Performance Implications of Delayed-Write Policy in a Multilevel File System Cache

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

Center for Information Technology Integration
The University of Michigan
Ann Arbor

ABSTRACT

Employing a delayed-write caching policy in a multilevel cache hierarchy may allow significant improvements in file system storage capacity as well as improving file system access over low-speed links. Cheap, high capacity mass storage systems can be viable as primary data repositories if front-end caching is used to mask the high access times typical of mass storage devices.

Using distributed file system traces and cache simulations, we explore extensions and modifications to the traditional client caching model employed in such file systems as AFS and Sprite. 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.

When clients and servers are separated by a slow link, a delayed-write intermediate caching server can mediate traffic over the slow link to make better use of link bandwidth. In an example taken from mobile computing, an intermediate server might be used as a ''docking station'' at a user's home for mobile computers that need to 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.

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 currently employed in file systems such as AFS [1, 2] and Sprite [3]. Specifically, we are evaluating various caching strategies and topologies through simulations using distributed file system traces.

The core of our research to date deals with intermediate cache servers: machines that are logically positioned between clients and servers. Intermediate servers act as file servers to file system clients and as clients to file servers. Intermediate servers reduce the request load of the primary file servers by responding to requests from client machines. Simulations from previous work indicate that, in practice, an intermediate server with a 2GB cache in an AFS environment would cut read traffic to primary servers in half [4].

We are studying the effectiveness and optimization of an integrated mass storage system based on a cache hierarchy [5], similar to that used by Plan 9 from Bell Labs [6]. 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 to gauge its effectiveness.

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.

We plan to 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 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.

2. Related Work

Much of the work on multilevel caches examines processor memory caches, which show great success in improving CPU performance [7]. 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 [8]. 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 [9] 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. 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 [10]. 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 [11]. 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. 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 [12]. 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 [5]. 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, as part of his research into ‘‘disconnected AFS,’’ implements a ‘‘fetch-only’’ mode for use when a disconnected client is connected to the network [13]. In this mode, the client fetches data not found in the local cache, but defers write requests, acting as though the client were disconnected from the network. Thus, files are saved in the local cache but are not passed to the principal servers. Fetch-only mode allows the user of a low-speed link to greatly increase performance by choosing when to use the limited bandwidth available to perform write operations.

Should automated consistency prove valuable in the low-speed network environment, Srinivasan and Mogul’s work with Spritely NFS [14, 15] and Transarc’s work with DFS [16] provide a useful analysis of consistency issues in a delayed-write caching system.

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.

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.`

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.

The second data set, which includes a portion of the first, spans a significantly 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. 320,725 distinct files are referenced 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 client's network address, the file's `FID` (unique identifier for a file), and the offset and length of the data being requested. 143,712 (35%) of the 6-day requests and 411,764 (36%) of the 24-day requests are `STOREDATA` requests.

Since the first two data sets consist only of file system requests received by the AFS servers, requests that were satisfied in the clients' caches do not appear in the data sets. The implication is that simulation results based on these data sets are accurate only when simulated client cache sizes are greater than the actual client cache sizes. Our clients typically cache 500 to 2,000 data chunks; results from simulating client cache sizes greater than 2,000 chunks are believed accurate.

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`. Thus, references to these files do not appear in the traces.

The fourth data set was collected by the creators of the experimental file system, xFS [10], to evaluate their system through simulation. Blaze's `rpcspy` program [17] 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.

Since `close` system calls do not appear in NFS network traffic, heuristics were used by Dahlin et al. (in a post-processing step) to add them. The `close` calls are necessary to reflect the semantics of AFS, *i.e.*, store on close.

† 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 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 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 `FID`s, 4,250,065 `Block READ` requests, 6,864 `Close RW` requests, 97,729 `Close WRITE` requests and 882,625 `Block WRITE` 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 past work, we have performed 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 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.

