Coordinated Placement and Replacement for Large-Scale Distributed Caches *

Madhukar R. Korupolu [1] Michael Dahlin [1]

Abstract

In a large-scale information system such as a digital library or the web, a set of distributed caches can improve their effectiveness by coordinating their data placement decisions. In this paper, we examine the design space for cooperative *placement* and *replacement* algorithms. Our main focus is on the placement algorithms, which attempt to solve the following problem: given a set of caches, the network distances between caches, and predictions of the access rates from each cache to a set of objects, determine where to place each object in order to minimize the average access cost. Replacement algorithms also attempt to minimize access cost, but they work by selecting which objects to evict when a cache miss occurs.

Using simulation, we examine three practical cooperative placement algorithms including one that is provably close to optimal, and we compare these algorithms to the optimal placement algorithm and several cooperative and non-cooperative replacement algorithms. We draw five primary conclusions from these experiments: (1) cooperative placement can significantly improve performance compared to local replacement algorithms particularly when the space of individual caches is limited compared to the universe of objects; (2) although the *Amortized Placement* algorithm is only guaranteed to be within 14 times the optimal, in practice it seems to provide an excellent approximation of the optimal; (3) in a cooperative caching scenario, the recent *GreedyDual* local replacement algorithm performs much better than the other local replacement algorithms because it implicitly coordinates the replacement decisions of the caches; (4) our *Hierarchical GreedyDual* replacement algorithm yields further performance improvements over the GreedyDual algorithm especially when there are idle caches in the system; and (5) a key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses.

1 Introduction

Consider a large-scale distributed information system, such as a digital library or the world wide web, that provides access to shared objects. Caching popular objects close to the clients is a fundamental technique for improving the performance and scalability of such a system. Caching enables requests to be satisfied by a nearby copy and hence reduces not only the access latency but also the burden on the network and server.

[1]Department of Computer Science, University of Texas at Austin, Austin, TX 78712. Email: {madhukar,dahlin}@cs.utexas.edu.

*This work was supported in part by an NSF CISE grant (CDA-9624082) and grants from Intel, Novell, and Sun. Dahlin was also supported by an NSF CAREER grant (9733842).

A powerful paradigm to improve cache effectiveness is *cooperation*, where caches cooperate both in serving each other's requests and in making storage decisions. In the simplest caching scheme caches never consult one another, and when a cache miss occurs, a cache contacts the server directly. In the widely deployed and studied Harvest and Squid hierarchical caching systems [4, 22], the caches are arranged in a hierarchy, and each cache cooperates with a few sibling and parent caches to service requests. In a more general scenario, each cache would cooperate with all other caches in a cooperative caching system. More general cooperation such as this is particularly attractive in environments where machines trust one another such as within an Internet service provider, cache service provider, or corporate intranet. In addition, cooperation across such entities could be based on peering arrangements such as are now common for Internet routing.

There are two orthogonal issues to cooperative caching: finding nearby copies of objects (*object location*) and coordinating the caches while making storage decisions (*object placement*). The object location problem has been widely studied [2, 4, 21]. Many recent studies on the object location problem (e.g., Summary Cache [7], Cache Digest [19], Hint Cache [20], CRISP [9] and Adaptive Web Caching [27]) generalize from hierarchies to more powerful cache-to-cache cooperation scenarios. However, these algorithms do not address the object placement issue.

Efficient cache coordination algorithms would greatly improve the effectiveness of a given amount of cache space and are hence crucial to the performance of a cooperative caching system. We believe that the importance of such algorithms will increase in the future as the number of shared objects continues to grow enormously and as the Internet becomes the home of more large multimedia objects. Although the falling cost of disk storage allows caches to grow rapidly, it also drives down the cost of server storage. Hence in the long run the universe of servers supplying objects will have much more data than any individual cache can store.

In this paper we focus on the cache coordination issue and provide placement and replacement algorithms that allow caches to coordinate storage decisions. The placement algorithms attempt to solve the following problem: given a set of cooperating caches, the network distances between caches, and predictions of the access rates from each cache to a set of objects, determine where to place each object in order to minimize the average access cost. On the other hand, replacement algorithms determine which objects are to be evicted when a cache miss occurs.

We examine an optimal placement algorithm and three practical placement algorithms and compare them to several local, uncoordinated replacement algorithms and a new hierarchical extension to the GreedyDual algorithm [3, 25]. We drive this comparison with simulation studies based on both synthetic and trace workloads. The synthetic workloads allow us to examine system behavior in a wide range of situations, and the trace allows us to examine performance under a workload of widespread interest: web browsing.

Based on these experiments, we reach five primary conclusions.

- Cooperative placement can significantly improve performance compared to local replacement algorithms particularly when the space of individual caches is limited compared to the universe of objects.

- It was established in an earlier theoretical work by Korupolu, Plaxton and Rajaraman [14] that, under the hierarchical model for distances, the *Amortized Placement* algorithm is always within a constant factor (about 13.93) of the optimal. But for practical purposes, this factor is still large. Based on our new experiments here, we infer that the Amortized Placement algorithm yields an excellent approximation of the optimal for a wide range of workloads. This is an important result for two reasons. First, in systems that can generate good estimates of access frequencies, Amortized Placement is a practical algorithm that can be expected to provide near-optimal performance. Second, for large-scale studies of cache coordination, Amortized Placement can provide a practical "best case" baseline against which to test other algorithms. In addition, we find that a simplified version of the algorithm called *Greedy Placement* also provides an excellent approximation of optimal even though in theory its performance can be arbitrarily worse than optimal.

- In a cooperative caching scenario, the *GreedyDual* local replacement algorithm performs much better than the other local replacement algorithms because its inclusion of access cost in its replacement decisions provides an implicit channel for coordinating cache replacement decisions.

- Our *Hierarchical GreedyDual* replacement algorithm yields improved performance over the GreedyDual local replacement algorithm especially when there are idle caches in the system.

- A key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses.

The rest of this paper is organized as follows: First, we provide background on the cooperative caching and coordinated placement. Then, in Section 3 we describe the algorithms we study. Sections 4 and 5 detail our experimental results under synthetic and trace workloads, respectively. Section 6 surveys related work, and Section 7 summarizes our conclusions.

2 Background

The advantages of coordinated caching are two-fold. First, by avoiding unnecessary duplication, more objects can be stored closer to the clients. The potential gains of this effect become more pronounced as the degree of similarity of interest among nearby clients increases. Second, at any instant of time there may be several caches that are either idle or almost idle, and coordination would allow a busy cache to utilize a nearby idle cache [6, 8].

As a simple example, consider a pair of nearby clients u and v that are accessing a pair of shared objects A and B. Suppose that (i) each of u and v has a cache that is capable of holding one object; (ii) a distant node w has copies of both objects A and B; (iii) $distance(u, v) = 1$ and $distance(u, w) = distance(v, w) = 10$; and (iv) the access frequency for object A is 2 units and for object B is 1 unit at both the clients.

Under a non-coordinated caching strategy, both the clients would typically end up keeping object A in their caches, and the requests for B would have to go to w. Hence the overall access cost would be $2 \cdot 0 + 1 \cdot 10 + 2 \cdot 0 + 1 \cdot 10 = 20$ units. On the other hand, with coordination, if one of them can store object A while the other stores object B, the overall cost would be just $2 \cdot 0 + 1 \cdot 1 + 2 \cdot 1 + 1 \cdot 0 = 3$ units. The above example illustrates that by avoiding unnecessary duplication, more objects can be stored closer to the clients. A noteworthy point is that the coordination improved not only the global access cost, but also the individual access costs.

