DYNAMIC HIERARCHICAL CACHING IN LARGE-SCALE DISTRIBUTED FILE SYSTEMS

Matt Blaze
Rafael Alonso

CS-TR-353-91

October 1991

Dynamic Hierarchical Caching in Large-Scale Distributed File Systems

Matt Blaze
Rafael Alonso

Princeton University

ABSTRACT

Most Distributed File Systems (DFSs) are based on a flat client-server model in which each client interacts directly with the file server for all file operations. While this model works well for relatively small systems in which the file server has adequate capacity for all its clients, it does not scale to large numbers of clients or systems in which the clients are connected to the server through low-bandwidth links. Server traffic can be reduced substantially if clients keep even a modest-sized cache of previously read files. Intuitively, the benefits of caching can be increased by organizing clients into a hierarchy, in which only a small number of machines communicate directly with the file server, providing intermediate caching services to machines below them in the hierarchy. While this potentially reduces server traffic for widely shared files, it can introduce a significant delay for clients low in the hierarchy for access to files with a low degree of sharing.

This paper describes a simple method for constructing dynamic hierarchies on a file-by-file basis. The results of a trace-driven simulation of a dynamic hierarchical filesystem are presented, yielding a reduction in server traffic of a factor of more than two for shared files compared with a flat scheme and without a large increase in client access time. An algorithm to maintain cache consistency with low overhead by detecting missed cache invalidation messages is given.

1. Introduction

Distributed file systems (DFSs), such as NFS [1] and AFS [2], are widely accepted as a convenient mechanism for sharing and distributing files among small- and medium- size groups of computers. Although the benefits of file system semantics could extend equally well to larger systems, consisting of many thousands of machines spread over a wide geographic area, current systems do not scale up well enough to make such systems practical. Although a number of commercial and experimental systems do aim for various kinds of scalability, there is no system that would support, say, 100,000 machines located on several continents, all mounting a common /usr/bin directory to which software may be added or changed from time to time. Such large scale systems, to the extent that they exist at all, rely on file replication protocols (such as *ftp, rdist,* and even *mail)* that may be less convenient, more difficult to administer, or less reliable than file system semantics. As networked computing systems become more prevalent, the demand for highly scalable file systems that are flexible, transparent, and easy to administer can be expected to grow as well.

A DFS is scalable to the extent that client activity (and existence) is hidden from the server. Client caches can be quite effective in this regard, and one way to achieve greater scale is to make better use of remotely cached data. Intuitively, the total client cache miss rate would seem to be a lower bound on server traffic. If clients can make use of data in other client caches, however, it may be possible to avoid client-server interactions for cache misses on shared files.

Section 2 of this paper reviews briefly the trade-offs and file access patterns that influence DFS caching performance and then surveys the performance of a number of different caching schemes, both "flat" and hierarchical. In Section 3 we describe a simple scheme for building dynamic client hierarchies on a file by file basis, and give the results of a trace-driven simulation that suggests that such a scheme can reduce server traffic by a factor of more than two, without a large degradation of client performance. Finally, in Section 4, we discuss maintaining dynamic hierarchies in the presence of network failure.

2. Scalability and Caching

2.1. Behavior of Shared Files

The effectiveness of a caching strategy can be measured in terms of either its impact on the client (how long does it take to read a file) or on the server (how well is client activity hidden from the server). For scalability, we are more concerned with the latter, since it is the server's ability to process client requests that has the potential to cause a bottleneck. Obviously, the file access characteristics of a system have a large impact on the success of a caching strategy. In a previous paper [3], we examined a trace of workstation file system activity collected at DEC-SRC and used this trace data to simulate the impact of various flat caching schemes. The data consisted of all filesystem activity of 112 Firefly workstations over a 5-day period. For simplicity, due to the sheer volume of trace data, we examined only the 623,383 open for read, open for write, and unlink operations. Since the vast majority of file accesses in the trace are sequential and of the entire file (as has been generally observed to be true of Unix file access), this did not affect our results significantly. Compared with other traces analyzed in the literature [5][6][7], the DEC-SRC data appears to be typical of other Unix system traces, at least with regard to the properties that have been measured in these other traces.

