Design Considerations for Distributed Caching on the Internet*

Renu Tewari, Michael Dahlin, Harrick M. Vin
Department of Computer Sciences, University of Texas at Austin
{tewari,dahlin,vin}@cs.utexas.edu

Jonathan S. Kay
Cephalapod Proliferationists, Inc.
jkay@cs.utexas.edu

Technical Report UTCS TR98-04
February 1998 (Revised October 1998)

Abstract

In this paper, we describe the design and implementation of an integrated architecture for cache systems that scale to hundreds or thousands of caches with thousands to millions of users. Rather than simply try to maximize hit rates, we take an end-to-end approach to improving response time by also considering hit times and miss times. We begin by studying several Internet caches and workloads, and we derive three core design principles for large scale distributed caches: (1) minimize the number of hops to locate and access data on both hits and misses, (2) share data among many users and scale to many caches, and (3) cache data close to clients. Our strategies for addressing these issues are built around a scalable, high-performance data-location service that tracks where objects are replicated. We describe how to construct such a service and how to use this service to provide direct access to remote data and push-based data replication. We evaluate our system through trace-driven simulation and find that these strategies together provide response time speedups of 1.27 to 2.43 compared to a traditional three-level cache hierarchy for a range of trace workloads and simulated environments.

1 Introduction

The growth of the Internet and the World Wide Web allow increasing number of users to access vast amounts of information stored at geographically distributed sites. However, long round-trip propagation delays between client and server sites, as well as hot spots of network and server load yield high latencies for information access.

Caching provides an opportunity to combat this latency by allowing users to fetch data from a nearby cache rather than from a distant server. But because users tend to access many sites, each for a short period of time, hit rates of per-user caches are low. Thus, some organizations have begun to utilize shared proxy caches [19] or hierarchical caches [8] so that each user can benefit from data fetched by others. Current shared cache architectures face a dilemma. On one hand, they wish to share data among a large number of clients to achieve good hit rates. On the other hand, as a shared cache system services more clients, the response time it provides to any one client worsens due to the increased distance between client and cache, the increased load on the cache, or the increased number of levels in the cache hierarchy. Thus, these hierarchies of data caches achieve modest hit rates [2, 14, 19, 21], can yield poor response times on a cache hit [30, 36], and can slow down cache misses.

This paper examines techniques for building systems of shared, distributed caches that scale to hundreds or thousands of caches with tens of thousands to millions of users. We believe our techniques will be of interest to system designers building large-scale, distributed cache infrastructures in a range of environments including network service providers, independent service providers, cache service providers [9, 33, 40, 45], collections of caches linked by formal service agreements [39], and large intra-nets.

Using measurements of several caches on the Internet and analysis of several traces of web traffic, we first attempt to understand the factors that limit the performance of current web caches. We find that to provide good performance to the end user, it is important not only to maximize hit rates, but also to improve hit times and miss times. Based on these measurements, we derive three basic design principles for large-scale caches: (1) minimize the number of hops to locate and access data on both hits and misses, (2) share data among many users and scale to many caches, and (3) cache data closed to clients. Although these principles may seem obvious in retrospect, current cache architectures routinely violate them at a significant performance cost. For example, hierarchical caches in the United States are often seen as a way to reduce bandwidth consumption rather than as a way to improve response time.

To address these principles, we design a scalable, high-

*This work was supported in part by an NSF Research Infrastructure Award (CDA-9624082) and grants from IBM, Intel, Lucent Bell Laboratories, Mitsubishi Electronic Research Laboratories (MERL), NASA, Novell, and Sun Microsystems. Dahlin was also supported by an NSF CAREER grant (CCR-9733842), and Vin was also supported by an NSF CAREER grant (CCR-9624757).

performance data-location service that tracks where objects are replicated. We describe how to construct such a service and how to use this service to meet our design principles via direct access to remote data and push-based data replication. Through simulation using a range of workloads and network environments, we find that direct access to remote data can achieve speedups of 1.3 to 2.3 compared to a standard hierarchy. We also find that pushing additional replicas of data provides additional speedups of 1.12 to 1.25.

We construct our data-location service using a scalable hint hierarchy in which each node tracks the nearest location of each object. Scalability and performance of the hint hierarchy comes from four sources. First, we use simple, compact data structures to allow each node's view of the hint hierarchy to track the location many objects. Second, the location system satisfies all on-line requests locally using the hint cache; the system only sends network messages through the hierarchy to propagate information in the background—off the critical path for end-user requests. Third, the hierarchy prunes updates so that updates are propagated only to the affected nodes. Fourth, we adapt Plaxton's algorithm [35] to build a scalable, fault tolerant hierarchy for distributing information.

We have implemented a prototype of our system by augmenting the widely-deployed Squid proxy cache [45].[1] It implements hint caches, push-on-write, and self-configuring dynamic hierarchies.

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 provides an overview of our design, and Section 4 discusses implementation details and evaluates system performance. 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.

2.1 Workload characteristics

We examine how characteristics of web workloads stress different aspects of shared cache systems. We find that:

- Cache systems should *share data among many clients* to reduce compulsory misses (misses due to the first references to objects by clients) and *scale to large numbers of caches*.