We remark that avoiding duplication altogether is not a good solution either, since duplication is often essential for improving performance. For example, if the closest cached copy of an object is sufficiently far, then keeping a duplicate nearby may be useful. Or even if there is a reasonably nearby copy, if an object is frequently referenced, having an additional even closer duplicate could pay off.

Thus, in trying to decide what to store in the various caches, a good coordination strategy should balance the improved hit rates from reducing duplication against the improved hit times for increasing duplication of popular objects. Before attempting to design such coordination strategies, we first formulate a more precise problem statement. Subsection 2.1 defines the basic placement and replacement problems for coordinated caching, assuming a generic distance function between the nodes. Subsection 2.2 then describes our distance model which is based on the network-locality hierarchy. Note that though the problem formulation is a simplification of true Internet, it seems to capture the salient features of the placement problem.

2.1 The problem formulation

Consider a set of N distributed machines connected by a network. Let *dist* be a function that gives the the cost of communication, between any pair of machines, and suppose that these machines are accessing a set of M shared objects. (For simplicity, we assume that all objects have the same size and are read-only.) For each machine i, let $cachesize(i)$ denote the number of objects that can be stored in the cache at machine i.

We assume that all requests are satisfied by the closest copy of the requested object. In order to account for the cost of accessing objects for which no copy exists in the collection of caches, we assume that we are given a *miss penalty* Δ that is at least the maximum value of the function *dist*. We now define the cost of accessing an object α from a machine i. If there is at least one copy of α in the network , then the access cost $c(i, \alpha)$ equals $dist(i, i')$, where i' is the closest machine that has a copy of α; otherwise $c(i, \alpha)$ equals Δ.

A *placement* assigns copies of objects to machines subject to the cache size constraints. For any machine i and object α, let $f(i, \alpha)$ denote the access frequency for object α at machine i. Given these frequencies, the aim of any coordinated placement strategy is to fill the available cache space such that overall access cost is minimized. The average access cost, or simply the cost, of a placement P is given by the sum over all machines i and objects α of $f(i, \alpha) \cdot c(i, \alpha)$. Thus, our placement problem is to find a placement with the minimum cost given the distance function, the cache sizes, and the frequency function.

Compared with placement algorithms, replacement algorithms also attempt to minimize the access cost, but they proceed by selecting which objects to evict when a cache miss occurs rather than explicitly computing a placement based on access frequencies.

We remark that even though our problem formulation does not explicitly minimize the network load and the server load, these would typically reduce when the access cost is minimized. This is because the latter objective would encourage objects to be stored closer to the clients, which leads to reduced load on the network as well as the server.

2.2 Hierarchical distance model

The distance (or communication cost) function between machines in modern wide-area networks is not entirely arbitrary but is strongly guided by the fact that these networks have a natural hierarchical structure. Moreover, for any pair of machines, the distance is essentially captured by the "biggest" step or link between the two machines. As a simple example, the distance between a machine A in the University of Texas and a machine B in Berkeley University is essentially the same as that between A and the third machine C in Berkeley. This is essentially the cost of going over the link between Texas and Berkeley.

Such a distance structure can be modeled by a "network-locality hierarchy" tree T^* and a *diameter* function *diam* that satisfy the following properties: (i) the leaves of the tree are the machines of the network, (ii) if u is the parent of v in T^*, then $diam(u)$ is at least $diam(v)$, and (iii) for any two machines i and i', $dist(i,i')$ equals $diam(u)$, where u is the least common ancestor of the leaves i and i' in T^*.

The network-locality hierarchy models a system where each machine has a set of nearby neighbors, all at about the same distance d_1, and then a set of next-closest neighbors, all at the same distance d_2 and so on. We emphasize that our network locality hierarchy is distinct from the caching hierarchy used in Harvest [4] and other hierarchical caching systems. In the latter systems, the hierarchy corresponds to the actual topology of the network and each internal node is a physical node with its own cache space. On the other hand, our network locality hierarchy is a *logical* tree whose main purpose is to capture the network distances between the machines, which exist only at the leaves of the tree.

The above distance model captures a large class of distributed networks. For example, if the tree T^* consists of exactly one internal node, then the associated cost function models a local-area network of workstations. In fact, this is precisely the model used in [1, 15] in the study of caching schemes for networks of workstations. On the other hand, a tree with several levels captures larger-scale networks such as intranets and the Internet. Similar hierarchical structures for wide-area networks are implicitly used in several previous studies [4, 11, 18, 20, 21, 22, 26].

For the remainder of the paper, we assume that the network distances follow this hierarchical model.

3 Algorithms

In this section, we present three practical cooperative algorithms. One of them is a replacement algorithm, called *Hierarchical GreedyDual*, while the other two are placement algorithms, called *GreedyPlace* and *AmortPlace*. We compare these with several non-cooperative and cooperative algorithms. On one side, we compare with the non-cooperative local algorithms such as LRU, GreedyDual, and MFU to estimate the benefits of cooperation. On the other side, we compare with an optimal cooperative placement algorithm that gives us a limit on the best we can hope to achieve. For simplicity, we assume that all objects have the same size.

3.1 Purely local algorithms

To serve as a baseline for comparing performance, we examine four algorithms that make all their placement or replacement decisions locally without consulting any other cache.

MFUPlace. In this placement algorithm, if the size of the cache is k, then the cache stores the k most frequently used objects. This strategy works best when the accesses are drawn from a fixed probability distribution and are uncorrelated.

LRU Replacement. This is a well-known replacement algorithm which has been demonstrated to yield good performance in main memory caching of file systems. When a cache miss occurs, this algorithm picks the least recently used object from the cache for eviction.

LFU Replacement. This replacement algorithm maintains the (local) frequency of access to each object. When a cache miss occurs, the object with the lowest (local) frequency of access is chosen to be evicted from the cache. Unlike MFUPlace, this dynamic replacement decision could result in a less frequently used object displacing a more frequently used object. This strategy too works best when the accesses are drawn from a fixed probability distribution and are uncorrelated.

GreedyDual Replacement. This is a generalization of the LRU algorithm to the case where each object has a different but fixed miss cost [3, 25]. The motivation behind the GreedyDual algorithm is that the objects with larger cost should stay in the cache for a longer time.

The algorithm maintains a *value* for each object that is currently in the cache. When an object is fetched into the cache, its value is set to its fetch cost. When a cache miss occurs, the object with the minimum value is evicted from the cache, and the values of all the other objects in the cache are reduced by this minimum value. And if an object in the cache is accessed (or 'touched'), then its value is restored to its fetch cost.

From an implementation point of view, it would be expensive to modify the value of each cache object, upon each cache miss. However, this expense can be avoided by noting that it is only the relative values, and not the absolute values, that matter [3]. In an efficient implementation, upon a cache miss, the minimum valued object is evicted from the cache and no other values are modified. However, when an object is touched or added, its value is set to its fetch cost plus the value of the minimum-valued object in the cache.

Our experiments show that, in a cooperative scenario, the GreedyDual algorithm performs much better than the other local replacement algorithms. This is because even though the GreedyDual algorithm makes entirely local decisions, its cost-value structure enables some implicit coordination with other caches. In particular, an object that was fetched from a nearby cache would have a smaller value than an object that was fetched from far. Hence the latter object would typically stay in the cache for a longer time, thereby reducing unnecessary replication among nearby caches.

However, this limited degree of coordination does not exploit all the benefits of cooperation. For example, the idle caches are not exploited by the nearby busy caches.

3.2 Cooperative placement algorithms

Below, we first describe an optimal placement algorithm. The high running time and bandwidth requirements of this algorithm make it impractical for use with large input instances, and hence our sole use for this algorithm is as a benchmark for evaluating other placement algorithms. This algorithm and its proof of optimality appear in an earlier paper [14]; we include a brief discussion here for completeness.

