Coordinated Placement and Replacement for Large-Scale Distributed Caches

Madhukar R. Korupolu and Michael Dahlin, *Member*, *IEEE*

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 noncooperative 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 *amortizing 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 *greedy-dual* local replacement algorithm performs much better than the other local replacement algorithms; 4) our *hierarchical-greedy-dual* replacement algorithm yields further improvements over the greedy-dual 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.

Index Terms—Cache, cooperative, distributed, hierarchical, placement, replacement, web.

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 [26]) also focus on the location problem, but none of these address the placement issue which is the focus of this article.

Efficient coordinated object placement 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 placement based on access frequencies, they proceed by evicting objects when cache misses occur.

Coordinated caching helps for two reasons. First, coordination allows a busy cache to utilize a nearby idle cache [5], [7]. Second, coordination balances the improved hit time achieved by increasing the replication of popular objects against the improved hit rate achieved 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, and greedy-dual [2], [24]) 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.

- *M.R. Korupolu is with Akamai Technologies, Cambridge, MA 02139. E-mail: madhukar@alumni.cs.utexas.edu.*
- *M. Dahlin is with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712. E-mail: dahlin@cs.utexas.edu.*

Manuscript received 2 Aug. 1999; revised 20 Sept. 2000; accepted 1 Dec. 2000.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110344.

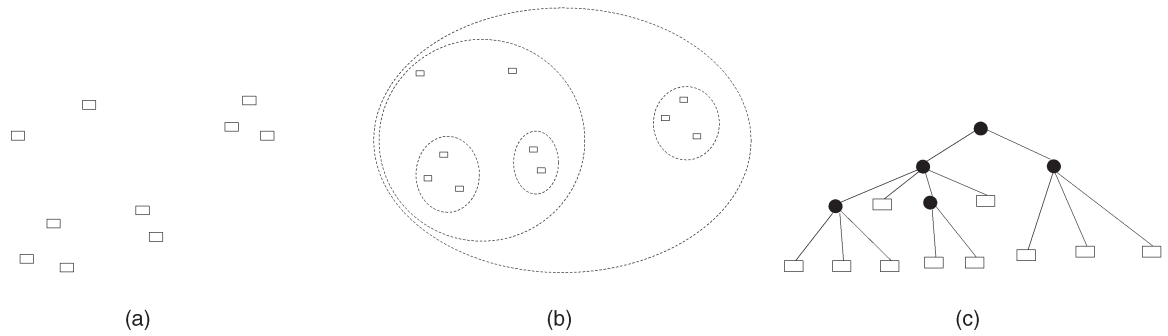


Fig. 1. Model for network distances. (a) A set of cooperating caches, (b) a clustering, and (c) the cluster tree.

- 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 et al. [13] that, under a hierarchical model for distances, the *amortizing placement* algorithm is always within a constant factor of the optimal. Although this earlier proof only guarantees that the amortizing placement algorithm is within a factor of 14 of the optimal, in this article, we find that it is within 5 percent 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, amortizing placement is a practical algorithm that can be expected to provide near-optimal performance. Second, for large-scale studies of cache coordination, amortizing 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 amortizing 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 greedy-dual algorithm works well for stand-alone caches. Our contribution is to examine the performance of this algorithm 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-greedy-dual* replacement algorithm yields further improvements over the greedy-dual 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 article 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. 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 Fig. 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 $\mathcal{C}$ is a child of cluster $\mathcal{C}'$ if $\mathcal{C}$ is immediately contained within $\mathcal{C}'$.

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 of these nodes. This model is the same as the ultrametric model used by Karger et al. in [12].

2.1 Noncooperative Local Algorithms

In this section, 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 access 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. The distinction between this strategy and the MFU placement strategy is that, while the MFU placement strategy picks a set of objects to place in the cache and

TABLE 1
Notation for the Placement Algorithms

Input:

- Set of caches and the cache sizes.
- Set of objects.
- Access frequencies from each cache to each object.
  - Let $f(u, \psi)$ denote the frequency from node u to object ψ.
  - Let $f(\mathcal{C}, \psi) = \sum_{u \in \mathcal{C}} f(u, \psi)$, denote the aggregate frequency from cluster $\mathcal{C}$ to object ψ.
- Cluster tree T with diameters for each cluster.
  - Let $diam(\mathcal{C})$ denote the diameter for cluster $\mathcal{C}$.
  - Let $p(\mathcal{C})$ denote the parent of cluster $\mathcal{C}$ in T.
- A maximum cost called *penalty* which must be paid if no cache has the object.
  - Define $diam(p(root))$ to be *penalty*.

