Bandwidth Constrained Placement in a WAN

Arun Venkataramani Phoebe Weidmann Mike Dahlin
{arun, phoebe, dahlin}@cs.utexas.edu
Department of Computer Sciences
University of Texas at Austin

ABSTRACT

In this paper, we examine the *bandwidth-constrained placement problem*, focusing on trade-offs appropriate for wide area network (WAN) environments. The goal is to place copies of objects at a collection of distributed caches to minimize expected access times from distributed clients to those objects subject to a maximum bandwidth constraint at each cache. We develop a simple algorithm to generate a bandwidth-constrained placement by hierarchically refining an initial per-cache greedy placement. We prove that this hierarchical algorithm generates a placement whose expected access time is within a constant factor of the optimal placement's expected access time. We then proceed to extend this algorithm to compute close to optimal placement strategies for dynamic environments.

1. INTRODUCTION

This paper addresses the bandwidth constrained placement problem for Wide Area Networks (WANs). The goal is to place copies of objects at a collection of distributed caches to minimize expected access times from distributed clients to those objects. We assume a cooperative caching model [10] in which a cache miss at one location may be satisfied by another cache in the system. A number of scalable request-forwarding directory schemes have been developed to enable large-scale cooperative caching in WANs [1, 12, 26] and commercial *content distribution networks* of cooperating caches have been deployed by companies such such as Akamai and Digital Island.

Although traditional caches are filled when client *demand requests* miss locally and cause data to be fetched from a remote site, hit rates might be significantly improved by pushing objects to caches before clients request them [16, 27]. The *distributed cache placement problem* attempts to select which objects should be pushed to which caches in order to optimize performance. A number of researchers have examined the space-constrained placement algorithms – in which cache storage space places limits on what can be cached where – in local area networks [10, 13, 21] and wide area networks [20, 19]. However, little attention has been paid to bandwidth constrained placement.

Unfortunately, for WAN replication, bandwidth constraints may be more restrictive to replication than space constraints. A number of web-trace simulations have indicated that only modest cache sizes are needed to achieve maximum hit rates [11, 15, 26]. Furthermore, maintaining a pushed copy of an object in a cache consumes not only disk space but also network bandwidth to update that copy when the origin data changes. Gray and Shenoy [14] compare the dollar cost of transmitting an object across the Internet to the dollar cost of storing the object on disk and indicate that network costs will be greater than disk storage cost for objects whose lifetime is less than 13 months.

In this paper, we extend Korupolu et al.'s [20] space-constrained placement algorithm to address bandwidth constraints. The bandwidth-constrained problem differs from the simple space constrained model used by Korupolu et al. in several important ways:

- The bandwidth-constrained cost model depends on object update frequency and must account for the on-demand replication of objects that naturally occurs as demand-reads are processed. Conversely, the space-constrained cost model does not depend on object-update frequency, and a good space-constrained placement may not be improved by on-demand replication because any new replica must displace a previously placed object.

- In a bandwidth-constrained placement model, replicas of data initially present in caches should be left there, while such initial copies have no advantage over other objects under the space-constrained model.

- After a good bandwidth-constrained placement is calculated, it may take considerable time to send objects to their caches. In fact, in some cases bandwidth-constrained placement may be a continuous process as copies of objects are pushed to caches at some low background bandwidth, new objects are created, and existing objects are updated. Thus, it is important for a bandwidth-constrained placement algorithm to work well in a dynamically changing environment and provide good performance at intermediate points in its execution. Conversely, Korupolu et al.'s algorithm only seeks to optimize the cost of the final, complete placement and runs in a batch mode as opposed to an incremental one.

This work was supported in part by the Texas Advanced Technology Program. Dahlin was also supported by an NSF CAREER award (CCR-9733842) and an Alfred P. Sloan Research Fellowship.

We extend Korupolu et al.'s algorithm and show that our *Fixed-t_{fill}* algorithm generates a final placement that is within a constant factor of the cost of a bandwidth-constrained optimal placement under empty-cache initial conditions. We then show that our *InitFill* algorithm also provides a final placement that is within a constant factor of the optimal even when considering an initial placement. Finally, we show that our *DoublingEpoch* algorithm generates series of placements that is continuously within a constant factor of the optimal placement at any time t, but that to do so, it expands the required bandwidth by at most a factor of 4. This means that the *DoublingEpoch* algorithm is well-suited to a dynamic environment where objects are updated, demand-read copies appear, and new objects appear because it requires no *a priori* estimate of what point in time for which to optimize the placement schedule.

Our placement algorithms have several limitations. First, the constant cost and bandwidth-expansion factors that bound our worst case performance appear large (about a factor of 14 for the cost bound and 4 for the bandwidth expansion bound). However, previous experimental evaluation of the space-constrained placement algorithm indicates that for practical topologies and workloads, its behavior closely approximates the ideal algorithm despite similarly large constants [19]; our intuition leads us to expect similar behavior in the bandwidth constrained case. Second, although the *DoublingEpoch* algorithm is robust in a dynamic environment and provides a constant-factor approximation of the optimal bandwidth-expanded placement *within* any interval across which it is run, it must be restarted when system conditions significantly change, and it is not provably near-optimal *across* a series of executions. Finally, all of our constant-factor bounds assume uniform unit-sized objects. The nonuniform-size placement problem is NP-hard. We present a simple heuristic that extends our algorithms to accommodate variable-sized objects. Future work is needed to experimentally validate our conjectures that for practical topologies and workloads our *Fixed-t_{fill}*, *InitFill*, and *DoublingEpoch* algorithms will be nearly optimal within any interval, that our heuristics for chaining the *DoublingEpoch* algorithm across intervals work well, and that our heuristics for variable-sized objects work well.

The rest of this paper is organized as follows: In section 3, we describe the system architecture and the hierarchical distance and cost model we are considering. In section 4, we introduce the basic *Fixed-t_{fill}* algorithm that is a straightforward extension of the Greedy algorithm in [20] and prove that its cost is within a constant factor of the optimal. In section 5, we remove the restrictions assumed while developing the *Fixed-t_{fill}* algorithm and extend it to relate to more realistic scenarios. Herein, we introduce the Doubling Epoch algorithm and and analyze it for performance bounds. In section 6, we explain a heuristic algorithm to handle variable sized objects, and finally in section 7, we summarize our conclusions. The appendix outlines the proof of optimality to within a constant factor of the *Fixed-t_{fill}* algorithm. We also show that that the placement problem for variable sized objects reduces to the partition problem and is hence intractable and unyielding to efficient approximation algorithms.

