Hash-Based Virtual Hierarchies for Caching in Hybrid Content-Delivery Networks*

Zizhen Yao, Chinya V. Ravishankar, and Satish K. Tripathi
Department of Computer Science and Engineering
The University of California, Riverside 92507
{yzizhen,ravi,tripathi}@cs.ucr.edu

May 13, 2001

Abstract

Two classes of strategies are in use for improving information access performance on the Web. The first approach is to tailor the information dissemination strategy to application semantics, and to select either push or pull as the strategy for data delivery. The second approach is to enhance caching performance through cooperative caching methods. Hash-based object allocation and the use of cache hierarchies are examples of this second approach.

We propose a novel cooperative caching strategy for supporting a hybrid of the push and pull models, based on a virtual hierarchy of caches. The virtual hierarchy is defined by hashing on the object and cache names, so that a distinct virtual hierarchy is created for each object. Unlike static hierarchies, where the root can become a bottleneck, our approach creates a different hierarchy and root for each object, so that loads are evenly balanced. Clients can determine the hierarchy for each object locally using a hierarchy of hashing functions.

Our strategy allows the use of push to maintain strong consistency control for popular and frequently-changed objects, while pulling less popular objects on demand. This paper presents the details of the approach, as well as analytical and experimental studies of its performance.

Keywords: Proxy hierarchy, consistent hashing, cooperative caching.

1 Introduction

WWW activity accounts for the bulk of the traffic on the Internet, since much of the information content of the Internet is available as objects on the world-wide web. Therefore, any strategy that reduces

*This work was supported in part by grants from Tata Consulting Services, Inc., and the Digital Media Innovations Program of the University of California.

web traffic will also relieve Internet congestion significantly. Since Internet browsers are becoming the primary mechanism for information access, reducing client latency is also an important goal.

A common approach is to use proxy caches to address both problems. Proxies attempt to service requests locally, and reduce the need to go directly to servers. Each proxy is also typically assigned to a group of clients, so that it can aggregate their request streams, thereby improving hit rates. It is possible to carry this idea further, and organize several proxies into a group. In principle, proxies in a group can serve each other, and cooperate to further improve hit rates. However, for a proxy cluster to be effective, the overhead of determining which proxy has a given object must be minimal.

The use of hashing for the management of proxy clusters was first introduced in [13, 16]. This method offers several advantages; in particular, it eliminates the overhead of maintaining explicit object-proxy mappings, causes no replication of objects across proxies, and makes reconfiguration after failures quite straightforward. A related idea was described subsequently in [14]. The work in [13, 16] was extended in [15] to static physical hierarchies. In this paper, we show how to construct virtual hierarchies by using the ideas in [13, 16]. Such virtual hierarchies offer several advantages, particularly load balancing.

1.1 Effects of server behavior

The performance of proxies depends not simply on client access patterns, but also on server behavior. The two general approaches to the dissemination of information on the WWW are the "push" and "pull" strategies. In the push approach, the information source (the "server") pro-actively sends the latest version of each object to all subscribers. In the pull approach, the server remains passive, and each client retrieves objects as needed from the server.

Caching has been used with both push and pull to reduce network traffic and client access latency. Under the push model, the cache is refreshed at each push, so the client always sees the latest version in the cache. Under the pull model, the cached copy must be explicitly validated. The push and pull approaches have their respective benefits and drawbacks, and are suitable for use in different contexts. Our goal in this paper is to present a caching strategy that helps to combine these two approaches.

Because the push model is perceived as carrying a risk of high overhead, it is often implemented in practice as a pull rather than a true push [10]. Since clients typically require the latest versions of objects, overhead is highest for popular but frequently-modified objects. A recent study of a busy web site [1] has found that modified objects account for over half of the repeated accesses from the same domain. Most current web caching systems address this problem through a weak consistency scheme that associates a TTL (Time-To-Live) flag with each object in the cache. A cached object is considered up-to-date if its TTL has not expired. Otherwise, clients validate the freshness of a cached copy by examining the server's response to an **if-not-modified** query that specifies the object in question. The server returns the current version of the object if the cache is out of date, or a not-modified response otherwise.

This approach is unsatisfactory for at least two reasons. First, clients may still receive stale copies since objects may have been modified at the server before their TTL has expired, unbeknownst to the cache or the client. Second, the validation message exchanges can cause significant overhead. For example, it is

shown in [1] that 37% of the validation requests in a busy web site

studied invoke the server response **not-modified**. This overhead might be absent if the cache used a stronger consistency model.

Our own study of proxy traces indicates that a protocol using **not-modified** responses reduces latency at the client only by 15%–25% over a simpler protocol where the server simply returns the requested object each time.

The work in [1] also reports that the total number of modifications is several orders of magnitude smaller than the total number of accesses, and that the changes between successive versions of objects are often very small. These observations suggest that a more efficient consistency protocol could have significant benefits.

1.1.1 Characteristics of Push and Pull

The pull model is conservative. Its chief advantage is that objects are fetched on demand, so that work is never wasted. This model is widely deployed in current web cache architectures, even when the high-level appearance is that of a push [10]. Its chief drawback is that clients may see a high latency since objects may have to be fetched from server repeatedly. In addition, cache validation may be needed. From the server's point of view, requests are not clustered, and must be processed separately even if they request the same object. Thus, pull requests for a hot object can generate heavy server loads, and even cause congestion at server.

In contrast, the push strategy is optimistic and aggressive, and delivers objects in anticipation of usage. Objects are therefore available immediately on request. However, if the pushed objects are not used before they are outdated, network bandwidth as well as processing time at server and cache is wasted. System performance can be severely degraded due to network congestion and processor over-utilization.

Push can reduce the server workload considerably. When an object is modified, the server can send corresponding updates to all potential clients in one series of operations, and not have to be disturbed by repeated requests for those objects, or by cache validation requests. Bandwidth is conserved as well if the pushed content will be referenced in future. In this case, no **if-modified-since (IMS)** requests or **not-modified** responses are needed, and updates can be compressed efficiently using delta encoding [20]. The size of transferred data can be therefore reduced as much as an order of magnitude [20, 1].

For less popular objects, however, pulls are far more effective. There is clearly a tradeoff between push and pull, since neither works well in all situations. Fortunately, they do complement each other, since when one behaves badly, the other strategy shows satisfactory performance. Push is more appropriate for popular and frequently-modified objects, since a popular page is likely to remain popular even after modification. The prediction of future usage for such objects can be easy and accurate. Since no cache validation is needed for push, a significant improvement of performance can result from the elimination of validation protocol messages for these objects. In our analysis of proxy trace data from NLANR [1], we find that such validation from server is very expensive. The average latency perceived by the client when

[1] We use the web traces from ftp://ftp.ircache.net/Traces/ for 01/15/2001

the cache is valid is only 15% to 25% lower than the latency when the server returns the entire object.

The characteristics of the push model require us to carefully consider the following scalability issues when the numbers of clients and objects increase.

Membership Management overhead: Clients are both passive and unaware of updates in a push scheme, so the server keeps track of all sites to which updates must be pushed. Such client site lists must be maintained on a per-document basis. Since the numbers of clients and documents grow with time, these lists can eventually become unmanageable. This problem may be addressed by grouping the clients or clustering documents, thereby increasing the granularity of list entries, and reducing the size and number of lists, respectively.

Propagation Delay and Bandwidth Consumption: Without underlying multicast support, the server must to send the data to each of its clients one by one, which can be very expensive when the number of clients is large. What is more, the paths to push the content to clients located close to each other may overlap. In such a case, bandwidth is wasted since the same data are sent repeatedly over the same link.

1.2 Our goals and approach

In general, caching efficiency depends on how well we can predict which objects are likely to be fetched in future by whom, and how soon they are likely to be modified. Such predictions are hard to make, since the web is highly dynamic, and both the popularity and modification rates of objects vary significantly with time. Since not all clients need every object, it is inefficient to push all the objects to caches near the clients.

In practice, recent access history still provides a wealth of information for predictions of the near future. One strategy would be to apply the techniques used by clients for prefetching [22, 23]. However, this approach may be less than ideal in our case, since servers tend to have more information than do clients. For example, servers know the sizes of objects, the embedded components for html files, the history of accesses over a big population of clients and objects, and exact times of modification. Prefetching strategies tend to use none of this information.

We propose a novel hybrid push-pull model based on a virtual hierarchy of caches defined by a hashing scheme. Our strategy is to use push to maintain strong consistency control for popular and frequently-changed objects, while pulling less popular objects on demand. A push model can actually be more efficient than a pull model if the access rate is high compared to the change rate, and updates are small compared to original object. Since validation is not needed, both the latency perceived by clients and network traffic can be reduced. In addition, the recency of the pages users get is guaranteed.

The hierarchy is defined by hashing on the object and cache names as in [13, 16], so that a distinct virtual hierarchy is created for each object. In approaches which use static proxy hierarchies [15], the root can become a bottleneck. In contrast, our approach assigns a different root to each object, and given the structure of the skeleton, clients are able to determine the hierarchy for each object locally using our

hierarchy of hashing functions.

Major advantages of such schemes are load balancing and fault tolerance. We will discuss some these issues in detail in Section 3. This approach reduces client latency and network bandwidth by addressing three subordinate objectives: to increase the hit ratios for web caching, to alleviate congestion by balancing workloads among proxy caches and servers, and to reduce the number of **if-modified** requests while still maintaining the freshness of cached objects.

Our scheme can also be viewed as an architecture for a Content Delivery Network (**CDN**), intended to enable content providers distribute content close to the end user. CDNs must address network bandwidth availability, latency issues, and content server scalability and traffic congestion issues during peak usage periods. Our hash-based virtual hierarchy scheme provides an efficient solution for content distribution, routing and management, that accelerates web accesses as well as relieves the congestion at the original servers.

The rest of this paper is organized as follows. In Section 2, we discuss some recent related work. In Section 3, we introduce our hash-based hierarchical architecture and the hybrid push and pull model based on it. Section 5, presents an analytical model for of our scheme and its performance. In Section 6, we present some simulation results based on synthetic data. We summarize our results discuss our future work in Section 7.

