ABSTRACT

MULTILEVEL CACHING
IN
DISTRIBUTED FILE SYSTEMS

by
Daniel Alan Muntz

Chair: Peter Honeyman, Toby Teorey

This thesis explores ways in which intermediate cache servers affect the performance and scalability of distributed file systems. Caching plays an important role in improving the performance of distributed file systems. Using distributed file system traces and cache simulations, we explore extensions and modifications to the traditional client caching model employed in file systems such as AFS and Sprite.

We find that an *intermediate cache server*—a machine logically interposed between clients and servers that provides cache service to the clients—may typically reduce read requests to primary file servers by 40% to 60%. This decreases file server load, thus increasing system scalability.

We are considering a mass storage based file system, where the primary file system resides on a bank of tapes. An intermediate cache server is an essential component in this tape-based mass storage file system, necessary for overcoming the access latency of the mass storage system. Employing a delayed-write caching policy at the intermediate cache server in this system allows the intermediate server to satisfy typically 70% to 80% of file system requests that are not satisfied by client caches. This fact, combined with high client cache hit rates (typically 80%) lend to the practicality of our system.

We also investigate modifications to the AFS cache model that improve file system performance over increasingly prevalent low-speed networks. We find that an intermediate cache server on the client side of the low-speed link can increase

client performance by mediating low-speed link traffic. We find that an intermediate cache server employing a delayed-write caching policy can typically delay 50% to 74% of write requests until a daily sync period. Performing these writes during inactive periods improves performance for interactive traffic over the low-speed link as well as for other low-speed link traffic.

These three topics are related to one another by virtue of the aspects of the architecture that they share. We use four file system traces to perform detailed trace-driven simulations to evaluate intermediate cache servers in these three roles. The same file system traces and simulations can be used to provide results applicable to each.

MULTILEVEL CACHING
IN
DISTRIBUTED FILE SYSTEMS

by

Daniel Alan Muntz

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1994

Doctoral Committee:

Adjunct Associate Professor Peter Honeyman, Co-Chair
Professor Toby J. Teorey, Co-Chair
Adjunct Assistant Professor Charles J. Antonelli
Assistant Professor Elke A. Rundensteiner
Professor Daniel Teichroew

Dedicated

to my parents, Betty and Ronald Muntz

and in memory of L.B.R.

ACKNOWLEDGEMENTS

I am deeply grateful to Professor Peter Honeyman, who has been generous in his guidance and support throughout the course of my studies, and would like to express my sincere gratitude to Professors Charles Antonelli and Toby Teorey for their advice and support regarding my research. I would also like to thank my dissertation committee members, Professors Elke Rundensteiner and Daniel Teichroew, for their time and effort in reading both my proposal and dissertation and in providing valuable suggestions.

I also owe a debt of gratitude to Ted Hanss and the Center for Information Technology Integration (CITI) for providing me with a wonderful research environment and excellent computing facilities. I consider myself truly fortunate to have enjoyed surroundings so conducive to research.

Thanks also go to my co-workers at CITI: Larry Huston, Mike Stolarchuk, Jim Rees and Jerry Winters for providing valuable ideas and support; Bill Hampton, Elaine Lauerman, Becky Priebe and the other (ex)systemgroupers for their indispensable aid; Kati Bauer, Chris Tanis, and Terri Saarinen for their help on many occasions.

I would like to thank IBM for providing financial support for much of this research, and the University of Michigan for a Rackham Dissertation Grant.

I thank the xFS crowd, M. Dahlin, C. Mather, R. Wang, T. Anderson and D. Pat for supplying us with their file system traces and the following people at DEC-SRC who contributed to other traces made available to us: Susan Owicki, B. Kumar, Jim Gettys, Deborah Hwang, and Andy Hisgen.

I thank Wendy Motooka for her inspirational thesis work. And last and most, I thank my loving wife Harumi for helping me throughout this entire process.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

CHAPTER 1

INTRODUCTION

1.1 Overview

As distributed file systems grow, so does the need to increase scalability–greatly increased size must not mean greatly decreased performance. At the Center for Information Technology Integration (CITI), we are studying tools and techniques for offering file service to a huge client base, perhaps as many as 30,000 end systems.

The typical user at the University of Michigan is currently provided with only about 10 megabytes (MB) of online storage. With the increasing presence of multimedia technology, users will need the ability to store and access files many megabytes in size. Thus distributed file systems must be able to provide transparent and timely access to large quantities of storage.

The use of distributed file systems over low-speed links, *e.g.,* dialup lines, is also increasing. Although recent advances in modem technology have made speeds as high as 28.8kbps (kilobits per second) common, this is still quite slow compared to standard Ethernet (10 megabits per second). Ideally, users would be able to use the distributed file system to perform the same tasks independent of their location. While this may not be possible (depending on what tasks the user wishes to perform), increasing performance in the low-speed network context moves us toward this goal.

To address these issues we are investigating scale- and performance-enhancing extensions to the traditional client caching model currently employed in file systems such as AFS [12] [13] and Sprite [20]. Specifically, we are evaluating various caching strategies and topologies through simulations using distributed file system traces. We are exploring three topics, united by their use of cache hierarchies and the methods we use to examine them:

Increased Scalability: Intermediate cache servers reduce load on primary servers and increase the number of clients they can serve.

Increased Storage: We describe an integrated mass storage file system, an essential part of which is a hierarchical cache structure to allow clients fast access to slow mass storage.

Increased low-speed network performance: We evaluate modifications to the AFS caching model to improve client performance by making better use of the limited bandwidth of low-speed networks.

Intermediate cache servers are machines that are logically positioned between clients and servers. They act as servers with respect to clients and act as clients with respect to the file servers. Intermediate servers reduce request load experienced by the primary file servers by satisfying requests from client machines.

Intermediate cache servers provide a generic method for increasing the scalability of distributed file systems. In an environment where distributed file system clients cannot perform their own caching [9], an intermediate cache server greatly reduces server load; however, intermediate cache servers can also be effective even when clients perform local caching. Simulations show that an intermediate server in an AFS environment may typically cut read traffic to the primary servers in half. Write traffic is unaffected because AFS employs a write-through model.

An intermediate cache can also act as a "floating" cache for individual clients that exceed their local caches temporarily, *e.g.,* a client with a 20 MB cache could use an intermediate server to perform some task better suited to a 100 MB cache. An intermediate server with a large cache, placed in a computing lab containing client machines with smaller caches, may be effective in this manner.

We also examine the effectiveness and optimization of an integrated mass storage system based on a cache hierarchy, similar to that used by Plan 9 [22] from Bell Labs. The system will likely consist of a file system based on a slow mass storage media, *e.g.,* 8mm tape drives, fronted by a several gigabyte disk cache, which is in turn fronted by client caches.

Home and mobile computers connected to the Internet via low-speed links (*e.g.,* 28.8kbps modems) are becoming increasingly prevalent. Running AFS on such machines is appealing in that the user is provided with a consistent view of her files, independent of the client machine being used. However, the write-through caching policy makes AFS use over low-speed links less convenient. A delayed-write policy could greatly increase performance in this scenario.

1.2 Thesis Organization

Cache hierarchies have proven extremely beneficial to increasing the performance of processor memory caches. There have also been studies of the use of cache hierarchies to improve scalability in distributed file systems. These studies and other related work are discussed in Chapter 2.

Chapter 3 describes the data sets used in our simulations. We look at issues of data collection, the composition of the data sets, and their limitations.

