Coordinated Management of Cascaded Caches for Efficient Content Distribution[*]

Xueyan Tang & Samuel T. Chanson

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
E-mail: {tangxy, chanson}@cs.ust.hk

Abstract

Large-scale content delivery systems such as the web often deploy multiple caches at different locations to reduce access latency and network traffic. These caches are usually organized in a cascaded fashion where requests not hitting a lower level cache are forwarded to a higher level cache. The performance of cascaded caching depends on how the cache contents are managed, including object placement and replacement schemes. This paper presents a general analytical framework for coordinated management of cascaded caches. The object placement problem is formulated as an optimization problem and the optimal locations for caching objects are computed by a dynamic programming algorithm. Based on the framework, we propose a novel caching scheme that incorporates both object placement and replacement strategies. The proposed scheme makes caching decisions for the set of caches lying on the delivery path of a request in a coordinated fashion. Simulation experiments based on real traces from web caches have been conducted under two different cascaded caching architectures: en-route caching and hierarchical caching. The results show that for both architectures, the proposed scheme significantly outperforms existing schemes that consider object placement or replacement at individual caches only.

1 Introduction

Caching is an important technique to improve the performance of large-scale content delivery systems such as the web [7, 22]. Due to the growing volume of concurrent web accesses, the Internet is becoming increasingly congested and many popular web sites are suffering from overload conditions. As a result, considerable latency is often experienced in retrieving web objects. The caches are located between the origin servers and the clients, often physically close to the target users. They maintain local copies of frequently accessed objects so that the requested contents have a shorter distance to travel, thereby reducing both the access latency and wide area network traffic. Moreover, with caching, the load on the origin servers is also alleviated, saving hardware and support costs for the content providers.

To obtain the full benefits of caching, multiple caches are often deployed in the network. They are usually structured in a cascaded fashion where requests are forwarded by a lower level cache to a higher level cache in case of cache miss. We refer to this type of cache organization as *cascaded caching architecture*. Examples of cascaded caching architectures include hierarchical caching [6, 21] and en-route caching [10, 15]. In hierarchical caching, the caches are arranged in a tree structure (see Figure 1). Each bottom level cache is associated with a set of clients. A client request is first sent to the bottom level cache and then iteratively forwarded up the hierarchy until the request is satisfied. If the root cache does not have the target object, the request is finally directed to the origin server. The emerging en-route caching architecture (see Figure 2) is motivated by recent advances in transparent caches [1] and their supporting techniques [14, 15, 18]. In this architecture, the caches are associated with routing nodes in the network and are called *en-route caches*. An en-route cache intercepts all client requests passing through the associated routing node. If the requested object is in the cache, the object is sent to the client and the request is not propagated further upstream. Otherwise, the routing node forwards the request along the regular routing path towards the origin server. If no en-route cache has the target object, the request is eventually serviced by the origin server. En-route caching has the advantage that it is transparent to both the origin servers and the clients. Moreover, since no request is detoured off the regular routing path, the additional bandwidth consumption and network delay for cache misses are minimized.

The overall performance of cascaded caching depends

[*]The work described in this paper was supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (Project No. HKUST6066/00E).

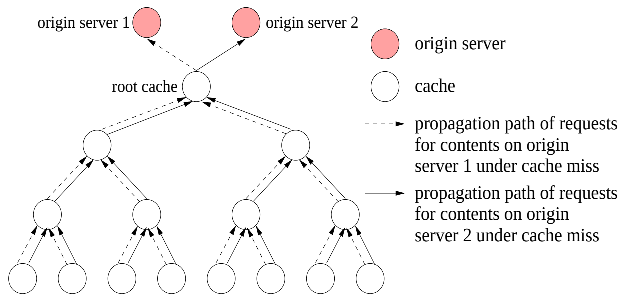

origin server 1
origin server 2
root cache
origin server
cache
propagation path of requests for contents on origin server 1 under cache miss
propagation path of requests for contents on origin server 2 under cache miss

Figure 1. Hierarchical Caching

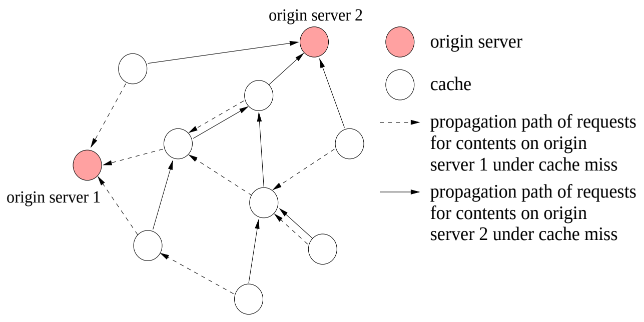

origin server 2
origin server 1
origin server
cache
propagation path of requests for contents on origin server 1 under cache miss
propagation path of requests for contents on origin server 2 under cache miss

Figure 2. En-Route Caching

on how the cache contents are managed, including object placement and replacement algorithms. From the above examples, it can be seen that normally more than one cache is located between the origin servers and the clients. Dynamically determining the appropriate number of object copies and placing them in suitable caches are challenging tasks. Allocating too many copies of unpopular objects in the network is wasteful of cache space while assigning too few copies of popular objects may not reduce their access latency sufficiently. As the cascaded caches are often geographically distributed, the network distance between caches should be taken into consideration. Moreover, when a new object is inserted into a cache, other objects may need to be removed in order to create room. The interaction effect between object placement and replacement in the set of candidate caches further complicates the problem. Existing caching schemes consider either object placement or replacement at individual caches only [3, 17] and hence are not able to produce the best possible performance.

This paper presents a general analytical framework for coordinated management of cascaded caches. We formulate the object placement problem as an optimization problem and compute the optimal locations for caching objects using a dynamic programming algorithm. Based on the framework, we propose a novel caching scheme that integrates both object placement and replacement strategies. In our approach, the objects are dynamically placed in the caches lying on the delivery path from the server to the client in a coordinated fashion. The performance of the proposed scheme has been studied by simulation experiments with two different cascaded caching architectures. Real traces collected from web caches were used in the experiments. The results show that the proposed scheme significantly outperforms existing schemes that try to optimize the placement or replacement algorithm at individual caches only.

The rest of the paper is organized as follows. Section 2 presents the analytical framework and proposes a new coordinated caching scheme. Section 3 describes the experimental setup for performance evaluation. The simulation results are presented and discussed in Section 4. Section 5 summarizes the related work, and finally, Section 6 concludes the paper.

2 Coordinated Cache Management

As seen from the examples in Section 1, the propagation mechanism of cache misses can be deployed at either the application level or the network level. Without loss of generality, we model the cascaded caching architecture as a graph $G = (V, E)$, where V is the set of nodes representing the origin servers and caches, and E is the set of links (logical and/or physical) between the nodes over which the requests are forwarded in case of cache miss. Each origin server hosts a collection of objects and the object sets associated with different servers are disjoint. For each object O, a non-negative cost $c(u, v, O)$ is associated with each link $(u, v) \in E$. It represents the cost of sending a request for object O and the associated response over the link (u, v). If a request has to travel through multiple links before reaching the target object, the access cost of the request is simply the sum of the corresponding costs on all these links. Note that our model is independent of the cost function. The term “cost” in our analysis is used in a general sense. It can be interpreted as different performance measures such as network latency, bandwidth consumption and processing cost at the cache, or a combination of these measures. Specific cost functions may depend on factors including the size of object O and/or the speed of link (u, v).

