Online Hierarchical Cooperative Caching

Xiaozhou Li[1,2] C. Greg Plaxton[1,2] Mitul Tiwari[1,3] Arun Venkataramani[1,4]

ABSTRACT

We address a hierarchical generalization of the well-known disk paging problem. In the hierarchical cooperative caching problem, a set of n machines residing in an ultrametric space cooperate with one another to satisfy a sequence of read requests to a collection of (read-only) files. A seminal result in the area of competitive analysis states that LRU (the widely-used deterministic online paging algorithm based on the "least recently used" eviction policy) is constant-competitive if it is given a constant-factor blowup in capacity over the offline algorithm. Does such a constant-competitive deterministic algorithm (with a constant-factor blowup in the machine capacities) exist for the hierarchical cooperative caching problem? The main contribution of the present paper is to answer this question in the negative. More specifically, we establish an $\Omega(\log \log n)$ lower bound on the competitive ratio of any online hierarchical cooperative caching algorithm with capacity blowup $O((\log n)^{1-\varepsilon})$, where ε denotes an arbitrarily small positive constant.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems - Computations on Discrete Structures; C.2.4 [**Computer Communication Networks**]: Distributed Systems - Distributed Applications

General Terms

Algorithms, Performance

[1]Department of Computer Science, University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712–0233. Email: {*xli,plaxton,mitult,arun*}*@cs.utexas.edu.*

[2]Supported by NSF Grant CCR–0310970.

[3]Supported by NSF Grant ANI–0326001.

[4]Supported by Texas Advanced Technology Project 003658–0503–2003 and by IBM.

SPAA'04, June 27–30, 2004, Barcelona, Spain.

Keywords

Online computation, hierarchical cooperative caching

1. INTRODUCTION

The traditional *paging* problem, which has been extensively studied, is defined as follows. Given a cache and a sequence of requests for files of uniform sizes, a system has to satisfy the requests one by one. If the file f being requested is in the cache, then no cost is incurred; otherwise a uniform retrieval cost is incurred to place f in the cache. If need be, some files, determined by an online caching algorithm that does not know the future request sequence, are evicted to make room for f. The objective is to minimize the total retrieval cost by wisely choosing which files to evict. The cost of the online algorithm is compared against that of an optimal offline algorithm (OPT) that has full knowledge of the request sequence. Following Sleator and Tarjan [10], we call an online algorithm c-competitive if its cost is at most c times that of OPT for any request sequence. It is well-known that an optimal offline strategy is to evict the file that will be requested furthest in the future.

The paging problem is also known as *caching* if the files have nonuniform size and retrieval cost. In their seminal paper, Sleator and Tarjan [10] have shown that LRU (Least-Recently-Used) and several other deterministic paging algorithms are $\frac{k}{k-h+1}$-competitive, where k is the cache space used by LRU and h is that used by OPT. They have also shown that $\frac{k}{k-h+1}$ is the best possible among all deterministic algorithms. We call $\frac{k}{h}$ the *capacity blowup* of LRU. For files of nonuniform size and retrieval cost, Young [13] has proposed the LANDLORD algorithm and shown that LANDLORD is $\frac{k}{k-h+1}$-competitive. As stated in [13], the focus of LANDLORD "is on simple *local* caching strategies, rather than distributed strategies in which caches cooperate to cache pages across a network".

In cooperative caching [6], a set of caches cooperate in serving requests for each other and in making caching decisions. The benefits of cooperative caching have been supported by several studies. For example, the Harvest cache [5] introduce the notion of a hierarchical arrangements of caches. Harvest uses the Internet Cache Protocol [12] to support discovery and retrieval of documents from other caches. The Harvest project later became the public domain Squid cache system [11]. Adaptive Web Caching [14] builds a mesh of overlapping multicast trees; the popular files are pulled down towards their users from their origin server. In local-area network environments, the xFS system [1] utilizes cooperative caches to obtain a serverless file system.

A cooperative caching scheme can be roughly divided into three components: *placement*, which determines where to place copies of files, *search*, which directs each request to an appropriate copy of the requested file, and *consistency*, which maintains the desired level of consistency among the various copies of a file. In this paper, we study the placement problem, and we assume that a separate mechanism enables a cache to locate a nearest copy of a file, free of cost, and we assume that files are read-only (i.e., copies of a file are always consistent). We focus on a class of networks called *hierarchical networks*, the precise definition of which is given in Section 2, and we call the cooperative caching problem in such networks the *hierarchical cooperative caching (HCC) problem*.

Our notion of a hierarchical network is constant-factor related to the notion of hierarchically well-separated tree metrics, as introduced by Bartal [3]. Refining earlier results by Bartal [3], Fakcharoenphol *et al.* [7] have shown that any metric space can be approximated by well-separated tree metrics with a logarithmic distortion. Hence, many results for tree metrics imply corresponding results for arbitrary metric spaces with an additional logarithmic factor.

If the access frequency of each file at each cache is known in advance, Korupolu *et al.* [9] have provided both exact and approximation algorithms that minimize the average retrieval cost. In practice, such access frequencies are often unknown or are too expensive to track. Since LRU and LANDLORD provide constant competitiveness for a single cache, it is natural to ask whether there exists a deterministic constant-competitive algorithm (with constant capacity blowup) for the hierarchical cooperative caching problem.

In this paper, we answer this question in the negative. We show that $\Omega(\log\log n)$ is a lower bound on the competitive ratio of any deterministic online algorithm with capacity blowup $O((\log n)^{1-\varepsilon})$, where n is the number of caches in the hierarchy and ε is an arbitrarily small positive constant. In particular, we construct a hierarchy with a sufficiently large depth and show that an adversary can generate an arbitrarily long request sequence such that the online algorithm incurs a cost $\Omega(\log\log n)$ times that of the adversary. Interestingly, the offline algorithms associated with our lower bound argument do not replicate files.

On the other hand, if an online algorithm is given a sufficiently large capacity blowup, then constant competitiveness can be easily achieved. Appendix A shows a simple result that, given $(1+\varepsilon')d$ capacity blowup, where d is the depth of the hierarchy (i.e., $d = \Theta(\log n)$) and ε' an arbitrarily small positive constant, a simple LRU-like online algorithm is constant-competitive. Note that in terms of d, our lower bound result yields that if the capacity blow up is $O(d^{1-\varepsilon})$, then the competitive ratio is $\Omega(\log d)$. Hence, our results imply that there is a very small range of values of the capacity blowup that separates the regions where constant competitiveness is achievable and unachievable.

Drawing an analogy to traditional caching, where LRU and LANDLORD provide constant competitiveness, we may think that a constant-competitive algorithm exists for HCC, being perhaps a hierarchical variant of LRU or LANDLORD. In fact, we began our investigation by searching for such an algorithm. Since the HCC problem generalizes the paging problem, we cannot hope to achieve constant competiveness without at least a constant capacity blowup. (In this regard, we remark that the results of [9] are incomparable as they do not require a capacity blowup.)

