CITI Technical Report 95–9

Evaluating Delayed Write in a Multilevel Caching File System

D. Muntz, P. Honeyman, and C.J. Antonelli

Center for Information Technology Integration
The University of Michigan
Ann Arbor

ABSTRACT

Delayed write in a multilevel file system cache hierarchy offers a way to improve performance in diverse client/server scenarios, such as integrating mass store into a distributed file system or providing distributed file system access over low-speed links. Using file system traces and cache simulations, we explore extensions and modifications to the traditional client caching model employed in such file systems as AFS, Sprite, and DFS.

High cache hit rates at an intermediate cache server—a machine logically interposed between clients and servers that provides cached file service to the clients—combined with high client cache hit rates lend practicality to an integrated mass storage file system. In such a system, magnetic tape or optical-based mass storage devices may be used as a first-class data repository, fronted by disk and RAM caches to offer acceptable access times to the large, but slow, mass storage system.

Similarly, a high cache hit rate is necessary for users accessing file systems via low-speed links, where a delayed write intermediate caching server can mediate traffic to make better use of available bandwidth. In an example taken from mobile computing, an intermediate server might be used as a docking station at a user's home. This arrangement would be convenient for users of mobile computers who upload large amounts of data generated while operating in disconnected mode. Simulations of delayed write caching strategies are applicable to both the mass storage and low-speed link scenarios.

October 24, 1995

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

Evaluating Delayed Write in a Multilevel Caching File System

D. Muntz, P. Honeyman, and C.J. Antonelli

Center for Information Technology Integration
The University of Michigan
Ann Arbor

1. Introduction

At the Center for Information Technology Integration (CITI), we are investigating potential scale- and performance-enhancing extensions to the traditional client caching model employed in distributed file systems such as AFS [1, 2], Sprite [3], and DFS [4]. The core of our research to date deals with intermediate cache servers: machines logically positioned between clients and servers, acting as both client and server in the file system. As a client, an intermediate server caches files and directories on the local disk. As a server, it satisfies client requests from the same local disk cache.

Intermediate servers reduce the request load of the upstream file servers by satisfying requests from client machines. Simulations from previous work indicate that, in practice, an intermediate AFS server with a 2GB cache would cut read traffic to primary servers in half [5].

We are studying the organization and optimization of an integrated mass storage system based on a cache hierarchy [6], similar to that used by Plan 9 from Bell Labs [7, 8]. The system consists of a file system based on a mass store, *e.g.*, magnetic tape or optical-based storage, attached to a file server with several gigabytes of disk cache and possibly hundreds of megabytes of RAM cache. As in our earlier studies of intermediate cache effectiveness, we simulate the proposed mass storage AFS system with traces collected from running systems.

Home and mobile computers connected to the Internet via low-speed links, *e.g.*, modems, are increasingly prevalent. AFS on such machines offers a consistent view of files, no matter what client machine is used, but the write-through caching policy makes AFS over low-speed links less convenient. Preliminary simulation results suggest that a delayed write policy would greatly increase performance in this situation.

In the remainder of this paper, we explore a delayed write scheme that automatically maintains AFS consistency semantics unless automated consistency maintenance proves to be unnecessary, *e.g.*, if remote usage patterns do not appear to justify the added cost. A delayed write policy reduces write latency, but at the cost of more complex consistency semantics. While the latter cost is difficult to estimate, the potential benefit can be measured through trace-driven simulation.

In the next section, we describe existing or proposed file system architectures that include intermediate file service. The sections that follow describe the operation of the simulator and the trace data that we use in our simulation. We then provide detailed analysis of the results of simulating delayed write in an intermediate AFS environment in the context of an application to mass store systems. We then extend our analysis to low-speed networks and conclude with a description of potential extensions to our work.

2. Related work

Much of the work on multilevel caches examines processor memory caches, which show great success in improving CPU performance [9]. Multilevel caches in hardware typically have high hit rates. This and the large difference in access times at the various levels of the cache hierarchy explain the success of multilevel caching in processor memories [10]. 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 [11] investigate the effectiveness of a dynamic cache hierarchy in a distributed file system environment. They conclude that their system could cut the 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. Clients in this system can act as file servers for other clients, potentially complicating security and archival issues.

Dahlin *et al.* investigate xFS, an experimental file system in which client machines handle many of the tasks normally performed by file servers [12]. For example, 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 has potential complications similar to the work done by Blaze and Alonso, albeit a greater payoff.

Makaroff and Eager consider the performance of caching in systems where caching takes place at both clients and servers [13]. They conclude that client caching degrades server cache effectiveness 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.

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. Because Plan 9's WORM drives have much smaller seek times than do CITI's tape devices, avoiding access to mass storage is less important to Plan 9 researchers.

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 [14]. Their conclusion on the importance of prefetching may be applicable to the CITI mass storage project.

Antonelli and Honeyman describe the integrated mass storage system under consideration at CITI [6]. This system uses the mass store as a first-class data repository, with a disk-based file system and 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.