2 Related Work

Much work has been underway recently in various related areas, including web caching, web cache consistency, and content delivery networks. Web caching has been well studied to reduce clients perceived latency. However, individual web caches have achieved only moderate hit ratios due to limited temporal locality. Though cooperative caching methods have been studied to address this problem [6, 7, 8, 9, 11, 13], it is observed in [2] that such schemes have performance benefits only within limited clients' population bounds. Scaling beyond the bound provides only minor improvement. Other techniques reported in the literature include prefetching [22, 23], and reverse caching.

Prefetching approaches generally rely on the proxy to predict which documents a user might reference next, and exploit the idle time between user requests to push or pull the documents to the user. Prefetching can be very effective in reducing latency of clients with slow modem connections. On the other hand, when servers become the bottleneck, an approach called reverse proxy caching is quite effective. In this method, caches are deployed near the server to handle some of its workload. Requests made to the server are distributed among these caches to ensure a high level of quality of service.

Broadcast disks [26] have also been proposed as a scalable solution to the problem of a single source having to serve an enormous number of clients. It is a classic example of the push model. Servers periodically broadcast data to large population of clients who listen on the broadcast medium and retrieve information as needed. However, access latency can be as high as a full broadcast cycle if the object of interest has just been broadcast. To reduce such high latency, workers have suggested using caching and

prefetching at clients [27] and optimizing broadcast schedules at the server [28].

A recent study of workload characterization of a busy web site [1] pointed out that over half of the repeated accesses on a given day were to objects modified that day.

Such frequently-modified objects must be frequently refreshed, suggesting that more efficient consistency models can be very beneficial. It has also been suggested that server-based invalidation can be an efficient means of ensuring strong consistency, especially compared to client polling on every read [21]. Another study [24] describes a adaptive TTL approach that adjust a document's time-to-live based on observations of its lifetime. Adaptive TTL exploits the fact that file lifetime distributions tend to be bi-modal, so that if a file has not been changed for a long time, it tends to stay unchanged. Studies [24, 25], have shown that adaptive TTL can keep the probability of stale documents within reasonable bounds. Though such approaches improve or ensure the freshness of the cached objects, they can not decrease the latencies of access to modified objects.

2.1 Hash-based Allocation of Objects to Caches

The use of hashing for the management of proxy clusters was first introduced in [13, 16]. To locate an object in a cluster of caches, the cluster members are treated as buckets in a hash table, and one of them selected using a hash function. The novelty of the work in [13, 16] is that the hash value is computed on a combination of the cache name and the object name, rather than on the object's name itself. If there are n caches in the cluster, the n resulting hash values are computed, and the cache yielding the highest hash value is selected. Since all clients use the same hash function, they all effectively agree on object-cache mappings.

This method offers several advantages; in particular, it eliminates the overhead of maintaining explicit object-proxy mappings, causes no replication of objects across proxies, and makes reconfiguration after failures quite straightforward. A related idea was described subsequently in [14]. The work in [13, 16] was extended in [15] to static physical hierarchies. In this paper, we show how to construct virtual hierarchies by using the ideas in [13, 16]. Such virtual hierarchies offer several advantages, particularly load balancing.

This technique is referred to as Highest Random Weight (HRW) mapping. A related technique was subsequently described in [14]. If the hash function is well chosen, objects will hash with equal probability to all caches, so that the clients effectively see a single large cache n times as large as a single cache. Since the hashing takes all clients consistently to the same cache for any given object, we also achieve request aggregation. The major advantages of hash-based schemes are low overhead (no or little inter-cache communication), and high hit rates due to efficient utilization of cache space.

HRW is also known to be robust, and causes minimal disruption. Whenever a cache comes up or goes down in a hash-based scheme, objects may need to be re-mapped to preserve the integrity of the method. In a badly-designed hash routing scheme, the fraction of objects that are in incorrect caches can be large. This fraction is referred as the disruption coefficient in [16, 13]. For simple hash functions based on modulo arithmetic, the disruption coefficient is close to unity. In contrast, HRW is fault tolerant. When any cache goes down, the mapping of the objects in the other caches will not change. The objects cached in the

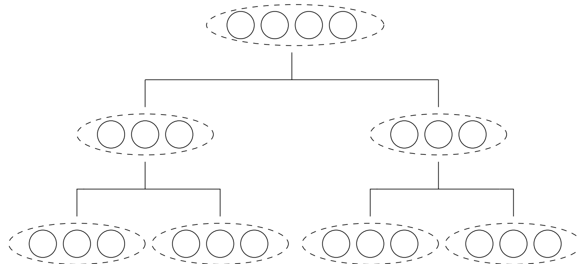


Figure 1: Physical hierarchy

failing cache are re-mapped into the cache with the second highest HRW score. If the score function is well designed, these objects will be re-distributed to the remaining caches evenly, maintaining the nice property of load balancing.

3 Our Proposed Approach

The following insight motivates our scheme. While hierarchies can raise hit rates by aggregating requests up the hierarchy, such aggregation is useful only on a per-object basis. In other words, when a request for object O_i arrives at a proxy p in a hierarchy, the only beneficial component of the history at p is the earlier stream of requests for O_i. It is not helpful that requests for other objects O_j had also been serviced by c. In fact, from O_i's point of view, requests for these other O_j consume processor, memory, and bandwidth resources, and are undesirable.

Statically defined hierarchies aggregate requests for all objects as one proceeds up the hierarchy, and are therefore less than ideal from this perspective. In particular, the Cache Array architecture described in [15] (see Figure 3) extends the work in [16, 13] to physical cache hierarchies. A cluster of proxies exists at each interior node of a tree of caches, and each proxy serves only a portion of requests coming into the node. However, when hit ratios are low, and the number of children at each interior node is large, the number of requests handled by the cache array at a node can increase geometrically as we go up the hierarchy. In particular, the root is likely to become a bottleneck since all requests to the server are routed through it. For instance, if the hit ratio is 50% at some node, and its parent has ten children, then the workload at its parent is five times of the workload at each children. Therefore, either the number or the processing power of proxies at upstream interior nodes must be increased to deal with the aggregated traffic from lower level.

As explained below, the virtual hierarchy approach defines a different hierarchy for each object, resulting in uniformity of traffic and processing workloads across the nodes. In particular, each node is equally likely to serve as root for the same number of objects. No single node is forced to serve as root for the entire hierarchy.

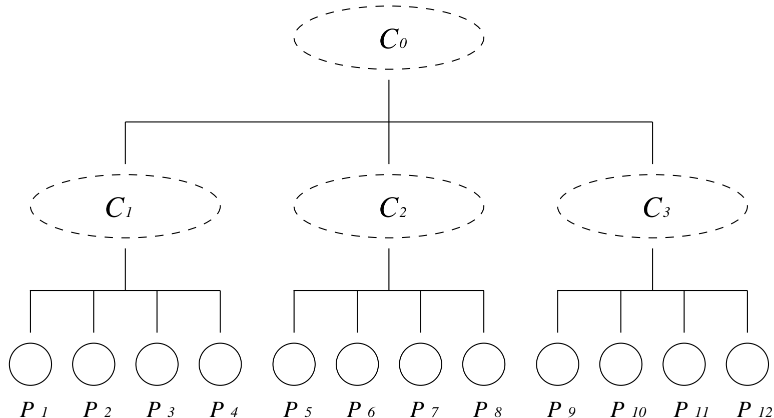


Figure 2: Skeleton

3.1 Skeletons and Virtual Hierarchies

Our scheme is best understood as comprising two phases: a preliminary *skeleton construction* phase and an access-time *virtual hierarchy construction* phase. Participating nodes are first organized into a tree-structured *skeleton*, by hierarchically clustering nodes according to some physically or logically defined metric. Some obviously relevant physical metrics are the hop count, the latency, and the bandwidth of data transfer between nodes. However, one may also use logical metrics, grouping nodes, for example, according how similar their object access patterns are. Any metric is reasonable if it can improve hit rates and result in significant performance improvements. Figure 3 depicts a skeleton constructed over the proxy caches $P_1, P_2, \cdots, P_{12}$. The caches themselves are at the leaf level, while the internal nodes of the skeleton are virtual nodes (or clusters) such as C_1, C_2, and C_3. The skeleton is rooted at cluster C_0, which includes the subclusters, C_1, C_2, and C_3. Proxies within a cluster are closer to each other under the skeleton's metric than to proxies in other clusters.

Once such a skeleton has been defined, object-specific *virtual hierarchies* may be constructed using *hierarchical hashing* by applying a hashing technique like HRW (see Section 2.1) at each cluster. A hash function H_{C_i} is first defined for each cluster C_i in the skeleton, and object-specific virtual hierarchies are constructed on top of skeletons through these hash functions. Given an object O, hierarchical hashing defines a path in the skeleton within which O will be found, provided it exists in one of the caches in the skeleton.

hierarchical hashing is used To see how hierarchical hashing is used locate an object O, let us proceed down the skeleton from the root, at each step, applying hash function H_{C_i} to O at cluster C_i to select one of C_i's children. This selected child is called the "prime" for O at C_i. As we will see, hashing ultimately resolves each such prime to a real leaf-level cache $\Pi_{C_i,O}$, which serves all requests for O forwarded from C_i's children. Since at the upper levels of the skeleton, the selected child is always a cluster C_j, the process is repeated at C_j using H_{C_j}. We thus descend down the skeleton until we finally select a leaf cache c, which then is designated to be $\Pi_{C_i,O}$. Primes at each cluster are object-specific, so this process results in

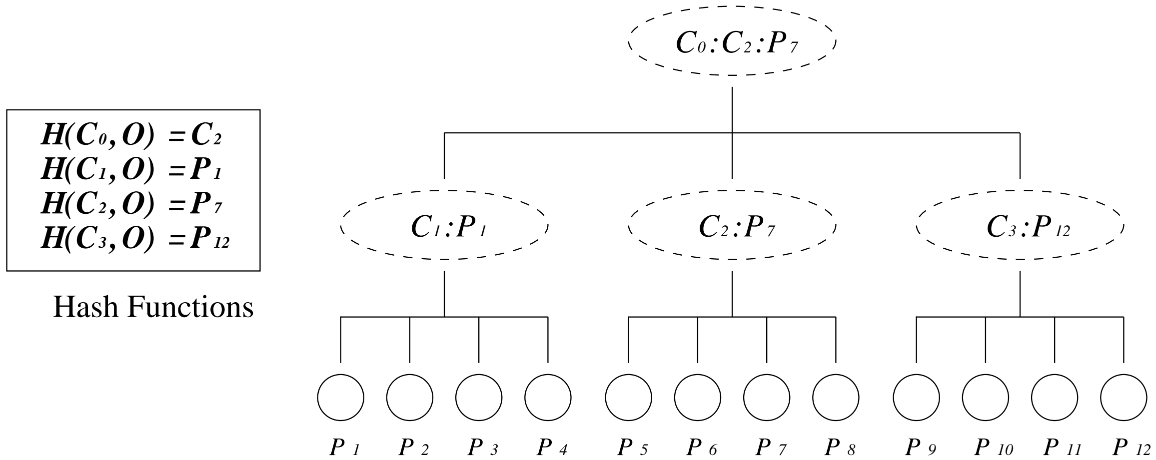


Figure 3: Virtual Hierarchy

an object-specific virtual hierarchy. Each cache miss results in a request being forwarded to the parent in the virtual hierarchy for the object. Primes at the root cluster are referred to as "root" proxies.

