as a measure of client access time and processing cost. The question is whether the increased client access cost is justified by the decrease in server traffic cost.

The simulation results suggest that dynamic hierarchies may work well in practice. For all cache sizes, decreasing the value of Δ (increasing the depth of the tree) greatly reduced the transfers at the server, always by at least a factor of two for $\Delta = 2$. Although the overall number of transfers did increase, the increase was always smaller than the corresponding decrease in server transfers and was under 25% for $n \geq 128$ files. Figures 21 and 22 show the number of file transfers at the server (solid curves) and in the network (dotted curves) for various values of n and Δ after one day of cache warm-up (the figures for cold start are similar but slightly higher). The axies on theses graphs have infinite Δ at the left; infinite Δ is equivalent to a non-hierarchical, flat scheme.

Decreasing Δ is not without cost to the server, however, since although the number of invalidation messages it must transmit in the event a file is overwritten decreases, it must send lists of children to clients it does not wish to service. The number of these overhead messages is not dependent on client cache size (since the transmitter of the message does not know whether the file is still in the receiver's cache and so must always send the message), but is dependent on Δ. Tables 4 and 5 show the number of overhead messages handled by the server and in the network as a whole for various values of Δ.

Based on these simulations, it would appear that the impact of the dynamic hierarchy depends on the relative cost of a file transfer compared with an invalidation or redirection message. This is influenced strongly by the underlying network topology and protocols, the maximum basic transfer unit, the sizes of the files transferred, and so on. Files transferred in our simulation averaged around 11k bytes. The non-communication costs (such as reading a file from disk as opposed to checking an

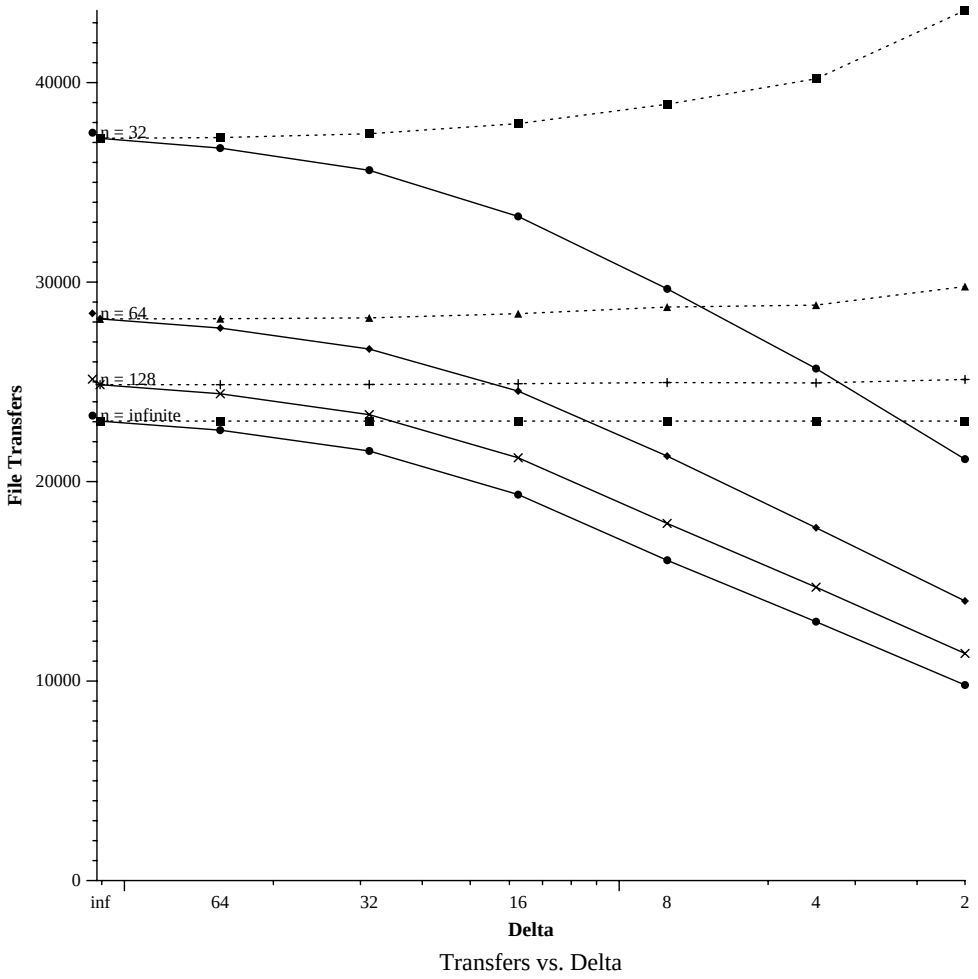

Figure 21: File Transfers in Dynamic Hierarchy Simulation, Princeton Trace

	Overhead Traffic	
Δ	Server	Total
2	23381	42196
4	13405	27251
8	12236	21442
16	13228	17802
32	13933	15616
64	14092	14574
∞	14119	14119