Regardless of the underlying propagation mechanism used, the propagation paths of cache misses for a given origin server are represented by a tree topology [6, 11, 15] (see Figures 1 and 2) which we shall refer to as the *distribution tree*. The distribution trees for different origin servers can be almost identical (e.g., in the case of hierarchical caching) or very different (e.g., in the case of en-route caching). Throughout this paper, the terms “delivery path” and “request path” (or “path” in short) between an origin server and a client/cache always refer to the distribution tree associated with the origin server.

Clients issue requests for objects maintained by the origin servers. Under a cascaded architecture, multiple caches are often located on the request path from the client to the origin server. Note that the access cost from a client to the

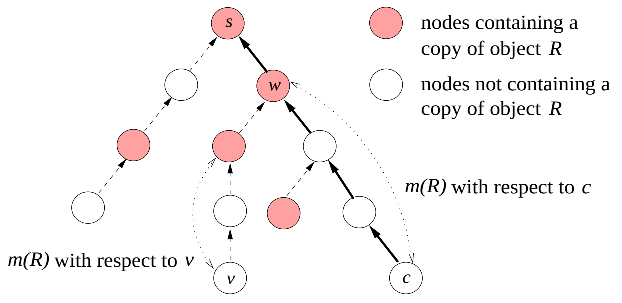

s
w
c
v
nodes containing a copy of object R
nodes not containing a copy of object R
m(R) with respect to c
m(R) with respect to v

Figure 3. Cascaded Caching

first cache on the path is independent of the cache management scheme. For simplicity, this cost is not included in the analytical model and we shall assume the request originates from the first cache. Moreover, it is known that most web objects are relatively static i.e., the access frequency is much higher than the update frequency [13]. We shall assume the objects stored in the caches are up-to-date (e.g., by using a cache coherency protocol [9] if necessary). Figure 3 shows a request originating from cache c going along a path (represented by solid edges) to the origin server s. For any pair of nodes v_1 and v_2 on the path, we say v_2 is at a higher level than v_1 if v_2 is closer to s. With cascaded caching, the request is satisfied by the node at the lowest level (denoted by w) containing the target object. Notice that w is the same as s if the requested object is not in any cache along the path between c and s. After the request reaches w, the target object (denoted by R) is sent along the same path back to the client. To reduce the cost of future accesses to object R, a copy of R can be dynamically placed in some of the caches along the path between w and c as R is being sent to the client. The issues to be investigated include: (i) which caches (if any) should object R be placed (object placement problem); and (ii) which objects to be removed from a cache if there is not enough free space (object replacement problem).

2.1 Problem Formulation

The object placement problem is trivial if cache sizes are infinite, in which case all available objects can be stored in every cache to minimize the total access cost. However, infinite cache sizes are impractical for very large content delivery systems such as the web. Due to limited cache space, one or more objects may need to be removed from the cache when a new object is inserted. Removing an object increases its access cost (referred to as *cost loss*) while inserting an object decreases its access cost (referred to as *cost saving*). The object placement problem under the cascaded caching architecture is further complicated by *caching dependencies*, i.e., a placement decision at one node affects the performance gain of caching the same object at other nodes. The optimal locations to cache an object depends on the potential cost loss and cost saving at every node along the delivery path. Our objective is to minimize the total access cost of all objects.

We start by computing the cost saving and the cost loss of caching an object at individual nodes. Consider a node v. Let $f(O)$ be the *access frequency* of object O observed by node v, i.e., the rate of requests passing through v and targeting for O. Let $m(O)$ be the *miss penalty* of object O with respect to v. The miss penalty is defined as the additional cost of accessing the object if it is not cached at v. In our cascaded caching model, $m(O)$ is given by

$$m(O) = \sum_{(u_1,u_2)\in PATH(v,v')} c(u_1,u_2,O),$$

where v' is the nearest higher level node of v that caches O, and $PATH(v,v')$ is the set of links on the path between v and v' (see Figure 3).

Let R be the requested object. Clearly, the cost saving of caching R at v is $f(R)\cdot m(R)$.

Computing the cost loss of caching R at v is a bit more complicated. Let $O_1, O_2, \ldots, O_k$ be the objects currently cached at v. The cost loss introduced by removing object O_i from the cache is given by $f(O_i)\cdot m(O_i)$. Obviously, the purged objects should introduce the least total cost loss while creating enough space to accommodate R. This is equivalent to the knapsack problem and is NP-complete. A fast greedy heuristic of the knapsack problem that works well in practice is as follows. Notice that the normalized cost loss (NCL, i.e., the cost loss introduced by creating one unit of free space) of purging O_i is $\frac{f(O_i)\cdot m(O_i)}{s(O_i)}$, where $s(O_i)$ is the size of object O_i. The objects in the cache are ordered by their NCLs, and the replacement candidates are selected sequentially starting from the object with the smallest NCL until sufficient space is created for R. The cost loss of caching R at v is computed by summing up the cost losses introduced by all selected objects.

We now formulate the object placement problem on the path between w and c. Consider the snapshot when a request for object R is being served (see Figure 4). Let $A_0 = w$ be the origin server or the high level cache satisfying the request, $A_n = c$ be the node where the request originates, and $A_1, A_2, \ldots, A_{n-1}$ be the intermediate caches on the path from A_0 to A_n.

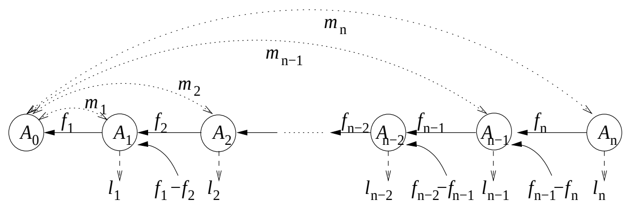


Figure 4. System Model

Let m_i denote the miss penalty of object R with respect to A_i, then $m_i = \sum_{j=1}^{i} c(A_{j-1}, A_j, R)$. Let f_i be the access frequency of object R at A_i. Since requests for R that go through A_i must also pass through $A_{i-1}, A_{i-2}, \ldots, A_1$, we have $f_1 \geq f_2 \geq \cdots \geq f_n$. Suppose R is cached at r intermediate nodes $A_{v_1}, A_{v_2}, \ldots, A_{v_r}$ where $r \geq 0$ and $1 \leq v_1 < v_2 < \cdots < v_r \leq n$. Taking into consideration the caching dependencies of R along the path, the total cost saving is given by $\sum_{i=1}^{r}((f_{v_i} - f_{v_{(i+1)}}) \cdot m_{v_i})$, where $f_{v_{(r+1)}}$ is set to 0. Now let l_i be the cost loss of evicting objects at node A_i to create enough space for R, then the total cost loss is $\sum_{i=1}^{r} l_{v_i}$. Therefore, the overall reduction in the total access cost of all objects is given by

$$\sum_{i=1}^{r}((f_{v_i} - f_{v_{(i+1)}}) \cdot m_{v_i} - l_{v_i}). \tag{1}$$

Our objective is to place object R in a subset of the caches $\{A_1, A_2, \ldots, A_n\}$ that maximizes the cost reduction (1), thereby minimizing the total access cost.