To illustrate this idea, consider Figure 3.1. Given object O, we start at the root cluster C_0 and compute $H_{C_0}(O)$, picking C_2. We now apply the hash function H_{C_2} to O, and select proxy P_7. P_7 is therefore the prime for O at C_0 and C_2. Similarly, P_1 and P_{12} are picked as the primes for O at C_1 and C_3 by applying their respective hash functions. The virtual hierarchy for O is therefore obtained as follows: P_7 at the root, P_1 and P_{12} as P_7's children, and the other proxies as leaves.

3.2 Characteristics of Virtual Hierarchies

Several properties distinguish virtual hierarchies from static hierarchies. First, virtual hierarchies decouple the physical organization of nodes from their logical organization. Our scheme creates a different hierarchy for each object, so that one may expect as many logical hierarchies overlaying the physical organization of the caches as there are objects.

Since each node sees the aggregate effect of all these logical hierarchies, the traffic and processing loads become evenly distributed across the nodes in the hierarchy. In particular, each node is equally likely to serve as root for the same number of objects.

Unlike the approach in [15], there are also no specialized "interior" nodes in our approach. The skeleton constructed in the first phase is a conceptual hierarchical organization of caches. Nodes at higher levels in the skeleton are simply groups of groups. Real nodes in our virtual hierarchies are really leaf-level caches, and can be identical to each other.

In addition, each object O_i is stored in each cluster only at its prime cache, so only one copy of each object exists at the lowest level cluster. In non-hash-based architectures, objects would quickly become replicated in all proxies [13, 16]. Although objects may be copied from high level primes to lower level primes, we can ensure that only popular objects are replicated by choosing a popularity-based cache

replacement policy, such as LFU. Our virtual hierarchy approach is thus far much effective in aggregating requests from lower levels.

The structure of the skeleton and the hashing functions can both be stored in a name server, and retrieved by participating nodes. Name server lookups can be quite fast, but the hierarchy information can also be cached locally at the nodes to reduce name server lookups. Such caching is effective since we do not expect that the skeleton will need to be reorganized whenever a node fails. Our virtual hierarchy method for object access uses a hashing method [13, 16] that is resilient to node failures. The skeleton can be generated either manually, or automatically using a distributed hierarchy construction algorithm such as [30].

4 A Push-Pull Hybrid Using Virtual Hierarchies

Virtual hierarchies lead naturally to an efficient hybrid push-pull content delivery system that functions as a reverse cache closer to servers and as a cooperative cache closer to clients. Pulls proceed up the hierarchy, as primes pull objects from higher-level primes, while pushes propagate down the hierarchy as primes push objects to lower-level primes. Servers push to root primes, and clients pull from leaf-level primes.

Typically, we would expect an object to be pushed from the root to a certain level in the skeleton, and for clients to pull the object down from that level. The upper level primes therefore form a content delivery network and reverse caching system for servers, while the lower primes act as a general cooperative caching system. We can tune the level to which objects are pushed according to object access and modification patterns and network conditions. Pushing an object farther down the hierarchy makes it possible for client requests to be served with lower latency, at the risk of wasted work when a pushed object is not accessed before it is pushed again. Pulling the object from higher up in the hierarchy can result in higher latencies, but minimizes the chances of wasted work. We will discuss this fundamental tradeoff in detail in Section 4.1.

Physical hierarchies do not support push-pull hybrids well since they aggregate overhead at upstream caches. Virtual hierarchies are better suited for this purpose since each proxy sees an equal share of the workload. A virtual hierarchy can also be regarded as an application-level multicast tree, in which pushed data traverse a tree edge only once, if the edge is on the paths to multiple receivers. The virtual hierarchy design helps to avoid a single, or a small number of rendezvous points that are likely to cause congestion. Our delivery tree is object specific, and can be computed on the fly for each data delivery instance. Content routing in this case is straightforward.

Hashing schemes such as HRW contribute robustness and load balancing, while hierarchies improve scalability. Thus, hash-based schemes and hierarchies can complement each other. In a large network, the distance between proxies is likely to be non-uniform under any chosen metric. Therefore, the average distance between any two proxies could be very high. However, all proxies are treated identically in hash-based schemes, so that the average inter-cache communication latency is the average latency between all

pairs of proxies. The use of hierarchy allows the exploitation of locality to decrease the average distance between proxies. Thus, it makes hashing scalable.

On the other hand, a hierarchical structure can increase the number of hops on the physical communication path, and add save-and-forward delays at each hop. The net effect might increase the average response time. This tradeoff depends on inter-proxy distances, similarity of access patterns across proxies, and processing overhead at proxies. Thus, neither a flat structure nor a deep hierarchy seems to be particularly efficient.

4.1 A Subscription System for Pushed Objects

To develop the push-pull hybrid idea further, we propose a system of subscriptions, in which an object O is pushed from a node to every child that subscribes to O. As discussed in Section 1.1.1, it is generally preferable to push more frequently accessed objects and pull less frequently accessed ones. We propose using the ratio of the access rate to the change rate (denoted by γ) to decide whether to push or pull an object. We subscribe to all objects whose γ values are above a given threshold, and pull the others. This push/pull decision is made separately for each object-cache pair. We can approximate the access rate and modification rate at a cache by the reference count and update counts, respectively. References and updates are counted over a period of time called the subscription interval.

Pushed objects are naturally always up to date. For pulled objects, we have considered three consistency approaches: adaptive TTL, polling, and invalidation. Adaptive TTL provides weaker consistency than the others. We have chosen invalidation as our method for two reasons. First, it has been pointed out in [21] that strong cache consistency can be maintained for the Web using invalidation at little extra cost. Second, invalidation can be easily incorporated into our system with server support for pushing.

Three types of request messages are sent in the direction of pull (bottom-up): **object-request**, **subscribe** and **unsubscribe**. Two types of messages are sent in the direction of push (top-down): **update** and **invalidate**. **Object-request** messages are triggered by upward propagation of requests from clients. **Update** and **invalidate** messages are triggered by the downward propagation of object modification messages from the server. Other messages are generated by the web caching system internally.

Figure 4.1 defines the operations performed in our scheme upon arrival of each type of message. We do not address cache replacement policies here. However, to ensure an unbroken path for delivery of updates, we require objects to remain in proxies as long there are subscribing lower-level primes. When an subscribed object must be evicted from a proxy, an explicit **unsubscribe** request must be sent to the higher-level prime.

4.2 Handling Congestion and Failures

Though our hash-based design balances workloads well, congestion may still occur because of bursts of requests for hot objects, or because of increases in activity within certain clusters. In addition, node and link failure are always possible. When outages occur, we must find uncongested and available paths to objects, not simply the shortest paths.

- An **update** for O arrives from above:

  1. Forward the **update** to subscribing lower-level primes.
  2. Send **invalidate** to non-subscribing lower-level primes which have accessed O since last modification.
  3. If subscription interval has expired:
     - Clear reference and update counts if $\gamma >$ threshold, or if subscribing lower-level primes exist. Send a **unsubscribe** request to higher-level prime otherwise.
  4. If subscription interval has not expired, increment the **update** counter.

- An **invalidate** for O arrives from below:

  1. Invalidate locally cached copy.
  2. Send **invalidate** to lower-level primes which have accessed O since its last modification.

- An **object-request** for O arrives from below:

  1. Return a copy if O is cached, else forward **object-request** to higher-level prime.
  2. Increment reference count.
  3. Subscribe to O if O is not subscribed to and γ reaches the threshold.

- A **subscribe** arrives from below:

  1. Add the sender to subscription list for O.
  2. Forward **subscribe** to higher-level prime if no subscription for O is in effect.

- An **Unsubscribe** arrives from below:

  1. Remove sender from subscription list for O.
  2. If subscription list is empty, send **unsubscribe** request to higher level prime.

Figure 4: Operation of Hybrid Push-Pull Scheme

In our system, redirection of requests in such cases can be done elegantly by simply re-mapping the requests. When a request to a cache times out, the requester forwards the request to the next cache in the ordering defined by the hash function [13]. The unreachable proxy is marked as inactive, so that future requests mapping to it can be similarly redirected without waiting for timeout. The congested cache can be reactivated after a certain interval. This interval can be determined by general back-off technique, such as an exponential back-off algorithm.

Congestion, network failure, or proxy failure can also cause proxies to return stale objects, since update or invalidation messages may be lost. In such cases, a proxy may not even be aware of the congestion at higher-level primes. To avoid this situation, a cache must poll higher-level primes when there is no communication between the two for a long period of time. If the polled proxy does not respond, all cached objects received from it are invalidated locally and at lower-level primes.

5 An Analytical Model of Push and Pull

We present a formal model to predict the behavior of proxy caches in our push-pull hybrid scheme. Our model makes several assumptions:

- All objects are cacheable, equally popular, and of the same size.
- The time between requests and updates for object O_i are exponentially distributed with parameters λ_i and μ_i, respectively, at all caches.
- All caches have unbounded size.
- The response time is determined by the queueing and service times at each cache, and propagation delays between caches. Propagation delay is constant.
- The hierarchy is homogeneous, the clusters at the same level of the hierarchy are of equal size, and each cluster contains the same number of subclusters (or caches).
- Primes receive requests directly from clients, without the mediation of local proxies.

In the following analysis, we first ignore consistency control overhead. In Section 5.5, we will discuss the effect of invalidation in detail.

