Placement Algorithms for Hierarchical Cooperative Caching

Madhukar R. Korupolu[*] *C. Greg Plaxton*[*] *Rajmohan Rajaraman*[†]

Abstract

Consider a hierarchical network of machines in which each machine periodically issues a request for an object drawn from a fixed set of unit-size objects. Suppose further that the following conditions are satisfied: (i) the frequency with which each machine accesses each object is known, (ii) each machine has a cache of known capacity, (iii) any cache can be accessed by any machine, and (iv) any request is satisfied by the closest machine with a copy of the desired object. In such an environment, it is desirable to fill the available cache space with copies of objects in such a way that the average access cost is minimized. We provide both exact and approximate polynomial-time algorithms for this hierarchical placement problem. Our exact algorithm is based on a reduction to min-cost flow, and does not appear to be practical for large problem sizes. Thus we are motivated to search for a faster approximation algorithm. Our main result is a simple constant-factor approximation algorithm for the hierarchical placement problem that admits an efficient distributed implementation.

1 Introduction

Cooperative caching [12] is an emerging paradigm in the design of scalable high-performance distributed systems. In traditional caching schemes, the primary function of a cache is to act as a fast intermediate storage between a client or a collection of clients and the servers. In such schemes, each request is satisfied by the cache associated with the requesting machine or by the server of the requested object. Moreover, the storage decisions made by one cache are independent of those made by other caches in the system. The defining characteristic of cooperative caching schemes, on the other hand, is that the caches cooperate in serving one another's requests as well as in storage decisions.

A number of recent studies have discussed the benefits of cooperative caching for distributed file systems and large-scale information systems such as digital libraries and the World Wide Web. These studies include analytical results (e.g., [2, 19]), simulation experiments (e.g., [7, 14, 15]), and prototypes and products (e.g., Harvest [8, 10], xFS [1]). The widely deployed and studied Harvest cache system [8], employs a hierarchical arrangement of object caches to improve access performance. In the Squid cache system [25], which is a successor to Harvest, the caches cooperate via the Internet Cache Protocol [26] to serve one another's misses and thus reduce overall traffic. In the context of local-area networks, the xFS system [1] utilizes cooperative caches to obtain a serverless file system.

This paper studies an important component of cooperative caching schemes, which we refer to as placement. A cooperative caching scheme can be loosely divided into three components: placement, search, and consistency. The *placement* component determines where to place the copies of the objects. The *search* component directs each request to an appropriate copy of the requested object. Finally, the *consistency* component maintains the desired level of consistency among the various copies of an object.

In [2], Awerbuch, Bartal, and Fiat study a general on-line cooperative caching problem on arbitrary networks and present a polylog(n)-competitive algorithm, where n is the number of machines in the network. Their result is impressive in that they provide solutions to the search, placement, and consistency problems in a general adversarial setting. The shortcomings of their scheme, however, lie in the polylogarithmic factor overheads in performance and cache space requirement. In this paper, we study a special case of cooperative caching with the aim of developing simple algorithms that obtain near-optimal (e.g., constant-factor approximation) solutions. We focus our attention on the placement component of cooperative caching and develop placement algorithms for a class of networks

[*]Department of Computer Science, University of Texas, Austin, TX 78712. Supported by NSF grant CCR–9504145. Email: {madhukar,plaxton}@cs.utexas.edu.

[†]College of Computer Science, Northeastern University, Boston, MA 02115. Part of this work was done while the author was at DIMACS, which is an NSF Science and Technology Center, funded under contract STC–91–19999 and partially supported by the New Jersey Commission on Science and Technology. Email: rraj@ccs.neu.edu.

that we refer to as *hierarchical networks*. Our definition of hierarchical networks, which is based on the ultrametric cost model used in [18], is motivated by the fact that modern wide-area networks tend to admit natural hierarchical decompositions. In fact, the existence of a hierarchical decomposition is implicit in several previous studies (e.g., see [10, 24, 25]).

1.1 The placement problem. We address the following placement problem for hierarchical networks. Let $\mathcal{O}$ denote the set of objects and let $size(\sigma)$ denote the cache size at machine σ. We are given for each machine σ and each object α the access frequency for α at σ. Further assume that for any machine σ and object α, the cost of satisfying an access request for α originating at σ is given by the communication cost between σ and τ, where τ is the closest machine that holds a copy of α. The objective of the placement problem is to determine a placement of object copies in the machine caches subject to space constraints such that the average access cost over all machines and all objects is minimized. See Section 2 for a formal statement of the problem.

Our problem formulation is most suitable for applications in which writes are infrequent and changes in the access pattern over short time intervals are moderate. Infrequent writes imply a low overhead in maintaining consistency among the copies of an object, allowing us to separate the concerns of consistency and placement. Moderate changes in the access pattern can be addressed by invoking the placement algorithm at regular intervals. Finally, we consider our assumption that each request is satisfied by the nearest copy of the requested object. This assumption is justified by the existence of algorithms for the search component that direct each request to a nearby, if not the nearest, copy of the requested object [3, 14, 22, 24].

1.2 Our results. We first present a polynomial-time exact algorithm for the hierarchical placement problem based on a reduction to minimum-cost flow. Our reduction, described in Section 3, generalizes the approach of Leff, Wolf, and Yu [19], who solved the problem for the special case of a single-level hierarchy. While the algorithm of Section 3 runs in polynomial time, the degree of the polynomial is sufficiently high to make the algorithm largely impractical.

Thus, we focus on the goal of developing a fast, simple constant-factor approximation algorithm for the hierarchical placement problem that admits an efficient distributed implementation. We first consider a greedy algorithm that obtains a placement with two passes over the hierarchical network: a bottom-up pass followed by a top-down pass. The greedy algorithm, described in Section 4, is based on a natural local improvement heuristic. We show, however, that the approximation ratio of the greedy algorithm is $\Omega(n)$. The lower bound proof leads us to a natural refinement of the greedy algorithm, the amortizing algorithm, which is also described in Section 4. Like the greedy algorithm, the amortizing algorithm starts with a placement in which each machine places in its cache the locally optimal set of objects. The algorithm then iteratively improves the placement in a bottom-up manner as the machines cooperate and share information about the access frequencies across larger regions of the network. This bottom-up pass is followed by the same top-down pass as in the greedy algorithm.