2.2 Dynamic Programming Solution

For the purpose of analysis, we first provide a generalized definition of the problem.

Definition 1 Given $f_1, f_2, \ldots, f_n, f_{n+1}$; $m_1, m_2, \ldots, m_n$; and $l_1, l_2, \ldots, l_n$ ($n > 0$), where $f_1 \geq f_2 \geq \cdots \geq f_n \geq f_{n+1} = 0$, $m_i \geq 0$ and $l_i \geq 0$ $(i = 1, 2, \ldots, n)$. Let k be an integer such that $0 \leq k \leq n$, and $v_1, v_2, \ldots, v_r$ be a set of r integers such that $1 \leq v_1 < v_2 < \cdots < v_r \leq k$. The objective function $\Delta cost(k : v_1, v_2, \ldots, v_r)$ is defined as

$$\Delta cost(k : v_1, v_2, \ldots, v_r) = \sum_{i=1}^{r}((f_{v_i} - f_{v_{(i+1)}}) \cdot m_{v_i} - l_{v_i}),$$

where $f_{v_{(r+1)}} = f_{k+1}$. If r = 0, define $\Delta cost(k : \phi) = 0$. Finding r and $v_1, v_2, \ldots, v_r$ that maximize $\Delta cost(k : v_1, v_2, \ldots, v_r)$ is referred to as the k-optimization problem. □

The object placement problem formulated in Section 2.1 is simply an n-optimization problem i.e., maximizing $\Delta cost(n : v_1, v_2, \ldots, v_r)$. In the following, we develop a dynamic programming algorithm to solve the problem. Theorem 1 proves that the optimal solution to the problem must contain optimal solutions to some subproblems.

Theorem 1 Let $1 \leq k \leq n$ and $r > 0$. Suppose $v_1, v_2, \ldots, v_r$ is an optimal solution to the k-optimization problem, and $u_1, u_2, \ldots, u_{r'}$ is an optimal solution to the $(v_r - 1)$-optimization problem, then $u_1, u_2, \ldots, u_{r'}, v_r$ is also an optimal solution to the k-optimization problem.

Proof: By definition,

$$\begin{array}{rl} & \Delta cost(v_r - 1 : u_1, u_2, \ldots, u_{r'}) \\ \geq & \Delta cost(v_r - 1 : v_1, v_2, \ldots, v_{(r-1)}). \end{array}$$

Therefore,

$$\begin{array}{rl} & \Delta cost(k : u_1, u_2, \ldots, u_{r'}, v_r) \\ = & (f_{u_1} - f_{u_2}) \cdot m_{u_1} - l_{u_1} + (f_{u_2} - f_{u_3}) \cdot m_{u_2} - l_{u_2} \\ & + \cdots + (f_{u_{r'}} - f_{v_r}) \cdot m_{u_{r'}} - l_{u_{r'}} \\ & + (f_{v_r} - f_{k+1}) \cdot m_{v_r} - l_{v_r} \\ = & \Delta cost(v_r - 1 : u_1, u_2, \ldots, u_{r'}) \\ & + (f_{v_r} - f_{k+1}) \cdot m_{v_r} - l_{v_r} \\ \geq & \Delta cost(v_r - 1 : v_1, v_2, \ldots, v_{(r-1)}) \\ & + (f_{v_r} - f_{k+1}) \cdot m_{v_r} - l_{v_r} \\ = & (f_{v_1} - f_{v_2}) \cdot m_{v_1} - l_{v_1} + (f_{v_2} - f_{v_3}) \cdot m_{v_2} - l_{v_2} \\ & + \cdots + (f_{v_{(r-1)}} - f_{v_r}) \cdot m_{v_{(r-1)}} - l_{v_{(r-1)}} \\ & + (f_{v_r} - f_{k+1}) \cdot m_{v_r} - l_{v_r} \\ = & \Delta cost(k : v_1, v_2, \ldots, v_{(r-1)}, v_r). \end{array} \tag{2}$$

On the other hand, since $v_1, v_2, \ldots, v_r$ is an optimal solution to the k-optimization problem,

$$\begin{array}{rl} & \Delta cost(k : u_1, u_2, \ldots, u_{r'}, v_r) \\ \leq & \Delta cost(k : v_1, v_2, \ldots, v_{(r-1)}, v_r). \end{array} \tag{3}$$

Combining (2) and (3), we have

$$\begin{array}{rl} & \Delta cost(k : u_1, u_2, \ldots, u_{r'}, v_r) \\ = & \Delta cost(k : v_1, v_2, \ldots, v_{(r-1)}, v_r). \end{array}$$

Hence, the theorem is proven. □

We need the following definitions before presenting the recurrences for dynamic programming.

Definition 2 Let $0 \leq k \leq n$. Define OPT_k to be the maximum value of $\Delta cost(k : v_1, v_2, \ldots, v_r)$ obtained in the k-optimization problem, and L_k is the largest index in the optimal solution. If the optimal solution is an empty set, define $L_k = -1$. □

Obviously, $OPT_0 = 0$ and $L_0 = -1$. From Theorem 1, we know that if $L_k > 0$,

$$OPT_k = OPT_{(L_k - 1)} + (f_{L_k} - f_{k+1}) \cdot m_{L_k} - l_{L_k}.$$

Hence, we can check all possible locations of L_k and select the one that maximizes the objective function when calculating OPT_k. Therefore, we have

$$\left\{ \begin{array}{lll} OPT_0 & = & 0 \\ OPT_k & = & max\{0,\ OPT_{i-1} + (f_i - f_{k+1}) \cdot m_i - l_i \\ & & (i = 1, 2, \ldots, k)\}, \qquad \text{for } 1 \leq k \leq n, \end{array} \right.$$

and

$$\left\{ \begin{array}{l} L_0 = -1 \\ L_k = \left\{ \begin{array}{ll} v & \text{if index } v \ (1 \leq v \leq k) \text{ satisfies } OPT_k \\ & \quad = OPT_{v-1} + (f_v - f_{k+1}) \cdot m_v - l_v, \\ -1 & \text{if } OPT_k = 0. \end{array} \right. \end{array} \right.$$

The original object placement problem (i.e., the n-optimization problem) can be solved using dynamic programming with these recurrences. After computing OPT_k and L_k ($0 \le k \le n$), we can start from $v_r = L_n$ and obtain all indices in the optimal solution by setting $v_i = L_{(v_{(i+1)}-1)}$ iteratively for all $1 \le i < r$. Theorem 1 ensures the correctness of this calculation procedure.

2.3 Coordinated Caching Scheme

In this subsection, we present a new coordinated caching scheme based on the previous analysis.

Every cache maintains some meta information on the objects in the form of object descriptors. An object descriptor contains the object size, the access frequency (and/or the timestamps of recent accesses) and the miss penalty of the object with respect to the associated node. When a request is issued at node A_n for object R, each node A_i on the path between A_0 and A_n piggybacks the corresponding information f_i, m_i and l_i on the request message as it passes through the node. When the request arrives at A_0, A_0 computes the optimal locations to place the requested object based on the piggybacked information using the dynamic programming algorithm, and sends the decision together with the object back to node A_n. Along the way to A_n, the intermediate nodes on the path adjust their cache contents according to the caching decision. If the object is instructed to be cached at A_i, A_i executes the greedy heuristic given in Section 2.1 to select replacement candidates and updates its cache accordingly.