2. RELATED WORK

The placement problem has traditionally been treated as one constrained by space. Korupolu *et al.* [20] studied the problem of coordinated placement for hierarchical caches with space constraints, *i.e.* fixed cache sizes. They proved that under the hierarchical model of distances, the space constrained *Amortized Placement* algorithm is always within a constant factor (about 13.93) of the optimal. Though for practical purposes this factor is rather large, the experimental work in [19] suggests that this algorithm yields an excellent approximation of the optimal for a wide range of workloads. In addition, the simplified greedy version of the amortized algorithm introduced in [20] has also been shown to provide an excellent approximation of the optimal, though in theory its performance can be arbitrarily far from optimal.

In an earlier study, Awerbuch, Bartal and Fiat [2] provide a polylog(n)-competitive on-line algorithm for the general space constrained placement problem under the assumption that the size of each cache in the on-line algorithm is polylog(n) times larger than the size in the optimal algorithm. The placement problem for a network of workstations modeled as a single level hierarchy has been studied by Leff, Wolf and Yu [21]. They provide heuristics for a distributed implementation of their solution. However, the heuristics make use of particular properties of single-level hierarchies and are not applicable to arbitrary hierarchies.

Replacement algorithms attempt to solve the problem of determining which object(s) are to be evicted when a cache miss occurs. Relevant studies of replacement algorithms have been done in [6, 30]. In the space constrained scenario, replacement algorithms may also be viewed as placement algorithms starting with an empty placement. The work in [19] shows that the hierarchical version of the *Greedy-Dual* replacement algorithm exhibits good cooperation and performs well in practice. However, for the bandwidth constrained placement problem, objects can be allowed to remain in the caches until they are modified, and hence replacement policy is not an issue.

The object placement problem has an orthogonal counterpart, namely the *object location* problem. The object location problem has been widely studied [5, 8, 25]. Recent studies such as Summary Cache [12], Cache Digest [24], Hint Cache [26], CRISP [23] and Adaptive Web Caching [31] generalize from hierarchies to more powerful cache-to-cache cooperation scenarios. Some recent studies have combined the location problem with the placement problem by remembering routing information at intermediate nodes [17, 22, 23, 29]. However, in this paper, we do not deal with lower level routing issues and separately consider only the placement problem.

To the best of our knowledge, the placement problem with bandwidth constraints has not been studied for hierarchical caching networks.

3. SYSTEM ARCHITECTURE

The system architecture is modeled as a set of N distributed machines and a set S of origin servers connected by a network. Assume that these machines are accessing a set of M shared objects maintained at any of the servers in S and cached at the machines. For each machine i, there is a fixed available bandwidth denoted by $bw(i)$ to push objects into the cache at machine i. The size of the cache at every machine is assumed to be very large. The cost of communication between any pair of machines i and j is given by the function $dist(i,j)$.

Requests for objects are made by clients at or near these machines. If there exists a local copy of a requested object α at machine i, then the object is served locally. If not, a directory (*e.g.* a summary [12] or a hint cache [26]) is consulted to find the nearest copy of object α, which is fetched and served to the requester as well as stored locally at machine i. This implies that all future requests for object α at machine i will be local hits. Thus the cost of satisfying an access request for an object α at a machine i, denoted by $c(i,\alpha)$ is given by the cost of communication $dist(i,j)$ between i and j, where j is the closest machine that possesses a copy of object α. If no copy of object α resides on any of the caches in the network, then $c(i,\alpha)$ is defined to take a value Δ that denotes the *miss penalty.* In other words this is the cost of obtaining a requested object directly from an origin server rather than from one of the N cooperating caches. Note that Δ must at least be as large as the maximum value of the function *dist*.

An example of a system with such properties is a large scale content distribution network where bandwidth costs dominate storage costs and where the caches have a limited available bandwidth to receive object updates.

Hierarchical distance model

To make this problem tractable and applicable for practical distributed networks, we structure the distance function $dist(i,j)$ according to a hierarchical model. This hierarchy may be understood as a multilevel cluster or a *cluster tree* as shown in Fig 1. Such an organization of machines naturally reflects the way wide area network topologies are structured. For example in a collection of universities each machine belongs to the department cluster, which in turn belongs to the university cluster, and so on.

A *depth-0 hierarchy* is a singleton set containing one machine. A *depth-d hierarchy H* is a virtual node H plus a set of hierarchies $H_1, H_2, \ldots, H_k$, each of depth smaller than d, and at least one of which has depth exactly $d-1$. The hierarchies $H_1, \ldots, H_k$ are referred to as the children of H and H itself as the parent of $H_1, \ldots, H_k$. A hierarchy is associated with a diameter function *diam* which is defined as follows: If H is a singleton hierarchy, $diam(H) = 0$, else $diam(H) \geq \lambda diam(H_i)$, for all children H_i of H. (Such a hierarchy is said to be $\lambda - separated.$) For any hierarchy H, $machines(i)$ is defined to be the set of singleton hierarchies that are descendents of H. The distance function $dist(i,j)$ between any two machines i and j belonging to H is defined as $diam(H_1)$ where H_1 is the least depth hierarchy such that i and j belong to $machines(H_1)$.

Thus, a hierarchy is essentially a tree and we use the two terms interchangeably. It must be remembered that while understanding a hierarchy as a cluster tree, the caches/machines are only at the leaf nodes. The intermediate nodes are virtual and only maintain book-keeping information. The distance function described above ignores small variations in cost inside a subtree and models a system where each machine has a set of nearby neighbors, all at approximately the same distance d_1, and then a set of next closest neighbors all at approximately a distance d_2 and so on. Note that the term *hierarchical caches* used to describe caching systems like Harvest [5], Squid [28] etc. is different and the two models must not be confused. In the latter, the hierarchy models the actual network topology and interior nodes act as caches. In our model the hierarchy itself is a logical tree introduced solely to capture network distances.

The model described above is the same as the ultrametric model used by Karger et al. [18] Such a hierarchical structure can be used to capture a wide variety of distributed networks like intranets or WANs. A local area network for example is a *depth-1* hierarchy. It can also be used to model content distribution networks - the leaf caches are the geographically distributed pseudo-servers. A client request originating at an ISP is directed to one of these caches that may either serve the object or redirect the request to the closest neighboring cache which stores a copy. If none of the caches store the object, then the request is directed to the origin server for the object. It seems reasonable to think that a hierarchal distance model may serve as a reasonable approximation of a complex network such as the Internet. In fact recent results on approximation of general metrics by tree metrics [3, 4, 9] imply that any hierarchal placement algorithm may be used to obtain a placement algorithm for any arbitrary metric cost model with at most a blow up in the approximation factor by a polylog factor in the number of nodes.