Huston and Honeyman, in their research into disconnected operation for AFS, implement a partially-connected mode of file system operation [15], in which a client maintains a consistent cache but performs write operations only when excess bandwidth is available. This has a very beneficial effect on system performance as measured by conventional file system benchmarks, largely due to the performance improvements consequential to delayed write. Although a client maintains a cache that is consistent with respect to the server's state, distributed consistency is sacrificed somewhat, in that the servers and other clients are not immediately made aware of a partially connected client's updates. Should strict consistency prove critically important, Srinivasan and Mogul's work on Spritely NFS [16, 17] and Transarc's DFS [4] provide mechanisms for maintaining consistency while enjoying the performance advantages of delayed write.

3. The data

Our simulations use four sets of file system traces. The first 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 10:23:54 A.M. on Thursday, April 23, 1992 to 12:57:48 P.M. on Wednesday, April 29, 1992†. Data collection took place on the seven AFS servers described in Table 1.

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. The data references 176,733 different files in 413,493 trace records.

The second data set, which includes a portion of the first, spans a substantially longer period: 24 days from 6:34:04 A.M. on Tuesday, April 14, 1992 to 7:26:14 A.M. on Friday, May 8, 1992. This data collection took place on the file servers `homer` and `marge` and consists of requests from 486 clients. The data references 320,725 distinct files in 1,151,321 trace records. 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.

AFS clients exchange file data with the server via `FETCHDATA` and `STOREDATA` requests, whose functions follow directly from their names. Each `FETCHDATA` and `STOREDATA` request contains a timestamp, the

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

name	CPU	OS	disks
marge	IBM ES/9000 Model 720	AIX/370 V1.1	IBM 3380
homer			
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 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`.

client's network address, the file's `FID` (unique identifier for a file), and the offset and length of the data being requested. `STOREDATA` requests comprise 143,712 (35%) of the 6-day requests and 411,764 (36%) of the 24-day requests.

The first two data sets consist of file system requests received by the AFS servers; requests that were satisfied in the clients' caches do not appear in the data sets. Simulation results based on these data sets are accurate when simulated client cache sizes are greater than the actual client cache sizes; we return to this point in Section 4.

The third data set was collected from 37 IBM RS/6000 AFS client machines at CITI over a 2-month period from 3:07:00 A.M. on Wednesday, October 20, 1993 to 3:07:00 A.M. on Monday, December 20, 1993. This data set contains all read, write, `FETCHDATA`, and `STOREDATA` requests issued by each client machine for files in AFS. The 2-month data set contains 8,213,455 read requests, 3,125,928 write requests, 526,785 `FETCHDATA` requests, and 326,765 `STOREDATA` requests. 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`. References to these files do not appear in the traces.

For the 6-day and 24-day data sets, `FETCHDATA` and `STOREDATA` requests are the trace records of interest for our simulations. For the 2-month data, though, we are interested in the 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 following a series of writes, *e.g.*, when a file is closed. Read requests are used instead of `FETCHDATA` requests because `FETCHDATA` requests appear in the client traces only when cache misses occur at a client; using read requests allows simulations for all client cache sizes. This strategy also allows us to study alternative client cache replacement policies, although that is beyond the scope of this paper.

The fourth data set was collected by the creators of the experimental file system, xFS [12], to evaluate their system through simulation. Blaze's `rpcspy` program [18] was used to monitor network activity and generate traces for NFS clients on four Ethernets. During data collection, 4% of all network activity was dropped.

Because `close` system calls do not appear in NFS network traffic, Dahlin *et al.* used heuristics in a post-processing step to infer 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 `BlockREAD` requests (equivalent to `read` requests in the 2-month data) and the `CloseWRITE` and `CloseRW` requests (equivalent to `STORE-DATA` requests in the 2-month data).

The portion of the xFS data used in this paper spans seven days from 1:01:06 A.M. on Saturday, September 18, 1993 to 1:07:15 A.M. on Saturday, September 25, 1993. It contains requests from 237 clients, references to 127,215 unique `FIDs`, 4,250,065 `BlockREAD` requests, 6,864 `CloseRW` requests, 97,729 `CloseWRITE` requests, and 882,625 `BlockWRITE` requests.

4. The simulator

To determine the feasibility of the integrated mass storage system under consideration at CITI, we use trace-driven simulation to estimate how often file system users must wait for access to the slow‡ mass storage media.

In earlier work, we have simulated a distributed file system with a two-level cache design [19]. In those (simulated) environments, client machines are connected to an intermediate server which is in turn connected to a principal file server, as shown in Figure 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.

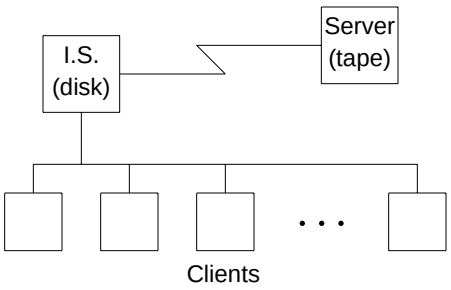

Figure 1. Topology of the intermediate server environment. File system requests from clients are passed to the intermediate server (I.S.). If the request cannot be satisfied from the intermediate server's cache, the request is forwarded to the primary server. Write requests cause invalidation of cached copies of the file on other clients.

