Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet

Renu Tewari, Michael Dahlin, Harrick M. Vin and Jonathan S. Kay

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
E-mail: {tewari, dahlin, vin, jkay}@cs.utexas.edu

UTCS Technical Report: TR98-04

Abstract

In this paper, we examine several distributed caching strategies to improve the response time for accessing data over the Internet. By studying several Internet caches and workloads, we derive four basic design principles for large scale distributed caches: (1) minimize the number of hops to locate and access data, (2) do not slow down misses, (3) share data among many caches, and (4) cache data close to clients. Although these principles are simple, they are often violated in traditional cache hierarchies. Based on these principles, we have designed, simulated, and implemented two strategies for organizing distributed caches. First, we separate data paths from metadata paths and use location hints to allow caches to directly access remote copies of data. Second, we use two simple push caching algorithms to move data near clients. Together, these strategies provide response time speedups of 1.27 to 2.43 compared to a traditional three-level data cache hierarchy for a range of trace workloads and simulated environments.

1 Introduction

The growth of the Internet and the World Wide Web have enabled increasing numbers of users to access vast amounts of information stored at geographically distributed sites. Due to the non-uniformity of information access, however, popular web pages create “hot spots” of network and server load, and thereby significantly increase the latency for information access.

Large-scale distributed caches appear to provide an opportunity to combat this latency because they allow users to benefit from data fetched by other users [10, 18], and their distributed architectures allow clients to access nearby copies of data in the common case. Current web cache systems define a hierarchy of data caches in which data access proceeds as follows: a client sends a request to a cache, and if the cache contains the data requested by a client, it replies with the data. Otherwise, the cache may ask its neighbors for the data, but if none of the neighbors possess the data, then the cache sends its request to its parent. This process recursively continues up the hierarchy until the data is located or the root cache fetches the data from the server specified in the request. The caches then send the data down the hierarchy to the client, and each cache along the path caches the data. Unfortunately, these hierarchies of data caches often achieve modest hit rates [2, 10, 15, 18], can yield poor response times on a cache hit [26, 32], and can slow down cache misses.

In this paper, we first attempt to understand the factors that limit the performance of current web caching hierarchies through measurements of several caches on the Internet and analysis of several traces of web traffic. Based on these measurements, we derive four basic design principles for large-scale caches: (1) minimize the number of hops to locate and access data, (2) do not slow down misses, (3) share data among many caches, and (4) cache data close to clients. Although these principles may seem obvious in retrospect, we find that current cache architectures routinely violate them at a significant performance cost. Based on these principles, we have designed, simulated, and implemented two strategies for organizing large-scale, distributed caches.

Our first strategy addresses the first three principles. This strategy has three salient features. First, it separates data paths from metadata paths and maintains a hierarchy of metadata that tracks where copies of data are stored. Second, it maintains location hints so that caches can locate nearby copies of data without suffering network latencies. Third, it uses direct cache-to-cache data transfers to avoid store-and-forward delays. Thus, whereas a traditional cache hierarchy often requires both requests and responses to traverse several caches in the hierarchy, our architecture minimizes the number of hops through the hierarchy by using location hints to go directly to the nearest cached copy of data and by using cache-to-cache data transfers to send the data directly back. Whereas a traditional hierarchy may forward cache misses through several caches before going to the server, caches in our system use hints to detect misses locally and then proceed directly to the server. Furthermore, our design facilitates widespread sharing of data among large numbers of caches to provide good hit rates by: (i) compressing the size of hints so that hint caches can track the contents of many data caches, (ii) using a metadata distribution hierarchy to distribute hints efficiently, and (iii) dynamically configuring the hierarchy to automate the management of systems with many caches. A novel aspect of this strategy is that although it uses a hierarchy to distribute metadata, it only stores data at the leaves of the hierarchy, and it always transfers data directly between caches rather than through the metadata hierarchy.

Our second strategy addresses the fourth design principle by redistributing data across the caches in the system. It does this by "push caching" [20] data near clients that have not referenced the data but that are likely to reference it in the future. Thus, it reduces or eliminates the compulsory miss penalty normally experienced the first time a particular client accesses an object. We bound the performance achievable by such algorithms and then examine two simple and efficient algorithms for choosing what data to push. One algorithm pushes data that have been recently updated, and the other pushes data that are widely shared.

We evaluate our strategies using simulation studies of three large workloads that contain accesses by users of Digital Corporation's proxy server [9], UC Berkeley's Home-IP service [18], and Prodigy ISP's dial-up web browsing service. Our algorithms focus on *improving hit and miss response times* rather than trying to improve global hit and miss rates. Thus, we parameterize our results using estimates of Internet access times based on our own measurements and those in the literature [32]. For these traces and network configurations, we find that our hint architecture provides speedups of 1.27 to 2.30 compared to a standard, three-level cache hierarchy and that push caching improves performance by up to an additional factor of 1.25. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional hierarchy.

We have implemented a prototype by augmenting the widely-deployed Squid proxy cache [39]. It implements a hint distribution hierarchy, local hint caches, and cache-to-cache data transfers. A key insight gained in this implementation was an understanding of the desirability of storing hints as small, fixed-sized records. We currently are in the process of adding automatic configuration of the hint hierarchy and push caching to the prototype.

In summary, this paper makes three primary contributions. First, by evaluating traditional hierarchies, we devise several simple and general design principles for distributed caches on the Internet. Second, we propose and evaluate a novel architecture for separating data and metadata paths in large cache systems. Third, we devise and evaluate several simple algorithms that implement scalable, hierarchical push caching, and we provide an upper bound on the performance achievable by such algorithms in distributed cache systems.

In this study we focus on a collective cache environment where caches are willing to cooperate to some degree to provide improved performance for all of their clients. We believe our model will be interesting to system designers building large-scale, distributed cache infrastructures in a range of environments including network service providers, independent service providers, cache service providers [6, 29, 35, 39], collections of caches linked by formal service agreements [34], and large intranets. Also, once the trade-offs in these cooperative environments are understood, we plan to examine ways to apply some of these techniques in more competitive environments.

The rest of the paper is organized as follows. Section 2 evaluates the performance of traditional cache hierarchies and examines the characteristics of several large workloads and then derives a set of basic design principles for large-scale, distributed caches. Section 3 examines how to separate data and metadata paths in these caches to improve response times of requests. Section 4 discusses techniques for distributing data across caches to increase the frequency of hits to fast, nearby caches. Section 5 surveys related work, and Section 6 summarizes our conclusions and outlines areas for future research.

2 Evaluating traditional cache hierarchies

In this section, we evaluate the performance of traditional cache hierarchies using measurements of several caches on the Internet and trace-driven simulations, with the goal of understanding the factors that limit cache performance. Based on these measurements, we derive the four design principles for distributed caching on the Internet that are summarized in Table 1.

Design Principle	Strategy
Minimize the number of hops to locate and access data (Section 2.1)	Separate data and metadata, hints, and cache-to-cache transfer (Section 3)
Do not slow down misses (Section 2.2)	Location-hints (Section 3)
Share data among many caches (Section 2.2)	Separate data paths and metadata paths, location-hints (Section 3)
Cache data close to clients (Section 2.1)	Push caching (Section 4)

Table 1: Design principles and strategies for distributed caching in the Internet

2.1 Access costs in a traditional cache hierarchy

Traditional hierarchical cache architectures such as Harvest [5] or Squid [39] define parent-child relationships among caches. Each cache in the hierarchy is shared by a group of clients or a group of children caches. In such cache hierarchies, data access proceeds as follows: if the lowest-level cache contains the data requested by a client, it sends the data to the client. Otherwise, the cache asks each of its neighbors for the data. If none of the neighbors possess the data, then the cache sends a request to its parent. This process recursively continues up the hierarchy until the data is located or the root cache fetches the data from the server. The proxies then send the data down the hierarchy and each cache along the path caches the data.

Although hierarchies such as Harvest and Squid were designed under the assumption that caches could be layered without adding much delay [5], we hypothesize that two aspects of this architecture as applied to Internet caches can significantly limit performance. First, the cost of accessing a series of caches in the hierarchy adds significant "store-and-forward" delays to higher-level cache hits and to cache misses [28]. Second, when high-level caches service a large number of clients distributed over a large geographic region, the network delays between a client and a high-level cache may be large, which reduces the benefit of hits to far-away caches.

To understand these effects better, we use two sources of information. First, to understand the details of performance in a controlled environment, we construct a test hierarchy and examine it under a synthetic workload. Second, to understand how such systems perform in real hierarchies and under real workloads, we examine Rousskov's measurements of several Squid caches deployed at different levels of a hierarchy [32]. Although Squid supports the Internet Cache Protocol (ICP) to allow a cache to query its neighbors before sending a miss to a parent [40], we are interested in the best costs for traversing a hierarchy, so neither configuration we examine uses ICP.

Both analysis yield similar results. Both suggest that there is a significant per-hop cost for traversing multiple levels of cache. This yields the first design principle in Table 1: *minimize the number of hops to locate and access data*. Both also support the hypothesis that distant caches may be expensive to access even without this per-hop cost due to network limits. This yields the third design principle in Table 1: *cache data close to clients.* In what follows, we describe both the experiments.

2.1.1 Analysis of a testbed hierarchy

We constructed a testbed to examine the relationship among caches arranged in a large, three-level hierarchy in which the level-1 (L1) cache services a department, a level-2 (L2) cache services a state, and a level-3 (L3) cache services a large region. Table 2 details our testbed hierarchy. Although these caches are distributed across a large geographic region, they are relatively well connected, so some less ambitious hierarchies may have similar characteristics. Each cache in this setup runs version 1.1.17 of the Squid proxy cache. The client and L1 cache are connected by a switched 10Mbit/s Ethernet. All of the sites are connected to the Internet via 45Mbit/s T3 connections, and the backbone connections between all sites are at least T3.