[1] The simulator and prototype are available at http://www.cs.utexas.edu/users/tewari/cuttlefish.

Trace	# of Clients	Accesses (millions)	Distinct URLs (millions)	Dates	# of Days
DEC [13]	16,660	22.1	4.15	Sep96	21
Berkeley [21]	8,372	8.8	1.8	Nov96	19
Prodigy	35,354	4.2	1.2	Jan98	2

Table 1: Trace workloads. Note: 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.

- Cache hit time constitutes a significant fraction of the total information access latency. Hence, cache architectures *should minimize the cost to access a cache.*

- Even an ideal cache will have a significant number of compulsory and communication misses (misses to objects that have changed since they were last referenced.) Thus, *cache systems should not slow down misses*.

We also find that capacity misses (misses to objects that have been replaced due to limited cache capacity) are a secondary consideration for large-scale cache architectures because 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, pre-fetching and push caching), capacity may again be a significant consideration.

2.1.1 Methodology

Our simulation experiments use three multi-day traces taken at proxies serving thousands of clients. Table 1 summarizes key parameters. In analyzing the cache behavior of these traces, we use the first two days of each trace to warm our caches before gathering statistics.

To determine when objects are modified and should not be serviced from the cache, we use the last-modified-time information provided in the DEC traces. For the other traces or when the DEC trace does not contain the last-modified-time information, 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 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 change. We do this for two reasons. First, techniques for approximating or providing strong cache consistency in this environment are improving [24, 29, 47], 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 affect our results. First, although we use traces with thousands of clients, it still represents only a small fraction of the client population on the web. Several studies [6, 14, 21] suggest that will improve as more clients are included in a cache system 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.

2.1.2 Sources of cache misses

Figure 1 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. Misses fall into four categories:

1. Compulsory misses. These misses correspond to the first access to an object. The two key strategies for reducing compulsory misses are increasing the number of clients sharing a cache system and prefetching. Facilitating sharing is an important factor in designing large scale caches, and we discuss sharing in detail in the next subsection. We do not address prefetching in this paper.

2. Capacity misses. These misses occur when the system references an object that it has previously discarded from the cache to make space for another object. Our original intuition had been that it would be important to coordinate the contents of different caches to minimize capacity misses. However, the data suggest that for shared caches, capacity misses are a relatively minor problem that can be adequately addressed by building cache nodes with a reasonable amount of disk space. This study, therefore, does not focus on coordinated cache replacement [11, 16].

3. Communication/consistency. These misses occur when a cache holds a stale copy of data that has been modified since it was read into the cache. Providing efficient cache consistency in large systems is a current research topic [24, 29, 47], and we do not focus on that problem here. We do note, however, that the data location abstraction we construct could also be a building block for a scalable consistency system [10].

4. Uncachable/error. Objects are marked "uncachable" or encounter errors for a number of reasons, some of which might be addressed by more sophisticated cache protocols that support better cache consistency, caching dynamically generated results [41], dynamically replicating servers [43], negative result caching [31, 8], and caching programs along with data [7, 42]. We do not address such protocol extensions here. Also, because 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.

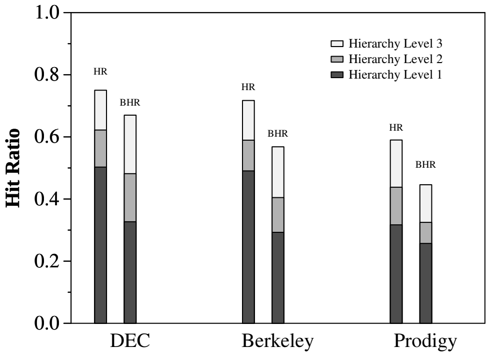


Figure 2: Overall per-read hit rate (HR) and per-byte hit rate (BHR) within infinite L1 caches shared by 256 clients, infinite L2 caches shared by 2048, and infinite L3 caches shared by all clients in the trace. As sharing increases, so does the achievable hit rate.