Cost Model

A *placement* assigns copies of objects to machines in a hierarchy. A *copy* (i,α) is a pair consisting of a machine i in the hierarchy and an object α. Every object α is associated with an expiry time denoted by $expiry(\alpha)$ that represents the point of time at which the object will be invalidated.[1]

For any machine i and object α, let $p(i,\alpha)$ denote the *probability* of access of object α at machine i before $expiry(\alpha)$. For any hierarchy H, the *aggregate* probability of access of an object α $p(H,\alpha)$ is defined as $1 - \Pi_{i \in machines(H)}(1 - p(i,\alpha))$, *i.e.* it represents the probability of access of object α at at least one of the leaf nodes in the tree rooted at H. The goal of the coordinated placement strategy is to push objects into the caches such that the overall cost of access to these objects is minimized.

In order to develop a notion of the overall cost of a placement given the probabilities of access of objects at the machines, we observe that if there is a request for an object α in any hierarchy H, none of whose machines have a copy of α, the incremental cost of leaving H to fetch a copy of the α is $p(H,\alpha) \cdot (diam(parent(H)) - diam(H))$. The overall access cost $cost(P,H,\psi)$ of a placement P over a hierarchy H and a universe of objects ψ is recursively defined to be $\sum_{\alpha \in \psi} p(h,\alpha) \cdot \delta(h,\alpha) \cdot (diam(parent(h)) - diam(h)) + \sum_{i \in [1,k]} cost(P, H_i, \psi)$, where $H_1, \ldots H_k$ are the immediate children of H and $\delta(h,\alpha)$ takes the value 0 if at least one machine in $machines(h)$ contains a copy of α, else 1. A formal justification of the cost function defined above may be arrived at by a straightforward proof of the claim that for a random series of n object requests in accordance with the distribution of the given probabilities of accesses, the expected total latencies encountered by two given placements are in the ratio of their costs as defined above, as $n \to \infty$.

In practice, access probabilities may be known or estimated using application specific benchmarks with knowledge of history based statistical information [7]. Since objects are associated with lifetimes and probabilities of access

[1] In practice, $expiry(\alpha)$ may be known or estimated *a priori*, *e.g.* a newspaper website may be updated every night at 2am or past update patterns may be used to predict the next update.

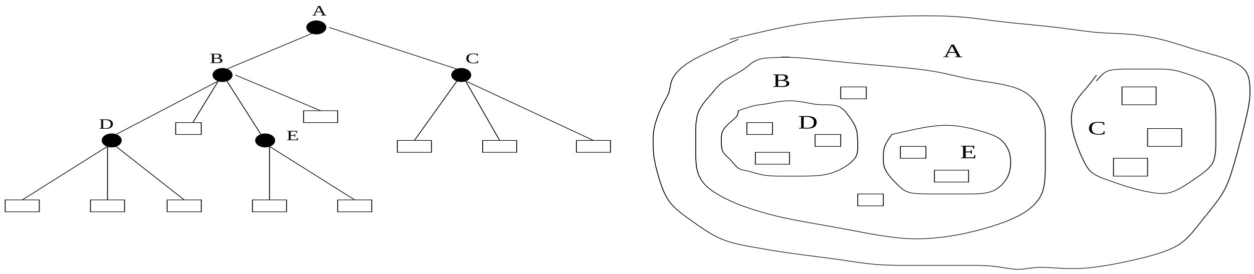

Fig 1. Model for network distances

are likely to show temporal variation, it seems natural to introduce a *fill-time* t_{fill} within which the placement has to be accomplished. Thus, given probabilities of access and the available bandwidth to push objects into machines, the goal of the bandwidth constrained placement problem is to compute the placement with the minimum cost that can be accomplished within the fill-time.

4. ALGORITHMS

In this section we present some core algorithms to handle the bandwidth constrained placement problem as defined above as well as its extensions to more realistic scenarios. We start with a basic algorithm called the *Fixed-t_{fill}* algorithm that solves the version of the placement problem defined in section 3 , *i.e.* it assumes that (i) all the caches are initially empty, (ii) objects do not get modified during t_{fill}, (iii) all object requests occur after the placement is complete, (iv) the probabilities of access and the universe of objects are fixed, and (v) all objects are of equal size. We first describe a simple greedy strategy that that captures the core idea of the *Fixed-t_{fill}* algorithm. We then describe an amortized version of this algorithm which adds a small technical adjustment needed for bounding the algorithm's worst case performance with respect to the optimal placement. Subsequently, in section 5 and 6 we proceed to progressively refine this algorithm to handle (i) non-empty initial placements, (ii) simultaneous placement and accesses to objects, (iii) object updates and introduction of new objects and (iv) variable sized objects.

All the algorithms described in this section have been described, for simplicity's sake, under the assumption of a centralized implementation. However, it is straightforward to transform the centralized implementation into an efficient distributed one by a procedure that parallels the approach outlined in [20].

4.1 The Greedy *Fixed-t_{fill}* Algorithm

The algorithm we present involves a bottom-up pass along the cluster tree. It starts with a tentative placement in which the caches at the leaves of the cluster tree pick the locally most valuable set of objects. Then the algorithm proceeds up the hierarchy by having each node improve the corresponding subtree's placement.

It must be emphasized that the problem of determining a good placement is conceptually distinct from accomplishing that placement itself. The latter involves routing issues which we do not address directly in this discussion. These issues are simply abstracted out in terms of an effective bandwidth $bw(i)$ available to push objects into machine i's cache.

We first introduce a few definitions. For any H, and a placement P over it, we define an object α to be *P-missing* if no copy of α exists in any of the caches in H. The benefit of an object α at a machine i in a placement P is defined as the increase in the cost of the placement over H were α to be dropped from i, and is denoted by *benefit*(i, α, P). It follows from the definition that the benefit of a copy is dependent on where other copies of the same object are distributed among the machines in the hierarchy. The *value* of a *P-missing* object α in a hierarchy H is defined to be $p(H, \alpha) \cdot (diam(parent(H)) - diam(H))$. The *Fixed-$t_{fill}$* greedy algorithm for bandwidth constrained placement is given below.

Input: A hierarchy U, the universe of objects ψ, a fill time t_{fill}, available bandwidth $bw(i)$ for $i \in machines(U)$, the access probability $p(i, \alpha)$ for all objects α and $i \in machines(U)$. Let all objects be of size *objectsize*.