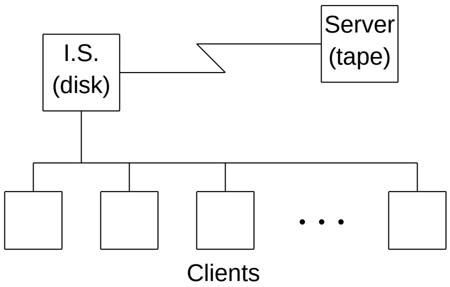

Figure 1. Topology of the intermediate server environment. File system requests from clients are passed to the intermediate server (I.S.). If the request cannot be satisfied from the intermediate server's cache, the request is forwarded to the primary server. Write requests cause invalidation of cached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

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 would be forced to flush its contents on occasion, the hit rates reported for these experiments are larger than can be achieved in reality.

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 following a series of writes, *e.g.*, at `close` time. Read requests are used instead of `FETCHDATA` requests, bacause `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 and allows us to study alternative client cache replacement policies (although that is beyond the scope of this paper).

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). 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, a ''hit'' is logged for that client and the next trace record is processed. Otherwise, in the case of a `FETCHDATA` (read) request, a ''miss'' is recorded for the client, and the cache on the intermediate server is checked for the requested chunk.

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

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.

`STOREDATA` requests cause the requested chunk to be cached at the client issuing the request and at its 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.

Of course, writes must eventually be propagated to the servers. If a client attempts a write when the intermediate's cache is filled, 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 [18].

Baker *et al.* report that most files are quickly deleted or overwritten [19]; 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 interest.

5. Hit rate simulations

In environments where access to the intermediate server is substantially faster than to the primary servers, misses are assessed at the intermediate server for requests that access the primary servers. In the mass storage system we are modeling, write requests are considered cache hits at the intermediate server; they are considered satisfied once they are completed in the intermediate server's cache, because the client that issued the request is not forced to wait while data are written to mass storage.

Some write requests can lead to a cache miss: when overwriting a partial chunk, the client first reads the complete chunk. If the read request cannot be satisfied out of the cache, a miss is assessed at this point. Ideally, the intermediate server could cache partially written chunks, eliminating the need to perform the read, but we do not assume this optimization.

Read requests that miss both the client cache and at the intermediate server's cache are counted as misses, as all other requests can be handled without accessing mass storage.

These experiments use the four data sets to calculate the expected write hit rate at the intermediate server. Intermediate cache hit rate is calculated for varying intermediate and client cache sizes, with estimated results for our environment lying between 2,000 and 3,000 client cache chunks.

When write requests are counted as hits, the effectiveness of the intermediate server is quite high even when intermediate cache sizes are modest, as shown in Figures 3–6.

