Online Hierarchical Cooperative Caching

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

February 6, 2004

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 deterministic constant-competitive 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. It is interesting to note that the offline algorithms associated with our lower bound argument do not replicate files. Accordingly, our lower bound also holds for the variant of the hierarchical cooperative caching problem in which replication is not permitted.

[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.

1 Introduction

The traditional caching problem, which has been extensively studied, is as follows. Given a cache and a sequence of requests for files, 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 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 [11], 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 caching problem is also known as *paging* if the files have uniform size and retrieval cost. In their seminal paper, Sleator and Tarjan [11] have shown that LRU (Least-Recently-Used) and several other deterministic paging algorithms are $k/(k-h+1)$-competitive, where k is the cache space used by LRU and h is that used by OPT. They have also shown that $k/(k-h+1)$ is the best possible among all deterministic algorithms. We call k/h the *capacity blowup* of LRU. For files of nonuniform size and retrieval cost, Young [14] has proposed the LANDLORD algorithm and shown that LANDLORD is $k/(k-h+1)$-competitive. As stated in [14], 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 arrangments of caches. Harvest uses the Internet Cache Protocol [13] to support discovery and retrieval of documents from other caches. The Harvest project later became the public domain Squid cache system [12]. Adaptive Web Caching [15] 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 an 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 an file, free of cost, and we assume that files are read-only (i.e., copies of an file are always consistent). We focus on a class of networks called *hierarchical networks*, the precise definition of which is given in Section 2.

Our notion of a hierarchical network is constant-factor related to the notion of hierarchically well-separted 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.* [10] 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. It follows that our lower bound also holds for the constrained file migration problem (which does not allow replication) restricted to ultrametric spaces.

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 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 competitivenss 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 [10] 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 polylog(n, Δ) capacity blowup, there exists deterministic polylog(n, Δ)-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. Section 3 presents the main result of our paper, a lowerbound for constant capacity blowup. Section 5 provides some concluding remarks. Appendix A presents an upperbound for sufficiently large capacity blowup.

2 Preliminaries

In this section we formally define the hierarchical cooperative caching (HCC) problem. We are given a fixed tuple $(\Psi, \Pi, \textit{distance}, \textit{size}, \textit{capacity}, \textit{miss\_penalty})$, where Ψ is a set of files, Π is a set of caches, *distance*: $\Pi \times \Pi \to \mathbf{R}$, *size*: $\Psi \to \mathbf{N}$, *capacity*: $\Pi \to \mathbf{N}$, and *miss_penalty*: $\Psi \to \mathbf{R}$. We assume that the function *distance* is an ultrametric (defined below) over Π. We assume that for any file f in Ψ, $\textit{miss\_penalty}(f)$ is at least as large as $\textit{diameter}(\Pi)$, where for any set of caches U, $\textit{diameter}(U)$ denotes the maximum of $\textit{distance}(u, v)$ over all caches u and v in U.

2.1 Ultrametrics and Hierarchical Networks

A distance function $d : \Pi \times \Pi \to \mathbf{R}$ is defined to be a metric if and only if d is nonnegative, symmetric, satisfies the triangle inequality, and $d(u, v) = 0$ if and only if $u = v$. An ultrametric satisfies an additional requirement: $d(u, v) \leq \max(d(u, w), d(v, w))$.

A more intuitive characterization of *distance* being an ultrametric is that the caches in Π form a "hierarchical network", or simply, a "hierarchy" defined as follows. A cache is *trivially* a hierarchy. A set of caches is a hierarchy if and only if the caches form the leaves of a tree with minimum degree 2 whose nodes are labeled as follows: the leaves of the tree are labeled with the value 0 and interior nodes are labeled with positive values that strictly increase along any path from a leaf to the root, and the distance between any pair of caches u and v is given by the label on the least common ancestor of u and v. Thus, every hierarchy is associated with a tree that is a subtree of the tree associated with Π.

If a hierarchy α contains a hierarchy β, then α is called an *ancestor* of β and β a *descendant* of α. If α is an ancestor of β and $\alpha \neq \beta$, then α is called a *proper* ancestor of β and β a proper descendant of α. If α is the smallest proper ancestor of β, then α is called the *parent* of β and β the *child* of α. Each hierarchy is associated with a *depth* which is a non-negative integer. The hierarchy Π has a depth of 0 and the depth of every other hierarchy is defined to be one more than the depth of its parent. The definitions of ancestor, descendant, parent, and depth thus match the standard meaning of these terms when used to describe the roots of the trees with which the hierarchies are associated.

2.2 The Hierarchical Cooperative Caching Problem

The goal of a hierarchical cooperative caching algorithm is to minimize the total cost incurred in the movement of files to serve an a priori unknown 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 P, we say that a copy of f is placed at u in P. A placement P is b-*feasible* if, for each cache u, $\sum_{(u,f)\in P} \mathit{size}(f) \leq b \cdot \mathit{capacity}(u)$. A 1-feasible placement is simply referred to as a *feasible* placement.