Initialization: For each $i \in machines(U)$, set $size(i) = bw(i) \cdot t_{fill}$. For each $i \in machines(U)$, select the $\lfloor \frac{size(i)}{objectsize} \rfloor$ objects with the highest local probability of access $p(i, \alpha)$ and call this local placement P_i. For each object α that gets selected at machine i, initialize $AssignedBenefit(i, \alpha) = p(i, \alpha) \cdot (diam(parent(i)) - diam(i))$

Iterative step: step d, $1 \le d \le depth(U)$

1. Compute the level-d placement for each depth d hierarchy H as follows: Let $H_1, \ldots, H_k$ be the constituting hierarchies that are the immediate children of the hierarchy H. Initialize the placement P_H over C to the union of the placements already computed at $H_1, \ldots, H_k$, *i.e.* $P_H = \cup_{i \in children(H)} P_i$.
2. *Update benefits:* For each object α in H that has one or more copies in the current placement P_H, let (i, α_p) be the *primary* copy, *i.e.* the copy with the highest local *AssignedBenefit.* The *AssignedBenefit* of this primary copy is increased by H's aggregate access probability times the cost of leaving hierarchy H, *i.e., AssignedBenefit*$(i, \alpha_p) +=$ $p(H, \alpha) \cdot (diam(parent(H)) - diam(H))$. All other copies of α are known as secondary copies and their *AssignedBenefits* remain unchanged. Let X denote the set of *P_H-missing* objects.
3. *Greedy Swap Phase:* While there is a *P-missing* object β in X, whose *value* is more than the copy (i, α) with the least *AssignedBenefit* in P, remove the copy (i, α) and substitute a copy of (i, β) with its *AssignedBenefit* initialized to its *value* just before the swap.

Object Insertion: After the placement P over the hierarchy H has been computed, send out the selected copies of objects into the corresponding machines in the order of their $AssignedBenefits$ computed at the end of the loop above.

The key idea is that the swapping procedure at every level continues till there exists a *P-missing* object with value greater than the least *beneficial* copy in the current placement. The greedy swapping procedure stated above uses *AssignedBenefits* instead. However, observe that the *AssignedBenefit* represents the *benefit* of a copy just before it gets swapped out, and that the *benefit* of a copy is at most equal to its *AssignedBenefit.* It follows that the greedy swapping rule is equivalent to one based on swapping out the copy with the least *benefit.*

It may be verified that the execution time complexity of the greedy algorithm given above is $O(C \cdot N)$, where C is the sum of the cache sizes at all of the leaf nodes and N is the total number of nodes in the hierarchy.

4.2 The Amortized *Fixed-t_{fill}* Algorithm

Though the greedy algorithm seems simple and promising, the placement it computes in the worst case could be arbitrarily far from optimal. The fundamental drawback of the algorithm is that a single secondary copy of some object may prevent swapping in of several missing objects. Though the benefit of the secondary copy may be larger than the value of each of the missing objects, on the whole it might be much less than the sum of all these values put together. Korupolu *et al.* [20] demonstrate that for the space constrained version of the greedy algorithm, this effect can lead to a placement arbitrarily far from optimal.

In order to overcome this problem, we augment the greedy algorithm with an amortization step using a potential function identical to the one used in [20]. 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. The Amortized *Fixed-t_{fill}* algorithm is as follows:

Initialization: Same as greedy except that we also set a potential ϕ_i for each machine i to zero.

Iterative Step: Same as in the greedy algorithm except that the potential ϕ_H is set to the sum of the potentials $\phi_1, \ldots, \phi_2$ computed by the children of H.
1. *Update Benefits:* Same as in the greedy algorithm.
2. *Greedy Swap Phase*: This procedure is similar to the swapping procedure in the greedy algorithm except that the potential ϕ is used to reduce the $AssignedBenefits$ of some copies.

(i) Let y_p be the primary copy with the least *Assigned-Benefit* and y_s the secondary copy with the least *Assigned-Benefit* in P_H. Let α be the highest-valued object in X, the set of all P_H*-missing* objects.

(ii): If *value*(α) > *min*(*AssignedBenefit*(y_p), *AssignedBenefit*(y_s) − ϕ_H), then perform one of the two following swap operations, and goto step 1.

- If *AssignedBenefit*(y_p) < *AssignedBenefit*(y_s) − ϕ_H, swap y_p with α. Set X to $X - \alpha + \alpha'$, where α' is the object corresponding to the copy y_p. Set $value(\alpha')$ to *Assignedbenefit*(y_p).
- Otherwise, remove y_s from Q and substitute it with α. Set X to $X - \alpha$ and reset the potential to $max(0, \phi_H -$ *Assignedbenefit*(y_s)).

3. *Update Potential:* Add the values of all the P_H*-missing* objects in X to ϕ.

Theorem 1: The cost of the amortized $Fixed - t_{fill}$ algorithm is within a constant factor of the cost of the optimal placement.

The proof of this theorem has been outlined in the appendix and closely follows that in [20]. Though the worst-case constant (about 14) is large for practical purposes, measurements indicate that the space constrained variation performs nearly optimally for the workloads examined in [19], and we expect similar performance for bandwidth constrained workloads. In the rest of the paper we just use the term *Fixed-t_{fill}* algorithm to mean amortized *Fixed-t_{fill}* algorithm.

5. THE DYNAMIC CASE

In the previous section we introduced a placement algorithm that is a straightforward extension of the space constrained amortized placement algorithm in [20]. However, we made several restrictive assumptions. In a content distribution network for example, we need to handle object updates dynamically. New copies of objects have to be continuously pushed out to the caches to maintain consistency. The universe of objects itself may change with time and the objects may have arbitrary sizes. In this section we perform a stepwise refinement of the *Fixed-t_{fill}* algorithm that addresses most of these constraints.

We first present the *InitFill* algorithm that computes a placement given an existing set of objects already placed at the machines. To relate this to a practical scenario, imagine a content distribution network where we have placed a subset of objects at the caches or content servers over the course of an hour. At this point a significant number of objects could possibly change, along with their probabilities of access. For example, a sudden important news event at a news website could cause new articles with higher popularity to appear. Since not all of the objects might have changed, we would still like to leverage the benefit of the objects already placed in the network. However, the *Fixed-t_{fill}* algorithm introduced in section 4, assumes the caches to be initially empty. The space constrained placement algorithm on which we based the *Fixed-t_{fill}* algorithm does not take into consideration the objects already present in the cache. In general, the *InitFill* algorithm is useful for iterative application of the placement algorithm.

