Performance Evaluation of a Demand-Driven Data Diffusion Algorithm for Hierarchical Caching

Raja Mukhopadhyay, Feng Wang, Keith Kong,
Dipak Ghosal, and S. Louis Hakimi

UC Davis Computer Science Technical Report CSE-2000-13

November 2000

Performance Evaluation of a Demand-Driven Data Diffusion Algorithm for Hierarchical Caching

Raja Mukhopadhayay†, Feng Wang‡, Keith Kong‡, Dipak Ghosal† and S. Louis Hakimi‡
†Department of Computer Science
‡Department of Electrical and Computer Engineering
University of California at Davis
Davis, CA 95616
{wangf,kkong,hakimi}@ece.ucdavis.edu, {raja,ghosal}@cs.ucdavis.edu

Abstract—
This paper proposes a new scheme for data diffusion in hierarchical caches. In hierarchical caches derived from the Harvest project, the data diffusion is achieved by the following algorithm: whenever an object comes down the hierarchy, it is cached at each caching node along the path. Moreover, when an object is dropped from the cache, neither the object itself nor any meta-information associated with it is passed up in the hierarchy. These policies result in inefficient placement and replication of objects in the caches. We propose a new algorithm in which each caching node maintains a reference count of the number of requests that arrive for each object over a certain window of time. This object meta-information is passed up the hierarchy when the object is dropped from the cache. Furthermore, when the object comes down the hierarchy, at each caching node the reference count of all the objects in the cache including that of the incoming object is used to make a caching decision. The proposed algorithm places and replicates objects in the cache based on their relative demands - popular objects are cached at lower levels while unpopular objects are placed at higher levels. This significantly reduced bandwidth usage and object download times particularly when cache space is limited. Using a simulation model, the above result is shown through a detailed sensitivity analysis with respect to cache size, number of objects, request rate, and network topology.

Keywords—**Hierarchical caching, Data diffusion, Coordinated cache replacement, Reference counting, Performance comparison.**

I. Introduction

WEB caching takes advantage of the high volume of redundant traffic in the Internet by storing frequently-accessed data close to the request sources [8] [11]. User requests are fulfilled quickly and locally, bypassing the possibly far away and heavily loaded server and congested intermediate links. The simplest level of caching takes place at the Web browser (WWW client) where copies of frequently requested objects are maintained on disk and memory so that subsequent requests by the user for those objects can be satisfied quickly. Another level of caching takes place at the proxy server, which typically satisfies requests from a pool of clients [12] [13] [1]. A file requested by one user and cached at the proxy server can be used to satisfy subsequent requests for that file by other users.

A higher degree of sharing of client requests is achieved by creating even larger client pools. This is done by creating a mesh of cooperative caching servers that behave as if it were a single large cache. In this scheme, a request that comes to a caching server can be served by any caching server in the mesh that has the associated object. Because the number of requests aggregated over all the caches is significantly higher than that of a single cache, the cache hit rates are correspondingly higher. A key design issue in such caching meshes is the logical organization of the participating caches. Two schemes have been suggested in the literature, namely, hierarchical caching and directory-based caching. This paper focuses on hierarchical caching.

In hierarchical caching [7] [2] [6], the servers in the caching mesh are organized in a hierarchy. A typical hierarchy consists of client caches at the fringes of the Internet, institutional caches at the "gateway" to the Internet, regional caches at the interconnects to the backbones, and finally, national caches. The hierarchical organization of caching meshes tends to map well to the hierarchical topology of the Internet, where traffic from multiple sources are aggregated at one level is fed into a higher capacity "concentrator" at the next level.

There are two key objectives in the the design of hierarchical caches. The first objective is to ensure that popular objects are cached at the lower levels of the hierarchy, while the less popular objects are cached at higher levels. The second objective is to ensure that when there is a cache miss at a particular caching server, the request is satisfied by some caching server in the same or higher level, thus precluding the need for the request to be serviced by the origin server[1]. These objectives reduce the average la-

This research was supported through NSF grants NCR-9703275 and ANI-9741668.

[1] In the remainder of this paper, we will use this term to represent the

tency perceived by the user and the overall network traffic. Both objectives, particularly the first one, depend on the data diffusion strategy and the cache replacement strategy. These strategies determine how the objects are placed in the various caching server in the hierarchy.

In this paper, we focus on the data diffusion strategy in hierarchical web caches that originated from the Harvest project [6]. Popular hierarchical caches derived from Harvest include the commercial product *NetCache* and the public-domain *Squid* caches [14]. Harvest's data diffusion strategy tries to replicate objects in the caching mesh using a very simple strategy: when an object is retrieved from the origin server or a caching server, it is cached at every caching server along the path to the client. Moreover, the cache replacement policy is a purely local operation. When an object is dropped from a cache, neither the object itself, nor any meta-information associated with it, is passed to other caching servers in the hierarchy. As we argue later, these strategies have associated penalties; when the cache space is limited, they do not result in efficient placement and replication of objects in the various caches.

This paper proposes a new data diffusion strategy for hierarchical caches. In this scheme each caching server locally maintains a measure of the "popularity" of each object, keeping a reference count of the number of requests for an object that arrives at a node over a certain window of time. This object meta-information is used to carry out a co-ordinated cache replacement policy. In particular, when a object is dropped from a cache, the reference count (meta-information) of the object is propagated up the hierarchy and aggregated with the local reference count at each level. Furthermore, the meta-information is propagated up to the level where the object is (or will be subsequently) cached. Finally, when the object comes down the hierarchy, at each caching node the reference count of all the objects in the cache including that of the incoming object is used to make a caching decision. This co-ordinated policy places and replicates objects in the cache that correspond to their current demands.

Based on detailed simulation analysis, we show that the way objects are distributed across the hierarchy in Harvest-derived caches [7] is inefficient. Our data diffusion scheme performs better than the Harvest scheme in that both the object download time and the bandwidth usage are lower. The benefits of our scheme are greatest when the cache size is small. While the gains are higher in case of topologies that are strictly hierarchical (like trees), the gains are significant even for mesh-like topologies.