The impracticality of the optimal algorithm motivates the search for a fast near-optimal algorithm that would admit efficient distributed implementations. In subsection 3.2.2, we present two candidate algorithms that compute a placement by a simple bottom-up pass through the network locality hierarchy. The original presentation of these algorithms in [14] involves two passes through the network locality hierarchy: a bottom-up pass that computes a *pseudo-placement*, and a top-down pass that refines this pseudo-placement to a placement. The additional notion of pseudo-placement was essential for proving the performance guarantees of these algorithms, but not for correctness. Here, we give an equivalent one-pass description of these algorithms avoiding the notion of pseudo-placement and highlighting the ease of implementation.

Recall that the tree T^* is the network-locality hierarchy with caches/clients at the leaves, and that the frequency function f was defined for the clients (leaves) only. We extend this definition to hold even for the internal nodes by defining $f(u, \alpha)$, for any internal node u and for any object α, to be the *aggregate* frequency from T^*_u to α (i.e., the sum over all leaves i in

T^*_u, of $f(i, \alpha)$). We also define the miss penalty $miss(u)$ to be Δ if u is the root of T^*, and $diam(parent(u))$ otherwise.

3.2.1 An optimal placement algorithm

An optimal algorithm for the placement problem was developed in [14], by a reduction to the minimum cost flow problem. This reduction generalizes the approach of Leff, Wolf, and Yu [15] who solved the problem for the special case of a single-level hierarchy.

The instance of the minimum-cost flow problem constructed by this reduction has $\Theta(NM)$ nodes, where N is the number of machines in the system and M is the number of objects in the system. Hence, the time complexity of this optimal algorithm will be at least quadratic in $\Theta(NM)$, even if we use the fastest known algorithms for computing the minimum cost flow. Moreover, because the algorithm is centralized, it requires prohibitive amounts of communication for transferring the access pattern information from all the clients to the central processor. Hence, although this algorithm may be applicable for small LANs, it is impractical for systems with many caches and millions of objects.

3.2.2 Simple near-optimal placement algorithms

These algorithms follow a natural greedy improvement paradigm that is common to several optimization problems. They start with a placement in which each machine places in its cache the locally most valuable set of objects. The algorithm then proceeds by iteratively improving the placement in a bottom-up manner, along the network-locality hierarchy, as the machines cooperate and share information about access frequencies across larger regions of the network.

Notation and terminology. A placement P is represented as a set of items, where each item is a triple of the form *(objectId, cacheId, benefit)*. The benefit of an item roughly corresponds to the amount by which the cost of P would increase if this item were dropped from P. Note that a placement P is legal if and only if for every cache i, the number of items in P that are located at i is at most $cachesize(i)$. If there are no copies of α in P, then we say that the object α is P-missing.

3.2.3 The greedy placement algorithm

We now present the bottom-up greedy algorithm, by giving separate descriptions of the computations performed at the leaf and internal nodes. For an efficient and scalable implementation where the internal nodes' computations are mapped to the leaves, the reader is referred to [14].

GreedyPlace: Leaf Node u

- Construct an optimal local placement Q for a leaf node u as follows. For each of the $cachesize(u)$ objects α most frequently accessed by u, set the benefit b of α to $f(u, \alpha) \cdot (miss(u) - diam(u))$, and add the item (u, α, b) to Q.

GreedyPlace: Internal Node u

- **Merge.** Let Q_i be the placement previously computed for u_i, the ith child of u. Initialize the placement Q to the union of the Q_i's.

- **Adjust benefits.** For each object α that has a copy in Q, pick its highest-benefit copy (breaking ties arbitrarily), and designate it as the α-*primary* copy. All other copies of

α are referred to as *secondary* copies. Increase the benefit of the α-primary copy by $f(u,\alpha) \cdot (miss(u) - diam(u))$. The benefits of the secondary copies are not changed.

- **Value missing objects.** Let X be the set of all Q-missing objects. For each object α in X, set its value to $f(u,\alpha) \cdot (miss(u) - diam(u))$.

- **Swapping.** While the value of the highest valued object α in X is larger than the benefit of the smallest benefit item y in Q, perform the following swap operation: Remove the item y from Q and add the item $(i, \alpha, value(\alpha))$ to Q. Remove α from X.

For efficient implementation, the set Q should be sorted in decreasing order of benefits and the set X of Q-missing objects should be sorted in decreasing order of values. Then for the swap phase we have two pointers, one starting at the tail of Q and the other starting at the head of the list of Q-missing objects. If the item and the object being pointed to satisfy the swap condition, we perform the swap and advance the pointers. Otherwise, the swap phase terminates.

The intuition for the above procedure for adjusting benefits is basically to ensure that the benefit of each item roughly corresponds to the amount by which the cost of Q would increase when this item is dropped from Q.

An interesting feature of the algorithm is implicit in the swapping step. Note that if k swaps are performed, then any assignment of the k incoming objects to the k vacated cache slots would yield a legal placement Q, possibly with differing costs. However, for simplicity we do not optimize the assignment of incoming objects to the set of available cache slots. Instead we pick an essentially arbitrary assignment. The intuition behind this simplification is that, because the incoming objects were not chosen earlier by any subtree of T_u^*, no single subtree by itself can gain significantly by keeping that object. An area for future work is to use this degree of freedom to further improve the quality of the resulting placement.

It is shown in [14] that the above GreedyPlace algorithm can be arbitrarily far from the optimal in the worst-case. The worst-case example used for this lower bound leads to a natural refinement of the greedy algorithm, called the *amortized* placement algorithm.

3.2.4 The amortized placement algorithm

The amortized algorithm is similar to the greedy algorithm, except that additionally we use a potential function ϕ to accelerate the removal of certain secondary copies in favor of taking the missing objects.

AmortPlace: Leaf Node u

- Same as in the greedy algorithm, except that we also set the potential ϕ to zero.

AmortPlace: Internal Node u

- **Merge.** Same as in the greedy algorithm, except that we also initialize the potential ϕ to the sum of the potentials $\phi_1, \ldots \phi_k$, computed by the children of u.

- **Adjust benefits.** Same as in the greedy algorithm.

- **Value missing objects.** Same as in the greedy algorithm.

- **Amortized swapping.** This procedure is similar to the swapping procedure in the greedy algorithm, except that the potential ϕ is used to reduce the benefits of certain items.

1. Let y_p be the smallest-benefit primary item in Q and let y_s be the smallest-benefit secondary item in Q. Let α be the highest valued (Q-missing) object in X.
2. If $value(\alpha) > min(benefit(y_p), benefit(y_s) - \phi)$, then perform one of the following two swap operations, depending on which of the two terms is smaller, and goto step (1).
   - If $benefit(y_p) < benefit(y_s) - \phi$: Suppose $y_p = (i, \alpha', B)$, remove y_p from Q, and add the item $(i, \alpha, value(\alpha))$ to Q. Also set X to $X - \alpha + \alpha'$ and $value(\alpha')$ to B.
   - Otherwise, suppose $y_s = (i, \alpha', B)$, remove y_s from Q, and add the item $(i, \alpha, value(\alpha))$ to Q. Also set X to $X - \alpha$, and reset the potential ϕ to 0 if $\phi < benefit(y_s)$ and to $\phi - benefit(y_s)$ otherwise.

- **Update potential.** Add the values of all the (Q-missing) objects in X to ϕ.

It was proved in [14] that the above AmortPlace algorithm is always within a constant factor of the optimal, for all hierarchal distance functions, for all cache sizes, and for all access patterns. The constant factor is less than 13.93. However, due to the simplicity of the algorithm, we believe that the performance would be much better in practice.