Our main result, shown in Section 5, is that the amortizing algorithm achieves a constant-factor approximation for hierarchical placement problem. In Section 6, we describe an efficient distributed implementation of the amortizing algorithm.

1.3 Related work. The study of cooperative caching was initiated by Dowdy and Foster in the context of allocating files in a distributed network [13]. A sequence of results [6, 20] obtained improved algorithms for centralized as well as distributed file allocation. These results, however, did not consider cache capacities at the individual machines. As mentioned earlier in this section, Awerbuch, Bartal, and Fiat [2] provide an polylog(n)-competitive on-line algorithm for the general placement problem under the assumption that the size of a cache in the on-line algorithm is polylog(n) times more than the size in the optimal algorithm. In contrast, we obtain an optimal centralized and a constant-factor approximate distributed algorithm for the off-line version of the problem on hierarchical networks without any blowup in the cache sizes.

In adopting a communication model based on a fixed cost function, we endeavor to separate the concerns of caching (a higher-level operation) from routing (a lower-level operation). In contrast, some recent papers have incorporated routing issues into caching by either combining the two problems or taking advantage of the available route information. For example, the algorithms developed in [16, 23, 27] tend to cache copies of an object in machines that either lie on or are close to the path along which the object is being transferred. Routing information is also used in the placement algorithms developed in [21], where the primary aim is to minimize the network congestion that may occur when requests and their responses are routed within the network. We also remark that cost models have been adopted in uniprocessor caching systems as well to model scenarios in which the costs incurred in the retrieval of objects on cache misses may differ with the objects [9, 17, 28].

2 Preliminaries

In this section, we formally define the placement problem and its restriction to hierarchical networks.

2.1 The placement problem. We adopt a standard model for arbitrary networks (e.g., see [2, 13, 15, 27]) in which we represent a network by a set $\mathcal{M}$ of machines and a function $dist : \mathcal{M} \times \mathcal{M} \to \mathbf{R}$ that gives the communication cost of sending a fixed-length message from one machine to another. Let $\mathcal{O}$ denote the set of m objects. For each machine σ, let $size(\sigma)$ denote the number of objects that can be stored in the cache at σ. (We assume that all objects are of the same size.) For any machine σ and object α, let $f(\sigma, \alpha)$ denote the access frequency for object α at machine σ.

A "placement" assigns copies of objects to machines subject to the cache capacity constraints. To facilitate a formal definition, we introduce the notion of an *item*. An item is a pair (α, σ), where α is an object and σ is a machine. We say that an item (α, σ) is a *copy* of α and is *located* at σ. Formally, a *placement* P is a set of items such that for any machine σ, the number of items in P that are located at σ is at most $size(\sigma)$.

We now define the total access cost, or simply, the *cost*, of a placement P. We assume that each access request is satisfied by the closest copy of the requested object. In order to account for the cost of accessing objects for which no copy exists in the placement, we assume that we are given a *miss penalty* that is at least the maximum value of the function *dist*. For any machine σ and object α, we obtain the associated access cost $c(P, \sigma, \alpha)$ as follows. If P places at least one copy of α in the network then $c(P, \sigma, \alpha)$ equals $dist(\sigma, \tau)$, where τ is the closest machine at which P places a copy of α; otherwise, $c(P, \sigma, \alpha)$ equals the miss penalty. The cost of P is given by the sum over all machines σ and objects α of $f(\sigma, \alpha) \cdot c(P, \sigma, \alpha)$. The *placement problem* is to find a placement with minimum cost.

2.2 The hierarchical placement problem. In this paper we focus on the special case of the placement problem in which the input network is a "hierarchical network" or, for brevity, a "hierarchy". We refer to this special case of the placement problem as the *hierarchical placement problem*. A *hierarchy* is a depth-d hierarchy for some nonnegative integer d. A *depth-0 hierarchy* is a singleton set containing one machine. For $d > 0$, a *depth-d hierarchy* u is a set of hierarchies $u_1, \ldots, u_k$, each of some depth smaller than d, and at least one of which has depth exactly $d-1$. We refer to the hierarchies $u_1, \ldots, u_k$ as the *children* of u, and to u as the *parent* of u_1 through u_k. A depth-0 hierarchy is *trivial* and a depth-d hierarchy, $d > 0$, is *nontrivial*. Each hierarchy u has an associated diameter $diam(u)$ and miss penalty $miss(u)$ such that the following hold: (i) if u is trivial then $diam(u) = 0$, (ii) if $parent(u)$ exists, then $diam(u) \leq diam(parent(u))$ and $miss(u) = diam(parent(u))$, and (iii) if $parent(u)$ does not exist, then $diam(u)$ is at most the miss penalty and $miss(u)$ is equal to the miss penalty.

We now define $machines(u)$ for any given hierarchy u. If u is trivial, then $machines(u)$ is $\{u\}$. Otherwise, $machines(u)$ is $\cup_{1 \leq i \leq k} machines(u_i)$, where $u_1, \ldots, u_k$ are the children of u. For any two machines σ and τ in $machines(u)$, $dist(\sigma, \tau)$ is defined as $diam(v)$, where v is the least-depth hierarchy such that σ and τ belong to $machines(v)$. (In tree terminology, v is the least common ancestor of the two leaves corresponding to machines σ and τ.) Note that the distance from any machine to its local cache is zero. If we wish to model a cache with nonzero local access latency, we can represent the associated machine σ as a a depth-1 hierarchy with diameter equal to the local access latency of the cache and containing two trivial hierarchies: one with zero access frequencies and storage capacity equal to $size(\sigma)$, and the other with zero storage and access frequencies equal to those of σ.

Hierarchies can be used to model a large class of distributed networks. For example, a homogeneous, k-machine local-area network may be well-modeled as a depth-1 hierarchy. In fact, this is precisely the model used in [1, 19] in the study of caching schemes for networks of workstations. And it seems plausible that some nontrivial hierarchy should provide a reasonable first-order model for a complex, heterogeneous wide-area network such as the Internet. It is also worth noting that recent results on the approximation of general metrics by tree metrics [4, 5, 11] imply that any hierarchical placement algorithm can be used to obtain a placement for any metric cost model while giving up an extra $O(\log n \log \log n)$-factor in the approximation.

