This paper appears in the Workshop on Internet Applications, July 1999

Coordinated Placement and Replacement for Large-Scale Distributed Caches *

Madhukar R. Korupolu Michael Dahlin
Department of Computer Sciences
University of Texas at Austin
{madhukar,dahlin}@cs.utexas.edu

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. 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 conclusions from these experiments: (1) cooperative placement can significantly improve performance compared to local replacement algorithms particularly when the size 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; (4) our* Hierarchical GreedyDual *replacement algorithm yields further 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. Caching popular objects close to 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 as well as the server.

A powerful paradigm to improve cache effectiveness is *cooperation*, where caches cooperate both in serving each other's requests and in making storage decisions. Such cooperation 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 [1, 3, 20]. Many recent studies (e.g., Summary Cache [6], Cache Digest [18], Hint Cache [19], CRISP [8] and Adaptive Web Caching [25]) also focus on the location problem, but none of these address the 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.

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 each object, determine where to place each object in order to minimize the average access cost. Compared to placement algorithms, replacement algorithms also attempt to minimize the access cost, but rather than explicitly computing a

*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).

placement based on access frequencies, they proceed by evicting objects when a cache miss occurs.

Coordinated caching helps for two reasons. First, coordination would allow a busy cache to utilize a nearby idle cache [5, 7]. Second, coordination can balance the improved hit time achieved by increasing the replication of popular objects against the improved hit rate by reducing replication and storing more unique objects.

In this work, we examine an optimal placement algorithm and three practical placement algorithms and compare them to several uncoordinated replacement algorithms (such as LFU, LRU, GreedyDual [2, 23]) and a novel coordinated replacement algorithm. We drive this comparison with simulation 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.

We draw five conclusions from these experiments.

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

- It was established in an earlier theoretical work by Korupolu, Plaxton and Rajaraman [13] that, under a hierarchical model for distances, the *Amortized Placement* algorithm is always within a constant factor of the optimal. Although the proof only guarantees that the amortized placement algorithm is within a factor of 14 of the optimal, in practice we find that it is within 5% 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 that can be used to evaluate other algorithms. In addition, we find that the *Greedy Placement* algorithm, which is a simplified version of the amortized algorithm, also provides an excellent approximation of the optimal even though in theory its performance can be arbitrarily worse than the optimal.

- Previous work [2] has shown that the GreedyDual algorithm works well for stand-alone caches. Our contribution is to examine its performance in cooperative caching scenarios. We find that, for cooperative caching, it significantly outperforms other local replacement algorithms because it includes miss costs in its replacement decisions, thereby creating an implicit channel for coordinating the caches.

- Our *Hierarchical GreedyDual* replacement algorithm yields further improvements over the GreedyDual 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.

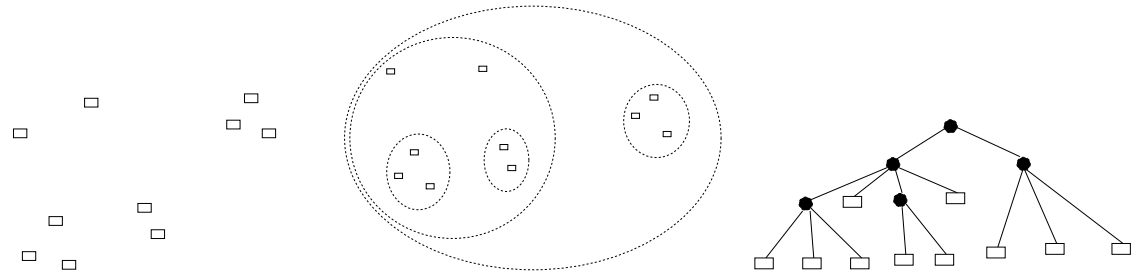
The rest of the paper is organized as follows: First, Section 2 describes the algorithms we study. Sections 3 and 4 detail our experimental results under synthetic and trace workloads, respectively. Section 5 surveys related work, and Section 6 summarizes our conclusions.