3.3 A cooperative replacement algorithm

The *Hierarchical GreedyDual* is a cooperative replacement algorithm that not only preserves the implicit coordination offered by GreedyDual but also enables busy caches to utilize the nearby idle caches. Our algorithm is a generalization of the GreedyDual algorithm and can be implemented efficiently even in a distributed setting.

Each (leaf) cache runs the local GreedyDual algorithm, using the efficient implementation described in subsection 3.1. Recall that this algorithm maintains a *value* for each object in the cache, and upon a cache miss, it evicts the object with the minimum value. In our hierarchical generalization, the evicted object is then "passed up" the network-locality hierarchy for possible inclusion in a nearby cache. When an internal node u in the network-locality hierarchy receives an evicted object α from one of its children, it first checks to see if a copy of α already exists in its subtree T_u^*. If not, it picks the minimum valued object β among all the objects cached in its subtree for possible eviction. A simple admission control test is then used to determine if α should replace β. If the copy of α was used more recently than the copy of β, then α replaces β and the new evicted object β is recursively passed on to the parent of u. Otherwise, the object α is recursively passed on to the parent of u.

We remark that the particular admission control test mentioned above is crucial for obtaining good performance. An important purpose of the admission control test is to prevent rarely-accessed objects from jumping from cache to cache without ever leaving the system. Such objects would typically have a high fetch cost since no other (nearby) cache would have stored them, and hence any fetch-cost based admission control test would hold on to such objects even after they are evicted by individual caches. This would result in worse performance than even the local GreedyDual algorithm. We avoid this problem by maintaining a *last-use* timestamp on every object in the cache. This timestamp is updated whenever the copy is accessed, either by the local client or by a remote client. With our admission control strategy, rarely used objects are eventually released from the system.

Note that in practice rather than "passing up" evictions, this algorithm would use data-location directories [7, 9, 19, 20] to determine if other copies exist in the subtree, and would use randomized [6] or deterministic [8] strategies to approximate the selection of β.

Parameter	Meaning	Default Value
L	Number of levels	3
D	Degree of each internal node	3
λ	Diameter growth factor	4
C	Cache size percentage	20% (synthetic workload) 1% (trace workload)
m	Number of objects local to each node	25 (synthetic only)
r	Sharing parameter	0.75 (synthetic only)
PAT	Access pattern	Uniform (synthetic only)
I	Idle cache factor	0

Table 1: Default system parameters.

4 Performance evaluation on synthetic workloads

This section explores the performance of the algorithms under a range of synthetic workloads. These workloads allow us to explore a broader range of system behavior than trace workloads. In addition, because the synthetic workloads are small enough be tractable under the Optimal algorithm, we can compare our algorithms to Optimal placement.

This section first describes our methodology in detail and then shows the results of our experiments. These results support four primary conclusions: (1) cooperative placement can significantly improve performance compared to local replacement particularly when the space of individual caches is limited compared to the universe of objects; (2) although the *Amortized Placement* algorithm is only guaranteed to be within 14 times the optimal, in practice it seems to provide an excellent approximation of optimal; similarly, although the Greedy Placement algorithm can, in principle, be arbitrarily worse than the Optimal, it also seems to provide an excellent approximation in practice; (3) in a cooperative caching scenario, the *GreedyDual* local replacement algorithm performs much better than the other local replacement algorithms studied; finally, (4) our *Hierarchical GreedyDual* local replacement algorithm yields improved performance over the GreedyDual replacement algorithm especially when there are idle caches in the system.

4.1 Methodology

We simulate a collection of caches that include a directory system such as that provided by Hint Cache [20], Summary Cache [7], Cache Digests [19] or CRISP [9] so that caches can send each local miss directly to the nearest cache with the data or directly to the server if no cache has the data. For the placement algorithms *MFUPlace, GreedyPlace, AmortPlace, or Optimal*, we compute the initial data placement according to the algorithm under simulation, and the data remain in their initial caches throughout the run. For the replacement algorithms *LFU, LRU, GreedyDual, or HrcGreedyDual*, we begin with empty caches, and for each request we modify the cache contents as dictated by the replacement algorithm. In that case, we use an initial warm-up stage to prime the caches before gathering statistics. Our simulator is event-based, but it does not model concurrency: it processes each client request completely before beginning the next one.

We parameterize the network architecture and workload along a number of axes. The parameters are defined in detail in the following two subsections. Table 1 summarizes the default values for these parameters.

4.1.1 Network architecture

Recall from subsection 2.2 that we model the distances between cache nodes using the hierarchical network model. In particular, each node has a collection of level-1 neighbors all at distance d_1 away from it, a collection of level-2 neighbors all at distance d_2 away and so on. We create an L-level tree in which leaves represent cache nodes and subtrees rooted at internal nodes represent collections of nodes that are near one another compared to nodes not in the subtree. For simplicity, we assume that all internal nodes have the same degree, D. The diameters of the subtrees are captured by a single parameter λ, called the diameter growth factor. The diameter for a subtree at level i is λ^i. The level of the root is L, the level of a leaf is zero, and the miss-cost for the hierarchy is λ^{L+1}.

For simplicity, we assume that all objects have the same size, and we express the size of a cache in terms of the number of objects that it can hold. We also set all cache sizes to be the same. The cache size percentage, C, is the percentage of the relevant objects that can simultaneously fit into a cache. More specifically, we set the cache size to $CM/100$ where M^* is defined differently for trace workloads and synthetic workloads. For a trace workload, M^* is the average number of objects that appear per day of the trace. For synthetic workloads, M^* is the maximum number of objects accessed by any node.

4.1.2 Workload

As observed in section 2, an important parameter for the performance of cooperative strategies is hierarchical similarity of interests. At one extreme, there is total similarity (all nodes access the same set of shared objects with the same frequencies) while at the other extreme there is absolutely no similarity (each node accesses its own set of local objects).

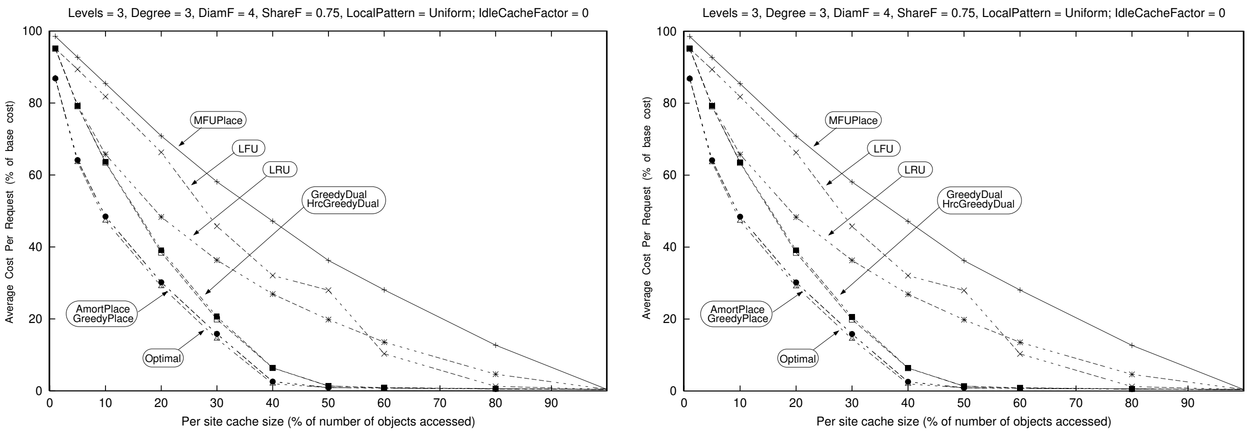
Our synthetic workload models such sharing by creating m objects for each subtree in the network-locality hierarchy. This pattern could represent a hierarchical organization such as a corporation or university where some objects are local to an individual, some to a group, some to a department, and some of organization-wide interest. The sharing parameter, r, determines the mix of requests to the "private", "group," "department," and "organization" collections of data. The fraction of requests that a client sends to level-i data is proportional to r^i. Note that as r varies from 0 to infinity, the degree of sharing increases: at $r = 0$, each client accesses its local objects only, and hence there is no sharing at all. As r increases, accesses to more widely shared objects start to dominate.