Hierarchies can also be used to model multi-level storage: we can introduce a hierarchy for each level of storage and incorporate the disparate speeds of the different levels in the distance function. This approach can be generalized to capture latencies within both the network as well as the caches in any heterogeneous distributed network containing caches with varying speeds.

2.3 Additional notation and terminology. We define a *u-placement* as a placement for the hierarchy u. Assume that hierarchy v is a descendant of hierarchy u and let P be some u-placement. Then we define $restrict(P, v)$ as the v-placement corresponding to the set of items in P that are located in $machines(v)$. For any placement P, object α, and hierarchy u, we define $copies(P, \alpha, u)$ as the number of items (α, σ) in P such that σ belongs to $machines(u)$. We extend the functions *size* and f, defined previously for machines, to apply to any hierarchy u as follows: (i) $size(u)$ is defined as the sum over all machines σ in $machines(u)$ of $size(\sigma)$, (ii) for any object α, we define $f(u, \alpha)$ as the sum over all machines σ in $machines(u)$ of $f(\sigma, \alpha)$.

3 A reduction to minimum-cost flow

In this section we provide a reduction from the hierarchical placement problem to the minimum-cost flow problem. Assume we are given a hierarchy u. Let C denote the sum over each object α, and each machine σ, of the quantity $f(\sigma, \alpha) miss(u)$. We construct an instance G_u of the minimum-cost flow problem with the following two properties: (i) for every u-placement P, there exists a feasible flow in G_u with cost equal to the cost of P minus C, and (ii) for every integral minimum-cost flow in G_u with cost C', there is a u-placement P with cost $C + C'$. It follows that a minimum-cost flow in G_u yields a minimum-cost u-placement.

3.1 The reduction. The instance G_u is constructed as follows. The vertex set consists of the following: (i) a vertex σ for every machine σ in $machines(u)$, (ii) a vertex $\langle \alpha, v \rangle$ for every object α and descendant v of u, and (iii) a source a and sink b. The edge set consists of four types of edges: (i) for each trivial descendant v of u and each object α, there is an edge $(\langle \alpha, parent(v) \rangle, v)$ with capacity 1 and cost $f(v, \alpha)(diam(v) - miss(v))$, (ii) for nontrivial proper descendant w of u and each object α, there are two parallel edges $(\langle \alpha, parent(w) \rangle, \langle \alpha, w \rangle)$ with capacities 1 and ∞, and costs $f(w, \alpha)(diam(w) - miss(w))$ and 0, respectively, (iii) for each object α, there are two parallel edges $(a, \langle \alpha, u \rangle)$ with capacities 1 and ∞, and costs $f(u, \alpha)(diam(u) - miss(u))$ and 0, respectively, and (iv) for each trivial descendant v of u, there is an edge (v, b) with capacity $size(v)$ and cost 0.

3.2 Proof of optimality. We now prove that the preceding reduction satisfies properties (i) and (ii). For property (i), we construct the flow as follows. For each edge $(\langle \alpha, parent(v) \rangle, v)$, we set the flow to 1 if $copies(P, \alpha, v) > 0$, and to 0 otherwise. For each unit-capacity edge $(\langle \alpha, parent(w) \rangle, \langle \alpha, w \rangle)$, we set the flow to 1 if $copies(P, \alpha, w) > 0$, and to 0 otherwise. For each infinite-capacity edge $(\langle \alpha, parent(w) \rangle, \langle \alpha, w \rangle)$, we set the flow to $\max\{0, copies(P, \alpha, w) - 1\}$. For each edge $(a, \langle \alpha, u \rangle)$, we set the flow to 1 if $copies(P, \alpha, u) > 0$, and to 0 otherwise. For each edge (v, b), we set the flow to the sum over all objects α of $copies(P, \alpha, v)$. It is easy to prove that the above flow is feasible and has cost equal to the cost of P minus C.

It remains to prove property (ii). Given a minimum-cost flow, we can define a corresponding u-placement P as follows: Place a copy of object α in the trivial hierarchy v if and only if the flow along edge $(\langle \alpha, parent(v) \rangle, b)$ is 1. Clearly this placement satisfies the space constraint at each machine. It remains to establish that the cost of the placement is $C + C'$. To establish this, we make the following crucial observation regarding the flows on adjacent parallel arcs. Let e_1 and e_2 denote the parallel edges with capacities 1 and ∞, respectively, between a pair of vertices. If the flow along e_2 is positive, then the flow along e_1 is 1. Otherwise, transferring a unit of flow from e_2 to e_1 would yield another feasible flow with smaller cost, a contradiction. This implies that whenever $copies(P, \alpha, w) > 0$ for some proper descendant w of u, then the flow along edge $(\langle \alpha, parent(w) \rangle, \langle \alpha, w \rangle)$ is 1, and hence there is an appropriate negative contribution to the cost of the flow.

4 A simple approximation algorithm

While the algorithm of Section 3 obtains an optimal solution to the hierarchical placement problem, its runtime complexity is prohibitively high, at least quadratic in the product of the number of machines n and the number of objects m. This motivates us to seek a faster approximation algorithm. Two candidate algorithms are presented in this section. The first algorithm, which we refer to as the greedy algorithm, uses a natural local improvement heuristic. We show, however, that the greedy algorithm has an approximation ratio of $\Omega(n)$. The lower bound proof leads us to a variant of the greedy algorithm, which we refer to as the amortizing algorithm. Section 5 establishes that the amortizing algorithm is a constant-factor approximation algorithm. Section 6 outlines an efficient distributed implementation of the amortizing algorithm.

The greedy and amortizing algorithms each use a bottom-up phase followed by a top-down phase to compute a placement. The bottom-up phase yields a "pseudoplacement", a generalization of a placement that is defined in Section 4.1. The top-down phase, which is the same for both algorithms, converts this pseudoplacement to a placement.