The operation of the simulator is straightforward. Trace records are processed in the order in which they were received by the AFS servers, or in the case of the 2-month data, in the order in which they were issued. (All clocks are synchronized via NTP [20].) When a `FETCHDATA` request (read request for the 2-month data) 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, the simulator records a ''hit'' for that client and processes the next trace record. Otherwise, in the case of a `FETCHDATA` (read) request, the simulator records a ''miss'' for the client and checks for the requested chunk in the intermediate server's cache.

If the chunk is found in the intermediate server's cache, the simulator records a hit and places the chunk into the client's cache. Otherwise, it records a miss and installs the chunk in both the intermediate server's cache and the client's cache, and processes the next trace record.

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

To evaluate client performance, we are interested in the fraction of requests for which clients need not wait for access to the slow mass storage. 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

$$\frac{\text{read hits} + \text{write requests}}{\text{read requests} + \text{write requests}}$$

which we refer to as the *write hit rate,* in contrast to

$$\frac{\text{read hits}}{\text{read requests} + \text{write requests}}$$

which is the usual definition of hit rate.

‡ Mass storage is considered slow, when compared to conventional disk storage media, either because of high access latencies, low data transfer rates, or both.

Writes must eventually be propagated to the primary server. If a client attempts a write when the intermediate server's cache is full, the client may be forced to stall while the intermediate server performs cache replacement, possibly involving writing data to the servers. Our simulations show that a sufficiently large intermediate cache coupled with a policy for regularly flushing dirty blocks to the servers makes this scenario unlikely. A simulation with the 24-day data, a 128,000-chunk intermediate server cache, and a policy of flushing dirty chunks on a daily basis indicated that the clients need never wait for the intermediate server to write to the servers. This is also true for the other data sets.

All read and write requests are guaranteed to succeed at the server. Cache hits and misses are accounted only for `FETCHDATA` (read) requests. The simulator reports the number of `FETCHDATA` (read) hits, `FETCHDATA` (read) misses, and the number of `STOREDATA` requests seen by each simulated machine. 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 as closely as possible [21].

AFS clients typically cache 500 to 2,000 data chunks. Results from simulating client cache sizes greater than those present in the trace collection environment are accurate because LRU is a *stack algorithm* [22], which implies that those chunks present in a cache at any point in the simulation will be present at the same point in a simulation with a larger cache.

We measure all cache sizes as a number of chunks, *e.g.,* we simulate the operation of an intermediate server with a 128,000 chunk cache. With a chunk size of 64 KB, a cache capable of holding 128,000 chunks requires approximately 8 GB of disk space in the worst case. However, it is rare for a file to break into a whole number of chunks, *e.g.*, many files are much smaller than a chunk, so chunks are not fully utilized. In our simulations, we found chunk utilization to average about 33%, *i.e.,* most of the time a chunk capable of holding as much as 64 KB of file data holds 20 KB or less. To estimate the storage requirement in bytes for a cache of n chunks, a simple rule of thumb is to multiply the number of chunks by 20 KB. Thus, we would expect a 128,000 chunk cache to consume about 2.5 GB.

Metrics other than hit rate could be used, *e.g.*, the traces contain timestamps so we could simulate the effect of an intermediate cache server on response time for client requests. We considered this metric, but concluded that our results would be influenced by the performance of the machines appearing in the traces. For example, an RS/6000 AFS server satisfies requests more quickly than an IBM RT AFS server, adversely affecting the accuracy and generality of simulated response times. On the other hand, hit rate metrics can be used to determine the response time of any system once the cost of hits and misses for that system are determined.

Hit rates are independent of system performance, but performance is highly dependent on the cost of cache hits and misses. For example, if a miss costs many tens of seconds as is common in a mass store environment, then all misses are deadly.

Baker *et al.* report that most files are quickly deleted or overwritten [23]; between 65% and 80% of files are destroyed within 30 seconds of creation. Thus, simulations to determine the parameters of an efficient delayed write policy in the integrated mass storage environment are of substantial interest.

5. Hit rate simulations

In environments where access to the intermediate server is substantially faster than to the primary servers, requests that access the primary servers are assessed as misses at the intermediate server. In the mass storage system we are modeling, once a write request is completed in the intermediate server's cache, the request is considered to be satisfied. If a write request does not require access to mass storage, it is counted as a hit.

Some write requests can lead to a cache miss: when overwriting a partial chunk, the client first reads the complete chunk. If this read request cannot be satisfied out of the cache, a miss is assessed. Conceivably, the intermediate server could cache partially written chunks, eliminating the need to perform the read, but this is tricky to coordinate and does not correspond to the actual operation of an AFS server, so we do not assume this optimization. Read requests that miss both the client cache and the intermediate server's cache are counted as misses; all other requests can be handled without accessing mass storage, so they are counted as cache hits.