For our experiments, we arranged for a specific level of the cache to contain an object of a specified size and for the other levels of the cache not to contain that object. We then used an instrumented client program to time how long it took to get that object from the hierarchy. We repeated this experiment 10 times for each configuration of the caches over the course of 3 hours during the late afternoon on several weekdays and discarded the high and low values observed. Each data point in the graphs in Figure 1 represents the mean of the remaining eight measurements.

Cache	Location	Machine
Client	CS department, UC Berkeley	166MHz Sun UltraSparcs
L1	CS department, UC Berkeley	166MHz Sun UltraSparcs
L2	CS department, UC San Diego	150MHz DEC 2000 Model 500
L3	CS department, UT Austin	166MHz Sun Ultra2
Server	CS department, Cornell University	DEC Alpha

Table 2: Testbed hierarchy setup

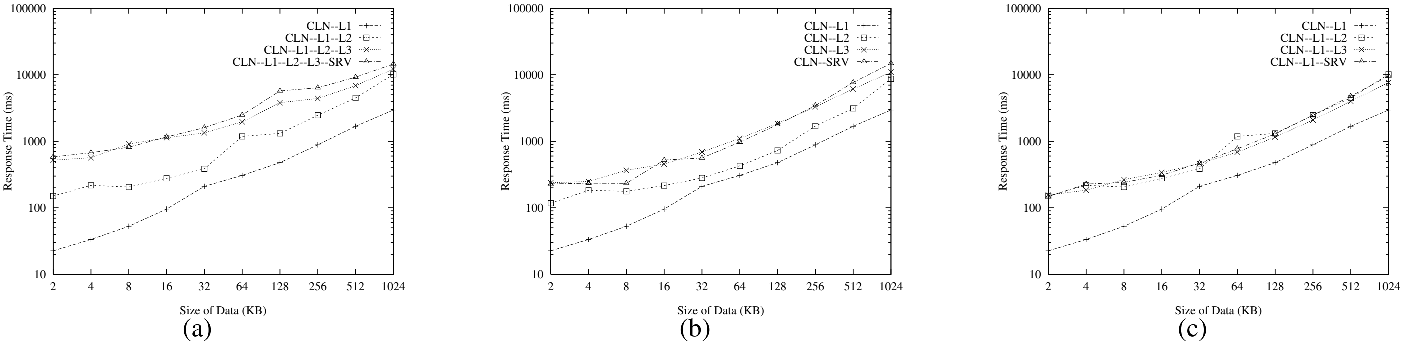


Figure 1: Measured access times in the testbed hierarchy for objects of various sizes: (a) objects accessed through the three-level cache hierarchy; (b) objects fetched directly from each cache and server; and (c) all requests go through the L1 proxy and then directly to the specified proxy or server.

Note that, in our experiments, the caches were idle other than our requests, which were made one at a time. If the caches were heavily loaded, queuing delays and implementation inefficiencies of the caches [26, 3] might significantly increase the per-hop costs we observe. Busy nodes would probably increase the importance of reducing the number of hops in a cache system.

Figure 1 summarizes the performance of the testbed hierarchy. Figure 1(a) shows the performance the system uses the standard three level data hierarchy. In contrast, Figure 1(b) shows the access time when the Berkeley client accesses the Berkeley, San Diego, and Austin caches *directly* by circumventing the hierarchy. Figure 1(c) shows the case when direct accesses must always go through the L1 cache such as when the L1 cache acts as a firewall for the clients. These measurements support and quantify the intuition that accessing a series of caches in a hierarchy incurs a significant cost. For instance, the difference between fetching an 8KB object from the Austin cache as part of a hierarchy compared to accessing it directly is 545 ms. Put another way, if the system could "magically" send requests directly to the correct level of the hierarchy and that level of the hierarchy send the data directly back to the client that needs it, a level-3 cache hit time could speed up by a factor of 2.5 for an 8KB object. In Section 3, we propose an architecture that approximates this ideal.

Figure 1(b) also indicates that even if a cache architecture were able to avoid the cost of multiple-hop store-and-forward, accessing distant caches is still more expensive than accessing nearby ones. This conclusion is not surprising; accessing a distant node on the Internet will increase latency due to increased routing distance and increased risk of encountering routing delays, and it will decrease bandwidth for analogous reasons. This experiment suggests that in addition to reducing the number of hops needed to access distant data, cache hierarchies should take action to access nearby data as often as possible. In Section 4, we propose algorithms that address this design principle.

2.1.2 Analysis of Squid hierarchies

Rousskov [32] has published detailed measurements and performance analysis of several Squid caches that are deployed on the Internet in the United States and Europe. We use that information to derive the estimates for the minimum and maximum access times to different cache levels summarized in Table 3.

	Client Connect		Disk		Proxy Reply		Total Hierarchical		Total Client Direct		Total via L1	
	min	max	min	max	min	max	min	max	min	max	min	max
Leaf	16ms	62ms	72ms	135ms	75ms	155ms	163ms	352ms	163ms	352ms	163ms	352 ms
Intermediate	50ms	550ms	60ms	950ms	70ms	1050ms	271ms	2767ms	180ms	2550ms	271ms	2767ms
Root	100ms	1200ms	100ms	650ms	120ms	1000ms	531ms	4667ms	320ms	2850ms	411ms	3067ms
Miss	550ms 3200ms						981ms	7217ms	550ms	3200ms	641ms	3417ms

Table 3: Summary of Squid cache hierarchy performance based on Rousskov's measurements.

We derive these estimates by computing the time that a leaf, intermediate, and root cache took to service a hit for a request sent from the next level down in the hierarchy. Rousskov breaks down hit response time into three components: "Client connect," which is the time from when the cache's accept() system call returns until the cache has a parsable HTTP request, "Disk," which is the time it takes to swap in data from the disk, and "Proxy reply," which is the time it takes to send the data back on the network. Since Rousskov gathered the measurements at the proxy caches, we must estimate the miss time by looking at his measurements of how long the top-level proxy spent waiting to connect to and receive data from miss servers. Rousskov measured these values over a 24-hour period for each cache and publishes the median value during each 20-minute period. Table 3 shows the minimum and maximum values seen for these 20-minute medians between the peak load hours of 8AM to 5PM.

We estimate bounds on the time to access an object in each level of a hierarchy by adding the minimum or maximum client connection and proxy reply times for all levels that must be traversed and then adding the disk response time for the highest level cache accessed. We estimate bounds on the total response time to directly access some level of the cache by adding the minimum or maximum client connection time, disk response time, and proxy reply time. Note that this calculation does not account for possible pipelining between the disk response time and the proxy reply time.

These results support the same general conclusions as the measurements of our testbed: hops are expensive, and accessing far away caches is expensive. These data suggest cache systems may pay a particularly high penalty for accessing distant caches during periods of high load. For example, directly accessing a leaf cache during periods of low load costs 163 ms which is twice as fast as the 320 ms cost of directly accessing a top level cache. However, the top level cache experienced a 20-minute period during which the median access time was 2850 ms, whereas the worst 20-minute median access time for the leaf cache was 352 ms. Our interpretation is that although accessing distant caches can be tolerable in the best case, caching data near clients may be an important technique for insulating clients from periods of poor performance.

2.2 Workload characteristics

In this subsection we examine how characteristics of web workloads stress different aspects of large cache systems in order to understand what techniques will be important for improving performance in such systems. This analysis yields the second and third principles listed in Table 1:

- Even an ideal cache will have a significant number of compulsory misses (misses to objects never referenced before in that cache) and communication misses (misses to objects that have changed since they were last referenced.) Thus, *cache systems should not slow down misses*.
- Cache systems should facilitate strategies that *share data among many clients* to reduce compulsory misses in low-level caches.

Analysis of these workloads leads us to make two other observations. First, we expect capacity misses (misses to objects that have been replaced due to limited cache capacity) to be a secondary consideration for large-scale cache architectures because

Trace	# of Clients	# of Accesses	# of Distinct URLs	Dates	# of Days	Client ID
DEC [9]	16,660	22.1 million	4.15 million	Sep. 1 – Sep. 21, 1996	21	preserved
Berkeley [17]	8,372	8.8 million	1.8 million	Nov. 1 – Nov. 19, 1996	19	preserved
Prodigy	35,354	4.2 million	1.2 million	Jan. 13 – Jan 15, 1998	3	dynamic IP

Table 4: Characteristics of trace workloads.

it is economical to build shared caches with small numbers of capacity misses. If more aggressive techniques for using cache space are used (for example, push caching), capacity may again be a significant consideration. Second, cache architectures should be scalable to allow large numbers of clients to share the cache to achieve good global hit rates. Scalability means that caches should avoid centralized bottlenecks and that the distance between clients and caches should not increase as system size increases.

2.2.1 Methodology

The simulation experiments in this study use three large-scale traces taken at proxies serving thousands of clients. Key parameters of these traces are summarized in Table 4. All three traces track the activities of thousands of clients over several days. For the DEC and Berkeley traces, each client has a unique ID throughout the trace; for the Prodigy trace, clients are dynamically bound to IDs when they log onto the system. In analyzing the cache behavior of these traces, we use the first two days of each trace to warm our caches before beginning to gather statistics. Due to time constraints and memory limitations of our simulation machines, the results in Section 4 use only the first seven days of the DEC and Berkeley traces.