Several paging problems (e.g., distributed paging, file migration, and file allocation) have been considered in the literature, some of which are related to the HCC problem. (See, e.g., the survey paper by Bartal [4] for the definitions of these problems.) In particular, the HCC problem can be formulated as the read-only version of the distributed paging problem on ultrametrics. And the HCC problem without replication is a special case of the constrained file migration problem where accessing and migrating a file has the same cost. Most existing work on these problems focuses on upper bound results, and lower bound results only apply to algorithms without a capacity blowup. For example, for the distributed paging problem, Awerbuch *et al.* [2] have shown that, given $\mathrm{polylog}(n, \Delta)$ capacity blowup, there exists deterministic $\mathrm{polylog}(n, \Delta)$-competitive algorithms on general networks, where Δ is the normalized diameter of the network. For the constrained file migration problem, Bartal [3] has given a deterministic upper bound of $\Omega(m)$, where m is the total size of the caches, and a randomized lower bound of $\Omega(\log m)$ in some network topology, and an $O(\log m \log^2 n)$ randomized upper bound for arbitrary network topologies. Using the recent result of Fakcharoenphol *et al.* [7], the last upper bound can be improved to $O(\log m \log n)$.

The rest of this paper is organized as follows. Section 2 gives the preliminaries of the problem. Sections 3 and 4 present the main result of our paper, a lower bound for constant capacity blowup. Section 5 provides some concluding remarks. Appendix A presents an upper bound for sufficiently large capacity blowup.

2. PRELIMINARIES

In this section we formally define the HCC problem. We are given a fixed six-tuple

$$(\mathcal{F}, \mathcal{C}, \mathit{dist}, \mathit{size}, \mathit{cap}, \mathit{penalty}),$$

where $\mathcal{F}$ is a set of files, $\mathcal{C}$ a set of caches, *dist* a function from $\mathcal{C} \times \mathcal{C}$ to $\mathbf{N}$, *size* a function from $\mathcal{F}$ to $\mathbf{N}$, *cap* a function from $\mathcal{C}$ to $\mathbf{N}$, *penalty* a function from $\mathcal{F}$ to $\mathbf{N}$, and $\mathbf{N}$ denotes nonnegative integers. We assume that *dist* is an ultrametric (defined below) over $\mathcal{C}$, and we assume that for every file f in $\mathcal{F}$, $\mathit{penalty}(f) \geq \mathit{diam}(\mathcal{C})$, where $\mathit{diam}(U)$ denotes $\max_{u,v \in U} \mathit{dist}(u, v)$ for every set of caches U.

2.1 Ultrametrics and Hierarchical Networks

A distance function $d : \mathcal{C} \times \mathcal{C} \rightarrow \mathbf{N}$ is defined to be a *metric* if d is nonnegative, symmetric, satisfies the triangle inequality, and $d(u, v) = 0$ if and only if $u = v$. An *ultrametric* is a special case of a metric that satisfy the inequality $d(u, v) \leq \max(d(u, w), d(v, w))$, which subsumes the triangle inequality $d(u, w) \leq d(u, v) + d(v, w)$.

An equivalent and perhaps more intuitive characterization of our ultrametric assumption is that the caches in $\mathcal{C}$ form a "hierarchical tree", or simply, a tree defined as follows. Every leaf node of the tree corresponds to a (distinct) cache. Every node in the tree has an associated nonnegative value, called the *diameter* of the node, such that for every two caches u and v, $\mathit{dist}(u, v)$ equals the diameter of the least common ancestor of u and v.

Since a hierarchical network has a natural correspondence to a tree, in the rest of this paper, we use tree terminology to develop our algorithms and analysis. In what follows, the definitions of ancestor, descendant, parent, and children follow the standard tree terminology. We use T to denote the tree of caches and we use *root* to denote the root of T. The *depth* of *root* is 0, and the depth of T is the maximum depth of any of its nodes. The *capacity* of a node is the total capacity of all the caches within the subtree rooted at that node. We impose an arbitrary order on the children of every internal node.

2.2 The HCC Problem

The goal of an HCC algorithm is to minimize the total cost incurred in the movement of files to serve a sequence of requests while respecting capacity constraints at each cache. To facilitate a formal definition of the problem, we introduce additional definitions below.

A copy is a pair (u, f) where u is a cache and f is a file. A set of copies is called a *placement*. If (u, f) belongs to a placement $\mathcal{P}$, we say that a copy of f is placed at u in $\mathcal{P}$. A placement $\mathcal{P}$ is *b-feasible* if the total size of the files placed in any cache is at most b times the capacity of the cache. A 1-feasible placement is simply referred to as a *feasible* placement.

Given a placement $\mathcal{P}$, upon a request for a file f at a cache u, an algorithm incurs an access cost to serve the request. If $\mathcal{P}$ places at least one copy of f in any of the caches, then the cost is defined to be $size(f) \cdot dist(u, v)$, where v is the closest cache at which a copy of f is placed; otherwise the cost is defined to be $penalty(f)$. After serving a request, an algorithm may modify its placement via an arbitrarily long sequence of the following two operations: (1) it may add any copy to $\mathcal{P}$ and incur an access cost as defined above, or (2) it may remove any copy from $\mathcal{P}$ and incur no cost.

Given a capacity blowup of b, the goal of an HCC algorithm is to maintain a b-feasible placement such that the total cost is minimized.

3. THE LOWER BOUND

In this section, we show that, given any constant capacity blowup b, the competitive ratio of any online HCC algorithm is $\Omega(\log d)$, where d is the depth of the hierarchy. We prove this lower bound algorithm by showing the existence of a suitable hierarchy, a set of files, a request sequence, and a feasible offline HCC algorithm that incurs an $\Omega(\log d)$ factor lower cost for that request sequence than any online b-feasible HCC algorithm. This result easily extends to analyzing how the lower bound on the competitive ratio varies as a function of nonconstant capacity blowup up to the depth of the hierarchy. In particular, with a capacity blowup of $d^{1-\varepsilon}$ for a fixed $\varepsilon > 0$, the competitive ratio of any online HCC algorithm is still $\Omega(\log d)$.

We present an adversarial argument for the lower bound. Let ON denote a b-feasible online HCC algorithm and ADV an adversarial offline feasible HCC algorithm. ON chooses a fixed value for the capacity blowup b, and ADV subsequently chooses an instance of an HCC problem (i.e., the six-tuple as introduced in Section 2) as follows. The hierarchy *root* consists of n unit-sized caches that form the leaves of a regular k-ary tree with depth $d = 4bk$. Thus, for a given choice of k, $n = k^{4bk}$. The set of files Ψ consists of $\Theta(\frac{n}{k})$ unit-sized files. The diameter of each hierarchy at depth $4bk - 1$ is 1, and the diameter of every non-trivial hierarchy is at least λ times the diameter of any child, where $\lambda > 1$. For any file f, $penalty(f)$ is at least $\lambda \cdot diam(root)$. Given an instance of an HCC problem as described in Section 2, we give a program that takes ON as an input and generates a request sequence and a family of offline HCC algorithms each of which incurs a factor $\Omega(\log d)$ less cost than ON. We use the name OFF to refer to one algorithm in this family.

