On-Line File Caching[*]

Neal E. Young[†]

Abstract

Consider the following file caching problem: in response to a sequence of requests for files, where each file has a specified *size* and *retrieval cost*, maintain a cache of files of total size at most some specified k so as to minimize the total retrieval cost. Specifically, when a requested file is not in the cache, bring it into the cache and pay the retrieval cost, and remove other files from the cache so that the total size of files remaining in the cache is at most k. This problem generalizes previous paging and caching problems by allowing objects of arbitrary size *and* cost, both important attributes when caching files for world-wide-web browsers, servers, and proxies.

We give a simple deterministic on-line algorithm that generalizes many well-known paging and weighted-caching strategies, including least-recently-used, first-in-first-out, flush-when-full, and the balance algorithm. On any request sequence, the total cost incurred by the algorithm is at most $k/(k-h+1)$ times the minimum possible using a cache of size $h \leq k$.

For *any* algorithm satisfying the latter bound, we show it is also the case that for *most* choices of k, the retrieval cost is either insignificant or at most a *constant* (independent of k) times the optimum. This helps explain why competitive ratios of many on-line paging algorithms have been typically observed to be constant in practice.

Key Words. Paging, browser, proxy, caching, competitive analysis.

1 Background and Statement of Results

The *file caching* problem is as follows. Given a cache with a specified size k (a positive integer) and a sequence of requests to files, where each file has a specified *size* (a positive integer) and a specified *retrieval cost* (a non-negative number), maintain files in the cache to satisfy the requests while minimizing the total retrieval cost. Specifically, when a requested file is not in the cache, bring it into the cache, paying the retrieval cost of the file, and remove other files from the cache so that the total size of files remaining in the cache is at most k.

Following Sleator and Tarjan [17], we say a file caching algorithm is $c(h, k)$-*competitive* if on any sequence the total retrieval cost incurred by the algorithm using a cache of size k is at most $c(h, k)$ times the minimum possible cost using a cache of size h. An algorithm is *on-line* if its response to a request does not depend on later requests in the sequence.

Uniform sizes, uniform costs.   With the restriction that all file sizes and costs are the same, the problem is called *paging*. Paging has been extensively studied. In a seminal paper, Sleator and Tarjan [17] showed that least-recently-used and a number of other deterministic on-line paging strategies are $\frac{k}{k-h+1}$-competitive. Sleator and Tarjan also showed that this performance guarantee is the best possible for any deterministic on-line algorithm.

A simple randomized paging algorithm called the marking algorithm was shown to be $2 \ln k$-competitive by Fiat et al. [6]. An optimal $\ln k$-competitive randomized paging algorithm was given by McGeoch and Sleator [16]. In [20], deterministic paging strategies were shown to be *loosely $O(\ln k)$-competitive*. This means roughly that for any sequence, for *most* values of k, the fault rate of the algorithm using a cache of size k is either insignificant or the algorithm is $O(\ln k)$-competitive versus the optimum algorithm using a cache of size k. Similarly, the marking algorithm was shown to be loosely $(2 \ln \ln k + O(1))$-competitive.

Uniform sizes, arbitrary costs.   The special case of file caching when all file sizes are the same is called *weighted caching*. For weighted caching, Chrobak, Karloff, Payne and Vishwanathan [4] showed that an algorithm called the "balance" algorithm is k-competitive. Subsequently in [20] a generalization of that algorithm called the "greedy-dual" algorithm was shown to be $\frac{k}{k-h+1}$-competitive. The greedy-dual algorithm generalizes many well-known paging and weighted-caching strategies, including least-recently-used, first-in-first-out, flush-when-full, and the balance algorithm.

Arbitrary sizes, cost = 1 or cost = size.   Motivated by the importance of file *size* in caching for world-wide-web applications (see comment below), Irani considered two special cases of file caching: when the costs are either all equal (the goal is to minimize the *number* of retrievals), and when each cost equals the file size (the goal is to minimize the total number of *bytes* retrieved). For these two cases, Irani [9] gave $O(\log^2 k)$-competitive randomized on-line algorithms.