Our simulation experiments use the four data sets to calculate the write hit rate at the intermediate server. We vary the client and intermediate server cache sizes and calculate the cache hit rate on the intermediate server with these values. The results of the simulations, shown in Figure 2, indicate great potential for a delayed write intermediate server in the role of shielding clients from lengthy delays in access to mass storage, even when intermediate cache sizes are modest.

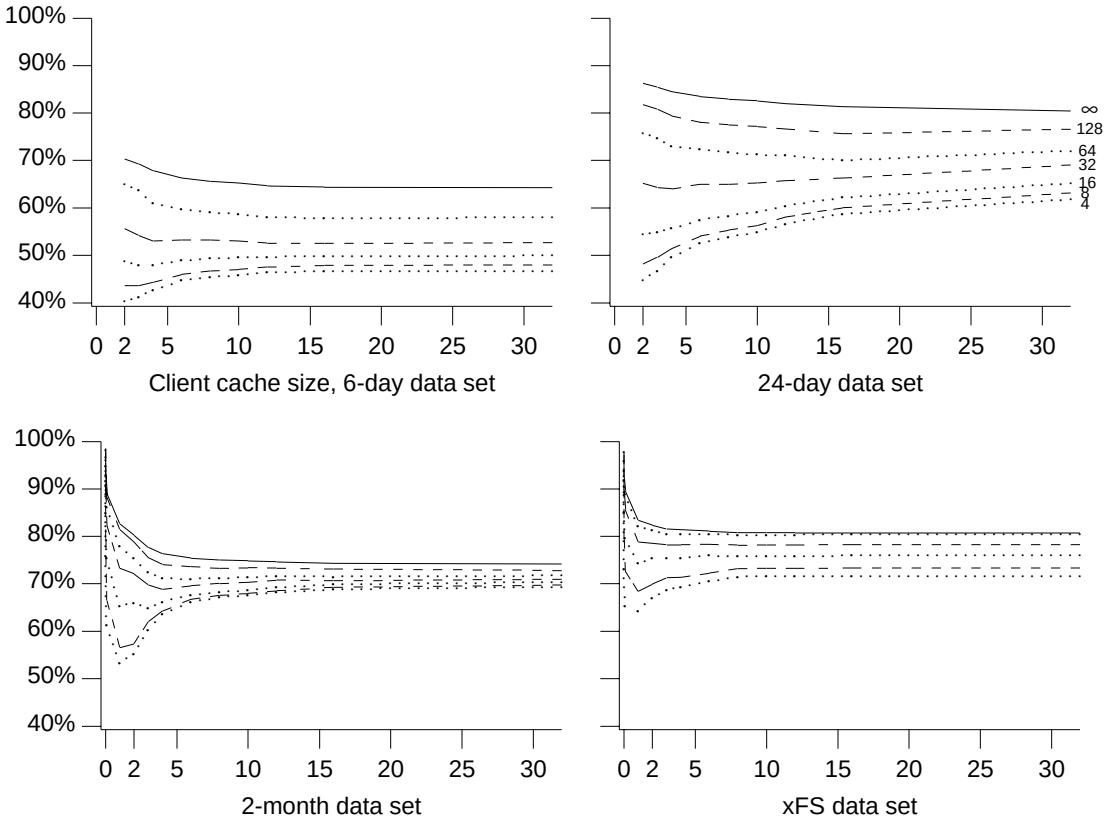

Figure 2. Intermediate cache hit rate *vs.* client cache size. These figures show how client cache size (measured in thousands of 64 KB chunks) affects hit rate at the intermediate server. The several curves show the effect on write hit rate as intermediate cache size is varied; the labels on the 24-day data figure give these sizes (in thousands of chunks).

In some cases the hit rate on the intermediate server grows as the client cache is increased. This is due to the client cache handling a greater number of read requests, which increases the relative frequency of write requests processed by the intermediate server. Because write requests are almost always cache hits at the intermediate server, the hit rate increases with client cache size (up to a point).

The relative insensitivity of the intermediate write hit rate to the intermediate cache size for the 2-month data is a consequence of the environment in which the data were collected. Unlike the other data sets, the 2-month data were collected from nearly identical machines used for software research and development activities by a small number of users, resulting in smaller client cache working set sizes, thus fewer client cache misses, and a concomitant decrease in the number of requests to the intermediate server.

Another way to view the effectiveness of the intermediate server is to fix the size of the client caches and vary the size of the intermediate server's cache. For these experiments, the client caches are fixed at 3,000 chunks, or, using our rule of thumb, about 60 MB. This is well within the range of typical values of client cache sizes in AFS. Figure 3 shows the effectiveness of various sizes of intermediate server caches for the four data sets.

Intermediate server effectiveness, as gauged by hit rate in an infinite intermediate server cache, is fairly uniform for all but the 6-day data. In this latter case, cache warm-up time consumes a larger percentage of the overall measurement period.