Since the contents of the caches change over time, the access frequency and miss penalty of an object with respect to a node need to be updated from time to time. The access frequency can be estimated based on recent request history which is locally available (e.g., by using a "sliding window" technique [17]). The miss penalty is updated by the response messages. Specifically, a variable with an initial value of 0 is attached to each object in the response message sent through a delivery path. At each intermediate node along the way, the variable is increased by the cost of the last link the object has just traversed. This value is then used to update the miss penalty of the object maintained by the associated cache. If the object is inserted into the cache, the variable is reset to 0 before the object is forwarded downstream. In this way, the miss penalties of the requested object are updated at all caches on the delivery path. To avoid unnecessary communication overhead, the miss penalty changes of the requested object at caches not along the path are not updated immediately. The same is true for miss penalty changes caused by object removals that may result from the insertion of the requested object. These changes would be discovered by the associated caches upon subsequent object requests/responses. Note that no additional message exchange or probing operation is used for information update. Therefore, the communication overhead in deploying coordinated caching is small.

2.4 Discussion

The size of an object descriptor is typically a few tens of bytes and is negligible compared to the average size of objects. Hence, the storage overhead of maintaining the descriptors of cached objects is very small. However, the descriptors of objects not stored in the cache are also needed for the purpose of evaluating their cost savings. Fortunately, it is not necessary to keep all of these descriptors in the cache as discussed below. The following theorem presents an important property of the coordinated caching scheme.

Theorem 2 The optimal solution $\{v_1, v_2, \ldots, v_r\}$ of the n-optimization problem satisfies the following inequalities:

$$f_{v_i} \cdot m_{v_i} \ge l_{v_i}, \quad i = 1, 2, \ldots, r.$$

Proof: Assume on the contrary that v_i is an index in the optimal solution of the n-optimization problem such that $f_{v_i} \cdot m_{v_i} < l_{v_i}$.

If $i = 1$, we have

$$\begin{aligned}
& \Delta cost(n : v_1, v_2, \ldots, v_r) \\
= \; & (f_{v_1} - f_{v_2}) \cdot m_{v_1} - l_{v_1} + \Delta cost(n : v_2, v_3, \ldots, v_r) \\
\le \; & f_{v_1} \cdot m_{v_1} - l_{v_1} + \Delta cost(n : v_2, v_3, \ldots, v_r) \\
< \; & \Delta cost(n : v_2, v_3, \ldots, v_r).
\end{aligned}$$

If $i > 1$, we have

$$\begin{aligned}
& \Delta cost(n : v_1, v_2, \ldots, v_r) \\
= \; & \Delta cost(v_{(i-1)} - 1 : v_1, v_2, \ldots, v_{(i-2)}) \\
& +(f_{v_{(i-1)}} - f_{v_i}) \cdot m_{v_{(i-1)}} - l_{v_{(i-1)}} + (f_{v_i} - f_{v_{(i+1)}}) \cdot m_{v_i} \\
& -l_{v_i} + \Delta cost(n : v_{(i+1)}, v_{(i+2)}, \ldots, v_r) \\
\le \; & \Delta cost(v_{(i-1)} - 1 : v_1, v_2, \ldots, v_{(i-2)}) \\
& +(f_{v_{(i-1)}} - f_{v_i}) \cdot m_{v_{(i-1)}} - l_{v_{(i-1)}} + (f_{v_i} \cdot m_{v_i} - l_{v_i}) \\
& +\Delta cost(n : v_{(i+1)}, v_{(i+2)}, \ldots, v_r) \\
< \; & \Delta cost(v_{(i-1)} - 1 : v_1, v_2, \ldots, v_{(i-2)}) \\
& +(f_{v_{(i-1)}} - f_{v_i}) \cdot m_{v_{(i-1)}} - l_{v_{(i-1)}} \\
& +\Delta cost(n : v_{(i+1)}, v_{(i+2)}, \ldots, v_r) \\
\le \; & \Delta cost(v_{(i-1)} - 1 : v_1, v_2, \ldots, v_{(i-2)}) \\
& +(f_{v_{(i-1)}} - f_{v_{(i+1)}}) \cdot m_{v_{(i-1)}} - l_{v_{(i-1)}} \\
& +\Delta cost(n : v_{(i+1)}, v_{(i+2)}, \ldots, v_r) \\
= \; & \Delta cost(n : v_1, v_2, \ldots, v_{(i-1)}, v_{(i+1)}, \ldots, v_r).
\end{aligned}$$

This implies that v_i can be removed from the solution $\{v_1, v_2, \ldots, v_r\}$ to obtain a higher value of function $\Delta cost$ in the n-optimization problem, which contradicts with the optimality of $\{v_1, v_2, \ldots, v_r\}$. Hence, the theorem is proven. □

Theorem 2 implies that the coordinated caching scheme only needs to consider placing objects in the caches where the replacement operation is *locally* beneficial (i.e., the cost saving outweighs the cost loss with respect to the single cache). Since both the miss penalty and the cost loss generally increase with the size of the requested object, it implies that the descriptors of objects with low access frequencies are less important for computing the optimal caching locations. Based on this observation, we propose to allocate a small auxiliary cache (we shall call it the *d-cache*) at each node to keep the descriptors of the most frequently accessed objects not in the main cache. The size of the d-cache is negligible compared to the main cache which stores the objects. If the requested object is not instructed to be cached at A_i and its descriptor is not in the d-cache, the descriptor is inserted into the d-cache at A_i when the object passes through. Simple LFU replacement policy can be used to manage the object descriptors in the d-caches.

If an intermediate node does not have the descriptor of the requested object in its d-cache, it would indicate this fact by attaching a special tag to the request message going upstream. Based on the attached information, A_0 removes nodes from the candidate set whose d-caches do not contain the object descriptor. The dynamic programming algorithm is then applied to the remaining nodes to compute the optimal locations for caching. The rationale behind this strategy is that the requested object is not frequently accessed at the excluded nodes compared to the other objects, and hence removing them from the candidate set would not affect the computation of optimal caching locations significantly.

It is easy to see that the time complexity of the dynamic programming algorithm is $O(k^2)$, where k is the number of nodes on the path that have the descriptor of the requested object in their d-caches ($k \leq n$). We expect that k is not very large in practice. This is because n is small for popular objects as they would be cached in the network with high density. On the other hand, the descriptors of unpopular objects would not be cached by most intermediate nodes resulting in small k. Therefore, the cost of computing the optimal locations to cache a requested object is low. Moreover, the overhead in maintaining object descriptors can be kept small by using judicious data structures. For example, descriptors of cached objects can be organized as a heap based on their normalized cost losses. In this way, the time complexity for each adjustment (e.g., insertion and removal) is $O(log\ m)$, where m is the number of cached objects. Object descriptors in the d-cache can be organized into one or more LRU stacks if the access frequencies of objects are estimated using a "sliding window" technique (see Section 3.2). As a result, the time complexity for each insertion and removal in the d-cache is $O(1)$.

3 Experimental Setup

We have performed simulation experiments based on real traces collected from web caches. The objective of the experiments is to compare the *relative performance* of different caching schemes. This section outlines the traces, system architectures and caching schemes used in our simulation.