Comment: the importance of sizes and costs.   File caching is important for world-wide-web applications. For instance, in browsers and proxy servers remote files are cached locally to avoid remote retrieval. In web servers, disk files are cached in fast memory to speed response time. As Irani points out (see [9] and references therein), file *size* is an important consideration; caching policies adapted from memory management applications that don't take size into account do not work well in practice.

Algorithm LANDLORD

Maintain a real value credit[f] with each file f in the cache.

When a file g is requested:

1. **if** g is not in the cache **then**
2.   **until** there is room for g in the cache:
3.   For each file f in the cache, decrease credit[f] by $\Delta \cdot \text{size}[f]$,
4.   where $\Delta = \min_{f \in \text{cache}} \text{credit}[f]/\text{size}[f]$.
5.   Evict from the cache any subset of the files f such that credit[f] = 0.
6.   Bring g into the cache and set credit[g] $\leftarrow$ cost(g).
7. **else** Reset credit[g] to any value between its current value and cost(g).

Figure 1: The on-line file caching algorithm LANDLORD. Credit is given to each file when it is requested. "Rent" is charged to each file in the cache in proportion to its size. Files are evicted as they run out of credit. Step 7 is not necessary for the worst-case analysis, but it is likely to be important in practice: raising the credit as much as possible in step 7 generalizes the least-recently-used paging strategy; not raising at all generalizes the first-in-first-out paging strategy.

Allowing arbitrary *costs* is likely to be important as well. In many cases, the cost (e.g., latency, total transmission time, or network resources used) will neither be uniform across files nor proportional solely to the size. For instance, the cost to retrieve a remote file can depend on the *distance* the file must travel in the network. Even accounting for distance, the cost need not be proportional to the size, e.g., because of economies of scale in routing files through the network. Further, in some applications it makes sense to assign different *kinds* of costs to different kinds of files. For instance, some kinds of documents are displayed by web browsers as they are received, so that the effective delay for the user is determined more by the latency than the total transmission time. Other documents must be fully transmitted before becoming useful. Both kinds of files can be present in a cache. In all these cases, assigning uniform costs or assigning every file's cost to be its size is not ideal.[1]

This paper: arbitrary sizes, arbitrary costs. This paper presents a simple deterministic on-line algorithm called LANDLORD (shown in Figure 1). LANDLORD handles the problem of file caching with arbitrary costs and integer sizes. The first result is:

---

[1] In many applications the actual cost to access a file may vary with time; that issue is not considered here, nor is the issue of cache consistency (i.e., if the remote file changes at the source, how does the local cache get updated? The simplest adaptation of the model here would be to assume that a changed file is treated as a new file; this would require that the local cache strategy learn about the change in some way). Finally, the focus here is on simple *local* caching strategies, rather than distributed strategies in which servers cooperate to cache pages across a network (see e.g. [11]).

Theorem 1 LANDLORD *is* $\frac{k}{k-h+1}$*-competitive for file caching.*

This performance guarantee is the best possible for any deterministic on-line algorithm.[2] File caching is not a special case of the k-server problem, although weighted caching is a special case of both file caching and the k-server problem.

LANDLORD is a generalization of the greedy-dual algorithm [20] for weighted caching, which in turn generalizes least-recently-used and first-in-first-out (paging strategies), as well as the balance algorithm for weighted caching. The analysis uses the potential function $\Phi = (h-1)\sum_{f\in\mathrm{LL}}\mathrm{credit}[f]+k\sum_{f\in\mathrm{OPT}}\mathrm{cost}(f)-\mathrm{credit}[f]$. The analysis is simpler than that of [20] for the special case of weighted caching.

In an independent work [3], Cao and Irani showed that LANDLORD (with step 7 raising credit[g] as much as possible) is k-competitive. They also gave empirical evidence that the algorithm performs well in practice.