At a high level, ON's lack of future knowledge empowers ADV to play a game analogous to a *shell game*[1]. In this game, OFF maintains a compact placement of files tailored for the request sequence that ADV generates, while ON is forced to guess OFF's placement and incurs relocation costs if it guesses incorrectly. When ON finally zeroes in on OFF's placement, OFF switches its placement around, incurring a small fraction of the relocation cost that ON has already expended, and repeats the game.

As an example, consider a simple two-level hierarchy associated with equal-sized departments within a university. A set of files, say A, are of university-wide interest, while the remaining files are of department-specific interest. The capacity constraints are set up in such a way that a department can either cache files of its interest or of the university's, but not both sets simultaneously. OFF stores all the files in A in an "idle" department, i.e., one with no access activity. On the other hand, ON has to guess the identity of the idle department. If ON guesses incorrectly, ADV creates requests that force ON to move files in A to a different department. The best strategy for ON is to evenly distribute files in A across all departments that have not yet been exposed as nonidle. Unfortunately, even with this strategy ON ends up incurring a significantly higher cost than OFF. Of course, in this simplistic case, ON can circumvent its predicament simply by a two-fold blowup in capacity and using the algorithm described in the Appendix A. In the rest of the paper, we present a formalization of the shell-game-like adversarial strategy and an extension of this strategy to hierarchies of nonconstant depth.

3.1 The Adversary Algorithm ADV

We fix $d+1$ disjoint sets of files $S_0, S_1, \ldots, S_d$ such that $|S_d| = 1$ and $|S_i| = k^{d-i-1}$ for all $0 \leq i < d$. We call i the depth of a file f if $f \in S_i$. We define the function $g(i, j)$, where $i \geq 0$ and $j > 0$, as

$$g(i,j) = k^{d-i} \cdot \left(\frac{i-1}{4k} + \frac{1}{2j}\right).$$

ADV is shown in Figure 1 and the key notations used in the algorithm (and the rest of the paper) are explained in Table 3.1. In ADV, the nonnegative integer N specifies the number of requests to be generated. The code in Figure 1 only shows how ADV generates a bad request sequence for ON. In Section 4, we show how to augment this code to obtain an offline algorithm that serves the same request sequence but incurs a much lower cost.

For every node, ADV maintains two integer fields, x and y, to summarize the state of ON. In ADV, π is a global variable that records the current node where ADV generates the next request. Initially, π is set to *root*. The program proceeds in rounds. At the end of each round, the algorithm generates a request. Based on ON's adjustment of its own placement, ADV adjusts π using the up loop and the down loop. The former moves π to an ancestor while the latter moves it to a descendant.

---
[1]Thimblerig played especially with three walnut shells.

Notation	Meaning
$\alpha.parent$	the parent of α
$\alpha.anc$	the ancestors of α
$\alpha.desc$	the descendants of α
$\alpha.depth$	the depth of α
$\alpha.diam$	the diameter of α
$\alpha.files$	S_i, where $i = \alpha.depth$
$\alpha.cap$	the total capacity of the caches in α
$\alpha.ch$	children hierarchies of α
$\alpha.placed$	the set of (distinct) files placed in the caches in α
$\alpha.load$	the number of files f in $\alpha.placed$ such that the depth of f is less than $\alpha.depth$
$\alpha.missing$	the set of files f such that the depth of f is $\alpha.depth$ but $f \notin \alpha.placed$
$\alpha.act$	$g(\alpha.depth, r)$, where $r = \lvert\{\beta : \beta \in \alpha.parent.ch : \beta.x = 0\}\rvert$, the "activation" value
$\alpha.react$	$g(\alpha.depth, k)$, the "reactivation" value
$\alpha.deact$	$g(\alpha.depth, 2k)$, the "deactivation" value

Table 1: Key notations.

3.2 Correctness of ADV

We show in this section that ADV is well-defined (i.e., $\pi \neq root$ just before line 12, π is not a leaf just before line 8, and line 14 finds a child) and that each round terminates with the generation of a request. For the sake of brevity, in our reasoning below, we call a predicate a *global invariant* if it holds everywhere in ADV (i.e., it holds initially and it holds between any two adjacent lines of the pseudocode in Figure 1).

LEMMA 3.1. *Let I_1 denote that every internal node has a child with the x field equal to 0, I_2 denote that π is an internal node, and I_3 denote that $\pi.load \geq \pi.deact$. Then $I_1 \wedge I_2$ is a global invariant and I_3 holds everywhere in the down loop.*

PROOF. The predicate $I_1 \wedge I_2$ holds initially because $\pi = root$ and $\alpha.x = 0$ for all α, and I_3 holds just before the down loop due to the guard of the up loop. We next show that every line of code out of the down loop preserves $I_1 \wedge I_2$ (i.e., if $I_1 \wedge I_2$ holds before the line, then it holds after the line) and every line of code in the down loop preserves $I_1 \wedge I_2 \wedge I_3$.

Every line of code out of the down loop preserves I_1 because none assigns a nonzero value to a x field. The only line that affects I_2 is line 5. We observe that $\pi \neq root$ just before line 5, due to the guard of the up loop and the observation that $root.load \geq root.deact = 0$. Hence, line 5 preserves I_2.

In the down loop, the only line that affects I_1 is 15, but I_3 and the inner **if** statement establish that π has at least two children with the x field equal to 0 just before line 14. Hence, line 15 preserves I_1. The only lines that affect I_2 are lines 9 and 14. We first observe that just before line 8, $\pi.depth < 4bk - 1$. This is because I_2 states that $\pi.depth < 4bk$ and I_3 implies that if $\pi.depth = 4bk - 1$, then $\pi.load \geq \pi.deact = bk - \frac{1}{4}$. Since $\pi.load$ is an integer, this implies that $\pi.load \geq bk$, which implies that $\pi.missing \supseteq S_{4bk} \neq \emptyset$, a contradiction to the guard of the down loop. Hence, $\pi.depth < 4bk - 1$ just before line 8. Therefore, line 9 preserves I_2. We now show that line 14 also preserves I_2. Let $A = \{\alpha : \alpha \in \pi.ch \wedge \alpha.x = 0\}$ and $B = \{\beta : \beta \in \pi.ch \wedge \beta.x > 0\}$. Let r denote $|A|$ and i denote $\pi.depth$. We observe that

$$
\begin{aligned}
& \sum_{\alpha \in A} \alpha.load \\
= \; & \sum_{\alpha \in A} \alpha.load + \sum_{\beta \in B} \beta.load - \sum_{\beta \in B} \beta.load \\
= \; & \pi.load + |S_i| - \sum_{\beta \in B} \beta.load \\
\geq \; & \pi.deact + |S_i| - \sum_{\beta \in B} \beta.react \\
= \; & g(i, 2k) + k^{d-i-1} - \sum_{\beta \in B} g(i+1, k) \\
= \; & r \cdot k^{d-i-1} \cdot \left(\frac{i}{4k} + \frac{1}{2r} + \frac{1}{2k} \right).
\end{aligned}
$$

