Online Hierarchical Cooperative Caching

Xiaozhou Li[1] C. Greg Plaxton[2,3] Mitul Tiwari[2,4]
Arun Venkataramani[5]

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 the "least recently used" (LRU) paging algorithm 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? In this paper, we present a deterministic hierarchical generalization of LRU that is constant-competitive when the capacity blowup is linear in d, the depth of the cache hierarchy. Furthermore, we exhibit an infinite family of depth-d hierarchies such that any randomized hierarchical cooperative caching algorithm with capacity blowup b has competitive ratio $\Omega(\log \frac{d}{b})$ against an oblivious adversary. Thus, our upper and lower bounds imply a tight bound of $\Theta(d)$ on the capacity blowup required to achieve constant competitiveness.

[1] Microsoft Digital Anvil, 400 West Cesar Chavez Street, 4th Floor, Austin, TX 78701. Email: xili@microsoft.com. Most of this work was done while the author was a Ph.D. student at UT Austin supported by NSF Grant CCR–0310970.

[2] Department of Computer Science, University of Texas at Austin, 1 University Station C0500, Austin, Texas 78712–0233.

[3] Email: plaxton@cs.utexas.edu. Supported by NSF Grant CCR–0310970.

[4] Email: mitult@cs.utexas.edu. Supported by NSF Grant ANI–0326001.

[5] Computer Science Building, 140 Governors Drive, University of Massachusetts, Amherst, MA 01003-9264. Email: arun@cs.umass.edu. Most of this work was done while the author was a Ph.D. student at UT Austin supported by Texas Advanced Technology Project 003658–0503–2003 and by IBM.

1 Introduction

In the classic disk paging problem, which has been extensively studied, we are given a cache and a sequence of requests for pages. When a page is requested, we incur a miss if it is not already present in the cache. In the event of a miss, we are required to load the requested page into the cache, which may necessitate the eviction of another page. Our goal is to minimize the cost of processing the request sequence, where the cost is defined as the number of misses incurred. A caching algorithm is *online* if it processes each successive request with no knowledge of future requests. A caching algorithm is *offline* if it is given the entire request sequence in advance.

An online algorithm is c-*competitive* if, for all request sequences τ, the cost incurred by the online algorithm to process τ is at most c times that incurred by an optimal offline algorithm. In the seminal paper introducing the notion of competitive analysis, Sleator and Tarjan [13] show that LRU (Least-Recently-Used) and several other online deterministic caching algorithms are $\frac{k}{k-h+1}$-competitive, where k is the cache capacity of the online algorithm and h is the cache capacity of the offline algorithm. They also show that $\frac{k}{k-h+1}$ is the best competitive ratio that can be achieved by any deterministic online caching algorithm. Young [16] proposes the LANDLORD algorithm that achieves competitive ratio $\frac{k}{k-h+1}$ for a case where the files being cached have nonuninform sizes and retrieval costs. Note that LRU and LANDLORD are constant-competitive assuming a constant-factor capacity blowup over the corresponding optimal offline algorithm.

In cooperative caching [9], 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 [7] introduces the notion of a hierarchical arrangements of caches. Harvest uses the Internet Cache Protocol [15] to support discovery and retrieval of documents from other caches. The Harvest project later became the public domain Squid cache system [14]. Adaptive Web Caching [17] builds a mesh of overlapping multicast trees; the popular files are pulled down towards their users from their origin servers. In local-area network environments, the xFS [1] system utilizes workstations cooperating with each other to cache data and to provide serverless file system services.

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 where the cost of communication among caches is specified by an ultrametric distance function, the precise definition of which is given in Section 2. An ultrametric corresponds to a kind of hierarchical distance function. For this reason, we call the cooperative caching problem in networks with ultrametric distance function the *hierarchical cooperative caching (HCC) problem*. This is an important problem because many actual networks have a hierarchical or approximately hierarchical structure. Furthermore, various caching schemes [7, 8, 14] for a wide area network suggest arranging caches hierarchically. Therefore, we believe that an ultrametric is appropriate for modeling the distance function among caches distributed over a wide area network.

Ultrametrics are equivalent up to a constant-factor to the hierarchically well-separated tree

(HST) metrics, as introduced by Bartal [3]. Refining earlier results by Bartal [3, 5], Fakcharoenphol et al. [10] have shown that any metric space can be approximated by the HST metrics with a logarithmic distortion. Hence, many results for the HST metrics imply corresponding results for arbitrary metric spaces, at the expense of an extra logarithmic factor.

For the case where the access distribution of each file at each cache is fixed and known in advance, Korupolu et al. [12] provide a polynomial-time algorithm for the HCC problem that minimizes the average retrieval cost and does not require a capacity blowup. In addition, they provide a faster constant-factor approximation algorithm that does not require a capacity blowup. On the other hand, the assumption in Korupolu et al. [12] of a fixed access distribution is rather strong. Furthermore, even in applications where the access distribution is relatively stable, it may be expensive to track.

Since the HCC problem generalizes the disk paging problem mentioned earlier, we cannot hope to achieve constant competitiveness for the HCC problem without at least a constant-factor capacity blowup. Our main motivation in pursuing the present research has been to determine whether there exists a constant-competitive algorithm with a constant-factor capacity blowup for the HCC problem. Since the LANDLORD algorithm by Young is designed for files with non-uniform retrieval cost, one could think of applying LANDLORD to solve the HCC problem. However, simply running LANDLORD at each cache does not provide a good competitive ratio for the HCC problem, since LANDLORD is not designed to exploit the benefits of cooperation among caches. As stated in Young [16], 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 this paper, we show that if an online algorithm is given a sufficiently large capacity blowup, then constant competitiveness can be achieved. In Section 4, we present a deterministic hierarchical generalization of LRU that is constant-competitive when the capacity blowup is linear in d, the depth of the cache hierarchy. We content ourselves by dealing with files of unit sizes only. However, a hierarchical generalization of LANDLORD can be used to deal with files of nonuniform sizes.