This paper: (ϵ, δ)-loosely c-competitiveness. In practice it has been observed that on "typical" request sequences, paging algorithms such as least-recently-used, using a cache of size k, incur a cost within a small constant factor (independent of k) times the minimum possible using a cache of size k [20]. This is in contrast to the theoretically optimal competitive ratio of k. A number of refinements of competitive analysis have been proposed to try to understand the relevant factors. Borodin, Irani, Raghavan, and Schieber [2], in order to model locality of reference, proposed the *access-graph* model which restricts the request sequences to paths in a given graph (related papers include [5, 10, 7]). Karlin, Phillips, and Raghavan [12] proposed a variant in which the graph is a Markov chain (i.e. the edges of the graph are assigned probabilities, and the request sequence corresponds to a random walk) (see also [14]). Koutsoupias and Papadimitriou [13] proposed the *comparative ratio* (for comparing classes of on-line algorithms) and the *diffuse adversary model* (in which the adversary chooses a probability distribution, rather than a sequence, from some restricted class of distributions).

In this paper we introduce a refinement of the aforementioned *loosely competitive* ratio [20] (another previously proposed alternative model). The model is motivated by two observations. First, in practice, if the retrieval cost is low enough in an *absolute* sense, the competitive ratio is of no concern. For instance, in paging, if the fault rate drops much below

$$\frac{\text{time to execute a machine instruction}}{\text{time to retrieve a page from disk}},$$

then the total time to handle page faults is less than the time to execute instructions, so that page faults cease to be the limiting factor in the execution time. Similar considerations hold in other settings such as file caching. To formalize

---

[2]Manasse, McGeoch, and Sleator [15] show that no deterministic on-line algorithm for the well-known k-server problem on any metric space of more than k points is better than $\frac{k}{k-h+1}$-competitive. This implies that, at least for any special case when all sizes are 1 (i.e. weighted caching), no deterministic on-line algorithm for file caching is better than $\frac{k}{k-h+1}$-competitive.

this, we introduce a parameter $\epsilon > 0$, and say that "low enough" for a request sequence r means "no more than ϵ times the sum of the retrieval costs" (the sum being taken over all requests). This is tantamount to assuming that handling a file of cost $\text{cost}(f)$ requires overhead of $\epsilon \, \text{cost}(f)$ whether it is retrieved or not.

Second, in many circumstances, we do not expect the input sequences to be adversarially tailored for our particular cache size k. To model this, rather than somehow restricting the input sequences, we allow all input sequences but for each, we consider what happens at a *typical* cache size k. Formally, for each sequence, we consider all the values of k in any range $\{1, 2, \ldots, n\}$, and we ask that the competitive ratio be at most some constant c for at least $(1 - \delta)n$ of these values, where δ is a parameter to the model.

Our model, which we dub "loose competitiveness", combines both these ideas:

Definition 1 *A file caching algorithm A is (ϵ, δ, n)-loosely c-competitive if, for any request sequence r, at least $(1 - \delta)n$ of the values $k \in \{1, 2, \ldots, n\}$ satisfy*

$$\text{cost}(A, k, r) \leq \max\left\{ c \cdot \text{cost}(\text{OPT}, k, r), \epsilon \cdot \sum_{f \in r} \text{cost}(f) \right\}. \tag{1}$$

A is (ϵ, δ)-loosely c-competitive if A is (ϵ, δ, n)-loosely c-competitive for all positive integers n.

Here $\text{cost}(A, k, r)$ denotes the cost incurred by algorithm A using a cache of size k on sequence r. OPT denotes the optimal algorithm, so that $\text{cost}(\text{OPT}, k, r)$ is the minimum possible cost to handle the sequence r using a cache of size k. The sum on the right ranges over all requests in r, so that if a file is requested more than once, its cost is counted for each request.

Since the standard competitive ratio grows with k, it is not a-priori clear that any on-line algorithm can be (ϵ, δ)-loosely c-competitive for any c that depends only on ϵ and δ. Our second result is the following.