In Chapter 4 we look at the network topology used in the simulations and the metrics with which we evaluate the performance of the cache hierarchy. And in Chapter 5 we examine some of the generic issues pertaining to intermediate cache servers.

Chapters 6 and 7 contain simulation results. The intermediate server's effect on scalability is evaluated in Chapter 6. In Chapter 7 we describe the mass storage based file system under consideration. Intermediate cache simulation results, as well as client cache experiments are presented.

Low-speed network issues are discussed in Chapter 8. The thesis ends with Chapter 9, a discussion of our contributions and conclusions.

CHAPTER 2

RELATED WORK

2.1 Scalability

Much of the work done with multilevel caches examines processor memory caches, which show great success in improving CPU performance [11]. Multilevel caches in hardware typically have high hit rates. This, and the large difference in access times to the various levels of the cache hierarchy, explain the success of multilevel caching in processor memories [7]. Multilevel caches in distributed file systems can exhibit both of these properties, suggesting they may be useful for increasing file system performance.

Blaze and Alonso [4] investigate the effectiveness of a dynamic cache hierarchy in a distributed file system environment. They conclude that their system could cut load on primary servers in half. These results, based on traces spanning approximately six days, assume whole-file caching at each level of the hierarchy with least-recently used (LRU) cache replacement. Their system increases scalability by allowing clients to serve files from their caches to other clients. As clients are added to the distributed file system, these same clients become responsible for tasks that would otherwise be handled by file servers. However, allowing clients to perform file service can present security problems. For example, if security is compromised on a client, it may be able to present falsified data to other clients, or write falsified data to the file servers. Blaze suggests that a cryptographic file system can resolve some of these issues, but this presents another set of complications that are beyond the scope of this thesis.

Dahlin, et al. investigate an experimental file system, xFS, in which client machines handle many of the tasks normally performed by file servers [8]. All file system data is stored in client caches and data transfers are performed directly between clients. Through the analysis of trace driven simulations, they conclude that the xFS protocol greatly reduces server load compared to AFS. This system suffers from the same security problems encountered with Blaze's system due to the client-to-client transfers of data. The xFS system incorporates a "no write through" policy (unlike AFS, modified files are not written to primary file servers when they are closed). This is similar to some of the work we do with *delayed write* policy for the intermediate cache server.

Makaroff and Eager consider the performance of caching in systems where caching takes place at both clients and servers [17] *i.e.,* two-level caches. They conclude that client caching degrades server cache performance and, somewhat surprisingly, that the overall miss ratio for both cache levels may increase slightly when client cache size is increased. These results are based on traces and cache sizes (maximum 2 MB) much smaller than those we are using.

2.2 Mass Storage

The Plan 9 system in use at Bell Labs has mass storage integrated into the file system. The file system resides on an optical WORM "jukebox" fronted by a disk cache, in turn fronted by a RAM cache. Avoiding access to the mass storage device in this system is not as important as in the system under consideration at CITI, as access to the WORM device is significantly faster than access to a bank of tapes, due to higher seek times for tape devices.

File migration to and from mass storage devices is also similar to the integrated mass storage system. Miller and Katz evaluate such a file migration system using 24 months of activity traces [18]. Their conclusion on the importance of prefetching may be applicable to the mass storage system we are considering.

Goldick, et al. have addressed the issue of increasing storage available via AFS by producing a system for interfacing AFS with mass storage systems [10]. It uses file migration and replication to optimize use of storage in their storage hierarchy and to make recently requested files available from random access storage devices. Our system does not use file migration, but relies on caching to make files available from faster media.

Antonelli and Honeyman describe the integrated mass storage system under consideration at CITI [1] evaluated as part of this thesis. This system uses the mass store as a first-class data repository, with a disk-based file system (and possibly RAM) serving as a cache of the mass store. All storage other than the mass store is used exclusively for caching. They focus on cache replacement policies appropriate for such an environment.

2.3 Low-speed Networks

Huston, as part of his research into "disconnected AFS," implements a *fetch-only* mode for use when a "disconnected" client is connected to the network [14]. In this mode, the client fetches data not found in the local cache, but defers write requests, behaving as though the client were disconnected from the network. Thus, files are saved in the local cache but are not passed to the primary servers. Fetch-only mode allows the user of a low-speed link to increase performance significantly by choosing when to use the limited bandwidth available to perform the write operations.

Should automated consistency prove valuable in the low-speed network environment, Srinivasan and Mogul's work with Spritely NFS [23] [24] and work with DFS [16] provide a useful analysis of consistency issues in a delayed-write caching system.

CHAPTER 3

DATA ISSUES

3.1 Introduction

To evaluate the effectiveness of the intermediate cache server in its various roles, we perform simulations based on file system traces collected in working distributed file system environments. Alternatively, we could perform simulations based on simulated workloads generated from mathematical models of file system usage. However, we are concerned about the ability of these models to capture (possibly important) features of file system activity, for example, activity varies greatly by time of day, day of the week, etc. Thus, we choose to base our studies on real work loads.

Gathering traces of actual file system usage presents several problems. The trace collection method must not disturb file system usage. For example, if the trace collection involves software running on file servers, the collection software must not consume resources (network bandwidth, server CPU time, etc.) to the point of altering usage patterns. Dealing with the large volume of data collected during tracing can also be problematic. Due to the type of information we require in the traces, as well as the duration of tracing, the traces tend to be quite large—for our traces, tens and hundreds of megabytes. Depending on how the traces are gathered, kernel modifications may be necessary.

Our simulations are based on four sets of traces. Two of these traces are of activity on AFS servers at Michigan. One set consists of requests from several AFS clients at CITI. The final trace is made up of NFS requests from machines at UC Berkeley.

name	CPU	OS	disks
`marge`	IBM ES/9000 Model 720	AIX/370 V1.1	IBM 3380
`homer`	IBM ES/9000 Model 720	AIX/370 V1.1	IBM 3380
`loki`	IBM ES/9000 Model 580	MVS/ESA V4.2	IBM 3380, 3390
`babble`	IBM RT Model 125	AOS V4.3 BSD	IBM 9332 SCSI
`bastion`	IBM RS/6000 Model 320H	AIX V3.1	IBM 400MB SCSI
`beachhead`			
`toehold`			

Table 3.1: **CITI AFS server environment.** AFS server activity was recorded on the seven servers shown. The 6-day data set consists of file system requests handled by all seven servers. The 24-day data set contains only requests to `homer` and `marge`.

3.2 The 6-day Data

AFS clients and servers interact via *remote procedure calls* (RPCs) [3]. For example, when a client wishes to access information in a file in AFS it issues a `FETCHDATA` RPC, requesting a portion of the contents of a specified file from an AFS server. A file system trace can be created by logging information about the RPCs received by AFS file servers.

The AFS servers for the `citi.umich.edu` and `umich.edu` cells were modified to generate trace records for each remote procedure call received [6]. The Rx layer (AFS's RPC layer) saw code added to accomplish the server tracing. The `rxgen` program, equivalent to NFS's `rpcgen`[25], was modified to generate code that logs information just prior to and just after the execution of the server-side of the RPC.

AFS clients exchange file data with the server via the `FETCHDATA` and `STOREDATA` RPCs, whose functions follow directly from their names. These are the only RPCs in which we are interested for these experiments. Each `FETCHDATA` and `STOREDATA` request contains a timestamp, the originating client's network address, the file's `FID` (unique identifier), and the offset and length of the data being requested.

The 6-day data set was collected from all of the AFS servers in the `citi.umich.edu` and `umich.edu` AFS cells over a 6-day period from 10:23:54A.M. on Thursday April 23, 1992 to 12:57:48P.M. on Wednesday April 29, 1992 [1]. Data collection took place on the seven AFS servers described in Table 3.1.