The rest of the paper is organized as follows. In Section II we outline the key ideas of hierarchical caching and describe the data diffusion scheme in caches derived from Harvest. In Section III, we describe the new data diffusion scheme. In Section IV, we describe the simulation model and the methods used for generating the client request profile. In Section V, we present the results. In Section VI we discuss our research in the context of other ongoing and related research. Finally, in Section VII, we give a summary of the results and highlight some future directions for research.

node that is the original and permanent repository of the object.

II. Hierarchical Caching

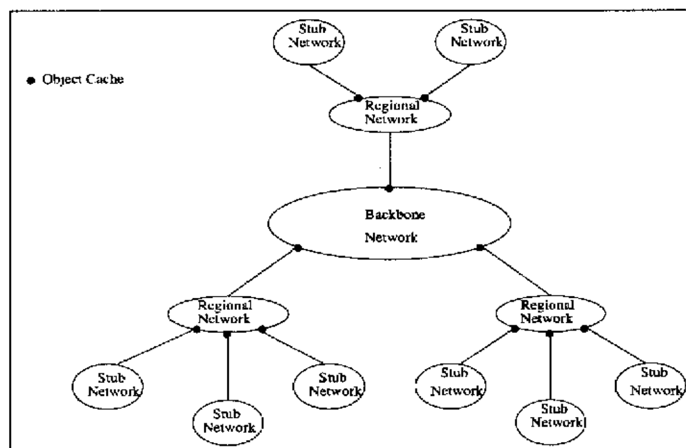

Fig. 1. An example hierarchical cache arrangement (referred later to as Topology I).

A typical organization of a hierarchical cache is illustrated in Figure 1. Each caching nodes has a parent, several siblings, and children. The term "neighbor" is used to refer to either a parent or siblings which are one cache-hop away. The caches in a caching hierarchy use the Internet Cache Protocol (ICP) [16] to communicate with each other.

The request resolution protocol in hierarchical caching is simple. When a cache gets a request, it checks whether it has the requested object. If it has the object, it satisfies the request by serving the file from its cache. Otherwise, it multicasts an ICP request to its neighbors. If one of the neighbors responds positively, a *neighbor hit*, then the object is retrieved from that neighbor. If multiple neighbors respond positively, the object is retrieved from the neighbor with the lowest measured latency. If none of the neighbors has the file, a *neighbor miss*, which is known either through a negative response or a timeout, the request is either forwarded to a selected parent which repeats the process or the request is sent directly to the origin server. If the request misses at each level of the hierarchy, a top-level parent fetches the object from the source, caches it, and passes it down the hierarchy to the leaf node that

originated the request. The following are some potential drawbacks with this approach [14].

- The cache coverage available to any cache is not "global" in the sense that a cache cannot benefit from objects that are cached in its "cousins" and their descendents. This is because requests only go up the hierarchy, never down.
- Fetched objects typically have to pass through several intermediate caching servers before reaching the original requester. Tewari et. al. [15] show that this leads to remarkably high latencies in some configurations, motivating shallower hierarchies. It also increases the load on the higher level servers, which have to field the cache misses from all their descendents.
- Lost ICP messages or busy neighbor caches can increase miss latencies. A caching server has to await for the "negative" responses from all its neighbors or time out before it can forward that query to its parent. The deeper the cache hierarchy, the higher is the miss latencies as the querying process has to through a higher number of levels.

As a result of these potential drawbacks, it is critical from performance point of view to obtain an efficient global distribution of objects across the caching hierarchy.

A. *Data Diffusion in Hierarchical Caches - The Harvest Way*

The data diffusion strategy in the Harvest scheme tries to replicate objects in the caching mesh. This is achieved by caching the object at every node along the path when it comes down the hierarchy. The motivation for caching the object at all nodes in the downward path is that subsequent requests for the object would have multiple points in the caching hierarchy from where it could get satisfied, thus decreasing the likelihood that a subsequent request would need to be satisfied by the origin server. This, however, has associated penalties, in that when the the cache space is full, some existing (potentially more popular) object would have to be dropped to make room for the incoming (potentially less popular) object.

The Harvest scheme also suffers from unnecessary replication of objects resulting in inefficient use of limited caching space. Consider the case in which a single client (a proxy server, for instance) makes a lot of requests for a object, but the request rates for that object from other clients are relatively low. When the object comes down, it would be cached at all nodes along the path, including the client. However, at all nodes apart from the client, since the request rate for the object is small, it is merely uses up cache space, and replacing a more popular file to make room for this object is clearly inefficient.

In Harvest derived caches, when an object is dropped from a cache, neither the object itself, nor any meta-information regarding it, is passed up in the hierarchy. This may be inefficient. For instance, consider an object that is cached at two nodes that have the same parent. Also, suppose that the request rate for the object at each child node is relatively small, but the combined request rate that the parent would see if the object was not cached at both the child nodes is high. Thus, when the object is dropped from the child nodes, the parent node does not get it until after the next request arrives, and even then, since the parent has no history about the file, it is likely that the object will be dropped again until its request profile builds up.

Intuitively, it is better for the objects to be cached as close as possible to the leaves of the hierarchy. This could be done if there were infinite cache space at the leaf nodes of the hierarchy. However, with finite cache space, there must be a mechanism which places more the popular objects at the lower levels and the less popular ones at the higher levels in the hierarchy. Moreover, the scheme must be responsive to the changes in the demand for the object. As the demand increases the object should "percolate down" the hierarchy and when the demand decreases, it should "bubble up" through the hierarchy.