2. Algorithms

In this section, we present several placement and replacement algorithms for coordinated caching. However, this is preliminary work in this direction and hence we make several simplifying assumptions in order to focus on the coordination problem. One assumption is that all the objects have the same size and are read-only. Second, we assume that the network distances (or communication costs) between node pairs are fixed and do not change over time. An interesting area for future work is to relax these assumptions.

In order to capture the varying degrees of locality between the nodes, we use a clustering based network model. This is illustrated in Figure 1 which shows a set of cooperating nodes, and a possible network-locality based clustering of these. This clustering is a natural consequence of how network topologies reflect organizational and geographic realities. For example, in a collection of universities, each node typically belongs to the department cluster which in turn belongs to the university cluster and so on. This cluster structure can be captured using a *cluster-tree* (or, a network-locality tree) as shown in the figure. The individual caches form the leaves of this tree, and the internal nodes correspond to the clusters. A cluster C is a child of cluster C' if C is immediately contained within C'.

Because communication between two clusters is likely to traverse the same bottleneck link regardless of which particular nodes are conversing, we use a simple model of network distances: each cluster has an associated *diameter*, and the distance between any pair of nodes is given by the *diameter* of the smallest cluster that contains both these nodes. (This model is same as the ultrametric model used by Karger et al. in [12].)



(a) A set of cooperating caches *(b) A clustering* *(c) The cluster tree*

Figure 1. *Model for network distances*

2.1. Non-cooperative local algorithms

In this subsection, we outline four baseline algorithms that make all their placement or replacement decisions locally without consulting any other cache.

MFU Placement. The cache looks at the local frequencies to the various objects, and if the size of the cache is k, it stores the k most frequently used objects.

LRU Replacement. When a cache miss occurs, this algorithm evicts the least recently used object.

LFU Replacement. When a cache miss occurs, this algorithm evicts the object with the least (local) frequency of access.

GreedyDual Replacement. This is a generalization of the LRU algorithm to the case where each object has a different but fixed miss cost [2, 23]. 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 only the relative values, not the absolute ones, that matter [2]. 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.

2.2. Cooperative placement algorithms

A *placement* assigns objects to caches without violating the cache size constraints. The cost of a placement P is defined in the natural manner: the sum over all nodes u and all objects α of the access frequency for object α at node u times the distance from node u to the closest copy of that object. The goal of a cooperative placement algorithm is to compute a placement with minimum cost. Even though we do not explicitly minimize the network load and the server load these would typically fall when the access cost is minimized. This is because the latter objective would encourage objects to be stored closer to the clients, thereby reducing the load on both the network as well as the server.

We study three cooperative placement algorithms. One of them is provably optimal, but unfortunately it is impractical for scenarios with large numbers of nodes and objects. The other two algorithms are not provably optimal, but they are much simpler and can be implemented efficiently even in a distributed setting.

2.2.1. An optimal placement algorithm

A centralized optimal algorithm for the placement problem was developed in an earlier paper [13], using a reduction to the minimum cost flow problem. The algorithm and its proof of optimality appear in [13], hence we do not reproduce it here.

Since the minimum cost flow problem is computationally intensive, this optimal algorithm incurs a high running time complexity. Moreover, since the algorithm is centralized, it requires all the frequency information to be transferred to a single node, thereby imposing a high bandwidth requirement. These factors make this algorithm impractical for use with large inputs, and hence our sole use for this algorithm is as a benchmark for evaluating other algorithms.

2.2.2. The greedy placement algorithm

This algorithm follows a natural greedy improvement paradigm, and involves a bottom-up pass along the

cluster-tree. It starts with a tentative placement in which each cache (i.e., a leaf in the cluster-tree) picks the locally most valuable set of objects. The algorithm then proceeds up the cluster-tree improving the placement iteratively.

In a general step, suppose we have computed the tentative placements for clusters $C_1, \ldots C_k$ which constitute a larger cluster C. While computing the placement for cluster C_i, the algorithm uses the access frequency information from within that cluster only. Now at cluster C, we first merge the placements computed for subclusters C_1 through C_k. The placement P obtained by this merging is clearly a starting placement for cluster C, but it may be improveable using the information about the aggregate frequencies across different subclusters in C.