[1]These data, representing the longest contiguous period during which tracing was operating on all servers, are actually a portion of a larger data set.

The 6-day data contain all requests for data chunks from 327 clients. A *chunk* is a segment of an AFS file, usually 64 KB in length. 176,733 different files are referenced in 413,493 trace records. 143,712 (35%) of the 6-day requests are `STOREDATA` requests.

3.3 The 24-day Data

The 24-day data set, which includes a portion of the 6-day data, was collected in the same manner. Collection occurred from 6:34:04A.M. on Tuesday April 14, 1992 to 7:26:14A.M. on Friday May 8, 1992. This data collection took place on the file servers `homer` and `marge` (described in Table 3.1), and consists of requests from 486 [2] clients. 320,725 distinct files are referenced in 1,151,321 trace records. 411,764 (36%) of the 24-day requests are `STOREDATA` requests. During the 6-day trace interval, 75% of the AFS requests were directed to `homer` and `marge`; it is likely that these servers were also predominant in the 24-day trace interval.

When the contents of an AFS file are requested, the client machine's cache manager first checks to see if the request can be satisfied from the client's local cache. If the request can be satisfied locally, no `FETCHDATA` RPC is sent to the server. Since the 6-day and 24-day traces consist exclusively of file system requests received by the AFS servers, requests satisfied in the clients' caches do not appear in the data sets. The implication is that simulation results are accurate only when simulated client cache sizes are greater than the actual client cache sizes. We explain this in detail in Chapter 4. Our clients typically cache 500 to 2,000 data chunks, so results from simulations of client cache sizes greater than 2,000 chunks are accurate.

3.4 The 2-month Data

The 6-day and 24-day data sets left unanswered questions in our early simulations. The server-gathered traces lack client references to AFS files that were satisfied by the clients' local caches. This deficiency precludes simulations of caching algorithms other than that used in the original environment (LRU). Traces spanning

---

[2] Not the processor.

a greater time are also of interest. We therefore decided to perform further data collection.

For this data collection we modified a kernel module for the IBM RS/6000. This module uses a kernel buffer to log all `read` and `write` requests for AFS files as well as `FETCHDATA` and `STOREDATA` requests issued by the client. The `read` and `write` requests reflect individual system calls to perform file I/O, whereas the `FETCHDATA` and `STOREDATA` requests are RPCs used for exchanging data with AFS servers. A user-level program running on each client machine periodically sends the buffered data to a central machine for permanent storage. File system benchmarks show that the modified kernel does not noticeably affect AFS performance.

The 2-month data collection took place on 37 IBM RS/6000 AFS client machines from 3:07:00A.M. on Wednesday October 20, 1993 to 3:07:00A.M. on Monday December 20, 1993. It contains 8,213,455 `read` requests, 3,125,928 `write` requests, 526,785 `FETCHDATA` requests and 326,765 `STOREDATA` requests. 288,599 unique files are referenced in the 2-month data set. The CITI RS/6000 AFS clients have many of the most frequently used binaries on their local disks, such as files typically found in `/bin`; thus, references to these files do not appear in the traces.

3.5 The xFS Data

The xFS data was collected by the creators of the experimental file system, xFS [8], to evaluate their system through simulation. Blaze's `rpcspy` program [5] was used to monitor network activity and generate traces for NFS clients on four Ethernets. `Rpcspy` passively examines packets on the Ethernet, so it provides a particularly unobtrusive method of trace gathering. During data collection, 4% of all network activity was dropped.

Since `close` system calls do not appear in NFS network traffic, heuristics were used by Dahlin et al. (in a postprocessing step) to add them. The `close` calls are necessary to reflect the semantics of AFS, *i.e.,* store on close. The trace records of interest for our simulations are the `Block READ` requests (equivalent to a `read` request in the 2-month data) and the `Close WRITE` and `Close RW` requests, equivalent to `STOREDATA` .

The portion of the xFS data used in this thesis spans seven days from

1:01:06A.M. on Saturday September 18, 1993 to 1:07:15A.M. on Saturday September 25, 1993. It contains requests from 237 clients, references to 127,215 unique `FID`s, 4,250,065 `Block READ` requests, 6,864 `Close RW` requests, 97,729 `Close WRITE` requests and 882,625 `Block WRITE` requests.

3.6 Summary

Table 3.2 summarizes characteristics of the four data sets.

	6-day	**24-day**	**2-month**	**xFS**
records	413,493	1,151,321	12,192,933	5,237,283
clients	327	486	37	237
files	176,733	320,725	288,599	127,215
`read`	NA	NA	8,213,455	4,250,065
`write`	NA	NA	3,215,928	882,625
`FETCHDATA`	269,781	739,557	526,785	NA
`STOREDATA`	143,712	411,764	326,765	104,593

Table 3.2: **Summary of the data sets.**

CHAPTER 4

SIMULATION AND EVALUATION

4.1 Introduction

There are several methods available for evaluating the performance of new computer constructs, applicable across a variety of domains: communication in a multiprocessor machine, network performance, processor cache performance, file system cache performance, etc.

An obvious method for this task is to build a prototype of the system one wishes to evaluate. Prototypes are likely to provide the most accurate information about the performance of a production system, and construction of a prototype can provide important information about implementation issues (and pitfalls) that might not otherwise occur to system designers. However, the resources may not exist to build a prototype, and less costly methods may prove a system unworkable without the expense of prototyping.

Another possibility is to use mathematical models to determine theoretically how systems will perform. Unfortunately, in the case of our intermediate server evaluation, it is possible to construct models where the intermediate server is wildly successful or an abysmal failure, with little information to determine which model is closer to reality. For example, if we assume there is little data sharing among users and client caches are always extremely effective, the intermediate server will be ineffective. However, if there is data sharing among clients and/or client caches are occasionally too small, an intermediate server can improve system performance.

We choose to evaluate the intermediate server with simulations based on file system traces of actual users working with production file systems. In so doing, we avoid the complexity and cost of constructing a prototype, while gaining some insight as to how the intermediate server might affect client and server performance for typical file system usage patterns.

4.2 The Simulator

We perform experiments simulating distributed environments with a two-level cache design. In the simulated environments of the experiments, the client machines are connected to an intermediate server, which is in turn connected to primary file servers as shown in Figure 4.1. In the context of a mass storage system, the disk system interposed between clients and the mass store has the same architectural role as an intermediate server, while the mass store itself is architecturally identical to a primary server.

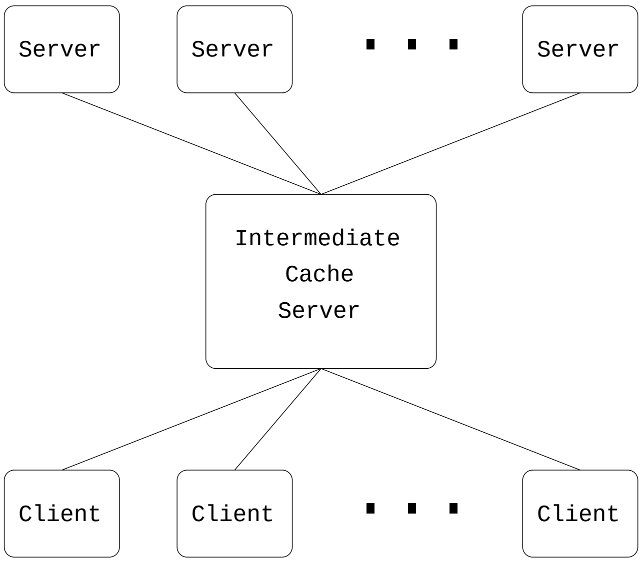

Figure 4.1: **Topology of the simulated environment.** File system requests from clients are passed to the intermediate server. If a request cannot be satisfied from the intermediate server's cache, the request is forwarded to the primary servers. Write requests cause invalidation of cached copies of the file on other clients.