Table 4: Dynamic Hierarchy Overhead Messages, Princeton Simulation

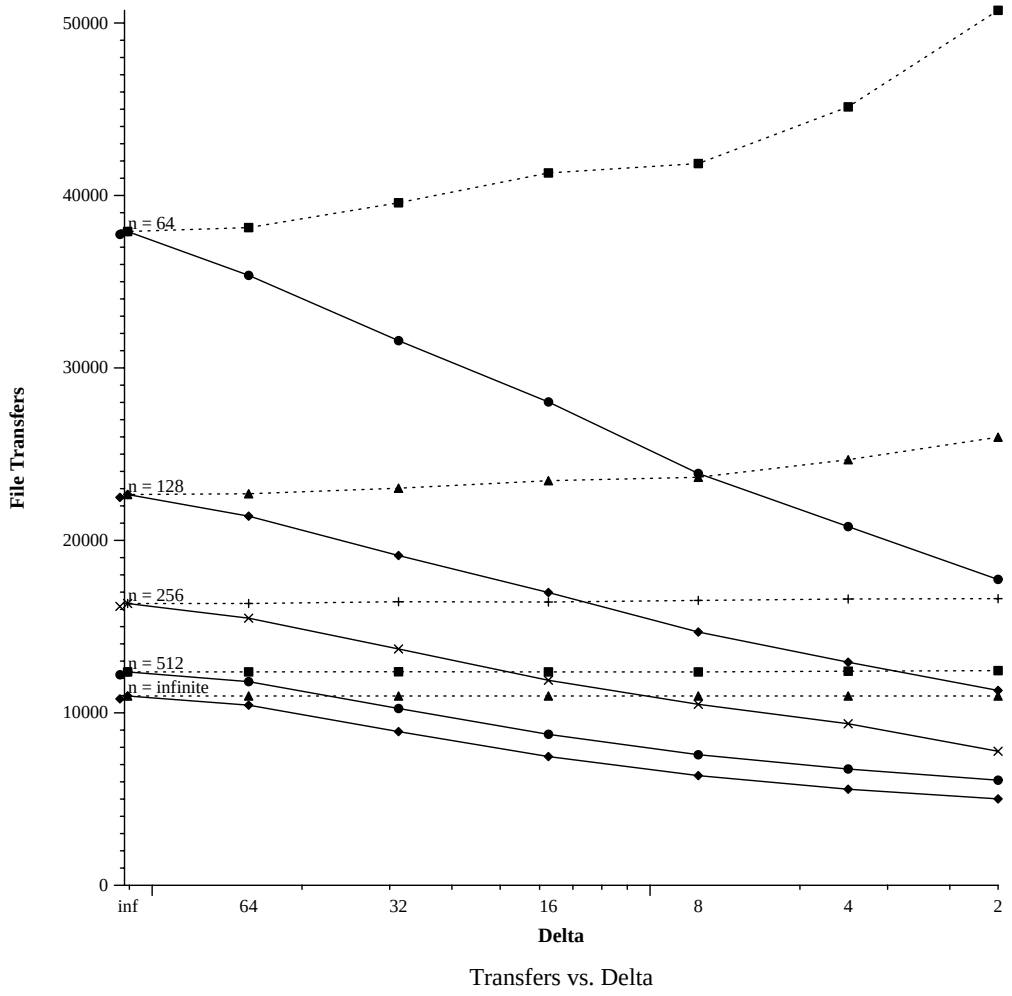

Figure 22: File Transfers in Dynamic Hierarchy Simulation, DEC-SRC Trace

	Overhead Traffic	
Δ	Server	Total
2	11939	17907
4	3638	9045
8	758	5375
16	410	3922
32	410	2476
64	410	942
∞	410	410

Table 5: Dynamic Hierarchy Overhead Messages, DEC-SRC Simulation

entry in an in-core table) come into play as well, although we do not consider them directly here. It is probably reasonable to assume that the cost of transferring a file greatly exceeds the cost of the other messages; if this were not true client-driven invalidation, which is used by NFS, would carry no benefit. If the cost of an overhead message is even as high as 10% of the cost of a file transfer, the overhead traffic cost is negligible in our simulation

5.2.4 Failure Models and Consistency

The simulation described in the previous section assumed that all hosts are well connected, reliable, and remain available to all other hosts at all times. Obviously, such assumptions are not likely to be valid in actual systems, particularly the large-scale systems for which such a scheme is designed. We identify two problems that must be addressed: non-connectivity in network topology, and network or server failure.

Connectivity Failure

The first problem is simply that, depending on the underlying network, not every host can always communicate with every other host, even if one host (the file server) can communicate with all of them. (In internet style networks, this can be caused by routing failures, and other networks, such as those with secure gateways, do not always guarantee symmetric connectivity). This is an issue since requests for files may be re-directed to other clients. If a client cannot communicate with any of the hosts a server directed it to, it must re-contact the server. At this point, the server can either replace one of the clients on the child list with the new client, or it can simply increase the number of clients it serves. If it chooses the former option, it must invalidate the old child first, which may trigger additional traffic when that client (or one of its children) reads the file again. If it chooses the latter option, it increases the size of its Δ. A third option is to maintain two Δ values, a soft one and a hard one. New clients are still re-directed when the value exceeds the soft Δ, but will be added to the child list if they fail to contact any of the existing children. New

clients can be added under these circumstances until the hard Δ is exceeded, at which point old children are invalidated and replaced. The cost of doing this depends on the connectivity of the network, of course. In the simulations above, only a logically fully-connected network was modeled.

Consistency

Consistency is the problem of guaranteeing that the clients' views of the system can differ by at most some criteria. In discussing file system cache consistency semantics, it is worthwhile to distinguish between the consistency provided under normal operation and the worst case under failure.

Under normal operation, consistency is determined entirely by the granularity of the cache validation messages; for example, client caches may be guaranteed to be up to date only at file open time or for the individual read and write operations.

A server or network failure can lead to inconsistency if a client does not receive an invalidation message for an over-written cached file. Although the problem of consistency in partitioned distributed systems is known to be a difficult one, a client can query the server from time to time ("keepalive" messages) to ensure that it is still alive and no messages are lost. Clearly, if such messages are too frequent, the benefits of caching are minimal; if the messages are not frequent enough, inconsistency can result.

An in-depth study of cache consistency issues is beyond the scope of this thesis; this section aims only to informally discuss the consistency issues that are unique to a dynamic hierarchical environment and to suggest directions for more formal work in this area.

To determine the minimum frequency of keepalive messages, one must identify the kind of consistency required. At one end of the spectrum, there is what has come to be known as single-machine "Unix semantics"; if a file changes, all future reads reflect the change immediately. Clearly, to maintain strict Unix semantics in any cached distributed file system requires client-driven invalidation, since there is no other way to guarantee that no server invalidate messages were lost and that the cache is current if the connection to the server is lost. Strictly speaking, in fact,

these client validation messages must be issued for each individual I/O operation, and not just, say, at open time, making true Unix semantics very expensive. In practice, most systems relax their consistency requirements in favor of performance; NFS, for example, checks only at file open time. Interestingly, under server-driven invalidation, the consistency under normal operation can be tighter (a cache entry may be invalidated as soon as the callback message is received at write-time), but degrades under failure, since a callback message may be lost if the network fails.

A high degree of consistency under failure can be maintained without resorting to client-driven invalidation, however. We identify two kinds of consistency semantics that a client may wish to maintain. The first, *time-bounded Unix semantics* maintains *temporal* constraints on inconsistency, and the second, *serializability*, maintains a *logical* constraint on inconsistency.

Time-bounded Unix semantics are simple to maintain under server-driven invalidation; clients check with the server whenever they have not otherwise communicated within some real-time bound, guaranteeing that cache reads are never more than this bound out of date. If it cannot reach the server, the read fails.

Another metric of consistency may be defined which similar to the database concept of one-copy serializability (1-SR). This requires simply that the global sequence of reads and writes be equivalent to some serial schedule on a single machine file system. Observe that if there is only one shared file server providing invalidation messages (and all writes always go synchronously through the server), this degree of consistency is automatic, since if the link between server an client is broken, the data in the client's cache is logically consistent as of the time it last communicated with the server. All reads from the cache, even if the data is *temporally* out of date. may be *logically* considered to have occurred just before the link with the server failed. If the client attempts to write data based on out of date reads from its cache, either the write will fail (because the link is down), or, if the link has returned, any queued invalidation messages can be logically considered to have occurred just *after* the write.