To determine when objects are modified and should thus no longer be provided from the cache, we use the last-modified-time information provided in the DEC traces. When the last modified times are not provided, we infer modifications from document sizes and return values to if-modified-since requests. Both of these strategies will miss some of the modifications in these traces.

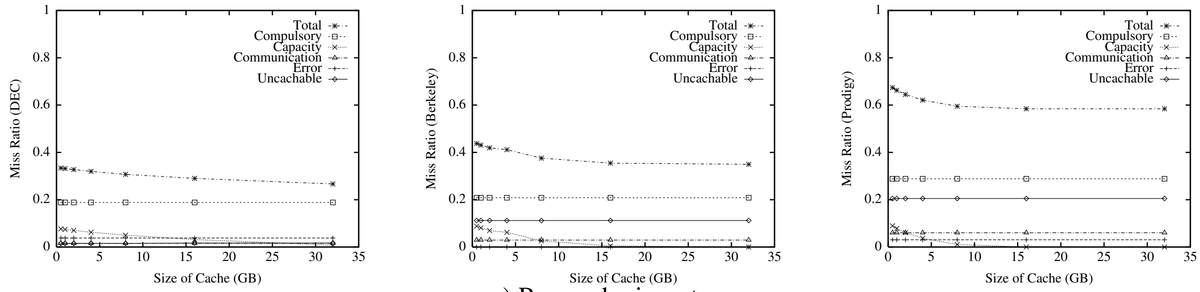
Current web cache implementations generally provide weak cache consistency via ad hoc consistency algorithms. For example, current Squid caches discard any data older than two days. In our simulations, we assume that the system approximates strong cache consistency by invalidating all cached copies whenever data changes. We do this for two reasons. First, techniques for approximating or providing strong cache consistency in this environment are improving [21, 25, 41], so we expect this assumption to be a good reflection of achievable future cache technology. Second, weak cache consistency distorts cache performance either by increasing apparent hit rates by counting “hits” to stale data or by reducing apparent hit rates by discarding perfectly good data from caches. In either case, this would add a potentially significant source of “noise” to our results.

These traces have two primary limitations that will affect our results. First, although we use traces with thousands of clients, the population of clients in the traces is still a small fraction of the web population. Gribble and Brewer’s analysis of the Berkeley trace indicates that achievable hit rates will improve as more clients are included in a cache system, but they caution that it is difficult to know at what point beyond the population sizes studied that such a trend will stop [18]. Duska et. al conclude that as the rate of requests a cache system handles increases, its achievable hit rate will also increase [10].

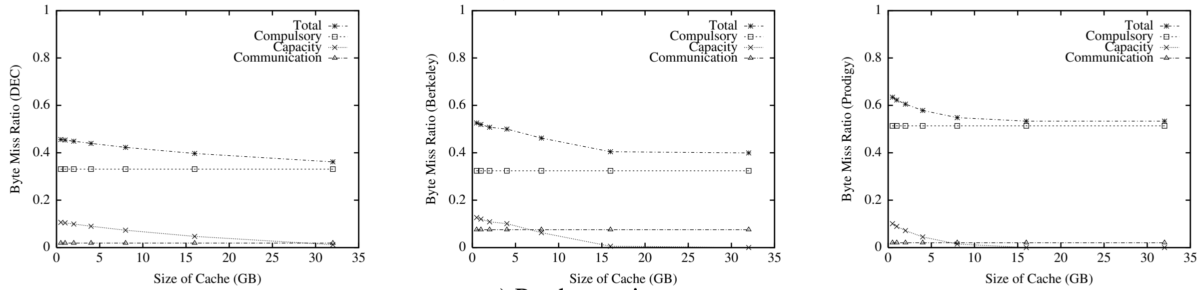
The second limitation of these traces is that they are gathered at proxies rather than at clients. Thus all of these traces will display less locality and lower total hit rates than would be seen by clients using such a system. Thus, our results should be interpreted as the hit rate for requests that miss in clients’ local caches or the response time for local misses; the total hit rates and response times observed by clients will generally be better than reported here.

2.2.2 Sources of cache misses

Figure 2 shows the breakdown of cache miss rates and byte miss rates for a global cache shared by all clients in the system as cache size is varied; the cache uses LRU replacement, and the figure caption explains the categories of misses we examine. For



a) Per-read miss rate



a) Per-byte miss rate.

Figure 2: Request miss rates and byte miss rates for the traces examined in this study. For each graph, we categorize misses as *capacity* for accesses to data that have been discarded from the cache to make space for other data, *communication* for accesses to data that have been invalidated from the cache due to an update, *error* for requests that generate an error reply, *uncachable* for requests that specify that the cache must contact the server to retrieve the result, for non-GET requests, or for requests that are designated as CGI requests in the traces, and *compulsory* for the first access to an object.

all of these traces, even an ideal cache will suffer a significant number of misses. Most of these misses are compulsory misses that occur the first time a client in the trace references an object. For the Berkeley and Prodigy traces, there are also a significant number of uncachable requests and communications misses. For multi-gigabyte caches, capacity misses are a relatively minor problem compared to the other sources of misses.

Given the relatively high rate of misses even with large caches, cache systems cannot afford to slow down misses. For example, in the DEC traces 19% of all requests result in compulsory misses (e.g., they are the first request to the data by any client in the trace). For these requests, any time spent in the cache system is wasted and will hurt overall response time.

Reducing the quantity or cost of global compulsory, uncachable, and error misses may become important if techniques such as those suggested by this study prove to be effective at harvesting the remaining potential hits and improving hit time. However, we do not address that problem here. Possible avenues include increasing the number of clients sharing a cache system [10, 18], allowing servers to send new data to cache systems before any nodes in the system request the data, providing better cache consistency [21, 25, 41] to reduce the number of requests marked as uncachable, caching dynamically generated results [36], dynamically replicating servers [37], and negative result caching [27, 5]. Since we are interested in studying the effectiveness of caching strategies, for the remainder of this study, we do not include "Uncachable" or "Error" requests in our results.

2.2.3 Sharing

Figure 3 illustrates the importance of enabling widespread sharing in large-scale cache systems. In this experiment, we configure the system as a three-level hierarchy with 256 clients sharing a L1 proxy, eight L1 proxies (2048 clients) sharing a L2 proxy, and all L2 proxies sharing an L3 proxy. As more clients share a cache, the compulsory hit rate for that cache falls because it becomes less likely that any given access to an object is the first access to that object. For example, in the DEC traces the hit rates improve from 50% for L1 to 62% for L2 and 78% for L3. Gribble and Brewer [18] and Duska et. al [10] reach similar

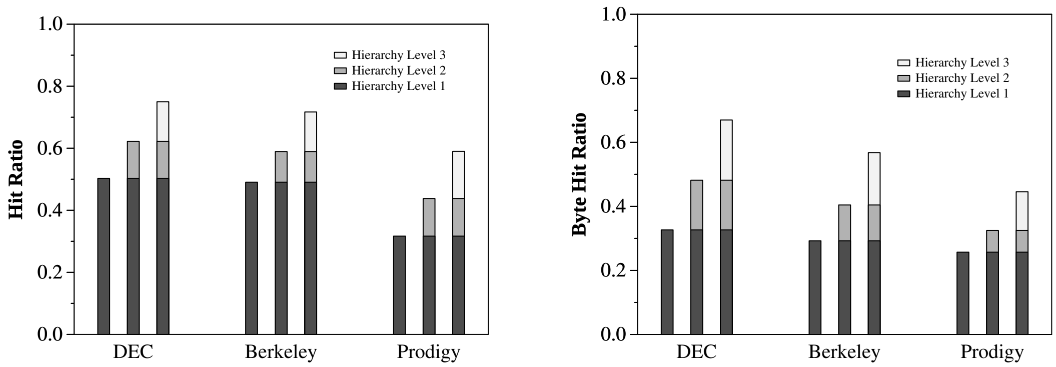


Figure 3: Overall per-read and per-byte hit rate within infinite L1 caches shared by 256 clients, infinite L2 caches shared by 2048, and infinite L3 caches shared by all clients in the trace. As sharing increases, so does the achievable hit rate.

conclusions. This characteristic of the workload suggests that cache systems should accommodate large numbers of clients and thereby reduce compulsory miss rates by avoiding bottlenecks that limit system scalability. Additionally, such systems will provide more scalable performance if the distance between clients and caches increases slowly as system size increases.

We will use this 3-level configuration of proxies as our default hierarchy for the remainder of this study.

2.3 Summary

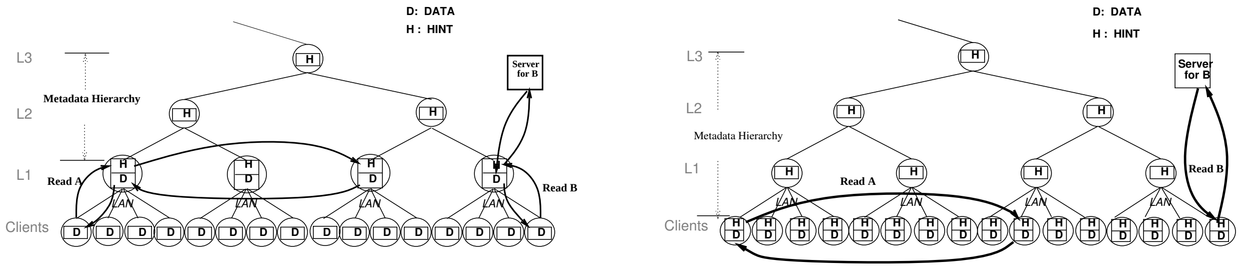
Although the design principles suggested by these measurements seem simple, traditional hierarchies of data caches violate them in various ways. First, a request may have to traverse several hops in a cache hierarchy to get to the data, and the data may traverse several hops to get back to the client. Second, cache misses will be significantly delayed by the having to traverse the hierarchy. With respect to the third principle, cache hierarchies appear to be able to do a relatively good job at sharing data among many caches: the root cache in such a hierarchy will see misses from all clients, so only the first client to access data will suffer a compulsory miss, and only the first client to access data after it changes will suffer a communications miss. Unfortunately, the shared, high-level caches in hierarchies also tend to be far away from clients with the most widely shared caches generally being the most distant. Thus, the strategy used to improve sharing in such hierarchies makes it hard to follow the fourth rule, cache data close to clients.