In some of the experiments discussed in this paper, the intermediate server has a potentially infinite cache. This is obviously impractical. Because an intermediate server with a finite cache may be forced to flush its contents on occasion, the hit rates reported for these experiments are larger than can be achieved in reality. However, such experiments are useful for calculating upper bounds for finite cache simulations.

Simulations based on the 6-day and 24-day data sets are driven by `FETCHDATA` and `STOREDATA` requests. Simulations based on the 2-month data, however, are driven by `read` requests and `STOREDATA` requests. `STOREDATA` requests are used instead of write requests because in AFS not all write requests are passed to the servers. The servers become involved only when a `STOREDATA` request is issued after one or more `write` requests. The xFS data uses `Block READ` requests and `Close WRITE` and `Close RW` requests (equivalent to `STOREDATA`). In the following description, *read* refers to any of {`FETCHDATA`, `read`, and `Block READ`}, and *write* refers to any of {`STOREDATA`, `Close WRITE`, and `Close RW`}.

The operation of the simulator is straightforward. Trace records are processed in the order in which they were received by the AFS servers. When a read request for a file from the server machine appears in the trace data, the simulator checks the local cache on the requesting machine to see if the request can be satisfied there. If the requested chunk is found in the local cache, a *read hit* is logged for that client and the next trace record is processed. Otherwise, in the case of a read request, a *read miss* is recorded for the client, and the cache on the intermediate server is checked for the requested chunk.

If the chunk is found in the intermediate server's cache, a hit is recorded for the intermediate server, and the chunk is placed in the client's cache. Otherwise, a miss is recorded for the intermediate server, the chunk is installed in both the intermediate server's cache and the client's cache, and the next trace record is processed.

Write requests cause the requested chunk to be cached at the client issuing the request and at the intermediate server. Write requests are merely counted and are not assessed as hits or misses at this point.

In this way, the input of trace records is processed until exhausted. All read and write requests are guaranteed to succeed at the server. Cache hits and misses are only accounted for read requests. The simulator reports the number of read hits, read misses, and the number of write requests seen by each simulated machine. Unless otherwise specified, the cache replacement policy used at all levels is LRU.

When a file is written by a client, the simulator invalidates that file in the cache of any other client holding a copy, mimicking AFS caching behavior [15].

4.3 Limitation of the Server Data

Data sets collected at the AFS servers (the 6-day and 24-day data sets) consist only of file system requests received by the AFS servers (recall that client requests satisfied from the client caches are not sent to the AFS servers). Thus, intermediate server simulations involving client cache sizes smaller than the actual client cache sizes present during data collection may not be accurate. However, simulations of client cache sizes greater than or equal to actual client cache sizes can be seen to be accurate.

The following example demonstrates this problem. Suppose both server traces and client traces were collected in an environment where all clients are capable of caching only two chunks. Further suppose that a client issues the following requests (in order):

1. `FETCHDATA`(file 1, chunk 1)
2. `FETCHDATA`(file 2, chunk 1)
3. `FETCHDATA`(file 1, chunk 1)

The client traces will contain each of these requests, whereas the server trace will contain only the first two requests; the third is satisfied from the client's cache.

Suppose we now attempt to perform a simulation of an intermediate cache server with a 2,000-chunk intermediate cache and client caches that can hold only one chunk. Based on the server trace, simulation would calculate an intermediate hit rate of zero: `FETCHDATA`(file 1, chunk 1) would miss in both the client and intermediate caches as would `FETCHDATA`(file 2, chunk 1). The client trace simulation would produce the correct answer of 0.33: the initial `FETCHDATA`(file 1, chunk 1) requests misses both the client and intermediate caches, the `FETCHDATA`(file 2, chunk 1) misses both caches and causes (file 1, chunk 1) to be flushed from the client's one-chunk-sized cache, the final `FETCHDATA`(file 1, chunk 1) requests misses the client cache but hits at the intermediate cache. If the simulation were performed to simulate an environment in which client caches could hold two or more chunks, then the server trace simulation would return the correct answer.

Simulations involving client cache sizes greater than those present in the trace collection environment are correct because LRU is a *stack algorithm* [26]. This means that the chunks present in a cache at a given point in the simulation will be present in the same cache, at the same point in the simulation if the cache size is increased.

4.4 Evaluating Scalability

When evaluating the intermediate server's ability to improve scalability of existing systems, we operate under the assumption that the primary server distributed file system code cannot be modified. In the case of AFS, the implication of this assumption is that the intermediate server will not be able to reduce server request load for write requests because AFS caching is write-through. Thus, we focus on the intermediate server's potential for reducing read requests to primary servers.

One metric we use to evaluate the intermediate server is the *read hit rate*

$$\frac{read\ hits}{read\ requests} \tag{4.1}$$

i.e., the percentage of read requests satisfied by the intermediate server. We also compare the number of read requests that must be handled by primary servers in environments with and without an intermediate server.

4.5 Evaluating Mass Storage and Low-speed Network Performance

To evaluate client performance in the mass storage context, we are interested in the fraction of requests for which clients need not wait for access to the slow mass storage. Similarly, for the low-speed network environment, we are interested in the fraction of requests for which clients are not delayed while waiting for requests to traverse the low-speed link. So while the usual definition of hit rate counts writes as misses, in our model writes are absorbed by the intermediate server; only read requests that miss both the client cache and the intermediate server cache result in the client waiting for service by the mass store. Therefore the hit rate metric we use is

$$\frac{read\ hits\ +\ write\ requests}{total\ requests} \tag{4.2}$$

which we refer to as the *write hit rate*.

Of course, writes must eventually be propagated to the servers. If a client attempts a write when the intermediate's cache is full, the client could be delayed while the intermediate performs cache replacement (possibly involving writing data to the servers). However, simulations show that a sufficiently large intermediate cache, coupled with a policy for regularly flushing dirty chunks to the servers, makes this scenario unlikely (see Section 7.4). Simulating with a 128,000-chunk intermediate cache, and a policy of flushing dirty chunks on a daily basis we find that the clients never have to wait for the intermediate to perform a write to the servers.

In the case of low-speed network evaluation of the intermediate server, the write hit rate is overly optimistic. For example, the link bandwidth may be used (impeding client performance) if a write must be performed to maintain file system consistency.

4.6 Rationale For Using Hit Rate Metrics

Metrics other than those mentioned above could be used. Because the traces contain timestamps, one could attempt to evaluate how an intermediate cache server affects file system response time for client requests. We considered this metric, but concluded that if this metric were used, our results would be tied to the performance of the machines appearing in the traces *e.g.,* an RS/6000 AFS server satisfies requests faster than an IBM RT AFS server. Results about response time based on response time in the traces would not be widely applicable.

By using hit rate metrics, we provide results that are longer lasting and more widely applicable. If a researcher can determine the cost of hits and misses in her system, she can use our results to estimate the benefits of intermediate caching. This is because hit rates are independent of system performance, but system performance is highly dependent on the cost of cache hits and misses.