Furthermore, we exhibit an infinite family of depth-d hierarchies such that any randomized online HCC algorithm with a capacity blowup b has competitive ratio $\Omega(\log \frac{d}{b})$ against an oblivious adversary. In particular, we construct a hierarchy with a sufficiently large depth and show that an oblivious adversary can generate an arbitrarily long request sequence such that the randomized online HCC algorithm incurs a cost $\Omega(\log \frac{d}{b})$ times that of an optimal offline algorithm. In terms of n, the number of caches, our lower bound result shows that the competitive ratio of any randomized HCC algorithm is $\Omega(\log \log n - \log b)$. Our upper and lower bounds imply a tight bound of $\Theta(d)$ on the capacity blowup required to achieve constant competitiveness.

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 (e.g., see 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 the cost accessing a file at distance d is equal to the cost of migrating the file a distance of d. 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] show that, given polylog(n, Δ) capacity blowup, there exists a deterministic polylog(n, Δ)-competitive algorithm for general networks, where Δ is the normalized diameter of the network. For the constrained file migration problem, if we let m denote the total capacity of the n caches, Bartal [3] gives a deterministic lower bound of $\Omega(m)$, a randomized lower bound of $\Omega(\log m)$, and a randomized upper bound of $O((\log m)\log^2 n)$. Applying the recent result of Fakcharoenphol et al. [10], the latter upper bound may be improved to $O((\log m)\log n)$.

The rest of this paper is organized as follows. Section 2 provides some preliminary definitions. Section 3 presents our lower bound. Section 4 presents our upper bound.

2 Preliminaries

Assume that we are given a set of caches, each with a specified nonnegative capacity, and a distance function h that specifies the cost of communication between any pair of caches. Such a distance function is a *metric* if it is nonnegative, symmetric, satisfies the triangle inequality, and $h(u, v) = 0$ if and only if $u = v$ for all caches u and v. A metric distance function h is an *ultrametric* if $h(u, v) \leq \max(h(u, w), h(v, w))$ for all caches u, v, and w; note that the latter condition subsumes the triangle inequality.

We now describe another method to specify a distance function over a set of caches. In this method, the distance function is encoded as a rooted tree where each node of the tree has an associated nonnegative diameter. There is a one-to-one correspondence between the set of caches and the leaves of the tree, and each leaf has a diameter of zero. The diameter of any node is required to be less than that of its parent. The distance between two caches is then defined as the diameter of the least common ancestor of the corresponding leaves. It is well-known (and easy to prove) that a distance function can be specified by a tree in this manner if and only if it is an ultrametric. We say that such a tree is λ-*separated*, where $\lambda > 1$, if the diameter of any node is at least λ times that of any of its children.

In all of the caching problems addressed in this paper, we assume that the distance function specifying the cost of communication is an ultrametric, and we adopt the tree view of an ultrametric discussed in the preceding paragraph. The main advantage of this view is that it enables us to leverage standard tree terminology in our technical arguments. Table 1 lists a number of useful definitions based on tree terminology.

Notation	Meaning
$root$	the root of the tree
$\alpha.parent$	the parent of α, where $\alpha \neq root$
$\alpha.ch$	children of α
$\alpha.anc$	the ancestors of α (including α)
$\alpha.desc$	the descendants of α (including α)
$\alpha.depth$	the depth of α, where the root is considered to be at depth 0
$\alpha.diam$	the diameter of α
$\alpha.caches$	the set of caches in the subtree rooted at α
$\alpha.cap$	the total capacity of the caches in $\alpha.caches$

Table 1: Some useful notation. The variable α refers to a tree node.

In the caching problems addressed in this paper, we refer to the objects to be cached as *files*. The files are assumed to be read-only, so we do not need to deal with the issue of consistency maintenance. Each file is assumed to be indivisible; we do not consider schemes in which a copy of a file may be broken into fragments and spread across multiple caches. Each file f has a specified size, denoted $size(f)$, and penalty, denoted $penalty(f)$. We assume that any file can fit in any cache, that is, the maximum file size is assumed to be at most the minimum cache capacity. The penalty associated with a file represents the cost per unit size to retrieve the file when it is not stored anywhere in the tree of caches, and is assumed to exceed the diameter of the tree.

A copy is a pair (u, f) where u is a cache and f is a file with size at most the capacity of u. 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 the total size of the files placed in any cache is at most b times the capacity of the cache.

A caching algorithm maintains a placement. Initially, the placement is empty. Two basic operations, *delete* and *add*, may be used to update a given placement P. A delete operation removes a copy from P; the algorithm incurs no cost for such a deletion. In an add operation, a copy (u, f) is added to P. If, prior to the add, P does not place a copy of f at any cache, then the cost of the add is defined to be $penalty(f)$. Otherwise, the cost is $size(f) \cdot dist(u, v)$, where v is the closest cache at which a copy of f is placed. A caching algorithm A is b-feasible if it always maintains a b-feasible placement.

A *request* is a pair (u, f) where u is a cache and f is a file. To process such a request, a caching algorithm performs an arbitrary sequence of add and delete operations, subject only to the constraint that (u, f) belongs to at least one of the placements traversed.

The HCC problem is to process a given sequence of requests with the goal of minimizing cost. For any (randomized) HCC algorithm A, and any request sequence τ, we define $T_A(\tau)$ as the (expected) cost for A to process τ. An online HCC algorithm A is c*-competitive* if for all request sequences τ and 1-feasible HCC algorithms B, $T_A(\tau) \le c \cdot T_B(\tau)$. (Remark: The asymptotic bounds established in this paper are unchanged if we allow an additive slack in the definition of c-competitiveness, as in [6, Chapter 1].)

An HCC algorithm is b-quasifeasible if it maintains a b-feasible placement before and after processing a request, and while processing a request, removal of at most one copy of a file from its placement makes its placement b-feasible. Observe that any b-quasifeasible HCC algorithm is a $(b + 1)$-feasible HCC algorithm. In Section 4, we present a deterministic constant-competitive $O(d)$-quasifeasible online HCC algorithm; by the preceding observation, our algorithm is $O(d)$-feasible.