This analysis suggests that the focus of a new cache architecture should be on improving hit and miss times, while providing a scalable way to to maximize sharing to maintain good hit rates. As noted in Section 2.2.2, further improvements to global hit rates may require more radical techniques for dealing with communications and compulsory misses and uncachable requests, which are outside the scope of this study. There does, however, appear to be significant room for improving the hit and miss response times of distributed caches on the Internet.

3 Separation of data and metadata

In this section we examine a design that instantiates the principle of separation of data paths from metadata paths in order to implement two specific strategies. First, rather than traversing a hierarchy of data caches to locate cached data, the nodes maintain a directory of location-hints so they can send requests directly to caches with the data. The metadata hierarchy is used to maintain and propagate hints. Second, when a hit occurs at a remote cache, rather than sending data through a series of data caches in a hierarchy, the system uses cache-to-cache transfers to send the data directly from the cache supplying it to the cache that needs it. Thus, the data path consists of at most one cache-to-cache hop.

Figure 4-a illustrates our basic design, and Figure 4-b shows an alternate configuration. In the basic design, when a client cannot find data in its local cache, it sends the request to its closest proxy cache (an L1 proxy cache). The L1 proxy cache



(a) Basic hint hierarchy

(b) Alternate hint hierarchy

Figure 4: (a) Basic and (b) alternate hint-hierarchy design.

first tries to satisfy the request from data stored in its local cache. On an L1 miss, the proxy consults its local copy of the hint directory to find the nearest copy of the data. If the location-hint directory suggests a remote cache location for the item, the proxy sends the request to that cache, and that cache returns the data (the lines marked *Read A* in the figure.) If the hint directory does not know of a remote copy or if the hint turns out to be incorrect, the L1 proxy sends the request directly to the server indicated in the URL (the lines marked *Read B* in the figure.). In the basic design, the data always resides at the L1 proxy caches. The metadata hierarchy is only used to maintain and propagate location hints.

An alternative design based on the same principles is illustrated in Figure 4-b. In this design, the metadata hierarchy is extended beyond L1 caches to the clients. Each client uses its local location-hint directory to decide which cache, if any, to access. This design is able to consult the hint directory more quickly than the first design because the hints are local to the client, and direct remote cache access is even faster because the L1 cache is skipped. However, many large sites do not allow clients to access the Internet directly. Also, the client hint directory may be space-constrained and so may be less effective than the larger hint directory that may be economical at a proxy shared by multiple clients. Since the first configuration is most widely applicable, we focus on it for most of this section. For environments that can choose either design, we compare the trade-offs between the two at the end of this section.

Several other cache systems use metadata directories or multicast to locate data and then allow direct cache-to-cache data transfers. The primary differences among these schemes is how they structure their metadata directories. In local area networks, cooperative caching [8, 1] and global shared memory [11] hash a global directory across a set of nodes, and clients access the distributed global directory with network messages; in a LAN, the cost of this extra network hop is acceptable. In wide area networks, the WAN-xFS proposal [7] uses a hierarchy of metadata nodes to improve scalability, and the CRISP cache [14] uses a centralized global directory. Recently, the designers of CRISP have hypothesized that hashing the global directory for scalability or caching portions of this global directory at clients might be a useful addition to their design [13]. Several systems, including the Internet Cache Protocol (ICP) [40] and Zhang et. al's adaptive caching proposal [42], replace directories with multicast queries to nearby caches.

In contrast with these approaches, our system uses a scalable hierarchy of location hints combined with caching of these hints near clients. Four aspects of this design help it meet the design principles developed in the previous section. First, by caching location-hints near the clients, the system can quickly determine where, if anywhere, to find the needed data. This supports the principles of minimizing the number of hops to locate data and the principle of not slowing down misses. Second, by using a metadata hierarchy we localize updates of location-hints; this improves scalability, and it also speeds up the propagation of hints. Third, the system automatically maps the metadata hierarchy across the data nodes using a randomized hash function for scalability and fault tolerance. Fourth, the system uses small, fixed-sized records to store location-hint information. Consequently, this design reduces access time on a hit or miss, facilitates more widespread sharing of cached data, and reduces the overhead of distributing hints.

A potential additional benefit of separation of data and metadata is that it may reduce capacity misses by making better use of cache space than a cache hierarchy that caches the same data at multiple levels of the hierarchy. If peer-to-peer transfers are

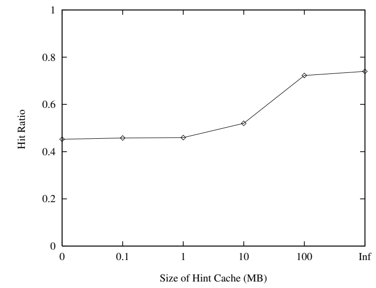


Figure 5: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache of the specified size. Each entry in the hint cache takes 16 bytes and they are stored in a 4-way set associative array with total size specified in MB on the x-axis.

allowed, hierarchical double caching is not needed to facilitate data sharing, so it will often be a waste of disk space. Because we do not expect capacity misses to be a significant limit on performance, we do not investigate this benefit in detail. However, this aspect of the design could come into play in space-constrained systems.

The next subsection describes these aspects of the design in detail, Section 3.2 discusses the implementation of our prototype, and then Section 3.3 examines the overall performance of this design.

3.1 Metadata hierarchy and location-hints

The goal of the metadata distribution hierarchy and hint caches is to provide a fast and scalable way for a node to determine where to go for data that is not locally cached. A hint is an $(objectId, nodeId)$ pair that indicates the node that caches the nearest known copy of an object. This subsection examines four aspects of the design: caching hints near clients, propagating hints through a distribution hierarchy, automatically configuring the hierarchy, and storing hints as small, fixed-sized records.

3.1.1 Caching location-hints near clients

Although this design specifies a metadata hierarchy for propagating hints, requests for hints are always satisfied locally by the leaf hint cache. By satisfying requests using only local information, hint caches observe the design principle of minimizing the number of cache hops to locate data. The system uses the hint hierarchy to propagate hints as a background task.

An advantage of maintaining hint caches rather than multicasting queries is that the propagation of hints happens independent of hint lookups. Conversely, multicast queries locate objects on demand by polling neighboring caches, potentially increasing latency. An additional advantage of maintaining hint caches is that a node with a hint cache can "query" virtually all of the nodes in a distributed system at once. This facilitates the design principle of maximizing sharing by allowing a cache to benefit from the contents of any other cache in the cooperative system. In contrast, multicast-based approaches generally limit sharing to a modest collection of nearby nodes in order to limit the overhead of responding to queries and to limit the time spent waiting to hear from all neighbors on cache misses. Multicast-based systems generally pursue more widespread sharing by breaking searches into several hops, each of which involves a multicast to a group of nodes nearer to the cache root or to the data server than the hop before. This effort to increase sharing, unfortunately, increases the number of hops on a hit or miss.

The downside of relying solely on hint caches is that they may sometimes contain incorrect information. Thus, hint caches trade potentially worse hit rates for better hit times or miss times. Two factors affect hit rates: (i) the capacity of the hint cache and (ii) the freshness of the hints.

First, a node's hint cache will only be effective if it can index significantly more data objects than the node can store locally, and it will observe the design principle of maximizing sharing if it can index most or all of the data stored by the caches in the cache system. As we describe in detail in Section 3.2.1, our system compresses hints to 16-byte, fixed-sized records. At this size, each hint is almost three orders of magnitude smaller than an average 10 KB data object stored in a cache [2]. Thus, if a

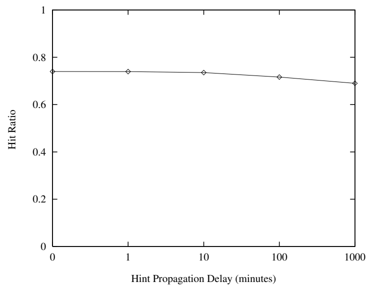


Figure 6: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache that is updated the specified number of minutes after objects appear or disappear from caches.

cache dedicates 10% of its capacity for hint storage, its hint cache will index about two orders of magnitude more data than it can store locally. Even if there were no overlap of what different caches store, such a directory would allow a node to directly access the content of about 63 nearby caches. Because the hint cache need only store one entry when an object is stored in multiple remote caches, coverage should be much broader in practice. Another way of viewing capacity is to consider the reach of a 500 MB index (10% of a modest, 5 GB proxy cache). Such an index could track the location of over 30 million unique objects stored in a cache system.

Figure 5 shows how the size of the hint cache affects overall performance in the DEC trace. Very small hint caches provide little improvement because they index little more than what is stored locally. For this workload, hint caches smaller than 10 MB provide little additional "reach" beyond what is already cached locally, but a 100 MB hint cache can track almost all data in the system.