Given a placement P, upon a request for a file f at a cache u, an algorithm must perform the operation $\texttt{serve}(P, u, f)$ that incurs an access cost $\mathit{cost}(P, u, f)$ to serve the request. If P places at least one copy of f in any of the caches, then $\mathit{cost}(P, u, f)$ is defined as $\mathit{size}(f) \cdot \mathit{distance}(u, v)$, where v is the closest cache at which a copy of f is placed; otherwise $\mathit{cost}(P, u, f)$ is defined as $\mathit{miss\_penalty}(f)$. After serving a request, an algorithm may modify its placement through an arbitrarily long sequence of any of the following operations. The operation $\texttt{fetch}(P, u, f)$ modifies P to $P \cup \{(u, f)\}$ and incurs a cost $\mathit{cost}(P, u, f)$. The operation $\texttt{discard}(P, u, f)$ modifies P to $P - \{(u, f)\}$ and does not incur any cost. Given a capacity blowup of b, the goal of a hierarchical cooperative caching algorithm is to maintain a b-feasible placement such that the total cost of movement of files involved in serving requests and performing fetch operations 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 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 blowups 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 adversary-style exposition of the framework to prove the lower bound. Let ALG denote a b-feasible online HCC algorithm and ADV an adversarial offline feasible HCC algorithm. ALG chooses a fixed value for the capacity blowup b, and ADV subsequently chooses an instance of an HCC problem, a six-tuple as introduced in Section 2, as follows. The hierarchy Π 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 is determined by the relation $n = k^{4bk}$. 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. For any file f, $miss\_penalty(f)$ is at least $\lambda \cdot diameter(\Pi)$. Given an instance of an HCC problem as described, in section 3.1, we give a program that takes ALG 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 ALG. We use the name ADV to refer to one algorithm in this family.

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

As an example, consider a simple two-level hierarchy consisting of equal-sized departments in a university. Some files are of university-wide interest, say A, while the rest are department-specific. 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. ADV stores all the files in A in an *idle* department, i.e., one with no access activity. On the other hand, ALG has to guess the identity of the idle department. If ALG guesses incorrectly, ADV creates requests that force ALG to move files in A to a different department. The best strategy for ALG is to evenly distribute files in A across all departments that have not yet been exposed as non-idle. Unfortunately, even with this strategy ALG ends up incurring a significantly higher cost than ALG. Of course, in this simplistic case, ALG can circumvent its predicament simply by a two-fold blowup in capacity and the algorithm described in the previous section. The rest of the paper brings to light the following main ideas: (i) a formalization of the shell-game-like adverserial strategy, (ii) extension of this strategy for hierarchies of nonconstant depth, and (iii) the applicability of the hierarchical strategy even to scenarios in which all files are distinct and replicated copies are not permitted in the system.

3.1 Definitions

We introduce additional definitions to describe the adversarial program below. The program is presented in an object-oriented style, where a hierarchy is represented as a class with fields and methods that ADV uses to generate requests and update its own placement based on ALG's actions. In the rest of this section we use the term hierarchy to mean both a set of caches separated by an ultrametric distance function and an instance of the class hierarchy.

The function $g(i, j)$ is defined as $k^{d-i} \cdot (\frac{i-1}{4k} + \frac{1}{2j})$. The set of $\frac{n-1}{k-1}$ files Ψ is arbitrarily divided into $4bk + 1$ exclusive sets, $S_0, S_1, \ldots, S_{4bk}$, such that $|S_{4bk}| = 1$ and $|S_i| = k^{d-i-1}$ for $0 \leq i < 4bk$. A hierarchy has two fields called *primary* and *secondary* that are reals used by ADV to keep track of ALG's actions. A hierarchy has the methods listed in Table 3.1.

In the adversary's program given below, *perch* is a global variable that records the current hierarchy where ADV generates requests. Initially, *perch* is set to Π. The program proceeds in rounds where the end of a round and the beginning of the next is marked by the generation of a request. Each round is an iteration of the loop in $main()$, i.e., execution of steps 3 and 4. Based on ALG's adjustment of its placement in step 3 of $request()$, ADV adjusts *perch* through the methods $heavy\_ancestor()$ and $weak\_descendant()$ in step 3. The former moves *perch* to an ancestor while the latter moves it to a descendant. We refer to the "tentative" values of *perch*, i.e., the value of **this** during the execution of the methods $heavy\_ancestor()$ and $weak\_descendant()$ as the *current hierarchy*.

At any time, ADV maintains a *marking placement* defined as follows. Hierarchies are marked as idle or non-idle. The top-level hierarchy is marked non-idle. Every non-trivial hierarchy at depth i that is not marked idle has

[1]Thimblerig played especially with three walnut shells.

Method	Return value
parent()	the parent hierarchy
depth()	the depth of the hierarchy
capacity()	the sum of $\mathit{capacity}(u)$ over all descendant caches u
leftmost()	the leftmost descendant cache
children()	the set of child hierarchies
placed()	the set of (distinct) files placed in any descendant cache
load()	the number of files f in *placed*() such that $f \in S_i$ for $i < \mathit{depth}()$
missing()	the set of files f such that $f \in S_i$ and $f \notin \mathit{placed}()$, where $i = \mathit{depth}()$
pristine()	the number of children with $\mathit{primary} = 0$.
activation()	$g(i, r)$, where $i = \mathit{depth}()$ and $r = \mathit{parent}().\mathit{pristine}()$.
reactivation()	$g(i, k)$
deactivation()	$g(i, 2k)$
heavy_ancestor()	defined below
weak_descendant()	defined below
request()	defined below
reset()	defined below

Table 1: Methods of the class hierarchy

exactly one child that is marked idle; this idle child places all files in S_i. Every cache places the file in S_{4bk}. We note that ADV is an offline HCC algorithm whose existence follows from the properties of the program as we show in Section 4. Till then, we defer the specification of the exact sequence of placements that ADV maintains.