An HCC algorithm is *nice* if on a request (u, f), it first adds a copy (u, f) to its placement and then performs an arbitrary sequence of add and delete operations.

Observe that a nice $(b + 1)$-feasible HCC algorithm A can simulate any b-feasible HCC algorithm B by first retrieving the requested copy, and then exactly following the steps of algorithm B. In performing this simulation, algorithm A incurs at most twice the cost of algorithm B. Hence, any b-feasible c-competitive HCC algorithm can be converted into a nice $(b + 1)$-feasible $2c$-competitive HCC algorithm. (Remark: With additional care it may be possible to argue that the factor of 2 appearing in the preceding observation can be eliminated.) In Section 3 we prove that for any nice b-feasible randomized online HCC algorithm A, there exists a request sequence τ and a 1-quasifeasible offline HCC algorithm B such that $T_A(\tau) = \Omega(\log \frac{d}{b}) \cdot T_B(\tau)$. By the foregoing observation, together with that of the preceding paragraph, we can conclude that the competitive ratio of any b-feasible randomized online HCC algorithm is $\Omega(\log \frac{d}{b})$.

3 The Lower Bound

In this section, we present a lower bound that holds for an arbitrary nice randomized b-feasible online HCC algorithm ON, where b is a positive integer. The lower bound holds for ON with respect to the trees drawn from an infinite family of k-ary, depth d trees, parameterized by integers d and k such that $d = 8bk - 1$. We denote a tree from this family by $T(d, k)$. Furthermore, the diameter of an internal node, α, of tree $T(d, k)$ is set to be λ^{d-i-1}, where $\lambda = \max(\frac{15}{7}, \Omega(\log k))$, $i = \alpha.depth$, and $0 \leq \alpha.depth < d$. Recall from the previous section that the diameter of a leaf is 0. For any file f placed in $T(d, k)$, $penalty(f)$ is set to be $\lambda \cdot root.diam$. Note that $T(d, k)$ is a λ-separated tree.

The lower bound discussed above is established in Sections 3.2 through 3.8. Section 3.1 illustrates some of the central ideas to be used in the proof of the lower bound by addressing a simpler problem. It is not necessary to read Section 3.1 before continuing to Section 3.2, but it might provide some useful intuition.

3.1 A Simple Lower Bound Result

In the present section, we restrict attention to instances of the HCC problem satisfying the following characteristics.

- There are n caches, each with capacity ℓ, where ℓ is a positive integer.
- Each pair of distinct caches are unit distance apart. Note that such a uniform metric space is a special case of an ultrametric; it corresponds to a one-level hierarchy in which the root (which does not have an associated cache) is at distance $1/2$ to each of its n children (the caches), and the shortest path between any pair of caches goes through the root.
- There are two sets of ℓ unit-sized files X and Y, and every access is to a file in $X \cup Y$.
- Each file in $X \cup Y$ has the same associated penalty $\rho = \Omega(\log n)$.

In addition, we make the following simplifying assumptions.

- The online algorithm is deterministic. We make this assumption primarily for ease of presentation. The $\Omega(\log n)$ lower bound presented in this section is easily generalized to hold for randomized online algorithms. The proof of our main lower bound is generalized to handle the randomized case.
- The capacity blowup b afforded to the online algorithm is 1. Our assumption that $b = 1$ allows for an easy $\Omega(\ell)$ lower bound argument, since the adversary can restrict all accesses to a single cache, and the results of Sleator and Tarjan [13] then imply an $\Omega(\ell)$ lower bound. But the results of Sleator and Tarjan also imply that if we allow b to be a constant greater than one, this approach cannot yield a non-constant lower bound. In fact, as we show in Section 4,

it is possible to give a constant-competitive algorithm for the general HCC problem if the capacity blowup b is at least the depth of the hierarchy. Since we are currently focusing on the special case of a constant-depth hierarchy, we cannot hope to establish a non-constant lower bound on the competitive ratio while allowing an arbitrary constant capacity blowup b.

Given the foregoing assumptions, we now sketch a proof of an $\Omega(\log n)$ lower bound on the competitive ratio. At the end of this section, we briefly indicate how the ideas used to prove this $\Omega(\log n)$ lower bound are generalized in Sections 3.2 through 3.8 to establish our main lower bound for the HCC problem.

Fix an online algorithm A. Our objective is to produce a request sequence σ and an offline algorithm B such that A's cost to serve σ is $\Omega(\log n)$ times that of B. Initially, the offline algorithm B loads one of its caches with the files in the set X, and loads each of the remaining caches with the files in the set Y. Note that the offline algorithm B incurs $\Theta((\rho + n)\ell)$ cost to establish this initial configuration. We generate a sufficiently long request sequence σ so that the cost incurred by A exceeds the initial configuration cost of B by an $\Omega(\log n)$ factor. This allows us to ignore the initial configuration cost of B in the remainder of this proof sketch.

The offline algorithm B maintains the invariant that one cache holds the set of files X, and every other cache holds the set of files Y. Typically, the configuration of B does not change when a read request is processed. Occasionally, B selects a new cache to hold X, and updates its configuration accordingly, paying $\Theta(\ell)$ cost. These updates to B's configuration partition time into epochs. Within each epoch, we ensure that the online algorithm A's cost to service each request is $\Omega(\log n)$ times that of B. Furthermore, we ensure that the total cost paid by A within each completed epoch is $\Omega(\ell \log n)$. Since the cost required by B to update its configuration at the end of each epoch is $\Theta(\ell)$, the desired lower bound follows.

To ensure that the online algorithm A's cost to service each request within an epoch is $\Omega(\log n)$ times that of the offline algorithm B, B maintains a partition of the n caches into two sets U and V. At the beginning of the epoch, all of the caches are in the set U. During the epoch, B periodically shifts caches from U to V. The epoch ends when there is exactly one cache remaining in U. The offline algorithm B ensures that the cache belonging to U throughout the epoch is the one that stores X in this epoch. Consequently, during the epoch, a request for a file in Y at a cache in V costs B nothing. At a general point within the epoch, the next request to be appended to σ is determined in the following manner.