5.1 Inbound traffic

We first consider the inbound traffic pattern in our hybrid hash scheme. We assign levels in the skeleton bottom up, i.e. each individual cache is at level 0, and the root cluster at level L, if L is the depth of the skeleton.

Let d^l be the fan-out at level l. The aggregated request rate for object O_i at level l is now $d^l\lambda_i$, so the miss ratio can be computed as $\mu_i/(\mu_i + d^l\lambda_i)$. The number of outbound messages due to cache misses therefore becomes

$$m_{i,l} = d^l\lambda_i \frac{\mu_i}{\mu_i + d^l\lambda_i}$$

Since our hierarchies are virtual and dynamic, caches are not statically assigned to a level in the hierarchy. A cache may function as a level-l prime for one group of documents, and a level-k prime for another. As a level-$(l+1)$ prime, a cache is expected to receive $(d-1)m_{i,l}$ requests in addition to the requests it receives as a level-l prime.

Assuming our hash function maps an object to each cache with the same probability, each cache is expected to be a level-l prime for $1/d^l$ of all the objects. Since we assume that the update and access pattern of the objects mapped to each cache is the same, the total number of request a cache receives is given by

$$\lambda = \sum_i \lambda_i + \sum_{l<L} \frac{1}{d^{l+1}} \sum_i (d-1) m_{i,l} \tag{1}$$

Let $\gamma_i = \lambda_i/\mu_i$, be the ratio of access frequency to modification frequency at the leaf level (see Section 3). A request from a client that results in a miss at the leaf level will be forwarded up the hierarchy, and will generate other requests at higher levels in the hierarchy. Let $\omega^{\uparrow}_{i,l}$ be the number of such forwarded requests for object O at level l, averaged over all the proxies.

$$\omega^{\uparrow}_{i,l} = \begin{cases} \frac{d-1}{d}\frac{1}{1+d^l\gamma_i} & \text{if } l < L \\ \frac{1}{1+d^l\gamma_i} & \text{if } l = L \end{cases}$$

Equation 1 can be simplified to

$$\lambda = \sum_i \lambda_i (1 + \sum_{l<L} \omega^{\uparrow}_{i,l})$$

In pure push mode, all updates are sent to all proxy caches, so the aggregate update rate μ at any proxy cache is

$$\mu = \sum_i \mu_i \tag{2}$$

In the ideal case, there are no cache misses in the push mode since all the objects are cached and fresh. In our subscription-based hybrid scheme, a miss occurs for O_i at a cache only if the cache has not received the updates for O_i, which can happen only if it has not subscribed to updates for O_i. In this case, γ_i, the ratio of the aggregated request rate this cache sees from all its descendents to the modification rate, is below some threshold, say, h. Thus, the rate of forwarded requests received due to cache misses is given by

$$M_1 = \sum_i \lambda_i (1 + \sum_{l: l<L,\, d^l\gamma_i<h} \omega^{\uparrow}_{i,l})$$

On the other hand, a proxy cache receives pushed updates for object O_i only if γ_i exceeds the threshold h. A level l prime only receives updates only if it is not also the $l+1$ prime for the same object. The root primes receive updates from server. The aggregate rate of modification at a cache is therefore

$$M_2 = \sum_i (\frac{1}{d^l} - \frac{1}{d^l+1}) \sum_{l:l<L, d^l\gamma_i>h} \mu_i + \sum_{i:d^L\gamma_i>h} \frac{1}{d^L}\mu_i \quad (3)$$

A single message pushed from the server to the root will generate other push messages within the hierarchy as it is disseminated to proxies. Let $\omega^{\downarrow}_{i,l}$ be the number of pushed message at the level l, averaged over all the proxies, corresponding to each request from the clients.

$$\omega^{\downarrow}_{i,l} = \begin{cases} \frac{d-1}{d}\frac{1}{d^l\gamma_i} & \text{if } l < L \\ \frac{1}{d^l\gamma_i} & \text{if } l = L \end{cases}$$

Equation 3 can be simplified to

$$M_2 = \sum_i \lambda_i \sum_{l:d^l\gamma_i>h} \omega^{\downarrow}_{i,l}$$

5.2 Outbound Traffic

In a pure pull model, a request that results in a miss is forwarded to the next level prime. If the miss occurs at a root prime, then the request is forwarded to the server. The number of outgoing request is therefore

$$\sum_i \lambda_i \sum_l \omega^{\uparrow}_{i,l}$$

In pure push model, the outbound traffic contains only update messages. The number of update messages a cache sends is

$$\sum_i \lambda_i \sum_{l:l<L} \omega^{\downarrow}_{i,l} = \sum_i (1 - \frac{1}{d^L})\mu_i$$

In hybrid model, the number of outgoing request is

$$M_3 = \sum_i \lambda_i \sum_{i:d^L\gamma_i<h} \omega^{\uparrow}_{i,l}$$

and the number of updates sent by a cache is

$$M_4 = \sum_i \lambda_i \sum_{l:l<L, d^l\gamma_i>h} \omega^{\downarrow}_{i,l}$$

These results are summarized in Table 5.2. The inbound traffic and outbound traffic at each cache are about equal, except when there is traffic inbound from clients or outbound to the server.

5.3 Modeling Workload

Let us define s_1 to be the service time to receive a request and return a response when there is no forwarding to the next level, s_2 to be the service time to send a request and receive a response when there may be forwarding, s_3 to be the service time to receive an update, and s_4 to be the service time to send an update. The overall workload for the hybrid scheme is now

	pull	push	hybrid
Inbound Request (M_1)	$\sum_i \lambda_i(1+\sum_{l<L}\omega^{\uparrow}_{i,l})$	$\sum_i \lambda_i$	$\sum_i \lambda_i(1+\sum_{l:l<L,d^l\gamma_i<h}\omega^{\uparrow}_{i,l})$
Inbound Update (M_2)	0	$\sum_i \lambda_i \sum_l \omega^{\downarrow}_{i,l}$	$\sum_i \lambda_i \sum_{l:d^l\gamma_i>h}\omega^{\downarrow}_{i,l}$
Outbound Request (M_3)	$\sum_i \lambda_i \sum_l \omega^{\uparrow}_{i,l}$	0	$\sum_i \lambda_i \sum_{i:d^l\gamma_i<h}\omega^{\uparrow}_{i,l}$
Outbound Update (M_4)	0	$\sum_i \lambda_i \sum_{l<L}\omega^{\downarrow}_{i,l}$	$\sum_i \lambda_i \sum_{l:l<L,d^l\gamma_i>h}\omega^{\downarrow}_{i,l}$

Table 1: Traffic rates in the push, pull, and hybrid models

$$\rho = \sum_{i=1}^{4} M_i s_i$$

ρ is also the utilization of the proxy, i.e., the fraction of the time that the proxy is busy. If ρ is greater than 1, the system is overloaded.

To study the impact of γ on overall workload, let us assume that for all objects O_i, O_j, $\gamma_i = \gamma_j = \gamma$. Our standard for comparison will be the ideal situation where there is a cache hit at the leaf level for all objects. In this case, there are no forwarding requests, and no updates are sent, the workload is then $\sum_i \lambda_i s_1$. We measure the performance of the system by a index P, which is defined as

$$P = \frac{\rho}{\sum_i \lambda_i s_1}$$

Unlike ρ, P is independent of the request arrival rate. In optimal situation, $P = 1$. Larger P values suggest greater amounts of overhead in the system, forwarded requests, and updates.

We assume that all requests to a proxy await service in a single queue. The service time includes transmission time, i.e., the time to send and receive packets from network interface, and the CPU and disk I/O overhead to look up and retrieve objects. If delta encoding is used, the compression and decompression time also needs to be taken into account.

In Figure 5, we show the impact of γ on amount of workload in hybrid scheme using different threshold values. We observe that pure push and pull are in fact, two limiting cases of our hybrid scheme, corresponding to the threshold values 0 and $+\infty$ respectively. We vary the value of γ within the range of 10^{-2} to 10^2, and within this range, pure push and pull correspond to threshold value 0.1 and 100, respectively. Recall that we push objects only to their primes within a cluster, and all requests for an object within a cluster are also forwarded to the corresponding prime. Therefore, the value of γ_i at a local prime is the product of the number of proxies within the cluster (10 in our case) and γ_i at the leaf proxy. The values of parameters (service time and propagation delay) we use in these figures are the same as those used later in our simulations (see Section 6.1). Our tests indicate that the metrics under study are stable across a wide range of parameter values.

The workload P (Figure 5(c) consist of pull (Figure 5(a)) and push components. For pure pull, the push overhead is 0. Since the maximum number of hops to access a requested object is determined by the height of the hierarchy, the pull overhead changes moderately, and is bounded by the height of hierarchy as γ decreases. For pure push, the pull overhead is is the raw workload from the clients, and is minimal.

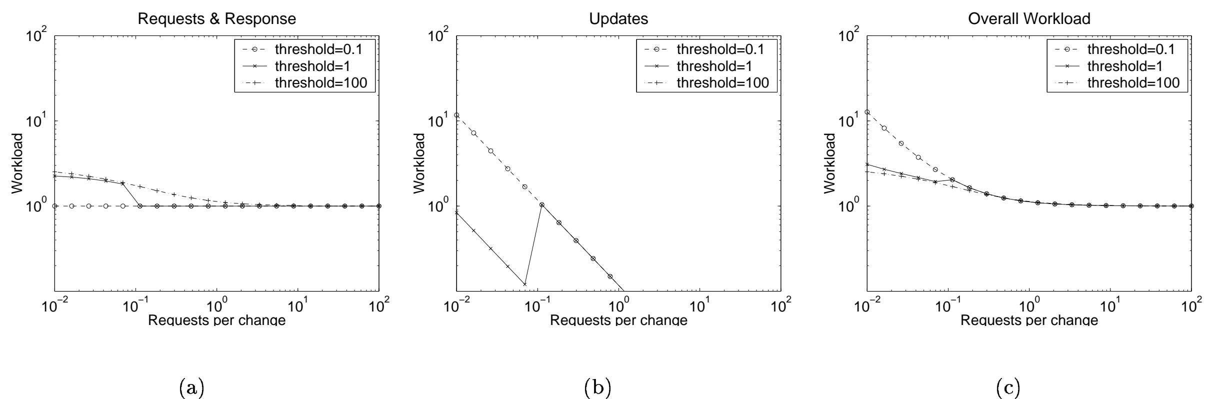


Figure 5: Workload in hybrid model

The push overhead is inversely proportional to γ. As γ increases, the overall workload for both push and pull converges to the raw workload. General speaking, workload for push is heavier than pull, because our analysis shows that update messages in pure push are more numerous than the forwarded messages due to cache misses in pure pull. However, if updates can be processed more efficiently the requests, it is possible for push to produce lower workloads than pull. For the hybrid scheme, the value of threshold can effectively control the pushing overhead. For example, using a threshold of 1 forces the number of update messages to be smaller than the number of requests from clients. As we can observe from figure 5(c), the curve for threshold 1 is very close to that of pure pull.