Output:

- A placement of objects among the caches.
  - Represented as a set of items, where each item is a triple of the form $(objectId, cacheId, benefit)$.

leaves them undisturbed throughout the period, the LFU replacement strategy can change the contents of the cache after each request. To illustrate, consider the following scenario involving two objects ψ and ψ' that are accessed alternately and a single cache that can hold just one object at any time. In such a scenario, the MFU placement strategy would pick one of the two objects and leave it in the cache throughout, thus incurring misses only half the time. The LFU replacement strategy, on the other hand, would alter the contents of the cache after each request, thus incurring misses all the time.

Greedy-dual replacement. This is a generalization of the LRU algorithm to the case where each object has a different fetch cost [2], [24]. The motivation behind the greedy-dual algorithm is that the objects with larger fetch costs 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, not the absolute ones, that matter [2]. In an efficient implementation, we use an additional variable—called *threshold*—to track the value of the object that was last removed from the cache. (The variable threshold is initially set to zero.) Upon a cache miss, the minimum valued object is evicted from the cache, the variable threshold is set to the value of this object, and no other values are modified. However, when an object is touched or added, its value is set to its fetch cost plus the threshold.

2.2 Cooperative Placement Algorithms

A *placement* assigns objects to caches without violating the cache size constraints. The cost of a placement is defined in the natural manner: the sum over all nodes u and all objects ψ of the access frequency for object ψ 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 be low 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 and 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. Table 1 presents the notation used for describing the placement algorithms.

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. The instance of the minimum-cost flow problem constructed by this reduction has $\Theta(nm)$ vertices, where n is the number of nodes and m is the number of objects in the system.

Since the minimum cost flow problem is computationally intensive, this optimal algorithm incurs a high running time complexity. In particular, even the fastest known algorithm for minimum cost flow takes at least quadratic time and, hence, the running time of this optimal algorithm is at least quadratic in the product of n and m. 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 $\mathcal{C}_1, \ldots \mathcal{C}_k$ which constitute a larger cluster $\mathcal{C}$. While computing the placement for cluster $\mathcal{C}_i$, the algorithm uses the access frequency information from within that cluster only. Now, at cluster $\mathcal{C}$, we first merge the tentative placements computed for subclusters $\mathcal{C}_1$ through $\mathcal{C}_k$. The placement Q obtained by this merging is clearly a starting placement for cluster $\mathcal{C}$, but it may be improved using the information about the aggregate frequencies across different subclusters in $\mathcal{C}$.

For example, there may be an object ψ that is not chosen in any of the clusters $\mathcal{C}_1$ through $\mathcal{C}_k$ since its access frequency within each cluster is small. But, its aggregate frequency in the larger cluster $\mathcal{C}$ may be large enough that the placement Q can be improved by taking a copy of object ψ and dropping a less beneficial item—for example, an item which is replicated many times or whose aggregate access frequency within the cluster C is not as high. To determine such useful swaps, we calculate a *benefit* for each item in Q and a *value* for each object not in Q. (Such objects are said to be Q-$missing$.)

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

The benefit of an item x in placement Q, 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 $\mathcal{C}_i$ calculates a local benefit for each item in its placement. After the merge step, the parent cluster $\mathcal{C}$ updates the benefits as follows: For each object that has one or more copies in Q, 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 $\mathcal{C}$'s aggregate access frequency to the object times the cost of leaving the cluster $\mathcal{C}$. The intuition is that, among all the copies of an object, the primary copy will be the last one to be removed from Q 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 Q. While there is a Q-missing object ψ whose value is more than the least beneficial item x in Q, we remove x from Q and substitute a copy of ψ. This phase terminates when the benefit of each item in Q is higher than the value of each Q-missing object.

This swapping phase concludes the computation for cluster $\mathcal{C}$ and the algorithm proceeds to the parent cluster of $\mathcal{C}$ iteratively.

In fact, the presentation of this algorithm in [13] involves two passes through the network locality tree: a bottom-up pass that computes a *pseudoplacement* and a top-down pass that refines this pseudoplacement to a placement. However, we note that the notion of pseudoplacement was essential only for proving the performance guarantees but not for correctness. Hence, here we gave an equivalent one-pass description avoiding the notion of pseudoplacement and highlighting the ease of implementation. Table 2 summarizes this simpler one-pass description by giving separate pseudocodes for the leaf computations and the internal node computations.

Although 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 Amortizing Placement Algorithm

The worst-case analysis leads us to a natural variant of the greedy algorithm called the amortizing algorithm. The high level intuition underlying this algorithm is as follows: When deciding which copies to exchange at a given stage, it can be difficult to decide whether to swap out a secondary copy with high benefit in favor of a primary copy (of a missing object) with low benefit. In such a case, the greedy algorithm prefers to keep the secondary copy because it has higher benefit, but this approach can fail because it waits too long to swap in missing objects. An alternative strategy is to always prefer primary copies over secondary copies, but it is easy to devise scenarios in which this strategy fails. (Note that such a strategy can be misled by introducing a large number of objects with access frequencies tending to zero.) The amortizing algorithm pursues a more balanced strategy: It uses the miss cost incurred at the current level to "justify" the removal of certain secondaries. The underlying intuition is that, since we have already committed to paying the miss cost, we can afford to incur a similar cost in order to make room for additional primary copies to be swapped in. This improved algorithm is summarized in Table 3.