Within each subtree's collection of data, we select objects according to a pattern PAT that is either "Zipf-like" or "Uniform." Thus, for a particular leaf cache v and object j that is local to subtree u and that is the kth-ranked object of the m objects local to subtree u, the fraction of node v's requests that go to object j ($F(v, j)$) is computed as follows:

$$
\begin{aligned}
F(v,j) &= 0 \quad \text{if } u \text{ is not an ancestor of } v, \\
&= ar^i \quad \text{if } u \text{ is an ancestor of } v \text{ and } PAT \text{ is "Uniform"} \\
&= \frac{ar^i}{k} \quad \text{if } u \text{ is an ancestor of } v \text{ and } PAT \text{ is "Zipf-like"}
\end{aligned}
$$

for an appropriate normalization constant a.

The above workloads ensure that all clients are "almost equally active". However in reality, there may be several caches that will be idle for periods of time. We model this effect by using another parameter I (called the idle cache factor) and by adding a special leaf called the idle leaf for each level-1 node. The idle leaf makes no access requests at all, but it has a cache of size



(a) No idle caches *(b) One idle cache per level 1 node*

Figure 1: *Varying Cache Size*

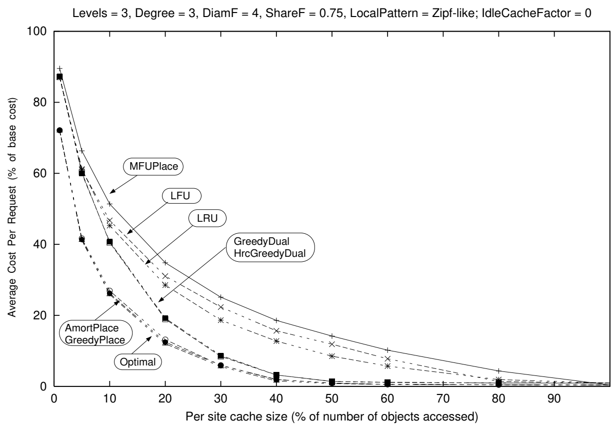


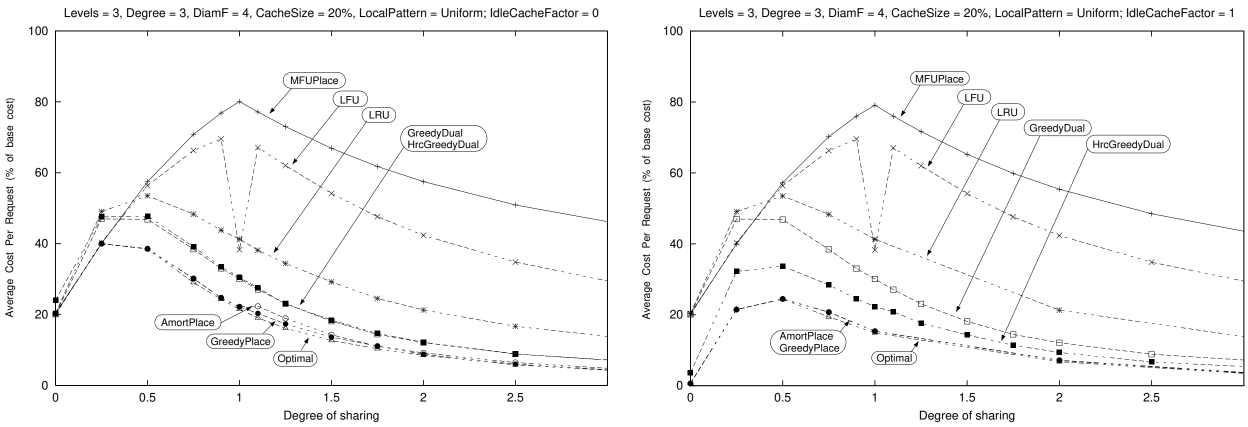
Figure 2: *Varying cache size (with PAT = "Zipf-like")*

I times that of the other leaves. As I is increased from 0 upwards, the amount of idle cache in the system increases.

4.2 Results

Figure 1 plots the performance of the algorithms as the cache size percentage C is varied from 1 to 100, with other parameters set to their default values shown in Table 1. The y-axis corresponds to the average cost per request, as a percentage of the base cost. The latter is the cost that is paid if there are no copies of the object in the hierarchy, and is given by the expression λ^{L+1}. The results for the case where the pattern within each category is Zipf-like, and not uniform, are similar and are presented in Figure 2. The primary conclusion from this data is that increasing coordination can improve performance, particularly with small caches. When comparing the three categories of algorithms—local (*MFUPlace, LFU, LRU, GreedyDual*), hierarchical replacement (*HrcGreedyDual*), and hierarchical placement (*AmortPlace, GreedyPlace, Optimal*)—hierarchical replacement generally outperforms local and hierarchical placement generally outperforms hierarchical replacement.

Within each category, the effect of increasing coordination can also be seen. Although *MFUPlace*, *LFU*, *LRU*, and *GreedyDual* are all "local" algorithms, their performance differs markedly.



(a) No idle caches *(b) One idle cache per level 1 node*

Figure 3: *Varying degree of sharing*

MFUPlace performs poorly because all caches in a subtree contain exactly the same objects from the subtree, which wastes cache space with inefficient replication. LFU and LRU do somewhat better because randomization has a similar effect to coordination—reducing the replication of the most frequently accessed objects while increasing the replication of less frequently accessed ones. Finally, the miss-cost consideration in GreedyDual makes it expensive to throw away objects that are not cached by nearby neighbors, which induces significant coordination across caches. In fact, for the "no idle cache" case, GreedyDual matches the performance of Hrc-GreedyDual. HrcGreedyDual outperforms GreedyDual when there is idle cache space to exploit as Figure 1-b shows.

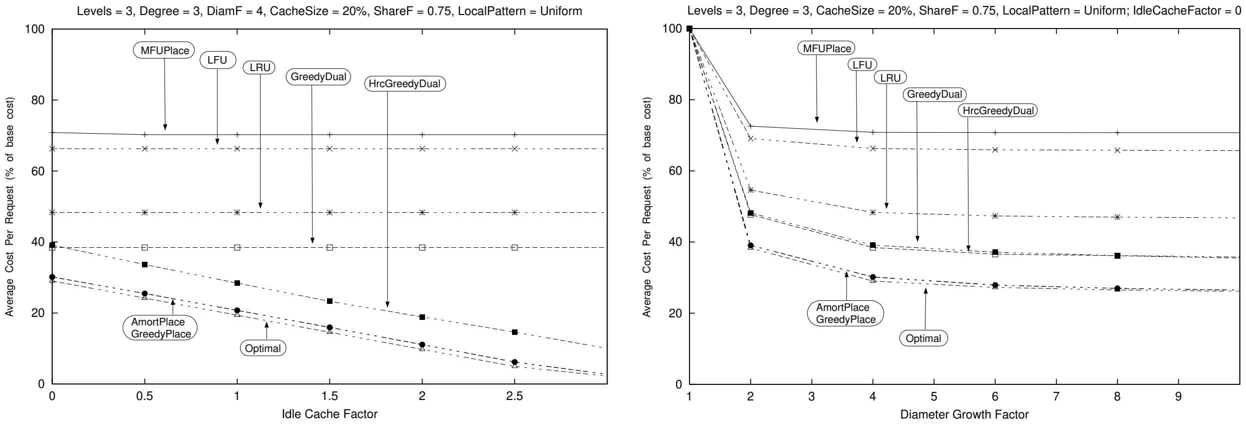
As we increase cache size, the performance of all these algorithms improves. None of the algorithms perform well when caches are tiny, but for small to medium sized caches, the coordinated algorithms significantly outperform traditional replacement algorithms.