(In the derivation above, the second equality is due to the guard of the down loop and the definition of *load*, and the first inequality is due to the guard of the outer **if** statement.) Hence, by an averaging argument, there exists a child δ of π such that

$$
\begin{aligned}
& \delta.load \\
\geq \; & k^{d-i-1} \cdot \left(\frac{i}{4k} + \frac{1}{2r} \right) \\
= \; & \delta.act.
\end{aligned}
$$

Hence, step 14 finds a child. And as shown above, $\pi.depth < 4bk - 1$ just before line 8. Hence, line 14 preserves I_2. The only lines that affect I_3 are 9 and 14. Both of these lines preserve I_3 because by definition, $\alpha.act \geq \alpha.deact$ and $\alpha.react \geq \alpha.deact$ for all α.

The claim of the lemma then follows. □

LEMMA 3.2. *The up loop terminates.*

PROOF. Every iteration of the up loop moves π to its parent, and $root.load \geq root.deact$ by definition. Hence, the up loop terminates. □

LEMMA 3.3. *The down loop terminates.*

```
    {initially, N ≥ 0, count = 0, π = root, root.x = root.y = root.act = g(0,k), and α.x = α.y = 0 for all α other than root}
1   while count < N do {main loop}
2       while π.load < π.deact do {up loop}
3           π.y := π.react;
4           for every child δ of π, set both δ.x and δ.y to 0;
5           π := π.parent
6       od; {end of up loop}
7       while π.missing = ∅ do {down loop}
8           if a child δ of π satisfies δ.x > 0 ∧ δ.load ≥ δ.react then
9               π := δ
10          else
11              if π has exactly one child with x equal to 0 then
12                  for every child δ of π, set both δ.x and δ.y to 0
13              fi;
14              π := a child δ of π such that δ.x = 0 ∧ δ.load ≥ δ.act;
15              set both π.x and π.y to π.act
16          fi
17      od; {end of down loop}
18      generate a request for an element in π.missing at an arbitrary cache in π;
19      ON serves the request and arbitrarily updates its placement;
20      count := count + 1
21  od {end of main loop}
```

Figure 1: The ADV algorithm.

PROOF. Every iteration of the down loop moves π to one of its children. By I_2 of Lemma 3.1, π is always an internal node. Hence, the down loop terminates. □

LEMMA 3.4. *ADV terminates after generating a sequence of N requests.*

PROOF. Follows from Lemmas 3.2 and 3.3. □

4. COST ACCOUNTING

In this section, we show that there exists an offline HCC algorithm OFF that serves the sequence of requests generated by ADV and incurs a cost that is a factor $\Omega\left(\log \frac{d}{b}\right)$ less than that incurred by any b-feasible online HCC algorithm.

4.1 Some Properties of ADV

We first prove some properties of ADV that follow directly from its structure. For the sake of brevity, for a property that is a global invariant, we sometimes only state the property but omit stating that the property holds everywhere.

LEMMA 4.1. *For all α, $\alpha.x = 0$ or $\alpha.x \geq \alpha.react$.*

PROOF. The claim holds initially because $\alpha.x = 0$ for all α. The only line that assigns a nonzero value to x is 15, which preserves the claim because by definition, $\alpha.act \geq \alpha.react$ for all α. □

LEMMA 4.2. *For all α, $\alpha.y$ equals 0 or $\alpha.react$ or $\alpha.x$.*

PROOF. The claim holds initially because $\alpha.y = 0$ for all α. The only lines that modify x are 4, 12, and 15. The only lines that modify y are 3, 4, 12, and 15. By inspection of the code, all of these lines trivially preserve the claim. □

LEMMA 4.3. *Let P denote the predicate that every node in $\pi.anc$ has a positive x value and every node that is neither in $\pi.anc$ nor a child of a node in $\pi.anc$ has a zero x value. Then P is a loop invariant of the up loop, the down loop, and the main loop.*

PROOF. Let A denote $\pi.anc$ and let B denote the set of nodes that are neither in A nor children of the nodes in A.

Every iteration of the up loop moves π to its parent. To avoid confusion, we use π to denote the old node (i.e., child) and π' to denote the new node (i.e., parent). An iteration of the up loop removes π from A, adds $\pi.ch$ to B, and sets the x value of $\pi.ch$ to 0. Therefore, it preserves P.

Every iteration of the down loop moves π to one of its children. To avoid confusion, we use π to denote the old node (i.e., parent) and π' to denote the new node (i.e., child). Suppose the down loop takes the first branch of the outer **if** statement. Then it adds π', which has a positive x value, to A and removes $\pi'.ch$ from B. Hence it preserves P. Suppose the down loop takes the second branch of the outer **if** statement. If line 12 is executed, P is preserved because line 12 preserves both A and B and only changes the x value of the nodes in neither A nor B. Then lines 14 and 15 preserves P because they add π', which has a positive x value after line 15, to A and removes $\pi'.ch$ from B. Hence, it preserves P.

The main loop preserves P because both the up loop and the down loop preserve P. □

LEMMA 4.4. *For all α, $\alpha.y \leq \alpha.x$.*

PROOF. The claim holds initially because $\alpha.x = \alpha.y = 0$ for all α. The only lines that modify the x or y field are 3, 4, 12, and 15. At lines 4, 12, and 15, the x and y fields become the same value. It follows from Lemma 4.3 and the guard of the up loop that just before line 3, $\pi \neq root$ and $\pi.x > 0$. It then follows from Lemmas 4.1 and 4.2 that line 3 preserves $\pi.y \leq \pi.x$. □

We now introduce the notion of an active sequence to facilitate our subsequent proofs. A sequence $\langle a_0, a_1, \ldots, a_r \rangle$, where $0 \leq r < k$, is called *i-active* if $a_j = g(i+1, k-j)$ for all $0 \leq j \leq r$.