TABLE 2
The Greedy Placement Algorithm

At a leaf u

- If the size of cache at u is k, pick the k most frequently used objects (locally) at u. For each such object ψ, set the benefit b to $f(u,\psi)\cdot(diam(p(u))-diam(u))$, and add the item (ψ,u,b) to the local placement Q.

At a cluster $\mathcal{C}$

1. **Merge.** Let Q_i be the placement computed for the ith child cluster of $\mathcal{C}$. (We assume that computations at the children of $\mathcal{C}$, if any, were already completed.) Initialize the placement Q to be the union of these Q_i's.
2. **Adjust benefits.** For each object ψ that has a copy in Q, pick its highest-benefit copy (breaking ties arbitrarily), and designate it as the *primary* copy for ψ. All other copies of ψ are *secondary*. Increase the benefit of the primary copy of ψ by $f(\mathcal{C},\psi)\cdot(diam(p(\mathcal{C}))-diam(\mathcal{C}))$. The benefits of the secondary copies are not changed.
3. **Value missing objects.** For each Q-missing object ψ', set $value(\psi')$ to $f(\mathcal{C},\psi')\cdot(diam(p(\mathcal{C}))-diam(\mathcal{C}))$. Let X be the set of all Q-missing objects.
4. **Greedy swap.** Pick the highest valued object ψ' from X, and the least benefit item y from Q. If $value(\psi')$ is greater than the $benefit(y)$ then swap ψ' for y and repeat step 4. The swap step involves removing the item y from Q, adding the item $(\psi', cacheId(y), value(\psi'))$ to Q, and then removing ψ' from X.
5. **Wind up.** If $\mathcal{C}$ is the root cluster, then return the placement Q. Otherwise proceed to the parent cluster of $\mathcal{C}$.

It is proven in [13] that the above amortizing 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 than this constant factor suggests.

2.2.4 A Note on Practical Implementation

A potential concern about the greedy and the amortizing algorithms is that the computation at a generic cluster $\mathcal{C}$ involves passing a potentially large amount of information—namely, the placement records for the placement Q computed for $\mathcal{C}$ as well as the access frequencies for each of the objects missing in Q—to the parent of $\mathcal{C}$. Of these, the former part is guaranteed to have no more than S elements, where S is the total cache space in the system; assuming that placement records are smaller than the objects themselves, the network cost of transmitting this information will be small compared to the cost of actually placing the data objects. The latter part of the information, however, can be unboundedly large in certain cases; a good example is the case where we have a large number of objects with access frequencies tending to zero.

In a practical implementation, we can reduce the amount of information passed by 1) ignoring objects whose access frequency is below a certain threshold, 2) restricting attention to only the S, or maybe a small multiple of S, most frequently accessed objects at each cluster, and 3) freezing the computation once a certain level of the hierarchy is reached, instead of going all the way to the root of the cluster tree. The thresholds for each of these three steps can be chosen depending on how much we are willing to compromise on the quality of the placement.

Another potential concern about the algorithms is scalability because—as described in Table 2 and Table 3—all the computation at the last level of recursion is performed at the single node corresponding to the root of the network-locality tree. This could induce a centralized bottleneck at this node. However, such bottlenecks can be avoided by distributing the computation of each internal node over all the leaves in its subtree. In fact, we remark that an efficient implementation of the greedy and amortizing algorithms that distributes the computation in such a load-balanced manner exists. For further details about such an efficient distributed implementation, we refer the reader to [13].

2.3 A Cooperative Replacement Algorithm

Our experiments in Section 3 show that, in a cooperative scenario, the greedy-dual algorithm performs much better than the other local replacement algorithms. This is because, even though the greedy-dual 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