5.4 Modeling Response Time

Next, we turn to the average latency in our hybrid caching system. We ignore propagation delay, and assume that the response time for a request is the sum of the processing and queueing times at each machine on the path from the client to the object's location.

There are four types of messages processed by a proxy, each with different arrival and service rates. Since we assume that all operations are sequential, we can model the scheme as an M/G/1 system, in which arrivals are Poisson process with rate equal to the sum of the arrival rates of each messages type, and the service time follows a hyper-exponential distribution with density

$$b(s) = \sum_{i=1}^{4} \alpha_i * \frac{1}{s_i} e^{-s/s_i}$$

where $\alpha_i = M_i / \sum_{i=1}^{4} M_i$

We have

$$\overline{s} = \sum_{i=1}^{4} \alpha_i s_i$$

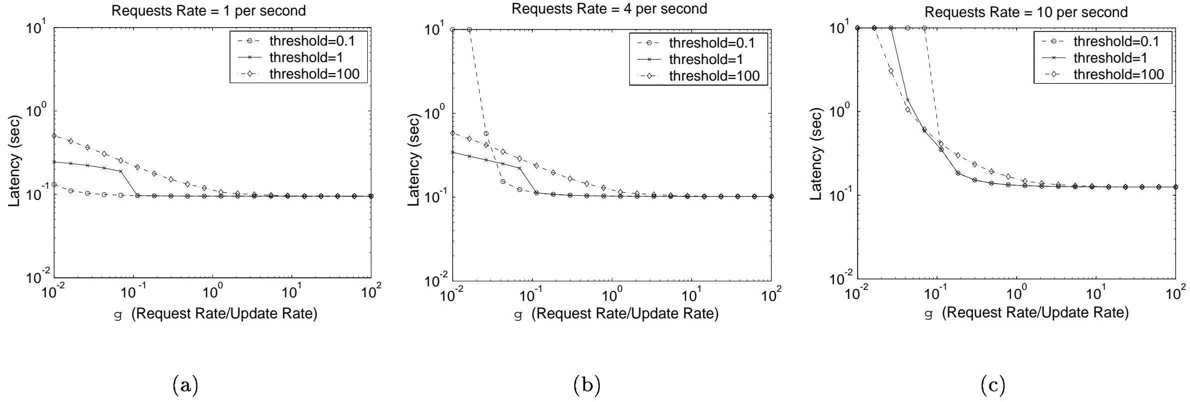


Figure 6: Response Time in hybrid model

$$\overline{s^2} = 2\sum_{i=1}^{4} \alpha_i s_i^2$$

The average waiting time for each message is given by Pollaczek-Khinchin mean value formula

$$W = \frac{M\overline{s^2}/2}{1-\rho} = \frac{\sum_{i=1}^{4} M_i s_i^2}{1-\rho}$$

The average response time for each request is determined by the number of hops to access the data, the waiting time and service time at each hop, and the propagation delay. Since there is queuing and processing involved at both sending and receiving ends, the average aggregated waiting time and service time for each request is given by

$$lat_1 = \frac{M_1(W+s_1) + M_3(W+s_2)}{\sum_i \lambda_i} \tag{4}$$

The average propagation delay is given by

$$lat_2 = \frac{\sum_i \lambda_i \sum_{l:d^l\gamma_i<h} \omega_{i,l}^{\uparrow} dist_{l,l+1}}{\sum_i \lambda_i} \tag{5}$$

where $dist_{l,l+1}$ defines the propagation delay between a layer l prime to its layer $l+1$ prime. When $l = L$, this is the time to retrieve an object at a root prime from the server.

The total response time is thus the sum of the two. $lat = lat_1 + lat_2$

In Figure 6, we show how response time is influenced by γ under various workload using different threshold. The plots show that the pure push scheme is likely to get congested when γ is very small. Proxies are swamped with enormous amounts of useless update messages. As γ increases, the performance of push scheme improves dramatically, and catches up with the pull scheme. The overhead of pushing updates begins to approximates the overhead due to cache miss, without any of the actual cache miss penalties. As γ continues to grow, both push and pull converge to optimum performance, because both updates and cache misses are rare.

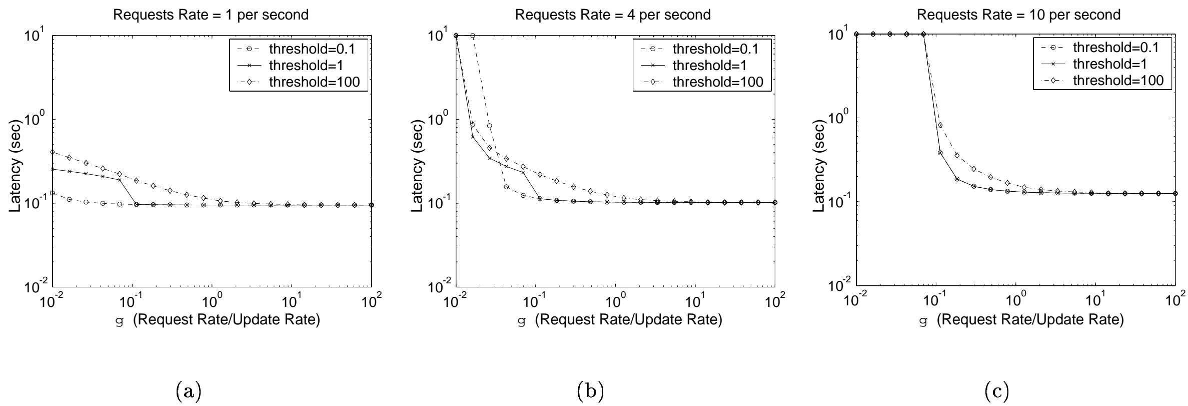


Figure 7: Response Time in hybrid model with Invalidation

Under different workloads however, the range of parameters γ for which push outperforms pull is different. We observe in Figure 5 that when $\gamma > 0.1$, the overhead of pushing updates is approximately the same as the cache miss penalty. In this case, push is always better than pull. For $\gamma < 0.1$, push is likely to be more efficient than pull for light workloads, because the extra overhead barely increases the queueing delay, and has minor negative effects. As workload increases, queueing delay increases, and pull outperforms push. Under heavy workload, however, the system becomes overloaded for both push and pull schemes. Under such circumstances, proxies, not the server, become the bottlenecks for data delivery.

The graphs also demonstrate the advantages of our hybrid scheme. In hybrid mode, we can set the threshold so that for small γ, the performance is approximately that of pure pull, while for large γ, it is the same as that of pure push. The curve for threshold=1 is close to the pure pull curve when $\gamma < 0.1$, and close to the pure push curve when $\gamma > 0.1$. In some cases, the performance is better than either pure push or pure pull. When the workload is light, and the overhead of pushing is comparatively low, we can use smaller thresholds. To achieve the best performance, we must adapt the system behavior to the dynamics of the environment.

We observe knees in all curves showing the effect of the threshold parameter. Consider the curves for threshold=0.1. When $\gamma < 0.1$, objects are pushed to the root level, and when $\gamma = 0.1$, objects are further pushed to local clusters. The increases in hit ratio when pushing further outweigh the push overhead, and the net effect is a steep decrease in response time.

5.5 Modeling Invalidation

Figure 7 shows the effects of using invalidation for consistency control. When an object O_i is modified, updates are sent to proxies subscribing to O_i, and invalidation messages are sent to others. This model more accurately describes the architecture we propose in Section 3.

We assume that an invalidation message is sent to a proxy upon modification O_i if and only if it has accessed O_i since its last modification. At a level-l prime for object O_i, the ratio of the number of

invalidation messages over all modification events of o_i is $\frac{d^l\lambda_i}{d^l\lambda_i+\mu_i}$. The rate of receiving an invalidation message of O_i at a level-l prime is

$$\mu_i \frac{d^l\lambda_i}{d^l\lambda_i+\mu_i} = m_{i,l}$$

The rate of invalidation turns out to be the same as the cache miss rate for a given object at the same level is because all cache misses in our model are due to object modifications. Figure 7 suggests that a system that includes invalidation behaves very similarly to systems that perform no invalidation. (We assume that the overhead for processing invalidations is half the overhead of pushing an update). The average latency increase moderately for pull and hybrid schemes, with small γ value. Above analysis show that invalidation does not cause system behavior to change very significantly.

In summary, the hybrid scheme gives us more flexibility to choose what objects worth to be pushed, and to which level. With threshold=1, we get decent performance in most situations. Under light workloads, we can use even smaller thresholds to improve hit ratios.

6 Experiments and Numerical Results

To study the model in greater detail, we conducted an extensive series of simulation experiments. Such experiments are needed to address the inherent limitations in analytical models. For example, cache misses are, in practice, not only due to object modifications, but to cache size limitations as well. More importantly, as pointed out in [29], as much as 40% of all access are first-time misses, which our model can not capture. The average change rates for objects are in reality weakly correlated with their access rates [17]. In our analytical model, we assume a strong correlation between the two.

6.1 Design of Simulation

We implemented a trace-driven simulator that models both our virtual hierarchy architecture as well as the CARP architecture [15], which is based on static hierarchies. We tested the performance of hybrid push/pull schemes on both architectures. We can measure the relative efficiency of push and pull using different threshold values.

6.1.1 Generation of synthetic data

For several reasons, we have used synthetic data based on an extensive study of NLANR traces instead of real trace data. First, modification information for web pages is incomplete in the available trace data. For proxies, modifications are typically detected through responses to **if-modified-since** requests for a cached object. It is likely that between two successive requests from the proxies, multiple updates occur at the server (see Section 6.1.1). It is also possible that some modification go undetected due to an inappropriate TTL value that mistakenly marks a stale object as fresh. Since modification behavior is an essential element in determining the effeteness of any strong consistency mechanism, we generate such information manually. Second, the characteristics of the trace data are strongly dependent on the implementation and

deployment of the caching scheme, which may be quite different from that used in our architecture. To avoid being influenced by any single set of parameters, we test the performance of our system over a wide range of parameters, and discuss the efficiency of our system in different environments. Next, we discuss several factors that are key concerns for performance.