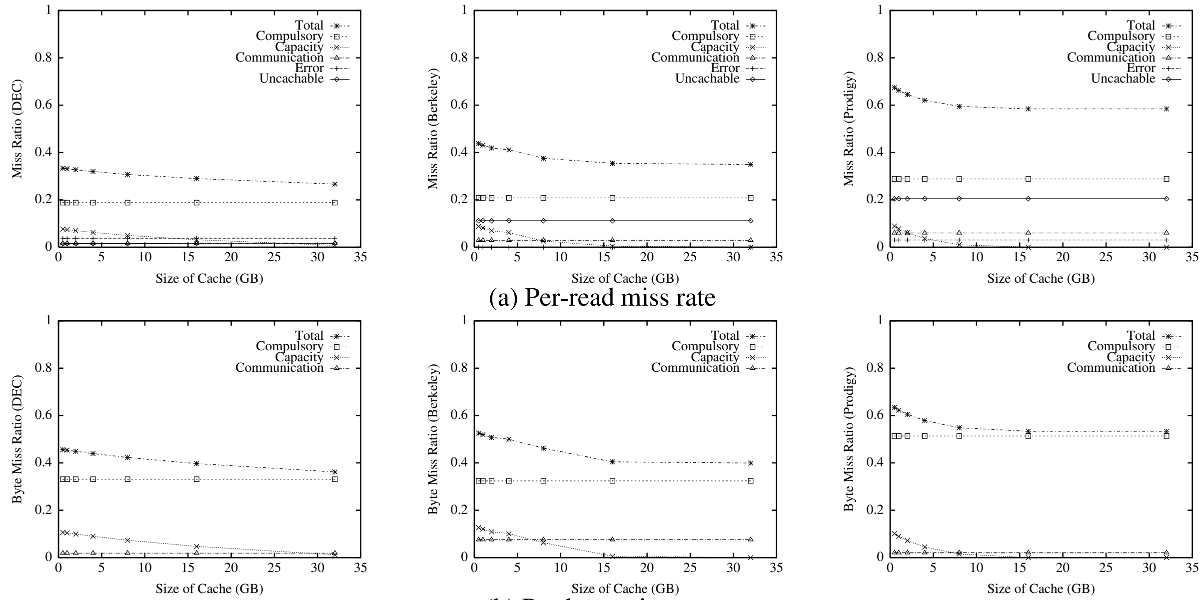
For all of the traces, even an ideal cache will suffer a significant number of misses. Thus, one key design principle is that in addition to having good hit rates and good hit times, cache systems should not slow down misses.

2.1.3 Sharing

Figure 2 illustrates the importance of enabling widespread sharing in large 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 miss 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 going from a 256-client shared cache to a 16,336-client shared cache improves the byte hit rate by nearly a factor of two. Prior studies [21, 14, 6] have also reached similar conclusions. This characteristic of the workload suggests that cache systems should accommodate large numbers of clients and thereby reduce compulsory miss rates.

For caches with different degrees of sharing, Figure 3 depicts the variation in the request response times with increase in the distance between clients and the shared cache. It illustrates a dilemma faced by the designers of shared proxy caches: although it is important to share a cache among many clients, it is also important that the shared cache be close to clients. For example, a 256-client cache with an average hit time of 50 ms can outperform a 16,336-client cache that averages 300 ms per access.

Another limitation of large, monolithic caches is load. Duska et. al [14] point out that shared caches require thousands of clients and that Maltzahn and Richardson [30] found that peak processor throughput of the Squid v1.1 proxy was less than 35 req/s on a Digital AlphaStation 250 4/266. Gribble and Brewer observe that request rates are bursty at the time scale of seconds [21]. In our traces, we found peak request rates of 2850 req/s (139 MB/s) over periods of 1 second and peak rates of 294 req/s (14.4 MB/s) over periods of 10 seconds. Although processor performance will improve, we



(a) Per-read miss rate

(b) Per-byte miss rate.

Figure 1: 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.

also expect the number of web users, per-request size, and per-user request rates to increase, especially as multimedia content becomes more prevalent.

2.2 Analyzing traditional cache hierarchies

Hierarchical caches attempt to resolve the dilemma of sharing versus locality and scalability by putting different caches near different groups of clients, and sending misses from these L1 caches through additional layers of cache exploit sharing. This subsection examines how such arrangements work in large-scale systems.

Traditional hierarchical cache architectures such as Harvest [8] or Squid [45] define parent-child relationships among caches. Each cache in the hierarchy is shared by a group of clients or a group of children caches. 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 caches then send the data down the hierarchy and each cache along the path stores the data.

Although hierarchies such as Harvest and Squid were designed under the assumption that caches could be layered without adding much delay [8], 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[2] to higher-level cache hits and to cache misses [32]. 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.

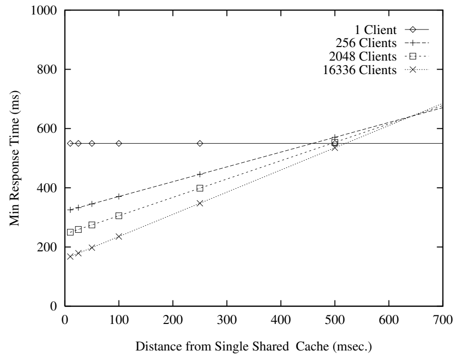


Figure 3: Response time for caches with different degrees of sharing as the distance between clients and caches increase

To examine these effects, 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 [36]. Although Squid supports the Internet Cache Protocol (ICP)

[2] In Squid, requests are propagated using store-and-forward but replies may be pipelined across nodes.

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

Table 2: Testbed hierarchy.

to allow a cache to query its neighbors before sending a miss to a parent [46], since we are interested in the best costs for traversing a hierarchy, neither configuration we examine uses ICP.

Our analyses demonstrates that:

- The per-hop cost for traversing multiple levels of cache is significant. Hence, the cache system should *minimize the number of hops to locate and access data*.
- Due to the network overhead, distant caches may be expensive to access. Hence, *data should be cached close to clients*.

2.2.1 Analysis of a testbed hierarchy

We constructed a testbed to examine the relationship among caches in a large, three-level hierarchy in which a 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. 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 runs version 1.1.17 of Squid. 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. An instrumented client then timed 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 4 represents the mean of the remaining eight measurements.