TABLE 3
The Amortizing Placement Algorithm

At a leaf u

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

At a cluster $\mathcal{C}$

1. **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.
2. **Adjust benefits.** Same as in the greedy algorithm.
3. **Value missing objects.** Same as in the greedy algorithm, except that we also initialize Δ to the sum of the values of all the Q-missing objects.
4. **Amortized swap.** Similar to the greedy swap, except that the potential ϕ is used to reduce the benefits of certain secondary items.
   (a) Pick the highest valued (Q-missing) object ψ' from X, and the least benefit primary and secondary items (y_p and z_s respectively) from Q.
   (b) If $value(\psi') > min(benefit(y_p), benefit(z_s) - \phi)$, then depending on which of the latter two terms is smaller perform one of the following two swap operations and then repeat step (4).
   - If $benefit(y_p) < benefit(z_s) - \phi$, then swap ψ' for y_p and adjust Δ. The swap step involves removing the item y_p from Q, adding the item $(\psi', cacheId(y_p), value(\psi'))$ to Q, and then removing ψ' from X. The variable Δ is set to $\Delta - value(\psi') + benefit(y_p)$.
   - Otherwise, swap ψ' for z_s, reduce the potential ϕ, and adjust variable Δ. As before, the swap step involves removing the item z_s from Q, adding the item $(\psi', cacheId(z_s), value(\psi'))$ to Q, and then removing ψ' from X. The potential ϕ is set to $\max(0, \phi - benefit(z_s))$ while the variable Δ is set to $\Delta - value(\psi')$.
5. **Update potential.** Add Δ to ϕ.
6. **Wind up.** If $\mathcal{C}$ is the root cluster, then return the placement Q. Otherwise proceed to the parent cluster of $\mathcal{C}$.

benefits of cooperation. For example, the idle caches are not exploited by nearby busy caches.

Hence, we devise *hierarchical-greedy-dual*, a cooperative replacement algorithm that not only preserves the implicit coordination offered by greedy-dual, but also enables busy caches to utilize nearby idle caches.

In this algorithm, each individual cache runs the local greedy-dual algorithm using the efficient implementation described in Section 2.1. Recall that the local greedy-dual 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 $\mathcal{C}$ receives an evicted object ψ from one of its child clusters, it first checks to see if there is any other copy of ψ among its caches. If not, it picks the minimum valued object ψ' among all the objects cached in $\mathcal{C}$. Then, the following simple admission control test is used to determine if ψ should replace ψ'. If the copy of ψ was used more recently than the copy of ψ', then ψ replaces ψ' and the new evicted object ψ' is recursively passed on to the parent cluster of $\mathcal{C}$. Otherwise, the object ψ itself is recursively passed on to the parent cluster of $\mathcal{C}$.

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 typically have a high fetch cost since no other (nearby) cache stores 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 greedy-dual 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, the 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 ψ'.

TABLE 4
Default System Parameters

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 only)
m	No. of local objects per node	25 (synthetic only)
r	Sharing parameter	0.75 (synthetic only)
PAT	Access pattern	Uniform (synthetic only)
I	Idle cache factor	1

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. For the sake of brevity and for ease of reference, we will use the phrase *GreedyPlace* as shorthand for the *greedy placement algorithm* throughout the rest of this paper. Similarly, we will use the phrases *AmortPlace*, *MFUPlace*, *GreedyDual*, and *HrcGreedyDual* as shorthand for the amortizing placement, the MFU placement, the greedy-dual, and the hierarchical-greedy-dual algorithms, respectively.

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 MFUPlace, GreedyPlace, AmortPlace, 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 4 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.

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 [23]. 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.

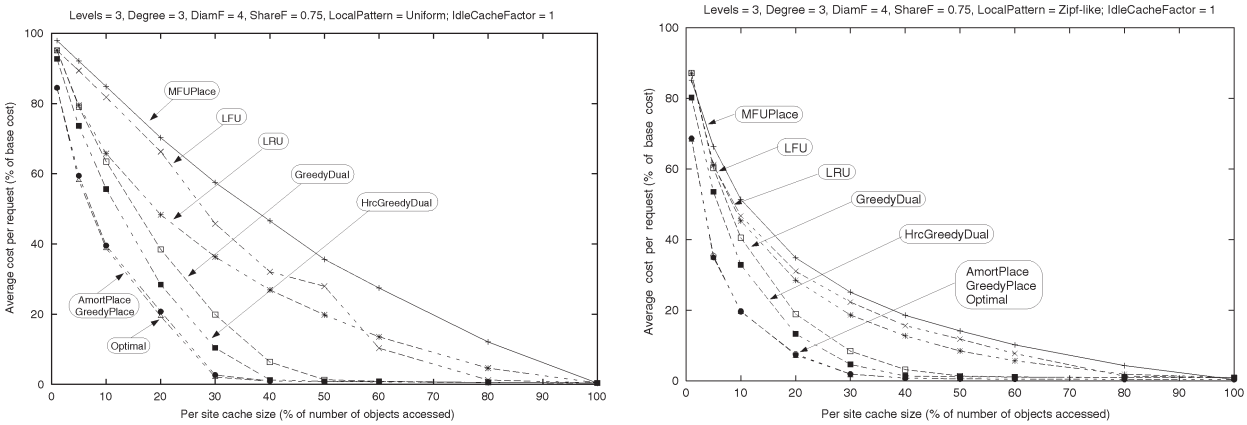
Within each cluster, we select objects according to a pattern PAT that is either "Zipf-like" or "Uniform." Thus, for a particular cache v and a particular object ψ that is local to cluster $\mathcal{C}$ and that is the kth-ranked object of the m objects local to $\mathcal{C}$, the fraction of node v's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

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

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.