3.1 Traces

We made use of the Boeing proxy traces [12] to simulate the request streams generated by the clients in our experiments. The traces were collected between 3-1-1999 and 3-5-1999. As the Boeing network contains a set of 5 proxies, complete daily traces (five traces, one per day) were first obtained by merging the traces collected at individual proxies based on the request timestamps. The number of requests logged in each daily trace was around 22 million and they were issued by more than 60,000 clients. Each trace entry includes the request time, the client, the target URL as well as the associated origin server, the size of the target object, and other information about the requests. The target URLs, origin servers and clients have been mapped to unique ID numbers to protect privacy[1]. However, performing the simulation experiments using the original traces far exceeds the memory capacity of our workstations (Sun Ultra-SPARC 5 with 256 MB memory). To overcome this, we extracted a subtrace from each daily trace to drive our simulation. The subtrace consists of requests for the most popular 100,000 objects, covering more than 50% of the requests in the original daily trace. Note that the extraction would not change the *relative* access frequencies of the objects. It is known that the access pattern of web objects follows Zipf-like distributions [4]. That is, the access frequency of the ith most popular object is proportional to $1/i^\theta$, where $\theta > 0$ is the Zipf parameter. Obviously, the requests in the subtraces follow the Zipf-like distribution with the same Zipf parameter as that of the original traces. Therefore, we believe that the results are valid for the purpose of comparing the *relative performance* of different caching schemes.

Each simulation run started with all caches being empty. The first half of the trace was considered the start up period and was run to bring the system into a relatively steady state. Statistics were collected for the second half of the trace only. Similar performance trends have been observed for different daily traces. Due to space limitation, we shall report only the experimental results of the 3-1-1999 trace in this paper.

[1]Since the mappings are not consistent from day to day, the duration of simulation runs is limited to one daily trace only.

3.2 System Architectures

Our experiments were conducted under two different cascaded caching architectures: en-route caching and hierarchical caching (see Section 1).

In the en-route architecture, the network topology was randomly generated using the Tiers program [5]. The network topology consisted of a wide area network (WAN) and a number of metropolitan area networks (MAN[2]), in Tiers terminology. We have performed experiments for a wide range of different network topologies. Although the absolute behavior changes somewhat for various topologies, the relative performance trends described in this paper hold for all experiments conducted. Due to space limitation, only the results of one sample topology are reported in Section 4. The characteristics of this topology are listed in Table 1.

Parameter	Value
Total number of nodes	100
Number of WAN nodes	50
Number of MAN nodes	50
Number of network links	173
Average delay of WAN links	0.146 second
Average delay of MAN links	0.018 second

Table 1. System Parameters for En-Route Architecture

The WAN is regarded as the backbone network, and no origin server or client is directly attached to the WAN nodes. The origin servers and clients are assumed to be co-located with the MAN nodes. In our experiments, the origin servers seen in the traces are randomly allocated to the MAN nodes in the simulated network. Moreover, the clients are also randomly assigned to the MAN nodes, and the requests issued by a client are assumed to originate from the associated MAN node. An en-route cache is associated with every WAN and MAN node in the network. Each network link connects two nodes and represents one hop in the network. For simplicity, the delay caused by sending a request and the associated response over a network link is set proportionally to the size of the requested object. The delays generated by the Tiers program for the network links are taken to be the delays of an average size object. The ratio of the average delays of WAN links and MAN links for the sample topology is approximately 8:1 (see Table 1). Routing paths from all nodes to a given origin server are set to the shortest-path tree rooted at the server (i.e., the associated MAN node). The average length of the routing path between an origin server and a client is about 12 hops.

In the hierarchical architecture, the network topology is represented by a full O-ary tree where every internal node has a fanout degree of O (see Figure 5). A cache is associated with every node in the tree. The origin servers are connected to the root of the tree while the clients are associated with the leaf nodes. In our experiments, the clients seen in the traces are randomly allocated to the leaf nodes in the tree. The default depth of the tree is set to 4 and the default fanout degree O is set to 3. Similar to the en-route architecture, the delay of sending a request and the associated response over a link is set proportionally to the object size. Suppose the level of the root node is 3 and the level of the leaf nodes is 0 (see Figure 5). The average delay of a network link is assumed to grow exponentially with the level of its lower end. Specifically, let d be the base delay and g be the growth factor. The average delay between a level-i node and its parent (located at level-$(i+1)$) is given by $g^i \cdot d$ $(i = 0, 1, 2)$. Moreover, the average delay between the root node and an origin server is set to $g^3 \cdot d$. The default values of d and g are set to 0.008 second and 5 respectively. We have tested a wide range of d and g values and observed similar trends in the relative performance of different caching schemes.

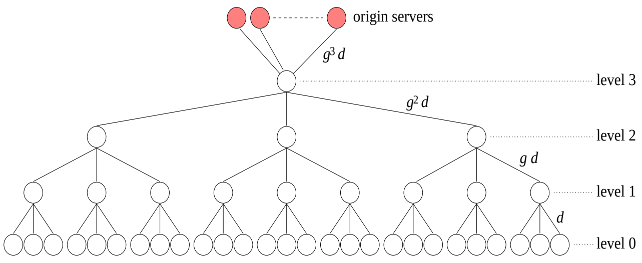


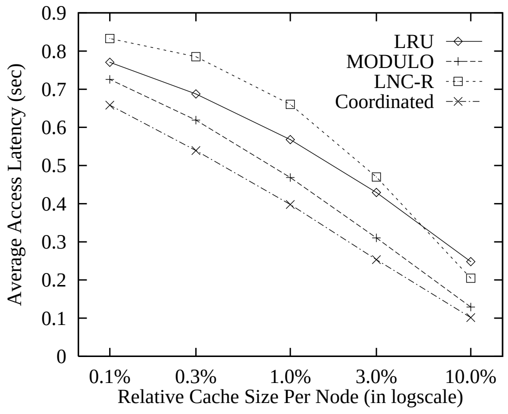
Figure 5. Network Topology for Hierarchical Architecture

Similar to other studies [4, 8, 17], the main cache size at each node is described relative to the total size of all objects (we shall call it the *relative cache size*). The d-cache size at each node is measured in terms of the number of object descriptors. We have performed experiments for different d-cache sizes relative to the main cache size and found that the results were similar when the main cache and d-cache were capable of accommodating the same order of objects and object descriptors respectively. By default, the d-cache size is set to 3 times the average number of objects the main cache can hold[3].

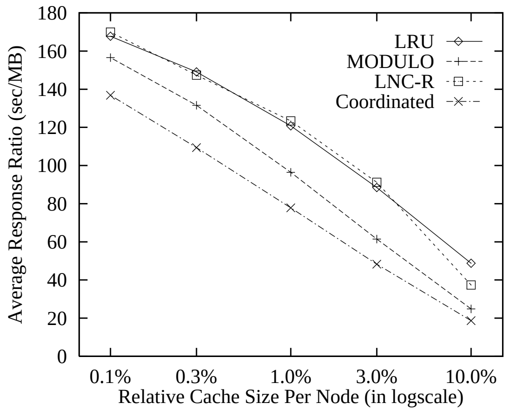
To make the caching schemes less sensitive to transient workload, a "sliding window" technique is employed to dynamically estimate the access frequency of an object [17]. Specifically, for each object, up to K most recent reference times are recorded in the descriptor and the access frequency is computed by $f(O) = \frac{\mathcal{K}}{t - t_{\mathcal{K}}}$, where $\mathcal{K} \le K$ is the number of references recorded, t is the current time and $t_{\mathcal{K}}$ is the $\mathcal{K}$th most recently referenced time. K is set to 3 in our simulation experiments [17]. To reduce the overhead,

[2]The term "MAN" is used in Tiers [5]. Essentially it applies to any high-speed interconnected local area networks.