We also note that the AmortPlace and GreedyPlace algorithms effectively match Optimal across a wide range of workloads.

4.2.1 Sensitivity to other parameters

Figure 3 shows performance as we vary the sharing parameter, r. Recall that for $r < 1$ clients are more likely to access "local" objects, for $r = 1$ clients are equally likely to access objects shared at all levels, and for $r > 1$ clients focus much of their attention on widely shared objects. For r between 0 and 1, smaller values improve performance for all of the algorithms because smaller values result in clients sending more requests to their "local" collection of objects, of which a large fraction will fit in their local caches under any of the algorithms studied. When $r > 1$, increasing r actually helps performance because sharing among caches becomes more effective. The performance spike for LFU at $r = 1$ occurs because a client is spreading its requests across all levels of objects evenly and it considers all objects equally likely to be referenced; all replacement decisions are ties and are broken randomly, which results in most objects being widely cached. Conversely, if r is, say, slightly less than 1, each cache always favors local objects over higher-level objects, which causes them to be too extensively replicated.

We also note that the same general patterns emerge as for the earlier experiment: across a wide range of sharing factors, algorithms with more coordination have better performance and GreedyPlace and AmortPlace closely track the performance of Optimal.



(a) Idle cache per level 1 node (b) Diameter growth factor

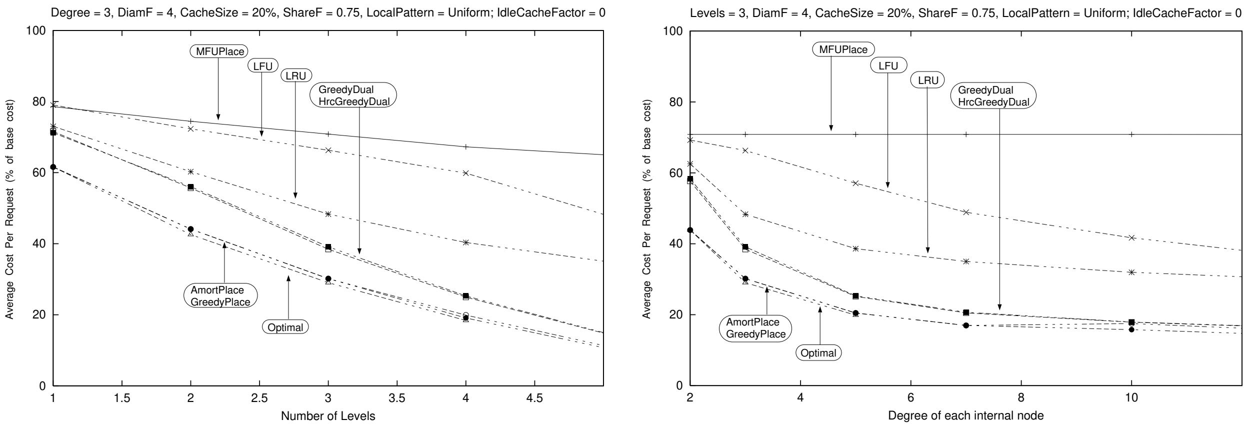
Figure 4: *Sensitivity to (a) varying the idle cache per level 1 node (I), and (b) varying the diameter growth factor (λ).*

Figure 4-a shows what happens as the amount of idle cache per group of three level-one caches increases. When load is not balanced, the "implicit" coordination of LRU and GreedyDual is not able to take advantage of the increasing imbalance. And as load becomes less balanced, the performance of the explicitly coordinated algorithms—HrcGreedyDual, AmortPlace, GreedyPlace, and Optimal—improves.

Figure 4-b shows the impact of varying the diameter growth factor, λ. For small λ, all cache hits and misses have similar cost, and all algorithms have approximately the same performance. For large values of λ, performance is dominated by the number of misses, so the coordinated algorithms perform well. The lines are flat for large λ because neither the number of objects nor the total cache space changes with λ, so the fraction of objects that are not stored in the cache system is constant. The average access cost, which is largely dominated by the accesses to these objects, increases at the same rate as the base cost. Therefore the average access cost as a percentage of the base cost is almost constant.

Figure 5-a shows the variation as L, the number of levels in the hierarchy, increases. Here the total cache space increases at a much faster rate than the total number of objects. (Note that (1) the number of leaves in the tree, say n, is D^L; (2) total number of nodes in the tree is at most $2n$; and (3) each leaf accesses $(L+1)m$ objects and hence has a cache of size $C(L+1)m/100$. Therefore the total cache space in the tree is $nC(L+1)m/100$ while the total number of objects is at most $2nm$.) Hence more objects can be cached in the system, thereby reducing the number of accesses that need to pay the base cost. Hence the average access cost as a percentage of the base cost keeps decreasing.

The variations with increasing D are shown in Figure 5-b. Even though there is an increase in n, the number of leaves in the tree, the ratio of the total number of objects to the total available cache space does not change. Therefore the fraction of accesses that have to pay the base cost is almost fixed and hence there is not much variation in the performance of the algorithms with increasing degree.



(a) Number of levels (b) Degree of internal nodes

Figure 5: *Sensitivity to (a) varying the number of levels (L) and (b) varying the degree of each internal node (D).*

5 Performance evaluation on trace workloads

In this section our goal is to evaluate the performance of the various placement and replacement algorithms on trace workloads. Whereas the synthetic workloads allow us to examine performance over a wide range of situations, the trace workloads allow us to quantify performance gains that may be available under a specific workload of broad interest.

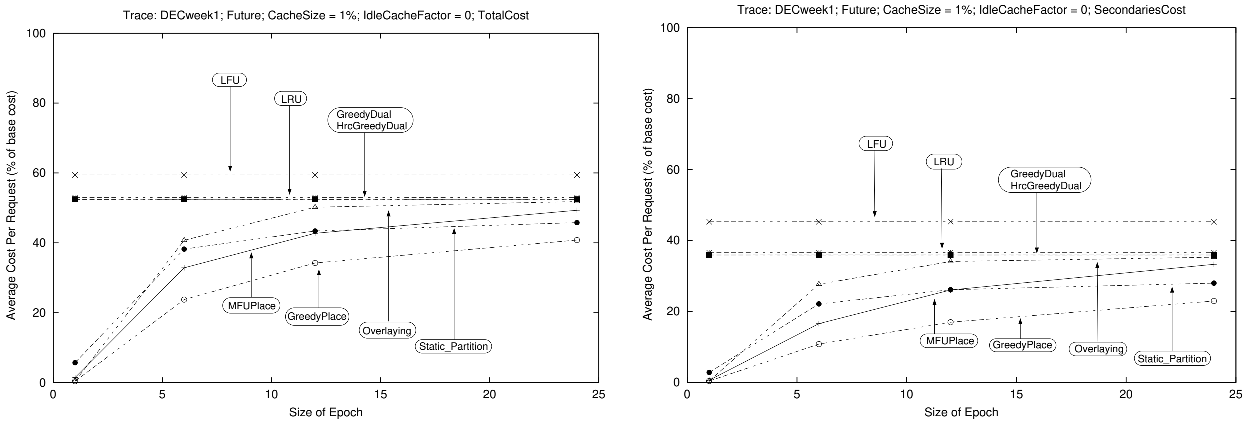
The main conclusions for the synthetic workload are also supported here. In addition, we find that a key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses. As a result, it appears that hybrid placement-replacement algorithms may offer the best option.