Not only can the hints be stored in a reasonable amount of space, maintaining them consumes little network bandwidth. For example, for the DEC trace, the most heavily-loaded hint cache sees an average of 1.9 hint updates per second. As Section 3.2 describes, each hint update is 20 bytes, so maintaining this hint cache consumes only 38 bytes per second of bandwidth. Certainly this degree of bandwidth consumption poses no difficulty for T1 or better-connected sites; in fact, it is only about 1% of the bandwidth of a 33.6Kbit/s modem. Of course, peak bandwidths may be much higher than average bandwidths, but it appears that even a modestly-well connected host will be able to handle hint updates with little effort.

A second possible limitation of hints is that they may be stale. Updates to the hints will take some time to propagate through the system, during which time caches may make suboptimal decisions about where to send their requests. Figure 6 quantifies the dependence of global hit rate on the amount of time it takes to update hints in the system. In this experiment, we assume that whenever an object is dropped from a cache to make space for another object or an object is added to a cache, none of the hint caches learn of the change for an amount of time specified on the x-axis in the figure. This experiment suggests that the performance of hint caches will be good as long as updates can be propagated through the system within a few minutes. To facilitate this, our system uses a metadata hierarchy that preserves locality for hint updates, it uses a scalable hierarchy to avoid bottlenecks, and it uses small hint records to reduce the network overheads of propagation.

In our system, hint errors can be suboptimal positives (they direct a request to a far away node when a closer one also has the data), false positives (they direct a request to a node that doesn't have the data), or false negatives (they say no node has the data when, in fact, one does). On a suboptimal positive, a node pays a higher cost to access data than it should have. On a false positive, the remote cache replies with an error code and the first node sends the request to the server specified in the URL. On a false negative, the node sends its request to the server specified in the URL. In both the false positive or false negative case we do not make another attempt to locate the data in the hierarchy. One could imagine a system that queries other directories in a hierarchy or that multicasts when a hint cache query fails to reveal a nearby cache, but we expect that when the hint cache fails, it is unlikely that a hit will result, so further time spent searching the cache is likely to be wasted. This design decision reflects the principle of not slowing down misses.

Organization	Average update load at root
Centralized directory	5.7 updates/second
Hierarchy	1.9 updates/second

Table 5: Average number of location-hint updates sent to the root during the DEC trace. This simulation counts the total number of updates seen at the root of the hierarchy for a 3-level hierarchy with 64 L1 proxy caches, each of which serves 256 clients from the DEC trace. For the centralized directory, all nodes send updates to a single, centralized directory.

3.1.2 Propagating location-hints

Our system propagates hint updates through a metadata hierarchy to reduce the amount of information sent globally. When a node in the metadata hierarchy learns about a new copy of data from a child (or the local data cache if it is the leaf), it propagates that information to its parent only if the new copy is the first copy stored in the subtree rooted at the parent. The node determines this by examining its local hint cache; if the parent had already informed the node of a copy in the parent's subtree, then the node terminates the update propagation. Similarly, when a node learns about a new copy of data from a parent, it propagates that knowledge to its children if none of its children had previously informed it of a copy.

This protocol allows the hierarchy to limit the extent to which updates propagate through the tree. When some cache in the system references data cached nowhere else, that copy is the closest one to everyone in the system, so knowledge of it propagates to all the caches in the system. If a cache in the same subtree then loads the data into its cache, few other caches would regard that copy as significantly closer than the first copy, so it is not cost-effective to distribute information about that copy widely. On the other hand, if a far away cache loads the data, about half of the caches in the system would learn about this new, closer copy. The next copy might be of interest to a quarter of the caches, and so on.

Table 5 examines how effective the metadata hierarchy is at filtering traffic by comparing the number of updates seen by the root under the DEC trace for two configurations: a centralized directory which receives all updates and hierarchy that filters updates to reduce load. This experiment suggests that the filtering function of the simple hierarchy reduces load significantly compared to the centralized directory scheme. To further improve scalability, both the centralized directory or the root could be distributed across multiple physical nodes. We describe how our system automatically splits all internal nodes in the tree in the next subsection.

One potential drawback of terminating updates is that when an object is first brought into the system, the accesses from many caches could overload the holder of that first copy. However, in our system, when later caches pull copies into their subtrees, their neighbors learn of the new copy and will soon decide that is a better place to go for that data. In essence, the system builds a distribution tree for popular data. Using a hierarchical scheme rather than a single, centralized directory for distributing hint information helps this happen quickly, since nodes in a subtree near a new copy of data will learn of the new copy quickly, while updates to more distant nodes may take longer.

3.1.3 Self-configuring metadata hierarchy

In order to scale to large systems of caches, the hint distribution hierarchy is self-configuring. In particular, our system constructs a hierarchy using an algorithm developed by Plaxton et. al [31] to embed virtual hint distribution trees onto the nodes in the system. This algorithm has the following properties:

- *Automatic configuration.* Given a list of nodes and the approximate distances between them, the system builds trees automatically without manual specification of the parent/child relationships or manual assignment of objects to subtrees.
- *Fault tolerance and automatic reconfiguration.* As nodes enter or leave the system, the algorithm automatically reassigns children to new parents. This reassignment disturbs very little of the previous configuration.

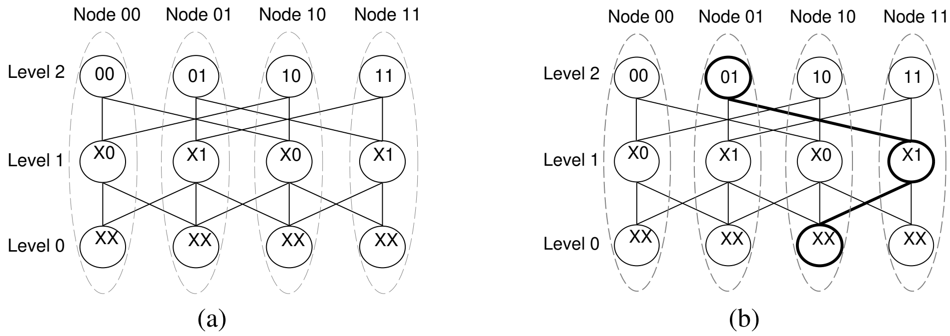


Figure 7: Plaxton et. al's randomized scalable tree algorithm for a binary tree. In part (b) of the figure, the hierarchy traversed by an object with ID $xxxxxxx01$ starting from node 10 is highlighted in bold.

- *Load distribution.* Different objects use different virtual trees depending on their object IDs. For example, if there are n nodes in the system, each node will be the root for roughly $1/n$ of the objects.
- *Locality.* Near the leaves of the virtual trees, the distance between parents and children tends to be small; near the roots, this distance is generally larger.

The algorithm works as follows. We assign each node in the system a pseudo-random ID (the MD5 signature of the node's IP address), and we assign each object in the system a pseudo-random ID (the MD5 signature of the object's URL.) Conceptually, the algorithm builds a tree for each object in which the IDs of nodes at high levels of the tree match the object's ID in many low-order bits, while the IDs of nodes at low levels of the tree match the object's ID in few low-order bits. The root node for an object is the node whose ID matches the object's ID in the most low-order bits.

Figure 7 illustrates the algorithm for a simple binary tree. The algorithm first embeds a set of virtual trees across the nodes as part (a) of the figure illustrates. In this setup phase of the algorithm, to construct level i of the hierarchy, each node finds two level $(i+1)$ parents. Both parents' IDs must match that node's bottom i bits, but one differs in the $(i+1)$th bit and the other matches in that bit (e.g., the second parent is the node, itself.) For locality, nodes choose the nearest eligible level $(i+1)$ parents. For low levels of the tree, only a few bits need to match, so parents tend to be quite nearby. At higher levels, a node often has to settle for a parent that is farther away. Note that there may some maximum value for i above which no valid level $(i+1)$ parents exist. When a node is removed (or added), its children at various levels select new parents. When a node is removed, of course, the new parents will generally be farther away than the old ones were.

An update of the metadata status for an object begins at the node where the status changes. At level i, a metadata node sends the update to the parent whose $(i+1)$th bit matches the object ID's $(i+1)$th bit. Part (b) of the figure shows the path that the system would use to notify a root hint cache that node 10 had just accessed an object with id $xxxxxxx01$. In our system, to accommodate flatter hierarchies with k-ary trees, nodes match $log(k)$ bits at a time to choose from among k parents.

3.2 Prototype implementation

We have added the protocol for maintaining location-hints to Squid version 1.1.20. There are three primary interface commands between Squid and the hint cache. One is an *inform* command, which tells the hint cache that a copy of a given object is now stored locally, and the hint cache should advertise it. The second is an *invalidate*, which tells the hint cache that that copy is no longer present; this causes the hint cache to advertise the non-presence, and to use the next best location (if one is known) of that object. The third is *find nearest*, which asks the hint cache to report the location of the nearest copy of an object, if one is known.

In addition, the hint module uses Squid's internal communications interfaces to propagate hint information. The system sends hint updates to neighboring caches as HTTP POST requests to the "route://updates" URL. Support for sending and receiving POSTs has been added to Squid in order to support both this facility and a best-effort object consistency protocol. Periodically, each cache POSTs to its neighbor a message containing the HTTP headers and the batch of all updates that the cache has seen in the most recent period; each update consumes 20 bytes: a 4-byte action, an 8-byte object identifier (part of the MD5 signature of the object's URL), and a 8-byte machine identifier (an IP address and port number.) Nodes randomly choose the period between updates using a uniform distribution between 0 and 60 seconds to avoid the routing protocol capture effects observed by Floyd and Jacobson [12].

It would be desirable if these updates were multicast efficiently to exactly the interested parties. However, the amount of data involved seems overly small to warrant the complexities of reliable multicast, and the interval until the next transmission concerning the object is too large to use raw UDP multicast. Thus, it seems wise to start by simply using TCP for data exchange. If an efficient inter-cache reliable multicast facility becomes available, it would be a useful optimization to our protocol.