Next, we discuss the issues that arise when choosing t_{fill} in a dynamic environment where objects could get modified during the placement, the universe of objects and associated access probabilities could change, requests for objects could occur simultaneously with the placement and the caches could start from a non-empty state. We show examples for which naive strategies for selection of the fill time, could lead to arbitrarily poor placements. We then present the *DoublingEpoch* algorithm that handles such dynamic object updates as well as modification of the access probability distribution and the universe of objects. We analyze this algorithm and prove that, given a 4X blow-up in bandwidth, this algorithm computes a placement whose cost is within a constant factor of the cost of the optimal. The *DoublingEpoch* algorithm is therefore useful for continuous application of the placement algorithm in a dynamic situation.

5.1 Initial Placement

Assume that we already have an initial placement over a hierarchy H given to us instead of empty caches. We modify the *Fixed-t_{fill}* amortized algorithm are as follows. We a) add sufficient free *virtual bandwidth* to each cache to store the already placed objects and b) artificially inflate the access probabilities of the already placed copies to force the algorithm to include them in the placement. The Following is the *InitFill* algorithm, given an initial placement P_1:

1. **Initialization:** For each machine i, set $size(i)$ to the sum of the combined size of the objects already present in the cache and $bw(i) \cdot t_{fill}$. For each copy $(i, \alpha) \in P_1$, set $p(i, \alpha) = 1$. Prioritize the copies in P_1 so that, ties while placing the first $\lfloor size(i)/objsize \rfloor$ objects at machine i during the initialization phase are always broken in favour of copies in P_1. (Note that the greedy/amortized algorithm allows us to do this.) For each of the already placed copies (i, α), set $AssignedBenefit(i, \alpha)$ to ∞.
2. Run the rest of the *Fixed-t_{fill}* as before on this modified problem instance, resulting in a final placement P.
3. Delete from P, all copies $(i, \alpha) \in P_1$. (This is so that the already placed objects don't have to be pushed into their corresponding caches again during the *Object Insertion* phase.)

Theorem 2: The placement computed by the *InitFill* algorithm is within a constant factor of optimal.
Proof: It is clear that none of the initially placed objects ever get swapped out during the course of the *Fixed-t_{fill}* algorithm, since their *Assignedbenefit* is set to ∞ at level 1. That they get placed at level 1 is ensured by setting their access probabilities to 1, prioritizing them and ensuring that sufficient bandwidth is available to place them. Let $P^{'}$ be the placement computed by the *InitFill* algorithm and $P^{'}_{opt}$ the optimal placement with the already placed objects with respect to the modified access probabilities. Let P_{opt} denote the optimal placement with the already placed objects with respect to the unmodified access probabilities. Then, by the result of optimality to within a constant factor $c(\approx 14)$ proved in the previous section, $c \cdot cost(P^{'}) \leq cost(P^{'}_{opt})$. However, $cost(P^{'}_{opt}) = cost(P_{opt})$ in spite of the modified access probabilities for the already placed objects because the terms corresponding to the already placed copies anyway contribute to zero in either of the costs. It follows that $c \cdot cost(P^{'}) \leq cost(P_{opt})$. □

The *InitFill* algorithm is attractive since it allows for an incremental implementation of the static algorithm. We could start with the cache at each machine being empty at some initial time t_0 and then onwards at various points of time, invoke the *InitFill* algorithm to recompute a placement.

We also point out that the *InitFill* algorithm may be used to compute close to optimal placements over a distance model wherein the servers are located at different distances from different caches and have varying performance properties. To accommodate this extension, for each server create a new dummy cache located in the network topology at the same place as the server and include each of these dummy caches in the virtual cluster hierarchy in the same manner as regular caches. Set the bandwidth of each dummy cache to be 0 and initialize the contents of each dummy cache to include the objects served by the corresponding server. The *InitFill* algorithm may now be used so that the cost of accessing an object from a dummy cache matches the miss cost of fetching it from the server.

5.2 Challenges in choosing t_{fill}

The discussion above assumes that a time epoch t_{fill} is known *a priori*. However, in real systems, it is potentially difficult to choose t_{fill} optimally. First, in some systems it may be difficult to predict when the universe of objects being placed (or their probabilities of access) will change. Second, even for a static set of objects and access probabilities, the placement algorithms in the previous section optimize the performance of the placement achieved *after* t_{fill} time has elapsed. If placement and client reads proceed in parallel (which may often be the case in real systems), then reads during the t_{fill} interval may see substantially sub-optimal performance during this transient interval. In this section we investigate the problem of minimizing the transient latency cost during the course of the placement itself and motivate the need to select an optimal sequence of epoch times to achieve the same.

Choosing too long a t_{fill} interval and then sending out the selected objects in the order of their *AssignedBenefits* to the individual caches may cause the transient access cost of the placement to be arbitrarily far from optimal. We illustrate this point using a specific example below. The transient access cost of a placement is defined as the average response time to service a request over the course of the complete interval.

Claim 1: Let $P_T(t), 0 \leq t \leq T$ be the placement strategy that computes a placement at time 0 using the *Fixed-t_{fill}* algorithm with a fill time T, and thereafter sends out the selected objects to the individual caches over the entire interval T in the order of their *AssignedBenefits* so that at any time t, $\lfloor t \cdot bw(i)/objsize \rfloor$ objects have been placed at cache i. Throughout the epoch of length T, requests are served by the system according to the probability distribution specified. There exist topologies and access probability distributions for which the transient access cost of $P_T(t)$ could be arbitrarily far from the optimal achievable.

Example: Consider a topology consisting of a single level hierarchy with n machines numbered 1 to n, diameter 1 and miss penalty n. Assume that there exist n data objects $\alpha_1, \alpha_2, \ldots, \alpha_n$ and that the access probability distribution is as follows: $p(1, \alpha_i) = P,\ 1 \leq i \leq n$ and $p(j, \alpha_i) = 0,\ 2 \leq j \leq n, 1 \leq i \leq n$. We define a *time-step* as a small value of t_{fill}, say δ_{fill}, such that for each of the leaf nodes i, $bw(i) \cdot \delta_{fill} \geq objsize$. Let the probability of access of object j at machine 1 in any time-step be ρ [2]. Further, assume that $T = n \cdot \delta_{fill}$

The T epoch placement algorithm places one copy of all the n objects on machine 1. The cost of this final placement is 0. However, this placement takes n time-steps to complete and the cost of the transient placement[3] at step i for object j is given by 0, if $j < i$ (because object j would have already been placed at the j'th time-step), and $(1-\rho)^i \cdot \rho \cdot n$ otherwise. The term $(1-\rho)^i \cdot \rho$ represents the probability that there is a

[2] In practice ρ could be related to P as $1 - (1 - \rho)^\eta = P$, where η is the number of time-steps in the object's lifetime

[3] a demand placement that happens to satisfy a request for an unavailable object.