Beyond a 64,000 chunk cache, a little more than a gigabyte, adding more disk space offers diminishing

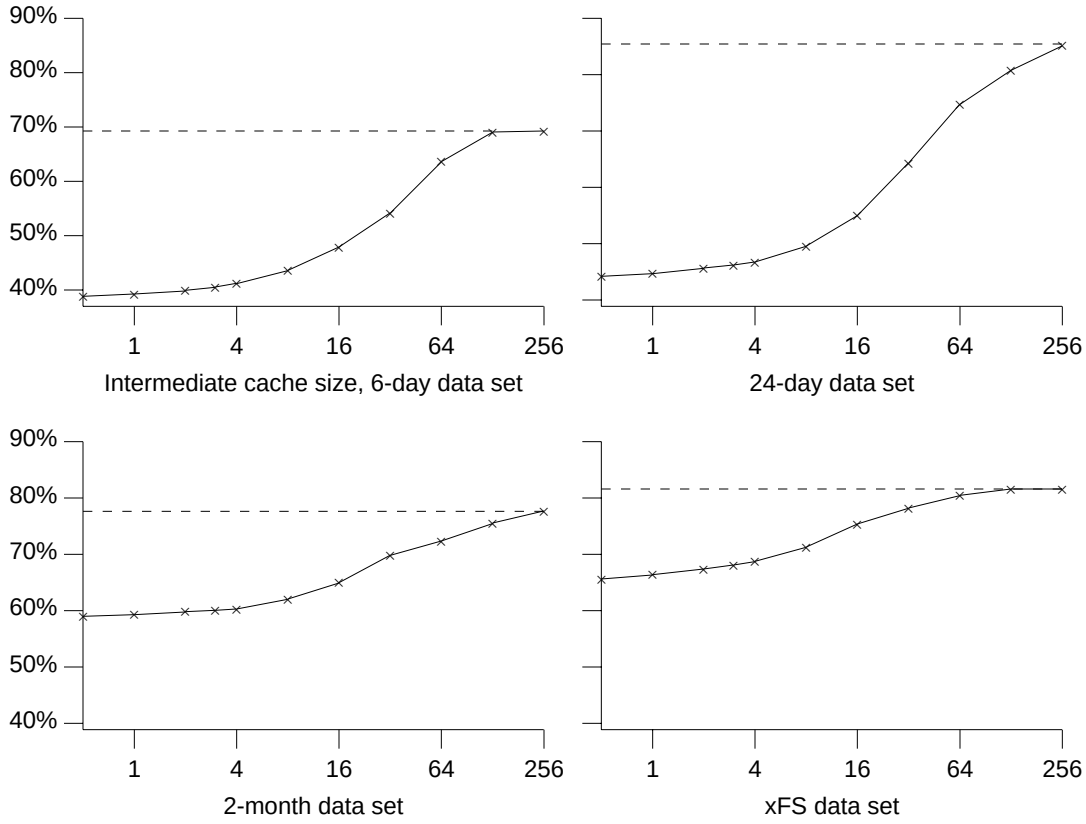

Figure 3. Percentage of requests satisfied by the intermediate server *vs.* intermediate cache size. These graphs show how intermediate cache size, measured in thousands of 64 KB chunks, influences the effectiveness with which the intermediate server handles requests not satisfied by the clients' caches. All clients are assumed to have 3,000-chunk caches. The dashed line at 69.3%, 85.4%, 77.6%, and 81.5% shows the effectiveness of an infinite intermediate server cache in the 6-day, 24-day, 2-month, and xFS data sets, respectively.

returns. With an intermediate cache of 128,000 chunks (or, about 2.5 GB), the hit rate is slightly less than in an infinite cache.

5.1. Client requests to mass storage

The high hit rates at the intermediate server result in a substantial reduction in requests to mass storage. Figure 4 shows in detail the effect of an intermediate server with 128,000 chunks (or, about 2.5 GB) on the client requests of the 24-day data set. The dotted line shows the requests seen by the intermediate server, while the solid line shows the requests passed to the mass store, *i.e.*, the cache misses at the intermediate server. Significant peak-clipping occurs throughout this interval. Requests handled by the server are reduced by 80.8%, from 953,661 to 183,324 over the 24-day period. Increasing the intermediate cache size to 256,000 chunks (or, about 5 GB) decreases requests to mass storage by 85.1%.

Graphs based on the other data sets also show substantial peak-clipping, as in Figure 4, although the request rates for the 2-month data set are lower, as it contains references from only 37 clients.

5.2. Discussion

The simulations described in this section indicate that intermediate servers may play an important role in the design of large distributed file systems based on slow mass storage systems. Intermediate servers that are capable of satisfying client write requests without incurring the penalty of accessing slow storage are shielded from client write traffic; similarly clients are protected from the lengthy delays associated with access to mass storage.