For example, there may be an object α that is not chosen in any of the clusters C_1 through C_k since its access frequency within each cluster is small. But its aggregate frequency in the larger cluster C may be large enough that the placement P can be improved by taking a copy of object α and dropping a less beneficial item from P. To determine such useful swaps, we calculate a *benefit* for each item in P and a *value* for each object not in P. (Such objects are said to be P-*missing*.)

The value of a P-*missing* object is C's aggregate access frequency for that object times the cost of leaving the cluster C to fetch that object. The latter quantity is the difference between the diameter of the parent cluster of C and that of the cluster C itself.

The benefit of an item x in placement P, on the other hand, corresponds to the increase in the cost of the placement when the item x is dropped. Benefits are computed in a bottom-up manner. Each subcluster C_i calculates a local benefit for each item in its placement. After the merge step, the parent cluster C updates the benefits as follows. For each object that has one or more copies in P, we pick the copy with the highest local benefit as the *primary* copy, and call all other copies as *secondary* copies. The benefits of the secondary copies are not changed, but the benefit of the primary copy is increased by C's aggregate access frequency to the object times the cost of leaving the cluster C. The intuition is that among all the copies of an object, the primary copy will be the last one to be removed from P, and its removal will increase the cost of the placement by the above amount.

Once the benefits and values have been computed, we use a simple greedy swapping phase to improve the placement P. While there is a P-missing object α whose value is more than the least beneficial item x in P, we remove x from P and substitute a copy of α. This phase terminates when the benefit of each item in P is higher than the value of each P-missing object.

This swapping phase concludes the computation for cluster C, and the algorithm proceeds to the parent cluster of C iteratively.

Though this greedy algorithm looks simple and promising, it is shown in [13] that its worst-case performance can be arbitrarily far from the optimal. However, we conjecture that such worst-case examples occur rarely and that the algorithm would perform well in practice.

2.2.3. The amortized placement algorithm

The worst-case analysis indicates that a drawback of the greedy algorithm is the following: A single secondary copy of some object may prevent the swapping in of several missing objects. Though the benefit of the secondary copy may be larger than the value of each of the missing objects, on the whole it might be much less than the sum of all these values put together.

To circumvent this drawback, the greedy algorithm is augmented with an amortization step using a potential function. The potential function accumulates the values of all the missing objects, and the accumulated potential is then used to reduce the benefits of certain secondary items thereby accelerating their removal from the placement. Due to space constraints, we omit the full description of the algorithm.

It is proved in [13] that the above amortized placement algorithm is always within a constant factor of the optimal. The constant factor is about 13.93. However, this factor is still large for practical purposes. We conjecture that, in practice, this algorithm will be much closer to the optimal.

2.3. A cooperative replacement algorithm

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 value structure enables some implicit coordination with other caches. In particular, an object that is fetched from a nearby cache has a smaller value than an object that is fetched from afar. Hence the former object would be evicted from the cache first, thus 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 nearby busy caches.

Hence we devise *Hierarchical GreedyDual*, a cooperative replacement algorithm that not only preserves the implicit coordination offered by GreedyDual but

also enables busy caches to utilize nearby idle caches.

In this algorithm, each individual cache runs the local GreedyDual algorithm using the efficient implementation described in subsection 2.1. Recall that the local GreedyDual 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" to the parent cluster for possible inclusion in one of its caches. When a cluster C receives an evicted object α from one of its child clusters, it first checks to see if there is any other copy of α among its caches. If not, it picks the minimum valued object β among all the objects cached in C. Then the following simple admission control test is 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 cluster of C. Otherwise, the object α itself is recursively passed on to the parent cluster of C.

From our experiments, we learned that the particular admission control test described above is crucial for obtaining good performance. This is because 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 repeatedly reinsert such objects even after they are evicted by individual caches. This can 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. With this timestamp based admission control strategy, rarely used objects are eventually released from the system.