It is natural to ask whether this model is equivalent to having I individual idle caches in each level-one cluster. The answer is yes, because, by assumption, the idle caches make no access requests at all and, hence, their only relevance to the rest of the system is 1) how much cache space they have and 2) how far they are from each of the nonidle caches. Given our clustering-based model for



(a) (b)

Fig. 2. Varying cache size. (a) With $PAT = "Uniform"$ and (b) with $PAT = "Zipf\text{-}like"$.

network distances (see Fig. 1), it turns out that both of these factors remain exactly the same whether we have I idle caches each of size one in each level one hierarchy or whether we have a single idle cache of size I in each level one hierarchy. Hence, both of these models are equivalent.

3.2 Results

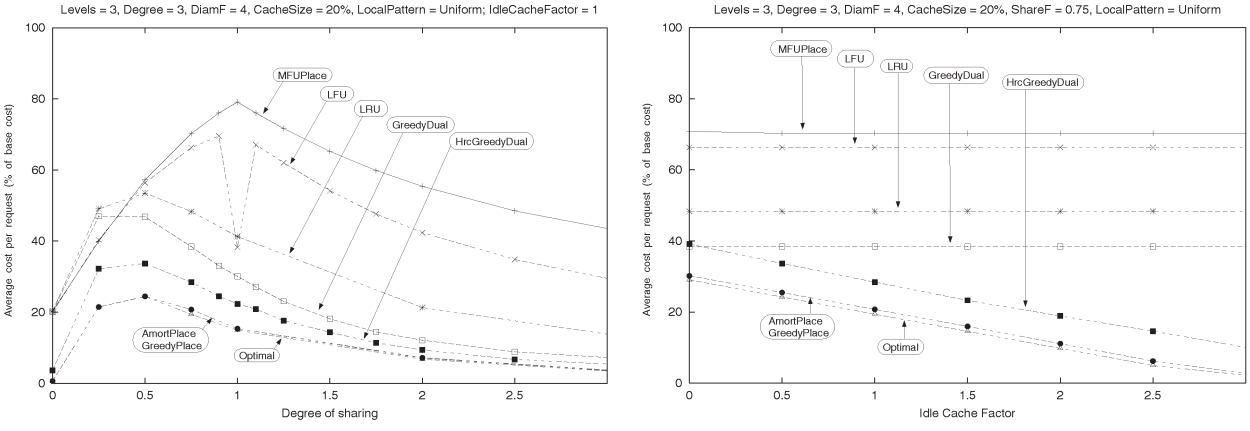
Fig. 2 plots the performance of the algorithms as the cache size percentage C is varied from 1 to 100, with other parameters set to their default values.

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 Fig. 2b.

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, and GreedyDual), cooperative replacement (HrcGreedyDual), and cooperative placement (AmortPlace, GreedyPlace, and 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 performs 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 (Fig. 3b). However, when there is idle cache space to exploit, HrcGreedyDual outperforms GreedyDual, as Fig. 2b 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



(a) (b)

Fig. 3. Varying degree of sharing and idle cache factor. (a) Varying degree of sharing and (b) varying idle cache per level-one cluster.

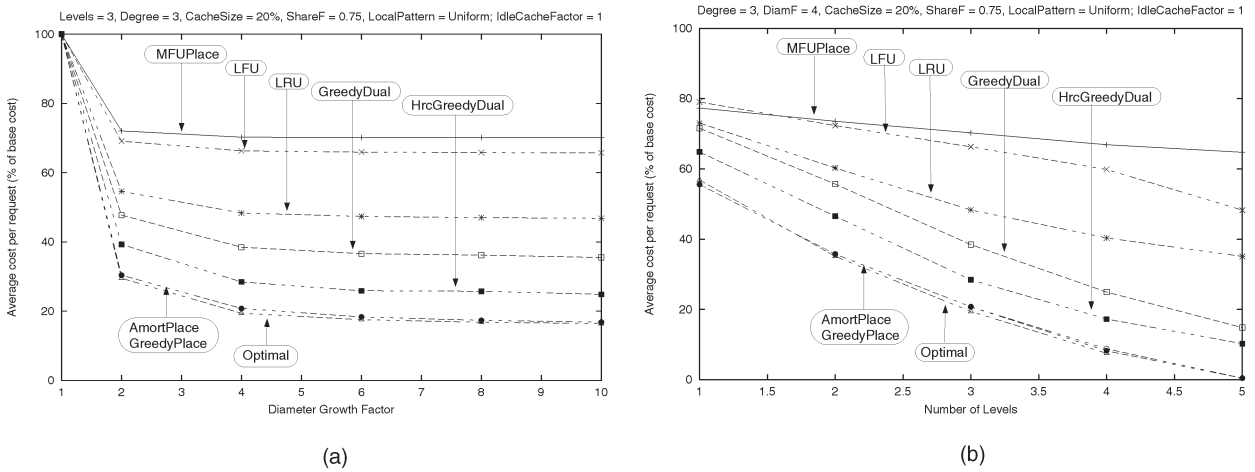


Fig. 4. Varying diameter growth factor and number of levels. (a) Diameter growth factor and (b) number of levels.

caches, the coordinated algorithms significantly outperform traditional replacement algorithms.