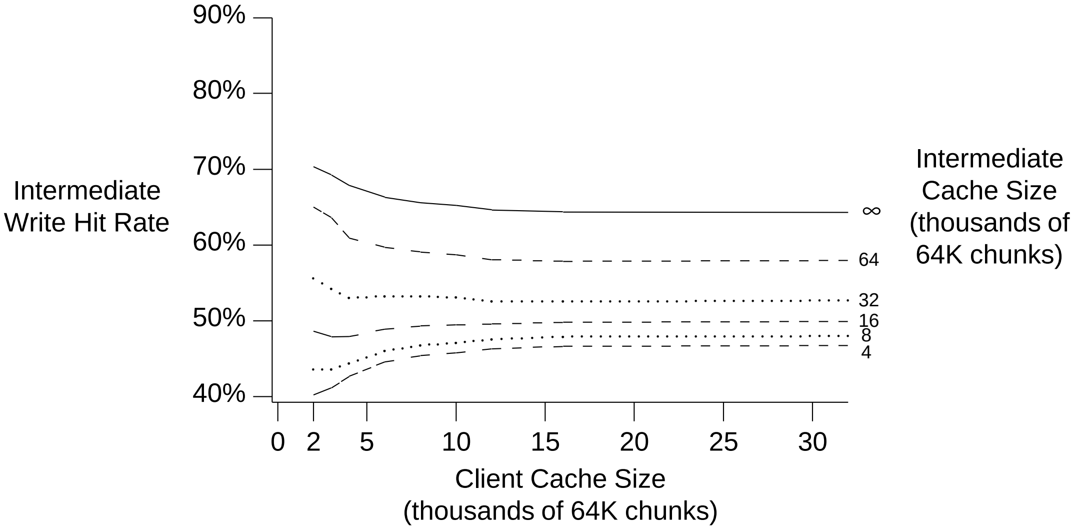

Figure 3. Client cache size *vs*. intermediate cache hit rate with the 6-day data. In this and the following three figures, we see how the client cache size (measured in thousands of 64K chunks) affects hit rate at the intermediate server. In these examples, the hit rate is defined to be (`FETCHDATA` request hits + `STOREDATA` requests) / total `FETCHDATA` and `STOREDATA` requests. The curves show variations in intermediate cache size. The intermediate cache hit rate can rise somewhat as client cache size grows, because the client caches satisfy more read requests.

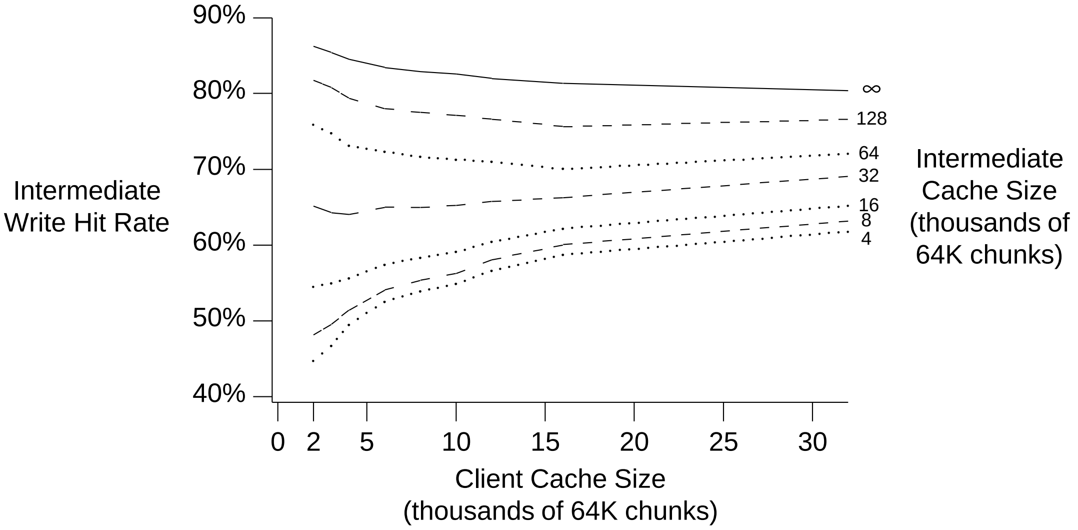

Figure 4. Client cache size *vs*. intermediate cache hit rate with the 24-day data.

