Coordinated Placement and Replacement for Large-Scale Distributed Caches *

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

Abstract

In a large-scale information system such as a digital library or the web, a set of distributed caches can improve their effectiveness by coordinating their data placement decisions. 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 *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.

Keywords: 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.

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

*This work was supported in part by an NSF CISE grant (CDA-9624082) and grants from Intel, Novell, and Sun. Dahlin was also supported by an NSF CAREER grant (9733842). A preliminary version of this paper appears in the Proceedings of the 1999 IEEE Workshop on Internet Applications, pages 62–71, July, 1999.

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 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, greedy-dual [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 *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% 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.

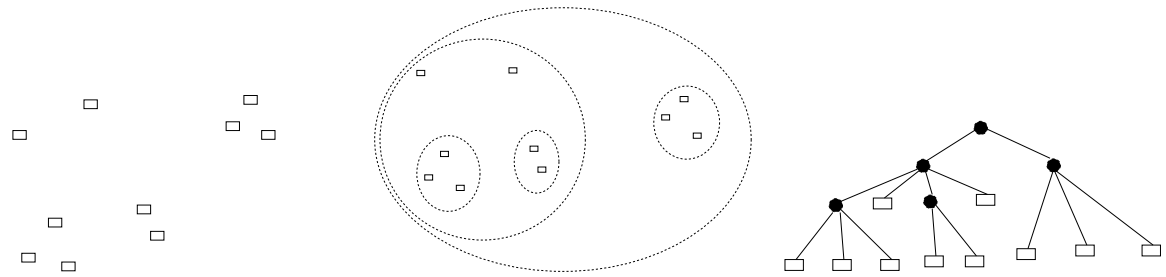
- A key challenge to coordinated placement algorithms is generating good predictions of access patterns based on past accesses.

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 Figure 1 which shows a set of cooperating nodes and a possible network-locality based clustering of these. This clustering is a natural consequence



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

Figure 1: *Model for network distances*

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 same as the ultrametric model used by Karger et al. in [12].

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 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.

Greedy-dual replacement. This is a generalization of the LRU algorithm to the case where each object has a different fetch cost [2, 23]. 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

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 $(objectId, cacheId, benefit)$.

Table 1: Notation for the placement algorithms.

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

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 $\mathcal{C}$.

Table 2: The greedy placement algorithm.

$\mathcal{C}_k$ 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 *pseudo-placement*, and a top-down pass that refines this pseudo-placement to a placement. However, we note that the notion of pseudo-placement 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 pseudo-placement 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.

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 y_s respectively) from Q.

   (b) If $value(\psi') > min(benefit(y_p), benefit(y_s) - \phi)$, 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 $benefit(y_p) < benefit(y_s) - \phi$, 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 y_s, reduce the potential ϕ, and adjust variable Δ. As before, the swap step involves removing the item y_s from Q, adding the item $(\psi', cacheId(y_s), value(\psi'))$ to Q, and then removing ψ' from X. The potential ϕ is set to $\max(0, \phi - benefit(y_s))$ 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 $\mathcal{C}$.

Table 3: The amortizing placement algorithm.

2.2.3. The amortizing 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, one at each level of the cluster tree. Though the benefit of the secondary copy is larger than the value of each such missing object, on the whole it might be much less than the sum of all these values put together.

To circumvent this problem, 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. Table 3 summarizes this improved algorithm.

It is proved 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.

Note that the above description of the greedy and the amortizing placement algorithms is still centralized. For example, 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 induces a centralized bottleneck at this node. However, such bottlenecks can be avoided by mapping the computation of each internal node to the leaves in its subtree in a load-balanced manner. For details about such a scalable distributed implementation, please see [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 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, 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'.

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, throughout the rest of this article, we use the phrase *GreedyPlace* (resp., *AmortPlace*, *MFUPlace*, *GreedyDual*, *HrcGreedyDual*) as a shorthand for the greedy placement (resp., amortizing placement, MFU placement, greedy-dual, hierarchical-greedy-dual) algorithm.

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 Hrc-GreedyDual, we begin with empty caches, and for each request we modify the cache contents as

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

Table 4: Default system parameters.

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. 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