CHAPTER 5

GENERAL INTERMEDIATE CACHE SERVER ISSUES

5.1 Introduction

In this chapter we discuss intermediate cache server issues that are common to the three applications we are considering. Security concerns raised by similar systems discussed in Chapter 2 are addressed here. We also look at cache consistency, namespace, and performance issues.

5.2 Security

Doster and Rees discuss modifications to Kerberos that allow *intermediate translators* secure access to AFS files on behalf of clients [9]. An intermediate translator is a machine that performs file system protocol translation. For example an intermediate translator might accept NFS requests (behaving as an NFS server for clients) and supply AFS files (acting as a client to an AFS server), making AFS files available to machines that understand only the NFS protocol. Their authentication scheme provides a proxy authentication method, allowing intermediate translators to access the file servers on behalf of the clients. This is precisely the mechanism needed for intermediate cache server authentication. In fact, the intermediate translator can perform caching on behalf of its clients.

An intermediate cache server should be treated like an AFS file server with respect to physical security. It should be placed in a secure room—as secure as the location of the AFS servers. An exception to this is the remote intermediate server discussed in Chapter 8. File system data located on such an intermediate server would be similar to data located in remote client caches currently in use, so no new security threats are introduced. Accessing secure machines via secure protocols overcomes problems experienced in the systems proposed by Blaze [4] and Dahlin, et al. [8].

5.3 Cache Consistency

AFS uses a *callback* mechanism: when a client opens a file, it receives a callback—a guarantee that the server will notify the client if the file is changed. For example, if one client caches a file that is subsequently modified by another client, the first client is notified that its cached copy is no longer current. In this manner, clients are guaranteed a consistent view of the file system. When intermediate cache servers use a write-through caching policy, AFS consistency semantics are easily maintained: an intermediate cache server receives a callback from the primary server when it opens a file on behalf of a client. The intermediate server in turn issues callbacks to clients accessing files from the intermediate server's cache. The intermediate server maintains a list of clients holding callbacks for each file, and if the primary server notifies the intermediate server that its callback is no longer valid, the intermediate server revokes the corresponding callbacks held by the clients.

A delayed write policy at the intermediate server complicates matters. Server modifications could allow AFS consistency semantics to be maintained (*e.g.,* allowing the intermediate server to issue invalidation messages to servers). It is also possible that AFS consistency semantics may be sacrificed in certain cases.

5.4 File System Namespace

File naming is straightforward: the file system namespace is unchanged. The addition of an intermediate cache server for our applications should be completely transparent to end users.

5.5 Performance

If the intermediate cache exhibits a low hit rate, it will serve only to delay file system requests, as all intermediate cache misses end up being processed by the primary servers. The intermediate server in this case adds a useless extra step. Thus, "high" intermediate cache hit rates are important to the functioning of the intermediate server (although "high" is application dependent). Performance is discussed in detail in subsequent chapters.

CHAPTER 6

INFLUENCE ON SCALABILITY

6.1 Introduction

By reducing the number of client requests that must be handled by primary file servers, intermediate cache servers increase the number of clients that a given set of primary servers can support. As clients are added to a distributed file system, intermediate servers can be added to help prevent degradation of server performance. In this way, intermediate cache servers increase distributed file system *scalability.*

In this chapter we examine the ability of intermediate servers to increase scalability in existing distributed file systems—the usefulness of the intermediate server in environments where server code is left unmodified. In earlier work [19], we looked at intermediate cache performance with respect to client performance. We used the hit rate metric, counting all write requests as cache misses. Because AFS uses a write-though policy, the intermediate server cannot reduce write request load experienced by the servers—all writes must be passed through. Thus, in this chapter we look at the intermediate server's ability to reduce the number of read requests handled by the primary servers. We use the read hit rate

$$\frac{read\ request\ hits}{read\ requests} \tag{6.1}$$

and counts of read requests handled by primary servers for our evaluation.

6.2 Read Hit Rate Simulations

In the first set of experiments, we use simulations to calculate the read hit rates that can be expected for the intermediate server cache. Various finite intermediate cache sizes are simulated as well as an intermediate server with an unbounded cache. The unbounded cache experiments provide an upper-bound on the effectiveness of the intermediate cache.

64K (65,536 bytes) is used for the cache chunk size, because this is the size used by AFS clients in the local AFS cells.

6.2.1 6-day data read hit rate

The first experiment uses the 6-day data to calculate read hit rate at the intermediate server when an intermediate is positioned between the clients and the primary servers. Intermediate cache read hit rate is calculated for varying cache sizes as shown in Figure 6.1.

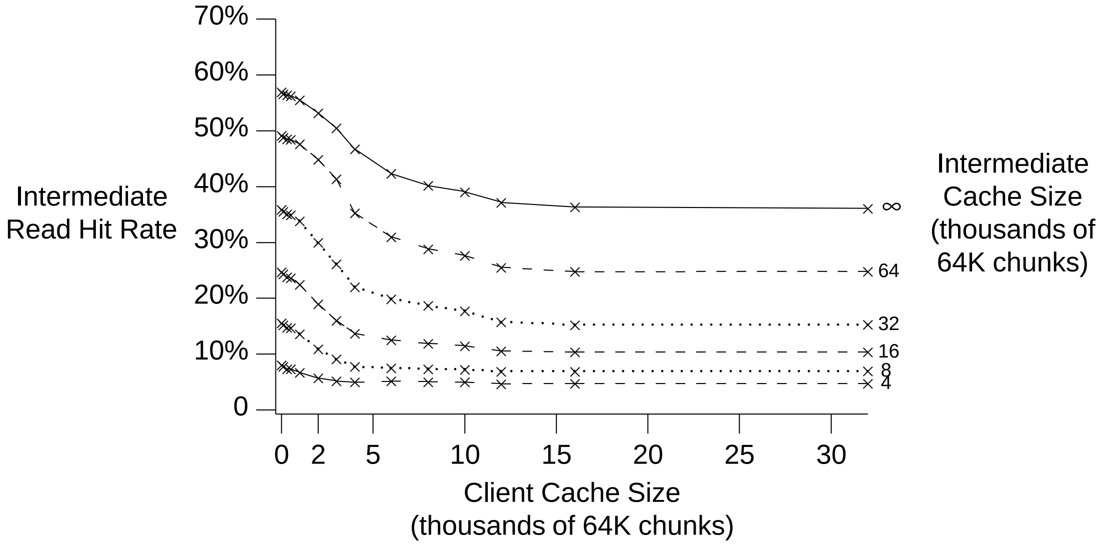

Figure 6.1: **Client cache size *vs.* intermediate server read hit rate with the 6-day data.** This graph shows the effect of client cache size, measured in thousands of 64K chunks, on intermediate cache read hit rates. The curves show variations in intermediate cache size, from top to bottom, respectively: infinite, 64,000 chunks, 32,000 chunks, 16,000 chunks, 8,000 chunks, and 4,000 chunks.

With clients capable of caching 2,000 chunks (typical in our environment), an intermediate server with a 64,000-chunk cache would be expected to reduce read requests to the main file servers by approximately 45%. An intermediate server would require a maximum of 4 GB of storage to cache 64,000 chunks, but in practice would require something closer to 1-2 GB, due to fragmentation (see Appendix A).

Notable in this graph and subsequent graphs is that as client cache size increases, the intermediate read hit rate curves do not cross. The implication of this is that if the only factor one is concerned with is intermediate hit rate, all disk cache should be placed at the intermediate server. However, eliminating client caches (or using small client caches) greatly degrades client performance.