3.2.1 Data structures and hint records

An important implementation detail in our prototype is our decision to use small, fixed-sized records and to store hints in a simple array managed as a k-way associative cache. In particular, our design stores a node's hint cache in a memory mapped file consisting of an array of small, fixed-sized entries. Each entry consumes 16 bytes: an 8-byte hash of a URL and an 8-byte machine identifier.[1]

We store a URL hash rather than a complete URL to save space in the table and to ensure that entries are of fixed size. Small records improve performance by allowing a node to store hint entries for a large number of objects; this facilitates the principle of maximizing sharing. Small records also reduce the network cost of propagating hints, and they allow a larger fraction of the hint table to be cached in a given amount of physical memory to avoid disk accesses.

Our design decision to work with URL hashes rather than complete URLs means that if two distinct URLs hash to the same value, the hint for one of them will be incorrect and a node may waste time asking another cache for data it may not have. In that case, because the read request contains the full URL, the remote node will return an error and the first node will treat the request as a miss. With 64-bit hash keys based on MD5 signature of the URLs, we do not anticipate that hash collisions will hurt performance.

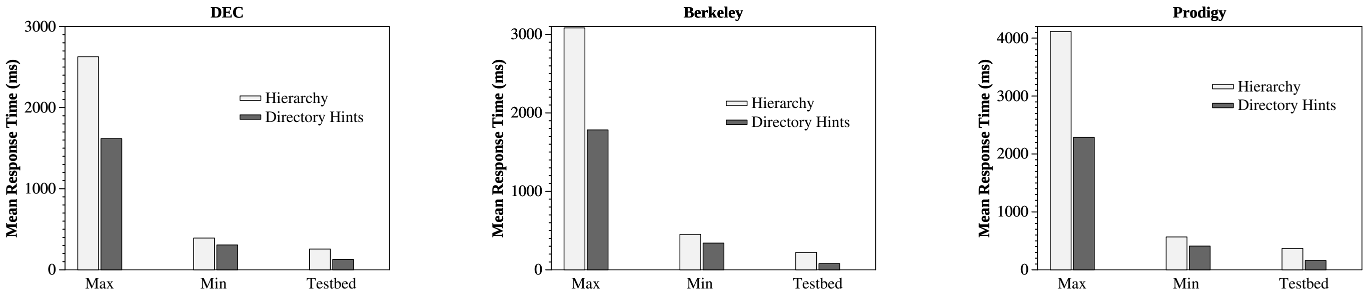
Fixed-sized records simplify and speed up manipulation of the on-disk data structures when the hint cache does not fit in physical memory. The system currently stores hints in an array that it manages as a 4-way associative cache indexed by the URL hash, and it maps this array to a file. Thus, if a needed hint is not already cached in memory, the system can locate and read it with a single disk access.[2]

We store entries in a 4-way associative cache rather than, for instance, maintaining a fully-associative LRU list of entries to reduce the cost of maintaining the hint cache when it does not fit in memory. We include a modest amount of associativity to guard against the case when several hot URLs land in the same hash bucket. One concern with using a hash-based data structure is that this approach does not cluster "hot" objects onto memory pages, so it may exhibit poor memory page locality. We are therefore considering adding a front-end cache of hint entries. However, it is not clear that any arrangement of a hint cache will yield good memory locality because the stream of references to the hint cache exhibits poor locality. In particular, reads of the hint cache have been filtered by the proxy's cache of objects stored locally, and once an entry for an object is read from the hint cache, that object will be stored in the proxy's cache; it is then unlikely that that hint will soon be referenced again. It does seem possible that there will be some temporal locality among updates to hint entries or between updates and reads (e.g., if an object is read into a cache by one node, that hint may soon be read by another). Our system does provide a mechanism for preferentially caching recently updated entries.

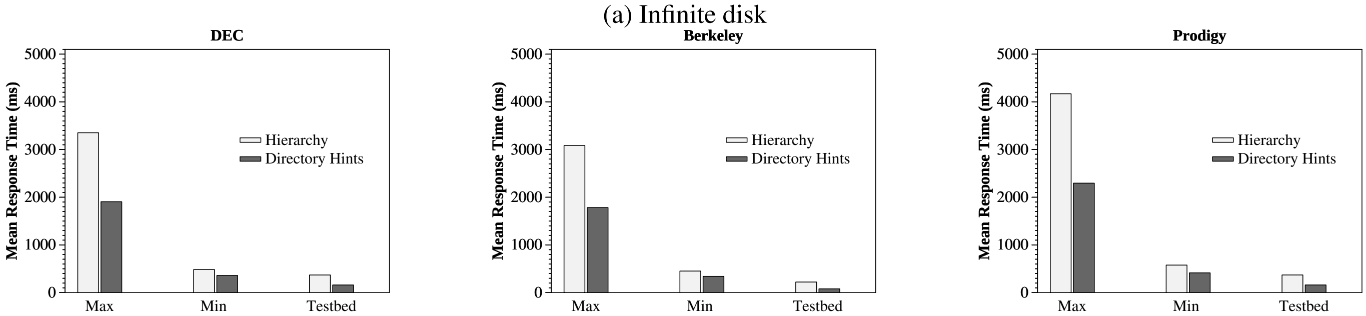
We measured the time to do a hint lookup to be 4.3 μs, in the case when the hint cache fits in memory and 10.8 ms, in the case when the hint must be faulted in from disk. These measurements were on a 200 MHz Sun Ultra-2, which is a 7.72 SPECInt95 machine.

[1] A special value for the hash is used to signify an invalid entry.

[2] We currently store the array in a file, so accesses may cause disk accesses of file system index structures. It would be trivial to store this array on the raw disk device if we find these metadata accesses to be expensive.



(a) Infinite disk



(b) Space constrained

Figure 8: Simulated performance for DEC, Berkeley, and Prodigy traces. The three groups of bars for each trace show the performance when the access times are parameterized by the *Testbed* times shown in Figures 1 or the *Min* and *Max* access times measured by Rousskov shown in the *Total Hierarchical* and *Total via L1* columns of Table 3. In figure (a) each node has infinite disk space for caching and hints. In figure (b) each node in the traditional hierarchy has 5 GB of disk used for caching objects, while each L1 node in the hint hierarchy system has 4.5 GB of disk for caching objects and each L1, L2, or L3 node in the hint hierarchy has 500 MB of disk for storing hints (notice that this arrangement gives more space to the standard hierarchy).

Traces	Max	Min	Testbed
Prodigy	1.80	1.38	2.31
Berkeley	1.79	1.32	2.79
DEC	1.62	1.28	1.99

Table 6: Performance of DEC, Berkeley, Prodigy traces measured as a ratio of the response time for the traditional data hierarchy to that of the hint directory architecture for the three access times shown in Figure 8

3.3 Performance evaluation

Figure 8 compares the simulated performance of the location-hints architecture to that of a traditional data cache hierarchy. For all workloads tested and for all cache access-time parameters tested, the hint-based algorithm significantly outperforms the standard hierarchy. Table 6 summarizes the performance for the different traces. The response times using hint directories are lower by a factor of 2.3 to 1.3 compared to the traditional data hierarchies. Notice that these improvements in response time come not from improving the global hit rate compared to a standard hierarchy but from improving hit times and miss times.

We also examined the trade-offs between the hint hierarchy configurations shown in Figure 4. Due to space limitations, we omit the graphs. In summary, for installations that allow clients to access distant caches directly, the choice of which architecture to use comes down to a trade-off between two design principles. On one hand, the (potentially) larger L1-proxy hint cache can index a larger number of remote cache entries, and it thus may facilitate the principle of widespread sharing better than the smaller client hint cache. On the other hand, accessing the client hint cache is faster than accessing the proxy hint cache, so the second configuration better fulfills the design principles of minimizing the number of hops on a hit or miss and of not slowing down misses; this advantage may be reduced if a system requires clients to send all requests through a proxy for security reasons. For the testbed parameters and the DEC trace, as long as client caches are large enough so that the false-negative rate for the client hint caches is below 50%, the alternate configuration is superior. At best, if client hint caches are large enough so that they match the hit rate of proxy hint caches, they improve response time by about a 20% compared to proxy hint caches for the testbed hierarchy and the DEC trace.

4 Push caching

The hint directory and direct forwarding algorithm described above does a good job at providing fast access to data stored in a distributed cache. However, from Figure 3 we note that a significant number of hits in the system are to distant caches and Figure 1 and Table 3 suggest that accessing these caches can be slow even with the direct-access facility provided by hints. For example, the data in Section 2.1 suggest that for the testbed hierarchy L1 cache accesses for 8KB objects are 4.75 times faster than direct accesses to caches that are as far away as L2 caches and 6.17 times faster than direct access to caches that are as far away as L3 caches. The data also suggest that the penalty for accessing distant caches is modest when Internet load is low (e.g., the *Total Direct Min* numbers in Table 3), but the penalty is high when the network is congested (e.g., the *Total Direct Max* numbers).

In this section we explore using push based algorithms to pursue the design principle: *cache data close to clients.* Unlike traditional caching schemes that cache data on demand, push caching anticipates future demands and sends objects to caches that have not requested them. We examine two simple algorithms that attempt to make controlled numbers of extra copies of data and distribute those copies through the system in order to replace hits to distant caches with hits to closer ones. We are in the process of implementing these algorithms in our prototype.