The overall strategy of ADV is as follows. First, it ensures that ALG places a copy of every file in S_i in the current hierarchy at depth i, or creates a request to enforce placement of a $\mathit{missing}$ copy. If ALG places a copy of all files in S_i in the current hierarchy, ADV moves the current hierarchy to a child in which ALG places a sufficient fraction of the load, i.e., files in $S_0 \cup \ldots \cup S_{i-1}$ that are placed in the current hierarchy. The selection of ADV either ensures that this child is not the idle child in ADV's placement, or that ADV changes to a different marking placement, identical to the previous one up to the selection of the idle child of the current hierarchy, by invoking $\mathit{reset}()$. The descend continues in this manner until it reaches a hierarchy all of whose capacity is used up by ALG to accomodate its load (from above). At this point, the current hierarchy cannot place any files in S_i allowing ADV to create a request for a missing file. The strategy described so far is embodied the method $\mathit{weak\_descendant}()$. However, it may happen that in order to satisfy requests for missing files, ALG moves some of its load out of the current hierarchy through `fetch` operations. If the load on the current hierarchy decreases below its $\mathit{deactivation}$ threshold, ADV moves the current hierarchy, via $\mathit{heavy\_ancestor}()$, to the nearest ancestor whose load is at least as high as its deactivation threshold. In the process, ADV also invokes $\mathit{reset}()$ on the children of each hierarchy it ascended from. Thereafter, it starts the descend phase until it reaches a hierarchy with a missing file.

The fields $\mathit{primary}$ and $\mathit{secondary}$ of a hierarchy are mainly used for accounting for costs incurred by ALG. However, $\mathit{primary}$ is also used to preserve the invariant that ADV only descends to a non-idle child. Whenever ADV descends from the current hierarchy α to a child β with $\mathit{primary} = 0$, $\beta.\mathit{primary}$ is set to a non-zero value. If ADV ascends to α's parent, the $\mathit{primary}$ field of all children of α is set to 0 through an invocation of $\alpha.\mathit{reset}()$. Changing the $\mathit{primary}$ field of a child from 0 to a non-zero value corresponds to ADV revealing that that is not the idle child of the current hierarchy in ADV's placement. When $k - 1$ of α's children have nonzero $\mathit{primary}$ fields and ADV is poised to descend to the idle child of α, ALG has zeroed in on the identity of the idle child. At this point, ADV invokes $\alpha.\mathit{reset}()$ and again moves to a placement where any of the children of α could be the idle child.

Program

```
main()
(1)    $perch := \Pi$;
(2)    **while true**
(3)        $perch := perch.heavy\_ancestor().weak\_descendant()$;
(4)        $perch.request()$;

heavy_ancestor()
(1)    **if** $load() < deactivation()$
(2)        $secondary := reactivation()$;
(3)        $reset()$;
(4)        **return** $parent().heavy\_ancestor()$;
(5)    **return this**;

weak_descendant()
(1)    **if** $missing() \neq \emptyset$
(2)        **return this**;
(3)    **if** $(\exists \delta : \delta \in children() \wedge \delta.primary > 0 \wedge \delta.load() \geq reactivation())$
(4)        **return** $\delta.weak\_descendant()$;
(5)    **if** $pristine() = 1$
(6)        $reset()$;
(7)    Let $\delta : \delta \in children() \wedge \delta.primary = 0 \wedge \delta.load() \geq activation()$
(8)    $\delta.primary := \delta.secondary := activation()$;
(9)    **return** $\delta.weak\_descendant()$;
```

3.2 Correctness of the Program

We now show that that the adversarial program is well-defined and each round terminates with the generation of a request, i.e., ADV can generate an arbitrarily long sequence of requests given any online HCC algorithm. Lemma 3.1 shows that $heavy\_ancestor()$ returns an ancestor with $load \geq deactivation()$. Lemma 3.2 shows an invariant that establishes that the program is well-defined at steps 7 and 8 of $weak\_descendant()$, i.e., δ is not null at step 8. Lemmas 3.3 and 3.4 show that $weak\_descendant()$ returns an ancestor and the next request can be generated.

Lemma 3.1 *An invocation of $heavy\_ancestor()$ (i) terminates, (ii) returns an ancestor with $load() \geq deactivation()$.*

Proof: (i) When $heavy\_ancestor()$ is invoked on Π at depth 0, it terminates via step 5 as both $load()$ and $deactivation()$ are 0, by definition, for the root. When $heavy\_ancestor()$ is invoked on a hierarchy at depth i, it either terminates via step 5 or invokes $heavy\_ancestor()$ on its parent at depth $i - 1$. Hence, it terminates and returns an ancestor.

(ii) Follows from steps 1 and 5 of $heavy\_ancestor()$. □

Lemma 3.2 *The predicate $I = I_1 \wedge I_2 \wedge I_3$ is an invariant of the program, where I_1, I_2, and I_3 are:*
(i) I_1: A non-trivial hierarchy always has a child with $primary = 0$.
(ii) I_2: When $weak\_descendant()$ is invoked on a hierarchy, $load() \geq deactivation()$.
(iii) I_3: At step 7 of $weak\_descendant()$, δ is not null.

```
request()
(1)    Generate a request for $f$ at leftmost() where $f \in missing()$;
(2)    Serve the request.
(3)    Let ALG serve the request and arbitrarily modify its placement.

reset()
(1)    foreach $\alpha \in children()$
(2)        $\alpha.primary := \alpha.secondary := 0$;
```

Proof: It can be easily checked that I holds initially. It thus suffices to show that I is preserved by every step. We list below why every conjunct of I is preserved by every step.