LEMMA 4.5. *For every internal node α, the nonzero x fields of the children of α form an i-active sequence, where $i = \alpha.depth$.*

PROOF. The claim holds initially because $\alpha.x = 0$ for all α. The only lines that modify the x field are 4, 12, and 15. Lines 4 and 12 preserve the claim because the x fields of the children of π all become 0. Line 15 preserves the claim (for $\pi.parent$) because $\pi.x$ becomes $\pi.act$, which by definition equals $g(i+1, k-j)$, where $i = \pi.parent.depth$ and j equals the number of children of $\pi.parent$ that have a positive x field. □

LEMMA 4.6. *Let $P(\alpha)$ denote the predicate that for all β that are not ancestors of α, $\beta.y \leq \beta.react$. Then $P(\pi)$ holds initially and $P(\pi)$ is a loop invariant of the up loop, the down loop, and the main loop.*

PROOF. The predicate $P(\pi)$ holds initially because $\pi = root$ and $\alpha.y = 0$ for all α. The up loop preserves $P(\pi)$ because every iteration first establishes $\pi.y = \pi.react$ and then moves π to its parent. The down loop preserves $P(\pi)$ because it does not set the y field to a nonzero value. The main loop preserves $P(\pi)$ because both the up loop and the down loop preserve $P(\pi)$. □

4.2 Colorings

In order to facilitate the presentation of an offline algorithm in Section 4.3, we introduce the notion of colorings in this section and the notion of consistent placements in the next.

A *coloring* of T (recall that T is the tree of caches) is an assignment of one of the colors {white, black} to every node in T so that the following rules are observed: (1) *root* is white, (2) every internal white node has exactly one black child and $k-1$ white children, and (3) the children of a black node are black. A coloring is called *consistent* (with ADV) if for every α, if $\alpha.x > 0$, then α is white.

For any coloring C and any pair of sibling nodes α and β, we define $swapc(C, \alpha, \beta)$ (swap coloring) as the coloring obtained from C by exchanging the color of each node in the subtree rooted at α with that of the corresponding node in the subtree rooted at β. (Note that the subtrees rooted at α and β have identical structure.)

4.3 Consistent Placements

A placement is *colorable* if there exists a coloring C such that: (1) for each white internal node α of T, the set of files α.*files* are stored in (and fill) the caches associated with the unique black child of α; (2) for each white leaf α of T, the (singleton) set of files α.*files* is stored in (and fill) the cache α. Note that in the preceding definition of a colorable placement, the coloring C, if it exists, is unique. A placement is called *consistent* if it is colorable and the associated coloring is consistent.

For any placement $\mathcal{P}$ and any pair of siblings α and β, we define $swapp(\mathcal{P}, \alpha, \beta)$ (swap placement) as the placement obtained from $\mathcal{P}$ by exchanging the contents of each cache in α with that of the corresponding cache in β. Note that for any colorable placement $\mathcal{P}$ with associated coloring C and any pair of sibling nodes α and β, the placement $swapp(\mathcal{P}, \alpha, \beta)$ is colorable, and its associated coloring is $swapc(C, \alpha, \beta)$.

4.4 The Offline Algorithm OFF

For every internal node α, we maintain an additional variable $\alpha.last$ defined as follows. First, we partition the execution of the adversary algorithm into epochs with respect to α. The first epoch begins at the start of execution. Each subsequent epoch begins when either line 4 or line 12 is executed with $\pi = \alpha$. The variable $\alpha.last$ is updated at the start of each epoch, when it is set to the child β of α for which the line 15 is executed with $\pi = \beta$ furthest in the future. (If one or more children β of α are such that line 15 is never executed with $\pi = \beta$ in the future, then $\alpha.last$ is set to an arbitrary such child β.) Note that the variables $\alpha.last$ are introduced solely for the purpose of analysis and have no impact on the execution of ADV.

At any point in the execution of ADV, the values of the *last* fields determine a unique coloring, denoted by C_{OFF}, as follows: *root* is white and the black child of each internal white node α is $\alpha.last$.

We define an offline algorithm OFF that maintains a placement P_{OFF} as follows. We initialize P_{OFF} to an arbitrary consistent placement with associated coloring C_{OFF}. We update P_{OFF} to $swapp(P_{\mathrm{OFF}}, \alpha, \beta)$ whenever line 4 or line 12 is executed, where α and β denote the values of $\pi.last$ before and after the execution of the line. The algorithm OFF uses the placement P_{OFF} to serve each request generated in line 18. The placement P_{OFF} is not updated when OFF serves a request; P_{OFF} is updated only at lines 4 and 12.

LEMMA 4.7. *Throughout the execution of ADV, P_{OFF} is colorable and has associated coloring C_{OFF}.*

PROOF. Immediate from the way P_{OFF} is updated whenever a *last* field is updated. □

LEMMA 4.8. *Execution of line 4 or line 12 preserves the consistency of C_{OFF}.*

PROOF. Assume that C_{OFF} is consistent before line 4. So π is white in C_{OFF} before line 4, because by Lemma 4.3, $\pi.x$ is positive before line 4. By the definition of C_{OFF}, before line 4, $\pi.last$ is black. Let α be $\pi.last$ before line 4, and let β be $\pi.last$ after line 4. Before and after line 4, the x values of the descendants of α are equal to 0. By Lemma 4.3, the x values of all proper descendants of β are equal to 0 before and after line 4. Since $\beta.x = 0$ after line 4, the x values of all descendants of α and β are equal to 0 after line 4. Hence, the *swapp* operation preserves the consistency of C_{OFF}. The same argument applies to line 12. □

LEMMA 4.9. *Execution of line 15 preserves the consistency of C_{OFF}.*

PROOF. Assume that C_{OFF} is consistent before line 15. Line 14 implies that $\pi \neq root$ just before line 15. Let π' denote $\pi.parent$. By Lemma 4.3, $\pi'.x > 0$ and hence π' is white before line 15. Therefore, by Lemma 4.7, $\pi'.last$ is the black child of π'.

Let t denote the start of the current epoch for π', i.e., t is the most recent time at which $\pi'.last$ was assigned. Just after time t, the x values of all children of π' were equal to 0. By the definition of t, no child of π' has been set to 0 since time t. By Lemma 3.1, every internal node has at least one child with x equal to 0. Therefore, from time t until after the execution of line 15, at most $k-1$ children of π' have had their x value set to a nonzero value. (Note that line 15 is the only line that sets x to a nonzero value.) Thus, by the definition of *last*, $\pi'.last.x$ remains 0 after the execution of this line. Thus, $\pi'.last \neq \pi$. Since π' is white and $\pi'.last$ is black in C_{OFF}, we conclude that π is white in C_{OFF}. So C_{OFF} remains consistent even with the additional constraint that π is required to be white. (Note that $\pi.x$ is set to a positive value by line 15.) □

LEMMA 4.10. *The placement P_{OFF} is always consistent.*

PROOF. We observe that C_{OFF} is always consistent, due to Lemmas 4.8 and 4.9, and the observation that lines 4, 12, and 15 are the only lines that can affect the consistency of C_{OFF} (because they are the only lines that modify the *last* field or the x field of any node). It then follows from Lemma 4.7 that P_{OFF} is always consistent. □