An important observation from this trace is that as a file is opened for reading more often, it very quickly becomes unlikely that the next operation on that file will be write or unlink. This suggests that the probability of cached data being invalid is low, and that caching reduces server traffic substantially. A trace driven simulation showed that even fairly small client caches reduce server traffic by 80 to 90 percent compared with an uncached scheme. Also significant is the observation that write traffic is concentrated on unshared files (files used at only one machine before being deleted or overwritten), while shared files still make up a large proportion of read traffic. See Figure 1.

Clients Previously Reading File	Reads		Writes/Unlinks	
	Number	Percent of total	Number	Percent of total
0 or 1	207558	35.64	103339	98.77
2	20593	3.54	923	0.88
3	9608	1.65	176	0.17
4	5681	0.98	70	0.07
5	4604	0.79	33	0.03
> 5	334256	57.40	90	0.09

Figure 1 - Number of Clients Reading File Prior to Each Read and Write

2.2. Client- vs. Server- Driven Cache Invalidation

Usually, the semantics of a file system require that cached copies be in some sense current and that files overwritten or deleted at the server be propagated to or invalidated at the cached clients. The invalidation policy used has a strong impact on server load.

There are two basic approaches to maintaining client cache consistency. The first is for the client to check with the server before each read from the cache. We call this "client-driven cache invalidation", and it is the approach taken by NFS. Clearly, the client does not hide its activity from the server, but there is

still some benefit, since the server need only confirm the validity of the data for each cache hit. This is only useful if the cost of verifying the unchanged status of a file and transmitting that information is substantially lower than the cost of actually reading and transmitting the data, and it does not scale up to large amounts of client activity in any case.

The other approach, "server-driven invalidation," is for the server to maintain a list of clients with cached copies of each file. When a remotely cached file is overwritten or deleted, each client with a copy is notified to flush the data from its cache (or, in some cases, the clients are sent a copy of the new data). The AFS "callback" mechanism is based on this approach. If files change rarely (as has been observed in practice), this approach reduces server traffic substantially. While this scales well to large amounts of client activity, it does not scale to large numbers of clients, since the server must maintain a record for each client that has a copy of each file.

Server-driven invalidation can be improved slightly, in terms of the number of clients the server must track and the number of messages it must transmit, by bounding the length of time the server guarantees it will provide invalidation messages. The server should select an "expire" time that occurs just before it expects the file to actually change. We call this "predictive" server-driven invalidation, and it is similar in concept to Gray and Cheriton's "leases" [4]. We simulated a predictive scheme, using the DEC-SRC data, in [3].

Server-driven invalidation does not prevent inconsistency if a server or network failure prevents the client from receiving an invalidation message. In Section 4 of this paper, we discuss algorithms for detecting lost invalidation messages.

2.3. Limiting Server State Information

Since server-driven invalidation requires the server to maintain state information proportional to the number of clients with cached copies, it is not at all obvious that the performance benefits of such a scheme justify its use in a large scale system. Clearly, it is impractical for a server to maintain a list of 10,000 clients with copies of a widely used file. On the other hand, those same 10,000 clients would quickly overwhelm the server if client-driven invalidation were used. One solution, used by AFS, is for the server to invalidate older clients if too many new clients cache a particular file. This still puts a limit on the scalability of such systems, however, since the system degenerates into a client-driven invalidation scheme for clients beyond the number the server is willing to serve.

Another approach, also employed by AFS, is to identify by hand immutable files (and hierarchies) and not maintain server records for clients caching them. While this is a good solution for files that really never do change, there still may be files used by a large number of clients but which may need changes propagated to clients with ordinary filesystem semantics. The utility of this approach is also limited by the ability to identify the immutable files, which may not be obvious.

Clearly, neither the client- nor the server- driven invalidation schemes scale for large numbers of clients reading mutable files. Organizing clients into a hierarchy, such that only a limited (and manageable) number of clients communicate directly with the server, would seem to be a promising solution. Invalidation messages propagate from the server down to the clients, and file accesses not in the client cache but in a cache higher in the tree are handled without server intervention. Several systems[2][8] allow some use of hierarchies, set up statically based on network topology. These static hierarchies are particularly useful for directories of immutable "system" files, such as /bin.