The greedy and amortizing algorithms, though well-defined for arbitrary hierarchies, are only intended to be directly applied to hierarchies that are "λ-separated" for some constant $\lambda > 1$. A hierarchy u is *λ-separated* if and only if $miss(v) \geq \lambda \cdot diam(v)$ for every descendant v of u. It is straightforward to show that any c-approximation algorithm for the λ-separated hierarchical placement problem implies a $c\lambda$-approximation algorithm for the hierarchical placement problem. (The main idea is to transform any hierarchy into a λ-separated hierarchy by rounding up each diameter to the nearest integral power of λ and identifying adjacent nodes with the same diameter.) Thus, for the purposes of obtaining a constant-factor approximation algorithm, we may assume without loss of generality that the input hierarchy is λ-separated for some constant $\lambda > 1$.

The rest of this section is organized as follows. Section 4.1 defines the notion of a pseudoplacement, derives some useful technical machinery, and describes the top-down phase of the two algorithms. Section 4.2 and 4.3 define the bottom-up phase of the greedy and amortizing algorithms, respectively.

4.1 Pseudoplacements. We now extend the notion of a placement to that of a pseudoplacement, by allowing copies of objects to be located at hierarchies, and not just at machines. In order to formally define a pseudoplacement, we first modify the definition of an item: An item is a pair (α, u), where α is an object and u is a hierarchy. In order to accommodate our earlier notion of an item, we identify the pairs (α, σ) and $(\alpha, \{\sigma\})$ for any object α and machine σ. For any set of items S, object α, and hierarchy u, we define $\mathit{copies}(S, \alpha, u)$ as the number of items in S of the form (α, v) where v is some descendant of u. For any hierarchy u, a *u-pseudoplacement* Q is a set of items, each of which is of the form (α, v) for some descendant v of u, such that the following space constraint is satisfied for all descendants w of u: the sum over all objects α of $\mathit{copies}(Q, \alpha, w)$ is at most $\mathit{size}(w)$. A *pseudoplacement* is a u-pseudoplacement for some hierarchy u. Note that any u-placement (resp., placement) is a u-pseudoplacement (resp., pseudoplacement).

For any object α, u-pseudoplacement Q, and machine σ in $\mathit{machines}(u)$, the cost of an access to α at σ, given by $c(Q, \sigma, \alpha)$, equals $\mathit{miss}(u)$ if $\mathit{copies}(Q, \alpha, u) = 0$. If $\mathit{copies}(Q, \alpha, u) > 0$, then $c(Q, \sigma, \alpha)$ is defined as $\mathit{diam}(v)$, where v is the least-diameter hierarchy such that σ belongs to $\mathit{machines}(v)$ and $\mathit{copies}(Q, \alpha, v) > 0$. The cost of Q is then defined as the sum, over all machines σ and objects α, of $f(\sigma, \alpha) \cdot c(Q, \sigma, \alpha)$. (Note that the above definitions agree with those given in Section 2.1 for the special case in which Q is a placement.)

We say that a pseudoplacement Q' is a *refinement* of another pseudoplacement Q if there is a one-to-one mapping ϕ from Q to Q' such that for any item x in Q: (i) x and $\phi(x)$ are copies of the same object, and (ii) the hierarchy at which $\phi(x)$ is located by Q' is a descendant of the hierarchy at which x is located by Q. It is easy to see from the definitions of refinement and cost that if Q' refines Q then the cost of Q' is at most the cost of Q. Any arbitrary u-pseudoplacement Q can be refined to a u-placement by a simple top down procedure as follows. (This procedure forms the top-down phase of the greedy and amortizing algorithms.) If there is an item of Q located at u, then, by the space constraints, there exists a child v of u in which the sum over all objects α of $\mathit{copies}(Q, \alpha, v)$ is less than $\mathit{size}(v)$. We pick an arbitrary item (α, u) from Q and replace it by (α, v), thus reducing the number of items located at u. By a repeated application of this step, first at u and then recursively at the children of u, the u-pseudoplacement Q can be refined to a u-placement. (We remark that for any pseudoplacement, a minimum-cost refined placement can be obtained by solving a suitably defined weighted matching problem.)

We now define a swapping procedure that is used in all of our bottom-up pseudoplacement algorithms. First, we need some additional notation. For any u-pseudoplacement Q, we define a *Q-item* (resp., *Q-copy*) as an item (resp., copy) belonging to Q. An object α is *Q-missing* if and only if $\mathit{copies}(Q, \alpha, u) = 0$. For any Q-missing object α, we define the *Q-value* of α as the cost of the empty u-pseudoplacement minus the cost of the u-pseudoplacement $\{(\alpha, u)\}$.

The Swapping Procedure: $\mathit{Swap}(u, Q, \alpha, x)$

- The input is a u-pseudoplacement Q, a Q-missing object α, and a Q-item x.
- Remove x from Q, that is, set Q to $Q \setminus \{x\}$.
- The previous step ensures that $|Q|$ is less than $\mathit{size}(u)$. Hence we can set Q to $Q \cup \{(\alpha, u)\}$ without violating any of the space constraints.

4.2 The greedy algorithm. For any Q-copy x of object α, we define the *Q-min-benefit* of x as the amount by which the cost of Q would increase if x were removed from Q. Given a nonempty pseudoplacement Q, the *greedy elimination rule* removes from Q the item with the least Q-min-benefit; we call this item the *Q-victim*.

Given a pseudoplacement Q containing one or more copies of object α, the *greedy α-elimination rule* removes from Q the copy of α with the least min-benefit. We call this item the *(Q, α)-victim*. For any Q-copy x of object α, let Q' denote the last pseudoplacement containing x in the sequence of pseudoplacements obtained by repeated application of the greedy α-elimination rule, starting from Q. We define the *Q-benefit* of x as the Q'-min-benefit of x. We remark that the Q-min-benefit of x is at most the Q-benefit of x. Furthermore, the Q-benefit of the Q-victim (resp., (Q, α)-victim) is equal to its Q-min-benefit. It follows that the greedy elimination rule (resp., greedy α-elimination rule) can be implemented by removing the Q-copy of object α with the least Q-benefit. For any pseudoplacement Q, the *Q-candidate* is the Q-missing object with the highest Q-value. If there are no Q-missing objects, it is convenient to define the Q-value of the Q-candidate as an arbitrary negative value.