4.5 A Potential Function Argument

Let ON denote an arbitrary online b-feasible algorithm. In this section, we use a potential function argument to show that ON is $\Omega\left(\frac{\nu}{\nu'}\right)$- competitive, where

$$\nu = \min\left(\frac{\lambda}{8}, \frac{\ln k}{4} - \frac{1}{4}\right)$$

and $\nu' = \frac{\lambda}{\lambda-1}$. Let T_{ON} denote the total cost incurred by ON. Similarly, we let T_{OFF} denote the total cost incurred by OFF, except that we exclude from T_{OFF} the cost of initializing P_{OFF}. (This initialization cost is taken into account in the proof of Theorem 1 below.) We define Φ, a potential function, as:

$$\begin{aligned}\Phi \;=\; & \nu \cdot T_{\mathrm{OFF}} - \nu' \cdot T_{\mathrm{ON}} + \\ & \sum_{\alpha \in \pi.anc \wedge \alpha \neq root} \alpha.parent.diam \cdot \alpha.x + \\ & \sum_{\alpha \notin \pi.anc} \alpha.parent.diam \cdot (\alpha.x - \alpha.y + \alpha.load)\end{aligned} \qquad (1)$$

For convenience of exposition, we account for the cost of moving from the empty placement to the first placement separately.

LEMMA 4.11. *The cost incurred by $swapp(\mathcal{P}, \alpha, \beta)$ is at most $2 \cdot k^{d-i} \cdot \alpha.parent.diam$, where $i = \alpha.depth$.*

PROOF. The cost incurred is the cost of exchanging the files placed in α and β with each other, which is at most $2 \cdot \alpha.cap \cdot \alpha.parent.diam = 2 \cdot k^{d-i} \cdot \alpha.parent.diam$. Note that α and β have the same capacity. □

LEMMA 4.12. *The predicate $\Phi \leq 0$ is a loop invariant of the up loop.*

PROOF. Every iteration of the up loop moves π to its parent. To avoid confusion, we use π to refer to the old node (i.e., child) and we use π' to refer to the new node (i.e., parent). Consider the change in Φ in a single iteration of the up loop. ON incurs no cost in the up loop. By the definition of Φ, line 3 preserves Φ. By Lemma 4.4, line 4 does not increase Φ. Let $i = \pi.depth$. By Lemma 4.11, after the execution of line 4, OFF incurs a cost of at most $c = 2 \cdot k^{d-i-1} \cdot \pi.diam$ to move from the current consistent marking placement to the next. Thus, the total change in Φ in an iteration is at most

$$\begin{aligned} & \nu \cdot c - \pi'.diam \cdot (\pi.y - \pi.load) \\ \leq\; & \nu \cdot c - \pi'.diam \cdot (\pi.react - \pi.deact) \\ =\; & \nu \cdot c - \pi'.diam \cdot (g(i,k) - g(i,2k)) \\ =\; & \nu \cdot c - \pi'.diam \cdot k^{d-i-1} \cdot \frac{1}{4} \\ \leq\; & \nu \cdot c - \frac{\lambda}{8} \cdot c \\ \leq\; & 0. \end{aligned}$$

(In the derivation above, the first inequality is due to the guard of the up loop and line 3, and the second inequality is due to the assumption that the diameters of the nodes are λ separated.) □

LEMMA 4.13. *The predicate $\Phi \leq 0$ is a loop invariant of the down loop.*

PROOF. Every iteration of the down loop moves π to one of its children. To avoid confusion, we use π to refer to the old node (i.e., parent) and π' to refer to the new node (i.e., child). ON incurs no cost in the down loop. We consider the following three cases.

Suppose that the outer **if** statement takes the first branch. In this case, OFF does not incur any cost. Thus, the change in Φ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ \leq\; & \pi.diam \cdot (\pi.react - \pi.react) \\ =\; & 0, \end{aligned}$$

where the inequality is due to Lemma 4.6 and the guard of the outer **if** statement.

Suppose that the outer **if** statement takes the second branch and that line 12 is not executed. In this case, OFF does not incur any cost. Thus, the change in Φ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ \leq\; & \pi.diam \cdot (\pi'.x - \pi'.load) \\ \leq\; & 0, \end{aligned}$$

where the first inequality is due to Lemma 4.4 and the second inequality is due to lines 14 and 15.

Suppose that the outer **if** statement takes the second branch and that line 12 is executed. By Lemma 4.11, in this case, OFF incurs a cost of $c = 2 \cdot k^{d-i-1} \cdot \pi.diam$.

Thus, the change in $\phi(\pi)$ due to line 12 is at most

$$\begin{aligned}
& \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.y) \\
\leq\ & \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.react) \\
=\ & \nu \cdot c - \pi.diam \cdot \sum_{j=1}^{k-1} (g(i+1, k-j) - g(i+1, k)) \\
=\ & \nu \cdot c - \pi.diam \cdot k^{d-i-1} \sum_{j=1}^{k-1} \left(\frac{1}{2(k-j)} - \frac{1}{2k} \right) \\
\leq\ & \nu \cdot c - \left(\frac{\ln k}{4} - \frac{1}{4} \right) \cdot c \\
\leq\ & 0.
\end{aligned}$$

(In the above derivation, the first inequality follows from Lemma 4.6 and the first equality follows from Lemma 4.5.) By the analysis of the previous case (i.e., the outer **if** statement takes the second branch but line 12 is not executed), lines 14 and 15 do not increase Φ. Thus, every iteration of the down loop preserves $\Phi \leq 0$. □

LEMMA 4.14. *Lines 18 to 20 preserve* $\Phi \leq 0$.

PROOF. The guard of the down loop ensures that there exists a file in $\pi.missing$ just before line 18. Thus, ON incurs a cost at least $\pi.parent.diam \geq \lambda \cdot \pi.diam$ at line 19. OFF incurs a cost at most $\pi.diam$ because it stores all the files in $\pi.missing \subset S_i, i = \pi.depth$, in a child of π. Let u be the cache where the request is generated, and let A be the set of nodes on the path from π to u, excluding π. Since ON adds a file in $\pi.missing$ to u, the change in Φ is at most

$$\begin{aligned}
& \nu \cdot \pi.diam - \nu' \cdot \lambda \cdot \pi.diam + \sum_{\alpha \in A} \alpha.parent.diam \\
\leq\ & \pi.diam \cdot (\nu - \nu' \cdot \lambda) + \pi.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\
\leq\ & \pi.diam \cdot \left(\nu - \nu' \cdot \lambda + \frac{\lambda}{\lambda - 1} \right) \\
\leq\ & 0.
\end{aligned}$$