High intermediate cache hit rates coupled with high client cache hit rates make this application of delayed write in a multilevel caching file system a practical and inexpensive way of combining the advantages of

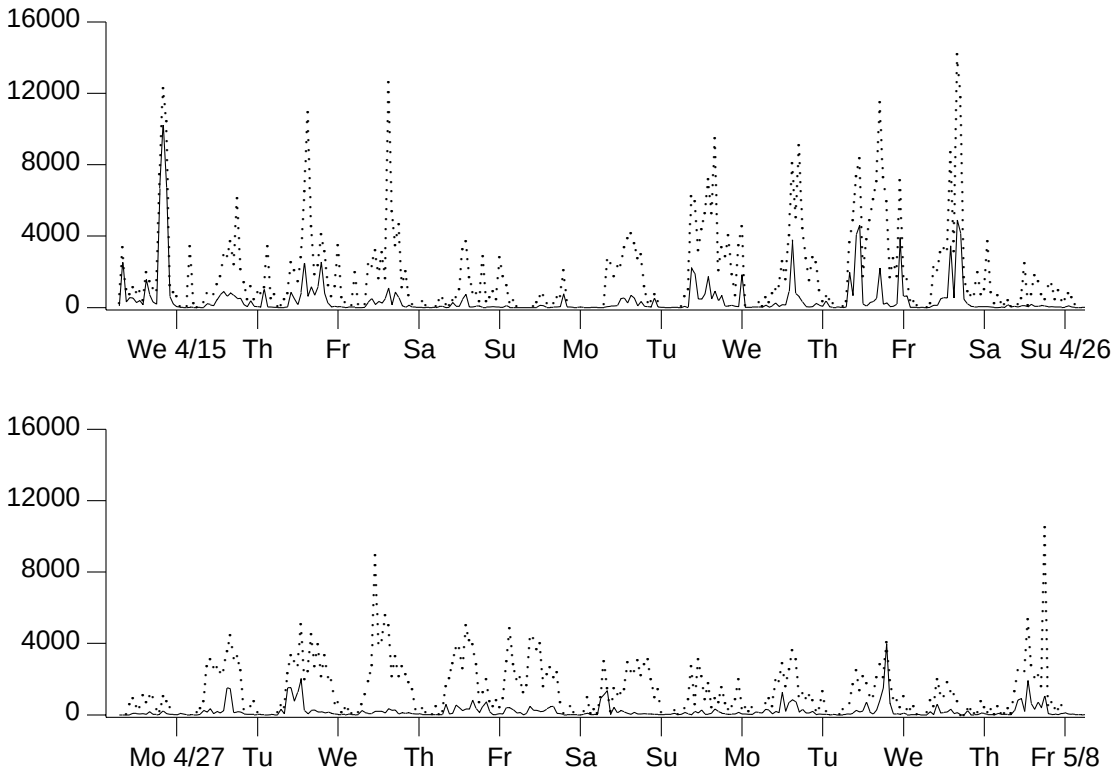

Figure 4. Slow storage request load reduction with the 24-day data. This graph shows the effect that an intermediate server with a 128,000-chunk cache has on the number of requests that must be satisfied from mass storage. The dotted line shows the number of requests handled by the mass storage system when the intermediate server is absent. The solid line shows the number of requests the mass storage system must handle when the intermediate is present. Load reduction is significant over most of the interval. Request counts are based on 1-hour intervals.

mass storage and distributed file systems. In the data sets we are working with, hit rates in the client caches are in the 80–90% range [5]. Of the remaining requests, we have shown that 60–80% are satisfied without incurring lengthy delays caused by the mass store. Our simulations indicate that clients will enjoy prompt responses to their requests 99% of the time or more.

6. Application to low-speed networks

The write hit rate provides a measure of client performance in delayed write caching systems where there is a significant penalty for performing the actual write operation to the primary file server. Another kind of slow write performance occurs with the use of distributed file systems over low-speed lines, *e.g.*, home and mobile computers accessing AFS with SLIP or PPP over dialup links. Delayed write caching for such machines may be particularly effective because write cost is so high. Data sharing is low in this configuration [24], reducing or eliminating the need for automated consistency checks. An intermediate server on the remote-side of the SLIP link can greatly improve performance by allowing client writes to complete when data have been stored in the intermediate server's cache.

An intermediate server in this scenario can be viewed as a ''docking station'' to which mobile home clients quickly off-load data that were written during disconnected operation [25]. Later, the intermediate server can convey write operations to the primary server, using overlapping RPC requests where possible.

In the case of home use, there are likely few or possibly only one machine accessing the distributed file system. Thus, one machine may act as both an intermediate server—offering files out of its cache to other local clients—and as a client.

A delayed write policy offers the ability to avoid writing data that are subsequently overwritten or deleted (a frequent occurrence [23, 26]), which could substantially benefit low-speed network users. Writes that cannot be avoided could be delayed until a time when traffic over the slow link is low. Delayed write

caching may be useful as an option that can be turned on and off by hand. For example, if a user wishes to build a kernel at home, delayed write could be enabled manually to defer storing object files until long after the build, if at all.

A policy of periodically flushing written data may be sufficient for low-speed network use. The period must be fairly large or the benefit of delayed write is lost. Deferring writes for 12- or 24-hour periods—performing the writes during low-usage periods—could prove beneficial if synchronized with the backup schedule of the primary servers. Of course, users must also be given the option of flushing manually. An important advantage of these two approaches is that modifications to the AFS server are unnecessary.