Access Distribution

Prior studies have concluded [17] that Web requests follows a Zipf-like distribution, C/i^{α}, with a α close to or smaller than 1. For proxy caches, α is typically in the range $0.6 - 0.9$, but it is shown in [1] that the range of α values in server traces is consistently and significantly higher than that reported in previous proxy-based studies. Specifically, the value of α is typically in the range $1.4 - 1.6$.

It is also pointed out in [17] that α at the proxy decreases as the total number of popular servers increases. This phenomenon may have significant consequences for simulations of our architecture, since the proxy hierarchy in our design is deployed between clients and server. Since ours is a hybrid model, it is conceivable that some proxies in the hierarchy will simulate server behavior, compounding the effects upon α.

In our experiments, α is in the range $0.7 - 1.4$. Also, the popularity of the most frequently accessed documents deviates from the Zipf curve. To accommodate these objects, assume that the request pattern at the proxies is aggregation of the pattern of requests to a set of web servers, and that for each web server, requests follow the Zipf distribution. Therefore, the access distribution is a mixture of Zipfs. For simplicity, we assume that α is the same for each distribution.

We assume that the arrivals of requests is a Poisson process, and inter-arrival times are exponentially distributed with parameter λ as the index of workload.

Modification Distribution

Some previous work has also studied the modification process of web contents [1, 19, 17]. Unlike web access patterns, which can be modeled as Zipf distributions, modification patterns are more irregular, diverse, and hard to summarize in succinct rules. However, there are still some distinctive and consistent features about the modification process. For example, the distribution of average times between two successive modification of an object is known to be bimodal. That is, an object is either modified frequently, or rarely. It is also observed in [19] that as the number of references increases, an object is more likely to be updated at fixed, regular intervals.

It is shown in [1] that in 95% of instances, the time duration between two successive modification of an object is under one day.

In our experiments, the modification distribution is defined by a coarse-grained density function. We assume that hot objects are likely to be modified more frequently, and categorize objects into categories based on access frequencies. Thus, 0.05% may be extremely popular index pages, 0.8% may be hot pages, 20% may be warm pages, and 80% cold pages. Each category is now assigned a portion of modification events, for example, warm pages may account for 80% of updates. All objects within a category may

be modified with=th equal probability. These numbers are selected so that objects in each category are several times more likely to be changed than those in the next category, and cold objects are hardly changed at all. Again, we assume that modifications follow an exponential distribution with parameter μ, which measures the frequency of updates of all the objects at the server.

We assume in our simulation that all objects are of equal size. We distinguish three types of messages: control messages, object responses, and modification messages. Control messages include **http-request**, **subscribe** and **unsubscribe**, and **invalidate**. Modification messages contain the latest version of the object, which may be compressed using delta encoding. Transmission times for messages of each type follows an exponential distribution.

6.2 Simulation of the Hierarchical Cache system

In our experiments, we simulate a 3-level (leaf level, upper level, and root level) skeleton. Each cluster has ten lower sub-clusters or proxies. Thus, there $10^2 = 100$ proxies in all. Each proxy has the same workload, that is, a request may originate from any cache with equal probability. Each cache is assigned a certain capacity, which determines the average processing time of each request. We set the time for an object lookup to be 10ms and the time for retrieving an object from disk to be 30ms. The expected time to read and write a message from lower layer network stack is several milliseconds, depending on the type of the message. Specifically, we take the transmission time be 3, 6 and 10 milliseconds, respectively for control messages, modification messages and full responses.

We assume that operations at each proxy are sequential. Queueing delays occur when there are concurrent requests in a proxy. We assume queueing requests are scheduled by a First-In-First-Out (FIFO) policy. The total response time for a request is defined to be the sum of processing time, transmission time, the queueing delay and propagation delay at each proxy or server that it reaches.

We specify that as requests are forwarded up the skeleton, the propagation delay increases linearly, based on the assumption that the diameter of the cluster increases linearly as its size grows exponentially. For example, we take the propagation delay of response message between proxies in the same the lowest level cluster to be 50ms, between proxies in the next level cluster as 100ms, and so on. In physical hierarchy, we assume that the average propagation delay is the same between proxies at any two adjacent layer. For simplicity, we assume that queueing delay at the intermediate nodes is not counted, and that propagation delay is independent of the traffic on the network.

The number of documents served in our simulation is $200,000$. The NLANR proxy mesh trace on 01/15/2001 shows that $341,712$ html pages were referenced by 9 NLANR root proxies during the day. However, new objects are created from time to time, and a significant number of accesses are to the objects that have not been accessed before by clients from the same domain. Therefore, the active sets of objects at each proxy change over this time. To simulate such object dynamics with a fixed size population, we associate each cached object with an expiration period. That is, if an object has not been accessed before expiring, it will be replaced. The next time this expired object is accessed, it can be regarded as a new object. We use LRU as our replacement algorithm. However, if a proxy has lower level primes

subscribing to an object, then the object remains in the cache even after it expires. When such an object is evicted from a proxy, the proxy is required to send an explicit **unsubscribe** message. We refer to the expiration period as TTL in the rest of our discussion. A larger TTL value corresponds to higher stability of objects population referenced by a proxy.

To eliminate of effect of the cold start transient phase due to initially empty caches, we begin measurement after the performance has reached stability. That is, when the average response time varies little, say within 20 milliseconds, over a considerable period of time. The simulation lasted $20,000$ seconds after reaching stability.

6.3 Experimental results

Our simulations are designed to answer the following questions: (1) How are response time, hit ratio and traffic intensity affected by modification and access patterns, and (2) what are the relative efficiencies of the push, pull and hybrid schemes under each configuration? The intent of our simulation is not to obtain accurate predictions of performance under specific system configurations and workloads, but rather to elicit information on performance trends under a wide range of parameters, so that our conclusions are not restricted to particular environments.

The metrics of interest in our simulation include the average response time, network bandwidth consumption, and the hit ratio at each level of skeleton. In the following discussion, we refer hash-based virtual hierarchy as HVH, and cache array routing protocol as CARP. Unless otherwise specified, we are talking about the HVH scheme. We refer to the hybrid schemes with threshold 0.1 and 100 as the pure-push, pure-pull schemes, respectively.

The results of the simulation are as follows.

6.4 Update Rate

The range of update rates under test was from one update every 8 milliseconds to one update per second. The workload is one request every 125 milliseconds. We select 8 milliseconds to be the lower bound for the update interval since the system is congested below that level.

Comparing the simulation results (Figure 8(a)) with the analytical results (Figure 8(b)), leads to the following observations. First, both figures show the same trend. Pure push performs pretty badly when updates are very frequent, outperforms pure pull under moderate update rates, and performs roughly as well as pure pull when updates are infrequent. The hybrid scheme with threshold 1 gives very good performance over various update rates.

Our analytical results are more pessimistic than our simulation results when update rates are high. The reason is that many updates are not noticed by clients in practice, since only modifications during an object's cache lifetime trigger invalidation or update message. On the other hand, the analytical results are too optimistic when update rates are low, because first misses account for a significant number of cache misses. Such misses are not update related.

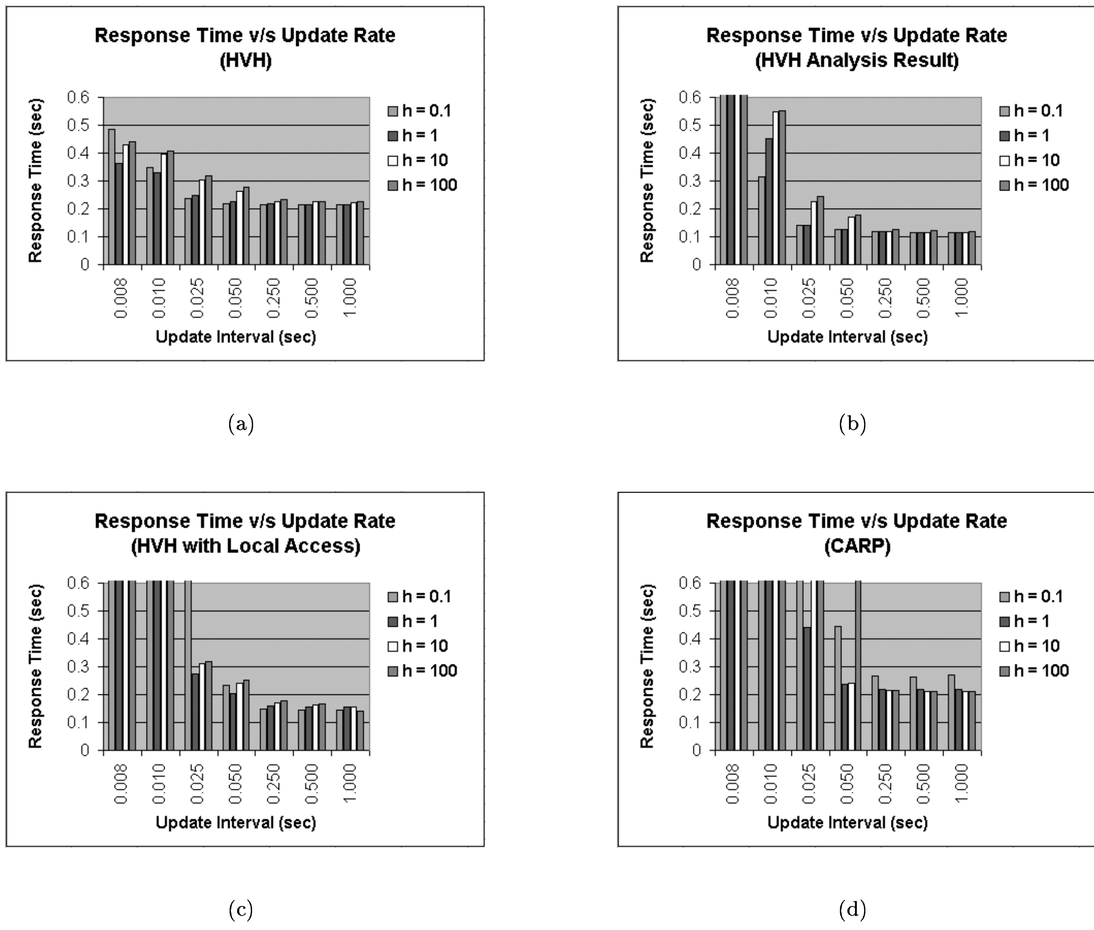
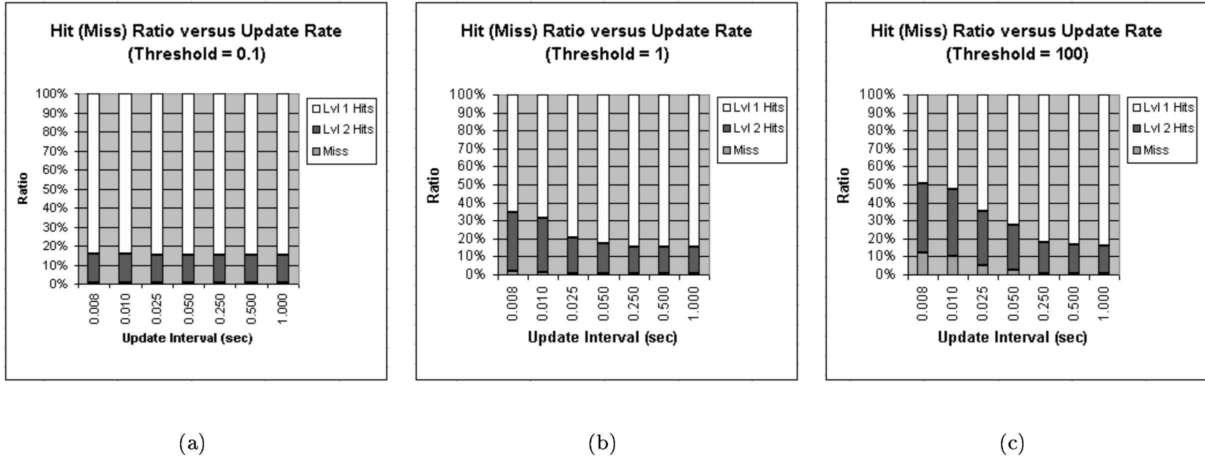