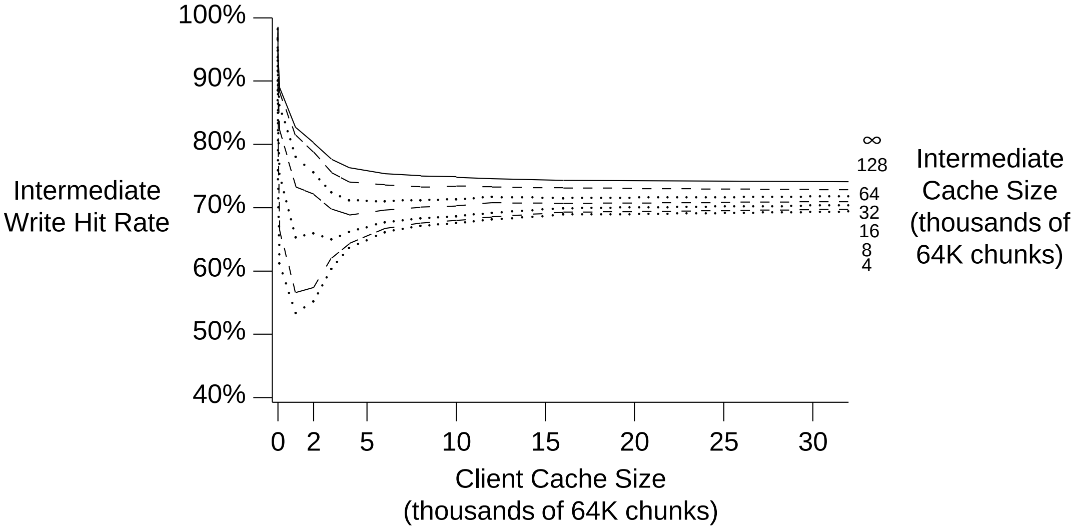

Figure 5. Client cache size *vs*. intermediate cache hit rate with the 2-month data.

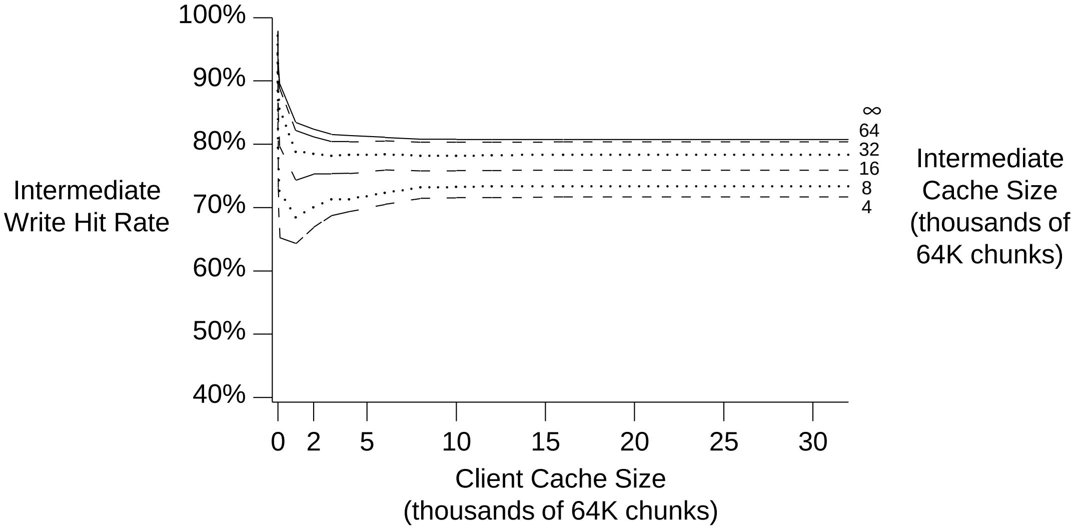

Figure 6. Client cache size *vs*. intermediate cache hit rate with the xFS data.

The increase in intermediate cache write hit rate as the client cache grows can be attributed to the client cache handling a greater number of read requests, which increases the relative frequency of write requests processed by the intermediate server.

5.1. Client requests to mass storage

This section shows alternative views of the intermediate server's ability to satisfy requests that were not satisfied by the client cache. For these experiments, the client caches are fixed at 3,000 chunks.

Figure 7 shows the effectiveness of various sizes of intermediate server caches using the 6-day data.