For a practical implementation, algorithm would use data-location directories [6, 8, 18, 19] to determine if other copies exist in the subtree, and would use randomized [5] or deterministic [7] strategies to approximate the selection of β.

3. Performance evaluation on synthetic workloads

This section explores the performance of the above 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 the optimal placement.

This section first describes our methodology in detail and then shows the results of our experiments. These results support the first four conclusions listed in Section 1.

Param.	Meaning	Default
L	Number of levels	3
D	Degree of each internal node	3
λ	Diameter growth factor	4
C	Cache size percentage	20% (synth. only)
m	No. of local objects per node	25 (")
r	Sharing parameter	0.75 (")
PAT	Access pattern	Uniform (")
I	Idle cache factor	1

Table 1. Default system parameters.

3.1. Methodology

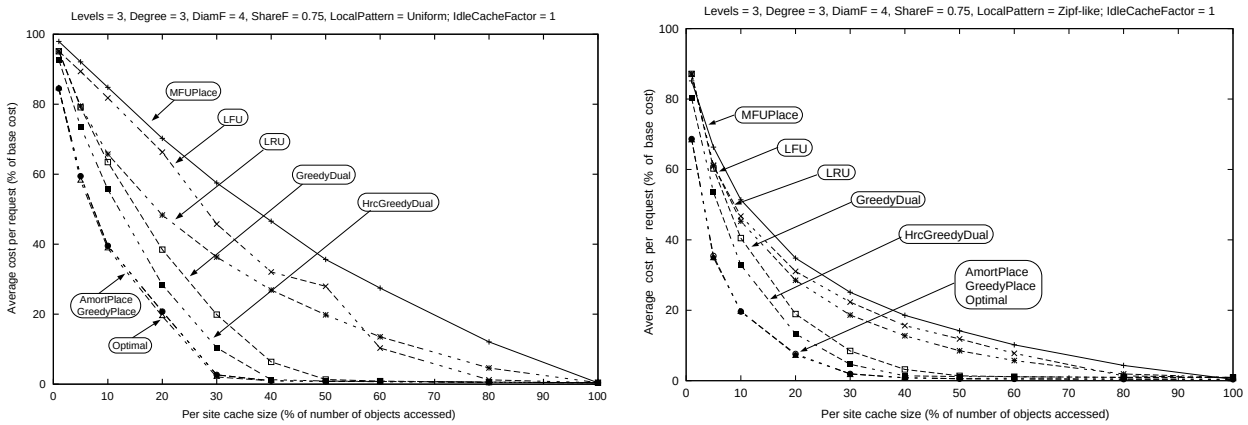
We simulate a collection of caches that include a directory system, such as the ones provided by Hint Cache [19], Summary Cache [6], Cache Digests [18] or CRISP [8], so that caches can send each local miss directly to the nearest cache or server that has the data. For the placement algorithms MFU, Greedy, Amortized, and 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, and 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 phase to prime the caches before gathering statistics.

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.

3.1.1. Network architecture

Recall from Section 2 that the distances between the cache nodes are completely specified once the network-locality tree and the cluster diameters are given. We create an L-level network-locality tree with the degree of each internal node being D. The root is considered to be at level L and the leaves are at level zero. The cluster diameters are captured by λ, the diameter growth factor. The diameter for a cluster at level i is λ^i, and the cost of leaving the root cluster is λ^{L+1}.

Because all objects have the same size, it suffices to express the size of a cache in terms of the number of objects it can hold. We set all cache sizes to be the same, using a single parameter C which is called the cache size percentage. Specifically, the cache size at a node is set to $CM^*/100$, where M^* is the average number of objects accessed by the node.