- First, if some file x in X is not stored in any of A's caches, then a request is generated for x at a particular fixed cache, say the cache with the lowest numerical identifier. In this case A pays at least $\rho = \Omega(\log n)$ to service the request, while B pays at most one unit.

- Second, if there exists a file y in Y that is not stored by A at some cache v in V, then we generate a request for y at v. In this case A pays at least one unit to service the request, while B pays zero.

- Otherwise, there exists (by a simple averaging argument) a cache u in U at which A stores at least $|X|/|U|$ files belonging to the set X. The offline algorithm shifts u from U to V, and a request is generated for a file y in Y that A does not currently store at u. As argued in the next paragraph, as long as $|U| > 1$, we are able to ensure that the cache u shifted to V is not the cache that the offline algorithm B is using to store X in the current epoch. Therefore, A pays at least one unit to service the request, while B pays zero.

Therefore, in every case, A's cost to service a request within an epoch is $\Omega(\log n)$ times that of B.

To maintain the invariant that U contains the cache w currently used by the offline algorithm B to store X, w is chosen in an adversarial manner. Informally, the online algorithm is forced to play a shell game in each epoch, where the shells are the n caches and the online algorithm makes successive guesses as to the identity of w. Since the online algorithm is deterministic, w can be chosen to be the nth guess of the online algorithm.

It remains to prove that the total cost paid by the online algorithm A within each completed epoch is $\Omega(\ell \log n)$. As we have argued above, A stores at least $|X|/|U|$ files belonging to the set X in cache u when u is shifted to V. Since no other cache is shifted to V until A stores all of Y in u, A incurs a cost at least $|X|/|U|$ before V grows again. Thus the total cost incurred by A during a completed epoch is at least $|X| \cdot \sum_{2 \le i \le n} 1/i = \Omega(\ell \log n)$, establishing the desired lower bound.

In the sections that follow, we establish our main lower bound by generalizing the shell game described above to ultrametrics corresponding to complete, regular trees of non-constant depth. The basic intuition is that the online algorithm is forced to play many shell games in parallel, one corresponding to each node of the tree. The adversarial nature of the shell games ensures that the online algorithm generally makes incorrect guesses related to the hierarchical configuration maintained by the offline algorithm. These incorrect guesses erode the capacity advantage b enjoyed by the online algorithm: By employing a sufficiently deep tree, the online algorithm can be forced to devote all of the space in certain caches to files associated with such incorrect guesses. The next request is introduced at such a cache.

3.2 ADV **Algorithm**

In Figure 1, we present an algorithm for an *oblivious adversary* [6, Chapter 4], ADV, that constructs a request sequence σ of any given length N. The following definitions are useful for developing the ADV algorithm.

- For any nonnegative integer i and positive integer j, let
$$g(i,j) = k^{d-i} \cdot \left(\frac{i-1}{8k} + \frac{1}{4j} \right).$$
- For any node α, we define associated "reactivation" and "deactivation" values $\alpha.react = g(\alpha.depth, k)$ and $\alpha.deact = g(\alpha.depth, 2k)$.
- We define the "activation" value of α, denoted $\alpha.act$, as $g(\alpha.depth, r)$ where $r = |\{\beta : \beta \in \alpha.parent.ch : \beta.x = 0\}|$. Note that $\alpha.act$ is a function of the program state, since it depends on the values of certain program variables (i.e., $\beta.x$ for all β in $\alpha.parent.ch$).
- We fix $d+1$ disjoint sets of unit-sized files $F(i)$, $0 \le i \le d$, such that $|F(i)| = \lceil k^{d-i-1} \rceil$ for $0 \le i \le d$. Each request in the request sequence σ generated by ADV involves a file drawn from these sets.

{initially, $N \geq 0$, $count = 0$, $\pi = root$, $root.x = root.y = root.act = g(0, k)$,
$\alpha.x = \alpha.y = 0$ for all $\alpha \neq root$, and σ is empty}
1 **while** $count < N$ **do** {main loop}
2 **while** $\pi.load < \pi.deact$ **do** {up loop}
3 $\pi.y := \pi.react$;
4 for every child δ of π, set both $\delta.x$ and $\delta.y$ to 0;
5 $\pi := \pi.parent$
6 **od**; {end of up loop}
7 **while** $\pi.missing = \emptyset$ **do** {down loop}
8 **if** a child δ of π satisfies $\delta.x > 0 \wedge \delta.load \geq \delta.react$ **then**
9 $\pi := \delta$
10 **else**
11 **if** π has exactly one child with x equal to 0 **then**
12 for every child δ of π, set both $\delta.x$ and $\delta.y$ to 0
13 **fi**;
14 $\pi :=$ a child δ of π such that $\delta.x = 0 \wedge \delta.load \geq \delta.act$;
15 set both $\pi.x$ and $\pi.y$ to $\pi.act$
16 **fi**
17 **od**; {end of down loop}
18 append to σ a request for an element in $\pi.missing$ at an arbitrary cache in $\pi.caches$;
19 $count := count + 1$
20 **od** {end of main loop}

Figure 1: The ADV algorithm. We remark that the y field maintained at each node has no impact on the computation of the request sequence σ. (To see this, note that the y field is written, but never read.) The y field has been introduced to facilitate our analysis.

- We define $\alpha.placed$ as the set of distinct files placed by ON in $\alpha.caches$ after processing the request sequence given by the program variable σ. Since ON is a randomized algorithm, $\alpha.placed$ is a random set.
- We define $\alpha.load$ as the expected value of $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed|$.
- We define $\alpha.missing$ as the set of all files f in $F(\alpha.depth)$ and $\Pr(f \in \alpha.placed) \leq \frac{1}{2}$.