We also note that the amortizing and the greedy placement algorithms effectively match optimal across a wide range of workloads.

3.2.1 Sensitivity to Other Parameters

Fig. 3a 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.

Fig. 3b 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.

Fig. 4a shows the impact of varying the diameter growth factor, λ. For small λ, all algorithms have approximately the same performance since all cache hits and misses have more or less the same cost. For large values of λ, performance is dominated by the number of misses and, hence, 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 and, 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.

Fig. 4b shows the performance as the number of levels in the hierarchy, L, is varied. For our workloads, increasing L increases the total cache space in the system more quickly than it increases the total number of objects. (To be precise, note that 1) the number of leaves in the tree, call it n, is D^L; 2) the 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 also reduces.

Finally, Fig. 5 shows the performance as the degree of each internal node, D, is increased. Again, for our synthetic workloads, increasing D increases the total number of nodes in the system. However, the ratio of the total number of objects to the total available cache space (which is $\frac{2nm}{nC(L+1)m/100}$) does not change even as D changes. Therefore, the fraction of accesses that have to pay the base cost is

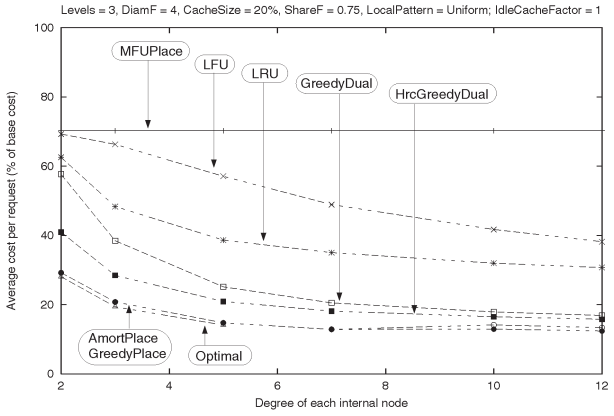


Fig. 5. Varying the degree of each node.

almost fixed and, hence, there is not much variation in the average access cost when it is expressed as a percentage of the base cost.

On the whole, the basic trends in all of these experiments are similar to those described earlier in Section 3.2: Algorithms with more coordination have a better performance and, moreover, the greedy and the amortizing placement algorithms closely match the optimal in their performance.

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.

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 29 August 1996 to 22 September 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, etc.).

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 (i.e., the leaves of the network-locality tree), 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. First, random mapping of clients to nodes will eliminate any similarity of interests among nearby clients. Second, with about 600 random nodes multiplexed to each node, 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 default 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 [23].) 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 benchmark for the best any prediction technique can achieve. The second one is a naive and simple 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 preceeding 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

Fig. 6a shows the performance of the 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

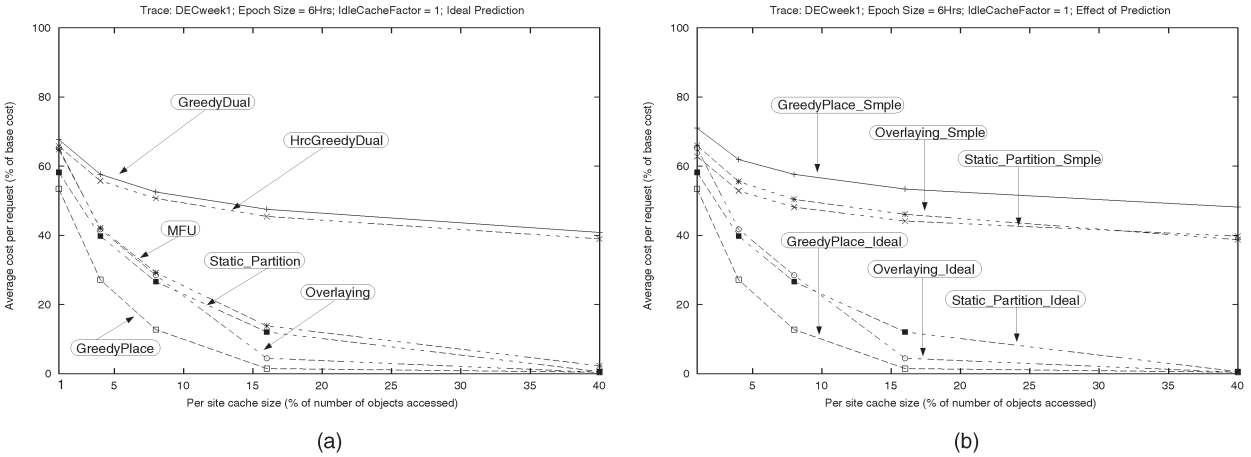


Fig. 6. DEC proxy trace: Varying cahce size. (a) With ideal prediction and (b) effect of prediction.

of access patterns (e.g., subscription-based systems). The effect of increasing coordination can also be seen: The hierarchical-greedy-dual outperforms the greedy-dual 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.