If there is more than one file server, or the clients have a cache hierarchy, it is still fairly inexpensive to maintain 1-SR. Observe that it is sufficient to ensure that once data written by a client is read by a remote machine, future reads by the writer

must be current as of just after the write. That is, only after writing data that may be read by other clients does a client need to be certain that it has not missed any invalidation messages. In a cache hierarchy, this may be accomplished for almost no overhead under normal operation by having the server maintain a list of failed callback attempts.

In a hierarchy, a client can loose an invalidation message if any machine from which it received a file has failed or if any machine from which those machines received the file have failed. Observe that the list of machines that could deny an invalidation message to the client is the list of machines from which it received "redirection" messages for each file currently in its cache. Once a client writes externally readable data, it must verify that each machine in the "invalidation path" for each file in the cache has not failed to propagate an invalidation message. Assuming that writes block until complete, a simple way to do this is for each client to notify the server when it cannot propagate an invalidation message to another client. The server maintains a list of these "dead" clients which it sends to future writers on their next write to any file. After writing and receiving the dead client list, the client removes any files currently in the cache dependent on any client on the list. A global clock (such as that described in [15]) is helpful for serializing the times of the failures among the clients. Note that with this scheme there is no overhead in the case where no failures have occurred, and overhead is proportional to the number of failures times the number of writers.

5.3 Conclusions

This section described a number of strategies for reducing bottlenecks by allowing clients to receive cache miss services from more than one machine. Static strategies, such as a simple hierarchy of clients or a hierarchy of secondary servers, reduce the server's load but at a high latency cost to the clients for unshared files. A dynamic hierarchy, in which only those files which are actually widely shared are propagated through the hierarchy, appears to avoid these latency penalties. Based on trace-driven simulations, a dynamic hierarchy can reduce the load on the server by a factor of three

to five, with very low added client latency.

It is tempting to infer that this means that a dynamic hierarchy allows a file server to serve three times as many clients as it would otherwise be able to, or that file servers for dynamic hierarchies need have only a third the processing power as those serving flat caching schemes. Obviously, the applicability of these results to scalable systems is highly dependent on the workload; these traces studied just one week of activity on less than 250 clients. It would be desirable to conduct experiments on workloads of larger systems, over longer time periods, and in computing environments outside research laboratories. However, intuition suggests that as traces cover longer periods and more clients, the proportion of cache misses of files present in other caches will increase, making the effects of dynamic hierarchies even more pronounced.

It is important to view any simulation based on traces of existing systems with some caution when applying them to large-scale systems, since no truly large-scale systems exist to measure yet. It is, however, probably reasonable to expect that that the overall trend of widely shared files becoming read-only to apply to an even greater extent to large scale systems. Such systems will likely be used to support file distribution of infrequently changing files, such as those in `/bin` directories. Obviously, the applications of the future will have a strong impact on the file access patterns, and the best we can do is make educated guesses based on current workloads.

To make dynamic hierarchies practical, several other issues must be addressed. Security is obviously an important consideration, since clients will often require some assurance that the cached copy they are reading is a valid replica of the "official" copy. It is possible that digital signatures provide a solution to this problem, but this thesis does not address this issue. It may also be desirable to include some sort of load-balancing facility, such that clients can move themselves down in the hierarchy if they cannot themselves afford to serve other clients. Again, this may be an interesting direction for future work.

The following chapter discusses a simple prototype dynamic hierarchical file system, and provides real time data to augment the simulation studies described in this chapter.

Chapter 6

Prototype Implementation

In the previous chapter we saw, through trace-driven simulation, that dynamic hierarchies have the potential to substantially reduce file server traffic and thereby improve client performance in large-scale distributed file systems. Trace-driven simulation alone, however, is not sufficient to show that dynamic hierarchies would be practical in a real system. Simulations, while useful for making rough predictions about performance, do not always tell the entire story. There is no guarantee that the cost models used in a simulation accurately reflect the costs incurred in practice, and the models used in the simulations of the previous chapter were particularly primitive in that they count only messages and file transfers. Even when very complex simulation cost models are used that count costs in all aspects of system performance, there is no guarantee that an actual implementation could be constructed that follows the same model. Furthermore, simulation results tell us nothing about the "software engineering" complexity of constructing a real implementation; it is possible that a system might be so complex to preclude building a practical or even prototype implementation, even if the algorithms themselves are easy to simulate. Finally, simulation tells us nothing about the aspects of the system not considered in the simulation; for example, the semantics of the system may be unacceptable to real users, regardless of the performance.

This chapter describes a prototype implementation of a dynamic hierarchical file system. The aim of the implementation is as a "proof of concept" of the algorithms

described in the previous chapter and to provide additional, real time, performance data to augment our simulation results. In Section 6.4, we are able to compare the simulation predictions directly with the observed server performance.

6.1 Implementation Design Goals and Principles

A prototype differs from a production system in several ways, particularly with regard to performance and reliability. Prototype implementations need not be concerned with carefully tuned performance or platform-dependent optimization issues that are not directly related to the concepts the prototype is being built to test. Reliability is often perhaps the greatest area of divergence between prototype and production; while a production system must generally be carefully designed and tested to avoid failure, data loss, or incorrect results under unusual conditions, in a prototype implementation it is often sufficient to simply be aware of the conditions under which the prototype does not perform properly.

The motivation for the prototype file system arises out of our desire to gain simple real time data and provide a proof-of-concept rather than to build a file system for day-to-day use by real users. Our goals were therefore simple:

- Support of basic file system operations, including the creation and deletion of files, reading and writing file data, and basic directory and file attribute operations. More sophisticated file system functions, such as links (hard and symbolic), access control and authentication, device access, special files, atomic locking and rename operations, and so forth, are of only secondary interest and are not required in the initial prototype. (Most of these secondary features were not, in fact, included in the implementation).

- Simple interface. The interface to the file system should be simple but should model, to some extent, the interface to a "real" file system. Ideally, programs should require no modifications to use the prototype, and the underlying operating system should require little or no change.

- Realistic performance. In this case, realistic means that the performance should be good enough that real users are willing to use it (for at least some applications) and that the response time and throughput are at least within small constant factors of what could be expected from a carefully tuned production system.

- Reliability under normal operation. The prototype system should not be subject to sudden failure or data loss under normal operation. Normal operation is defined as when all nodes and the network are up and when all clients are using their interfaces legally. Some failure or data loss, however, is tolerable when a node fails or if a client abuses its interface in some way.

In designing the prototype, we tried to be guided by principles that would facilitate realization of these goals, even at the expense of performance. In particular, we attempt to:

- Use existing software where possible. This can be restated as "use existing interfaces where possible". Note that there is some performance penalty here, since existing interfaces may not provide the most efficient way to do something.