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

The benefits of an intermediate cache that handles all file requests are surprisingly small, according to one experiment. Muntz and Honeyman [9] used the DEC-SRC trace data to simulate a two level cache hierarchy. All 112 clients interacted with an intermediate infinite cache server instead of the actual server.

While this did cause a small reduction in server traffic, from the client's perspective the intermediate cache was rarely used, even when the client cache was small. Depending on the size of the client caches, the hit rate at the intermediate cache was between 7% and 70%, falling off very rapidly (to about 10%-20%) for even small client cache sizes. So although there was a modest benefit to the server, the intermediate cache actually introduced a substantial delay for the clients, since most requests not satisfied by the client's own cache were not in the intermediate cache either, and had to be fetched from the file server anyway. We found similar results when restricting ourselves to the open and unlink operations from the same data. (A more encouraging result is obtained by eliminating home directory accesses from the intermediate cache, yielding an intermediate server hit rate of above 33%). Based on these results, a multi-layer hierarchy of finite caches used for all client file accesses could be expected to yield an even smaller reduction in server load at an even higher cost to the clients.

3. Dynamic Hierarchies

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

A measure of the potential for a multi-level caching scheme is the proportion of client cache misses that are for files that exist in another client's cache. We simulated a simple flat caching scheme with the DEC-SRC trace. Each client maintained a fixed sized (in terms of the number of files) cache, using server-driven invalidation and an LRU replacement policy. Each client cached any file it read or wrote. (Caching written as well as read files is an important optimization, cutting the miss rate in half). Access to non-shared (or not widely shared) hierarchies, such as /tmp and user's home directories, were considered to be on a local disk and not counted, though such files did occupy space in the client cache. With warm client caches (simulated by eliminating the first trace-day from the statistics), about 57 to 67 percent of client cache misses were of files that existed in another cache. Surprisingly, this was fairly insensitive to the actual size of the client cache (we simulated client caches that held between 32 and infinitely many files). Even for a cold cache startup, the proportion remained above 50 percent. See Figure 2.

Cache Size (in files)	Percent of misses in other client caches	
	(warm start)	(cold start)
32	64.83	60.30
64	67.66	61.08
128	64.81	56.76
192	64.31	55.74
256	63.86	55.10
320	61.25	53.44
384	59.34	52.53
448	57.61	51.84
512	57.46	51.85
∞	60.86	52.75

Figure 2 - Cache Misses of Files Active in Other Client Caches

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

The problem, of course, is that although there may be a high probability that a file exists in another cache, the client must determine where it is. Clearly, broadcasting a request to all other clients does not scale. Organizing the clients into a static hierarchy, with clients "nearest" the server either relaying requests to the server or serving them out of their own caches, will suffer the same (or worse) performance problems as the infinite-intermediate server discussed in the previous section, and the question of how to

best organize such a hierarchy is not at all obvious.

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

We propose a simple scheme for maintaining a dynamic hierarchy. Each client maintains a conventional fixed-size LRU *file cache* of files it reads or writes. In addition, it maintains a separate *name cache* indexed by file name. The name cache contains a record for each file read giving the source of the file (the *parent,* which may be another client or the actual file server) and a list of up to Δ *children* that must be notified if the file is changed. The name cache must be large enough to accommodate at least the files in the file cache. The server also maintains a record of children with copies of each file that it must notify should the file change. When client reads a file, it looks for the file in the file cache; if the file is present, it is read in the usual manner (we assume whole-file caching). If the file is not in the file cache, the name cache is checked; if there is a record for the file in the name cache, the file is requested from the parent. If no record is found in the name cache, the file is requested from the server. The reply consists of either the file requested or a list of clients that already have cached copies. In the latter case, one is selected from the list (either randomly or according to some criteria such as proximity) and the request is repeated to that machine, which also will respond with either the file or a list of its children. The procedure is repeated until the file is finally received, at which time the client creates a name cache entry for the file with the address of the parent, if none existed already.