(i) I_1: The only point in the code where the $primary$ field of a child is set to a non-zero value is in step 8 of $weak\_descendant()$. From I_2, δ is not null and step 8 of *weak_descendant*() is well-defined. After step 6, the number of children with $primary = 0$ is greater than 1 as $reset()$, if executed, sets it to k. Thus, step 8 is executed only when there is more than one child with $primary = 0$. Hence, the lemma.

(ii) I_2: The first invocation of *weak_descendant*() in an iteration of the loop in $main()$ is on a hierarchy returned by *heavy_ancestor*(). Hence, at this invocation, $load() \geq deactivation()$ from lemma 3.1(ii). Subsequent recursive invocations of $weak\_descendant()$ are on a child δ such that $\delta.load() \geq \delta.reactivation()$ (step 4) or $\delta.load() \geq \delta.activation()$ (step 9, well-defined by I_2). Since, by definition, $\delta.reactivation()$ and $\delta.activation()$ are both greater than $\delta.deactivation()$, the lemma holds for all invocations of $weak\_descendant()$.

(iii) I_3: Consider an invocation of $weak\_descendant()$ on a hierarchy at depth i. Let A denote the set of children with $primary = 0$ and B that with $primary > 0$. Let r denote $|A|$. From I_1, $r > 0$. By inspection of the code, step 7 is executed only if $missing()$ returned $\emptyset$ in step 1. Thus, $\sum_{\alpha \in A} \alpha.load() + \sum_{\alpha \in B} \alpha.load() = load() + |S_i|$. Also, by inspection of the code at step 3, $\alpha.load() < \alpha.reactivation())$ for all children α in B. Therefore, $\sum_{\alpha \in B} \alpha.load() < \sum_{\alpha \in B} \alpha.reactivation() < (k - r) \cdot (k^{d-i-2} \cdot \frac{i+2}{4})$. From invariant 3.2, $load() \geq deactivation() = k^{d-i-1} \cdot \frac{i}{4}$. By definition, $|S_i| = k^{d-i-1}$. Hence, $\sum_{\alpha \in A} \alpha.load() = load() + |S_i| - \sum_{\alpha \in B} \alpha.load() > r \cdot [k^{d-i-1} \cdot (\frac{i}{4k} + \frac{1}{2r}) + \frac{k^{d-i-2}}{2}]$ on simplifying and rearranging the terms. Thus, there exists a child α in A such that $\alpha.load() > k^{d-i-1} \cdot (\frac{i}{4k} + \frac{1}{2r}) = \alpha.activation()$. Hence, the lemma. □

Lemma 3.3 *An invocation of $weak\_descendant()$ (i) terminates, (ii) returns a hierarchy such that $missing()$ is non-empty.*

Proof: (i) Consider an invocation of $weak\_descendant()$ on a hierarchy at depth $4bk$. By lemma 3.2(ii), $load() \geq deactivation()$. By definition, for a hierarchy at depth $4bk$, $deactivation() = bk^{d-i} \geq b \cdot capacity()$. Thus, $load() \geq b \cdot capacity()$. Since ALG has a capacity blowup of at most b, $missing()$ returns a non-empty set (the set S_{4bk}) and the invocation terminates at step 2. Now, consider an invocation of $weak\_descendant()$ on a hierarchy at depth i for $i < 4bk$. By inspection of the code, this invocation either terminates via step 2 or recursively invokes $weak\_descendant()$ on a suitable child δ at depth $i + 1$ via step 4 or 9. Thus, by reverse induction on the depth of the hierarchy that $weak\_descendant()$ is invoked on, the lemma follows.

(ii) Follows from step 2 of $weak\_descendant()$.

Lemma 3.4 *In step 1 of request(), missing() returns a non-empty set.*

Proof: The method $request()$ is invoked only on a hierarchy returned by $weak\_descendant()$. Thus, by lemma 3.3(ii), $missing()$ returns a non-empty set in step 1 of $request()$. □

4 Cost Accounting

In this section, we show that there exists a nonempty family of offline HCC algorithms each of which serves the sequence of requests generated by the program given in the previous section and incurs a cost that is a factor $\Omega(\log d)$ less than that incurred by any b-feasible online HCC algorithm. The algorithm ADV introduced in the previous section is an algorithm in this family.

First, we list some properties that directly follow from the structure of the adversary's program. For proving properties about the recursive methods $heavy\_ancestor()$, $weak\_descendant()$, and $request()$, we introduce the notion of a *level invariant* analagous to the standard notion of a loop invariant for iterative programs. A *level* is defined as the set of instructions starting from the invocation of a method till the next recursive invocation of the method, if one exists, and the end of the method otherwise. A predicate P is a level invariant of a recursive method if execution of a level of the method preserves P.

Lemma 4.1 *For any hierarchy, (i) secondary is restricted to the values* $\{0, reactivation(), primary\}$*; (ii)* $primary = 0$ *or* $primary \geq reactivation()$.
Proof: (i) By inspection of the code, the only points where $secondary$ is modified are at step 2 of $heavy\_ancestor()$, step 8 of $weak\_descendant()$, and step 2 of $reset$ and the invariant is preserved in each one of the cases.