- Use existing protocols. In particular, we favor protocols for which existing simple interfaces exist that provide reliable data transfer (such as TCP/IP) over those that require application level detection of lost data (such as UDP-based RPC implementations).

- Use specialized high level tools, languages, and protocols. For example, we favor the use of such protocols as NFS or ftp to transfer file data over a more efficient, general purpose protocol that has a more complex interface.

- Avoid the kernel. Real file systems are generally implemented inside the Unix kernel for performance reasons. Unfortunately this adds to the complexity of implementation and makes performance profiling much harder. Kernel based implementations also require that the system support a complete Unix vnode interface, which may entail a more sophisticated prototype than desired.

These "minimalist" principles are appropriate to prototype design for two reasons. First, of course, they require less engineering effort to produce the prototype than would be required by a performance-oriented approach. Second, and perhaps more importantly, the prototype is more likely to be correct if its components consist of well understood parts put together in a straightforward way.

6.2 Prototype File System Architecture

This section describes the design of PFS, the Prototype dynamic hierarchical File System. (PFS may also be taken to stand for "Parasitic File System", since clients "latch on" to the caches of other clients rather like a parasite takes root in a host organism.)

Recall that in a dynamic hierarchy, a client may obtain a particular file from either the file server or another client's cache. In fact, the only difference between the services provided by clients and those provided by the server is that the server always has a local copy of every file, while other clients may need to first obtain a fresh copy of a file before completing some requests. Therefore, the server structure is almost a proper subset of that of the client.

Each client maintains a cache of files as well as a name cache containing sources of files and invalidation obligations. The procedure for obtaining a file is similar to the algorithms described in the previous chapter: first check the data cache, then the name cache for the source of the file if it has been previously read, otherwise try to obtain the file from the server. The server may either supply the file, or, if Δ clients have previously cached the file, it sends the list of previous clients instead, in which case the procedure is repeated with one of the clients on the list. Each client runs a server which processes these requests from other clients. All writes go directly to the server, which propagates invalidation messages to the clients on its name list, who propagate the messages to their clients, and so on.

Whole file caching is used in the prototype implementation. As soon as any bytes of a file are read, the entire file is fetched. A real implementation would probably use a large chunk of the file rather than the whole file, so that very large random access

files do not cause pathological behavior.

No special provisions are made for recovery; the prototype assumes that clients do not go down in the middle of operations and actually do propagate their invalidation messages when they say they do.

All messages and requests are sent via remote procedure calls implemented on top of TCP/IP.

6.2.1 Server Architecture and Implementation

The server must simply process requests from clients and propagate invalidation messages when a file is overwritten. It assumes that all writes occur "in band"; writes not done through the server do not cause invalidate messages to be sent.

All files are stored under the standard Unix file system. The server also maintains (in a standard file) a *client database*, which is directory of clients with cached copies for each file.

The basic structure of the server is shown in Figure 23. The *file manager* provides the interface to the file system (and client database). It sends invalidation messages to clients when a file is overwritten or deleted and manages concurrency issues. The *server daemon* listens for requests from clients on the network interface and either sends the file data or the list of previous clients from the client database. The server daemon blocks until a request is complete, and therefore can process only one client request at a time. (In principle, multiple copies of the server daemon can be run to yield better concurrency, but this is not completely supported by the implementation of the file manager in the prototype implementation.) A configuration file is used to determine which file system to provide service for, the value of Δ, the size and location of the client database, and which clients to serve. No sophisticated access control or authentication is included in the prototype.

The prototype server was implemented under 4.3 BSD (AOS) Unix on the IBM-RT/PC. It consists of just over 1500 lines of C code, and runs entirely at user level.

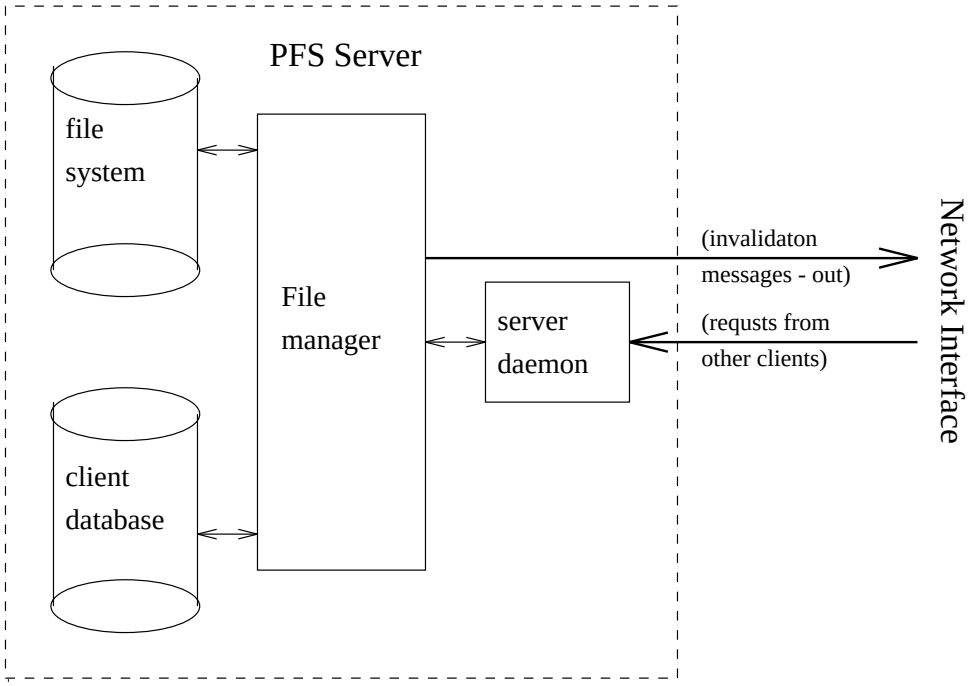

Figure 23: Prototype File System Server Architecture

6.2.2 Client Architecture and Implementation

The client's job is functionally similar to that of the server, but with additional requirements and constraints. The basic structure of the client includes most of the server's design, although several components have been changed slightly and a number of additional components are included.

The basic client design is shown in Figure 24. The *server daemon* is identical to that of the server. The client also must be accept invalidation messages when cached copies are out of date; this is handled by the *validation daemon*, which listens for and accepts these messages in the form of RPCs from the network interface. The interface to the actual user programs is via the *NFS server*. The NFS server implements a subset of the NFS protocol [22] and waits for messages on the *localhost* interface. The client kernel uses this interface by issuing a standard NFS *mount* command on its localhost interface. Client programs can then use the server just as they would any other remote NFS file system (although several features are lacking, including hard and symbolic links).