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 [30, 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 4(a) shows the performance when the testbed uses the standard three level data hierarchy. In contrast, Figure 4(b) shows the access time when the Berkeley client accesses the Berkeley, San Diego, and Austin caches *directly* by circumventing the hierarchy. Figure 4(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.

Figure 4(b) also indicates that even if a cache architecture were able to avoid the cost of multiple-hops, 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 network congestion. 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.

2.2.2 Analysis of Squid hierarchies

Rousskov [36] 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.

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," (the time from when the cache's accept() system call returns until the cache has a parsable HTTP request,) "Disk," (the time it takes to swap in data from the disk,) and "Proxy reply," (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 published the median values during each 20-minute period. Table 3 shows the minimum and maximum values seen for these 20-minute medians between of 8AM and 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

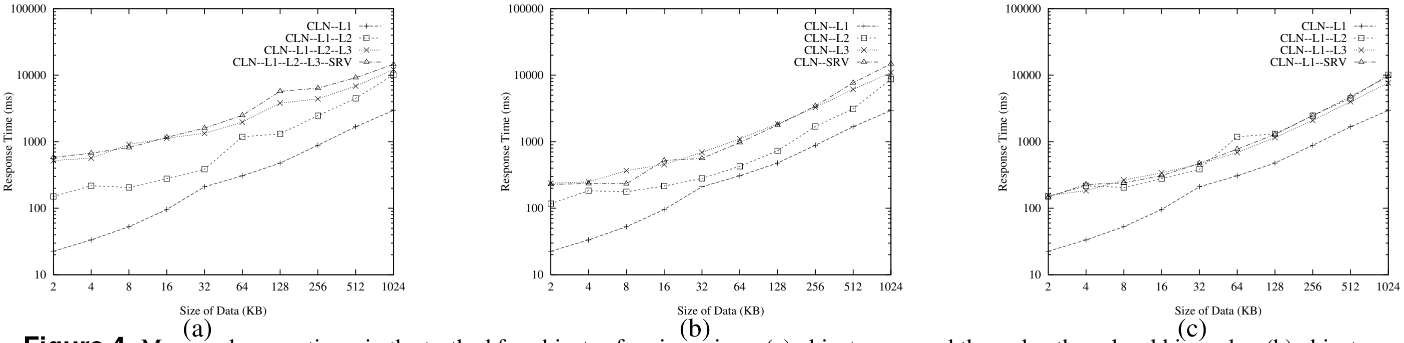


Figure 4: Measured access times in the testbed for objects of various sizes: (a) objects accessed through a three-level hierarchy; (b) objects fetched directly from each cache and server; and (c) all requests go through the L1 proxy and then directly to the proxy or server.

	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	16	62	72	135	75	155	163	352	163	352	163	352
Intermediate	50	550	60	950	70	1050	271	2767	180	2550	271	2767
Root	100	1200	100	650	120	1000	531	4667	320	2850	411	3067
Miss	min:550, max: 3200						981	7217	550	3200	641	3417

Table 3: Summary of Squid cache hierarchy performance based on Rousskov's measurements. All the measurements are in ms.

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.

3 Design overview

In the previous section, we derived three basic design principles for large-scale caches: (1) minimize the number of hops to locate and access data on both hits and misses, (2) share data among many users and scale to many caches, and (3) cache data close to clients. In this section, we describe the strategies we pursue to address these design principles and provide a high-level description of our specific implementation of these strategies. The next section will provide a more detailed, quantitative analysis of the system and our design decisions.

Our system is build around a scalable data-location service called the hint hierarchy which allows each node to know the nearest location of each object. Scalability and performance of the hint hierarchy comes from four sources. First, we use simple, compact data structures to allow each node's view of the hint hierarchy to track the location of many objects. Second, the location system satisfies all on-line requests locally using the hint cache; the system only sends network messages through the hierarchy to propagate information in the background—off the critical path for end-user requests. Third, the hierarchy prunes updates by propagating them only to the affected nodes to support a global index while using minimal bandwidth. Fourth, we adapt Plaxton's algorithm [35] to build a scalable, fault tolerant hierarchy for distributing information.

The hint hierarchy provides the information needed by two basic strategies for providing good performance.

The first basic strategy is *direct access* of remotely cached data. Under the ideal direct access strategy, a client locates the nearest copy of data using an oracle and sends its request directly to the nearest shared cache that contains the desired data or directly to the server if no cache contains the data. Caches and servers that receive such requests send the data directly back to the client. As we illustrated in the previous section, traditional single-level caches can limit sharing while multi-level caches suffer high per-hop costs when they force hits and misses to traverse multiple layers of a hierarchy. This ideal direct access strategy, on the other hand, minimizes the number of hops to locate and access data (principle 1) while allowing widespread sharing (principle 2).