(ii) The only nonzero value assigned to $primary$ is at step 8 of $weak\_descendant()$ where it is set to $activation() \geq reactivation()$ by definition. □

Lemma 4.2 *Let* $P(\alpha)$ *denote the predicate that* $\beta.primary > 0$ *for all ancestors* β *of* α *except* Π. *Then, (i)* $P(\textbf{this})$ *is a level invariant of* $heavy\_ancestor()$ *and* $weak\_descendant()$*; (ii) Whenever* $heavy\_ancestor()$ *is invoked on a hierarchy other than* Π, $primary > 0$.
Proof: Assume that $P(\textbf{this})$ is true before an invocation of $weak\_descendant()$. If the current level terminates via step 2, $P(\textbf{this})$ trivially continues to hold at the end of the level. If not, $weak\_descendant()$ is invoked via step 4 or step 9 on a suitable child δ. In either case, by inspection of steps 3 and 8, $\delta.primary > 0$. Thus, $P(\textbf{this})$ is a level invariant of $weak\_descendant()$. In any level of $heavy\_ancestor()$, the $primary$ fields of ancestors remain unmodified and the level either terminates via step 5 or invokes $heavy\_ancestor()$ on its parent. Thus, $P(\textbf{this})$ is a level invariant of $heavy\_ancestor()$.

(ii) Assume that $P(perch)$ is true just before an invocation of $perch.heavy\_ancestor()$ in step 3 of $main()$. From lemma 4.2 and induction on the depth of recursion in the invocation of $heavy\_ancestor()$, $P(perch.heavy\_ancestor())$ is true. Similarly, by induction on the depth of recursion in an invocation of $weak\_descenda$
$P(perch.heavy\_ancestor().weak\_descendant())$ is true. Thus, step 3 of $main()$ preserves $P(perch)$. In the beginning of $main()$, $P(perch)$ is trivially true as $perch$ is initialized to Π. Thus, $P(perch)$ is always true just before step 3 of $main()$. Again, by induction on the depth of recursion, $P(\textbf{this})$ is true whenever $heavy\_ancestor()$ is invoked. Hence, (ii) follows. □

Lemma 4.3 *For any hierarchy,* $secondary \leq primary$.
Proof: Let P denote the predicate $primary \leq secondary$ for all hierarchies. The only points in $weak\_descendant()$ where the $primary$ or $secondary$ field of any hierarchy is modified is in step 6 through $reset()$ and step 8. In either case, P is a level invariant of $weak\_descendant()$, and by induction on the depth of recursion, P is preserved by $weak\_descendant()$. The only points in $heavy\_ancestor()$ where the $primary$ or $secondary$ field of any hierarchy is modifed is in step 2 and step 3 through $reset()$. Step 3 sets the $primary$ and $secondary$ fields of all children to 0 and hence does not affect P. Since $load() = deactivation() = 0$ by definition for Π, an invocation of $\Pi.heavy\_ancestor()$ never reaches step 2. Thus, by lemma 4.2(ii), $primary > 0$ at step 2. When $secondary$ is set to $reactivation()$ in step 2, P is preserved by lemma 4.1. Thus, P is a level invariant of $heavy\_ancestor()$, and by

induction, is preserved by any invocation of $heavy\_ancestor()$. Since $request()$ doesn't affect P, it is an invariant of the program. □

Definition 1 *A sequence $a_0, a_1, \ldots, a_r, 0 \leq r < k$, is defined to be i-active if and only if $a_j = g(i+1, k-j), 0 \leq j \leq r$.*

Lemma 4.4 *When $reset()$ is invoked on a hierarchy α, the non-zero primary fields of the children of α form an i-active sequence, where $i = \alpha.depth()$.*
Proof: : Let $P(\alpha)$ denote the predicate that the non-zero $primary$ fields of the children of α form an i-active sequence. For a child β of α, the only points in the code where $\beta.primary$ is modified are in $\alpha.reset()$, where it is set to 0, and in step 8 of $\alpha.weak\_descendant()$, where it is set to $\alpha.activation()$. At the beginning of $main()$, $P(\alpha)$ is trivially true as the $primary$ field of every hierarchy is 0. An invocation of $\alpha.reset()$ preserves $P(\alpha)$ as it sets the $primary$ fields of all the children of α to 0. Step 8 of $weak\_descendant()$ preserves $P(\alpha)$ as $\delta.primary$ is set to $\delta.activation()$ which by definition equals $g(i+1, k-j)$ where j is the number of siblings of δ with $primary > 0$. Thus, $P(\alpha)$ is preserved by every step except those in $\alpha.reset()$. It follows that $P(\alpha)$ is true before and after every step outside of $\alpha.reset()$. Hence, the lemma. □

Lemma 4.5 *Let $P(\alpha)$ denote the boolean condition that $\beta.secondary \leq \beta.reactivation()$ is true for all β that are not ancestors of α. Then, (i) $P($**this**$)$ is a level invariant of $heavy\_ancestor()$; (ii) $P($**this**$)$ is a level invariant of $weak\_descendant()$.*
Proof: (i) Assume that $P($**this**$)$ is true at the beginning of an invocation of $heavy\_ancestor()$. If the current level terminates via step 5, $P($**this**$)$ trivially continues to hold at the end of the level. If not, $secondary$ is set to $reactivation()$ in step 2. Subsequently, until step 4, $P(parent())$ is true at which point $parent().heavy\_ancestor()$ is invoked. Hence, (i) follows.