4.1 Push algorithms

Push algorithms attempt to balance the overhead of making and storing extra replicas with the benefit of reducing access times when the system can access data from a closer replica than before. For systems that cannot predict future accesses with certainty, push algorithms face a dilemma — creating more replicas incurs high costs but makes it more likely that a replica will be near where it needs to be.

We consider four design criteria for the push algorithms examined here. First, the algorithms should strive for efficiency. An algorithm is efficient if a large fraction of the replicas it creates actually succeed in improving system response time. Second, the algorithms should provide significant performance gains; an efficient algorithm that only pushes a small number of objects during the course of the trace (but which succeeds in slightly improving performance with each push) would not be worth pursuing. Third, the algorithm should be adaptive; if the algorithm is running in a system that is resource-rich, it should aggressively replicate data to improve response time even at the cost of efficiency. Conversely, if the system is resource-poor, it should be more conservative. Fourth, the algorithm should be simple both to ease implementation and to avoid consuming too many resources figuring out what to push; with potentially millions of objects in a cache, the push system cannot spend too much effort tracing the popularity of or making decisions about any one of those objects.

The class algorithm we consider observes two important restrictions.

1. We assume that the algorithms are not supplied with knowledge of future access patterns. They must therefore predict future access patterns based on the past. In particular, we do not assume any external directives about future accesses such as hoard lists [22, 24] or server hits [30].

2. We limit pushing or prefetching to increasing the number of copies of data that are already stored at least once in the cache system. Thus, our algorithms can only affect the number of L1, L2, and L3 hits, not the number of system-wide misses. Notice that more aggressive prefetching algorithms than the ones we examine could could fetch objects that are not cached anywhere in the cache hierarchy by accessing the original servers. For example, one could imagine having the cache hierarchy "crawl" the Internet in the background, looking for new pages. Clearly such an algorithm could further improve performance by reducing the number of complete misses endured by the system. However, in this study we restrict ourselves to algorithms that do not increase the load on servers outside of the cache system.

We examine two push algorithms and compare them to a base case and best case.

4.1.1 Base case and best case

For reference, we compare our results to two extreme cases. In the base case, the system only replicates data on demand and in response to client accesses to the data. Our first base case algorithm is the standard data hierarchy algorithm, which creates replicas of data along the hierarchy path between the client and the level of the cache that supplies the data (e.g., on a L3 hit, the hierarchy creates new L2 and L1 replicas). Our second base case algorithm is the hint hierarchy algorithm described in Section 3.

We also compare our results to a best case. In the best case algorithm, we imagine that whenever an object is fetched into the cache system, it appears not only in the cache that referenced the data but also in all caches that will reference it in the future. We further assume that these extra copies do not consume additional disk space at the extra caches, so the caches' local hit rates are not damaged by this aggressive replication. This scenario replaces all L2 and L3 hits with L1 hits and does not change the number of misses. Our best case algorithm represents the best performance achievable by a push algorithm that meets the two restrictions mentioned above. Note that a system with large amounts of spare bandwidth and disk space could approach this level of performance by broadcasting all data fetched on misses to all caches. We seek more efficient algorithms that can work well when disk space or network bandwidth or both are constraints.

4.1.2 Push updates

We first consider a simple, efficient algorithm that approximates update-based cache consistency. This algorithm is based on the observation that when an object is modified, a good list of candidates to reference the new version of the object is the list of caches that previously cached the old version of the object. Thus, when the cache system fetches an object due to a communication miss (see Figure 2), it sends copies of that object to caches that were storing the previous version of that object.

For cache systems that use invalidate-based cache consistency, caches track objects that they recently invalidated from their cache, and when they receive hints about locations of objects on that list, they send read requests to the hinted location. For cache systems that use polling-based consistency, the hints should be augmented with the objects' version numbers or last-modified-times. In that case, if a cache hears the location of a new version of an object it is caching, it should fetch that object.

This algorithm can adapt to limit the resources consumed by hint propagation in two ways. First, whenever a cache updates an object, the cache "ages" the object by moving it down the LRU list. Thus, objects that are updated many times without being read will be evicted from the cache. In resource-rich configurations, this aging will be slow, while in resource-poor systems it will be fast. Second, caches place an upper limit on the update-fetch bandwidth they will consume and discard update-fetch requests that exceed that rate.

With respect to the four design criteria discussed above, we note that this algorithm restricts itself to looking at a particular situation where it can be relatively confident that the copies of the object it makes will be useful. This allows the algorithm to achieve good efficiency at a cost of limiting the maximum performance it can achieve. The algorithm is somewhat adaptive in that it can be configured to use more or fewer resources by changing the aging rate depending on the environment. Finally, the algorithm is simple — it makes decisions based on information already present in the system, and it uses events already present in the system to notify caches when to consider making copies of particular objects.

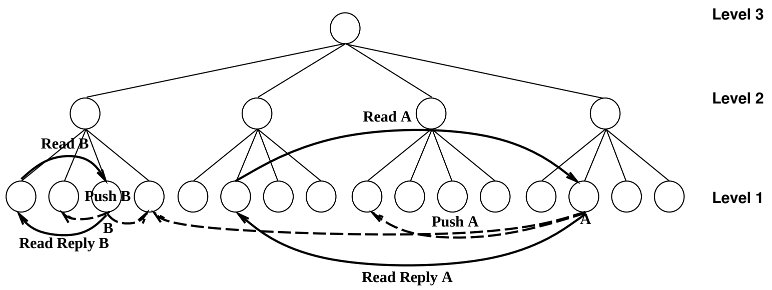


Figure 9: Hierarchical push algorithm for 3-level hierarchy. Once two level-2 subtrees fetch object A, the algorithm pushes object A to all level-2 subtrees. Once two level-1 subtrees under a level-2 subtree fetch object B, the algorithm pushes object B to all level-1 nodes under that level-2 parent.

4.1.3 Hierarchical push on miss

The hierarchical push on miss algorithm is more general and less selective than the push updates protocol. Because the algorithm considers more objects as candidates for replication it is less efficient than the previous algorithm, but it can achieve more significant performance gains.

As Figure 9 illustrates, when a cache fetches an object from a cousin for which a level-l parent is the least common ancestor in the metadata hierarchy, the cache supplying the object also pushes the object to a random node in each of the level-$(l-1)$ subtrees that share the level-l parent. For example, in a three-level metadata hierarchy when a cache fetches data from a distant cache that shares a level-3 parent, the system creates one copy of that object in each of the level-2 hierarchies. Thus, the next access to the data will cost at most a level-2 access. Similarly, on a level-2 hit, the algorithm pushes the object to *all* level-1 caches that share the same level-2 parent.

The intuition behind this algorithm is that if two subtrees in a hierarchy access an item, it is likely that many subtrees in that hierarchy will access that item. Notice that although this algorithm is simple and does not explicitly track object popularity, it results in the desired effect that popular items are more widely replicated than unpopular items. Also note that if there is locality within subtrees, items popular in one subtree but not another will be more widely replicated in the subtree where the item is popular.

With respect to the four design criteria discussed above, this algorithm is less specialized than the previous algorithm. This allows it to replicate objects that the previous algorithm cannot consider. By considering a wider range of candidates for replication, this algorithm may be less efficient, but it may also provide better latency improvements. The algorithm can be adapted to resource rich environments (push to k nodes in a subtree rather than just 1 node in a subtree) or resource poor environments (push a copy to $\frac{1}{j}$ of the eligible subtrees). We will examine three configurations: push-1 sends a copy to 1 node in each eligible subtree, push-half sends a copy to half of the nodes in each eligible subtree, and push-all sends a copy to all nodes in an eligible subtree. Finally, the algorithm is simple — it makes decisions based on information already present in the system; also, it uses events already present in the system to notify caches to consider whether to make copies of particular objects.

4.2 Evaluation

We use our simulator to evaluate these algorithms. The base cases correspond to the standard hierarchy and our hint algorithm from the previous section. The best case corresponds to a system that uses our hint algorithm, but for which all L3 and L2 hits are replaced by L1 hits. We then examine both the update push and the hierarchical miss push algorithms.

Although we do not expect capacity misses to be a major factor in understanding systems that replicate data on-demand, in push based systems, speculative replication may displace more valuable data from caches. To monitor this effect in our

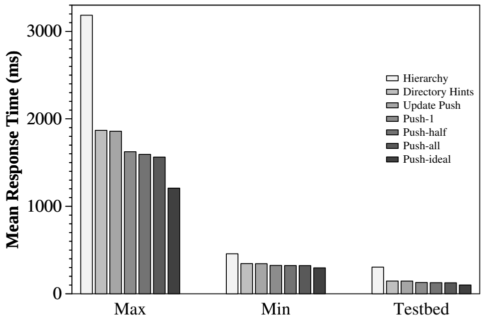
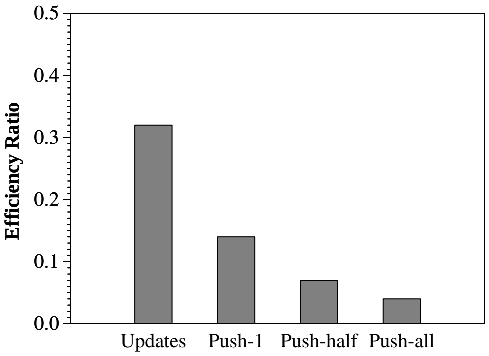
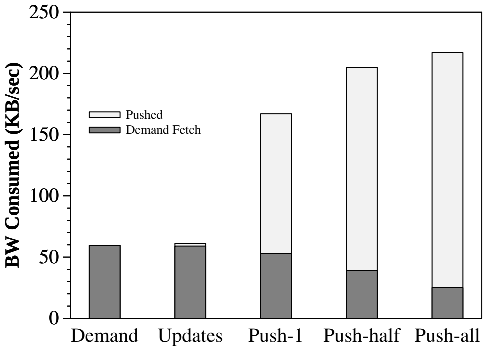


Figure 10: Simulated response time for DEC trace workload for six algorithms: no push (data hierarchy), no push (hint hierarchy), update push, push-1 (send a copy to 1 node in each eligible subtree), push-half (send a copy to half of the nodes in each eligible subtree), and push-all (send a copy to all nodes in an eligible subtree).