6.2.2 24-day data read hit rate

Higher intermediate server read hit rates are experienced with the 24-day data as seen in Figure 6.2. The "warm-up" period for the intermediate cache, the amount of time before the read hit rate stabilizes, is approximately four days. The warm-up period is a much smaller fraction of the 24-day data, which helps explain the somewhat higher read hit rates.

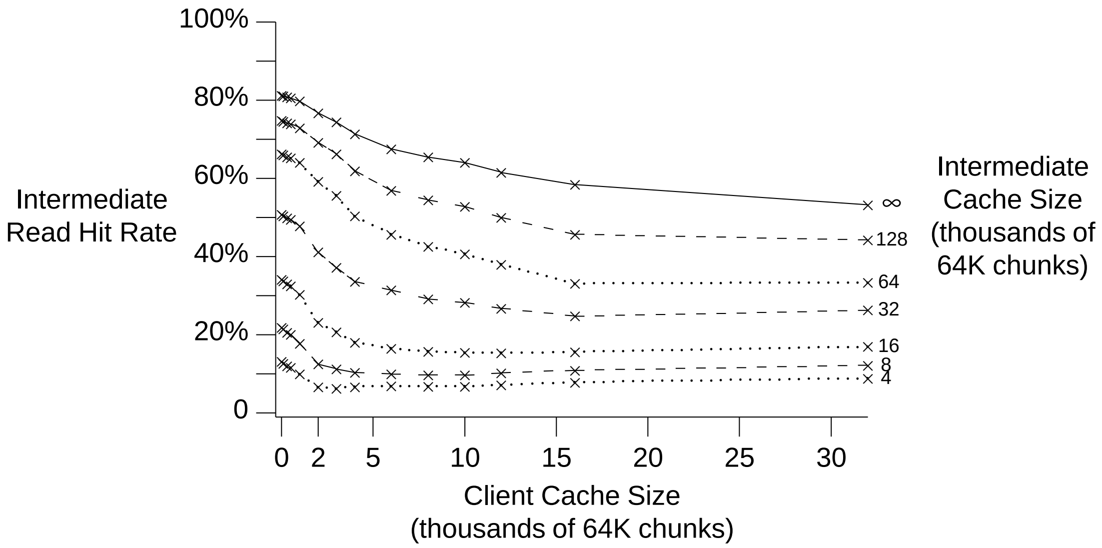

Figure 6.2: **Client cache size *vs.* intermediate server read hit rate with the 24-day data.** This graph shows the effect of client cache size, measured in thousands of 64K chunks, on intermediate server read hit rates. The curves show variations in intermediate cache size, from top to bottom, respectively: infinite, 128,000 chunks, 64,000 chunks, 32,000 chunks, 16,000 chunks, 8,000 chunks, and 4,000 chunks.

With 2,000-chunk client caches and a 64,000-chunk intermediate cache, results from the 24-day data predict a reduction in server read request load of approximately 60%. If the intermediate server's cache is increased to 128,000 chunks (8 GB maximum, 2-4 GB in practice), the expected reduction of read traffic rises to about 70%.

An interesting feature of this graph is the increase in intermediate server hit rate (with finite intermediate caches) as the client cache size grows. Client requests typically occur in bursts, and a large burst of requests from a single client can lower the effectiveness of the intermediate server, especially with respect to other clients. Larger client caches shield the intermediate server from some of this bursty traffic, thus raising intermediate server hit rates.

6.2.3 2-month data read hit rate

Figure 6.3 shows the 2-month data read hit rates. The 2-month data contains all references to AFS files performed on the client machines, so simulations are accurate for all client cache sizes.

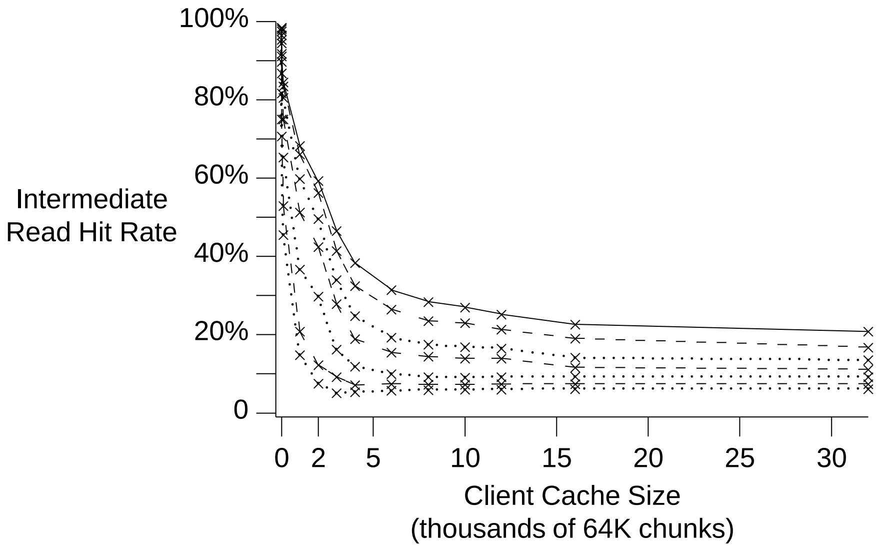

Figure 6.3: **Client cache size *vs.* intermediate server read hit rate with the 2-month data.** This graph shows the effect of client cache size, measured in thousands of 64K chunks, on intermediate server read hit rates. The curves show variations in intermediate cache size, from top to bottom, respectively: infinite, 128,000 chunks, 64,000 chunks, 32,000 chunks, 16,000 chunks, 8,000 chunks, and 4,000 chunks.

Although the read hit rates drop off rapidly as client cache size increases, they are still in the 40%-60% range for typical client cache sizes coupled with modest intermediate caches. The intermediate cache read hit rates decrease more drastically with the 2-month data set than with the 6-day or 24-day data sets. This is *probably* due to higher hit rates at the client caches with the 2-month data, though this hypothesis cannot be tested because client cache simulations cannot be performed on the server-based data sets. The low number of shared binaries among the 2-month data clients also contribute to lower intermediate read hit rates.

6.2.4 xFS data read hit rate

Results from the xFS data are similar to the results based on the other data sets, as shown in Figure 6.4. As with the 6-day data, the short duration of the xFS data, 7 days, accounts for some depression of the read hit rate numbers for these simulations.

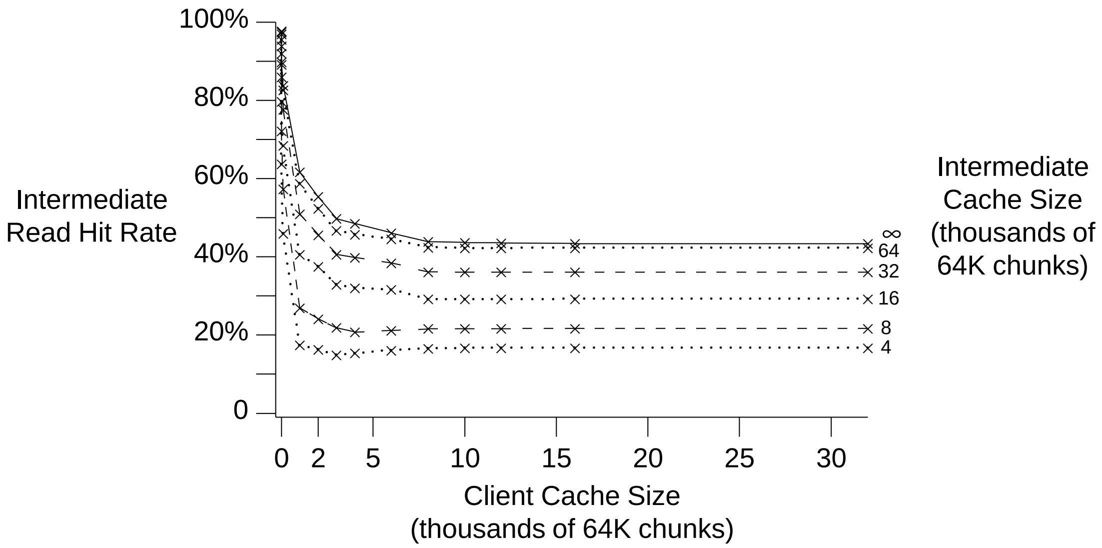

Figure 6.4: **Client cache size *vs.* intermediate server read hit rate with the xFS data.** This graph shows the effect of client cache size, measured in thousands of 64K chunks, on intermediate server read hit rates. The curves show variations in intermediate cache size, from top to bottom, respectively: infinite, 64,000 chunks, 32,000 chunks, 16,000 chunks, 8,000 chunks, and 4,000 chunks.