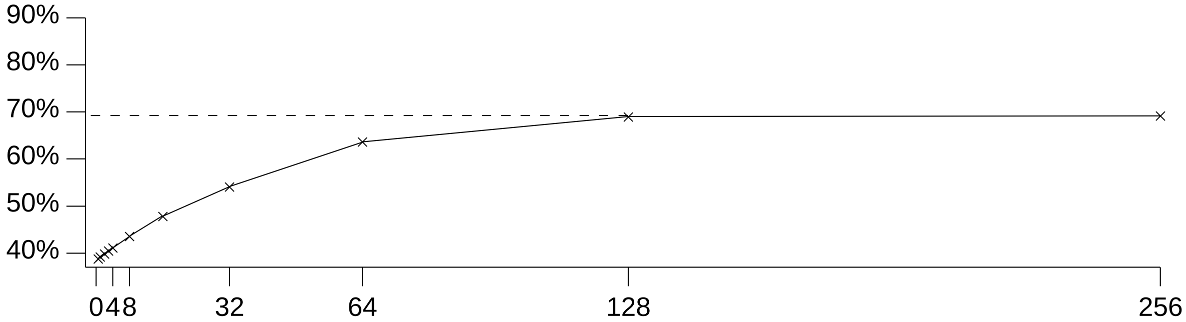

Figure 7. Percentage of requests satisfied by the intermediate server *vs.* intermediate cache size with the 6-day data. This graph and the following three graphs show how intermediate cache size, measured in thousands of chunks, influences the effectiveness with which the intermediate server handles requests not satisfied by the clients' caches. (All clients are assumed to have 3,000-chunk caches.) The dashed line at 69.3% shows the simulated effectiveness of an infinite intermediate server cache.

Intermediate server effectiveness, shown in Figure 8, is higher in the 24-day data, due in part to warm-up time for the intermediate server cache. An infinite intermediate server cache would satisfy 85.4% of all client requests, as represented by the dashed line.

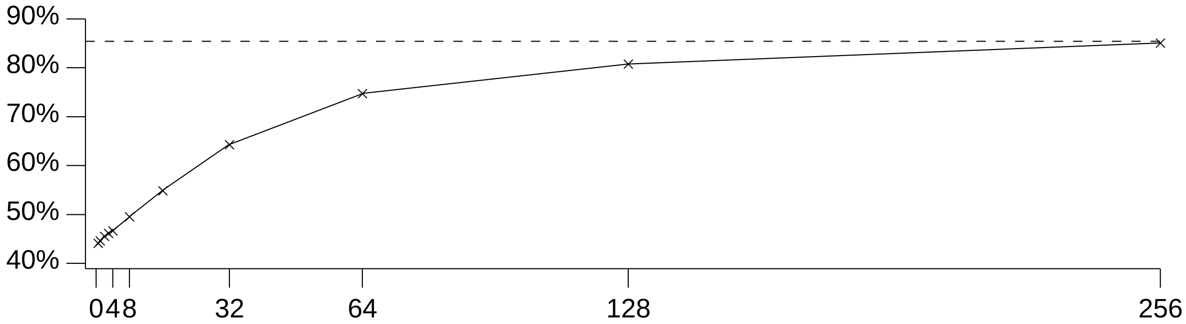

Figure 8. Percentage of requests satisfied by the intermediate server *vs.* intermediate cache size with the 24-day data. The dashed line at 85.4% shows the theoretical effectiveness of an infinite intermediate server cache.

And similarly, for the 2-month data:

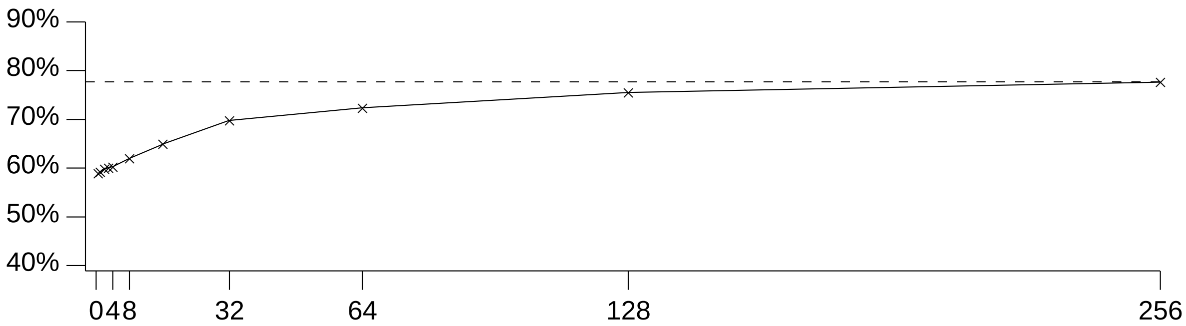

Figure 9. Percentage of requests satisfied by the intermediate server *vs.* intermediate cache size with the 2-month data. The dashed line at 77.6% shows the theoretical effectiveness of an infinite intermediate server cache.