(ii) Assume that $P($**this**$)$ is true at the beginning of an invocation of $weak\_descendant()$. If the current level terminates via step 2, $P($**this**$)$ trivially continues to hold at the end of the level. If not, the current level terminates via step 4 or 9. For any child γ, $P($**this**$)$ implies $P(\gamma)$ irrespective of any modifications to $\gamma.secondary$. Thus, $P(\delta)$ is true at either of steps 4 or 9 just before the invocation of $\delta.weak\_descendant()$. Hence, (ii) follows. □

Lemma 4.6 *When $weak\_descendant()$ is invoked on a hierarchy α, $\beta.secondary \leq \beta.reactivation()$ for every β that is not an ancestor of α.*
Proof: : Let P be as defined in lemma 4.5. Assume that $P(perch)$ is true just before some invocation of $perch.heavy\_ancestor()$. From lemma 4.5(a) and induction on the depth of recursion in the invocation of $heavy\_ancesto$
$P(perch.heavy\_ancestor())$ is true. Similarly, from lemma 4.5(b) and induction on the depth of recursion in the invocation of $weak\_descendant()$, $P(perch.heavy\_ancestor().weak\_descendant())$ is true. Thus, step 1 of $main()$ preserves $P(perch)$. At the beginning of $main()$, $P(perch)$ is true as the $secondary$ field of every hierarchy is 0. Therefore, $P(perch)$ is true before every invocation of $perch.heavy\_ancestor()$. Again, inductively applying lemmas 4.5(a) and (b), the lemma to be proved follows. □

4.1 A Potential Function Argument

We use a potential function argument to show that there exists an offline algorithm that serves the requests generated by the program and incurs a cost that is at least a factor $\nu = \min(\frac{\lambda}{8}, \frac{\ln(k-1)}{4})$ less than the cost incurred by an online b-feasible algorithm.

We introduced a marking placement in Section 3. At any point in the execution of the adversarial program, the $primary$ and $secondary$ fields of hierarchies are said to represent the current state of the program. A marking placement is said to be *consistent* with the state of the program if the idle child of every non-trivial hierarchy has $primary = 0$. Two marking placements P and Q are said to be *adjacent* if they differ in the selection of the idle child at exactly one hierarchy.

Lemma 4.7 *The set of consistent marking placements is always non-empty*
Proof: From lemma 3.2(ii), every hierarchy always has at least one child with $primary = 0$. Thus, by definition, a consistent marking placement always exists. □

Lemma 4.8 *The cost incurred in moving from a marking placement P to an adjacent marking placement Q is at most $2k^{d-i-1} \cdot \alpha.diameter()$, where α is the hierarchy where P and Q have different idle children.*
Proof: Let β and γ be the the idle children of α in P and Q respectively. The cost incurred in moving from P to Q is the total cost of exchanging files placed in β and γ with each other. This cost is at most $2 \cdot \beta.capacity() \cdot \alpha.diameter() = 2k^{d-i-1} \cdot \alpha.diameter()$. □

Lemma 4.9 *Let P denote a consistent marking placement just before an invocation of $reset()$. Then, there exists an adjacent consistent placement Q just after the invocation.*
Proof: An invocation of $reset()$ on a hierarchy α sets the $primary$ fields of all of α's children to 0. A consistent marking placement P just before the invocation can be moved to a consistent marking placement Q at the end of the invocation by simply switching the idle child of α in P with that in Q. □

Lemma 4.10 *After m invocations of $reset()$, there exists a sequence of adjacent marking placements $P_1, \ldots, P_{m+1}$, such that P_i and P_{i+1} are consistent with the state of the program just before and after the i'th $reset()$.*
Proof: Follows from lemmas 4.10, 4.8 and inductive application of lemma 4.9. □

Let the sequence of placements maintained by ADV since the beginning of $main()$ be denoted by $P_0, P_1, \ldots, P_m$ where P_0 denotes the empty placement, $P_i, 1 \leq i \leq m$ are consistent marking placements, and P_m is the current placement. We call such a sequence an *execution sequence*. By lemma 4.10, such a sequence exists. We divide time into epochs where the end of an epoch and the beginning of the next is marked by an invocation of $reset()$ on any hierarchy. Initially, ADV moves from P_0 to P_1 by incurring a suitable cost. Thereafter, during an epoch, the placement remains unchanged, but while moving from one epoch to another, the current placement is moved to the next placement in the sequence. Next, we show that the total cost incurred by ADV in moving through such a sequence of placements is at least a factor ν less than any b-feasible online HCC algorithm.

A hierarchy α is associated with a potential function $\phi(\alpha)$ defined as follows:

$$\begin{aligned} \phi(\alpha) \quad = \quad & \sum_{\beta \notin Ancestors(\alpha)} \beta.parent().diameter()(\beta.primary - \beta.secondary + \beta.load()) \; + \\ & \sum_{\beta \in Ancestors(\alpha)} \beta.parent().diameter() \cdot \beta.primary \end{aligned} \tag{1}$$

We define an overall potential function Φ as follows:

$$\Phi = \phi(\alpha) + \nu \cdot T_{\text{ADV}} - T_{\text{ALG}} \tag{2}$$

where α is defined as $perch$ at points just before and after steps in $main()$ and as **this** during the execution of the methods $heavy\_ancestor()$, $weak\_descendant()$, and $request()$; T_{ALG} is the total cost incurred by ALG since the beginning of $main()$ and T_{ADV} is the total cost incurred by ADV after it moved to the first nonempty placement P_1 in its sequence. For convenience of exposition, we account for the one-time cost of moving from the empty placement P_0 to P_1 for ADV separately.