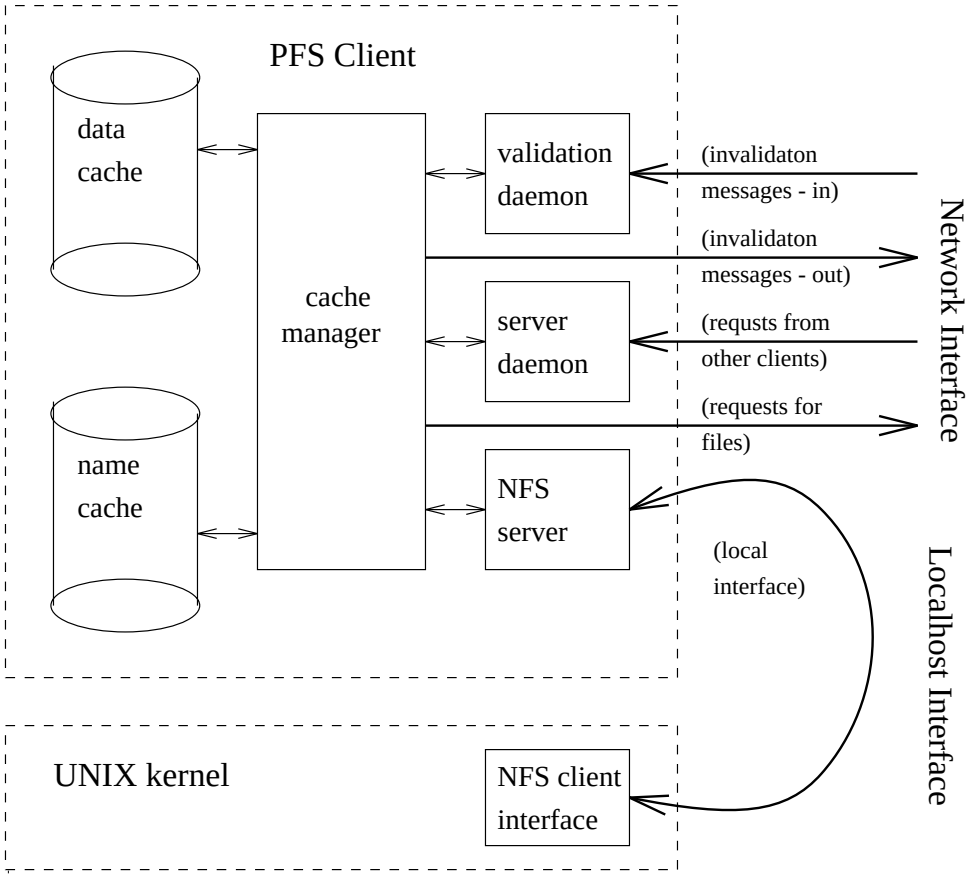

Figure 24: Prototype File System Client Architecture

Like the server, copies of all files and client data are stored under the standard Unix file system. Unlike the server, however, these are maintained as fixed size caches; an entry is not guaranteed to be present when required, and adding an entry may first entail the deletion of an existing one. The *cache manager* is analogous to the server's file manager, with the additional capacity to manage a cache instead of a file system and with several interfaces added. When a request for a file comes in (either from the server daemon or the NFS server), the cache manager first looks for the file in the data cache; if it is not present there, it checks for a previous source for the file in the *name cache*. If an entry for the file is present in the name cache, requests it from the host specified there, otherwise the file is fetched from the server, subject to redirection as described in the previous chapter. When the copy of the file finally arrives, the cache manager makes a copy in the data cache, after first selecting an entry for replacement if needed. When a file is invalidated (either through a write from the NFS server or a message from the validation daemon), the file is deleted from the data cache if it is still present and any clients for that file in the name cache are notified.

No provision is made for the possibility that a message cannot be delivered due to network failure; if a message times out, it is simply lost, leading to a possible inconsistency.

Like the server, the client is implemented under 4.3 BSD (AOS) on the IBM-RT. It is more complex than the server code although still small, consisting of about 4500 lines of C (including the NFS server, which is almost half of the total code). Again, configuration files are used to set the size of the caches (in files), the value of Δ, etc. The client is configured to use the server via an entry in `/etc/fstab` pointing to the localhost interface and the use of the standard NFS `mount` command. Clients cannot also be used as standard NFS servers because of conflicting use of port numbers.

6.3 Basic Implementation Performance

The performance of the system proved to be adequate for the purposes of the prototype although probably not sufficient for day to day use. We measured the basic

Operation	Time (ms)		
	Client (CPU)	Server (CPU)	Response (Real)
Overhead Msg	14.4	3.1	21.5
0K File Xfer	16.8	8.1	49.8
1K File Xfer	18.9	10.0	52.2
14K File Xfer	56.8	37.3	153.1
40K File Xfer	153.2	101.6	366.1

Table 6: Prototype Performance

performance of the system in three ways: the CPU time (user+system) used at the client side, the CPU time used at the server side, and the real response time of an operation from start to finish on an otherwise unloaded client, server, and network. Several operations were measured, including overhead (invalidation and redirection) messages and various size file transfers. These statistics were taken with client and server running on IBM-RT model 135 workstations. All measurements represent the average over 1000 operations (in a loop), and all code was first put in real memory to avoid paging times. The measurements are given in Table 6. All times are in milliseconds.

Observe that the real time latency of an overhead message is about half that of a 0K file transfer, and about one eighth that of a 14K transfer. 14K file transfers are shown because that is approximately the median size of files transferred in the Princeton and DEC-SRC traces; 1K and 40K are the 80th and 20th percentiles of file sizes, respectively. Note that these times are for the PFS code only; they do not include latency or CPU time introduced by the client side NFS code, and do not include context switches into and out of of other software. Note also that the times given include actual disk I/O; the buffer caches were cleared before running the tests, and no file names were re-used.

6.4 Trace-Driven Workload Analysis

While the numbers given in the previous section are helpful for evaluating the absolute quality of the implementation, they do not themselves provide a basis for comparison with other file system techniques. In the previous chapter, we saw simulation results based on the number of files and messages transferred using various caching techniques with user trace data. In this section, we compare the real-time performance under these traces of the prototype implementation with a Δ of 2 and with a Δ of ∞. That is, this tells us how well dynamic hierarchies do compared with an equivalent-quality implementation of a conventional server-invalidate file system.

We used the Princeton and DEC-SRC traces to generate workloads for the prototype. Seven IBM-RTs were used as client machines and one was used as the server. Since the DEC-SRC trace has 112 clients and the Princeton trace had over 250, clients randomly shared physical machines, although each client ran a private copy of the client software and all client–client communication went through the network interface. To keep the clients synchronized with respect to the original traces, "sync" points were introduced whenever a client read must occur after another client's operation. Clients were otherwise free to issue their next operation as soon as the previous one completed; this kept server utilization near 100%, and allowed the one week traces to complete in a little less than one day. (This high server utilization is a bit artificial, but makes the results relevant to large-scale systems, where server utilization is expected to be high).