request for object j at machine 1 for the first time in step i. Therefore, the average response time to service a request for object j is given by $\sum_{i\in[1,j]} n\cdot(1-\rho)^i\cdot\rho = n\cdot(1-(1-\rho)^{j+1})$. The transient access cost of $P_T(t)$ is thus:

$$AccessCost_1 = \frac{1}{n}\sum_{j\in[1,n]} n\cdot(1-(1-\rho)^{j+1}) = O(n)$$

A better strategy is to place a copy of object i at machine i during step 1, and then to place a copy of object i at machine 1 during step i. For this placement, the cost to service a request for object j in step 1 is $\rho\cdot n$ and is bounded by $\rho\cdot 1$ in subsequent steps. The average response cost for a request for to object j is at most $\frac{1}{n}(\rho\cdot n + n\cdot\rho\cdot 1) = \rho+1$. Thus, it follows that the transient access cost of this placement strategy is

$$AccessCost_2 = \frac{1}{n}\sum_{j\in[1,n]}(\rho+1) = O(1)$$

Thus, from the above it is clear that the transient access cost of $P_T(t)$ to could be arbitrarily far from the optimal. □.

From the above example, it is clear that choosing t_{fill} to be too long may lead the system to defer placing important objects because a *more valuable* location will later become available, resulting in a higher transient miss rate and therefore a higher average response time per request. This suggests that even if objects are static, *i.e.* they do not change, the naive strategy of computing a placement for a huge epoch and then pushing out objects in the order of their final *benefits* may not be efficient with respect to the time-averaged access cost of the placement.

Conversely, choosing t_{fill} to be too short can cause the system to waste work by placing a copy at a sub-optimal location before placing it at the *right* place at a later time. The system therefore may end up taking more overall bandwidth or time (number of epochs) to complete a placement that is as good as the optimal placement for a larger epoch. We show this formally below again with an example:

Claim 2: Let P be the placement computed by the *Fixed-*t_{fill} algorithm for a given fill time T. Let δ_f denote a tiny epoch such that for all machines i, $\delta_f\cdot bw(i) \geq objsize$. Then the number of iterations of the *InitFill* algorithm required to compute a placement that is at least as good as P could be as many $T/\delta_f\cdot log(n)$, where n is the total number of machines in the hierarchy.

Example: Consider a topology consisting of a single level hierarchy with n machines numbered $1,\ldots,n$, diameter d and miss penalty Δ. Assume that the set of objects and the distribution of probabilities of access to objects are as follows: (i) every object is accessed at exactly one machine, there are k distinct objects accessed at every machine. (ii) the probability of access of any object α at machine i, $p(i,\alpha) > \frac{\Delta}{d}\cdot p(i+1,\beta)$, where β is any object accessed at machine $i+1$. Assume further that all machines have the same bandwidth and every iteration of the δ_f epoch algorithm allows at most one object to be inserted into any machine. Assume T/δ_f to be k, so that the cost of the T epoch placement is 0 (every object gets placed where it is accessed). At the end of the first iteration of the δ_f epoch InitFill algorithm, machine i possesses a copy of the object with the $i'th$ highest probability of access. Thus, the first k/n iterations place objects accessed at machine 1 at all the machines. The next $k/(n-1)$ iterations place copies of objects accessed at machine 2 at machines $2,3,\ldots,n$, and so on, without displacing the secondary copies of objects accessed at machine 1 because of the above constraint on the probabilities of accesses. Thus, the number of iterations of the δ_f epoch algorithm before machine n can place a copy of an object it accesses is $k/n+k/(n-1)+k/(n-2),\ldots,k/1 = \Omega(k\cdot log(n))$. Thus, the time taken by an iterative small epoch algorithm could be a factor of $log(n)$ more than the corresponding long epoch version for the cost of a placement computed by the former to match that computed by the latter. □

5.3 The Doubling Epoch Algorithm

Based on the intuition gathered by the discussion above we present here an algorithm that does not assume knowledge of a fill time *a priori* and that varies the epoch times in a manner such that the overall placement at any time is *close* to optimal, *i.e.* the placement sampled at any time t can be proved to be within a constant factor of the optimal achievable with 1/4'th the bandwidth available to fill the caches. We first present the algorithm, perform a worst case analysis and then proceed to explain why the bound is reasonable.

Input: A hierarchy H, a set of $objsize$ sized objects ψ, and probabilities of access $p(i,\alpha)$ of objects $i\in\psi$ at $i\in machines(H)$, and an initial placement P_0
Algorithm:
(i) Initialize epoch length $T_0 = \delta_{fill}$, where δ_{fill} is the minimum value such that for all $i\in machines(H)$, $bw(i)\cdot\delta_{fill} \geq objsize$.

(ii) for($i=0, T_0=\delta_{fill}$; *until done*; $i++, T_i = T_{i-1}*2$)
Run *InitFill* algorithm for epoch length $= T_i$, with the current placement resulting from the previous run.

(iii) Goto (i)

The *until done* in the above algorithm means that the loop iteratively executes runs of *InitFill* algorithm until such time that a *change* occurs or all objects end up getting placed at every machine that they are accessed at. A change could possibly be an object update, a demand placement, introduction of a new object or a change in its probability of access at a particular machine. Theorem 3 below bounds the worst case performance of this algorithm.

Theorem 3: Assume that the bandwidth to the caches is $B[\ ]$. At time t seconds after a change, the doubling epoch algorithm creates a placement that is within a constant factor of the optimal placement that can be achieved by a system with bandwidth $B[\ ]/4$ in time t with the same initial conditions. *i.e., the doubling epoch algorithm computes a placement that is within a constant factor of optimal at any instant between changes with a 4X blow-up in bandwidth.*

To prove theorem 3, we first introduce the following lemma:
Lemma 1: For any two given initial placement I_1 and I_2, such that I_2 is a superset of I_1, the *InitFill* algorithm starting with the initial placement I_2 with a fill time T, computes a placement P that is within a constant factor of the optimal placement P_{opt} obtained by starting with the placement I_1 and the same fill time T.