[3]Note that this ratio is not equal to the ratio of their physical storage capacities. The capacity of the d-cache is much smaller than that of the main cache.



(a) Average Access Latency



(b) Average Response Ratio

Figure 6. Access Latency and Response Ratio vs. Cache Size (En-Route Architecture)

the access frequency estimate of an object is only updated when the object is referenced, and at reasonably large intervals (currently set at 10 minutes) to reflect aging.

3.3 Caching Schemes

In our experiments, the access cost function in the coordinated caching scheme is taken to be the link delay. This implies the generic cost in the analytical model (see Section 2) is interpreted as the access latency. To study the performance of the coordinated caching scheme, the following existing algorithms were included in the experiments for comparison purposes. Note that none of these schemes optimize placement and replacement strategies in an integrated fashion.

- LRU: This is a standard caching algorithm. The requested object is cached by every node through which the object passes. If there is not enough free space, the cache purges one or more least recently referenced objects to make room for the new object. No d-cache is used in this scheme.

- MODULO [3]: This is a modified LRU scheme that employs a simple placement optimization. On the delivery path from the origin server (or the high level cache) to the client, the object is cached at the nodes that are a fixed number (called *cache radius*) of hops apart. The caches use LRU replacement policy to remove objects when necessary. Similar to LRU, no d-cache is used in this scheme. Note that a cache radius of 1 degenerates the MODULO scheme to the LRU scheme.

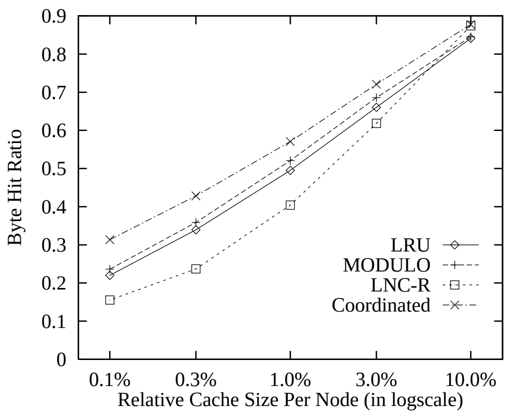
- LNC-R [16]: This is a cost-based caching algorithm shown to be effective in the context of a single web cache. It optimizes cache replacement by removing objects with the least normalized cost losses (i.e, $\frac{f(O)\cdot m(O)}{s(O)}$). Similar to LRU, the requested object is cached by all nodes along the delivery path and accordingly, for each intermediate cache, the miss penalty of the object is set to the delay associated with the immediate upstream link. Similar to the coordinated caching scheme, the descriptors of the objects not in the main cache are maintained in the d-cache to allow better estimation of access frequencies.

4 Performance Results

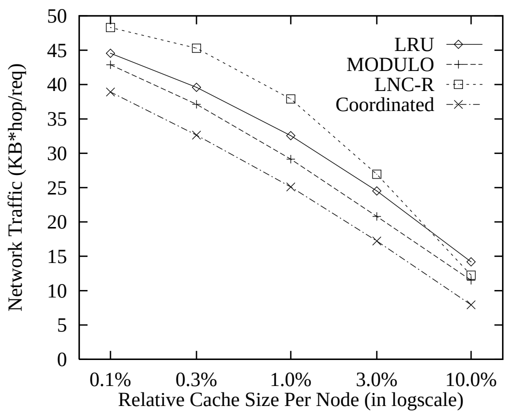
4.1 En-Route Caching Architecture

First, we compare the performance of different caching schemes under the en-route architecture. Figure 6(a) shows the average access latency as the relative cache size at each node increases from 0.1% to 10%. Since the objects have very different sizes, we also plotted in Figure 6(b) the average response ratio. The response ratio of a request is defined as the ratio of its access latency to the size of the target object. The lower the average response ratio, the better the performance. This metric is more objective as the effect of object size is eliminated. Moreover, users are likely to expect short delays for small objects and willing to tolerate longer delays for larger objects. We have conducted experiments with different cache radii for the MODULO scheme and found that a cache radius of 4 gives the best performance under our experimental settings. Thus, a cache radius of 4 was used in the experiments reported in this section for MODULO.

As can be seen, all caching schemes provide steady performance improvement as the cache size increases. The coordinated scheme significantly reduces the average access latency and response ratio compared to the other schemes examined. This shows the importance of managing object placement and replacement strategies in an integrated fashion. To achieve the same access latency, the schemes that



(a) Byte Hit Ratio



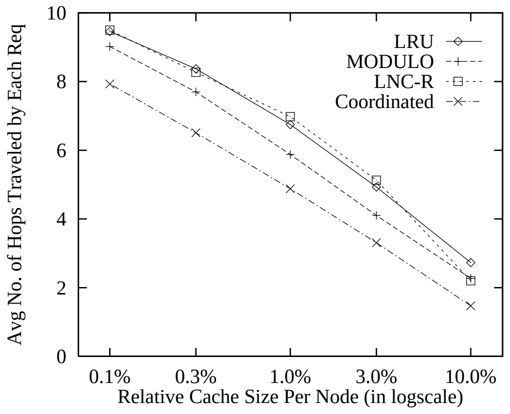
(b) Network Traffi c

Figure 7. Hit Ratio and Network Traffic vs. Cache Size (En-Route Architecture)

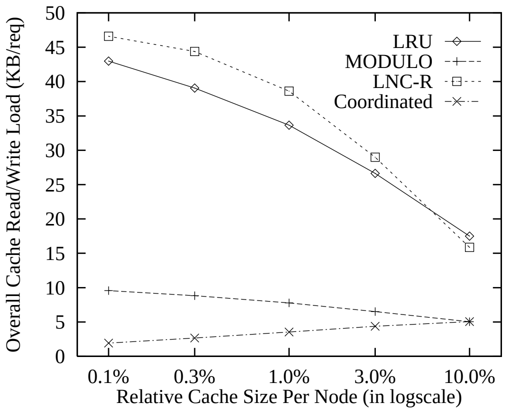
do not optimize placement decisions (LRU and LNC-R) would require 3 to 10 times the cache space of the coordinated scheme (note that the cache size axis is in logscale). Although LNC-R incorporates various cost factors to optimize cache replacement decisions, its performance is similar to or sometimes even worse than that of LRU. This is not surprising because cache contents change very frequently due to frequent replacements if the requested objects are inserted into all intermediate caches on the delivery path. Therefore, the miss penalties of cached objects change very frequently under LNC-R and are less helpful in optimizing replacement decisions. The relative improvement of coordinated caching over LRU and LNC-R increases with cache size. Coordinated caching outperforms LRU and LNC-R by more than 60% and 50% respectively in average access latency at a relative cache size of 10%. On the other hand, even though MODULO utilizes cache space more efficiently than LRU and LNC-R by storing the same object at fewer locations, Figures 6(a) and (b) show that it still performs much worse than coordinated caching over a wide range of cache sizes. This is because the placement decision of MODULO is not based on the access frequencies of objects so that popular and unpopular objects are cached with the same density in the network. The impact of MODULO's disadvantage is less significant as the cache size increases. Therefore, the absolute performance difference between coordinated caching and MODULO decreases with increasing cache size. However, the relative improvement of coordinated caching remains significant for large cache sizes. From Figure 6, at a cache size of 10%, the coordinated scheme outperforms MODULO by about 25% in terms of average access latency and response ratio. It is worthwhile pointing out that the performance of MODULO with cache radii other than 4 could be much worse than that reported here. Moreover, from our experiments with many other system configurations, the cache radius to achieve the best possible performance of MODULO is different for different system configurations, making automated deployment difficult. This will be made clear in Section 4.2.