(In the above derivation, the last inequality follows from $\nu' = \frac{\lambda}{\lambda-1} > \frac{1}{\lambda-1} + \frac{1}{8}$.) At line 19, ON is allowed to make arbitrarily many updates to its own placement. Suppose an update causes the *load* of some nodes to increase. Then by the definition of *load*, the set of nodes with an increased *load* value form a path from, say α, to a leaf, and ON incurs a cost of at least $\alpha.parent.diam$. Let the set of nodes on this path be B. Since the diameters of the nodes on this path are λ separated, the change of Φ is at most

$$\begin{aligned}
& \sum_{\beta \in B} \beta.parent.diam - \nu' \cdot \alpha.parent.diam \\
\leq\ & \alpha.parent.diam \cdot \sum_{j \geq 0} \lambda^{-j} - \nu' \cdot \alpha.parent.diam \\
=\ & \frac{\lambda}{\lambda - 1} \cdot \alpha.parent.diam - \nu' \cdot \alpha.parent.diam \\
=\ & 0.
\end{aligned}$$

The claim of the lemma then follows. □

THEOREM 1. ON *is* $\Omega\left(\frac{\nu}{\nu'}\right)$*-competitive.*

PROOF. Initially, $\Phi = 0$. By Lemmas 4.12, 4.13, and 4.14, $\Phi \leq 0$ is a loop invariant of the main loop. Therefore, by Lemmas 4.1 and 4.4, $T_{\text{ON}} \geq \frac{\nu}{\nu'} \cdot T_{\text{OFF}}$ holds initially and is a loop invariant of the main loop. Let c be the cost incurred by OFF in moving from the empty placement to the first placement. Note that T_{ON} serves every request with a cost at least 1 (because the diameter of an internal node is at least 1). Hence, given an arbitrarily long sequence of requests, T_{ON} grows unbounded. Therefore, we can make $\frac{T_{\text{ON}}}{T_{\text{OFF}}+c}$ arbitrarily close to $\frac{\nu}{\nu'}$ by increasing the length N of the request sequence generated by the program. □

The $\Omega\left(\log \frac{d}{b}\right)$ bound on the competitive ratio for a capacity blowup $b = d^{1-\epsilon}$, where $\epsilon > 0$, claimed in the beginning of Section 3, follows from $d = 4bk$ and that OFF can choose an arbitrarily large λ.

5. DISCUSSION

Cooperative caching has in fact found its application in areas other than distributed systems. For example, in NUCA (NonUniform Cache Architecture), a switched network allows data to migrate to different cache regions according to access frequency [8]. Although NUCA only supports a single processor at the time of this writing, multiprocessor NUCA is being developed, with data replication as a possibility.

6. 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 Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583, January 1996.

[3] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, October 1996.

[4] Y. Bartal. Distributed paging. In A. Fiat and G. J. Woeginger, editors, *The 1996 Dagstuhl Workshop on Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 1998.

[5] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28:119–125, 1995.

[6] 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.

[7] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, June 2003.

[8] C. K. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS), pages 211–222, October 2002.
[9] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38:260–302, 2001.
[10] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
[11] D. Wessels. Squid Internet object cache. Available at URL http://squid.nlanr.net/squid, January 1998.
[12] D. Wessels and K. Claffy. RFC 2187: Application of Internet Cache Protocol, 1997.
[13] N. E. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.
[14] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, 1997.

APPENDIX

A. AN UPPER BOUND

We show in this section that, given $2(d+1)$ capacity blowup, where d is the depth of the hierarchy, a simple LRU-like algorithm, which we refer to as HLRU *(Hierarchical LRU)*, is constant competitive. For the sake of simplicity, we assume that every file has unit size and uniform miss penalty. Our result, however, can be easily extended to handle variable file sizes and nonuniform miss penalties using a method similar to LANDLORD [13].

A.1 The HLRU Algorithm

Every cache in HLRU is $2(d+1)$ times as big as the corresponding cache in OPT. HLRU divides every cache into $d+1$ equal-sized segments numbered from 0 to d. For a hierarchy α, we define $\alpha.small$ to be the union of segment $\alpha.depth$ of all the caches within α, and we define $\alpha.big$ to be the union of $\beta.small$ for all $\beta \in \alpha.desc$.

For the rest of this section, we extend the definitions of a *copy* and a *placement* (defined in Section 2.2) to internal nodes as well. A copy is a pair (α, f) where α is a node and f is file that is stored in $\alpha.small$. A placement refers to a set of copies. The HLRU algorithm, shown in Figure 2, maintains a placement $\mathcal{P}$. In HLRU, a node α uses a variable $\alpha.ts[f]$ to keep track of the timestamp of a file f.

A.2 Analysis of the HLRU Algorithm

For any node α and file f, we partition time into *epochs* with respect to α and f as follows. The first epoch begins at the start of execution, which is assumed to be at time 1. Subsequent epochs begin whenever line 11 is executed.

We define $\alpha.ts^*[f]$ to be the time of the most recent access to file f in node α in the current epoch with respect to node α and file f. If no such access exists, we define $\alpha.ts^*[f]$ to be 0.

For convenience of analysis, we categorize the file movements in HLRU into two types: *retrievals* and *evictions*. Upon request of a file, the HLRU algorithm first performs a retrieval (from the beginning of the code to line 5 of the first iteration of the loop) of the file from the nearest cache that has a copy. Each subsequent iteration of the loop performs an eviction (from line 6 of an iteration to line 5 of the next iteration) of a file from $\alpha.small$ to $\alpha.parent.small$ for some node α.

```
    {upon a request for f at (cache) α}
1   t := now;
2   do
3       flag := false;
4       P := P ∪ {(α, f)};
5       α.ts[f] := max(α.ts[f], t);
6       if capacity is violated at α.small then
7           f := file with smallest nonzero α.ts[f];
8           P := P\{(α, f)};
9           if f ∉ α.big then
10              t := α.ts[f];
11              α.ts[f] := 0;
12              α := α.parent;
13              flag := true
14          fi
15      fi
16  while flag
```

Figure 2: The HLRU algorithm.

LEMMA A.1. *Before and after every retrieval or eviction, for any node α and file f, $f \in \alpha.big$ iff $\beta.ts[f] > 0$ for some $\beta \in \alpha.desc$.*

PROOF. Initially, both sides of the equivalence are false. If both sides of the equivalence are false, the only event that truthifies either side is a retrieval of f at a cache u within α, which in fact truthifies both sides. It remains to prove that if both sides of the equivalence are true, and if one side becomes false, then the other side becomes false.

The only event that falsifies the left side is an eviction of the last copy of f in $\alpha.big$ from $\alpha.small$. Prior to this eviction, $\beta.ts[f] = 0$ for all proper descendants β of α (since the equivalence holds for β) and $\alpha.ts[f] > 0$. The eviction then sets $\alpha.ts[f]$ to 0, falsifying the right side.