Lemma 4.11 *An invocation of $heavy\_ancestor()$ preserves the invariant $\Phi \leq 0$.*
Proof: : Consider the change in Φ in a single recursive level of execution of $heavy\_ancestor()$ on a hierarchy at depth i. If the level terminates via step 5, Φ remains unchanged. If not, from lemma 4.8, ADV incurs at most a cost $c_1 = diameter() \cdot 2k^{d-i-1}$ in moving to the next consistent marking placement in its execution sequence at the end of the invocation of $reset()$ in step 3. In this case, from step 1 and definitions of $load()$ and $deactivation()$, $i > 0$. At the end of the level, $parent().heavy\_ancestor()$ is invoked at step 4. From (1), the decrease in $\phi(\textbf{this})$ in this level equals $parent().diameter() \cdot (secondary - load())$ at the end of the level. From steps 1 and 2, this decrease is at least $parent().diameter() \cdot (reactivation() - deactivation()) = parent.diameter() \cdot k^{d-i-1} \cdot \frac{i}{4} \geq \frac{\lambda}{8} \cdot c_1 \geq \nu \cdot c_1$. Thus, Φ does not increase due to a single level of execution of $heavy\_ancestor()$, i.e., $\Phi \leq 0$ is a level invariant of $heavy\_ancestor()$. By induction on the depth of recursion in an invocation of $heavy\_ancestor()$, the lemma follows. □

Lemma 4.12 *An invocation of $weak\_descendant()$ preserves the invariant $\Phi \leq 0$.*
Proof: : Consider the change in Φ in a single recursive level of execution of $heavy\_ancestor()$ on a hierarchy at depth i. If the level terminates via step 2, Φ remains unchanged. If not, the program execution can take three paths each of which is analyzed below with respect to the resulting change in Φ. Note that ALG does not incur any cost in any of the cases.

(i) *Via step 4*: In this case, ADV does not incur any cost. At the end of the level, $weak\_descendant()$ is invoked on a suitable child δ. From (1), the decrease in $\phi(\textbf{this})$ in this level equals the value of $\delta.load() - \delta.secondary()$ at the end of the level. From step 3, $\delta.load()$ is at least $\delta.reactivation()$. From 4.6, $\delta.secondary()$ is at most $\delta.reactivation()$. Thus, the decrease in Φ in the level is non-negative.

(ii) *Via step 9 without executing step 6*: In this case, ADV does not incur any cost. At the end of the level, $weak\_descendant()$ is invoked on a suitable child δ. From (1), the decrease in $\phi(\textbf{this})$ in this level equals $\delta.load() - \delta.primary$ at the end of the level. From steps 7 and 8, $\delta.load()$ is at least $\delta.activation()$ and $\delta.primary$ is equal to $\delta.activation()$. Thus, the decrease in Φ in the level is non-negative.

(iii) *Via step 9 after executing step 6*: From lemma 4.8, ADV incurs a cost of at most $c_1 = diameter() \cdot 2k^{d-i-1}$ in moving to the next consistent marking placement in its execution sequence at the end of the invocation of $reset()$ in step 6. We only analyze steps 5 and 6 as the analysis of steps 7 and 8 is identical to the previous case. Invocation of $reset()$ sets the $primary$ fields of all the children to 0. From step 5 and lemma 4.4, the values of the $primary$ fields of the children form a sequence $g(i+1, k-r), 1 \leq r \leq k-1$, just before the invocation of $reset()$. Thus, in the decrease in $\phi(\textbf{this})$ in step 6 is $diameter() \cdot \sum_{0 \leq r \leq k-1} g(i+1, k-r) \geq diameter() \cdot \sum_{1 \leq r \leq k} k^{d-i-1} \cdot (\frac{1}{2r}) \geq diameter() \cdot k^{d-i-1} \cdot \frac{\ln(k-1)}{2} \geq \nu \cdot c_1$. Thus, the decrease in Φ is non-negative.

Thus, $\Phi \leq 0$ is a level invariant of $weak\_descendant()$. By induction on the depth of recursion in an invocation of $weak\_descendant()$, the lemma follows. □

Lemma 4.13 *An invocation of request preserves the invariant $\Phi \leq 0$.*
Proof: In $request$, a request is generated for a file f in $perch.missing()$. By lemma 3.4, such a file is guaranteed to exist. Thus, ALG incurs a cost of at least $parent.diameter() \geq \lambda \cdot diameter()$ and ADV incurs a cost of at most $diameter()$. In $request()$, $\phi(\textbf{this})$ increases by at most $parent().diameter()$. Thus, from (2), it follows that Φ does not increase. Hence, the lemma. □

Theorem 1 *The total cost incurred by* ALG *is at least a factor ν greater than that incurred by* ADV *after any number of requests.*
Proof: At the beginning of $main()$, $\Phi = 0$. From lemmas 4.11,4.12, and 4.13, $\Phi \leq 0$ is an invariant of the loop in $main()$. From fact 4.1(ii), it follows that the function ϕ is non-negative. Therefore, $T_{\text{ALG}} \geq \nu \cdot T_{\text{ADV}}$ is an invariant of the loop. Let C be the cost incurred by ADV in moving from the empty placement P_0 to the first

consistent marking placement P_1. The total cost, $T_{\text{ADV}} + C$, incurred by ADV can thus be made arbitrarily close to $\frac{T_{\text{ALG}}}{\nu}$ by appropriately increasing the length of the request sequence generated by the program. $\square$

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

5 Discussion

Because each set of files S_i is relevant to every depth-i hierarchy, our offline adversary makes widespread use of replication. Interestingly, this replication is not essential to our lower bound argument. In particular, rather than associating the same set of files S_i with each depth-i hierarchy, we can modify our argument by associating with each hierarchy α a unique set of files of size equal to $|S_i|$ where $i = \alpha.\mathit{depth}()$. With this modification, our lower bound argument continues to hold with essentially no changes. Thus our lower bound results are also applicable to the variant of the HCC problem in which no replication is allowed.