Theorem 2 *Every $\frac{k}{k-h+1}$-competitive algorithm is (ϵ, δ)-loosely c-competitive for any $0 < \epsilon, \delta < 1$ and $c = (e/\delta)\ln(e/\epsilon) = O((1/\delta)\log(1/\epsilon))$.*

(Throughout the paper e is the base of the natural logarithm.) The interpretation is that for *most* choices of k, the retrieval cost is either insignificant or the competitive ratio is constant.

This result supports the intuition that it is meaningful to compare an algorithm against a "handicapped" optimal algorithm (most competitive analyses consider the case $h = k$). A strong performance guarantee, even against a handicapped optimal algorithm, may be as (or more) meaningful than a weak performance guarantee against a non-handicapped adversary.

Our proof is similar in spirit to the proof in [20] for the special case of paging, but the proof here is simpler, more general, and gives a stronger result.

Of course the following corollary is immediate:

Corollary 1 LANDLORD *is (ϵ, δ)-loosely c-competitive for $c = (e/\delta)\ln(e/\epsilon) = O((1/\delta)\log(1/\epsilon))$.*

This helps explain why the competitive ratios of the many on-line algorithms that LANDLORD generalizes are typically observed to be constant.

For completeness, we also consider randomized algorithms:

Theorem 3 *Let $0 \leq \epsilon, \delta \leq 1$. Any $\alpha + \beta \ln \frac{k}{k-h+1}$-competitive algorithm is (ϵ, δ)-loosely c-competitive for $c = e\alpha + e\beta \ln[(1/\delta) \ln(e/\epsilon)] = O(\log[(1/\delta) \log(1/\epsilon)])$.*

It is known (e.g. [19, 18]) that the marking algorithm (a randomized on-line algorithm) is $(1 + 2 \ln \frac{k}{k-h})$-competitive for paging and $(1 + 2 \ln k)$-competitive for $h = k$. It follows by algebra that the marking algorithm is $1 + 2 \ln 2 + 2 \ln \frac{k}{k-h+1}$-competitive. Although a stronger result can probably be shown, this simple one and Theorem 3 imply the following corollary:

Corollary 2 *The marking algorithm is (ϵ, δ)-loosely c-competitive for paging for $c = e + 2e \ln 2 + 2e \ln[(1/\delta) \ln(e/\epsilon)] = O(\log[(1/\delta) \log(1/\epsilon)])$.*

Finally, we show Theorem 2 and Corollary 1 are tight up to a constant factor:

Theorem 4 *For any ϵ and δ with $0 < \epsilon < 1$ and $0 < \delta < 1/2$, LANDLORD is not (ϵ, δ)-loosely c-competitive for $c = (1/8\delta) \log_2(1/2\epsilon) = \Theta((1/\delta) \log(1/\epsilon))$.*

2 Analysis of LANDLORD.

Theorem 1 LANDLORD *is $\frac{k}{k-h+1}$-competitive for file caching.*

Proof: Define potential function

$$\Phi = (h-1) \cdot \sum_{f \in \mathrm{LL}} \mathrm{credit}[f] + k \cdot \sum_{f \in \mathrm{OPT}} \mathrm{cost}(f) - \mathrm{credit}[f].$$

Here LL denotes the cache of LANDLORD; OPT denotes the cache of OPT. For $f \notin$ LL, by convention $\mathrm{credit}[f] = 0$. Before the first request of a sequence, when both caches are empty, Φ is zero. After all requests have been processed (and in fact at all times), $\Phi \geq 0$. Below we show that at each request:

- if OPT retrieves a file of cost c, Φ increases by at most kc;

- if LANDLORD retrieves a file of cost c, Φ decreases by at least $(k - h + 1)c$;

- at all other times Φ does not increase.

These facts imply that the cost incurred by LANDLORD is bounded by $k/(k - h + 1)$ times the cost incurred by OPT.