III. The Proposed Demand-Driven Data Diffusion Algorithm

In this section we our proposed data diffusion algorithm. The pseudo-code for the algorithm are listed in three parts (Parts 1 - 3). In the proposed algorithm each caching server maintains reference count for all objects for which it has received a request[2]. If T denote the current instant, the reference count for a given object O at a given node N is the number of requests that have been received at N for O in the time interval $[T - W, T]$, W being the length of the time interval over which we maintain the reference count. When a node receives a request for an object, it increases its reference count for that object. It the object is in the local cache, it serves the request immediately, else it forwards the request to the next node on the path to the origin server, which then handles it recursively. When the object comes down from some caching server, possibly the origin server, each node along the path tries to cache the object. If there is space in the cache, this attempt succeeds. However, if the cache is full, the attempt fails if the reference count for the object is lower than the reference counts for all the objects currently in the cache, otherwise it succeeds. In other words, nodes use a LFU cache replacement strat-

[2]We assume that this meta-information will be maintained even when the object is not in the cache. In practice, meta-information of objects for which there has been no requests over some long interval, will be periodically purged.

egy with the reference count being used as the frequency measure.

When an object is dropped from a node, its reference count as measured at that node is passed up to the next node (which we will refer to as the parent node) along the path to the origin server. The parent node adds the reference count it receives from the child with its own reference count for that object. If the parent node has the object in its cache, it does nothing more. If it does not have the object in its cache, it tries to see what the result would be of trying to cache the object in its cache at that instant (even though it does not have the object), with the merged reference count. If the decision was that the object would be cached, once again it does nothing more. However, if the decision was that the object would be dropped, the parent sends up the reference count it received from the child to its own parent, which then handles it recursively. The intuition behind this is to ensure that the meta-information about the object, which in this case is its reference count is propagated up the hierarchy. However, we want this meta-information to propagate till the right point in the hierarchy, this point being a node where the object is already cached or some node which "thinks" that it would be able to cache the object the next time it comes around. Propagating the meta-information all the way up the hierarchy may be inefficient because then the object may be wrongly cached at multiple locations because of "high" reference counts induced by drops taking place lower in the hierarchy.

When a node which has an outstanding (yet to be satisfied) request for an object gets a request for the same object, it increases its reference count for the object and decides whether it should generate some meta-information regarding the request and pass it up the hierarchy. Once again, the choice is decided by whether the node "expects" to cache the object once it gets it. The intuition behind this is to ensure that nodes see the right reference counts for objects, which would require meta-information to propagate till the point in the hierarchy where the object is cached or "expected" to be cached.

The proposed scheme to solves the problems that were pointed out in the Harvest and Harvest derived schemes. In the proposed scheme, when an object comes down the hierarchy, it is not necessarily cached at all nodes along the path. Although, an *attempt* is made to cache the object at all nodes along the path, whether an attempt at a particular node succeeds or not depends on how popular the object is at that particular node. Also, at no point is the incoming object treated any differently than objects already in the cache; the reference count of an object is its sole property that decides whether an object gets cached at a node. This is in contrast with the strategy in Harvest's scheme in which the incoming object is definitely cached at all nodes along the path. Also, unlike the Harvest scheme, our scheme makes the participating caches cooperate in order to ensure that each node sees the right reference count for an object.

Pseudo-code - Part 1: Definitions

Definitions:

OS(O): origin server for object O;
R(O): a request for the object O;
R(O, M): request for the object O from the node M;
RC(O, N): reference count for object O at node N;
LC(N): local cache at node N;
Min(C): object in cache C that has the lowest reference count;
P(N, X): the node next to N on its path to node X;
Outstanding(O, N): boolean that is TRUE if node N has already made a request for object O that is yet to be satisfied, FALSE otherwise;
V(O, N): set of nodes to which node N will have to serve object O once it receives it;
MetaInfo(O, C): packet that informs the recipient to increase the reference count for object O by C;

Boolean **WouldBeCached** (Object O, Node N)
D $\leftarrow$ Min(LC(N));
if RC(D, N) < RC(O, N) **then**
  return TRUE;
else
  return FALSE;
end if

IV. Simulation Model

To compare the performance of the new scheme henceforth referred to as Scheme II, with the scheme used in Harvest derived caches referred to as Scheme I, the operation of both schemes were simulated. The simulator is a C++ program which extensively uses LEDA [10] object libraries. The inputs to the simulator comprise the network topology specified in Graph Meta Language (GML) format and the request profile for each object.

A. Network Model

The network consists of three kinds of nodes - 1) clients - leaf level caching servers where requests enter the network, 2) origin servers - servers which are the initial repository of certain requested objects, and 3) intermediate caching servers - caching servers which lie somewhere on the path between a client and an origin server. The nodes

Pseudo-code - Part 2: Operations performed by the origin server and the client.

```
    Origin Server:
    for all incoming request R(O) do
      send down O to M
5:  end for

    Client(N):
    for all incoming request R(O) do
      RC(O, N) = RC(O, N) + 1;
10:   if (O in local cache) then
        serve O
      else if ¬ Outstanding(O, N) then
        forward R(O) to P(N, OS(O));
        Outstanding(O, N) = TRUE
15:   else if (not WouldBeCached(O, N)) then
        send MetaInfo(O, 1) to P(N, OS(O))
      else
        do nothing
      end if
20: end for

    for all object O that N receives do
      Outstanding(O, N) = FALSE;
      if (LC(N) is not full) then
25:     place O in LC(N)
      else
        D ← Min(LC(N));
        if (RC(D, N) < RC(O, N)) then
          eject D from LC(N);
30:       send DropInfo(D, RC(D)) to P(N, OS(D))
          place O in LC(N);
        end if
      end if
    end for
```