(a) With PAT = "Uniform" *(b) With PAT = "Zipf-like"*

Figure 2. *Varying Cache Size*

3.1.2. Workload

As observed in Section 1, an important parameter for the performance of cooperative strategies is the degree of similarity of interests among nearby nodes. 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 cluster in the network. This pattern could represent a hierarchical organization 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. The fraction of requests that a client sends to level-i objects is proportional to r^i. Note that as r varies from 0 to infinity, the degree of sharing increases: when $r < 1$, clients are more likely to access "local" objects, when $r = 1$ they are equally likely to access objects from all levels, and when $r > 1$ they focus much of their attention on "global" objects.

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

$$\begin{aligned} F_v(j) &= 0 && \text{if } C \text{ does not contain } v, \\ &= ar^i && \text{if } C \text{ contains } v \text{ and } PAT \text{ is "Uniform"}, \\ &= \frac{ar^i}{k} && \text{if } C \text{ contains } v \text{ and } PAT \text{ is "Zipf-like"}, \end{aligned}$$

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 are idle for periods of time. We model this effect by using another parameter I (called the idle cache factor) and by adding a special cache called the idle cache for each level-one cluster. The idle cache makes no access requests at all, but it has a cache of size I times that of any other cache. As I is increased from 0 upwards, the amount of idle cache space in the system increases.

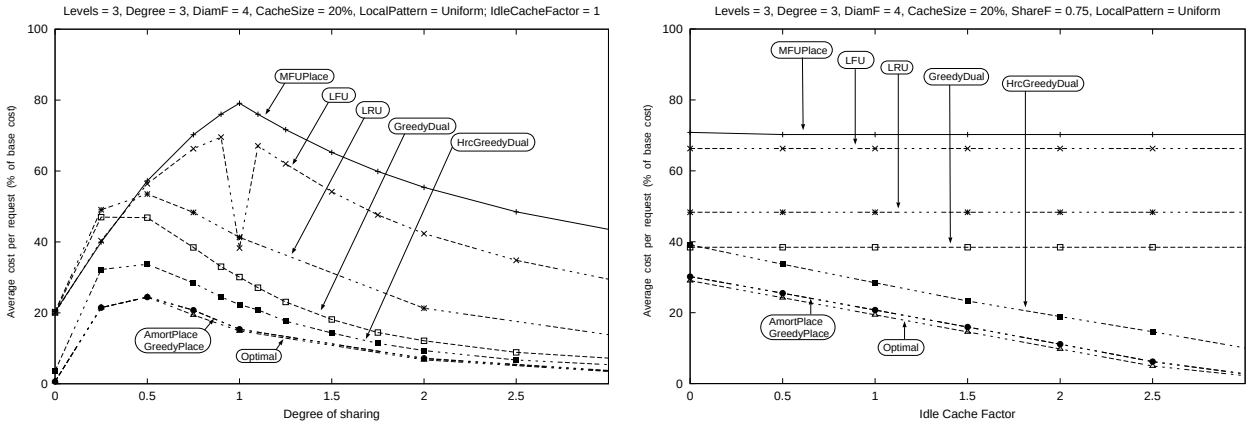
3.2. Results

Figure 2 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.

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 are similar and are presented in Figure 2-b.

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*), cooperative replacement (*HrcGreedyDual*), and cooperative placement (*AmortPlace, GreedyPlace, Optimal*)—cooperative replacement generally outperforms local, and cooperative placement generally outperforms cooperative 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. MFUPlace per-



(a) Varying degree of sharing *(b) Varying idle cache per level-one cluster*

Figure 3. *Varying degree of sharing and idle cache factor*

forms poorly because caches tend to contain exactly the same objects from the cluster, which wastes cache space with inefficient replication. LFU and LRU do somewhat better because randomization has an effect similar 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 HrcGreedyDual (Figure 3-b). However, when there is idle cache space to exploit, HrcGreedyDual outperforms GreedyDual as Figure 2-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 amortized and the greedy placement algorithms effectively match optimal across a wide range of workloads.

3.2.1. Sensitivity to other parameters

Figure 3-a shows performance as we vary the sharing parameter, r. When r is between 0 and 1, smaller values have better 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.