We now present the bottom-up greedy algorithm. We give separate descriptions of the computations performed at the trivial and nontrivial hierarchies.

Greedy: Trivial Hierarchy u

- Any trivial hierarchy u is a singleton set containing some machine σ. We construct an optimal pseudoplacement Q for u as follows: For each of the $size(\sigma)$ objects α most frequently accessed by σ, we add the item (α, u) to Q.

Greedy: Nontrivial Hierarchy u

- Let Q_i be the u_i-pseudoplacement previously computed at u_i, the ith child of u. We initialize Q to the u-pseudoplacement equal to the union of Q_i's.
- While the Q-benefit of the Q-victim is at most the Q-value of the Q-candidate, call *Swap*$(u, Q, Q$-candidate$, Q$-victim$)$.

In Appendix A, we present a simple example showing that the approximation ratio of the greedy algorithm is $\Omega(n)$. The example is instructive in that it leads us to a natural variant of the greedy pseudoplacement algorithm, the amortizing pseudoplacement algorithm.

4.3 The amortizing algorithm. A Q-copy of object α is *Q-primary* if and only if it is the last copy of object α to be removed from Q under repeated application of the greedy α-elimination rule; otherwise, it is *Q-secondary*. For any pseudoplacement Q, the *Q-secondary-victim* is the Q-secondary item with the least Q-benefit. If there are no Q-secondary items, it is convenient to define the Q-benefit of the Q-secondary-victim as $+\infty$.

Amortizing: Trivial Hierarchy u

- As in the greedy algorithm, except that we also set the potential Φ to zero.

Amortizing: Nontrivial Hierarchy u

- **Combining.** Let the children of u be $u_1, \ldots, u_k$. This step is the same as the combining step in the greedy algorithm, except that we also initialize two variables related to the potential function. The potential Φ is initialized to the sum of the potentials $\Phi_1, \ldots, \Phi_k$ computed by the k children of u. The auxiliary variable Δ is initialized to the sum of the Q-values of all Q-missing objects.
- **Amortized Swapping.** This procedure is similar to the main loop of the greedy algorithm, except that the potential Φ is used to reduce the benefits of certain secondary items: (i) let a denote the Q-benefit of the Q-secondary-victim minus Φ, let b denote the Q-benefit of the Q-victim, and let c denote the Q-value of the Q-candidate; (ii) if $a \le \min(b, c)$ then call *Swap*$(u, Q, Q$-candidate$, Q$-secondary-victim$)$, subtract c from Δ, set Φ to $\max(0, -a)$, and goto (i); (iii) if $b \le c$ then call *Swap*$(u, Q, Q$-candidate$, Q$-victim$)$, add $b - c$ to Δ, and goto (i).
- **Update Potential.** Add Δ to Φ.

5 Analysis of the amortizing algorithm

In this section we prove our main result, namely, that the cost of the pseudoplacement constructed by the amortizing algorithm is within a constant factor of optimal. To facilitate our analysis, we introduce another bottom-up pseudoplacement algorithm that we call the bridging algorithm. The bridging algorithm is not an algorithm in the usual sense, but rather a family of algorithms parameterized by an arbitrary placement P. The high-level plan of our analysis is to prove the following two claims: (i) no choice of the choice of placement parameter P allows the P-bridging algorithm to outperform the amortizing algorithm, and (ii) the cost of any placement P is at least a constant fraction of the cost of the pseudoplacement computed by the P-bridging algorithm.

5.1 The P-bridging algorithm. For any object α, u-pseudoplacement Q, and Q-copy x of α, we define the *Q-region* of x as the set of all machines σ in *machines*(u) such that $c(Q, \sigma, \alpha)$ would increase if x were removed from Q. For any u-pseudoplacement Q and u-placement P, we define a Q-copy x of object α to be *(Q, P)-min-matched* if and only if P has a copy of α in the Q-region of x. Otherwise, x is *(Q, P)-min-unmatched.* Let Q be a u-pseudoplacement containing one or more copies of object α, and assume that P is a u-placement. Let A denote the set of all (Q, P)-min-unmatched copies of α. If A is nonempty, the *bridging (P, α)-elimination rule* applied to Q removes the item in A with the least Q-min-benefit; otherwise, it removes the $((Q, P)$-min-matched) copy of α with the least Q-min-benefit. We call this item the *(Q, P, α)-quasivictim.*

The P-Bridging Algorithm

- Let P be a u-placement. For any descendant v of u, the P-bridging algorithm computes a pair of v-pseudoplacements $(Q, \tilde{Q})$. If $(Q, \tilde{Q})$ denotes the pair of u-pseudoplacements computed for u, then the final output of the P-bridging algorithm is $\tilde{Q}$.
- For any descendant v of u, the v-pseudoplacement Q is computed as in the amortizing algorithm. At a trivial descendant v of u, the v-pseudoplacement $\tilde{Q}$

is equal to the v-pseudoplacement Q. At a nontrivial descendant v of u, $\tilde{Q}$ is initialized to the union of the $\tilde{Q}_i$'s computed at the children, just as Q is initialized to the union of the Q_i's. The amortizing algorithm then proceeds to update Q via a sequence of calls to the swapping procedure. Suppose that a call of the form $\mathit{Swap}(v, Q, \alpha, x)$ is made to update Q, and let x be a copy of object β. Then the P-bridging algorithm makes a corresponding call of the form $\mathit{Swap}(v, \tilde{Q}, \alpha, (\tilde{Q}, R, \beta)$-quasivictim) to update $\tilde{Q}$, where $R = \mathit{restrict}(P, v)$.

For any u-placement P, u-pseudoplacement Q, and Q-copy x of object α, let Q' denote the last u-pseudoplacement containing x in the sequence of u-pseudoplacements obtained by repeated application of the bridging (P, α)-elimination rule, starting from Q. We define the (Q, P)-*quasibenefit* of x to be the (Q', P)-min-benefit of x, and we say that x is (Q, P)-*matched* (resp., (Q, P)-*unmatched*) if and only if it is (Q', P)-min-matched (resp., (Q', P)-min-unmatched). We remark that the Q-min-benefit of x is at most the (Q, P)-quasibenefit of x. Furthermore, the (Q, P)-quasibenefit of the (Q, P, α)-quasivictim is equal to its Q-min-benefit. It follows that the bridging (P, α)-elimination rule can be implemented by removing the item (resp., Q-copy of object α) with the least Q-quasibenefit.