In AFS, one client cannot see data written by another until the writer performs a `STOREDATA` operation, usually when the file is closed, whereupon it is guaranteed that all clients will see the new data when the file is opened. Delayed write subverts this guarantee. Where data sharing is high, protocol modifications would be required to support both delayed write and the AFS consistency guarantee. We are considering a scheme in which clients augment a delayed write policy by alerting the primary server when they have data that would ordinarily have been written. Bandwidth is conserved by sending a short notification instead of the actual data. Either the client or server could request that the actual data be sent at any point: the client might flush dirty data to free up cache space, while the server could request data to enforce consistency. This design is similar to and inspired by the Sprite [3], Spritely NFS [17], and DFS [4] protocols.

6.1. Simulations

When using the write hit rate to evaluate client performance in a mass storage system, writes are considered to be hits. However, in low-speed network scenarios, some writes must be performed at certain times, *e.g.*, writes necessary to maintain consistency, degrading client performance. Thus, the write hit rate is overly optimistic for these cases.

We therefore look at how an intermediate server performing delayed write caching affects the number of AFS requests that must be transferred over the low-speed link (other than at the periodic sync time). When there is no intermediate caching, all read misses and stores must be performed immediately. When an intermediate server is in use, we count the number of read misses, the number of dirty blocks flushed (blocks that have been modified but not written back to the server), and blocks that must be written to maintain file system consistency (dirty blocks referenced by other clients). Blocks written during the (daily) synchronizing process in the delayed write scheme are not assessed—they are presumed to occur when the client is quiescent, thus they do not adversely affect user performance.

For the simulations, we assume that each client machine in the 2-month data is actually an intermediate server and client (or equivalently, that the clients perform delayed write caching). The first experiment determines the effectiveness of a 3,000-chunk intermediate cache at reducing requests over the slow link for each of the clients in the 2-month data.

The average reduction per intermediate server with 3,000-chunk (or, about 60 MB) intermediate caches is 54%. The overall reduction is 32%. When cache size is increased to 16,000 chunks (or, about 320 MB), there is a 74% average reduction in the number of requests that must traverse the low-speed link at times other than the sync period. The overall reduction with 16,000-chunk caches is 55%.

6.2. Drawbacks of simulation

None of our data sets reflect usage patterns over low-speed networks—all were collected from machines on networks at least as fast as Ethernet. By basing simulations on this data, we are evaluating the ability of users to perform the same tasks over low-speed links that they perform over Ethernet; this may not reflect reality.

We considered gathering traces from low-speed network clients, but this presents several problems, *e.g.*, whether such traces reflect realistic usage patterns in the simulated environment. Adding a delayed write caching intermediate server on the client side of a low-speed link would greatly increase user-perceived file system performance. This might dramatically alter usage patterns, limiting the usefulness of the write-through traces. Furthermore, most low-speed network users at CITI do little more with the file system over dialup lines than read mail or perform other small tasks, which does not provide interesting grist for our simulation mill.

6.3. Discussion

Deferring write requests to low-usage periods can greatly benefit low-speed network users. Performing writes across slow links hinders interactive use and other file system requests. We find that a large fraction of requests—typically 50% to 75%—can be deferred until a daily sync is performed.

7. Conclusions and future work

Intermediate cache servers implementing a delayed write policy play a major role in the architecture of CITI's mass store-based file system. Intermediate servers are necessary due to the high seek times incurred in accessing a mass store. Our simulations of delayed write intermediate servers suggest that they can greatly reduce the number of requests that must wait for the mass store. We introduced the write hit rate as a metric for evaluating delayed write caching where write traffic can be delayed until the system is otherwise idle and suggest that this is the case with the mass store file system.

Delayed write intermediate cache servers may also prove useful in low-speed network environments. The high value placed on bandwidth in such environments makes it important to control and reduce traffic over the network. An intermediate cache server can delay file system writes for hours, giving transmission priority to interactive traffic, for example. Delayed data that are subsequently deleted need never traverse the slow link at all.

In the future, we would like to examine whether specialized caching strategies on the intermediate server and the clients could be employed to improve performance. The use of LRU as the cache replacement policy for both clients and intermediate servers may be counterproductive, due to the overlap in cached data. With the high cost of accessing slow mass storage, even small improvements in hit rate are important. We plan to develop and evaluate caching algorithms tailored to the mass storage and low-speed network applications, and to study caching algorithms that have been suggested by others. Willick *et al.* [27] and Korner [28] investigate alternatives to LRU caching for a distributed file system. We may perform simulations using their suggested caching schemes in the context of multilevel caching. However, it is difficult to guess the effectiveness of these caching policies for our use, because both papers use short trace durations and small client caches. We may also explore caching algorithms based on those suggested by Maffeis [29]. Specifically, variations of the File Length Algorithm may prove useful in environments where large files might otherwise (under LRU for example) cause performance problems.