And the xFS data:

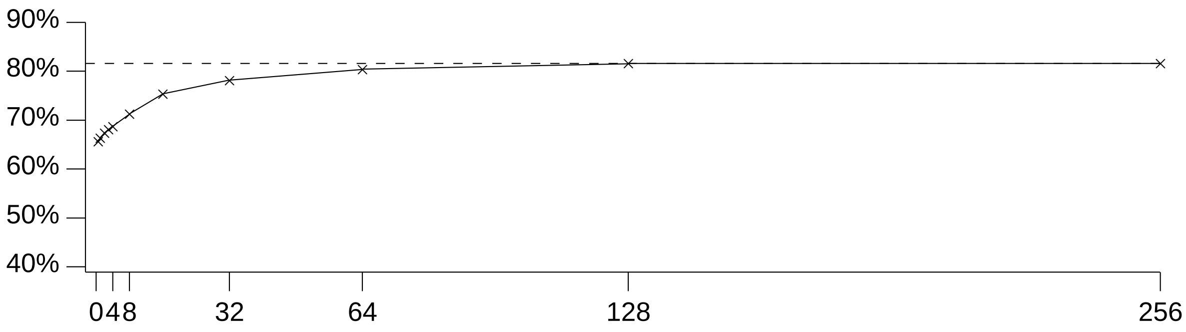

Figure 10. Percentage of requests satisfied by the intermediate server *vs.* intermediate cache size with the 2-month data. The dashed line at 81.5% shows the theoretical effectiveness of an infinite intermediate server cache.

The graphs in Figure 11 display in detail the reduction in requests to mass storage for the 24-day data when an intermediate server with 128,000 chunks (8 GB) in its cache is interposed. 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. With an intermediate cache size of 128,000 chunks, requests handled by the server are reduced by 80.8%, from 953,661 to 183,324. An intermediate cache size of 256,000 chunks decreases requests to mass storage by 85.1%.