6.4.1 Server Load

The total CPU time used by the server during the trace driven runs allows a direct comparison between the implementation and the simulation results. The simulation told us how many file transfers and overhead messages would be performed for a given workload, but did not directly tell us what the server load would be. The best we can do with the simulation results is multiply the number of server messages by the measured server CPU time for a message plus the number of server file transfers by the median file size server transfer time. This figure is only an approximation

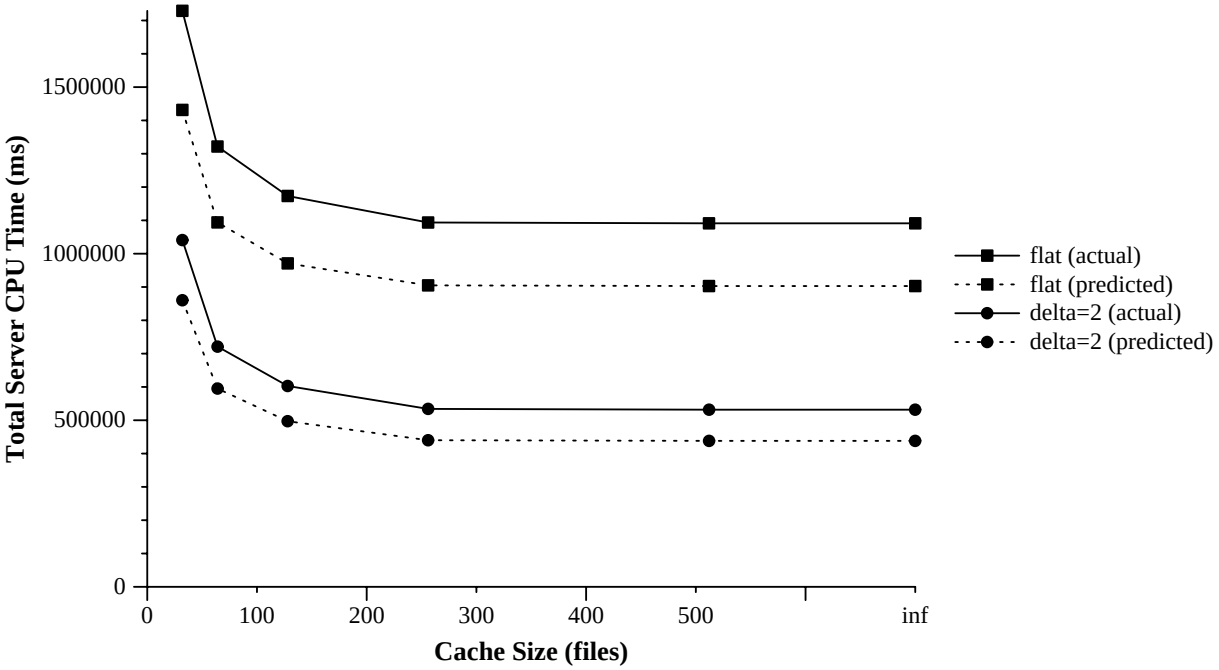

Figure 25: Total Server Run Time – Princeton Workload

of actual server load, however. Fluctuations in the sizes of files actually transferred as well as delays introduced by the environment (paging and context switches, for example) could result in differences between the predicted and observed server loads. In fact, the simulations were slightly optimistic in predicting server load, by about 20%, although this was roughly constant for all data points. Figures 25 and 26 give the actual and and predicted server load for the Princeton and DEC-SRC workloads, respectively. The solid curves indicate actual load and the dotted curves indicate predicted load calculated by the simple multiplication above. Runs were made with $\Delta = \infty$ and $\Delta = 2$ for various cache sizes. Times are given in milliseconds.

6.4.2 Client Load

We measured the total client response time for each of the two workloads; that is, the time from the start of the run to the last client completing its last operation. Time required for any application processing of the files was not included; clients simply throw away the data once read and move on to the next operation. Two sets of runs

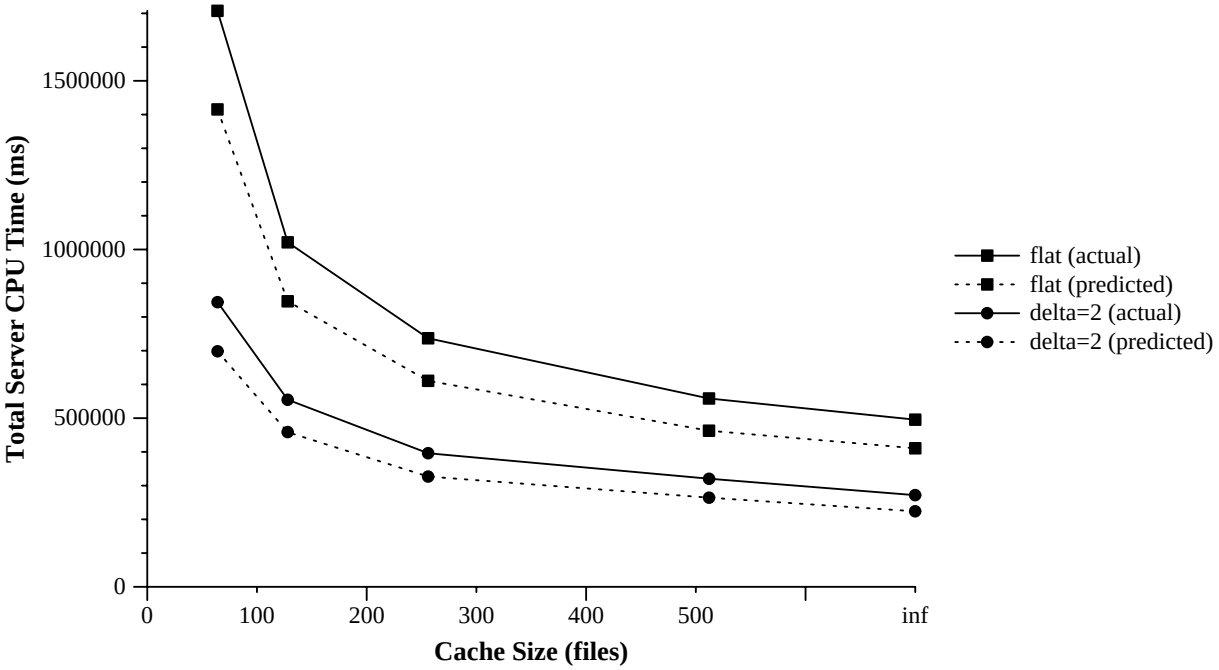

Figure 26: Total Server Run Time – DEC-SRC Workload

were made for each workload, one with $\Delta = 2$ (full dynamic hierarchy) and another with a $\Delta = \infty$ (conventional flat system). Runs were made with various client cache sizes from 32 files to ∞ files.