Cooperative caching as an idea 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 [9]. Although NUCA only supports a single processor at the time of this writing, multiprocessor NUCA is being developed, with data replication as a possibility [8]. As discussed in the previous paragraph, our lower bound is applicable to this setting whether or not replication is allowed.

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. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125 (or 119–126??), 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] J. Huh and C. K. Kim. Personal communication.

[9] 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.

[10] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38:260–302, 2001.

[11] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

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

[13] D. Wessels and K. Cla. RFC 2187: Appliation of Internet Cache Protocol, 1997.

[14] N. E. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.

[15] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web Caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, 1997.

A An Easy Upper Bound

We show in this section that, given a factor $2d$ blowup in capacity, where d is the depth of the hierarchy, a simple LRU-like algorithm 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.

The algorithm, which we refer to as the *Hierarchical Least Recently Used* (HLRU), works as follows. Let $\mathcal{P}$ denote the current placement maintained by the algorithm. Every cache u in HLRU is $2d$ times as big as the corresponding cache in OPT. HLRU divides every cache u into $2d$ equal-sized parts. For a hierarchy α at level i, the *logical cache* of α is defined to be the union of the ith part of all the caches below α. We use $\text{LRU}(\alpha, f)$ to denote an LRU algorithm that runs at the logical cache of α. $\text{LRU}(\alpha, f)$ adds f to the logical cache; it returns either the evicted file or **nil** if no file is evicted.

```
HLRU($\alpha, f$)
(1)     serve($\mathcal{P}, \alpha, f$);
(2)     repeat
(3)         $\alpha, f := \mathit{parent}(\alpha), \text{LRU}(\alpha, f)$
(4)     until $\alpha = \textbf{nil} \vee f = \textbf{nil}$
```

HLRU incurs two kinds of costs: retrieval costs and eviction costs. Retrieval costs account for the cost of creating a copy of a file at a local cache from an existing copy at remote cache through the `serve` operation in step 1. Eviction costs account for the cost of moving files via step 3 to logical caches at higher levels through a `fetch` at the destination followed a `discard` at the source. Hence, the retrieval cost can be thought of as moving files downwards and the eviction cost can be thought of moving files upwards. Since a file has to be moved down before being moved up and all caches start empty, the cost of moving files up is bounded by the cost of moving files down. Hence, to show that HLRU is constant competitive, it suffices to show the following lemma.

Lemma A.1 *The retrieval cost of HLRU is at most a constant factor of that of OPT.*

Proof Outline: We think of the top-level hierarchy as a tree with logical caches at interior nodes with an edge connecting a node to its parent. Thus, a hierarchy corresponds to a node of the tree. Let $\mathit{nf}(e, \text{ALG})$ denote the number of downward file movements generated by algorithm ALG along edge e. It suffices to show that for every edge e, $\mathit{nf}(e, \text{HLRU})$ is at most a constant factor of $\mathit{nf}(e, \text{OPT})$. Consider an arbitrary node α. The number of downward file movements on the edge e from $parent(\alpha)$ to α (if α is the root, then from the server to α) equals the number of file misses at α generated by an algorithm. For HLRU, a file miss occurs if the file is not in any logical cache within the subtree rooted at α. For OPT, a file miss occurs if the file is not in any physical cache below α. Let the total capacity of the physical caches below α of OPT be C. Hence, the capacity of the logical cache at α is $2C$. To facilitate the comparison between HLRU and OPT, consider a third unrealistic algorithm, XLRU, that uses the LRU algorithm and a single cache of capacity $2C$ to serve the requests generated by the physical caches below α of OPT and behaves the same as OPT at other caches. By the well-known result of Sleator and Tarjan [11], $nf(e, \text{XLRU}) \leq 2 \cdot nf(e, \text{OPT})$. We next compare $nf(e, \text{XLRU})$ and $nf(e, \text{HLRU})$. We make a key observation that, as an invariant, the files in the logical caches within the subtree rooted at α of HLRU is a superset of the files in the cache of XLRU. Suppose this is not true, then there is a first time that some file f is in XLRU but not in HLRU. By the HLRUalgorithm, this happens only if f is older than $2C$ other files, because the capacity of the logical cache at α is $2C$ and f is evicted from α. But f is in XLRU, which implies that f is one of the $2C$ most recently requested files. A clear contradiction. Hence, at all times, if a file is in XLRU, then it is in HLRU. It then follows that $nf(e, \text{HLRU}) \leq nf(e, \text{XLRU})$. Hence the lemma. $\square$

The proof outline above assumes that the hierarchies are well-separated, i.e., the diameter of the parent of any hierarchy is at least λ times its diameter, where $\lambda > 1$. However, by using a preprocessing phase similar to that used in [10], any ultrametric can be converted to a well-separated hierarchy with a constant factor distortion of distances. The following theorem then immediately follows from lemma A.1.

Theorem 2 *HLRU is constant competitive.*