The hint hierarchy allows us to approximate this direct access strategy using *hint caches.* Clusters of nearby clients send their requests to a shared L1 proxy cache. The proxy consults a local hint cache that contains a mapping from the ID of the object being requested to the ID of the nearest proxy cache that contains the object. The first proxy sends the request directly to the second, which sends the data directly back. The client's proxy then sends the data to the client that issued the request.[3]

Our second basic strategy is *push*. As the data in the previous section indicate, even with direct access, the cost of accessing data from a nearby cache is much lower than the cost of accessing data from a distant cache. For example, in the testbed hierarchy L1 cache accesses for 8KB objects are 4.7 times faster than direct accesses to caches that are as far away as L2 caches and 6.2 times faster than direct access to caches that are as far away as L3 caches. The goal of push is to improve hit time by pushing objects near caches that are likely to reference them in the future. The ideal push algorithm postulates that whenever an object is brought into the system, it is magically pushed to all L1 caches in the system that will reference the data in the future; thus all L2 and L3 hits become L1 hits.

We examine several simple, practical approximations to this ideal that are enabled by the information in the hint hierarchy. *Push-on-update* is based on the observation that when an object is modified, the proxies caching the old version of the object are a good list of candidates to reference the new version of the object. Thus, when a cache fetches an object due to a cache consistency miss, it pushes the object to caches storing old versions of it. The *push-shared* algorithm is based on the intuition that if two subtrees of a node in the metadata hierarchy access an item, it is likely that many subtrees in the hierarchy will access the item. Thus, when one proxy fetches data from another, it also pushes the data to a subset of proxies that share a common metadata ancestor. Both of these algorithms are simple, and they achieve a significant fraction of the total performance achieved by the ideal strategy. However, there is room for more sophisticated algorithms to achieve further gains.

4 Detailed design and evaluation

This section details our design and implementation. For each aspect of the design, we examine overall performance and major design decisions, and we discuss areas of potential concern. When appropriate, we use simulation results to examine our decisions.

We have implemented a prototype of our system that implements a hint hierarchy, hint caching, and push caching [25]. It is based on Squid 1.1.20[45] and named Cuttlefish.

4.1 Hint caches and direct access

Local hint caches allow caches to approximate the direct access strategy. A hint is an $\{$objectId, nodeId$\}$ pair where nodeId identifies the closest cache that has a copy of objectId. In order to facilitate the principles of minimizing hops and allowing widespread sharing, the hint cache design allows caches to store large numbers of hints and to access them quickly.

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.

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.

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 returns an error and the first node treats the request as a miss. With 64-bit hash keys based on MD5 signatures of URLs, we do not anticipate that hash collisions will hurt performance. In fact, some hint cache implementors may consider reducing the keys to 32 bits to increase the reach of the cache for a given hint cache size.

Fixed-sized records simplify and speed up access 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.[4]

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.

Our implementation of hint caches approximates the ideal of direct access. However, three aspects of the design may limit its performance. First, we focus on a proxy-based implementation of hint caches rather than a configuration where all clients maintain local hint caches. Second, the hint cache data structure is smaller and slower than the ideal oracle. Third, hint caches may contain an out of date picture of where objects are cached.

[3] An obvious optimization, not currently implemented in our prototype, is to send the object directly back to the client, and then in the background push the data to the client's proxy.

[4] 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.

4.1.1 Overall performance and configuration

We will use the following simulation configuration throughout the paper. As in Figure 2, we assume that when the system is configured as a 3-level hierarchy, clusters of 256 nearby clients share an L1 cache, groups of 2048 clients share an L2 cache, and all clients in the trace share an L3 cache. We parameterize the distance between clients in these categories using the *Testbed* times shown in Figures 4 and the *Min* and *Max* access times measured by Rousskov. Our direct access and hint configurations are similar, but they use the *Total Direct* and *Total via L1* times from those figures. We simulated performance for both infinite-sized caches and finite, 5 GB caches. For the finite cache case, we reserved 10% of the space for hint caches when appropriate. Due to space limitations, unless otherwise noted we display the graph for infinite caches and omit the graphs finite-caches. As Figure 1 suggests, we found that cache size is a secondary factor for performance, and our finite cache results are qualitatively the same as our infinite cache results.

To quantify the overall performance of the system and examine the impact of assuming a proxy-based implementation, Figure 5 shows the simulated performance for the DEC, Berkeley, and Prodigy traces under a series of algorithms. The figure provides two baselines for performance. It includes a traditional 3-level hierarchy and a *best case* L1 cache that is shared by all clients in the system, but that is as close to all clients as the smaller L1 caches used in the standard configuration.

First, note that this configuration is designed to let us explore how different architectures balance sharing, locality, and scaling. As the bars for the *best case* L1 cache show, if the underlying physical network and cache machines allow sharing to be increased without increasing cache access delays, the best configuration is a large, shared cache.

Direct access significantly outperform the traditional 3-level hierarchies with speedups ranging from 1.3 to 2.9. The additional network hop for the proxy-based hint cache configuration hurts performance modestly compared to the ideal system. The realistic implementation achieves speedups of 1.3 to 2.3. For this set of network topologies and workloads, another reasonable alternative is for each group of 256 clients to share a L1 caches but not to share caches further. Such an approach falls short of the ideal direct access protocol by as much as a factor of 1.53 and falls short of the hint cache protocol by as much as a factor of 1.3.

An obvious extension to our proxy-based hint cache implementation is to allow clients to maintain local hint caches and send their requests directly to remote proxy caches. We focus on the proxy-based case because we believe it would make deployment easier by (1) amortizing the cost of hint cache storage across multiple clients and (2) allowing deployment of the new cache technology by modifying a relatively small number of proxies rather than modifying client browsers. In the future, more aggressive implementations of the hint hierarchy could extend hint caches to clients to more closely approximate the ideal forwarding case.