Figure 7(a) plots the byte hit ratio curves as a function of relative cache size for different caching schemes. To capture the system's caching behavior, the byte hit ratio is defined as the ratio of the number of bytes served by the caches as a whole (as opposed to those served by the origin servers) to the total number of bytes requested. It provides an indication on the amount of load reduction at the origin servers. By making optimal caching decisions along delivery paths, the coordinated scheme greatly improves the byte hit ratio over the other schemes examined, especially for smaller cache sizes. This implies substantial load reduction at the origin servers. In contrast, MODULO and LNC-R, which optimize object placement or replacement only, have similar and/or worse performance compared to LRU. When the cache size is very large, all schemes are capable of caching popular objects somewhere in the network. Therefore, as shown in Figures 7(a), the relative improvement of the coordinated scheme in byte hit ratio decreases with increasing cache size. At large cache sizes, the main advantage of coordinated caching is to bring popular objects closer to the clients by judicious placement of objects in the caches. Figures 7(b) shows the average network traffic (measured in byte$\times$hops) required to satisfy a request. It can be seen that the coordinated caching scheme results in considerably lower network traffic than the other schemes. Coordinated caching reduces the network traffic by 44%, 31% and 35% compared to LRU, MODULO and LNC-R respectively at a cache size of 10%.

The workload of the caches consists of two parts: (i) looking up the requested objects and forwarding requests in case of cache miss; and (ii) reading/writing objects from/into the cache. It is obvious that the lower the cache load, the more scalable the caching scheme. Under the en-route caching architecture, the overhead of object look-up and request forwarding is proportional to the number of

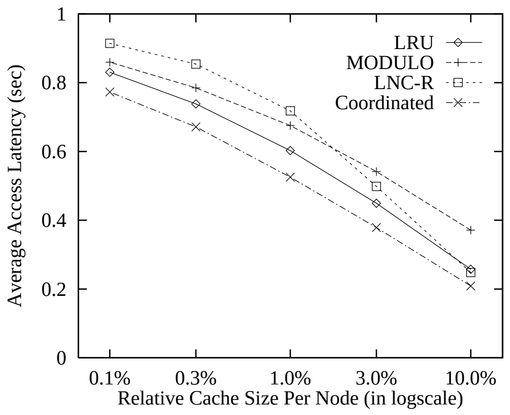


(a) Average Number of Hops Traveled

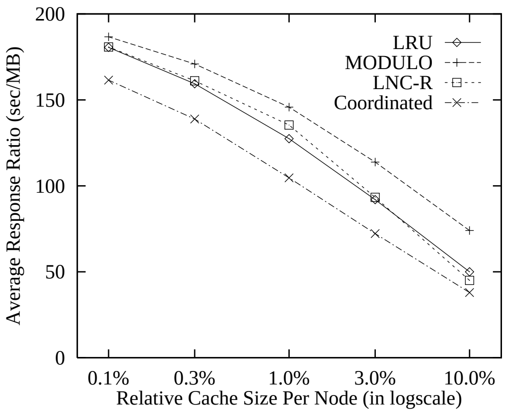


(b) Overall Cache Read/Write Load

Figure 8. Number of Hops Traveled and Cache Load vs. Cache Size (En-Route Architecture)



(a) Average Access Latency



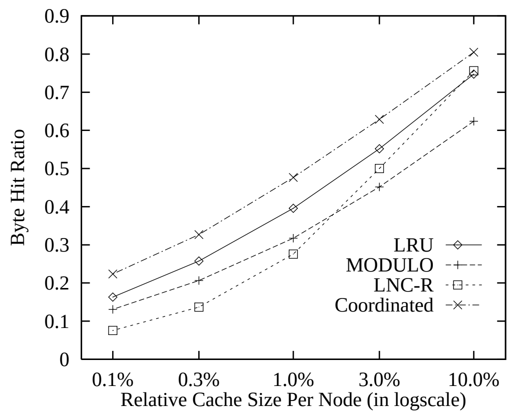
(b) Average Response Ratio

Figure 9. Access Latency and Response Ratio vs. Cache Size (Hierarchical Architecture)

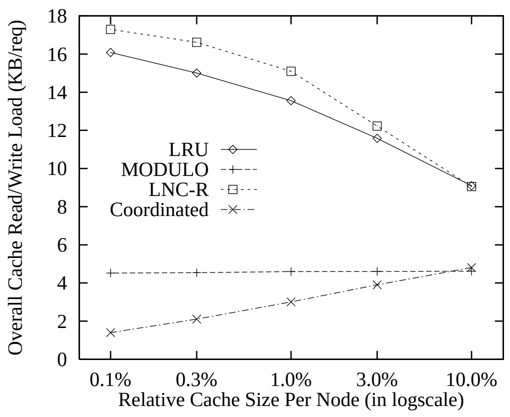
hops a request travels before hitting the target object. Figure 8(a) shows that on average, a request goes through fewer hops in the coordinated caching scheme than in the other schemes. This further demonstrates the advantage of coordinated caching in placing popular objects closer to the clients. An additional benefit is the lower look-up and forwarding load on the caches. Reading and writing objects occur in the following situations. A request causes a read operation on an en-route cache if the request leads to a cache hit, and a decision to insert an object into the cache introduces a write operation. The read load is necessary in serving requests while the write load represents an overhead for caching. To compare the overall read/write load, we calculated the mean aggregate read and write load (measured in bytes) introduced by each request on all the caches. As shown in Figure 8(b), coordinated caching has the lowest read/write load among all the schemes studied. By contrast, LRU and LNC-R introduce 3 to 24 times the read/write load of coordinated caching because they do not consider object placement optimization. MODULO has lower read/write load compared to LRU and LNC-R, but its load is still much higher than that of the coordinated scheme. Experimental data show the read load takes up 75% to 80% of the overall read/write load in coordinated caching. Since the read load of coordinated caching is higher than the other schemes due to its higher cache hit ratio (see Figure 7(a)), the results presented in Figure 8(b) suggest that the coordinated scheme involves substantially lower write load (i.e., overhead) on the caches.