Algorithm ADV is oblivious since it constructs the request sequence without examining the random bits used by ON during its execution. Whenever line 18 of ADV is executed, a request is appended to σ and ON processes this request. The main technical result to be established in this section is that, for N sufficiently large, $T_{\text{ON}}(\sigma)$ is $\Omega(\log \frac{d}{b})$ times $T_A(\sigma)$ for an optimal 1-quasifeasible offline algorithm A.

3.3 Correctness of ADV

We show in this section that ADV is well-defined (i.e., $\pi \neq root$ just before line 5, 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 x field equal to 0, I_2 denote that π is a 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 outside 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 $I_1 \wedge I_2 \wedge I_3$.

Each line of code outside the down loop preserves I_1 because such lines do not assign a nonzero value to an 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. It follows that every line of code outside the down loop preserves $I_1 \wedge I_2$.

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 x field equal to 0 just before line 14. Hence, if $I_1 \wedge I_3$ holds before line 15, then I_1 holds after line 15.

We now argue that for each line of code in the down loop if $I_1 \wedge I_2 \wedge I_3$ holds before execution of the line then I_2 holds after execution of the line. It is sufficient to prove that $\pi.depth < 8bk - 1$ (i.e., π is not a leaf) just before line 8 and that if the assignment statement of line 14 is executed, the RHS is well-defined (i.e., some child δ of π satisfies $\delta.x = 0$ and $\delta.load \geq \delta.act$). To establish the former claim, let us assume to the contrary that $\pi.depth = 8bk - 1$ just before line 8. (Note that I_2 implies that $\pi.depth$ cannot take on a higher value.) By the guard of the down loop, the probability that $\pi.placed$ contains the lone file in $F(8bk - 1)$ is at least $\frac{1}{2}$. Furthermore, I_3 implies that $\pi.load \geq \pi.deact = g(8bk-1, 2k) = b - \frac{1}{8k}$. It follows that the expected number of files stored by ON in the cache associated with the leaf π is at least $b - \frac{1}{8k} + \frac{1}{2} > b$, which is a contradiction since $\pi.cap = 1$ and ON is b-feasible. Hence, $\pi.depth < 8bk - 1$ just before line 8.

We now argue that if the assignment statement of line 14 is executed, the RHS is well-defined. 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

$$
\begin{aligned}
& \sum_{\alpha \in A} \alpha.load \\
= \; & \sum_{\alpha \in \pi.ch} \alpha.load - \sum_{\beta \in B} \beta.load \\
\geq \; & \pi.load + \frac{|F(i)|}{2} - \sum_{\beta \in B} \beta.load \\
\geq \; & \pi.deact + \frac{|F(i)|}{2} - \sum_{\beta \in B} \beta.react \\
= \; & g(i, 2k) + \frac{k^{d-i-1}}{2} - \sum_{\beta \in B} g(i+1, k) \\
= \; & k^{d-i} \cdot \frac{i}{8k} + \frac{k^{d-i-1}}{2} - (k-r) \cdot k^{d-i-1} \cdot \frac{(i+2)}{8k} \\
= \; & r \cdot k^{d-i-1} \cdot \left(\frac{i}{8k} + \frac{1}{4r} + \frac{1}{4k} \right).
\end{aligned}
$$

(In the derivation above, the first inequality is due to the definition of $load$ and the guard of the down loop, i.e, for each file f in $|F(i)|$, $\Pr(f \in \alpha.placed) > \frac{1}{2}$, and the second inequality is due to I_3 and the guard of the outer **if** statement. Formula for $|F(i)|$ is valid in the second equality since $i < 8bk - 1$.) Hence, by an averaging argument (note that $r > 0$ by I_1), there exists a child δ of π such that $\delta.x = 0$ and

$$
\begin{aligned}
& \delta.load \\
\geq \; & k^{d-i-1} \cdot \left(\frac{i}{8k} + \frac{1}{4r} \right) \\
= \; & \delta.act.
\end{aligned}
$$

Hence, the RHS of line 14 evaluates to a node.

Recall that we wish to show that every line of code in the down loop preserves $I_1 \wedge I_2 \wedge I_3$. Thus far we have established that for each line of code in the down loop, if $I_1 \wedge I_2 \wedge I_3$ holds before execution of the line then $I_1 \wedge I_2$ holds after execution of the line. It remains to show that for each line of code in the down loop, if $I_1 \wedge I_2 \wedge I_3$ holds before execution of line then I_3 holds after execution of the line. The only lines in the down loop that affect I_3 are 9 and 14. By I_2, π is a node and by definition, $\alpha.react \geq \alpha.deact$ and $\alpha.act \geq \alpha.deact$ for all α. Hence, if $I_2 \wedge I_3$ holds before lines 9 and 14, then I_3 holds after both of these lines.

This completes our proof of the lemma. □

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.*

Proof: Every iteration of the down loop moves π to one of its children. By I_2 of Lemma 3.1, π is always a well defined node. Hence, the down loop terminates. □

Lemma 3.4 *After generating a sequence σ of N requests,* ADV *terminates.*

Proof: Follows from Lemmas 3.2 and 3.3. □

3.4 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 3.5 *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 3.6 *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 3.7 *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 X denote $\pi.anc$ and let Y denote the set of nodes that are neither in X nor children of the nodes in X.

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 X, adds $\pi.ch$ to Y, 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 X and removes $\pi'.ch$ from Y. 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 leaves X and Y unchanged and only changes the x value of the nodes in neither X nor Y. Then lines 14 and 15 preserves P because they add π', which has a positive x value after line 15, to X and removes $\pi'.ch$ from Y. Hence, it preserves P.

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

Lemma 3.8 *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 3.7 and the guard of the up loop that just before line 3, $\pi \neq root$ and $\pi.x > 0$. It then follows from Lemmas 3.5 and 3.6 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 3.9 *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 3.10 *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 $P(\pi)$. □

3.5 Colorings

In order to facilitate the presentation of an offline algorithm in Section 3.6, we introduce the notion of colorings in this section and the notion of consistent placements in the next.