Figure 8: Response Time vs. Update Rate

As Figure 8(d) shows, CARP's performance saturates at far lower update rates under the same set of parameters. Static hierarchies are far more likely to be congested at higher updates rate due to aggregation of the workload at the root-level proxies. This is consistent with our earlier discussion. In CARP, the superiority of the hybrid scheme over the pure push and pure pull models is even more noticeable. In addition to reducing root proxy congestion, the hybrid scheme also requires fewer update messages than pure push, and results in fewer cache miss requests than pure pull.

6.4.1 Local Access Lookups

In a hierarchical organization, we may choose to attach each leaf-level proxy c to a group of clients. When a request for object O_i arrives at c from one of these clients, an attempt can be made to first lookup O_i in c instead of forwarding the request directly to the appropriate prime. We term this strategy as "local access". Responses are cached in the local proxies, and updates/invalidations are also delivered to them



(a) (b) (c)

Figure 9: Hit ratio vs. Update Rate

correspondingly. While the local access scheme permits replication within the cluster and increases local hit ratios, it can also increase cache miss penalty. Figure 8(c) shows that with local access, proxies are more likely to be congested when update rates are high, but local hits helps to decrease response time when updates are rare.

We now turn to two major factors influencing response time: hit ratio and network traffic.

6.4.2 Hit ratio

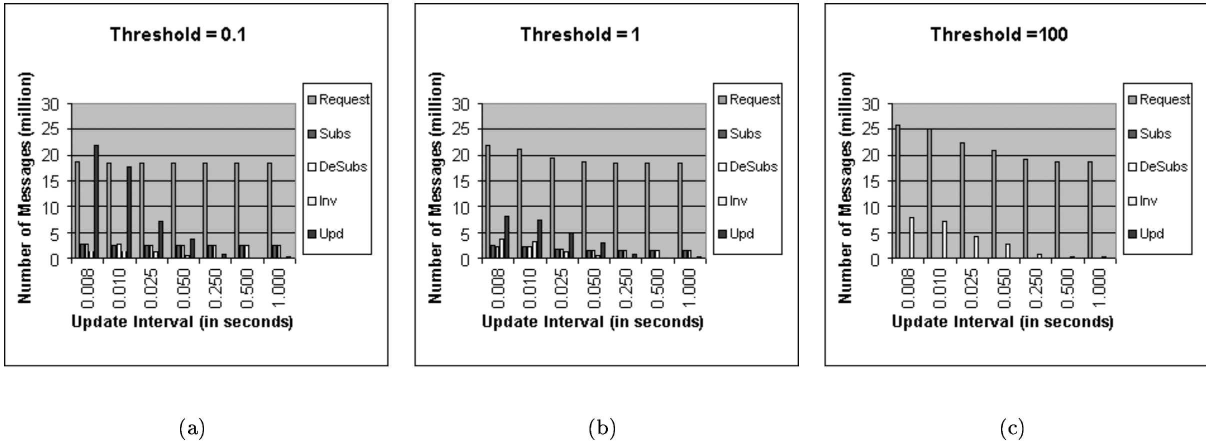
Under high update rate, the first-level hit ratio for pure push is around 18% higher than that of the hybrid scheme, which is in turn around 16% higher than that of pure pull. Root-level cache misses rates are negligible for the pure push and hybrid schemes, but can be over 10% for pure pull. As update rates decrease, the hit ratio at each level becomes approximately the same no matter what threshold is in use, since first-time misses dominate. First-time misses can not be eliminated by pushing updates.

We observe that for the pure-push scheme the hit ratio at each level remains quite stable as the update rate changes (see Figure 9). In the pure push scheme, all updates are pushed to leaf level, and do not cause any cache misses.

6.4.3 Network Traffic Components

In the pure-push scheme (see Figure 10(a)), the number of requests remains constant as the update rate changes, since this number is determined by hit ratio at each level, which remains constant according to our previous discussion. The number of updates, on the other hand, increases in proportion to update rate. There are no invalidation messages.

In the pure-pull scheme (see Figure 10(c)), the number of requests increase moderately with the update rate. There are no update messages. Because an object need not be invalidated unless it has been referenced



(a) (b) (c)

Figure 10: Number of Messages vs. Update Rate

after its last invalidation, the number of invalidation messages in pure pull is much smaller than the number of update messages in pure push.

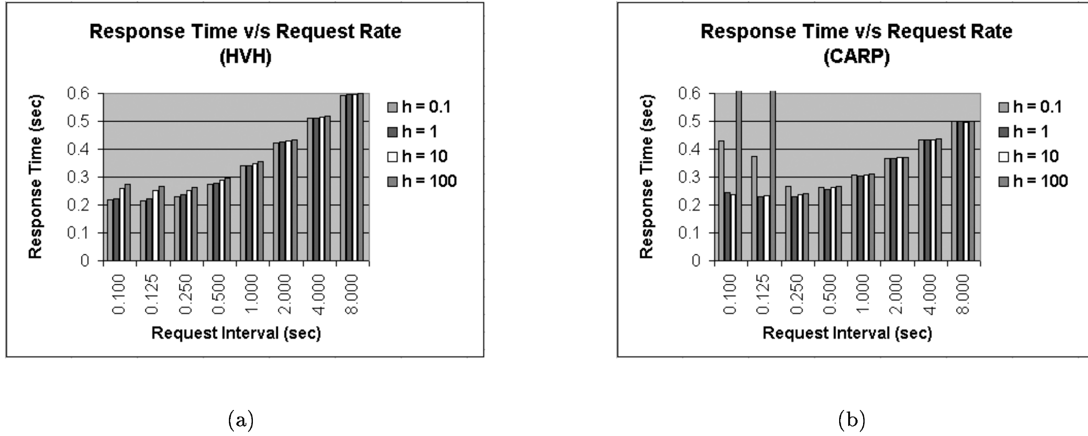
In the hybrid scheme with threshold value 1 (see Figure 10(b)), the number of update messages is comparable to the number of invalidation messages in pure pull, while a small number of invalidation messages coexist. The number of request messages is smaller than in pure pull.

Now consider the net effect of the above two factors in determining the response time. When updates are frequent, with lots of consistency overhead or cache miss penalty involved, the system approaches its full utilization, and queueing delay dominates the total response time. In this case, reducing network traffic becomes the first priority. As update rate decreases, the store-and-forward and propagation delays on the data access path dominate the response time. Therefore, increasing the hit ratio to serve requests locally becomes more important.

6.4.4 Workload

We varied the workload from one requests every 0.1 second to about one request every 8 seconds from clients at each proxy. A study by Rousskov and Soloviev [3] has shown that the traffic intensity at top-level proxies in NLANR international hierarchy is about 10 requests at peak hours, and about $12-14$ at root proxies. As per the discussion in 6.1, we assume each proxy can handle a maximum of 20 http requests per second in our simulation. We note that our concern is not with the actual request rate, but with the system utilization. We are aware that [4] reports significantly higher request rate (500 requests per second for a single cache), and various single host proxies from the Web Caching Bake-off report throughputs ranging from 96-690 requests per second [5]. Our conclusions will remain valid under higher workload, given a higher-capacity system.

We conducted out two sets of simulations for each workload, one under low update rate (Figure 11) and one under high update rate (Figure 12). The update rate aggregated over all objects is twice and ten



(a) (b)

Figure 11: Response Time vs. workload (aggregated update rate= 2*arrival rate)

times the requests rate from clients per proxy, respectively.

Under low update rate (Figure 11(a)), push, pull or hybrid scheme all show similar performance, with push performing slightly better. Because the workload is well-balanced in HVH, no congestion occurs under the simulated requests rates. Paradoxically, however, the response time actually increases as workload decreases. The reason is that there is less overlap among requests as proxies aggregate fewer requests from clients, and the hit ratio will be compromised. Since the queueing delays are low under such workloads, any decrease in hit ratio will increase response time.