2,000-chunk client caches and a 64,000-chunk intermediate cache result in an intermediate read hit rate of 52.4%—higher than the 6-day and 2-month data results, but lower than the 24-day data result.

6.3 Read Hit Rate Over Time

Because the intermediate cache starts out empty (or *cold*) the intermediate read hit rate increases over time. The following experiments examine the time needed for the intermediate cache to become *hot*. A 128,000-chunk intermediate cache and 3,000-chunk client caches were chosen for these simulations. The results are similar to those generated from other configurations.

Figure 6.5, based on the 6-day data collection, spans approximately 146 hours. It shows that the intermediate server hit rate starts to level out at around four days into the traces, though a longer trace duration is necessary to confirm this.

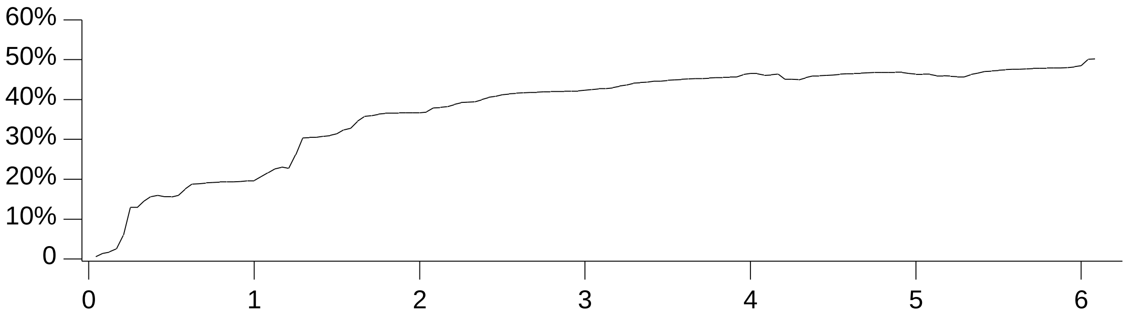

Figure 6.5: **Intermediate cache warming up (6-day data).** This graph is based on a client cache size of 3,000 chunks and an infinite intermediate cache and shows how the intermediate server hit rate increases over time (days). Data points are at 1-hour intervals.

The 24-day data clearly shows the intermediate server hit rate leveling off after approximately four days (Figure 6.6). From this graph, we conclude that the intermediate hit rate would would not rise substantially given more time to "warm up."

The graph of the 2-month data, however, does not level off after four days (Figure 6.7). This reflects the fact the the 2-month data set is biased towards non-shared files *e.g.,* files in user home directories. Thus, the intermediate server does not build (via LRU caching) a large collection of files that are shared by many clients—shared files being a primary source for intermediate server cache hits. An intermediate cache server fronting a group of clients from which few requests for shared files emanate may require a long period of use before its read hit rate levels off.

The "spike" at the beginning of Figure 6.7 (representing the first three one-

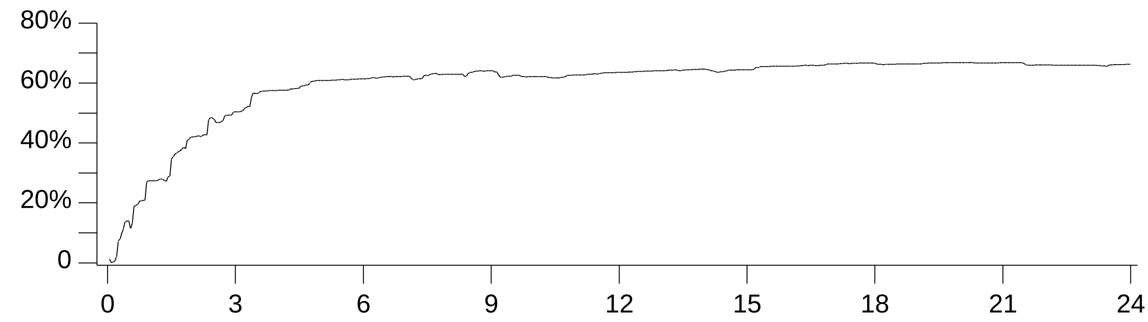

Figure 6.6: **Intermediate cache warming up (24-day data).** This graph is based on a client cache size of 3,000 chunks and a 128,000-chunk intermediate cache and shows how the intermediate server hit rate increases over time (days). Data points are at 1-hour intervals.

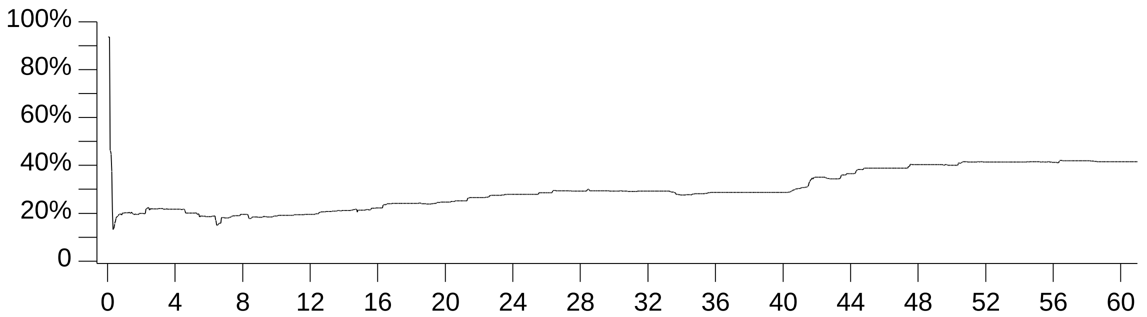

Figure 6.7: **Intermediate cache warming up (2-month data).** This graph is based on a client cache size of 3,000 chunks and a 128,000-chunk intermediate cache and shows how the intermediate server hit rate increases over time (days). Data points are at 1-hour intervals.

hour intervals) is due to system tasks running shortly after the start of trace collection. Since the client caches start out empty, the first client to execute these tasks causes the remote files associated with these tasks to be cached at the intermediate server. Other clients, starting with empty caches, subsequently running these tasks, cause a high percentage of hits at the intermediate server.

The xFS data warming graph (Figure 6.8) is quite similar to that of the 6-day data. The short duration of the xFS trace makes assessment of the leveling trend difficult.