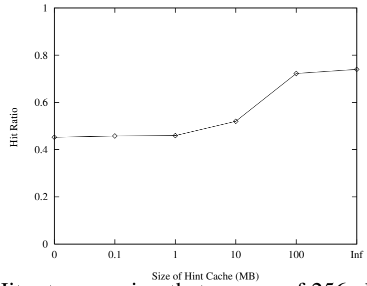


Figure 6: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache of the specified size. Each entry in the hint cache takes 16 bytes and they are stored in a 4-way set associative array with total size specified in MB on the x-axis.

4.1.2 Hint cache size

One obvious worry associated with hints is the storage requirements. How much storage is required to track the nearest copy all objects in a large cache system? This number turns out to be surprisingly reasonable.

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 4.1, 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 cache dedicates 10% of its capacity for hints, 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. Finally consider that in October 1998 a 6 GB disk costs under $160, suggesting that flat indices of hundreds of millions to billions of objects are feasible.

Figure 6 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.

4.1.3 Delayed hint propagation

Although the system uses direct requests and direct data transfers for client requests, it propagates hints through a hier-

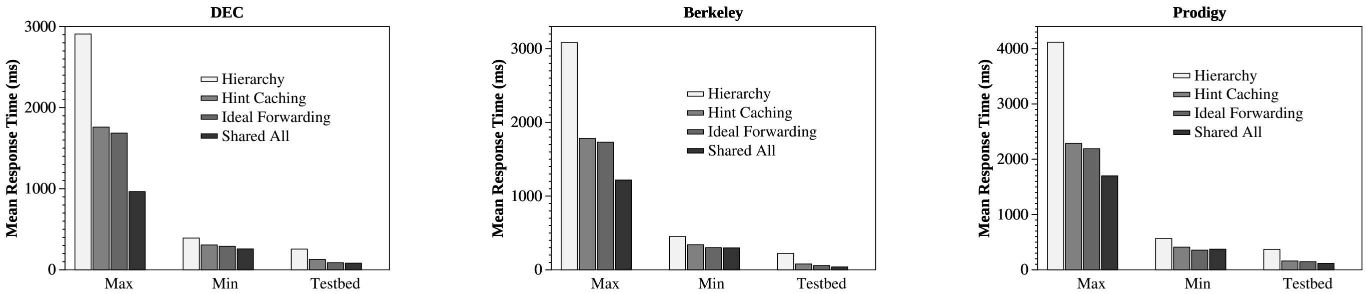


Figure 5: Simulated performance for DEC, Berkeley, and Prodigy traces. The three groups of bars for each trace show the performance when the access times are parameterized by the *Testbed* times shown in Figures 4 or the *Min* and *Max* access times measured by Rousskov shown in the *Total Hierarchical* and *Total via L1* columns of Table 3. Within each group, we show performance for a 3-level hierarchy, idealized forwarding, hint caches, and an idealized cache shared by all clients in the system and as close to each as their normal L1 cache.

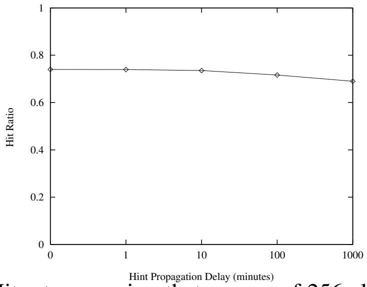


Figure 7: Hit rate assuming that groups of 256 clients from the DEC trace each access an infinite proxy cache and that each proxy cache can access other proxy caches via a hint cache that is updated the specified number of minutes after objects appear or disappear from caches.

archy. We defer detailed discussion of hint propagation until after our description of how the system configures the hierarchy in Section 4.3. Here, we examine the general issue of delayed hint propagation. Whereas the oracle in the idealized direct access protocol always knows where to find the nearest copy of data, actual hint updates take time to propagate through the system, so caches may make suboptimal decisions about where to send their requests.

Figure 7 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. As Section 4.3.2 will detail, to facilitate fast propagation, 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.

4.2 Push

The ideal push algorithm uses future knowledge to redistribute data, thus transforming all L2 and L3 hits into L1 hits.[5] Actual push algorithms must approximate this by sending data they predict will be referenced near caches they predict will reference it.[6]

4.2.1 Algorithms

Because of the large numbers of objects that pass through the system, we focus on simple algorithms that do not explicitly track the reference frequency of individual objects. Examining whether more sophisticated algorithms of this sort would be profitable is a subject of future work.

We first consider *push-on-update*, a simple and 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. Thus, when the cache system fetches an object due to a communication miss, it sends copies of that object to caches that were storing the previous version.

Our prototype implements push-on-update by having nodes track which objects they have supplied to other caches. When a node loads a new version of an object, it forwards the new version to the nodes to which it supplied the old version.

Our second algorithm, *push shared* dynamically builds a distribution tree for popular objects. As Figure 8 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 a copy of that

[5] 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.

[6] We assume that our 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 [26, 28] or server hits [34].