5.2 The amortizing algorithm is as good as the P-bridging algorithm. In order to analyze the P-bridging algorithm, and to relate its performance to that of the amortizing algorithm, it turns out to be useful to compare the Q-benefit of a given item x removed by a swap updating Q with the $(\tilde{Q}, P)$-quasibenefit of the item y removed by the corresponding swap updating $\tilde{Q}$. Lemma 5.5 below shows that if y is $(\tilde{Q}, P)$-matched (resp., $(\tilde{Q}, P)$-unmatched), then the Q-benefit of x is greater (resp., less) than or equal to the $(\tilde{Q}, P)$-quasibenefit of y. Before establishing Lemma 5.5, we state a number of useful definitions and lemmas related to multisets. Lemmas 5.1 through 5.4 are straightforward and are stated without proof.

For any multiset of reals A, we define $\mathit{Sum}(A)$ as the sum of the elements of A. For any multiset of reals A, and any integer i such that $0 \le i \le |A|$, we define $\mathit{Big}(A, i)$ (resp., $\mathit{Little}(A, i)$) as the multiset consisting of the i largest (resp., smallest) reals in A. For any two multisets of reals A and B such that $|A| = |B|$, we write $A \le B$ to mean that the ith largest element of A is less than or equal to the ith largest element of B, $1 \le i \le |A|$. We define a triple (A, B, C) to be *good* if and only if the following conditions hold: (i) A, B, and C are finite multisets of reals, (ii) $|A| + |B| = |C|$, (iii) $A \ge \mathit{Little}(C, |A|)$, (iv) $B \le \mathit{Big}(C, |B|)$, and (v) $\mathit{Sum}(A) + \mathit{Sum}(B) \le \mathit{Sum}(C)$.

LEMMA 5.1. *For any real a, the triples $(\{a\}, \{\}, \{a\})$ and $(\{\}, \{a\}, \{a\})$ are good.* ■

LEMMA 5.2. *Let (A, B, C) be good and assume that $C' = \mathit{Big}(C, |C|-1)$. If A is empty, then let $A' = A$ and $B' = \mathit{Big}(B, |B|-1)$. Otherwise, let $A' = \mathit{Big}(A, |A|-1)$ and $B' = B$. In either case, (A', B', C') is good.* ■

LEMMA 5.3. *Assume that (A_i, B_i, C_i) is good, $1 \le i \le k$, and let $A = \cup_{1 \le i \le k} A_i$, $B = \cup_{1 \le i \le k} B_i$, and $C = \cup_{1 \le i \le k} C_i$. Then (A, B, C) is good. Furthermore, if B is empty and a is the maximum element of A, then $(A \setminus \{a\}, \{a\}, C)$ is good.* ■

LEMMA 5.4. *Let (A, B, C) be good and let d be a nonnegative real. Assume that c is a maximum element of C and let $C' = (C \setminus \{c\}) \cup \{c + d\}$. If B is nonempty, then let $A' = A$ and $B' = (B \setminus \{b\}) \cup \{b + d\}$ where b is a maximum element of B. Otherwise, let $B' = B$ and $A' = (A \setminus \{a\}) \cup \{a + d\}$ where a is a maximum element of A. In either case, (A', B', C') is good.* ■

LEMMA 5.5. *Let α be an arbitrary object, and let $(Q, \tilde{Q})$ denote the current pseudoplacements at some point in the execution of the P-bridging algorithm. Let A (resp., B) denote the multiset of the $(\tilde{Q}, P)$-quasibenefits of the $(\tilde{Q}, P)$-unmatched (resp., $(\tilde{Q}, P)$-matched) copies of α in $\tilde{Q}$. Let C denote the multiset of the Q-benefits of the copies of α in Q. Then (A, B, C) is good.*

Proof Sketch: We argue that for each change to the pseudoplacement Q, the corresponding change to the pseudoplacement $\tilde{Q}$ preserves the invariant that (A, B, C) is good. There are three ways in which the pseudoplacement Q can change: (i) an item is added, (ii) an item is removed, and (iii) the pseudoplacements computed at the children are combined. When an item is added, the same item is added to $\tilde{Q}$, and Lemma 5.1 can be used to show that the invariant continues to hold. When a copy of some object is removed, a copy of the same object is removed from $\tilde{Q}$, and Lemma 5.2 can be used to show that the invariant continues to hold. When the pseudoplacements Q_i computed at the children are combined to form Q, the pseudoplacements $\tilde{Q}_i$ are combined to form $\tilde{Q}$, and Lemmas 5.3 and 5.4 can be used to show that the invariant continues to hold. (Lemma 5.4 takes care of the increase in benefit/quasibenefit associated with the primary/quasiprimary copy of each object.) ■

LEMMA 5.6. *For any u-placement P, the cost of the u-pseudoplacement produced by the amortizing algorithm is no higher than the cost of the u-pseudoplacement produced by the P-bridging algorithm.*

Proof: The claim follows by applying Lemma 5.5 to the final pseudoplacements produced by the two algorithms, and making use of condition (v) in the definition of a good triple. ■

5.3 The P-bridging pseudoplacement is nearly as good as P. In Appendix B, we sketch the proof of the following key lemma.

LEMMA 5.7. *For any λ-separated hierarchy u and u-placement P, the cost of the u-pseudoplacement computed by the P-bridging algorithm is at most $(1+3\cdot\frac{\lambda}{\lambda-1})$ times the cost of P.* ■

Taking P to be an optimal u-placement, we conclude that the P-bridging algorithm computes a u-pseudoplacement with cost at most a constant factor times optimal.

5.4 The main theorem. Using Lemmas 5.6 and 5.7, together with the fact that the top-down refinement pass of the amortizing algorithm introduces no additional cost, we now obtain the following result.

LEMMA 5.8. *For any λ-separated hierarchy u, the cost of the u-placement computed by the amortizing algorithm is at most $(1+3\cdot\frac{\lambda}{\lambda-1})$ times optimal.* ■

Lemma 5.8 assumes that the given hierarchy u is λ-separated as defined in Section 4. If this is not the case, we first transform the input hierarchy into a λ-separated hierarchy as described in Section 4, introducing an extra factor of λ into the approximation bound.