Proof: By the proof of optimality to within a constant factor of the *InitFill* algorithm, we assert that for any given epoch time, the placement P computed by the *InitFill* algorithm starting with the initial placement I_2 is within a constant factor of the optimal placement starting with the initial placement I_2. However, it is also trivially true that cost of the optimal placement starting with I_2 is at most the cost of the optimal placement obtained by starting with I_1, (given the same fill epochs), since I_2 is given to be a superset of I_1. It follows that P is within a constant factor of the optimal placement starting with I_1. □

Proof of Theorem 3: Starting from an initial placement I_1, at any time t the doubling epoch algorithm has completed epochs of length $t/4, t/8, \ldots$ (and has partially completed the epoch of length $t/2$). Denote the placement at the beginning of the epoch of length $t/4$ by I_2. It is clear that I_2 is a superset of I_1. Since the doubling epoch algorithm has completed an epoch of length $t/4$, the resultant placement is at least as good as the placement we would have obtained in time t with $1/4$'th the bandwidth and starting with I_1. This follows from lemma 1 above. □

Note that the bandwidth blow up of 4 is a loose worst case estimate. The proof relies only on the fact that at time t, we have completed a sub-epoch of length $t/4$. However, we also will have completed sub-epochs of length $t/8, t/16, \ldots$. During the *short epochs* the system will place high-benefit objects into machines that are typically close to their optimal locations. The quadrupling result of bandwidth gives no credit for this. Thus, in practice, the epoch doubling algorithm will give much better performance than a 4X blow up in bandwidth.

It must also be emphasized that a 4X increase in bandwidth may not very damaging to the hit rate. Web caches typically exhibit a log-linear relationship between cache size and hit rate. Doubling a cache's size normally increases hit rate by less than 5% (*e.g.* for web caches) [15, 27], so a 4X blow-up in bandwidth often will not hurt the hit rate much. The epoch doubling algorithm handles the problem of choosing the *optimal* epoch, since we no longer have to pick an epoch *a priori*, whatever arbitrary changes occur in the set of objects or the access pattern. Between changes we are assured to be within a constant factor of the optimal. Observe that this property implies that for a static set of objects and access probabilities, the transient access cost of the *DoublingEpoch* algorithm with a bandwidth blow-up of four is also assured to be within a constant factor of the optimal achievable.

5.4 Dynamic Continuous Placement

The *InitFill* and the *DoublingEpoch* algorithms provide a basis for coping with systems that may be changing almost continuously because of (i) object updates causing previously placed copies to be invalidated, (ii) creation of new objects to be placed, (iii) a demand-read of an object by a cache resulting in an extra copy of an object, (iv) changes in the system's estimates of object-access probabilities, (v) changes in the system's estimate of network performance (e.g., fill bandwidths or inter-machine distances).

With the *DoublingEpoch* algorithm, whenever a change occurs, the system begins a new placement epoch with $t_{fill} = 1$ and with the updated situation as input. Unfortunately, the optimality result to within a constant factor with a $4X$ blow-up in bandwidth holds for the *DoublingEpoch* algorithm between changes, but not across changes. On one hand, if a change is *large*, resetting $t_{fill} = 1$ and *starting over* may be appropriate. On the other hand, if a change is *small*, a less radical adjustment to the schedule seems in order. Determining how to update a placement schedule so that the disruption to the original schedule is proportional to the scale of the change event is an interesting topic for future work. Some heuristics worth exploring are (i) periodic resets based on diurnal patterns of object updates, (ii) resetting when $c(P_{new}(k+1)) - c(P_{old}(k+1)) \geq \eta \cdot c(P_{old}(k+1))$, where $c(P_{new}(k+1))$ and $c(P_{old}(k+1))$ are the costs of the placements at the end of the *next* epoch, (the current epoch being the k'th) using the old and new object set and access probabilities resp. and $\eta < 1$ is a constant, (iii) resets based on monitoring objects updates or number of of new objects as a percentage of the current total number of objects.

Although the worst-case performance of resetting t_{fill} to a small value when changes occur may be poor, this approach may still offer a reasonable heuristic because it is conservative – using too short a t_{fill} interval causes the system to place *important* objects into key subtrees early (at the cost of not picking the best node within a subtree). In section 5.2 we proved that the δ_f epoch iterative algorithm could be a factor $log(n)$ worse in the number of epochs. However, it can be shown that with a factor h blow-up in the number of epochs, where h is the height of the hierarchy, the δ_f epoch iterative algorithm achieves a placement that is within a constant factor of the optimal achievable. The formal statement and a proof outline are given in the appendix.

6. VARIABLE SIZED OBJECTS

In this section we show that the bandwidth-constrained placement problem for variable sized objects, even for a static universe of unchanging objects and access probabilities, is a hard problem. This can be shown by a straightforward reduction from the Knapsack problem. However it is still tempting to look for approximation algorithms by tweaking the above algorithms appropriately. However, in the following, we establish that unless P=NP, no polynomial time algorithm can provide a finite approximation guarentee to the bandwidth-constrained hierarchical placement problem.

To establish this, suppose there does exist an approximation algorithm $\mathcal{A}$ that produces a placement of variable sized objects to within a constant factor c of the optimal, for a fixed c. We show that $\mathcal{A}$ can be used to solve the partition problem (which is known to be NP-complete).

The Partition problem takes as input a finite set $\mathcal{S}$ of objects with positive sizes and partitions them into two subsets S_1 and S_2 such that the sum of the sizes of objects in S_1 is equal to the sum of the sizes of objects in S_2.

Let $\mathcal{S} = \{a_1, a_2, \ldots, a_n\}$ be an instance of the partition problem. Let $s(a_i) \in Z^+$ be the size for each object, and $C = \sum_{i=1}^{n} s(a_i)$. In order to solve the partition problem using algorithm $\mathcal{A}$, consider a two level hierarchy consisting of two caches each of size $\frac{C}{2}$, connected by a root. Assume that the diameter of this hierarchy is 1 and the miss penalty is nc. Set the probability of access of each object at each cache to be a fixed value, say p. It is now straightforward to show that a partition of $\mathcal{S}$ exists if and only if $\mathcal{A}$ can produce a placement with cost less than n.

Analogous to the value density heuristic for the Knapsack problem, we can modify the Amortized *Fixed-t_{fill}* algorithm's swapping phase as follows:

Swapping phase: Swap out the object with the least *AssignedBenefit*-density *i.e.,* $\frac{Assignedbenefit}{size}$ and replace it with the *P-missing* object α with the highest *value*-density *i.e.,* $\frac{value}{size}$ that fits into the available cache space.

A theoretical analysis of the above algoritm is complicated. We intend to measure as part of future work performance of various heuristics for variable sized objects.

A good algorithm for the placement of variable sized objects immediately allows us to efficiently solve a restricted steady-state version of the bandwidth constrained placement problem, where the universe of objects and their probabilities of access are fixed, the only *changes* in the system are object updates, and all objects are of the same size *objsize*. Let $\mu(\alpha)$ denote the frequency of update of object α at the server(s). Assume that in steady-state all of the bandwidth to a cache is used for keeping objects stored in the cache fresh. Thus, the bandwidth $B(\alpha)$ consumed by a copy of α at any machine is $\mu(\alpha) \cdot objsize$. Given the frequency of access $f(i,\alpha)$ of object α at machine i, the steady-state bandwidth constrained problem is to minimize the overall cost of access, as defined in the frequencies model developed in [19]. Thus, the steady-state bandwidth constrained placement problem can be viewed as a direct instance of a space constrained placement problem with the size of object α as $B(\alpha)$ and the space available at machine i as $bw(i)$.