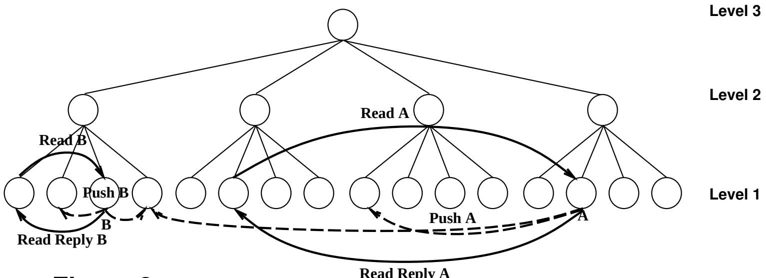


Figure 8: Hierarchical push algorithm for 3-level hierarchy. Once two level-2 subtrees fetch object A, the algorithm pushes object A to all level-2 subtrees. Once two level-1 subtrees under a level-2 subtree fetch object B, the algorithm pushes object B to all level-1 nodes under that level-2 parent.

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. We also examine more aggressive versions of the algorithm, *push half* and *push all* that push multiple copies of an object into different nodes in each subtree. Note that our prototype does not yet implement push shared.

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 ones. 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 they are popular.

4.2.2 Evaluation

Actual push algorithms may fall short of ideal ones in two ways. First, they may fail to push the right data near the right caches, so latency will suffer. Second, they may push data unnecessarily, thereby consuming excess bandwidth. Figures 9 and 10 compare simulations of the ideal push algorithm and base case algorithms without push to push-on-update and push-shared.

Although we do not expect capacity misses to be a major factor in systems that only replicate data on-demand, in push-based systems speculative replication may displace more valuable data from caches. To monitor this effect in our simulations, we report results for the space-constrained configuration in which each of the 64 L1 caches has 5 GB of capacity. Due to time constraints and memory limitations of our simulation machines, these results use only the first seven days of the DEC and Berkeley traces.

Figure 9 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 push-shared 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. Although a large fraction of objects pushed by push-updates are used by their destinations, updates are infrequent so its overall performance gains are small.

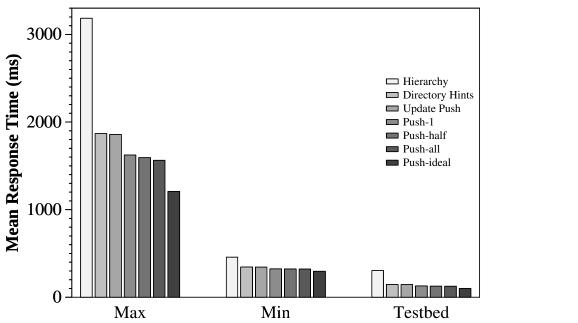


Figure 9: Simulated response time for DEC trace workload for six algorithms: no push (data hierarchy), no push (hint hierarchy), update push, push-1 (send a copy to 1 node in each eligible subtree), push-half (send a copy to half of the nodes in each eligible subtree), and push-all (send a copy to all nodes in an eligible subtree).

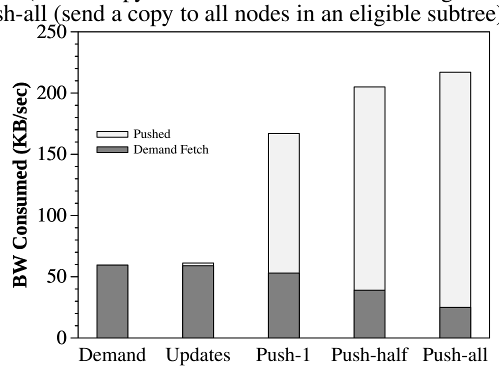


Figure 10: Bandwidth of push algorithms for the DEC trace.

Figure 10 shows the bandwidth consumed by the algorithms. The push-shared algorithms 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.

4.3 Hint hierarchy

The discussion above described how the hint hierarchy uses simple, compact data structures to track the location many objects and how hint caches satisfy on-line requests locally to avoid the network delays encountered when locating data in a traditional hierarchy or using ICP [46]. This section examines two remaining aspects of the hierarchy's design. First, although the hierachy can be visualized as a tree, in reality we use a more scalable data structure to provide scalability and fault tolerance. Second, we detail how the hierarchy propagates hints to prune updates so that updates are propagated only to affected nodes.

4.3.1 Scalable, self-configuring hierarchy

Rather than use a simple tree where nodes near the root become performance and reliability bottlenecks for scalability, the hint distribution hierarchy adapts an algorithm developed by Plaxton et. al [35] to embed virtual hint distribution trees onto the nodes in the system. The key idea in the algorithm is to assign pseudo-random IDs to all nodes and objects based on their IP address or URL. The system then logically constructs a tree for each object in which the root node matches the object in the most bits of any node and lower layers match in successively fewer bits. This approach balances load because each node acts as a leaf or low-level node for many objects (but has relatively few requests for such objects) and acts as a root or high level node for few objects (but gets relatively more requests for each of these). It achieves locality because near the leaves of virtual trees, the distance between parents and children tends to be small: since a large fraction of the nodes in the system can act as low level parents for an object, low-level children can choose nearby parents. Of course, near the roots, the distance is generally larger.