Pseudo-code - Part 3: Operations performed by an intermediate caching server.

```
    Intermediate Caching Server(C):
    for all incoming request R(O, M) do
      RC(O, C) = RC(O, C) + 1;
      if (O in LC(C)) then
5:      serve O
      else
        V(O, C) = V(O, C) ∪ M;
        if (not Outstanding(O, C)) then
          forward R(O) to P(C, OS(O));
10:       Outstanding(O, C) = TRUE;
        else if (¬ WouldBeCached(O, C)) then
          send MetaInfo(O, 1) to P(C, OS(O));
        else
          Do nothing
15:     end if
      end if
    end for
    for all object O that C receives do
      Outstanding(O, C) = FALSE;
20:   for all node v in V(O, C) do
        send O down to node v;
      end for
      V(O, C) = NULL;
      if LC(C) is not full then
25:     place O in LC(C);
      else
        D = Min(LC(C));
        if RC(D) < RC(O)) then
          eject D from LC(C);
30:       send MetaInfo(D, RC(D)) to P(C, OS(D));
          place O in LC(C)
        end if
      end if
    end for
35:
    for all MetaInfo(O, L) that C receives do
      RC(O, C) = RC(O, C) + L;
      if O in LC(C) then
        return
40:   end if
      if (¬ WouldBeCached(O, C)) then
        send MetaInfo(O, L) to P(C, OS(O));
      end if
    end for
```

are connected by bidirectional links, each having a certain bandwidth. Each node in our model also serves the function of forwarding the packets to the next hop.

A crucial aspect of the simulator is the method by which it models the transfer of objects from one node to another. Briefly, objects from a sending node are fragmented into packets, and these packets arrive in order at the destination node and are assembled there. For all the experiments, the route chosen for such transfers is the shortest path between the sender and the receiver. While the details of the model of the link and the node can be found in [25], it suffices to say that the models were carefully constructed to ensure fairness among the multiple connections.

B. *Request Generation*

The request profile for an object is a piece-wise linear graph which plots the request rate for the object over time.

The plotted request rate for the object is its request rate in the *aggregate*, i.e., the sum total of the rates at which requests are generated for the object by all the clients in the network. Once we have the aggregate request profile for an object, we break it up into per client request profiles. This is done by taking a fixed set of points from the piece-wise linear graph, and then for each point, distributing the corresponding request rate among the clients. This distribution is *pseudo-uniform*, i.e., each client's share of the aggregate request rate lies within a bounded interval of the uniform distribution value. Once we have the per client request profiles for an object, the actual request generation in the simulator takes place as follows. If at time T we need to generate a request for an object O at client C, we get the request rate for O at C at the instant T. Let this request rate be λ. Then the inter-arrival time is obtained by modeling the incoming stream of requests for O at C as a Poisson process with mean λ.

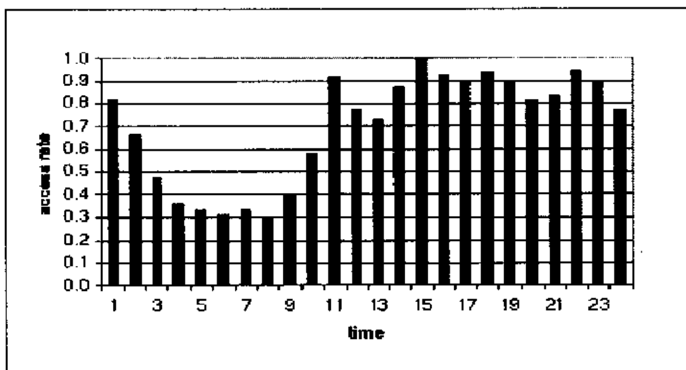

Fig. 2. Normalised object request rate over a 24 hour period at the NASA server.

We analyzed the NASA server [21] request log file, which records the exact arrival time of each request of which 20 server files were selected to cover the whole spectrum of possible request rates. We observed that the access logs show a similar request pattern for each day, as shown in Figure 2. We normalized the 24-hour access rate variation pattern and took that as the aggregate request profile for our objects. We used the 90/10 rule [3] to assign request rate to the objects. Accordingly, 10% of the objects are marked popular and aggregate request rate for the popular objects were scaled up by a factor of 81 times corresponding rate for the unpopular objects. Both popular and unpopular objects were uniformly distributed across the given number of origin servers.

C. Network Topologies

We considered three different topologies for our simulation. Topology I is a complete binary tree with 4 node levels. The root of the tree is an origin server, the leaves are the clients and the remaining nodes are caching servers. The tree topology permits a strict hierarchical ordering and we believe the differences between Scheme I and Scheme II would be the greatest in the case of such topologies. This is because of the fact that in a mesh with several different hierarchies induced by different origin servers, the "proper" place for an object in the mesh is somewhat diluted because of the conflicting choices from the different hierarchies.

In reality, however, networks are not like trees but more like meshes. We therefore ran our simulations on two such mesh networks. Figure 3 shows the first of them, which we will refer to as Topology II. This network was obtained by taking the backbone mesh of a major carrier services provider and attaching client nodes to it. As Figure 3 shows, the topology has 3 origin servers, 15 caching servers and 11 clients.

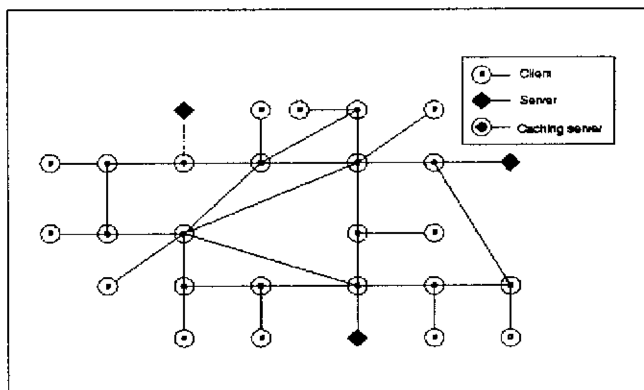

Fig. 3. Topology II

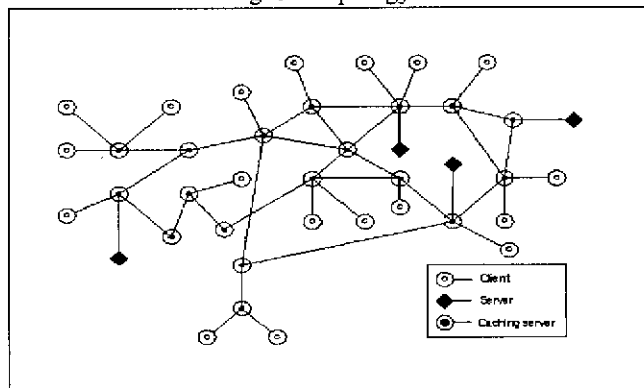

Fig. 4. Topology III

Topology III, as illustrated in Figure 4, was derived from Mapnet [22]. The raw data obtained from Mapnet's web site contains topology information on federal educational/research networks and commercial networks. To get an experimental topology of reasonable size, networks which had about 20 nodes each were identified. These topologies were then analyzed in groups of three to find the combination which maximizes the number of nodes