The actions affecting Φ following each request can be broken down into a sequence of steps, with each step being one of the following. We analyze the effect of each step on Φ.

- OPT **evicts a file** f.

  Since $\mathrm{credit}[f] \leq \mathrm{cost}(f)$, Φ cannot increase.

- OPT **retrieves a file** g**.**

  In this step OPT pays the retrieval cost $\text{cost}(g)$.

  Since $\text{credit}[g] \geq 0$, Φ can increase by at most $k \cdot \text{cost}(g)$.

- LANDLORD **decreases** $\text{credit}[f]$ **for all** $f \in$ LL**.**

  Since the decrease of a given $\text{credit}[f]$ is $\Delta \text{size}(f)$, the net decrease in Φ is Δ times
  $$(h - 1) \text{size}(\text{LL}) - k \text{size}(\text{OPT} \cap \text{LL}),$$
  where $\text{size}(X)$ denotes $\sum_{f \in X} \text{size}(f)$.

  When this step occurs, we can assume that the requested file g has already been retrieved by OPT but is not in LL. Thus, $\text{size}(\text{OPT} \cap \text{LL}) \leq h - \text{size}(g)$.

  Further, there is not room for g in LL, so that $\text{size}(\text{LL}) \geq k - \text{size}(g) + 1$ (recall that sizes are assumed to be integers). Thus the decrease in the potential function is at least Δ times

  $$(h - 1)(k - \text{size}(g) + 1) - k(h - \text{size}(g)).$$

  Since $\text{size}(g) \geq 1$ and $k \geq h$, this is at least $(h-1)(k-1+1) - k(h-1) = 0$.

- LANDLORD **evicts a file** f**.**

  LANDLORD only evicts f when $\text{credit}[f] = 0$. Thus, Φ is unchanged.

- LANDLORD **retrieves the requested file** g **and sets** $\text{credit}[g]$ **to** $\text{cost}(g)$**.**

  In this step LANDLORD pays the retrieval cost $\text{cost}(g)$.

  Since g was not previously in the cache (and $\text{credit}[g]$ was zero), and because we can assume that $g \in$ OPT, Φ decreases by $-(h - 1)\text{cost}(g) + k \text{cost}(g) = (k - h + 1)\text{cost}(g)$.

- LANDLORD **resets** $\text{credit}[g]$ **between its current value and** $\text{cost}(g)$**.**

  Again, we can assume $g \in$ OPT. If $\text{credit}[g]$ changes, it can only increase. In this case, since $(h - 1) < k$, Φ decreases. $\diamond$

3 Upper Bounds on Loose Competitiveness.

The following technical lemma is at the core of Theorems 2 and 3.

Lemma 1 *Let A be any $\tau(k, k - h)$-competitive algorithm for some function τ that is increasing w.r.t. k and decreasing with respect to $k - h$.*

  For any $b, \epsilon, \delta, n > 0$ (n an integer, $b < \delta n$), A is (ϵ, δ, n)-loosely c-competitive for
  $$c = \tau(n, b) \, \epsilon^{-(b+1)/(\delta n - b - 1)}.$$

Proof: Fix any request sequence r and $b, \epsilon, \delta, n > 0$. Define c as above. Say a value $k \in \{1, 2, \ldots, n\}$ is *bad* if

$$\text{cost}(A, k, r) > \max \left\{ c \cdot \text{cost}(\text{OPT}, k, r),\ \epsilon \cdot \textstyle\sum_{f \in r} \text{cost}(f) \right\}. \qquad (2)$$

We will show that at most δn values are bad.

Denote the bad values (in increasing order) $k_0, k_1, \ldots, k_B$. The form of the argument is this: on the one hand, we show that $\text{cost}(A, k_i, r)$ decreases exponentially with i; on the other hand, we know that (for each i) $\text{cost}(A, k_i, r)$ is not too small (e.g. smaller than ϵ times $\text{cost}(A, k_0, r)$); together, these will imply that B cannot be too large.