Each node in our system runs this algorithm independently using information from a membership module, which tracks which nodes have joined the system, and a configuration module based on the standard Squid `net_db` database, which tracks which nodes are alive and how far away they are. Nodes also monitor messages from other nodes to detect and correct inconsistencies in their views of the hierarchy. A node executes the reconfiguration algorithm whenever the membership or connectivity layer notifies the hierarchy of a significant change.

4.3.2 Hint propagation

Our system propagates hint updates through the hierarchy both to distribute load for different objects across different nodes and to reduce the amount of information sent globally. Whenever a cache loads a new object (or discards a previously cached one), it informs its parent. The parent propagates that information to its children or its parent or both using a limited flooding algorithm in which nodes only propagate changes relating to the nearest copies of data. For example, if a node already knows that one of its children has a copy of an object, then on receiving a hint from its parent about a new copy of the object, the node does not propagate the hint to its children. This is because, assuming that the hierarchy's topology reflects network locality, propagating the hint will not alter the information regarding the nearest copy of the object with respect to any of its children nodes.

Table 4 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.

Organization	Average	Peak	99%	90%
Centralized directory	4.8	14.4	10.2	7.4
Hierarchy (root)	2.1	9.8	6.2	4.0
Plaxton hierarchy (max-node)	0.15	2.4	1.1	0.6

Table 4: Number of location-hint updates sent to the root during the first seven days of the DEC trace. The columns show the average, peak, and 99th and 90th percentiles of requests per second over 10-second periods. All cache updates are sent to the centralized directory; the hierarchy line shows load when the hint hierarchy prunes requests but the system still uses a single root; the Plaxton hierarchy line shows load on the most heavily loaded 64 nodes, which acts as a root for about 1/64th of the objects and also acts at different levels of the hierarchy for other objects.

One potential danger 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 decide that is a better place to go for that data. In essence, the system dynamically builds a distribution tree for popular data. Using a hierarchical metadata 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.

5 Related work

Several Internet 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 wide area networks, the WAN-xFS proposal [10] uses a hierarchy of metadata nodes to improve scalability, and the CRISP cache [18] 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 [17]. Hints have also been examined for reducing coordination in LAN cooperative cache systems [38].

Two other systems index remote cache contents using Bloom filters [15, 1]. The primary difference between the Bloom filter systems and ours is our use of a scalable hierarchy to distribute location information. In contrast, the Bloom filter sytems broadcast or multicast updates to all copies of the index. Our distribution hierarchy makes it feasible to share data among a larger number of caches and also provides useful signals for our push algorithms. The specific data structure used to store location information is an implementation detail, and the choice may depend on the scale of the system being considered. Bloom filters provide a compact way to represent the contents of a single cache, but their total size grows linearly with the number of caches indexed times the number of objects in each cache. Hint cache hash tables start off larger, but grow with the number of *unique* items in the system.

Several systems, including the Internet Cache Protocol (ICP) [46] and Zhang et. al's adaptive caching proposal [48], replace directories with multicast queries to nearby caches. 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.

Several efforts to provide caching that is closely tied to network topology have focused on self configuration [4, 48], which is a core doal of our dynamic hierarchy.

Several studies have examined Internet workloads in depth with the goal of understanding how to improve performance [2, 6, 14, 21]. 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 [36] and Maltzahn et. al [30] 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 [8]. 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 [5, 37, 20]. Muntz and Honeyman [32] 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.

Several researchers have proposed improving the scalability of a data hierarchy by splitting responsibilities according to a hash function [12, 44]. 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 [22, 23, 34]. 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 [27] 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. To address these systems, he have constructed a hint hierarchy that supports direct access and push. Overall, our techniques provide speedups of 1.27 to 2.43 compared to a traditional cache hierarchy.

References

[1] A. Rousskov and D. Wessels. Cache Digests. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998.

[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] S. Bhattacharjee, K. Calvert, and E. Zegura. Self-organizing wide area network caches. Technical Report GIT-CC-97/31, Georgia Tech, 1997.

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

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. On the Implications of Zipf's Law for Web Caching. Technical Report 1371, University of Wisconsin, April 1998.

[7] P. Cao, J. Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proceedings of Middleware 98*, 1998.

[8] 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.

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

[10] 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.

[11] 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.

[12] P. Danzig. NetCache Architecture and Deployment. Technical Report TR3029, Network Appliance, Inc., 1998.

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

[14] 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.

[15] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1998.

[16] 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 ACM-Symposium on Operating Systems Principles*, pages 201–212, December 1995.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

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

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

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

[25] J. Kay, M. Dahlin, R. Tewari, and H. Vin. Introducing the Pushcache. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998.

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

[27] 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.

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

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

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

[37] 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.

[38] P. Sarkar and J. Hartman. Efficient Cooperative Caching using Hints. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 35–46, October 1996.

[39] 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.

[40] 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.

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

[42] A. Vahdat, T. Anderson, and M. Dahlin. "active naming: Programmable location and transport of wide-area resources". http://www.cs.utexas.edu/users/less/publications/research/anamedraft.Aug98 August 1998.

[43] 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.

[44] 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.

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

[46] 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.

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

[48] 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/.