Figure 27 shows the overall client response time for the Princeton workload; Figure 28 shows the response time for the DEC-SRC workload. All times are given in milliseconds.

Observe that in both workloads and for various cache sizes, the $\Delta = 2$ run had an overall time of about half that of the conventional flat run. This is consistent with the simulation results of the same workloads in the previous chapter. (Other, intermediate, values of Δ were tested as well, but not for all data points, so they are not presented as graphs here. The results are consistent with the simulations in the previous chapter, and are never better than $\Delta = 2$).

These workload-driven runs help answer the most important question left unanswered by the simulations: how well do dynamic hierarchies do in practice? The simulation tells us only how many files and messages are transferred, which is only

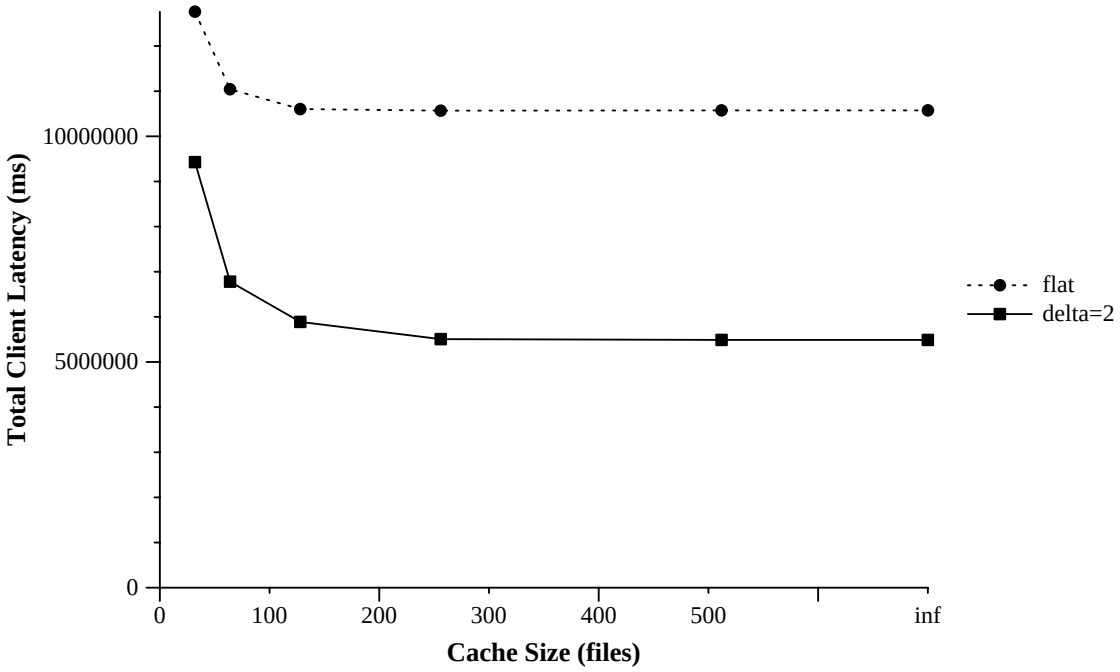

Figure 27: Overall Client Run Time – Princeton Workload

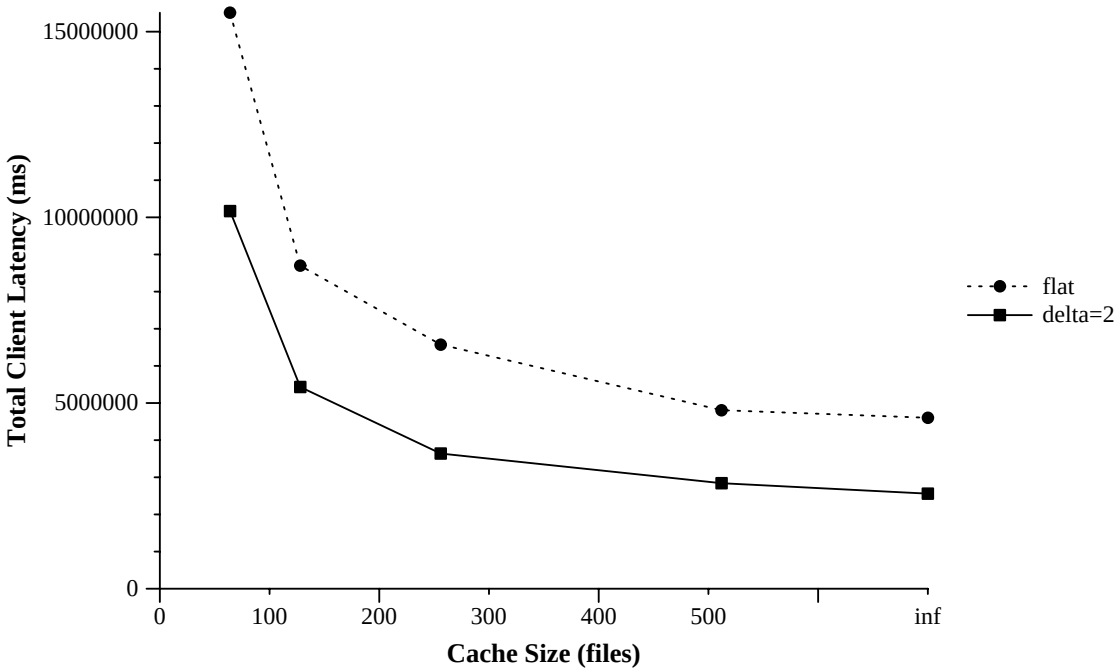

Figure 28: Overall Client Run Time – DEC-SRC Workload

indirectly relevant to real question of interest – actual client performance. The simulation did not tell us, for example, the impact of the slightly increased client load (caused by serving other clients) or whether the model of counting messages is relevant to client response time. From these runs, we see that the shift of two thirds of the server file transfer traffic to other clients (from the simulations) translated into a factor of two reduction in overall client response time.

The reader is cautioned not to conclude that the use of dynamic hierarchies will always result in a dramatic improvement in client performance. In particular, observe that the entire reason for the reduction in client latency is related to the reduction in the load of a server with near 100% utilization. If server utilization is not high, dynamic hierarchies will not improve performance at all, since work would be simply shifted from a lightly-loaded server to other clients.

6.5 Conclusions

This chapter described a simple prototype of dynamic hierarchical file system. Although probably unsuitable for actual production use, it is a functional working file system with reasonable, if sub-optimal, performance. The main purpose of the prototype is to help validate the simulation results of the previous chapter.