From the sequence of bad values, select the subsequence $k_0, k_{\lceil b \rceil}, k_{2 \lceil b \rceil}, \ldots$ and denote it $k'_0, k'_1, \ldots, k'_{B'}$. The properties of this sequence that we use are $k'_i - k'_{i-1} \geq b$ for each i and $B' \geq B/(b+1)$.

Since A is $\tau(k, k - h)$-competitive, choosing $k = k'_i$ and $h = k'_{i-1}$ shows that

$$\text{cost}(A, k'_i, r) \ \leq\ \tau(k'_i, k'_i - k'_{i-1}) \, \text{cost}(\text{OPT}, k'_{i-1}, r).$$

From the first term in the maximum in (2), $\text{cost}(A, k'_{i-1}, r) \geq c \cdot \text{cost}(\text{OPT}, k'_{i-1}, r)$. The condition on τ implies $\tau(k'_i, k'_i - k'_{i-1}) \leq \tau(n, b)$. Thus,

$$\text{cost}(A, k'_i, r) \ \leq\ (\tau(n, b)/c) \, \text{cost}(A, k'_{i-1}, r).$$

Inductively,
$$\text{cost}(A, k'_{B'}, r) \ \leq\ (\tau(n, b)/c)^{B'} \text{cost}(A, k'_0, r).$$

That is, for every b bad values, $\text{cost}(A, k_i, r)$ decreases by a factor of $\tau(n, b)/c$. The rest is algebra. As noted before, $\text{cost}(A, k'_{B'}, r) > \epsilon \, \text{cost}(A, k'_0, r)$. Combining with the above inequality gives $(\tau(n, b)/c)^{B'} > \epsilon$, which (by substituting for c and simplifying) gives

$$B' < \delta n/(b+1) - 1.$$

Combining this with $B' \geq B/(b+1)$ gives $B + 1 < \delta n$. That is, there are fewer than δn bad values. $\diamond$

Theorem 2 *Every $\frac{k}{k-h+1}$-competitive algorithm is (ϵ, δ)-loosely c-competitive for any $0 < \epsilon, \delta < 1$ and $c = (e/\delta) \ln(e/\epsilon) = O((1/\delta) \log(1/\epsilon))$.*

Proof: Fix any $\epsilon, \delta, n > 0$ (n integer). We need to show the algorithm is (ϵ, δ, n)-loosely c-competitive. Let $\tau(k, k - h) = k/(k - h + 1)$ and $b = \delta n / \ln(e/\epsilon) - 1$. If $b \leq 0$, then an easy calculation shows $c \geq n$, and since the algorithm is k-competitive, the conclusion holds trivially.

Otherwise ($b > 0$), we apply the technical lemma. With this choice of b, $\epsilon^{-(b+1)/(\delta n - b - 1)} = e$, so $c = e \, \tau(n, b)$. For this τ and b, $\tau(n, b)$ simplifies to $(1/\delta) \ln(e/\epsilon)$. $\diamond$

Theorem 3 *Let $0 \leq \epsilon, \delta \leq 1$. Any $\alpha + \beta \ln \frac{k}{k-h+1}$-competitive algorithm is (ϵ, δ)-loosely c-competitive for $c = e\alpha + e\beta \ln[(1/\delta) \ln(e/\epsilon)] = O(\log[(1/\delta) \log(1/\epsilon)])$.*

Algorithm LANDLORD for the special case of paging
_____

Maintain a value credit$[f] \in [0, 1]$ with each item f in the cache.

When an item g is requested:

1. **if** g is not in the cache **then**
2.   **if** there are no 0-credit items in the cache,
3.   **then** decrease all credits by the minimum credit.
4.   Evict from the cache any subset of the items f such that credit$[f] = 0$.
5.   Bring g into the cache and set credit$[g] \leftarrow 1$.
6. **else** Reset credit$[g]$ to any value between its current value and 1.