Fig. 6b shows the effect of prediction on the performance of the placement algorithms by comparing these algorithms with simple prediction at one extreme and ideal prediction at the other. With simple 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 under the simple predictor.

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 simple predictor uses only one type of locality, namely the temporal locality, out of the three localities expected in web accesses.

Finally, Fig. 7 shows the effect of varying epoch size on the performance of the algorithms. As expected, performance is excellent when the epochs are short. As epochs become longer, the placement becomes coarse grained and the performance gains are more modest. However, even with only daily reorganization, significant performance gains appear possible.

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 greedy-dual, at each cache. The greedy-dual local replacement algorithm was evaluated in [2], [11], [24], 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. The contributing factors are that the number of users as well as the number of objects are larger by orders of magnitude and that the network distances between one pair of nodes can be much different from that between another pair.

Recently, Yu and MacNair [25] studied the question of wide-area cache coordination, but under a 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 nonuniform 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 toward the clients before they ask for it, thus avoiding the compulsory misses. Such schemes involve two orthogonal components: predicting future access patterns and distributing the data

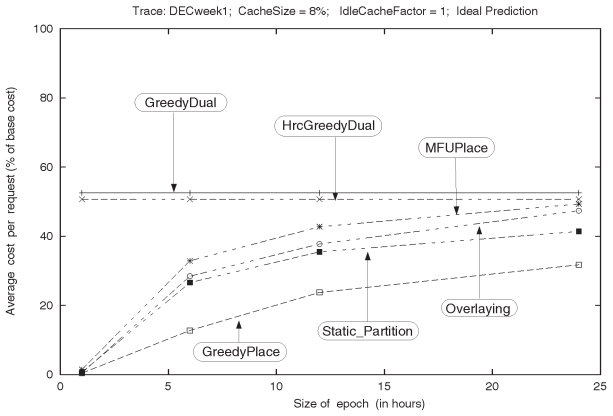


Fig. 7. DEX proxy trace: Varying epoch size.

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 the placement problem addresses 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 especially when the sizes of the individual caches are *small*.

Throughout this paper, we assumed that all objects are of the same size. It is worth noting that there are applications in which it is reasonable to reduce the case of variable-sized objects to the uniform-sized case by splitting each object of size k into k unit-size fragments and then placing the fragments independently. In other applications, such a reduction may not be apprpriate; thus, it remains an interesting open problem to extend our results to the case of variable-sized indivisible objects. Another interesting question is to extend the algorithms described here to cope with the fact that the network distances may vary dynamically depending on the load on the network.

The placement algorithms are excellent for applications such as subscription-based systems where we have a good knowledge of the future access patterns. For applications where we do not have the luxury of such knowledge, the hope is to predict the future access patterns based on the past access patterns. It is a bit troubling that our simple temporal-locality-based predictor does not do as well, but we believe that a more sophisticated predictor that exploits all three kinds of localities (namely geographic, temporal, and spatial) and other factors, such as time-of-day variances, would be able to achieve a significant portion of the gap between the ideal and the simple cases. Needless to say, this remains an interesting and promising open problem.

Acknowledgments

This work was supported in part by a US National Science Foundation CISE grant (CDA-9624082) and grants from IBM, Intel, Novell, Sun, tivoli, and the Texas Advanced Technology Project. Dahlin was also supported by an NSF CAREER grant (9733842) and an Alfred P. Sloan Research Fellowship. A preliminary version of this paper appears in the Proceedings of the 1999 IEEE Workshop on Internet Applications, pages 62-71, July, 1999. This work was done while Madhukar Korupolu was at the Department of Computer Science, University of Texas at Austin, Austin.

References

[1] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz, "The Harvest Information Discovery and Access System," *Proc. Second Int'l World Wide Web Conf.*, pp. 763-771, Oct. 1994.

[2] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms, *Proc. 1997 USENIX Symp. Internet Technology and Systems*, pp. 193-206, Dec. 1997.

[3] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell, "A Hierarchical Internet Object Cache," *Proc. USENIX Technical Conf.*, pp. 22-26, Jan. 1996.

[4] Digital Equipment Corp., "Web Proxy Traces," ftp://ftp.digital.com/pub/DEC/traces/proxy.webtraces.html. Sept. 1996.

[5] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," *Proc. First Symp. Operating Systems Design and Implementation*, pp. 267-280, Nov. 1994.

[6] L. Fan, P. Cao, J. Almeida, and A.Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *Proc. 1998 ACM SIGCOMM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm.*, pp. 254-265, Aug. 1998.