When either a client or server receives a request for a file, it first checks whether the requester is in the child list for that file. If not, and the number of children already on the list is equal to the predetermined parameter Δ, the list of children is sent to the requester. Otherwise, the requester is added to the child list for that file. If the requester was already on the child list or was just added to it, the file is sent to it. It is possible that the file is not in the local cache anymore, in which case it is fetched from the parent first, as described in the previous paragraph.

Writes are always done through the server, which sends invalidation messages to each client on its child list, who in turn send invalidation messages to each of their children for the file. Note that since the list of children is kept in the name cache, invalidation messages must be sent out to each child whenever a name cache entry is replaced.

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

We used the DEC-SRC data to simulate this scheme. Again, we simulated a single server for the shared hierarchies, and assumed that each client had another server for more local files such as /tmp and home directories, although these files did occupy client cache space. We simulated a logically connected network in which all clients can communicate at equal cost, and clients used a uniformly distributed random function for selection of a parent when re-directed by the server. The performance of the system is influenced by two major parameters: the client file cache size n and the maximum degree Δ. We measured values of n from 64 files to infinite, and values of Δ from 2 to 112. (Since there were 112 machines in the trace data, Δ values of 112 or more are for all practical purposes infinite and equivalent to a conventional, flat scheme.) A third parameter, the client name cache size, turned out to be very small in practice, and sizes above about 100k bytes per client were for all practical purposes infinite in the simulation.

We measured a number of parameters, including the number of file transfers handled by the server and overall in the network, and the number of overhead messages (invalidation messages and messages redirecting clients to children) sent by the server and in the overall network. We would expect decreasing Δ to decrease the traffic at the server but raise it in the overall network, since some client requests must be routed through more than one downstream client to be satisfied. Server traffic is a measure of server cost, while overall network traffic can be viewed as a measure of client access time and processing cost. The

question is whether the increased client access cost is justified by the decrease in server traffic cost.

The simulation results suggest that dynamic hierarchies may work well in practice. For all cache sizes, decreasing the value of Δ (increasing the depth of the tree) greatly reduced the transfers at the server, always by at least a factor of two for $\Delta=2$. Although the overall number of transfers did increase, the increase was always smaller than the corresponding decrease in server transfers and was under 25% for $n \geq 128$ files. Figure 3 shows the number of file transfers at the server (solid curves) and in the network (dotted curves) for various values of n and Δ after one day of cache warm-up (the figures for cold start are similar but slightly higher). Note this graph decreases from an infinite Δ at the left, since infinite Δ is equivalent to a non-hierarchical, flat scheme.