A *coloring* of $T(d,k)$ (recall that $T(d,k)$ is the tree of caches) is an assignment of one of the colors {white, black} to every node in $T(d,k)$ 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.)

3.6 Consistent Placements

A placement is *colorable* if there exists a coloring C such that: (1) for each white internal node α of $T(d,k)$, the set of files $F(\alpha.depth)$ are stored in (and fill) the caches associated with the unique black child of α; (2) for each white leaf α of $T(d,k)$, the (singleton) set of files $F(\alpha.depth)$ 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 P and any pair of siblings α and β, we define $swapp(P,\alpha,\beta)$ (swap placement) as the placement obtained from P by exchanging the contents of each cache in α with that of the corresponding cache in β. (It is convenient to assume that the children of each node in $T(d,k)$ are ordered from left to right. This induces an overall left to right ordering of $\alpha.caches$ and $\beta.caches$. For all i, the ith cache in $\alpha.caches$ corresponds to the ith cache in $\beta.caches$.) Note that for any colorable placement P with associated coloring C and any pair of sibling nodes α and β, the placement $swapp(P,\alpha,\beta)$ is colorable, and its associated coloring is $swapc(C,\alpha,\beta)$.

3.7 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 the 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 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 a 1-quasifeasible 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 $swapp(P_{\text{OFF}},\alpha,\beta)$ 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. Whenever line 18 is executed, a request is generated and the algorithm OFF uses the placement P_{OFF} to process this request. On a request (u,f), if there is not already a copy of file f at u, OFF creates a copy (u,f) in order to process the request and then immediately discards the copy. Note that the capacity constraint can be violated at u by one unit when the copy (u,f) is created, but the capacity is satisfied before processing the next request. Hence, the placement P_{OFF} remains the same before and after line 18, and P_{OFF} is updated only at lines 4 and 12.

Lemma 3.11 *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. $\square$

Lemma 3.12 *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 3.7, $\pi.x$ 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 3.7, 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 3.13 *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 3.7, $\pi'.x > 0$ and hence π' is white before line 15. Therefore, by construction of ADV, $\pi'.last$ 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 line 15. 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 3.14 *The placement P_{OFF} is always consistent.*

Proof: Lines 4, 12, and 15 are the only lines that can affect the consistency of C_{OFF} since they are the only lines that modify the $last$ field or the x field of any node. From Lemmas 3.12 and 3.13, these lines preserve the consistency of C_{OFF}. From Lemma 3.11 it follows that P_{OFF} is always consistent. □

3.8 A Potential Function Argument

In this section, we use a potential function argument to show that ON is $\Omega\left(\frac{\nu}{\nu'}\right)$-competitive, where

$$\nu = \min\left(\frac{\lambda}{16}, \frac{\ln k}{8} - \frac{1}{8}\right)$$

and $\nu' = \frac{\lambda}{\lambda-1}$. Let $T'_{\mathrm{OFF}}(\sigma)$ denote the total cost incurred by OFF to process request sequence σ, except that we exclude from $T'_{\mathrm{OFF}}(\sigma)$ 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:

$$\begin{aligned}\Phi &= \nu \cdot T'_{\mathrm{OFF}}(\sigma) - \nu' \cdot T_{\mathrm{ON}}(\sigma) + \\ &\quad \sum_{\alpha \in \pi.anc \wedge \alpha \neq root} \alpha.parent.diam \cdot \alpha.x + \\ &\quad \sum_{\alpha \notin \pi.anc} \alpha.parent.diam \cdot (\alpha.x - \alpha.y + \alpha.load)\end{aligned} \tag{1}$$

Lemma 3.15 *The cost incurred by $swapp(P, \alpha, \beta)$ 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 3.16 *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 3.8, line 4 does not increase Φ. Let $i = \pi.depth$. By Lemma 3.15, 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 placement to the next. Thus, the total change in Φ in an iteration is at most

$$\begin{aligned} & \nu \cdot c - \pi'.diam \cdot (\pi.y - \pi.load) \\ \leq\ & \nu \cdot c - \pi'.diam \cdot (\pi.react - \pi.deact) \\ =\ & \nu \cdot c - \pi'.diam \cdot (g(i,k) - g(i,2k)) \\ =\ & \nu \cdot c - \pi'.diam \cdot k^{d-i-1} \cdot \frac{1}{8} \\ \leq\ & \nu \cdot c - \frac{\lambda}{16} \cdot c \\ \leq\ & 0. \end{aligned}$$

(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 $T(d, k)$ is λ-separated.) □

Lemma 3.17 *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 3.10 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

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

where the equality is due to line 15 and the 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 3.15, in this case, OFF incurs a cost of $c = 2 \cdot k^{d-i-1} \cdot \pi.diam$, where $i = \pi.depth$. Thus, the change in Φ due to line 12 is at most

$$\begin{aligned} & \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.y) \\ \leq \; & \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.react) \\ = \; & \nu \cdot c - \pi.diam \cdot \sum_{j=0}^{k-2} (g(i+1, k-j) - g(i+1, k)) \\ = \; & \nu \cdot c - \pi.diam \cdot k^{d-i-1} \sum_{j=0}^{k-2} \left(\frac{1}{4(k-j)} - \frac{1}{4k} \right) \\ \leq \; & \nu \cdot c - \left(\frac{\ln k}{8} - \frac{1}{8} \right) \cdot c \\ \leq \; & 0. \end{aligned}$$

(In the above derivation, $\delta.x$ and $\delta.y$ denotes the values just before the execution of line 12, the first inequality follows from Lemma 3.10, the first equality follows from Lemma 3.9, and the second inequality follows from the fact that $H_{k-1} > \ln k$, where H_{k-1} denotes the $(k-1)$th harmonic number, that is, $H_{k-1} = \sum_{i=1}^{k-1} \frac{1}{i}$.) 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 $\Phi \leq 0$. $\square$

Lemma 3.18 *Lines 18 to 19 preserve* $\Phi \leq 0$.