shared by these networks. The topology thus obtained has 40 nodes, of which 4 are origin servers, 18 are clients and the remaining 18 are caching servers.

D. Other Simulation Parameters

- In the simulation model, while all the nodes have storage capacity, only the client nodes and the intermediate nodes perform caching. The capacity of the nodes is in units of number of objects.
- The object size was fixed at 10 Kbytes. The size of the request packets and the meta-information packets in case of Scheme II was 100 bytes.
- We have used three different request rates for each comparison, corresponding to heavy load, moderate load and low load. Also, we had two settings for the number of objects served by the origin servers. For each setting of the request rate, we varied the cache size so as to cover the entire spectrum, from very low to very high caching space.

V. Performance Comparison

We considered the server-side bottleneck scenario. This was achieved by setting the bandwidths of links attached to the servers to be 80kb/s while the bandwidths of all other links were set to be 800kb/s.

A. Improvement in Latency

Figures 5 and 6 show the percent reduction in latency for Topology II for 200 and 400 objects, respectively, while Figures 7 and 8 show the same for Topology III. For 200 objects, the request rates were 0.01, 0.005, 0.0025 requests/sec for unpopular objects. For the case of 400 objects, the request rates were 0.005, 0.0025. 0.00125 request/sec for unpopular objects. As mentioned before, the request rate for the popular objects are 81 times higher. The request rate for 400 objects was scaled down to keep the load on the network at roughly the same level.

Based on the figures, the we observe that Scheme II outperforms Scheme I. For any request rate, the gains are the highest for low cache sizes and then decrease with increasing cache size. When the cache space is quite large, the two schemes perform pretty similarly. At any given request rate, the *effective* load on the network is higher in the case of low cache space because most requests get satisfied by object transfers from far-away servers. When the cache space in the network increases, many requests get satisfied by client caches or by close-by caching servers, thus, the load on the network goes down. Since the network is more heavily loaded when cache space is low, Scheme I pays a correspondingly heavier price when obtaining an object from a "sub-optimal" location. Thus, the gains of Scheme II over Scheme I are the highest when the cache

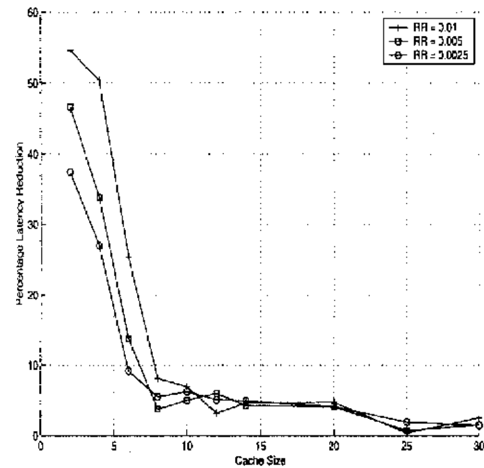

Fig. 5. Percent reduction in download time as a function of the cache size for different request rates for Topology II and 200 objects.

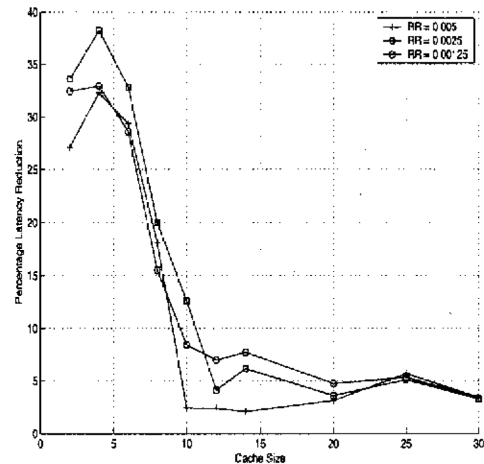

Fig. 6. Percent reduction in download time as a function of the cache size for different request rates for Topology II and 400 objects.

space in the network is low. When the cache space in the network is very large, the object distributions attained by both schemes is roughly the same, hence they perform almost similarly.

For a given load, i.e., request rate times the number of objects, as we move from 200 objects to 400 objects, we see that the gains are initially higher for the case of 200 objects but then the gains for 400 objects take over. This is expected because at very low cache sizes, in the case of 400 objects, there are a lot of objects being dropped from caches. This generates a lot of meta-information traffic adding to the congestion in Scheme II. For 200 objects this effect is lower and hence the gains are higher. However, as the cache size increases, in the case of 200 objects, most objects can be stored in nearby caches for both Schemes I and II. This is not the case with 400 objects, where because there are a large number of objects, many requests still get

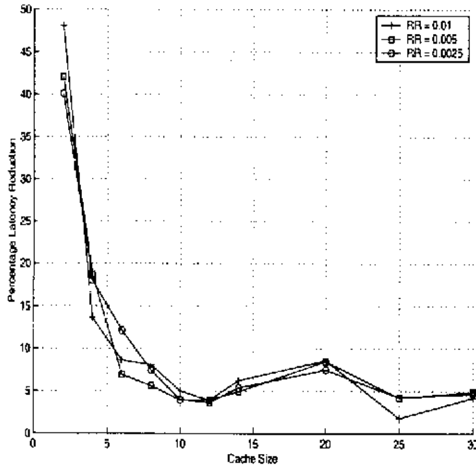

Fig. 7. Percent reduction in download time as a function of the cache size for different request rates for Topology III and 200 objects.