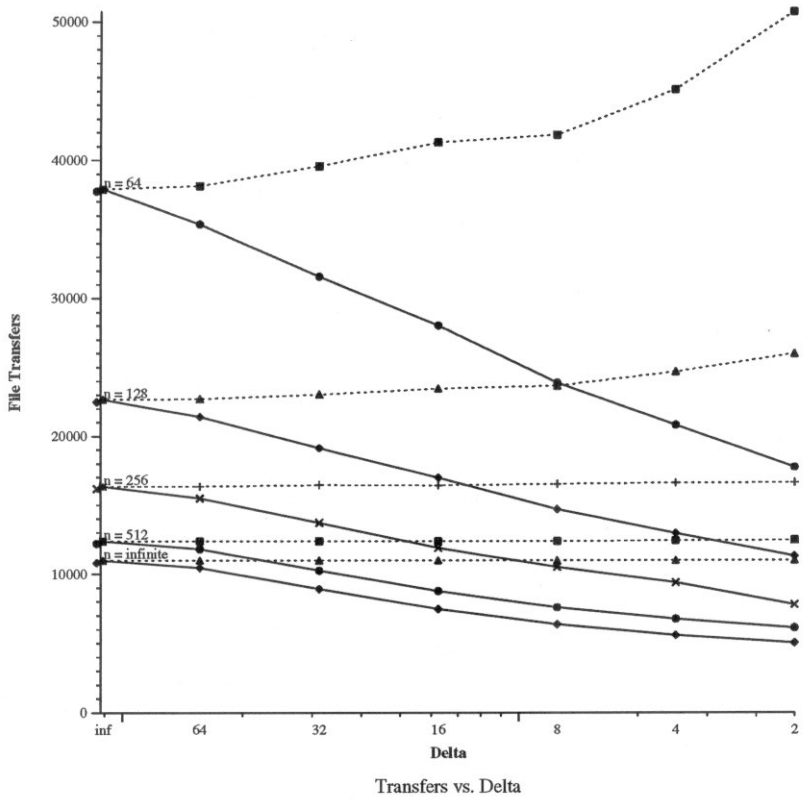


Figure 3 - File Transfers

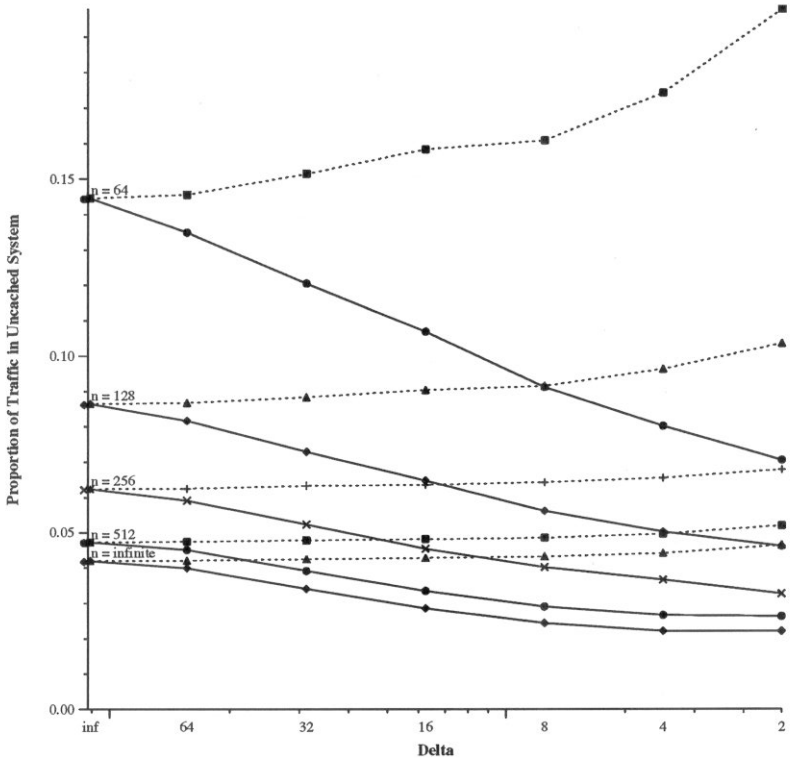
Decreasing Δ is not without cost to the server, however, since although the number of invalidation messages it must transmit in the event a file is overwritten decreases, it must send lists of children to clients it does not wish to service. The number of these overhead messages is not dependent on client cache size, but is dependent on Δ. Figure 4 gives a table of the number of overhead messages handled by the server and in the network overall for various values of Δ.

Based on the simulation, it would appear that the impact of the dynamic hierarchy depends on the relative cost of a file transfer compared with an invalidation or redirection message. This is influenced strongly by the underlying network topology and protocols, the maximum basic transfer unit, the sizes of the files transferred, and so on. Files transferred in our simulation averaged around 11k bytes. The non-communication costs (such as reading a file from disk as opposed to checking an entry in an in-core table) come into play as well, although we do not consider them directly here. It is probably reasonable to assume that the cost of transferring a file greatly exceeds the cost of the other messages; if this were not true client-driven invalidation, which is used by NFS, would carry no benefit. For a rough estimate of the expected actual performance of our dynamic hierarchy, Figure 5 compares the total volume of traffic for various values of n and Δ assuming that the cost of a file transfer is 15 times that of an overhead message. Traffic is represented in terms of the proportion of traffic that would occur if no caching were used. As

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

Figure 4 - Overhead Messages

before, the dotted curves represent overall network traffic and the solid curves represent server traffic. Note that the curves are substantially similar to Figure 3, which considers only file transfers.