Proof: Let the request appended to σ in line 18 be (u, f). The guard of the down loop ensures that f is in $\pi.missing$. Algorithm OFF incurs cost at most $\pi.diam$ to process such a request because it stores all the files in $F(\pi.depth)$ in a child of π, and $\pi.missing \subseteq F(\pi.depth)$.

Since ON is nice, it processes a request (u, f) in two phases as follows: in the first phase, ON adds a copy (u, f) to its placement; in the second phase, ON performs an arbitrary sequence of add and delete operations. If π is equal to $root$, then ON incurs expected cost at least $\frac{\lambda}{2} \cdot \pi.diam$ in the first phase since the miss penalty associated with any file is $\lambda \cdot root.diam$. If π is not equal to $root$, then ON incurs expected cost at least $\frac{1}{2} \cdot \pi.parent.diam = \frac{\lambda}{2} \cdot \pi.diam$ in the first phase.

Thus, in either case, ON incurs expected cost at least $\frac{\lambda}{2} \cdot \pi.diam$ it the first phase. Let X be the set of nodes on the path from π to u, excluding π. Note that $\alpha.load$, for $\alpha \in X$, may increase by 1 during the first phase.

The change in Φ due to the first phase of ON and due to OFF in processing a request is at most

$$\begin{aligned}
& \nu \cdot \pi.diam - \frac{\nu' \cdot \lambda \cdot \pi.diam}{2} + \sum_{\alpha \in X} \alpha.parent.diam \\
\leq \;& \pi.diam \cdot \left(\nu - \frac{\nu' \cdot \lambda}{2}\right) + \pi.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\
\leq \;& \pi.diam \cdot \left(\nu - \frac{\nu' \cdot \lambda}{2} + \frac{\lambda}{\lambda - 1}\right) \\
= \;& \pi.diam \cdot \left(\nu - \frac{\lambda^2 - 2\lambda}{2(\lambda - 1)}\right) \\
\leq \;& \pi.diam \cdot \frac{\lambda}{16} \cdot \left(\frac{15 - 7\lambda}{\lambda - 1}\right) \\
\leq \;& 0,
\end{aligned}$$

(In the above derivation, the second last inequality follows from $\nu \leq \frac{\lambda}{16}$ and the last inequality follows from $\lambda \geq \frac{15}{7}$.)

For analyzing the second phase of ON in processing a request, it is convenient to view the randomized online algorithm ON as a probability distribution over a collection of deterministic online algorithms. For each such deterministic algorithm A, we define an associated potential function Φ_A as in Equation 1, but with $T_{\mathrm{ON}}(\sigma)$ replaced by the cost incurred by A on σ, denoted $T_A(\sigma)$, and each term $\alpha.load$ appearing in the second summation replaced by $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed_A|$, where $\alpha.placed_A$ denotes the set of distinct files placed by A in $\alpha.caches$ after processing the request sequence σ. We denote $|(\cup_{0 \leq i < \alpha.depth} F(i)) \cap \alpha.placed_A|$ by $\alpha.load_A$. Note that $T_{\mathrm{ON}}(\sigma)$ is the expected value of $T_A(\sigma)$ when A is chosen at random from the probability distribution associated with ON. Similarly, for any node α, $\alpha.load$ is the expected value of $\alpha.load_A$ and Φ is the expected value of Φ_A. Thus it is sufficient to prove that for any A, each individual operation (i.e., each addition or deletion of a file) performed by A during the second phase does not increase Φ_A. For deletions, this claim is immediate since all terms in Φ_A are unchanged except that terms of the form $\alpha.load_A$ may decrease by one. When a file is added, the set of nodes with an increased $load_A$ value form a path P from some node, say α, to a leaf, and A incurs a cost of $\alpha.parent.diam$. Let the set of nodes on path P be Y. (Note that $root$ does not belong to Y since $root.load$ is always zero.) Since the diameters of the nodes of P are λ-separated, the change in Φ_A is at most

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

The claim of the lemma then follows. $\square$

Lemma 3.19 ON *is* $\Omega\left(\frac{\nu}{\nu'}\right)$*-competitive.*

Proof: Initially, $\Phi = 0$. By Lemmas 3.16, 3.17, and 3.18, $\Phi \leq 0$ is a loop invariant of the main loop. Therefore, by Lemmas 3.5 and 3.8, $T_{\mathrm{ON}}(\sigma) \geq \frac{\nu}{\nu'} \cdot T'_{\mathrm{OFF}}(\sigma)$ 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 ON serves every request with a cost at least 1 (because the diameter of an internal node is at least 1). Hence, $T_{\mathrm{ON}}(\sigma)$ tends to ∞ as N (the length of the request sequence σ) tends to ∞. Therefore, we can ensure that $\frac{T_{\mathrm{ON}}(\sigma)}{T'_{\mathrm{OFF}}(\sigma)+C} = \Omega\left(\frac{\nu}{\nu'}\right)$ by choosing N sufficiently large. $\square$

Theorem 1 ON *is* $\Omega\left(\log \frac{d}{b}\right)$*-competitive.*

Proof: Recall that λ is $\Omega(\log k)$. Hence, $\nu = \Theta(\log k)$ and $\nu' = \Theta(1)$. Lemma 3.19 then implies that ON is $\Omega(\log k)$-competitive. The theorem follows since $d = 8bk - 1$, that is, $k = \Theta(d/b)$. $\square$

It is also possible to express the preceding lower bound in terms of the number of caches n and the capacity blowup b. Since, $n = k^d$ and $d = 8bk - 1$, we have $n = k^{8bk-1}$. Solving these equations for k (e.g., using bootstrapping), we find that $k = \Theta(\frac{\log n}{b(\log\log n - \log b)})$ and hence,

$$\begin{aligned} \log k &= \Theta(\log\log n - \log b - \log(\log\log n - \log b)) \\ &= \Theta(\log\log n - \log b). \end{aligned}$$

It follows that ON is $\Omega(\log\log n - \log b)$-competitive.