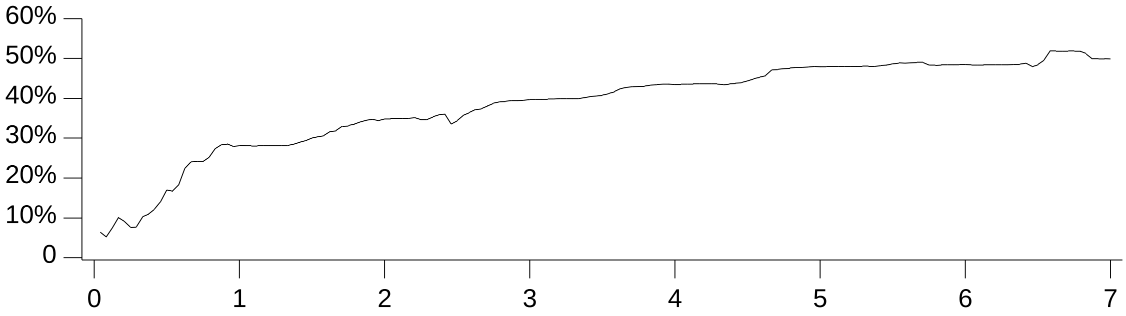

Figure 6.8: **Intermediate cache warming up (xFS data).** This graph is based on a client cache size of 3,000 chunks and a 128,000-chunk intermediate cache and shows how the intermediate server hit rate increases over time (days). Data points are at 1-hour intervals.

The hit rate over time experiments show that an intermediate cache, starting out cold, undergoes most of its warming during the first three to four days of use (with the exception of the 2-month data). Hit rate over time graphs can be useful for eliminating the effect of warm-up time on hit rate results. For a given data set, the hit rate over time graph can be generated and the warm-up period determined. Simulations can then be run that do not perform hit or miss accounting until after the warm-up time has passed.

6.4 Effect on Upstream Server Load

If intermediate caching is effective, the request load presented to a server is reduced. In this section, we consider the effect of the intermediate server's cache on upstream servers. To this end, we ran a set of simulation experiments to compare the read request rate seen by upstream servers when an intermediate server is present and when it is absent. Client writes are always presented to the upstream servers, whether or not an intermediate server is employed; since the intermediate server can have no effect on upstream server performance for writes, we ignore them and concentrate on read operations. In these experiments, client cache size is fixed at 3,000 chunks and load reduction for various intermediate cache sizes is simulated.

6.4.1 Server load reduction with the 6-day data

The first set of experiments are based on the 6-day data. Simulations involve finite intermediate cache sizes ranging from 500 chunks to 256,000 chunks as well as infinite intermediate cache experiments.

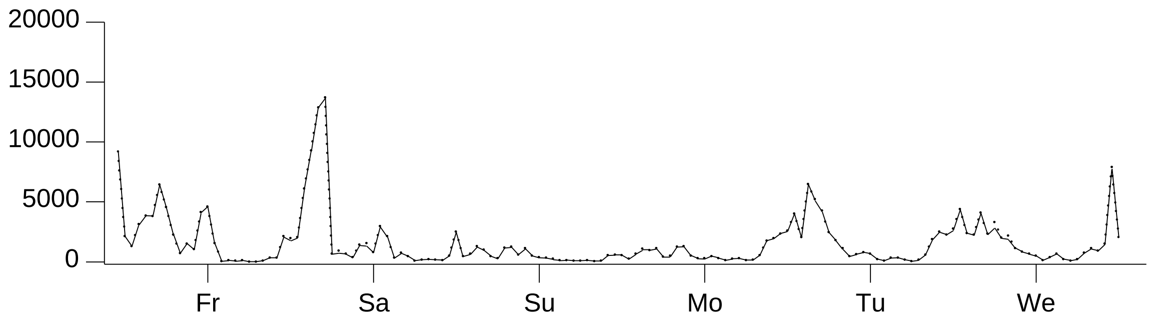

Figure 6.9: **Server load reduction with the 6-day data (1,000-chunk intermediate cache).** This graph shows the effect an intermediate server with a small cache (1,000 chunks) has on the number of read requests that must be processed through upstream servers. The dotted line shows the number of read requests that must be handled by the servers when the intermediate server is absent. The solid line shows the number of read requests the servers must handle when the intermediate server is present. With an intermediary cache of only 1,000 chunks, the two graphs substantially coincide. Read request counts are based on 1-hour intervals.

Figure 6.9 shows the request rate, in requests per hour, presented to upstream servers when an intermediate server is present (solid line) and when it is absent (dotted line). In Figure 6.9, these lines essentially coincide. Not surprisingly, the small intermediate cache, 1,000 chunks, yields virtually no reduction of server load. Load reduction is negligible for intermediate cache sizes less than 32,000 chunks. At 32,000 chunks there is some "peak-clipping," and at 64,000 chunks there are periods where load reduction is more noticeable as shown in Figure 6.10. An intermediate server with a cache size of 64,000 chunks (4 GB maximum, 1-2 GB in practice), would reduce the number of read requests to the primary servers by 41% in the simulated environment.

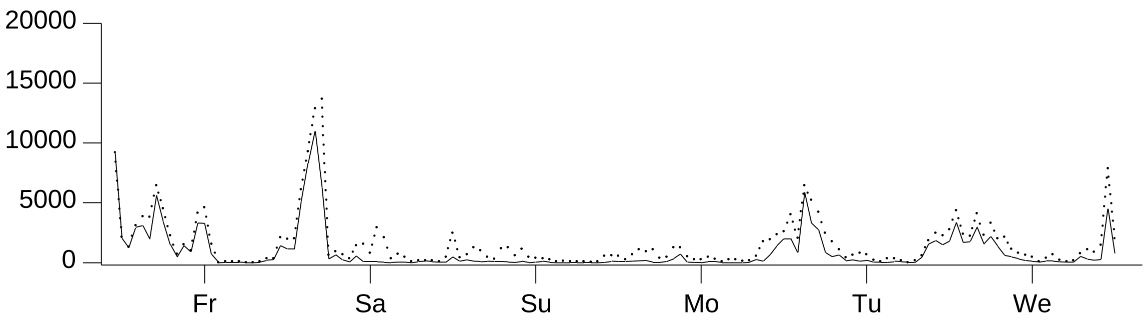

Figure 6.10: **Server load reduction with the 6-day data (64,000-chunk intermediate cache).** This graph shows the effect an intermediate server with a 64,000-chunk cache has on the number of read requests that must be processed through upstream servers. The dotted line shows the number of read requests that must be handled by the servers when the intermediate server is absent. The solid line shows the number of read requests the servers must handle when the intermediate server is present. Read request counts are based on 1-hour intervals.

At an intermediate cache size of 128,000 chunks there is significant load reduction at the two largest peaks (other than at 0), particularly in the peak occurring on Wednesday as shown in Figure 6.11. Based on average data chunk sizes found in our client caches, 128,000 chunks would require approximately 2 GB of storage (see Appendix A). Overall, the number of read requests reaching the primart servers is reduced by the intermediate server from 233,239 to 116,141, or 50%.

Figure 6.12 shows the percentage reduction in read requests reaching the server for various intermediate cache sizes for the 6-day data. A simulation was run in which the intermediate cache was unbounded. The dashed line at 50.4% in Figure 6.12 represents the load reduction when the intermediate server has an infinite cache.

6.4.2 Server load reduction with the 24-day data

The pair of graphs in Figure 6.13 show the request rate reduction caused by an intermediate server with a 128,000-chunk cache over the 576 hours of the 24-day data. With an intermediate cache size of 128,000 chunks, read requests handled by the server are reduced from 542,122 to 183,324 for a reduction of 66.2%. Significant peak-clipping occurs many times over this interval. An intermediate cache size of