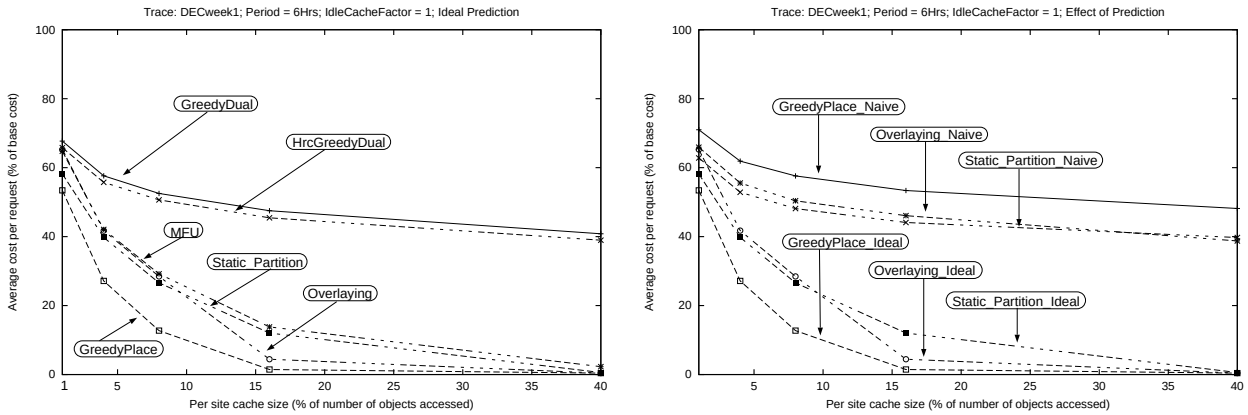
We also note that the same general patterns emerge as for the earlier experiment: across a wide range of sharing factors, algorithms with more coordination have better performance and GreedyPlace and AmortPlace closely track the performance of Optimal.

Figure 3-b shows what happens as the amount of idle cache in each level-one cluster is increased. The "implicit" coordination of LRU and GreedyDual is not able to take advantage of the increasing idle cache space. On the other hand, the performance of the explicitly coordinated algorithms—HrcGreedyDual, AmortPlace, GreedyPlace, and Optimal—improves.

Similar trends were observed when the other parameters (namely, L, D, and λ) were varied. A discussion of the sensitivity to these parameters will appear in the full version of this paper.

4. Performance evaluation on web-trace workloads

In this section our goal is to evaluate the performance of the various placement and replacement algorithms on trace workloads. 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.



(a) With ideal prediction *(b) Effect of prediction*

Figure 4. *Performance with the DEC procy trace*

4.1. Methodology

Our simulations use the Digital web proxy trace [4], 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. About 24 million events were logged 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, et cetera).

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 3.1. This synthetic network has 27 nodes, and we map each of the 16,000 trace clients randomly on to one of these 27 nodes.

We believe that such a random mapping will generally inhibit the performance of cooperative algorithms for two reasons. Firstly, random mapping of clients to clusters will eliminate any similarity of interests among nearby clients. Secondly, with about 600 random nodes multiplexed to each leaf cache, cache load is evenly balanced and hence there would be few chances to exploit idle cache memory.

The large number of objects in the trace makes it infeasible to run the optimal placement algorithm. Nevertheless, as seen in the previous section, we can obtain almost optimal placements by using the simpler greedy placement algorithm. Hence we focus on the greedy placement algorithm alone.

4.2. Design Issues

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 algorithms?

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, we set the epoch size to 6 hours.

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, and geographical localities among requests to predict future requests. (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.) However a study of these techniques is orthogonal to our current focus, and we do not delve into this question.

For the purposes of evaluating the placement algorithms, 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 bench-

mark 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, along with 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.

4.3. Results

Figure 4-a shows the performance of the various algorithms under the ideal prediction strategy described above. The x-axis shows the per-node cache size, as a percentage of the number of objects accessed by the node, and is varied from 1 to 40. All the other parameters are set to their default values.