In CARP, an increase of workload has manifest effect on response time. Upstream caches are likely to be swamped by forwarded requests from downstream proxies, or because they must forward updates to downstream proxies. Since the hybrid scheme generates less congestion, we see in Figure 11(b) that pure-pull and pure-push schemes perform worse than the hybrid scheme with threshold 1, and 10. As workload decreases, upstream proxies are no longer congested, and CARP performance approximates that of HVH. When workload drops below one request every 0.25 seconds, CARP actually performs better than HVH. Under these conditions, local hit ratios are lowered, and more requests are forwarded to upstream CARP proxies. Our simulation model assumes that the propagation delay between downstream and upstream proxies in CARP is less than the delay between lower-level primes and higher-level primes.

Under high update rates (Figure 12), we can recognize two phases. As the request rate decreases, the response time first decreases, then increases. The queueing delay in HVH becomes a factor under high requests rate, because more updates can cause more cache misses and greater consistency overhead. In such cases, CARP is nonetheless more likely to get congested.

6.4.5 The Zipf parameter α

To study the effects of α, we vary its value from 0.7 to 1.4. A higher α value suggests greater spatial locality. The request rate simulation was one request every 0.125 second, and the aggregated update rate

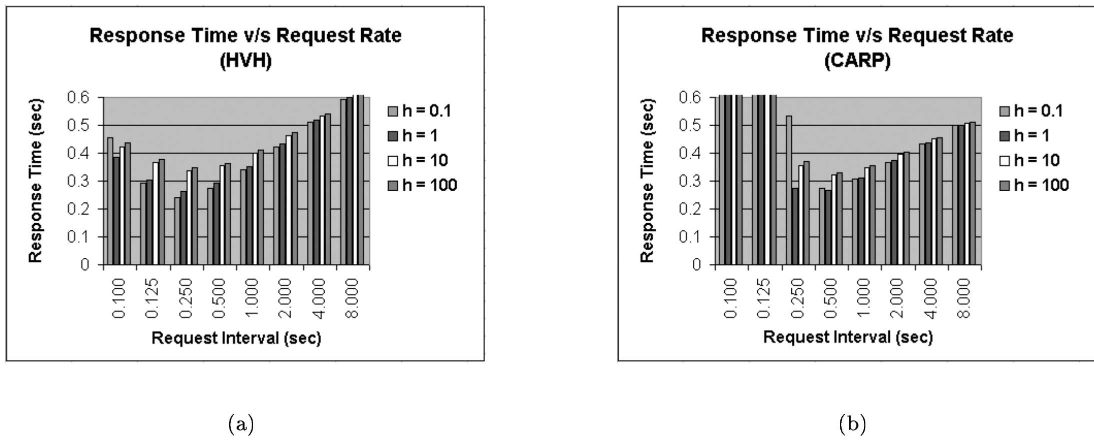


Figure 12: Response Time vs. workload (aggregated update rate= 10*arrival rate)

twice as much. According to Figure 13, caching becomes increasingly efficient as α increases. Varying α does not change the relative efficiency of hybrid schemes corresponding to different thresholds, though the difference becomes smaller with larger α values.

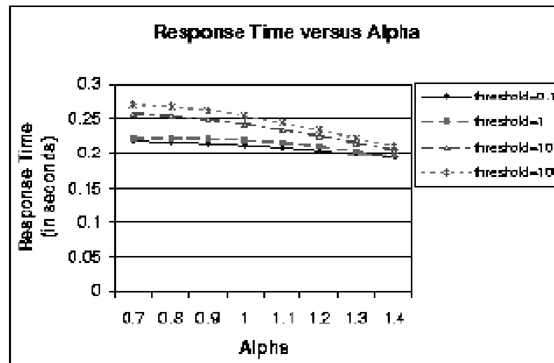


Figure 13: Response time versus α

The results of simulation can be summarized as follows: Push of updates can be very efficient as long as it does not cause over-utilization of the system. In the hybrid scheme, we can use the threshold as a tuning parameter to effectively control the overhead while improving hit ratio. Hash-based Virtual Hierarchy effectively balanced the workload, and relieve the congestion that is prevalent at upstream proxies in physical hierarchical system.

7 Conclusion and Future Work

We have presented a hash-based hybrid push-pull architecture, and analyzed and compared the performance of three web content delivery approaches: the push, pull and hybrid schemes. Our results show that our hash-based virtual hierarchy scheme can greatly relieve the congestion that is common at upstream caches in physical hierarchies, and that hybrid scheme can dynamically choose push or pull on a per-object basis to combine the benefits of both approaches, while keeping the overhead low. Our simulations show that hash-based virtual hierarchies for hybrid push/pull work efficiently and stably over a wide range of parameters and environment.

In our simulation, we assume requests and update arrivals are Poisson. Real network traffic tends to be more bursty. In addition, the change in popularity of documents is hard to model, so we need to test the system under real http traces.

We also know from the simulation and analysis results that we can improve performance if push and pull behaviors could adapt to workload dynamics. We need to investigate how proxies should communicate with each other to identify and respond to network conditions, while avoiding oscillations or delayed responses.

We also need more sophisticated algorithm to determine what objects to push. One possibility is to use the web link structure to deduce reference correlations among objects, and group related objects as a single push unit. Hints from server could also determine if a set of objects will be a new hot spot. This is especially helpful for newly-created objects.

In addition, we are interested in studying how to organize skeletons to achieve greater local hit ratios, and provide fast name service for primes lookup.

To reduce cache miss penalty, we need methods to reduce the number of hops to locate and access data. On cache misses, for example, some requests may be forwarded to the server directly if we know that cache hit probability in the hierarchy is very low.

Finally, there are many policy and administrative issues. We expect to build a prototype for our architecture.

8 Acknowledgements

We would like to acknowledge numerous helpful discussions with the members of the Tata Consulting Services Research Laboratories at Riverside, CA. This work was supported in part by grants from Tata Consulting Services, Inc., and the Digital Media Innovations Program of the University of California.

References

[1] Venkata N. Padmanabhan, Lili Qiu The Content and Access Dynamics of a Busy Web Site: Findings and Implications In *SIGCOMM 2000, Stockholm, Sweden, August 2000*

[2] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal CArdwell, Anna Karlin, Henry M.Levy. On the scale and performance of cooperative Web proxy caching. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*

[3] A. Rousskov, V. Soloviev On Performance of Caching Proxies In *Proceedings of ACM SIGMETRICS*, June 1998.

[4] P.Danzig. NetCache architecture and deployment. In *Proceeding of the 3rd Int. WWW Caching Workshop.*

[5] A. Rousskov, D. Wessels, G. Chisholm. The first ircache web cache bake-off. Technical report, National Laboratory for Applied Network Research, April, 1999.

[6] Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, and K.J.Worrel. A hierarchical Internet object cache. In *Proc. of the 1996 USENIX Technical Conf.*,pages 153-163, January 1996

[7] Squid Web Proxy Cache. http://squid.nlanr.net

[8] L.Fan, P.Cao, J.Almeida, and A.Z.Border. Summary Cache: A scalable wide-area wweb cache sharing protocol. In *Proc. of ACM SIGCOMM' '98* August 1998.

[9] J.-M. Menaud, V. Issarny, and M.Banatre. A new protocol for efficient transversal Web caching. In *Proc. of 12th Int. Symp. On Distributed Computing*, pages 288-302, September 1998.

[10] www.microsoft.com/standards/cdf.htm

[11] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching: Towards a New Global Caching Architecture. In *Computer Networks and ISDN Systems*, 3022-23):2169-2177, November 1998.

[12] J. Touch. The LSAM proxy cache-multicast distributed virtual cache. In*Proc. Of the 3rd Int. WWW Caching Workshop*, June 1998.

[13] David G. Thaler, Chinya V. Ravishankar. Using Name-Based Mapping to Increase Hit Rates. In *IEEE/ACM Transactions on networking*, Vol 6. No. 1, pages 1-13, February 1998.

[14] D. Karger, T. Leighton, D. Lewin, and A. Sherman. Web Caching with consistent hashing. In *Proc. of the 8th Int. World Wide Web Conf.* , May 1999.

[15] V. Valloppillil and K.W. Ross. Cache array routing protocol v1.0. ftp://ftp.isi.edu/internet-drafts/draft-vinod-carp-v1-03.txt, Feb 1998.

[16] David G. Thaler, Chinya V. Ravishankar. Using Name-Based Mapping for Rendezvous. Technical Report CSE-TR-316-96, *University of Michigan*, 1996

[17] L.Breslau, P.Cao, L.Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Proceedings of IEEE Infocom 99*, march 1999.

[18] Ross, K. Hash Routing for Collections of Shared Web Caches. In *IEEE Networks*, vol. 11, no. 6, November,December 1997, 37-44.

[19] F.Douglis, A. Feldman, B. Krishnamurthy, and J.c. Mogul Rate of Change and Other Metrics: A Live Study of the World Wide Web In *Proc. USITS '97*, December 1997

[20] J.C.Mogul, F.Douglis, A. Feldman, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proc. SIGCOMM '97*, April 1998

[21] Chengjie Liu and Pei Cao Maintaining Strong Cache Consistency in the World-Wide Web In *International Conference on Distributed Computing Systems*,1997

[22] L.Fan, Q.Jacobson, P.Cao, and W. Lin. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Proc. of SIGMETRICS'99*, May 1999.

[23] V.N.Padmanabhan and J.C.Mogul. Using Predictive Prefetching of Improve World Wide Web Latency. In *ACM SIGCOMM Computer Communication Review*, July 1996.

[24] V.Cate.Alex A global file system. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1-12, May 1992.

[25] J. Gwertzman, M. Seltzer World-wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference, San Diego, CA*, January 1996.

[26] S.Acharya, R.Alonso, M.Franklin, and S.Zonik. Broadcast Disks: Data Management for Asymmetric Communications Environment. In *Proc. ACM SIGMOD Conf*, May 1995.

[27] S.Acharya, M.Franklin, and S.Zonik. Prefetching from Broadcast Disks. In *Proc. of the 12th Intl. Conf. on Data Engineering*, February 1996.

[28] D.Aksoy and M.Franklin. Scheduling for Large-Scale On-Demand Data Broadcasting. In *IEEE INFOCOM '98*, 1998.

[29] A.Vahdat, M. Dahlin, T.Anderson, and A. Aggarwal Active Names: Flexiable Location and Transport of Wide-Area Resources. In *Proc. USITS'99*, October 1999.

[30] David G. Thaler and Chinya V. Ravishankar Distributed Top-Down Hierarchy Construction In *Proc. of the IEEE INFOCOM*, 1998