Workload-driven runs of the prototype were encouraging. The real-time results, compared with a conventional flat-caching file system, showed about a factor of two improvement, which is approximately what was predicted by simulations. This is important for two reasons. First, it tell us that, indeed, dynamic hierarchies have practical potential for reducing client response time in large, high utilization file systems. Secondly, it tells us that very simple simulation models (counting file transfers and messages) appear to be useful for predicting the performance of actual file systems.

In the next, final chapter, we discuss future directions for building practical very large-scale distributed file systems using a dynamic hierarchical approach.

Chapter 7

Conclusions

The preceding chapters have examined one aspect of the problem of building a massive-scale distributed file system. In systems built upon the client-server model, a natural service bottleneck occurs when the client workload exceeds the server's capacity. Client caching is a well-known technique for improving client performance, reducing server load, and increasing the maximum number of clients that can be served without bottlenecks. Only when a client cannot find a file in the cache is the server involved, and so the server load can be reduced substantially. Previous studies suggest that sufficiently large client caches can reduce server load by as much as 80%–90%. But server load remains proportional to client workload in such systems, and so, as the system scales up, the server will eventually become a bottleneck.

In this thesis we have identified a number of file access properties in Unix-based systems that suggest that server load can be reduced beyond what is possible in a conventional client–server implementation. In simulations based on workloads taken at two different sites, we found that a large proportion of client cache miss traffic is for files that have already been read by other clients and are still active in those other clients' caches. By exploiting this property, clients may be able to substantially reduce the load they place on the server, potentially enabling larger-scale systems to be constructed. In other words, clients may be able to reduce server load by sharing their caches. The chief problem is for clients to determine when to contact the server and when and where to go to another client for service. One approach is to organize

clients into a hierarchy, with the server at the root.

In Chapter 5, we saw that static hierarchies (and, equivalently, intermediate caches) do reduce server load but introduce substantial client latency when the hierarchy is searched for files that are not yet there. To reduce this latency, we propose a dynamic hierarchical scheme that creates a client hierarchy for each file based on usage patterns. Simulations suggest that this can reduce server load by a factor of two to three, with greatly reduced client latency compared with a static scheme. In Chapter 6, we describe an implementation of a file system based on dynamic hierarchies. Under workloads that generate high server utilization, this implementation showed an improvement in overall client response time of a factor of about two.

Perhaps the most important unresolved issue is how well dynamic hierarchies would perform under truly massive-scale workloads. These workloads, of course, do not yet exist to measure, so we can only speculate as to the impact of dynamic hierarchical approaches on such systems. It is possible, however, that it may actually be the case that the reduction in server load actually increases as the number of clients and client activity scales up. Observe that the number of "alternative" machines from which a client may receive service is the depth of the dynamic tree, which is approximately $\log_\Delta$ of the number of past readers. It is reasonable to conjecture that as this number goes up, so, too, does the probability that the server will not be reached, and so the percentage of client activity seen by the server could be inversely proportional to the log of the number of clients. This is, of course, only conjecture, and only experience with real massive scale workloads will tell.

Scale in distributed file systems is a complex problem, with unresolved issues in many areas including naming, security, semantics, consistency, network management, and system performance. All these areas pose problems in the design of truly large scale systems, although the latter, which was our focus, is perhaps the most quantitatively measurable. This does not diminish the importance of these other problems, however, and there remains considerable work to be done before a massive scale system could be practical. In that light, the most important contribution of this thesis is to identify an approach that should be incorporated into such systems.

Bibliography

[1] Aho, A.V., Denning, P.J, and Ullman, J.D. "Principles of Optimal Page Replacement." *J. ACM* 18:1 (January), 1971.

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

[3] Barak, A. and Kornatzky, Y. *Design Priniples of Operating Systems for Large Scale Multicomputers.* IBM Research, 1987.

[4] Blaze, M. "NFS Tracing by Passive Network Monitoring." *Proc. USENIX Winter Tech. Conf.,* 1992.

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

[6] Blaze, M., and Alonso, R. "Dynamic Hierarchical Caching for Large-Scale Distributed File Systems." *Proc. 13th Intl. Conf. on Distributed Computing Systems,* 1992.

[7] Bodnarchuck, R.R. and Bunt, R.B. "A Synthetic Workload model for a Distributed System File Server." *Proc. SIGMETRICS Conf,* May, 1991.

[8] Cate, V. and Gross, T. "Combining the Concepts of Compression and Caching for a Two-Level Filesystem." *Proc. Conf. on Architectural Support for Programming Languages and Operating Systems'*, April, 1991.

[9] Curry, D. "Nfswatch(1) Manual Page." *Unpublished Manual Page,* 1991.

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

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

[12] Gray, C. and Cheriton, D. "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency." *Proc, 12th ACM Symp. Op. Sys. Principles,* December, 1989.

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

[14] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanaryanan, M., Sidebotham, R.N., and West, M.J. "Scale and Performance in Distributed File Systems." *ACM Trans. Computing Systems,* Vol. 6, No. 1, (February), 1988.

[15] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System." *CACM,* July, 1978.

[16] Levy, E. and Silberschatz, A. "'Distributed File Systems: Concepts and Examples." *ACM Computing Surveys,* December 1990.

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

[18] Ousterhout, J. et al. "The Sprite Network Operating System." *IEEE Computer,* February 1988.

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

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

[21] Muntz, D. and Honeyman, P. "Multi-Level Caching in Distributed File Systems" *Proc. USENIX Winter Conference,* 1992.

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

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

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

[25] Robinson, J.T. and Devarakonda, M.V. "Data Cache Management Using Frequency- Based Replacement." *Proc. ACM SIGMETRICS Conference,* May 1990.

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

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

[28] Siegel, A., Birman, K., and Marzullo, K. "Deceit: A Flexible Distributed File System." *TR 89-1042,* Dept. Comp. Sci., Cornell University, Nov. 1989.

[29] Smith, A.J., "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms." *IEEE Transactions on Software Engineering,* July, 1981.

[30] Sreenivasan, K. and Kleinman, A.J. "On the Construction of a Representative Synthetic Workload." *CACM,* March 1974.

[31] Staelin, C. "File Access Patterns" *CS-TR-179-88,* Dept. Comp. Sci, Princeton U., 1988.

[32] Srinivasan, V. and Mogul, J.C. "Spritely NFS: Experiments with Cache-Consistency Protocols." *Proc. 12th Symp. on Operating Systems Principles,* 1989.

[33] Thacker, C., et al. "Alto: A Personal Computer." In *Computer Structures: Readings and Examples (2/e),* McGraw-Hill, 1981.

[34] Wilmont, R. "File Usage Patterns from SMF Data: Highly Skewed Usage." *Proc. Intl. Conf. on Management and Performance Evaluation of Computer Systems,* December 1989.

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