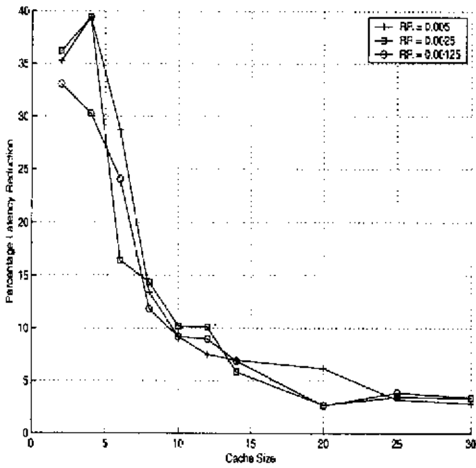

Fig. 8. Percent reduction in download time as a function of the cache size for different request rates for Topology III and 400 objects.

satisfied by caching servers many hops away. Scheme II does a better job of distributing objects across the caching mesh when cache space is low, therefore the gains for 400 objects become higher than that for 200 objects. When the cache space is very large, the gains for both 200 objects and 400 objects are almost similar, once again due to the fact that most requests get satisfied by nearby caching servers.

Finally, we note that for a given number of objects, the difference in the gains across different request rates quickly becomes small as the cache sizes increase. This is expected because even though the request rates differ by a factor two, as the cache space increases, a lot of the requests get satisfied from client caches. Hence, the effective load on the network becomes almost the same irrespective of the request rate. Thus the gains observed for the different request rates also turn out to be similar.

B. Bandwidth Gain

Figures 9 and 10 show the percent reduction in bandwidth usage for Topology II for 200 and 400 objects, respectively. The bandwidth usage, we refer to the total number of bytes shipped over all the links in the network for the satisfaction of the requests. Based on the figures, the following are the main observations.

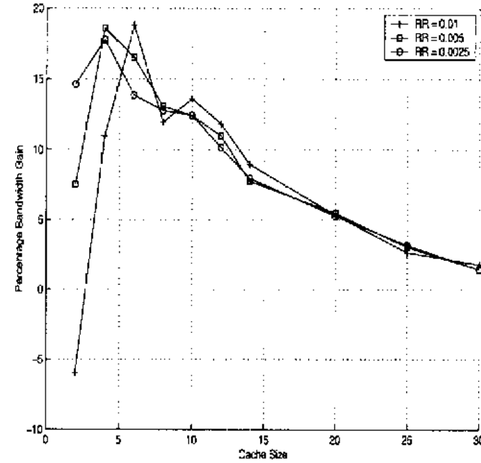

Fig. 9. Percent bandwidth gain as a function of the cache size for different request rates for Topology II and 200 objects.

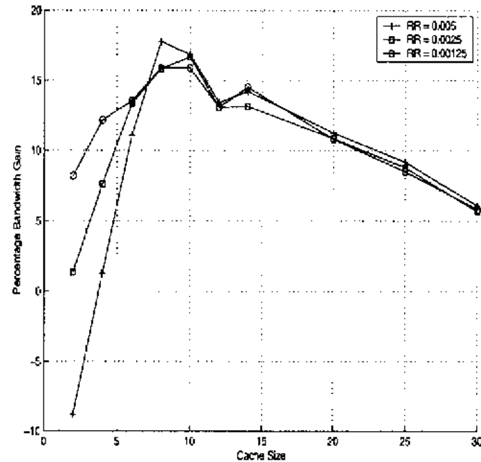

Fig. 10. Percent bandwidth gain as a function of the cache size for different request rates for Topology II and 400 objects.

We see that Scheme II is less bandwidth intensive than Scheme I, though the gains are modest in most cases. There are two underlying causes for this behavior. Firstly, even though Scheme II does a better job of keeping the right objects in the caches based on their reference counts, this is not achieved without cost. It has to send meta-information carrying packets telling nodes about the reference counts of objects in lower level caches. This adds to the amount of traffic in the network. Secondly, Scheme II does a better job of distributing the request load over

the network. This means that even though in some cases it transfers objects over a higher number of links than in Scheme I, these links are relatively lightly loaded. This brings about a reduction in the average latency, albeit using up more bandwidth.

For a given request rate and a given number of objects, we observe that the bandwidth gains keep increasing with increasing cache sizes upto a point, and then decreases. When the cache space is very low, there a large number of object drops from caches generating a lot of meta-information traffic in the network. Also, both schemes need to fetch a lot of the objects from origin servers because of the cache space crunch. As the cache space increases, Scheme II does a better job of obtaining objects from nearby servers. Moreover, the amount of meta-information traffic goes down because there are now fewer drops from caches. Thus, the bandwidth gains rise initially with increase in cache size. After a certain point, both schemes satisfy most of their request from identical locations, thus, the bandwidth gains start decreasing. When the cache space is very large, the two schemes attain almost identical distributions of objects in their caches, thus, the bandwidth gains are very small.

At a given load, i.e, request rate times the number of objects, as we move from 200 objects to 400 objects, we see that the gains are initially higher for 200 objects but then the gains for 400 objects take over. In other words, the point at which the maximum bandwidth gain is observed moves to the right as we move from 200 to 400 objects. This is because of the fact that with a higher number of objects, it takes higher cache sizes for the meta-information traffic in Scheme II to go down, and also for Scheme II to do a comparatively better job of distributing objects across caches. Thus, the point of maximum bandwidth gain is attained at higher cache sizes as we move from 200 to 400 objects.

We also note that the bandwidth gains are relatively insensitive to the request rate at moderate to high cache sizes. As discussed in Section V-A, even though the request rates are different, because of high cache sizes, most requests get satisfied by nearby caching servers, thus the effective load on the network is very similar. Thus, the gains observed for different request rates for moderate to high cache sizes are roughly similar.