The only event that can falsify the right side is an eviction of f from $\alpha.small$ such that, after the eviction, $f \notin \alpha.big$. (Note that eviction of f from $\beta.small$, for a proper descendant β of α, cannot falsify the right side because such an eviction ensures $\beta.parent.ts[f] > 0$.) Thus, falsification of the right side implies falsification of the left side. □

LEMMA A.2. *Before and after every retrieval or eviction, for any node α and file f,*

$$\alpha.ts^*[f] = \max_{\beta \in \alpha.desc} \beta.ts[f].$$

PROOF. Initially, both sides of the equality are zero. By the definition of $\alpha.ts^*[f]$, the value of $\alpha.ts^*[f]$ changes from nonzero to 0 (i.e., a new epoch with respect to α and f begins) at line 11. By the guard of the inner **if** statement, $f \notin \alpha.big$ just before line 11. Hence, by Lemma A.1, $\beta.ts[f]$ is 0 for all $\beta \in \alpha.desc$.

The value $\alpha.ts^*[f]$ increases due to some access of f at a cache u within α. The equality holds because the max value on the right side is at u.

Between the changes of $\alpha.ts^*[f]$, only eviction of f from α can change the max (reset it to 0) on the right side of the equality. This eviction also resets $\alpha.ts^*[f]$ to 0 because a new epoch begins. □

LEMMA A.3. *Before and after every retrieval or eviction, for any node α and file f, $\alpha.ts[f] \le \alpha.ts^*[f]$. Furthermore, just after line 8, if $f \notin \alpha.big$, then $\alpha.ts[f] = \alpha.ts^*[f]$.*

PROOF. The first claim of the lemma follows immediately from Lemma A.2. For the second claim, note that we are evicting the last copy of f in $\alpha.big$ from $\alpha.small$. By Lemma A.1, all proper descendants β of α have $\beta.ts[f] = 0$. So $\alpha.ts[f] = \alpha.ts^*[f]$ by Lemma A.2. □

In what follows, for the convenience of analysis, we define *root.parent* to be a fake node that has every file, and we define *root.parent.diam* to be the uniform miss penalty.

When a file is moved from cache u to v, for every node α on the path from the least common ancestor of u and v to v excluding the former, we charge a *pseudocost* of $\alpha.parent.diam$ to node α.

LEMMA A.4. *If a file movement (between two caches) has actual cost C and charges a total pseudocost of C', then*

$$C \le C' \le \frac{\lambda}{\lambda - 1} C.$$

PROOF. Suppose the file movement is from cache u to cache v. Let α be the least common ancestor of u and v and let B be the nodes on the path from α to v, excluding α. Then

$$\begin{aligned} & C \\ = \ & \alpha.diam \\ \le \ & \sum_{\beta \in B} \beta.parent.diam \\ = \ & C' \\ \le \ & \alpha.diam \cdot \sum_{j \ge 0} \lambda^{-j} \\ = \ & \frac{\lambda}{\lambda - 1} \cdot C. \end{aligned}$$

□

For any node α and file f, we define auxiliary variables $\alpha.in[f]$ and $\alpha.out[f]$ for the purpose of our analysis. These variables are initialized to 0. We increment $\alpha.in[f]$ whenever retrieval of file f charges a pseudocost to node α. We increment $\alpha.out[f]$ whenever eviction of file f charges a pseudocost to node α.

LEMMA A.5. *For any node α, the total pseudocost charged to node α due to retrievals is*

$$\sum_f \alpha.in[f] \cdot \alpha.parent.diam.$$

PROOF. Follows from the observation that whenever a pseudocost is charged to node α due to a retrieval, the pseudocost is $\alpha.parent.diam$. □

LEMMA A.6. *For any node α, the total pseudocost charged to node α due to an eviction is at most*

$$\sum_f \alpha.out[f] \cdot \alpha.parent.diam.$$

PROOF. Follows from the observation that whenever a pseudocost is charged to node α due to an eviction, the pseudocost is at most $\alpha.parent.diam$. □

LEMMA A.7. *For any node α and file f,*

$$\alpha.out[f] \le \alpha.in[f].$$

PROOF. We observe that if a pseudocost is charged to a node α as a result of a retrieval, then the retrieval truthifies $f \in \alpha.big$. Similarly, if a pseudocost is charged to node α as a result of an eviction, then the eviction falsifies $f \in \alpha.big$. It then follows that

$$\alpha.out[f] \le \alpha.in[f] \le \alpha.out[f] + 1$$

because $f \notin \alpha.big$ initially. □

LEMMA A.8. *For any node α, the set $\alpha.big$ always contains the most recently accessed $2 \cdot \alpha.cap$ files.*

PROOF. Let X denote the set of the most recently accessed $2 \cdot \alpha.cap$ files. We consider the places where a file is added to X or removed from $\alpha.big$.

A file f can be added to X only when f is requested at a cache u within α. In this case, f is added to $u.small$ and is not evicted from $u.small$ because it is the most recently accessed item. Hence, $f \in \alpha.big$.

A file f can be removed from $\alpha.big$ only when it is moved from $\alpha.small$ to $\alpha.parent.small$ as the result of an eviction and there is no other copy of f in $\alpha.big$. This means that f is chosen as the LRU item at line 7. Since f is the LRU item, there are $2 \cdot \alpha.cap$ items g in $\alpha.small$ such that $\alpha.ts[f] < \alpha.ts[g] \le \alpha.ts^*[g]$. By Lemma A.3, $\alpha.ts[f] = \alpha.ts^*[f]$ just after line 8. It follows from the definition of ts^* that $f \notin X$. □

In what follows, we use OPT to refer to an optimal offline algorithm.

LEMMA A.9. *For any node α, the total pseudocost due to retrievals charged to α by HLRU is at most twice the pseudocost charged to α by OPT.*

PROOF. Fix a node α. For OPT, we say that a request for a file f at a cache within α results in a miss if no copy of f exists at any cache within α at the time of the request. For HLRU, a miss occurs if no copy of f is in $\alpha.big$. By Lemma A.8, HLRU incurs at most as many misses as an LRU algorithm with capacity $2 \cdot \alpha.cap$ running on the subsequence of requests originating from the caches within α. (Note that LRU misses whenever HLRU misses.) By the well-known result of Sleator-Tarjan [10], such an LRU algorithm incurs at most twice as many misses as OPT.

Note that a miss results in a pseudocost of $\alpha.parent.diam$ being charged to α. Therefore, the total pseudocost charged to node α in OPT is at least the number of misses in OPT times $\alpha.parent.diam$. Furthermore, within HLRU, a pseudocost is charged to node α only on a miss. Therefore, the total pseudocost charged to node α in HLRU is at most the number of misses incurred by HLRU times $\alpha.parent.diam$. The claim of the lemma then follows. □

LEMMA A.10. *For any node a, the total pseudocost due to evictions charged to α by HLRU is at most four times the total pseudocost charged to node α by OPT.*

PROOF. Follows immediately from Lemmas A.5, A.6, A.7, and A.9. □

THEOREM 2. *HLRU is constant competitive.*

PROOF. Follows immediately from Lemmas A.4 and A.10. □