5.1 Methodology

Our simulations use the Digital web proxy trace [5], which was collected at a proxy serving about $16,000$ clients over a period of 25 days from August 29th, 1996 to September 22nd, 1996. The number of events logged were about 24 million, using about 4.15 million distinct URLs. For our simulations we use only the cacheable read accesses (i.e., events with *GET* method, without involving CGI scripts etc.). Each such read access specifies among other details, the client making the request, the URL being requested, and the time at which the request was made.

Because the trace does not provide any information regarding the architecture of the network connecting the clients, we use our standard synthetic architecture which was described in section 4.1. We map each trace client on to a random node in our synthetic network. We believe that such a random mapping will generally inhibit the performance of cooperative algorithms. This is because we expect the cooperative algorithms to perform better when there is good similarity of interests among clients that are close to each other. However, a random mapping of clients onto the network nodes would not preserve any such similarity, thereby hurting the performance of the cooperative algorithms.

Also note that we map the 16,000 nodes in the trace to 27 virtual leaves following our default mapping in Table 1. This may be a challenging mapping for coordination algorithms for a number of reasons: with about 600 random nodes multiplexed to each leaf cache, caches



(a) Cost of all requests included *(b) Cost of secondary requests only*

Figure 6: *Ideal Prediction*

are seldom idle and thus there are few chances to exploit idle cache memory. Also, the tree is relatively shallow and has a relatively small degree, which also may reduce performance gains.

As for the synthetic case, we set the cache size as $CM/100$, but because M is defined as the average number of objects seen in a day of the trace rather than the number of objects seen by a single client as in the synthetic workload, we use a smaller value of C ($C = 1\%$) as our default.

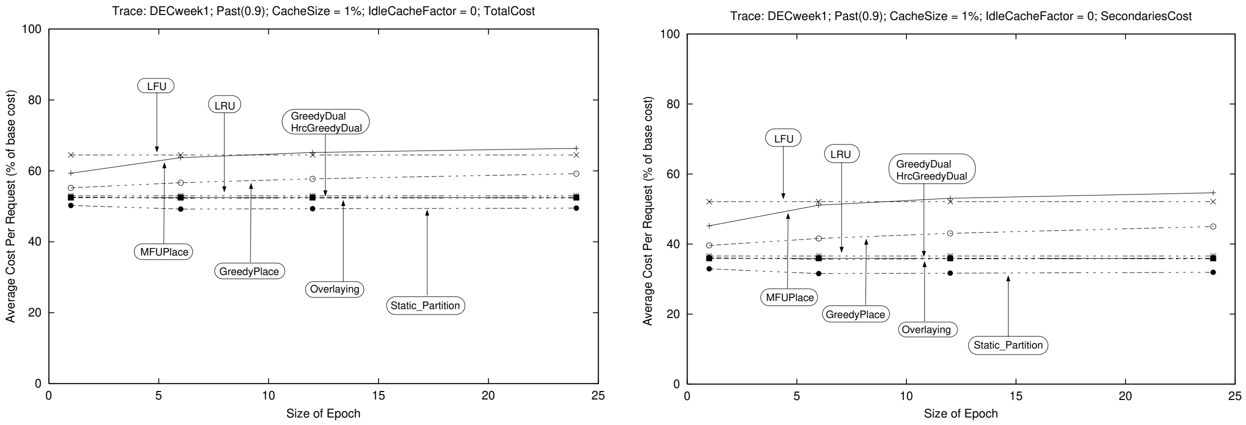
Because the trace includes large numbers of objects and large numbers of cache slots, it is no longer feasible to run the Optimal placement algorithm. The experiments from the previous section suggest that when the GreedyPlace algorithm is given good access-frequency predictions, its performance should closely approximate that of Optimal. Also, because the Amortization step does not appear to be necessary in practice, we omit discussion of the AmortPlace algorithm and focus on the simpler GreedyPlace algorithm.

5.2 Results

We have seen that the placement algorithms yield significant performance gains when the access patterns are stable and known, as was the case for the synthetic workloads. However, in reality access patterns change over time, and an effective placement strategy must be able to cope with these changes. A natural way of coping with dynamically-changing access patterns is to run the placement algorithms at regular intervals to reorganize the data more effectively. However, there are two crucial factors that affect the performance of such a strategy: (1) How frequently should the placement algorithms be run? and (2) How do we predict the access frequencies for use by the placement algorithm?

The dynamic versions of our placement algorithms break the time into *epochs* and run the placement algorithm at the beginning of every epoch. If the epoch size were too large, then the placement would get outdated and hence yield bad performance. On the other hand, if the epoch size were too small then the bandwidth cost of reorganizing the data would be prohibitive. For our experiments, the epoch size is a parameter which can be varied. The main focus of our trace-based study is to evaluate how the performance of the various algorithms changes as the epoch size is varied from one hour to one day.

A key challenge for placement algorithms is to predict the future access frequencies based on past accesses. Ideally, a sophisticated prediction technique would exploit the temporal, spatial,



(a) *Cost of all requests included* (b) *Cost of secondary requests only*

Figure 7: *Naive history-based prediction*

and geographical localities among requests to predict future requests.[1] However we believe that a study of these techniques is orthogonal to our current focus which is the design of good placement and replacement strategies. Hence to avoid digression, we do not delve deeply into this question.

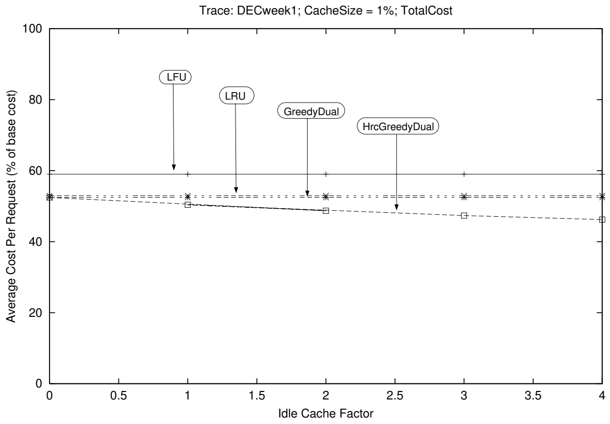
For the purposes of evaluating our placement strategies, we consider two extreme prediction strategies. The first one is an idealized predictor based on future knowledge that looks ahead into the next epoch to determine the access frequencies for each (client, object) pair. This unrealizable algorithm serves as a benchmark for the best any prediction technique can achieve. The second one is a naive predictor that computes the predicted access counts for the next epoch using the access counts from the earlier epochs, using a damping factor r: if the access count for a particular (client, object) pair was c_i during the ith last epoch, then that epoch contributes $c_i \cdot r^{i-1}$ to the predicted access count for the coming epoch.

Finally, to cope with the dynamic access patterns in these traces, we examine the performance of hybrid placement-replacement algorithms. These hybrid algorithms run a placement algorithm at epoch boundaries and also run a replacement algorithm during the epoch. We examine two hybridization techniques. *Static partition* divides the cache space into two portions and runs the placement algorithm on one portion and the dynamic replacement algorithm on the other. In our experiments, we use half of the cache for each partition. The second technique, *overlaying*, reorganizes the entire cache using the placement algorithm at the start of each epoch and then gives the replacement algorithm control of the entire cache during the epoch.