Table I shows the counts of multiple copies of popular and un-popular files in the cache for the two different schemes and for different request rates. The data was calculated from the periodic snapshots of the cache contents logged by the simulator. The count of multiple copies of popular and unpopular for each snapshot were aggregated overall the snapshots to create the table. From the table we observe that Scheme II is much more efficient in the placement and replication of files in the cache hierarchy. When the cache space is low, Scheme II does not cache unpopular files[3]. Scheme I, on the other hand, not only has unpopular files when there is not enough cache space at times there are multiple copies. This translates to the better download times and the lower bandwidth usage under Scheme II.

C. *The Tree Network*

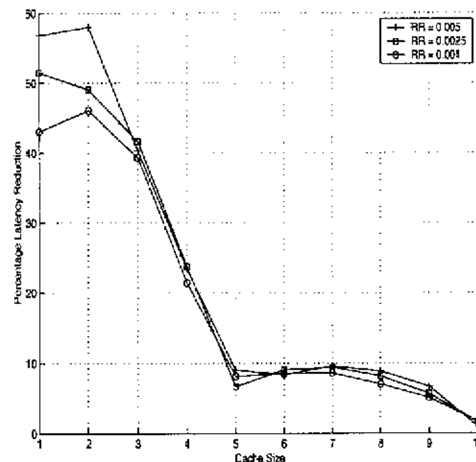

Fig. 11. Percent reduction in download time as a function of the cache size for different request rates for Topology I and 80 objects.

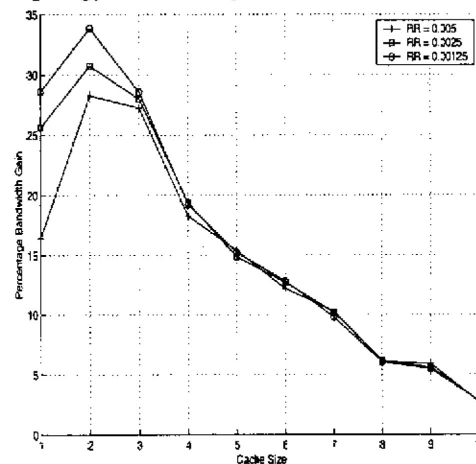

Fig. 12. Percent bandwidth gain as a function of the cache size for different request rates for Topology I and 80 objects.

We now present the results for Topology I, a 4-level complete binary tree in which the root is an origin server, the leaves are clients and the rest of the nodes are caching servers. Figures 11 and Figures 12 show the percent reduction in latency and the percent bandwidth gain for 80

[3]This data was also computed but it is not shown here due to page limitation.

TABLE I
COUNTS OF MULTIPLE COPIES OF POPULAR AND UNPOPULAR OBJECTS IN THE CACHE FOR TOPOLOGY III.

Cache	Req. Rate = 0.005				Req. Rate = 0.0025			
Size	Scheme I		Scheme II		Scheme I		Scheme II	
	Pop	Un-Pop	Pop	Un-Pop	Pop	Un-Pop	Pop	Un-Pop
2	1526	144	2178	0	1652	108	2146	0
4	3429	167	3839	0	3499	117	3845	0
6	3869	173	4053	0	3844	154	4054	0
10	4059	251	4059	0	4059	242	4059	1
12	4059	305	4059	11	4059	340	4059	7
14	4059	382	4059	28	4059	379	4059	34
20	4059	1135	4059	1001	4059	1276	4059	1015
25	4059	3491	4059	3553	4059	3507	4059	3618
30	4059	6819	4059	7756	4059	6952	4059	7736

objects for request rates of 0.005, 0.0025 and 0.001 requests/sec corresponding roughly to heavy, moderate and low load.

We observe that both the latency reduction and bandwidth gain curves show the same trends as their counterparts in Topology II and III. The interesting thing to note is that the gains are higher than those for Topology II and III. This is to be expected because the tree topology provides a case where there is a single hierarchy induced by a single origin server. Therefore, Scheme II which pushes object reference count meta-information *up* the hierarchy but not *across* it, performs best. In mesh-like networks, like Topology II and III, where the hierarchies induced by different origin servers mesh against each other, Scheme II's reference-counting based distribution of objects performs worse than in the tree case because a node may lie at different levels of the hierarchy for different hierarchies making the reference-counting distribution sub-optimal.

VI. RELATED LITERATURE

Hierarchical Web caching had its genesis in prior research on hierarchical caching in large-scale distributed file systems [2]. Evidence that several, judiciously placed file caches could reduce the volume of FTP traffic on the NSFNET backbone significantly is reported in [8]. In the context of the Web, hierarchical caching was first explored within the Harvest [6] [7] project.

Zhang, Floyd and Jacobson [20] deal with the problem of data dissemination in caching meshes. They divide the set of caching caching servers into overlapping groups and then use multicast delivery to disseminate data through the mesh. Their work however, does not explicitly tackle the problem of efficient object distribution, in their scheme, when an object is forwarded by a group through multicast, it is cached by all members of that group may be inefficient if the group size is very large.

A different approach to caching is server-initiated caching [9] [5] where the server "pushes" data intelligently towards a set of clients. In such schemes, the server maintains object reference profiles for each "client cluster" - however, it is not easy for the server to deduce topological information from the requests thereby making the task of data distribution tough. Moreover, this would further overload the already loaded Web servers, thus preventing the solution from scaling well.

Bhattacharjee [4] et. al. evaluated the benefits of associating caches with switching nodes throughout the network, rather than in a few locations. They also considered the use of various self-organizing or active cache management strategies for organizing the content of caches.

In a directory-based caching scheme [14] [15], a caching server queries a mapping server for the location of an object in the caching mesh when it does not have the object in its own local storage. If the mapping server returns with a positive response, the object is fetched from the referred server, else, the object is fetched from the origin server. Caching severs also update the mapping server when an object is added to or evicted from their cache.