Traffic, File Transfer Cost == 15x Message Cost

Figure 5 - Total Traffic in Dynamic Cache Hierarchy

4. Network Non-Connectivity and Failure

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

The first problem is simply that, depending on the underlying network, not every host can always communicate with every other host, even if one host (the file server) can communicate with all of them.

This is an issue since requests for files may be re-directed to other clients. If a client cannot communicate with any of the hosts a server directed it to, it must re-contact the server. At this point, the server can either replace one of the clients on the child list with the new client, or it can simply increase the number of clients it serves. If it chooses the former option, it must invalidate the old child first, which may trigger additional traffic when that client (or one of its children) reads the file again. If it chooses the latter option, it increases the size of its Δ. A third option is to maintain two Δ values, a soft one and a hard one. New clients are still re-directed when the value exceeds the soft Δ, but will be added to the child list if they fail to contact any of the existing children. New clients can be added under these circumstances until the hard Δ is exceeded, at which point old children are invalidated and replaced. The cost of doing this depends on the connectivity of the network, of course. In our simulation, we modeled only a logically fully-connected network.

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

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

A high degree of consistency can be maintained without resorting to client-driven invalidate, however. A client can maintain "time-bounded" Unix semantics by checking with the parent of a file read from the local cache if it has not otherwise communicated with the parent within some real-time bound, guaranteeing that cache reads are never more than this bound out of date. If it cannot reach the parent, the read fails.

Another metric of consistency is similar to the data base concept of one-copy serializability (1-CS). This requires simply that the global sequence of reads and writes be equivalent to some serial schedule on a single machine file system. This degree of consistency is provided automatically if flat server-driven invalidation is used and all reads and writes are through a single file server. If there is more than one file server, or the clients have a cache hierarchy, it is still fairly inexpensive to maintain 1-CS. Observe that it is sufficient to ensure that once data written by a client is read by a remote machine, future reads by the writer must be current. Upon writing a remotely readable file (one that is also mounted by another machine), the writer should verify its connection to the file's parent on subsequent reads from the cache. It needs to ensure that the data is up to date as of at least the time of the last write, so the parent, if it is not itself the server, must repeat the verification with its parent if it has not communicated with it since the write time, and so on up to the server. This requires a global clock, such as that described in [10]. A detailed proof is beyond the scope of this paper and is omitted.

It is potentially expensive to maintain 1-CS in a dynamic hierarchy, since each write can trigger a potentially large set of verification messages. In our trace, however, we found the actual level of traffic to be very small, adding less than 5% to the number of messages regardless of the client cache size or Δ.

5. Conclusions and Future Work

Our simulation results suggest that clients can effectively share cache data to reduce the load on the file server by about a factor of two. It is tempting to infer that this means that a dynamic hierarchy allows a file server to serve twice as many clients as it would otherwise be able to, or that file servers for dynamic hierarchies need have only half the processing power as those serving flat caching schemes. Obviously, the applicability of these results to scalable systems is highly dependent on the workload; our trace studied less than one week of activity on 112 clients. It would be desirable to conduct experiments on workloads of larger systems, over longer time periods, and in computing environments outside research laboratories. Intuition, however, suggests that as traces cover longer periods and more clients, the proportion of cache misses of files present in other caches will increase, making the effects of dynamic hierarchies even more

pronounced.

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

6. Acknowledgments

We are extremely grateful to Andy Hisgen at DEC SRC for making the trace data available to us. The following people at DEC SRC contributed to the file system tracing facility: Susan Owicki, B. Kumar, Jim Gettys, Deborah Hwang, and Andy Hisgen. The actual trace data was gathered by Andy Hisgen.

7. References

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

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

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

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

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

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

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

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

[9] Muntz, D. & Honeyman, P. "Multi-Level Caching in Distributed File Systems" *CITI-TR-91-3* University of Mich., 1991. (Expanded version to appear in *Proc. USENIX Winter Conference,* 1992).

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