4.2 Hierarchical Caching Architecture

Figures 9(a) and (b) show the average access latency and response ratio for different caching schemes under the hierarchical architecture. Similar to the case of en-route architecture, the coordinated caching scheme consistently provides the best performance over a wide range of cache sizes. For example, at a cache size of 3%, it outperforms LRU, MODULO and LNC-R by 22%, 37% and 23% in response ratio respectively. This further verifies the importance of coordinating object placement and replacement strategies. Similarly, for the same reason discussed in Section 4.1, LNC-R generally performs worse than LRU. On



(a) Byte Hit Ratio



(b) Overall Cache Read/Write Load

Figure 10. Hit Ratio and Cache Load vs. Cache Size (Hierarchical Architecture)

the other hand, different from the case of en-route architecture (Section 4.1), MODULO with a cache radius of 4 performs much worse than LRU as shown in Figures 9(a) and (b). This is because the distribution trees for different origin servers are almost identical under the hierarchical architecture, differing only in the link between the origin server and the root cache. As a result, by placing the requested objects in caches every 4 hops apart, the MODULO scheme leaves the caches at levels 1, 2 and 3 completely unused (see Figure 5), leading to significant performance degradation. In fact, MODULO with any cache radius greater than 1 would result in some caches unused. Under our hierarchical architecture, the best possible performance of MODULO is achieved with a cache radius of 1 in which case MODULO degenerates to LRU. In contrast, under the en-route architecture, the distribution trees for various origin servers are very different. Therefore, as shown in Section 4.1, MODULO makes more efficient use of cache space and outperforms LRU in the en-route environment.

Figures 10(a) shows the results of byte hit ratio as a function of relative cache size. It can be seen that the coordinated caching scheme achieves the highest byte hit ratio among all the schemes studied. MODULO does not make full use of the available caches and shows much lower byte hit ratio than that of LRU. On the other hand, similar performance trends in the aggregate read and write cache load have been observed for both en-route and hierarchical architectures. Even though the coordinated scheme has higher read load than the other schemes, Figure 10(b) shows that it generally results in the lowest total read/write load. This implies substantial reduction in the write load (i.e., the overhead) by the coordinated scheme. It is interesting to note that the overall read/write load of MODULO is the same for different cache sizes. This is because under the MODULO scheme with a cache radius of 4, only one cache (i.e., the leaf cache) is used between each origin server and its clients. A request introduces an amount of read load equal to the object size in case of a cache hit and involves the same amount of write load for a cache miss. Thus, the aggregate read and write load introduced by each request is independent of the cache size.

5 Related Work

Not much work has been done on coordinated management of cascaded caches. Bhattacharjee *et al.* [3] studied the benefits of the en-route caching architecture. They proposed an object placement strategy called MODULO, where requested objects are cached at the nodes that are a fixed number of hops apart along the delivery path from the server to the client. MODULO was shown to outperform the simple ”caching everywhere” strategy which places the object at every intermediate node. However, the object access frequency and the network distance between caches were not considered. Furthermore, all objects were assumed to have the same size and the simple LRU policy was used for cache replacement. The interaction between placement and replacement strategies was not investigated. Yu *et al.* [21] discussed collaboration between parent and children caches in the hierarchical caching architecture. However, no analytical modeling was provided. To the best of our knowledge, there has been no unified analytical framework on coordinated cache management for general cascaded architectures.

Early work on single web cache management had investigated simple extensions of traditional page replacement algorithms such as LRU and LFU [19]. More recent work has focused on cost-based cache replacement algorithms [16, 20]. The idea is to design a cost function to estimate the caching benefits of different objects and determine their removal order when there is not enough free space. A typical example is the LNC-R algorithm proposed by Scheuermann *et al.* [16]. LNC-R incorporates multiple factors in the cost function including the object size, the access frequency and the retrieval delay. The authors further proved that the suggested function is optimal (i.e., maximizing delay reduction for a single cache) in a simplified model [17]. However, like most other replacement policies for web caches, the LNC-R scheme automatically places a newly referenced object in the cache without evaluating whether it is beneficial to do so. As shown in our experimental study, optimizing replacement decisions alone (i.e., without appropriate placement optimization) does not lead to more effective use of cache space and improved access latency when there are a large number of interconnected caches. Object placement and replacement should be optimized in an integrated fashion to produce the best possible performance. Aggarwal *et al.* [2] proposed a generalized LRU scheme and used an admission control policy to decide whether or not caching an object is worthwhile. However, they did not consider cooperation among different caches.

6 Conclusion

Object placement and replacement are two important issues in cascaded cache management. In this paper, we have presented a general analytical framework for coordinated management of cache contents under cascaded caching architectures. The optimal placement decisions of objects are computed by a dynamic programming algorithm. Based on the framework, a novel coordinated caching scheme that integrates object placement and replacement strategies has been proposed. We have performed simulation experiments using real traces to compare the proposed scheme with a number of existing algorithms. The results show that the coordinated scheme effectively reduces access latency and improves cache hit ratio under different cascaded caching architectures. The proposed scheme considerably outperforms existing algorithms which consider either object placement or replacement at individual caches only.

References

[1] Proxy Cache Comparison. http://www.web-caching.com/proxy-comparison.html.

[2] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, January/February 1999.

[3] S. Bhattacharjee, K. L. Calvert, and E. W. Zegura. Self-organizing wide-area network caches. In *Proceedings of IEEE INFOCOM'98*, pages 600–608, Mar. 1998.

[4] 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*, pages 126–134, Mar. 1999.

[5] K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997.

[6] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 153–163, Jan. 1996.

[7] B. D. Davison. A web caching primer. *IEEE Internet Computing*, 5(4):38–45, July/August 2001.

[8] S. Jin and A. Bestavros. Popularity-aware greedydual-size web proxy caching algorithms. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 254–261, Apr. 2000.

[9] B. Krishnamurthy and C. E. Wills. Piggyback server invalidation for proxy cache coherency. *Computer Networks and ISDN Systems*, 30(1–7):185–193, Apr. 1998.

[10] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, Oct. 2000.

[11] B. Li, M. J. Golin, G. F. Italiano, X. Deng, and K. Sohraby. On the optimal placement of web proxies in the internet. In *Proceedings of IEEE INFOCOM'99*, pages 1282–1290, Mar. 1999.

[12] J. Meadows. Boeing proxy logs. ftp://researchsmp2.cc.vt.edu/pub/boeing, 1999.

[13] V. N. Padmanabhan and L. Qiu. The content and access dynamics of a busy web site: Findings and implications. In *Proceedings of ACM SIGCOMM'00*, pages 111–123, Aug. 2000.

[14] M. Rabinovich and H. Wang. Dhttp: An effi cient and cache-friendly transfer protocol for web traffi c. In *Proceedings of IEEE INFOCOM'01*, pages 1597–1606, Apr. 2001.

[15] P. Rodriguez and S. Sibal. Spread: scalable platform for reliable and effi cient automated distribution. *Computer Networks*, 33(1–6):33–49, June 2000.

[16] P. Scheuermann, J. Shim, and R. Vingralek. A case for delay-conscious caching of web documents. *Computer Networks and ISDN Systems*, 29(8–13):997–1005, Sept. 1997.

[17] J. Shim, P. Scheuermann, and R. Vingralek. Proxy cache algorithms: Design, implementation, and performance. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):549–562, July/August 1999.

[18] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, Jan. 1997.

[19] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for world wide web documents. In *Proceedings of ACM SIGCOMM'96*, pages 293–305, Aug. 1996.

[20] R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8–13):977–986, Sept. 1997.

[21] P. S. Yu and E. A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. *Computer Networks and ISDN Systems*, 30(1–7):215–224, Apr. 1998.

[22] M. Zari, H. Saiedian, and M. Naeem. Understanding and reducing web delays. *IEEE Computer*, 34(12):30–37, Dec. 2001.