The experiments show that, when the predictions are ideal, the placement algorithms perform significantly better than the replacement algorithms. This is particularly encouraging for systems that can provide good predictions of access patterns (e.g., subscription based systems). The effect of increasing coordination can also be seen: the Hierarchical GreedyDual outperforms the GreedyDual algorithm while the greedy placement algorithm performs better than the local-MFU placement algorithm. However hybridization hurts the performance of the placement algorithms when the predictions are good.

Figure 4-b shows the effect of prediction on the performance of the placement algorithms by comparing these algorithms with naive prediction at one extreme and ideal prediction at the other. With naive predictions, the most effective algorithms are the hybrid combinations of placement and replacement. Even with these combinations, the performance gains are modest for this set of parameters.

This suggests that developing more accurate frequency predictors could be a fertile area for future work. The reason for this optimism is that our current naive predictor uses only one type of locality, namely the temporal locality, out of the three localities expected in web accesses.

5. Related work

A number of recent studies have examined the question of what to store in caches. There are several studies and prototypes (e.g., [3, 2, 17, 21]) that employ purely local replacement strategies such as LRU or GreedyDual at each cache. The GreedyDual local replacement algorithm was evaluated in [2, 11, 23], but for single stand-alone caches only. Here we study their performance in a cooperative caching scenario.

The placement and replacement algorithms for local-area networks were studied by Leff et al. [14], Dahlin et al. [5], and Feeley et al. [7]. However the scenario of wide-area networks is vastly different and relatively unexplored.

Recently, Yu and MacNair [24] studied the question of wide-area cache coordination, but under a very simplistic model where all the network distances are assumed to be the same. In such a scenario, clearly the best strategy is simply to avoid duplication altogether. Here, we study a more general problem with non-uniform network distances.

The issue of server-initiated on-line replication has received a good deal of attention [10, 12, 15, 16, 22] recently. Two of these [12, 15] give theoretical results while the remaining three [10, 16, 22] present heuristics with empirical evaluation. In all of these studies, the concern is more with the issue of reducing server load when hot-spots occur (i.e., when the load on server increases) and less with the issue of reducing the latency when there are no hot spots.

Another useful technique for reducing the client latency is that of push caching [9, 19]. Here the server keeps track of client access patterns and pushes data towards the clients even before they ask for it, thus avoiding the compulsory misses. Such schemes involve two orthogonal components: predicting future access patterns and distributing the data according to these predictions. The studies in [9, 19] aim at evaluating the potential benefits of push caching by focusing on the first component and assuming that the cache sizes are infinite. Our study of placement problem address the second component of push caching, under the more realistic assumption that the cache sizes are bounded. The placement algorithms proposed here are ideal for systems, such as subscription-based services, where good predictions of access patterns are available.

6. Conclusions

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 this study, we evaluate several placement and replacement algorithms that allow caches to coordinate their storage decisions. Based on our simulation studies, we conclude that coordinated decision making can significantly improve performance. A fertile area for future work is to generate good predictions of access patterns based on past accesses. Other interesting questions are to extend the algorithms described here to handle multiple object sizes, and to cope with the fact that network distances may vary dynamically depending on the network load.

References

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

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

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

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

[5] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, 1994.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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, 1997.

[13] 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*, pages 586–595, January 1999. Invited to appear in the special issue of *Journal of Algorithms* devoted to selected papers from SODA '99.

[14] 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.

[15] 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.

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

[17] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. Technical Report RN/98/13, Dept. of Computer Science, University College London, UK, 1998. http://www.iet.unipi.it/ luigi/.

[18] A. Rousskov and D. Wessels. Cache digests. In *Proceedings of the 3rd Intl. WWW Caching Workshop*, June 1998. http://wwwcache.ja.net/events/workshop/.

[19] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the internet. In *Proceedings of the 19th Intl. Conference on Distributed Computing Systems*, 1999.

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

[21] 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 ACM-SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 1996.

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

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

[24] P. S. Yu and E. A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. In *Proceedings of 7th Intl. World Wide Web Conference*, Brisbane, Australia, 1998.

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