(a)



(b)

Figure 11: (a) Efficiency and (b) bandwidth of push algorithms for the DEC trace. For the efficiency graph, each bar shows the fraction of all bytes pushed that are later accessed. The bandwidth graph shows the bandwidth consumed by pushed data and data fetched on demand for the algorithms.

simulations, we report results for the space-constrained configuration in which each of the 64 L1 caches has 5 GB of capacity. For the best case simulations, we also limit the local cache to 5 GB, but we do not charge the caches any space for objects that have been pushed to them but not referenced by them.

In these simulations, we look at two figures of merit. First, we examine how much the push algorithm reduces system response time by replacing accesses to distant caches with accesses to nearby ones. Second, we examine efficiency by examining what fraction of data that is pushed is later used.

Figure 10 shows the simulated response time for the DEC trace under a range of push options. This experiment suggests that an ideal push algorithm could achieve speedups of 1.54 to 2.63 compared to the no-push data hierarchy, and speedups of 1.21 to 1.62 compared to the no-push hint hierarchy; the largest speedups come when the cost of accessing remote data is high such as the Max value in Rousskov's measurements (see Table 3). The hierarchical push algorithms described here achieves speedups of 1.42 to 2.03 compared to the no-push data hierarchy, and speedups of 1.12 to 1.25 compared to the no-push hint hierarchy. Update push does not achieve any appreciable performance improvement compared to the no-push, hint-hierarchy case.

Figure 11 shows the efficiency of the algorithms. This experiment suggests that the update push algorithm is efficient: one-third of the data it pushes are used. The hierarchical algorithms have efficiencies between 13% and 4% and they increase the bandwidth consumed by up to a factor of four compared to the demand-only case. This may be an acceptable trade-off of bandwidth for latency in many environments.

5 Related work

In Section 3, we discussed a number of cache systems that separate data from metadata using metadata hierarchies, directories, or multicast. This section considers other areas of work relevant to understanding our results.

Several studies have examined Internet workloads in depth with the goal of understanding how to improve performance. Arlitt and Williamson [2] examined basic workload characteristics for six servers. Duska et. al [10] examined workloads for several 3-week traces, Gribble and Bewer [18] examined a large trace of Berkeley HomeIP users. These studies support the conclusion that cache architectures that scale are important because increasing the number of users sharing a cache system increases the hit rates achievable by that system. Rousskov [32] and Maltzahn et. al [26] measure the performance of cache servers deployed in the field and examine why the performance of these servers has been worse than laboratory tests predicted [5]. These studies suggest that long network round trip latencies may be a significant factor.

Hierarchical caching has been examined in the context of file systems [4, 33, 16]. Muntz and Honeyman [28] concluded that the additional hops in such a system often more than offset improvements in hit rate and characterized the extra level of cache as a "delay server." We reach similar conclusions in the context of Internet caching, leading to our design principle of minimizing the number of hops on a hit or miss.

Hierarchical caching has been more widely accepted on the Internet, and it is common in areas with low-speed connections to the Internet [29, 35]. In addition, many sites add proxy caching to their corporate firewalls. A widely deployed and studied system has been the publicly-available Harvest cache [5] and its successor, the Squid cache [39]. A key motivation of our design is that we disagree with the conclusion that "cache hierarchies do not significantly reduce cache performance on cache misses" [5]. The evidence suggests to us that for large scale systems with long network delays or busy servers or both, reducing the number of hops is vital to performance.

Several researchers have proposed improving the scalability of a data hierarchy by splitting responsibilities according to a hash function [38]. This approach may work well for distributing load across a set of caches that are near one another and near their clients, but in larger systems where clients are closer to some caches than others, the hash function will prevent the system from exploiting locality.

Several studies have examined push caching and prefetching in the context of web workloads [19, 20, 30]. These systems all used more elaborate history information to predict future references than the algorithm we examine. Because large, shared caches do a good job at satisfying references to popular objects, we explore prefetching strategies that will work well for the remaining large number of objects about whose access patterns little is known. Kroeger et. al [23] examined the limits of performance for caching and prefetching. They found that the rate of change of data and the rate of accesses to new data and new servers limits achievable performance.

6 Conclusions

Although caching is increasingly used in the Internet to reduce network traffic and the load on web servers, it has been less successful in reducing response time observed by clients. We examine several environments and workloads and conclude that this may be because traditional hierarchical caches violate several basic design principles for distributed caching on the Internet. We examine two specific strategies based on these principles. First, we find that by minimizing the number of hops on a hit or miss, supporting widespread sharing, not slowing down misses, and avoiding centralized bottlenecks, a strategy of direct cache-to-cache data transfers supported by hint directories can improve performance by a factor of 1.27 to 2.30 for workloads and configurations we examine. Second, we find that simple push algorithms can support the principle of accessing data from nearby caches and achieve a further improvement of up to a factor of 1.25. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional cache hierarchy.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] M. Arlitt and C. Williamson. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996. http://www.cs.usask.ca/projects/discus/discus_reports.html.

[3] G. Banga and P. Druschel. Measuring the Capacity of a Web Server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[4] M. Blaze. *Caching in Large-Scale Distributed File Systems*. PhD thesis, Princeton University, January 1993.

[5] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[6] Inktomi Corporation. The Economics of Large-Scale Caching. http://www.inktomi.com/Tech/EconOfLargeScaleCache.html, 1997. http://www.inktomi.com/Tech/EconOfLargeScaleCache.html.

[7] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 150–160, May 1994.

[8] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[9] Digital Equipment Corporation. Digital's Web Proxy Traces. ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, September 1996.

[10] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[11] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 201–212, December 1995.

[12] S. Floyd and V. Jacobson. The Synchronization of Periodic Routing Messages. *IEEE Transactions on Networking*, 2(2):122–136, April 1994.

[13] S. Gadde, J. Chase, and M. Rabinovich. Directory Structures for Scalable Internet Caches. Technical Report CS-1997-18, Duke University Department of Computer Science, November 1997.

[14] S. Gadde, M. Rabinovich, and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 93–98, May 1997.

[15] Steve Glassman. A Caching Relay for the World Wide Web. In *Proceedings of the Third International World Wide Web Conference*, pages 69–76, May 1994.

[16] J. Goldick, K. Benninger, W. Brown, C. Kirby, C. Maher, D. Nydick, and B. Zumach. An AFS-Based Supercomputing Environment. In *Proceedings of the Thirteenth Symposium on Mass Storage Systems*, pages 127–132, April 1993.

[17] S. Gribble. UC Berkeley Home IP HTTP Traces, July 1997. Available at http://www.acm.org/sigcomm/ITA/.

[18] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[19] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[20] J. Gwertzman. Autonomous Replication in Wide-Area Networks. Masters thesis, Harvard University, April 1995. Available as Tech Report TR-17-95.

[21] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[22] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[23] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[24] G. Kuenning. *Seer: Predictive File Hoarding for Disconnected Mobile Operation*. PhD thesis, The University of California at Los Angeles, 1997.

[25] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[26] C. Maltzahn, K. Richardson, and D. Grunwald. Performance Issues of Enterprise Level Web Proxies. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[27] P. Mockapetris and K. Dunlap. Development of the Domain Name System. *Computer Communications Review*, 18(4):123–133, August 1988.

[28] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems or Your cache ain't nuthin' but trash. In *Proceedings of the Winter 1992 USENIX Conference*, pages 305–313, January 1992.

[29] D. Neal. The Harvest Object Cache in New Zealand. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[30] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 22–36, July 1996.

[31] C. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

[32] A. Rousskov. On Performance of Caching Proxies. http://www.cs.ndsu.nodak.edu/ rousskov/research/cache/squid/profiling/papers, 1996.

[33] H. Sandhu and S. Zhou. Cluster-Based File Replication in Large-Scale Distributed Systems. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 91–102, June 1992.

[34] M. Schwartz. Formal Service Agreements for Scaling Internet Caching. In *NLANR Web Cache Workshop*, June 1997. http://ircache.nlanr.net/Cache/Workshop97/Papers/Schwartz/schwartz.html.

[35] N. Smith. The UK National Web Cache: A State of the Art Report. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[36] A. Vahdat and T. Anderson. Transparent Result Caching. In *Proceedings of the 1998 USENIX Technical Conference*, June 1998. To appear.

[37] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. WebOS: Operating System Services for Wide Area Applications. Tech Report CSD-97-938, U.C. Berkeley Computer Science Division, March 1997.

[38] V. Valloppillil and K. Ross. Cache Array Routing Protocol v1.0. IETF Internet Draft, October 1997. http://www.etext.org/Internet/Internet-Drafts/draft-vinod-carp-v1-02.txt.

[39] D. Wessels. Squid Internet Object Cache. http://squid.nlanr.net/Squid/, January 1998.

[40] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP), version 2. Request for Comments RFC-2186, Network Working Group, 1997. http://ds.internic.net/rfc/rfc2187.txt.

[41] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eightteenth International Conference on Distributed Computing Systems*, May 1998.

[42] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the NLANR Web Cache Workshop*, June 1997. http://ircache.nlanr.net/Cache/Workshop97/.