THEOREM 5.1. *For any hierarchy u, the cost of the u-placement computed by the amortizing algorithm is at most $\lambda\cdot(1+3\cdot\frac{\lambda}{\lambda-1})$ times optimal.* ■

The above approximation ratio is less than 13.93 for the optimal choice of $\lambda = 1+\sqrt{3}/2 \approx 1.866$.

6 An efficient distributed implementation

In this section we briefly sketch a fast distributed implementation of the amortizing algorithm. The amortizing algorithm determines a pseudoplacement for the given hierarchy u in a bottom-up manner. During the computation, the current pseudoplacement, along with associated control information, is distributed among the machines of the network. For each object α and descendant v of the given hierarchy, we designate a machine in *machines*(v) as the *manager* for α in v, denoted *manager*(v,α). For load balancing purposes, this manager is chosen at random from *machines*(v), where the probability of choosing a particular machine σ is *size*(σ)/*size*(v). For each descendant v, we choose a machine uniformly at random from *machines*(v) and designate it as the *leader* of v. The current pseudoplacement is distributed as follows. Let Q be a u-pseudoplacement. For a descendant v, if $x = (\alpha, v)$ is in Q, then *manager*(v,α) stores the following information: (i) a bit indicating that a copy of α is placed, (ii) a bit indicating whether this copy is a primary, and (iii) the benefit of this copy. If no copy of α is placed in v, then *manager*(v,α) maintains the value of the object α, which equals $f(v,\alpha)$*miss*(v). The leader of u maintains the potential that is calculated in the combining phase of the amortizing algorithm.

Let us now consider the process of computing a u-pseudoplacement Q. Let u_i denote the ith child of u. Let Q_i denote the u_i-pseudoplacement computed by the amortizing algorithm. The amortizing algorithm proceeds in three phases: combining, amortized swapping, and potential updating. In the combining phase, Q is set to the union of the Q_i's. The benefit of the primary copy of an object α in u is calculated by the machine *manager*(u,α) and the machine at which the primary copy is placed. Also, the calculation of the sum of the potentials associated with the child hierarchies is implemented by the leader of u and the leaders of the u_i's.

We implement the amortized swapping phase as an amortization subphase followed by a swapping subphase. Let X denote the list of secondaries in Q sorted according to the benefits in nonincreasing order. Let Φ denote the value of the potential after the combining phase. In the amortization subphase, we determine the largest prefix X' of X such that the sum of the benefits of secondaries in X' is at most Φ. (The items in X' are guaranteed to be swapped out during the swapping subphase.) Thus, the amortization subphase corresponds to a selection problem. Similarly, the swapping subphase corresponds to the following abstract selection problem. Let X and Y denote two lists of numbers sorted in nondecreasing and nonincreasing orders, respectively. (The lists X and Y correspond to the benefits of the copies in the pseudoplacement, and the values of the objects missing from the pseudoplacement, respectively.) Our goal is to determine a largest prefix X' of X and a prefix Y' of Y such that $|X'|$ equals $|Y'|$ and no element of X' is greater than any element of Y'.

While the amortization and swapping subphases are straightforward to perform sequentially, in the current setting the lists are distributed among the machines and are hence not available in a sorted order. We now briefly describe an efficient distributed recursive implementation of the amortization subphase; a similar approach can be used for the swapping subphase. If $|X| \le 1$, then the problem is trivial. Otherwise, we first select a splitter among the secondaries in X. Second, we determine the set Z of those secondaries in X with benefit at most that of the splitter. Third, we sum up the benefits of the secondaries in Z. Finally, in a manner that depends on whether the sum exceeds the given potential Φ, we define a new instance of the problem with fewer elements and recurse. Standard probabilistic arguments can be used to establish a logarithmic bound on the depth of recursion. Moreover, each of the level of the recursion can be implemented using simple distributed operations such as broadcast and sum.

The potential updating phase is straightforward to implement efficiently. In the computation of the pseudoplacement for a hierarchy u, the combining phase requires a constant number of messages per primary stored in the hierarchy. The amortized swapping and the potential updating phases each require $O(\log(size(u)))$ prefix sum operations. The top-down pass used to convert the pseudoplacement to a placement is easy to implement within the same complexity bound as the bottom-up pass.

References

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

[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] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 37:1021–1058, 1990.

[4] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, 1996.

[5] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 161–168, May 1998.

[6] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 39–50, May 1992.

[7] M. A. Blaze. Caching in large-scale distributed file systems. Technical Report TR-397-92, Department of Computer Science, Princeton University, January 1993. PhD Thesis.

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

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

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

[11] M. Charikar, C. Chekuri, A. Goel, and S. Guha. Rounding via trees: Deterministic approximation algorithms for group steiner trees and k-median. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 106–113, May 1998.

[12] 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.

[13] D. Dowdy and D. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14:287–313, 1982.

[14] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Computer Communications Review*, October 1998.

[15] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 51–57, May 1995.

[16] A. Heddaya and S. Mirdad. WebWave: Globally load balanced fully distributed caching of hot published documents. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 160–168, May 1997.

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

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

[19] 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.

[20] C. Lund, N. Reingold, J. Westbrook, and D. Yan. On-line distributed data management. In J. van Leeuwen, editor, *Proceedings of the 2nd Annual European Symposium on Algorithms*, Lecture Notes in Computer Science, volume 855, pages 202–214. Springer-Verlag, 1994.

[21] 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.

[22] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, June 1997.

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

[24] M. van Steen, F. J. Hauck, and A. S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of TINA '96*, pages 203–212, September 1996.

[25] D. Wessels. Squid Internet object cache. Available at URL http://squid.nlanr.net/Squid, January 1998.

[26] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2, request for comments rfc–2187. Available at URL http://ds.internic.net/rfc/rfc2186.txt, September 1997.

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

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

A Lower bound for the greedy algorithm

We construct a depth-$(n-1)$ hierarchy with n machines, for which the pseudoplacement computed by the greedy algorithm has a cost of $\Theta(n)$ times the optimal.