7. CONCLUSION

In this paper we have introduced the bandwidth constrained placement problem and presented an amortized *Fixed-t_{fill}* algorithm that is provably within a constant factor of the optimal. We have also provided the *Doubling Epoch* algorithm which generates a sequence of placements that are continuously within a constant factor of the optimal placement with at most a 4X blow-up in the the bandwidth. As a part of future work we intend to experimentally verify different heuristic algorithms for varying the epoch time and to analyze the placement problem for variable size objects.

REFERENCES

[1] Akamai. Fast internet content delivery with freeflow. In *White Paper*, Nov 1999.

[2] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583, January 1996.

[3] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *In Proceedings of the 30'th Annual ACM Symposium on Foundations of Computer Sciences*, pages 184–193, October 1996.

[4] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *In Proceedings of the 30'th Annual ACM Symposium on Theory of Computing*, pages 161–168, may 1998.

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

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

[7] V. Cate. Alex - a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

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

[9] M. Charikar, S. Guha, D. Shmoys, and É. Tardos. A constant-factor approximation algorithm for the k-median problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 1–10, May 1999.

[10] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[11] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Client Proxy Caches. In *USITS97*, Dec 1997.

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

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

[14] J. Gray and P. Shenoy. Rules of Thumb in Data Engineering. In *"Proc. 16th Internat. Conference on Data Engineering"*, pages 3–12, 2000.

[15] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.

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

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

[18] D. Karger, F. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In Proceedings of the 29'th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1998.

[19] Madhukar Korupolu and Mike Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *Workshop On Internet Applications*, June 1999.

[20] Madhukar Korupolu, Greg Plaxton, and Rajmohan Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 586–595, January 1999.

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

[22] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 284–293, October 1997.

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

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

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

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

[27] Arun Venkataramani, Ravi Kokku, Praveen Yalagandula, and Sadiya Sharif. The potential costs and benefits of long-term prefetching. Technical Report TR-01-13, The University of Texas at Austin, Dept. of Computer Sciences, 2001.

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

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

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

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

Appendix

Theorem 1: The *Fixed-t_{fill}* bandwidth constrained amortized placement algorithm is within a constant factor of the optimal.
Proof Outline: The proof of optimality of the amortized bandwidth constrained placement algorithm exactly parallels the proof of its space constrained counterpart introduced in [20]. The fundamental difference between the cost model we developed and that used in [20] is that the *access quotient* in the former is based on probabilities of access while in the latter, it is the frequency of access. The cost of a placement in the former is defined as $\sum freq(\alpha, i) \cdot diam(closest(\alpha, i))$, over all $i \in machines(H)$ and $\alpha \in \psi$, where $freq(\alpha, i)$ denotes the frequency of access of object α at machine i and $closest(\alpha, i)$ is the least common ancestor v of i such that at least one copy of the object α is present in $machines(v)$. We first claim that both the cost models are identical except for the definition of the aggregate access quotient at interior nodes.

Lemma 1: Let $f(\alpha, i)$ denote the frequency of access of object α at a leaf node i in a hierarchy H. Let $f(\alpha, v)$ at an interior node v denote the aggregate access quotient as defined in the frequencies model to be the sum of the frequencies of access of object α over all $i \in machines(v)$. Then,

$$\textstyle\sum_{u\in H,\alpha\in\psi} f(u,\alpha)\cdot\delta(\alpha,u)\cdot(diam(parent(u)) - diam(u)),$$
$$= \textstyle\sum_{i\in machines(H),\alpha\in\psi} f(u,\alpha)\cdot(dist(closest(\alpha,i))$$

The proof of the above lemma is by a straightforward rearrangement of terms. Thus, it follows that the amortized bandwidth constrained algorithm is within a constant factor of the optimal if the access quotient to an object at an interior node were defined as the sum of the access quotients to the same object over all leaf node descendents of that node.

The proof of optimality to within a constant factor in the frequencies case proceeds by introducing what is known as a bridging placement. The algorithm to compute the bridging placement is parameterized by a fixed, but arbitrarily chosen placement, say P. The bridging placement, say B, is an intermediate placement that is then proved to be at least as expensive as the corresponding amortized placement and is also proved to be within a constant factor $\lambda.(1+3\lambda/(\lambda-1))$ of P. Since P could be arbitrarily chosen and hence chosen as the optimal placement, it follows that the amortized placement is within a constant factor of the optimal. All the properties of the aggregate access quotient as defined in the frequencies model also extend the probabilities model. In particular we state below a property that relates the benefit of a copy to its value just before it gets swapped in. This property is crucial in deriving the constant factor of approximation $\lambda(1+3\cdot\frac{\lambda}{\lambda-1})$, and it is straightforward to show that it holds in the probabilities model as well.

Lemma 2: Let H denote a $\lambda - separated$ hierarchy for some $\lambda > 1$, let P denote an $H - placement$ in which a copy of an object α be placed at a machine i. Then $benefit(\alpha, i) \leq \lambda/(\lambda-1).value(H, \alpha)$.

Lemma 3: Let P be the optimal placement computed by the amortized *Fixed-t_{fill}* algorithm for a given epoch T. Let δ_f denote a tiny epoch such that for all machines i, $\delta_f \cdot bw(i) \geq objsize$. Then in $\frac{T}{\delta_f} \cdot h$ iterations, the δ_f epoch iterative algorithm, achieves a placement that is within a constant factor of P.
Proof Outline: The proof of this algorithm proceeds by showing that the δ_f epoch iterative algorithm achieves a placement whose cost is within a constant factor of that of the T epoch placement algorithm. The basic argument is that though the iterative algorithm introduces *greedy* replicas of copies at non optimal locations, the number of greedy replicas for every copy in the T epoch algorithm is at most h, the height of the hierarchy. The formal proof proceeds by introducing a variant of the *InitFill* algorithm known as the *Marking* algorithm that does not assign infinite benefits to initially placed copies, but never swaps them out either. It is shown that the *Marking* algorithm in $\frac{T}{\delta_f} \cdot h$ iterations, is within a $\frac{\lambda}{\lambda-1}$ factor of the T epoch placement. Finally, the δ_f iterative algorithm is shown to be within $\frac{\lambda}{\lambda-1}$ factor of

the *Marking* algorithm. □