[7] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath, "Implementing Global Memory Management in a Workstation Cluster," *Proc. 15th ACM Symp. Operating Systems Principles*, Dec. 1995.

[8] S. Gadde, M. Rabinovich, and J. Chase, "Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches," *Proc. Workshop Hot Topics in Operating Systems*, http://www.cs.duke.edu/ari/cisi/crisp-recycle.ps, May 1997.

[9] J.S. Gwertzman and M. Seltzer, "The Case for Geographical Push-Caching," *Proc. Fifth Workshop Hot Topics in Operating Systems*, pp. 51-57, May 1995.

[10] A. Heddaya and S. Mirdad, "WebWave: Globally Load Balanced Fully Distributed Caching of Hot Published Documents," *Proc. 17th Int'l Conf. Distributed Computing Systems*, pp. 160-168, May 1997.

[11] S. Irani, "Page Replacement with Multisize Pages and Applications to Web Caching," *Proc. 29th Ann. ACM Symp. Theory of Computing*, pp. 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," *Proc. 29th Ann. ACM Symp. Theory of Computing*, pp. 654-663, May 1997.

[13] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman, "Placement Algorithms for Hierarchical Cooperative Caching," *Proc. 10th Ann. ACM-SIAM Symp. Discrete Algorithms*, pp. 586-595, Jan. 1999. invited to appear in the special issue of *J. Algorithms* devoted to selected papers from SODA, 1999.

[14] A. Leff, J.L. Wolf, P.S. Yu, "Replication Algorithms in a Remote Caching Architecture," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 11, 1185-1204, Nov. 1993.

[15] C.G. Plaxton and R. Rajaraman, "Fast Fault-Tolerant Concurrent Access to Shared Objects," *Proc. 37th Ann. IEEE Symp. Foundations of Computer Science*, pp. 570-579, Oct. 1996.

[16] M. Rabinovich, I. Rabinovich, and R. Rajaraman, "Dynamic Replication on the Internet," Technical Report HA6177000-980305-01-TM, AT&T Labs-Research, Apr. 1998.

[17] L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache," Technical Report RN-98-13, Dept. of Computer Science, Univ. College, London, 1998.

[18] A. Rousskov and D. Wessels, "Cache Digests," *Proc. Third Int'l Web Caching Workshop*, http://wwwcache.ja.net/events/workshop/, June 1998.

[19] R. Tewari, M. Dahlin, H.M. Vin, and J.S. Kay, "Design Considerations for Distributed Caching on the Internet," *Proc. 19th Int'l Conf. Distributed Computing Systems*, June 1999.

[20] M. van Steen, F.J. Hauck, and A.S. Tanenbaum, "A Model for Worldwide Tracking of Distributed Objects," *Proc. 1996 Conf. Telecomm. Information Networking Architecture (TINA '96)*, pp. 203-212, Sept. 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," *Proc. ACM-SIGCOMM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm.*, Aug. 1996.

[22] O. Wolfson, S. Jajodia, and Y. Huang, "An Adaptive Data Replication Algorithm," *ACM Trans. Database Systems*, vol. 22, no. 4, pp. 255-314, 1997.

[23] A. Wolman, G.M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H.M. Levy, "On the Scale and Performance of Cooperative Web Proxy Caching," *Proc. 17th ACM Symp. Operating Systems Principles,* pp. 16-31, 1999.
[24] N.E. Young, "On-Line File Caching," *Proc. Ninth Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 82-86, Jan. 1998.
[25] P.S. Yu and E.A. MacNair, "Performance Study of a Collaborative Method for Hierarchical Caching in Proxy Servers," *Proc. Seventh Int'l World Wide Web Conf,* Apr. 1998.
[26] L. Zhang, S. Floyd, and V. Jacobson, "Adaptive Web Caching," *Proc. NLANR Web Cache Workshop,* June 1997.

Madhukar R. Korupolu received the PhD degree in computer science from the University of Texas at Austin in 1999. Prior to that, he received the BTech degree in computer science from the Indian Institute of Technology, Madras, in 1994 and the MS degree in computer science from the University of Texas at Austin in 1996. His primary research interests are in the design and analysis of algorithms and, particularly, in their novel applications to problems arising in the contexts of Internet and wireless technologies. Currently, he is a research scientist at Akamai Technologies Inc. in Cambridge, Massachusetts.

Mike Dahlin received the BS degree in electrical engineering from Rice University in 1991 and the MS and PhD degrees in computer science fromt the University of California at Berkeley in 1993 and 1995. He is an assistant professor and faculty fellow in the Department of Computer Sciences at the University of Texas at Austin as well as an Alfred P. Sloan research fellow. Dr. Dahlin's primary research interests are in distributed I/O systems and operating systems. He is a member of the IEEE and the IEEE Computer Society.