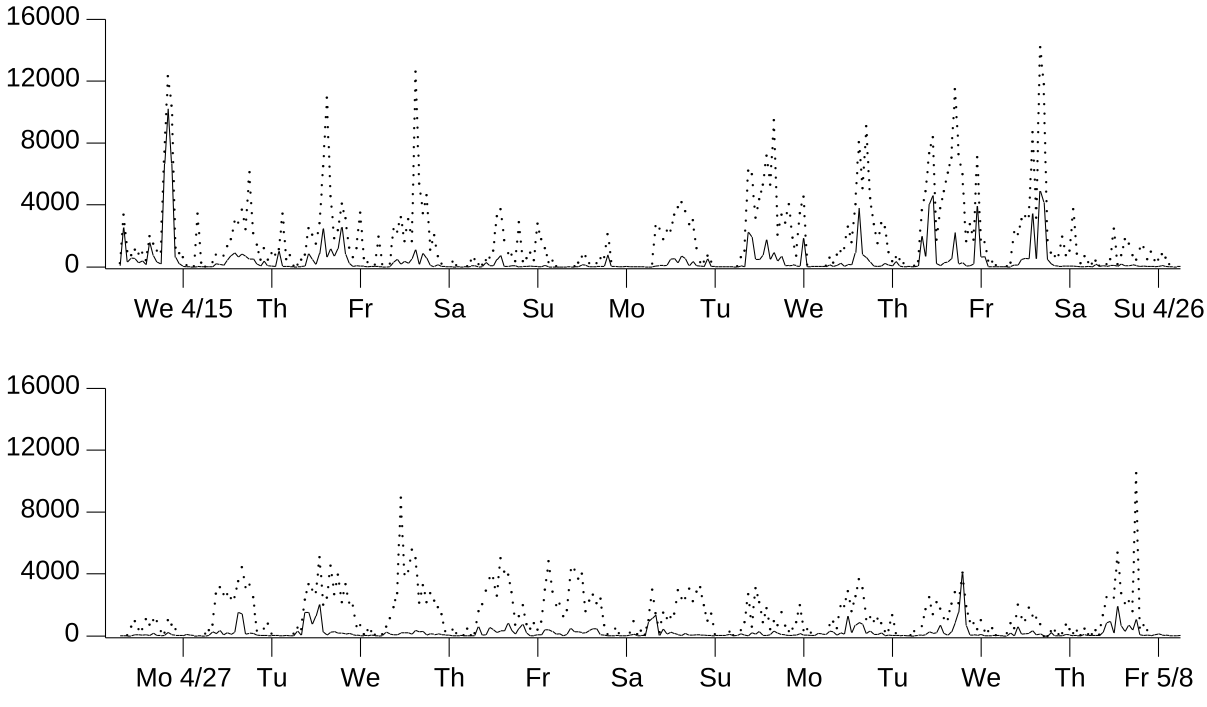

Figure 11. Slow storage request load reduction with the 24-day data (128,000-chunk intermediate cache). This graph shows the effect of an intermediate server with a 128,000-chunk cache has on the number of requests that must be satisfied from mass storage. Load reduction is significant over most of the interval. The dotted line shows the number of requests that must be 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. Request counts are based on 1-hour intervals.

Graphs based on the other data sets are similar to Figure 11 (though the request rates for the 2-month data set are lower, as it contains references from only 37 clients).

5.2. Discussion

Intermediate servers may prove useful in designing large distributed file systems based on slow mass storage systems. When intermediate servers are capable of satisfying client write requests without incurring the penalty of accessing slow storage, the effect is that the server is temporarily shielded from client write traffic and clients are protected from high delays. High intermediate cache hit rates coupled with high client cache hit rates would make such systems a practical and inexpensive way of combining the advantages of mass storage and distributed file systems.

6. Application to Low-Speed Networks

The write hit rate metric 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 AFS over low-speed lines, *e.g.,* home and mobile computers running SLIP or PPP over dialup links. Delayed-write caching for such machines may be particularly effective because write cost is high and because we expect data sharing to be low, 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 has been stored in the intermediate server's cache.

The intermediate server in this scenario thus becomes useful as a ''docking station'' to which mobile home clients can quickly off-load data that was written during disconnected operation [13]. Later, the intermediate server can convey the write requests 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 [19, 20]), which could substantially benefit low-speed network users. Writes that cannot be avoided could be delayed until a period when traffic over the slow link is low. Delayed-write caching may be useful as an option that can be manually turned on and off. 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.

A policy of periodically synchronizing written data may be sufficient for low-speed network use. The synchronization period would have to be fairly large or the benefit of delayed-write for this purpose is lost. Deferring writes for 12- or 24-hour periods—performing the writes during low-usage periods—could prove beneficial, as this could be synchronized with the daily backup schedule of the primary servers. Of course, users must also be given the option of syncing 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 would compromise this guarantee if applied to shared files. Where data sharing is high, protocol modifications are 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-through. 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 is similar to the Sprite [3], Spritely NFS [15], and DFS [16] 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) syncing process in the delayed-write scheme are not assessed as they are presumed to occur when the client is not otherwise in use, *i.e.,* not impeding user perceived performance.

For the simulations, we assume that each client machine in the 2-month data is actually an intermediate server and client (or, that they are clients performing 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 intermediate caches is 54%. The overall reduction is 32%. When cache size is increased to 16,000 chunks, 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 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 presented several problems, *e.g.*, whether traces in the traced environment reflect usage patterns in the simulated environment. Adding a delayed-write caching intermediate server on the client side of a low-speed link could greatly increase user-perceived file system performance, dramatically altering usage patterns and limiting the usefulness of traces collected in a write-through environment. Most CITI low-speed network users do little more with the file system over dialup lines than read mail or perform other small tasks, which would not provide interesting simulation material.

6.3. Conclusions

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 percentage 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 will play a major role in the realization of CITI's mass store-based file system. Intermediate servers are necessary due to the high seek times incurred in accessing mass storage devices. Our simulations of intermediate servers suggest they may also be sufficient for this purpose, *i.e.,* they can greatly reduce the number of requests that must be satisfied from the mass store. We introduce 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 the ability to control and reduce traffic over the link important. 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.