References

1. J.H. Howard, ''An overview of the Andrew file system,'' pp. 23–26 in *Proc. Winter USENIX Conf.*, Dallas (1988).
2. J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M. West, ''Scale and performance in distributed file systems,'' *ACM Transactions on Computer Systems* **6**(1), pp. 51–81 (1988).
3. M. Nelson, B. Welch, and J. Ousterhout, ''Caching in the Sprite network file system,'' *ACM Transactions on Computer Systems* **6**(1), pp. 134–154 (1988).
4. M. Kazar, B. Leverett, O. Anderson, V. Apostolides, B. Bottos, S. Chutani, C. Everhart, W. Mason, S. Tu, and E. Zayas, ''DEcorum file system architectural overview,'' pp. 151–163 in *Proc. Summer USENIX Conf.*, Anaheim (1990).
5. D. Muntz, *Multilevel caching in distributed file systems*, PhD Thesis, University of Michigan (1995).
6. C.J. Antonelli and P. Honeyman, ''Integrating mass storage and file systems,'' in *Proc. 12th IEEE Symp. on Mass Storage Systems*, Monterey (1993).
7. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ''Plan 9 from Bell Labs,'' in *Proc. UKUUG Summer Conf.*, London (1990).
8. S. Quinlan, ''A cached WORM file system,'' *Software–Practice and Experience* **21**(12), pp. 1289–1299 (1991).

9. J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Morgan Kaufmann Publishers, Inc., Palo Alto (1990).

10. H. Bugge, E. Kristiansen, and B. Bakka, ''Trace-driven simulations for a two-level cache design in open bus systems,'' pp. 250–259 in *Proc. 17th Intl. Symp. on Computer Architecture*, Seattle (1990).

11. M. Blaze and R. Alonso, ''Dynamic hierarchical caching in large-scale distributed file systems,'' in *Proc. 12th Intl. Conf. on Distributed Computing Systems*, Yokohama (1992).

12. M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, ''A quantitative analysis of cache policies for scalable network file systems,'' pp. 150–160 in *Proc. SIGMETRICS '94*, Nashville (1994).

13. D. Makaroff and D. Eager, ''Disk cache performance for distributed systems,'' pp. 212–219 in *Proc. 10th Intl. Conf. on Distributed Computing Systems*, Paris (1990).

14. E. Miller and R. Katz, ''An analysis of file migration in a UNIX supercomputing environment,'' pp. 421–433 in *Proc. Winter USENIX Conf.*, San Diego (1993).

15. L.B. Huston and P. Honeyman, ''Partially connected operation,'' pp. 91–97 in *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, Ann Arbor (1995).

16. V. Srinivasan and J. Mogul, ''Spritely NFS: implementation and performance of cache-consistency protocols,'' Research Report 89/5, Digital Equipment Corporation Western Research Laboratory (1989).

17. V. Srinivasan and J. Mogul, ''Spritely NFS: experiments with cache-consistency protocols,'' in *Proc. 12th ACM Symp. on Operating System Principles*, Litchfield Park (1989).

18. M. Blaze, ''NFS tracing by passive network monitoring,'' pp. 333–343 in *Proc. Winter USENIX Conf.*, San Francisco (1992).

19. D. Muntz and P. Honeyman, ''Multilevel caching in distributed file systems,'' pp. 305–313 in *Proc. Winter USENIX Conf.*, San Francisco (1992).

20. David L. Mills, ''Network time protocol (version 3): specification, implementation and analysis,'' RFC 1305, Network Information Center, SRI International, Menlo Park (1992).

21. M. Kazar, ''Synchronization and caching issues in the Andrew file system,'' pp. 28–36 in *Proc. Winter USENIX Conf.*, Dallas (1988).

22. A.S. Tanenbaum, *Modern operating systems*, Prentice-Hall, Inc., Englewood Cliffs (1992).

23. M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, ''Measurements of a distributed file system,'' in *Proc. 13th ACM Symp. on Operating System Principles*, Pacific Grove (1991).

24. J.J. Kistler and M. Satyanarayanan, ''Disconnected operation in the Coda file system,'' *ACM Transactions on Computer Systems* **10**(1) (1992).

25. L.B. Huston and P. Honeyman, ''Disconnected operation for AFS,'' pp. 1–10 in *Proceedings USENIX Symp. on Mobile and Location-Independent Computing*, Cambridge (1993).

26. J. Ousterhout, H.L. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, ''A trace-driven analysis of the UNIX 4.2 BSD file system,'' in *Proc. 10th ACM Symp. on Operating System Principles*, Orcas Island (1985).

27. D. Willick, D. Eager, and R. Bunt, ''Disk cache replacement policies for network fileservers,'' pp. 2–11 in *Proc. 13th Intl. Conf. on Distributed Computing Systems*, Pittsburgh (1993).

28. K. Korner, ''Intelligent caching for remote file service,'' pp. 220–226 in *Proc. 10th Intl. Conf. on Distributed Computing Systems*, Paris (1990).

29. S. Maffeis, ''Cache management algorithms for flexible filesystems,'' *ACM SIGMETRICS Performance Evaluation Review* **21**(2) (1993).