At its very core, the trade-off between hierarchical caching and directory-based caching is a trade-off between space and time. Hierarchical caching stores no meta-data, all available storage space is used to cache objects. However, not storing any meta-data means that it is "dumb" when it comes to locating an object in the caching mesh, spending a relatively larger amount of time in the "search", leading to increased object fetch times. Directory-based caching, on the other hand, requires more storage space, dedicating a chunk of the same to the storage of meta-

data. This extra storage however speeds up object lookups thereby decreasing object fetch times.

It is natural to consider hybrid schemes which have both hierarchies and directories [24] - the directories are not global but manage only a part of the hierarchy. In such a hybrid scheme, on a cache miss, the node would query a mapping server for the location of the object. If the mapping server responds positively, the node fetches the object from the referred caching server, else it would forward the request for resolution to the next level of the hierarchy.

VII. Conclusion

To sustain the explosive growth of the Web, it is crucial to develop schemes that provide good quality of service to users while efficiently using the existing bandwidth of the infrastructure. Caching provides both these benefits by storing popular objects close to where the requests originate. Although, caching in itself is not a new concept, its application to the Web is. The problem is challenging primarily because of scalability reasons which has resulted in the evolution from single-point caches to caching meshes. In this work, we proposed a new data diffusion scheme for hierarchical caches which performs better than the corresponding one in Harvest derived caches. We identified the parameter space under which the benefits from our scheme were the highest and the parameter space where both the schemes performed similarly. Even though our demand-driven data diffusion scheme was studied in the context of hierarchical caching, the underlying notion of co-ordinated object management through reference counting can be applied to hybrid schemes that employ both directory-based and hierarchical caching.

References

[1] Marc Abrams, Charles Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox, "Caching Proxies: Limitations and Potentials," in *Proceedings 4th International WWW Conference*, Boston, December 1995.

[2] Rafael Alonso and Matthew Blaze, "Dynamic hierarchical caching for large-scale distributed file systems," in *Proceedings of the Twelvth International Conference on Distributed Computing Systems* June 1992.

[3] M. F. Arlitt, C. L. Williamson, "Web server workload characterization: the search for invariants," *1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Philadelphia, PA, USA, Vol. 24, No. 1:126-37, 1996.

[4] S. Bhattacharjee, K. Calvert, and E. W. Zegura, "Self-Organizing Wide-Area Network Caches," in *Proceedings of IEEE INFOCOM 98*, San Francisco, CA, March 1998.

[5] Azer Bestavros, "Demand-based document dissemination to reduce traffic and balance load in distributed information systems," in *Proceedings of the 1995 Seventh IEEE Symposium on Parallel and Distributed Processing*, San Antonio, TX, October 1995.

[6] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz, "Harvest: A scalable, customizable discovery and access system," *Technical Report CU-CS-732-94*, University of Colorado, Boulder, 1994.

[7] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz, and Kurt Worrel, "A Hierarchical Internet Object Cache," in *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, Jan 1996.

[8] Peter B. Danzig, Michael F. Schwartz, and Richard S. Hall, "A case for caching file objects inside internetworks," in *Proceedings ACM SIGCOMM 92 Conference*, Aug 1992, 281-292.

[9] James Gwertzman, Margo Seltzer, "The Case for Geographical Push-caching, " in *Proceedings of the 1995 Workshop on Hot Operating systems*, 1992.

[10] The LEDA Home Page <URL: http://www.mpi-sb.mpg.de/LEDA/leda.html>

[11] T. Berners Lee, "Propagation, replication and caching," World Wide Web Consortium, March 1995.

[12] A. Luotonen and K. Altis, "World-Wide Web Proxies", in *Computer Networks and ISDN systems*, Vol 27, 1994.

[13] Ari Luotonen, *Web Proxy servers*, Prentice Hall, 1997.

[14] Gadde, S., Chase, J., and Rabinovich, M. A taste of crispy Squid, in *Proceedings of the Workshop on Internet Server Performance (WISP'98)*, June 1998.

[15] Renu Tewari, Michael Dahlin, Harrick Vin, and John Kay, "Beyond hierarchies: Design Considerations for distributed caching on the Internet," *Technical Report TR98-04*, Department of Computer Sciences, University of Texas at Austin, February 1998.

[16] RFC2186,2187, <URL: http://ircache.nlanr.net/Cache/reading.html>

[17] Vinod Valloppillil and Keith W. Ross, "Cache Array Routing Protocol v1.0," *Internet-Draft*, June 1997. Available at http://www.etext.org/Internet/Internet-Drafts/draft-vinod-carp-v1-03.txt.

[18] James Gwertzman and Mario Seltzer, "World-wide Web Cache Consistency," in *Proceedings of the 1995 Workshop on Hot Operating Systems*, 1995.

[19] K. Worrell, "Invalidation in Large Scale Network Object Caches," *Master's Thesis*, University of Colorado, Boulder, 1994.

[20] L. Zhang, S. Floyd, and V. Jacobson, "Adaptive Web Caching," in *Proceedings of the NLANR Web Cache Workshop*, June 1997.

[21] The NASA Home Page <URL: http://www.nasa.gov>

[22] The Mapnet WebPage <URL: http://www.mapnet.com>

[23] Stephen Williams, Marc Abrams, Charles Standridge, Ghaleb Abdulla, and Edward A. Fox, "Removal Algorithms in Network Caches for World Wide Web Documents," ACM SIGCOMM 96, Stanford, CA, Aug 1996, 293-305.

[24] Rodriguez, P., Spanner, C., and Biersack, E. W., "Web caching architectures: Hierarchical and distributed caching," In *Proceedings of the 4th International Web Caching Workshop*, San Diego, April 1999.

[25] Raja Mukhopadhayay, "A New Demand-driven Data Diffusion Algorithm for Hierarchical Caching," *Master Thesis*, Department of Computer Science, University of California at Davis, Davis, CA 95616, June 1999.