Finally, 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 as well as studying caching algorithms that have been suggested by others. Willick [21] and Korner [22] investigate alternatives to LRU caching for a distributed file system environment. 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 [23]. 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 *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 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 (February, 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 (February, 1988).
4. D. Muntz and P. Honeyman, ''Multi-level Caches in Distributed File Systems Revisited,'' in preparation (1994).
5. C.J. Antonelli and P. Honeyman, ''Integrating Mass Storage and File Systems,'' *Proc. 12th IEEE Symposium on Mass Storage Systems*, Monterey (April, 1993).
6. R. Pike, D. Presotto, K. Thompson, and H. Trickey, ''Plan 9 from Bell Labs,'' in *UKUUG Proceedings of the Summer 1990 Conference*, London, England (July, 1990).
7. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann Publishers, Inc., Palo Alto (1990).
8. H. Bugge, E. Kristiansen, and B. Bakka, ''Trace-driven Simulations for a Two-level Cache Design in Open Bus Systems,'' pp. 250–259 in *17th Annual International Symposium on Computer Architecture*, Seattle (May, 1990).
9. M. Blaze and R. Alonso, ''Dynamic Hierarchical Caching in Large-Scale Distributed File Systems,'' in *Proc. 12th Intl. Conference on Distributed Computing Systems*, Yokohama, Japan (June, 1992).
10. M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson, ''A Quantitative Analysis of Cache Policies for Scalable Network File Systems,'' *Proc. SIGMETRICS 94*, Nashville, pp. 150–160 (May, 1994).
11. D. Makaroff and D. Eager, ''Disk Cache Performance for Distributed Systems,'' pp. 212–219 in *Proc. 10th Intl. Conference on Distributed Computing Systems*, Paris, France (May, 1990).
12. E. Miller and R. Katz, ''An Analysis of File Migration in a UNIX Supercomputing Environment,'' pp. 421–433 in *Winter 1993 USENIX Conference Proceedings*, San Diego (January, 1993).
13. L. Huston and P. Honeyman, ''Disconnected Operation for AFS,'' pp. 1–10 in *Mobile and Location-Independent USENIX Conference Proceedings*, Cambridge (August, 1993).
14. V. Srinivasan and J. Mogul, ''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols,'' Research Report 89/5, DEC-WRL, Palo Alto (May, 1989).
15. V. Srinivasan and J. Mogul, ''Spritely NFS: Experiments with Cache-Consistency Protocols,'' in *Proc. of the 12th ACM Symp. on Operating System Principles*, Litchfield Park (December, 1989).
16. 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 *Summer 1990 USENIX Conference Proceedings*, Anaheim (June, 1990).
17. M. Blaze, ''NFS Tracing By Passive Network Monitoring,'' pp. 333–343 in *Winter 1992 USENIX Conference Proceedings*, San Francisco.
18. M. Kazar, ''Synchronization and Caching Issues in the Andrew File System,'' pp. 28–36 in *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 1988).
19. M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout, ''Measurements of a Distributed File System,'' in *Proc. of the 13th ACM Symp. on Operating System Principles*, Pacific Grove (October, 1991).

20. 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. of the 10th ACM Symp. on Operating System Principles,* Orcas Island (December, 1985).

21. D. Willick, D. Eager, and R. Bunt, ‘‘Disk Cache Replacement Policies for Network Fileservers,’’ pp. 2–11 in *Proc. 13th Intl. Conference on Distributed Computing Systems*, Pittsburgh (May, 1993).

22. K. Korner, ‘‘Intelligent Caching for Remote File Service,’’ pp. 220–226 in *Proc. 10th Intl. Conference on Distributed Computing Systems,* Paris, France (May, 1990).

23. S. Maffeis, ‘‘Cache Management Algorithms for Flexible Filesystems,’’ *ACM SIGMETRICS Performance Evaluation Review* **21**(2) (December, 1993).