We label the n machines from 0 to $n-1$ and construct the hierarchy as follows. First, we create a depth-1 hierarchy with two children, machines 0 and 1. Then, for i running from 2 to $n-1$, we create a depth-i hierarchy with two children: (i) the depth-$(i-1)$ hierarchy constructed previously, and (ii) a trivial hierarchy containing machine i. The cache size of each machine is 1. The depth-i hierarchy has diameter n^{i-1} and miss penalty n^i, $1 \leq i < n$. Finally, we specify the frequency function. There are n objects α_i, $0 \leq i < n$.

For machine 0, $f(0, \alpha_i)$ equals $1/n^i$ for $0 \le i < n$. For machine 1, $f(1, \alpha_i)$ is 1 if i is 0, and 0 otherwise. For machine j, $2 \le j < n$, $f(j, \alpha_i)$ is 0 for all i.

For this example, the optimal placement puts a copy of α_0 at machine 1, a copy of α_1 at machine 0, and a copy of α_i at machine i for $2 \le i < n$. The cost of the optimal placement P is $2 - 2/n$ since $c(P, 0, \alpha_i)$ is 1 if i is 0, 0 if i is 1, and n^{i-1} otherwise. We claim that the cost of the greedy pseudoplacement is $n - 1$. (Furthermore, any refinement of the greedy pseudoplacement also has cost $n - 1$.) It can be shown by induction that the pseudoplacement obtained for the depth-i hierarchy, $1 \le i < n$, places a copy of α_0 at machines 0 and 1, and a copy of α_j at the depth-j hierarchy, $1 \le j \le i$. The cost of the final pseudoplacement Q is $n - 1$ since $c(Q, 0, \alpha_i)$ is n^i for $1 \le i < n$.

B Sketch of the proof of Lemma 5.7

In this section we sketch the high-level structure of our proof of Lemma 5.7. The complete proof is deferred to the full version of the paper.

Let u denote a λ-separated hierarchy for some constant $\lambda > 1$, and let P denote a u-placement. Our goal is to prove that the cost of the u-pseudoplacement computed by the P-bridging algorithm is at most c times the cost of P, where $c = (1 + 3 \cdot \frac{\lambda}{\lambda - 1})$. Recall that for each descendant v of u, the P-bridging algorithm computes a pair of v-pseudoplacements $(Q, \tilde{Q})$, where Q is the v-pseudoplacement computed by the amortizing algorithm. As in the P-bridging algorithm, let R denote the v-placement $\mathit{restrict}(P, v)$.

We break the computation at v into a sequence of atomic steps. Each iteration of the amortized swapping loop is an atomic step; the combining and potential update operations are also atomic steps. Note that some of the atomic steps modify Q and $\tilde{Q}$, but R remains fixed throughout the computation at v. We will prove that at the end of the computation at v, the cost of $\tilde{Q}$ is at most c times that of R. To facilitate this analysis, we introduce an additional v-pseudoplacement R'. At the beginning of the computation at a trivial hierarchy v, we set R' equal to R. In the combining atomic step initiating the computation at a nontrivial hierarchy v, we set R' to the union of the R_i''s computed at the children of v, just as Q (resp., $\tilde{Q}$, R) is set to the union of the Q_i's (resp., $\tilde{Q}_i$'s, R_i's). But for each object α such that $\mathit{copies}(R, \alpha, v) > 0$ and $\mathit{copies}(R', \alpha, v) = 0$, we add the item (α, v) to R'.

Like Q and $\tilde{Q}$, the v-pseudoplacement R' may be modified by the sequence of atomic steps executed at v. The modifications to R' are designed in part to ensure that at the end of the computation at v, $\tilde{Q}$ dominates R' in the following sense: For any item (α, w) in R', $\mathit{copies}(\tilde{Q}, \alpha, w) > 0$. It follows that at the end of the computation at v, the cost of $\tilde{Q}$ is at most that of R'. Thus it is sufficient to prove that the cost of R' is at most c times that of R.

While $\tilde{Q}$ dominates R' at the end of the computation at v, this property does not necessarily hold after each atomic step at v. For example, it need not hold after the combining step, since we might have added an item (α, v) to R' for which $\mathit{copies}(\tilde{Q}, \alpha, v) = 0$. Furthermore, as the computation at v proceeds, some items are swapped into $\tilde{Q}$ while others are swapped out. Swapping items out of $\tilde{Q}$ might necessitate further changes to R'. We reestablish the invariant that $\tilde{Q}$ dominates R' by removing from R' all items (α, w) such that $\mathit{copies}(\tilde{Q}, \alpha, w) = 0$ at the end of the computation at v. Some of these removals are performed during atomic steps within the amortized swapping loop, and the remainder are performed during the potential update atomic step.

Removing an item from R' results in a nonnegative increase in the cost of R'. Since our goal is to bound the cost of R' at the end of the computation at v by c times that of R, we need to bound the sum of these cost increases in terms of the cost of R. For each such increase δ', we prove the existence of an object α and a descendant w of v such that $\mathit{copies}(R, \alpha, w) = 0$ and $\delta = f(w, \alpha) \cdot \mathit{miss}(w) \ge \delta'$. It follows that the cost of R includes a term (corresponding to the cost of satisfying requests for α within w) that offsets the cost increase δ'. Of course, we need to ensure that a given term contributing to the cost of R is not used to offset an arbitrarily large number of increases in the cost of R'. To do so, we exhibit costs incurred by R in a systematic manner that avoids using the same cost term more than a constant number of times. Our analysis bounds the overall effective factor of reuse by $c - 1$. It follows that the final cost of R' is at most c times that of R.

The preceding proof sketch is high-level and informal. In the formal proof, we characterize the state of the computation at a given hierarchy v in terms of Q, $\tilde{Q}$, R, R', the potential-related program variables Φ and Δ, as well as a number of additional auxiliary variables. We then specify the effect of each atomic step on the state, and list a number of invariants that hold at the end of each atomic step. Though somewhat tedious, it is straightforward to prove that each of these claimed invariants is in fact an invariant, since the change in state associated with any atomic step is relatively minor. Lemma 5.7, which makes a claim about the state after completion of the computation at u, is an immediate consequence of our invariants.