4 An Upper Bound

We show in this section that, given $O(d)$ capacity blowup, where d is the depth of the cache hierarchy, a simple LRU-like algorithm, which we refer to as *Hierarchical LRU* (HLRU), is constant-competitive with respect to an optimal offline algorithm OPT. For the sake of simplicity, we assume that every file has unit size and uniform miss penalty. Our result can easily be extended to handle variable file sizes and nonuniform miss penalties using an approach similar to LANDLORD [16].

4.1 The HLRU Algorithm

In this section we present a $2(d+1)$-quasifeasible HLRU algorithm that is constant-competitive with respect to OPT. HLRU divides every cache into $d+1$ equal-sized segments numbered from 0 to d. (For generalizing our results to variable sized files, the segments should be contiguous. For the case of unit sized files considered here, the segments need not be contiguous.) For a node α, we define $\alpha.small$ to be the union of segment $\alpha.depth$ of all the caches in $\alpha.caches$, 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) to internal nodes as well. A copy is a pair (α, f) where α is a node and f is a 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 P. Note that when a copy (α, f) is added to P in line 4, file f is added to $\alpha.small$. In HLRU, a node α uses a variable $\alpha.ts[f]$ to keep track of the timestamp of a file f. For the convenience of presentation, we define $root.parent$ to be a fake node that has every file in $root.parent.small$ (and hence also in $root.parent.big$), and we define $root.parent.diam$ to be the uniform miss penalty.

```
    {On a request (α, f)}
 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.

4.2 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 the execution, which is defined to be time 1. Subsequent epochs begin just after the execution of line 11.

We define $\alpha.ts^*[f]$ to be the time of the most recent access to file f in a cache in $\alpha.caches$ 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 the convenience of analysis, we categorize the file movements in HLRU into two types: *retrievals* and *evictions*. On a request (u, f), the HLRU algorithm first performs a retrieval (this corresponds to the block of code from the beginning of the code to line 5 of the first iteration of the loop) of f from the nearest cache v that has a copy. Let α be the least common ancestor of u and v. Then the cost of such a retrieval is $\alpha.diam$. Let X denote the set of nodes on the path from α to u, excluding α but including u. For every node β in X, we charge a *pseudocost* of $\beta.parent.diam$ to node α for such a retrieval.

Each subsequent iteration of the loop performs an eviction (this corresponds to the block of code 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. We charge a pseudocost of $\alpha.parent.diam$ to α for such an eviction.

The only cost incurred by OPT is due to retrievals. Let OPT adds (or retrieves) a copy (u, f) by fetching f from v, α be the least common ancestor of u and v, and X be the set of nodes on the path from from α to u, excluding α but including u. Then the cost of such a retrieval is $\alpha.diam$. For every node β in X, we charge a pseudocost of $\beta.parent.diam$ to node α for such a retrieval by OPT.

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 a 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 4.1 *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, then according to the code in Figure 2, the only event that sets either side to true is a retrieval of f at a cache u in $\alpha.caches$, which in fact sets both sides to true. 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 (note that 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 (i.e., line 11) 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 $\beta.parent.ts[f] > 0$ (line 5). Thus, falsification of the right side implies falsification of the left side. $\square$

Lemma 4.2 *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) after line 11. By the guard of the inner **if** statement, $f \notin \alpha.big$ just before line 11. Hence, by Lemma 4.1, $\beta.ts[f]$ is 0 for all $\beta \in \alpha.desc$.

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

Between the changes of $\alpha.ts^*[f]$, only the eviction of f from $\alpha.big$ 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. $\square$

Lemma 4.3 *Before and after every retrieval or eviction, for any node α and file f, $\alpha.ts[f] \leq \alpha.ts^*[f]$. 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 4.2. For the second claim, note that we are evicting the last copy of f in $\alpha.big$ from $\alpha.small$. By Lemma 4.1, all proper descendants β of α have $\beta.ts[f] = 0$. So $\alpha.ts[f] = \alpha.ts^*[f]$ by Lemma 4.2. □

Lemma 4.4 *If a file movement (between two caches) has actual cost C and charges a total pseudocost of C', then*

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

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 but including u. Then

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

□

Lemma 4.5 *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 4.6 *For any node α, the total pseudocost charged to node α due to evictions 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 4.7 *For any node α and file f,*

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

Proof: We observe that if a pseudocost is charged to a node α as a result of a retrieval, then $f \notin \alpha.big$ before the retrieval and $f \in \alpha.big$ after the retrieval. 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

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

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

Lemma 4.8 *For any node α, $\alpha.big$ always contains the most recently accessed $2 \cdot \alpha.cap$ files by $\alpha.caches$.*

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 in $\alpha.caches$. 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 $\alpha.ts[f] < \alpha.ts[g] \leq \alpha.ts^*[g]$. By Lemma 4.3, $\alpha.ts[f] = \alpha.ts^*[f]$ just after line 8. It follows then from the definition of ts^* that $f \notin X$. □

Lemma 4.9 *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 in $\alpha.caches$ results in a miss if no copy of f exists at any cache in $\alpha.caches$ at the time of the request. For HLRU, a miss occurs if no copy of f is in $\alpha.big$. By Lemma 4.8, 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 in $\alpha.caches$. (Note that LRU misses whenever HLRU misses.) By the well-known result of Sleator and Tarjan [13], 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 4.10 *For any node α, the total pseudocost charged to α by HLRU is at most four times the total pseudocost charged to α by OPT.*

Proof: Follows immediately from Lemmas 4.5, 4.6, 4.7, and 4.9. □

Theorem 2 *HLRU is constant-competitive.*

Proof: Follows immediately from Lemmas 4.4 and 4.10. □

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. *Journal of Algorithms*, 28:67–104, July 1998.

[3] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. 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] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 161–168, May 1998.

[6] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[7] 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.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[9] 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.

[10] 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.

[11] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

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

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

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

[15] D. Wessels and K. Claffy. RFC 2187: Application of Internet Cache Protocol, 1997.

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

[17] L. Zhang, S. Floyd, and V. Jacobson. Adaptive Web Caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, 1997.