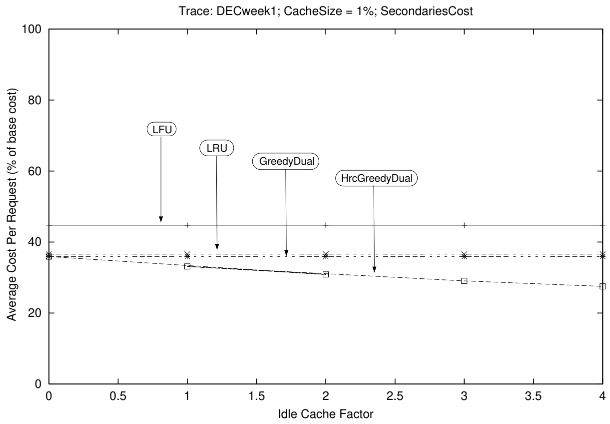
Figure 6 shows performance with the ideal, future-knowledge-based predictor and Figure 7 show performance with the simple history-based predictor. For both figures, the x-axis shows the epoch length in hours and the y-axis shows performance. Because significant numbers of web objects are referenced only once, which can have a significant effect on overall performance, we show both the overall performance (Figures 6-a and 7-a) and the performance for objects after their first reference (Figures 6-b and 7-b).

Both epoch length and the prediction algorithm are important for placement algorithms.

[1]Spatial locality refers to the fact that related objects such as objects from the same server or that are hyperlinked to each other tend to be accessed together. Geographical locality refers to the fact that clients that are close to each other may have similar interests.



(a) *Cost of all requests included*



(b) *Cost of secondary requests only*

Figure 8: *Replacement algorithms with varying idle cache factor.*

For the idealized prediction algorithm, performance is excellent when epochs are short. For longer epochs, placement is coarser-grained and gains are more modest. Even with only daily reorganization, however, performance gains can be significant. Also, when predictions are good, the placement algorithms perform well and the hybrid algorithms hurt performance.

When access frequency predictions are less precise, the placement algorithms are less effective. In Figure 7, the most effective algorithms combine coordinated placement and coordinated replacement. Even with this combination, under this set of parameters the performance gains are modest. This result suggests that developing more accurate frequency predictors could be a fertile area for future work.

As was noted above, the cache architecture used for this system may be a challenging one for placement algorithms. For example, we multiplex about 600 random clients per leaf cache, which severely limits the likelihood of locating idle cache space to exploit. Figure 8 shows the effect of adding idle cache to each L1-cluster. This change noticeably improves the performance of the hierarchical GreedyDual algorithm.

6 Related work

A number of recent studies have examined the question of what to store in the caches. There are several studies and prototypes (e.g., [4, 3, 16, 23]) that employ purely local replacement strategies such as LRU or GreedyDual at each cache. The GreedyDual local replacement algorithm which considers the differing miss costs and differing object sizes while making replacement decisions was presented and evaluated in [3, 12, 25]. However, none of these studies address the cache coordination issue.

For local-area networks, some fundamental cooperative caching heuristics were experimentally evaluated and found to yield significant benefits [6, 8]. An optimal algorithm for the placement problem on local-area networks was presented in [15]. Recently, in [26], the question of cache coordination has been studied, albeit under a simplistic model where all the network distances are assumed to be the same. In such a scenario, the simple strategy of avoiding duplication altogether is obviously the best strategy, and hence the coordination problem is trivial. Moreover their simulation experiments involve only one client.

For wide-area networks, the issue of server-initiated *on-line replication* has received a good deal of attention [11, 13, 17, 18, 24]. Two of these [13, 17] give analytical results while the remaining three [11, 18, 24] present heuristics with empirical evaluation. In all these studies, the concern is more with the issue of reducing server load when hot-spots occur (or when the load on server increases) and less with issue of reducing the latency when there are no hot spots.

By adopting a communication model based on a fixed cost function, we endeavor to separate the concerns of caching (a higher-level operation) from network routing (a lower-level operation). In contrast, some recent papers have incorporated routing issues into caching by either combining the two problems or making use of available routing information. For example, the algorithms developed in [11, 18, 24] tend to cache copies of an object in machines that either lie on or are close to the path along which the object is being transferred. We remark that fixed cost functions, similar to ours, have also been adopted in local replacement algorithms such as the GreedyDual, to model scenarios where the fetch costs vary from one object to another [3, 12, 25].

In push caching schemes [10, 20], the server keeps track of client access patterns and pushes data towards the clients even before they ask for it. This reduces the client latency by avoiding compulsory misses. There are two parts to these schemes: predicting future access patterns and distributing the data according to these predictions. The above studies aim at evaluating the potential benefits of push caching by focusing on the first part and assuming that the cache sizes are infinite. Our placement problems address the second part of these schemes, under the more realistic assumption that the cache sizes are bounded.

7 Conclusions

A current trend in large-scale cache systems is to generalize cooperation beyond traditional hierarchies. Much recent work has explored solutions to the general object-location problem. In this study, we examine the object-placement problem in this context. Based on our simulation studies, we reach five primary conclusions: (1) cooperative placement can significantly improve performance compared to local replacement algorithms; (2) although the factor separating our *Amortized Placement* and *Greedy Placement* algorithms from optimal placement can be relatively large in the worst case, in practice they seem to provide excellent approximations of optimal; (3) in a cooperative caching scenario, the *GreedyDual* local replacement algorithm performs much better than the other local replacement algorithms; (4) our *Hierarchical GreedyDual* replacement algorithm yields further performance improvements over the GreedyDual algorithm especially when there are idle caches in the system; and (5) a key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses. Hybrid placement-replacement algorithms appear to be one way to cope with inaccuracies in such predictions.

References

[1] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Rosselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, 1995.

[2] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The Harvest information discovery and access system. In *Proceedings of the 2nd International World Wide Web Conference*, pages 763–771, October 1994.

[3] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[4] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the USENIX 1996 Technical Conference*, pages 22–26, January 1996.

[5] Digital Equipment Corporation. Web proxy traces, september 1996. Available via ftp from ftp://ftp.digital.com/pub/DEC/traces/proxy.webtraces.html.

[6] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, 1994.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265, August 1998.

[8] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[9] Syam Gadde, Michael Rabinovich, and Jeff Chase. Reduce, reuse, recycle: An approach to building large internet caches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 1997. http://www.cs.duke.edu/ari/cisi/crisp-recycle.ps.

[10] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 51–57, May 1995.

[11] A. Heddaya and S. Mirdad. WebWave: Globally load balanced fully distributed caching of hot published documents. In *Proceedings of 17th International Conference on Distributed Computing Systems*, May 1997.

[12] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 701–710, May 1997.

[13] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[14] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999. To appear.

[15] A. Leff, J. L. Wolf, and P. S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204, 1993.

[16] P. Lorenzetti, L. Rizzo, and L. Vicisano. Replacement policies for a proxy cache. Technical report, 1997. http://www.iet.unipi.it/ luigi/research.html.

[17] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 570–579, October 1996.

[18] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the internet. Technical report, AT&T Labs – Research, April 1998.

[19] A. Rousskov and D. Wessels. Cache digests. In *Proceedings of the 3rd International WWW Caching Workshop*, June 1998. http://wwwcache.ja.net/events/workshop/31/rousskov@nlanr.net.ps.

[20] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR–98–04, Department of Computer Science, University of Texas at Austin, January 1998.

[21] M. Van Steen, F. J. Hauck, and A. S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of the 1996 Conference on Telecommunications Information Networking Architecture (TINA 96)*, pages 203–212, September 1996.

[22] D. Wessels. Squid internet object cache. http://squid.nlanr.net/Squid, January 1998.

[23] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla, and E.A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of the 1996 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, August 1996.

[24] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(4):255–314, 1997.

[25] N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, January 1998.

[26] P. S. Yu and E. A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. In *Proceedings of Seventh International World Wide Web Conference*, Brisbane, Australia, 1998.

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