Figure 2: LANDLORD as it specializes for paging. To get LRU, reset credit$[g]$ to 1 in line 6 and evict the single least-recently-requested 0-credit item in line 4. To get FWF, leave credit$[g]$ unchanged in line 6 and evict all 0-credit items in line 4. To get FIFO, leave credit$[g]$ unchanged in line 6 and evict the single 0-credit item that has been in the cache the longest in line 4. All of these strategies maintain credits in $\{0, 1\}$.

Proof: Much as in the preceding proof, take $\tau(k, k-h) = \alpha + \beta \ln(k/(k-h+1))$ and $b = \delta n / \ln(e/\epsilon) - 1$. If $b \leq 0$, then an easy calculation shows $c \geq \alpha + \beta \ln n$, so the conclusion holds trivially.

Otherwise $(b > 0)$, we apply the technical lemma. With this choice of b, $\epsilon^{-(b+1)/(\delta n - b - 1)} = e$, so $c = e \, \tau(n, b)$. For this τ and b, $\tau(n, b)$ simplifies to $\alpha + \beta \ln[(1/\delta) \ln(e/\epsilon)]$. $\diamond$

4   Lower Bound on Loose Competitiveness.

In this section we show the following theorem.

Theorem 4 *For any ϵ and δ with $0 < \epsilon < 1$ and $0 < \delta < 1/2$, LANDLORD is not (ϵ, δ)-loosely c-competitive for $c = (1/8\delta) \log_2(1/2\epsilon) = \Theta((1/\delta) \log(1/\epsilon))$.*

For the proof we adapt an unpublished result from [18]. We consider the least-recently-used (LRU) and flush-when-full (FWF) paging strategies. (Recall that paging is the special case of file caching when each size and retrieval cost is 1.) We assume the reader is familiar with FWF and LRU, but just in case here is a brief description of each. When an item not in the cache is requested and the cache is full, FWF empties the cache completely. In contrast, LRU evicts the single item that was least recently requested. Figure 2 describes how each is a special case of LANDLORD.

We give the desired lower bound for FWF. Since LANDLORD generalizes FWF, the result follows. This appears unsatisfactory, because it would be natural to restrict LANDLORD (in line 5) to evict only one file at a time (unlike FWF).

However, the same lower bound proof applies even to a version of LANDLORD that has this behavior. (We discuss this more after the proof.) Interestingly, the lower bound does *not* apply to LRU. In fact, for the sequences constructed for the lower bound, LRU is a near-optimal algorithm.

The proof uses the concept of *k-phases* from the standard competitive analysis framework. We define k-phases as follows. Let $s = s_1 s_2 \ldots s_n$ be any sequence of requests. Consider running FWF with a cache of size k on the sequence, and break the sequence into pieces (called *phases* or *k-phases*) so that each piece starts with a request that causes FWF to flush its cache. Thus, each phase (except the last) contains requests to k distinct items, and each phase (except the first) starts with a request to an item not requested in the previous phase.

The adversarial sequence. Fix any $\epsilon, \delta \geq 0$ with $\epsilon < 1$ and $\delta \leq 1/2$. Define (with foresight) c as in the theorem and let n be some sufficiently large integer. We will show that LANDLORD is not (ϵ, δ, n)-loosely competitive. Define $k_0 = \lceil (1 - \delta) n \rceil$. We will focus on k in the range $k_0, \ldots, n$, inductively constructing a sequence s such that each cache size in this range is bad for FWF in the sense of Condition (2). That is, for each such k, we will show $\mathrm{cost}(\mathrm{FWF}, k, s) > \max\{c\,\mathrm{cost}(\mathrm{OPT}, k, s), \epsilon|s|\}$. The number of k's in the range is $1 + n - k_0 > \delta n$, so this will show the desired result.

In the construction we will build sequences that contain a special request "**x**". Each occurrence of **x** represents a request to an item that is not requested anywhere else (so all occurrences refer to different items).

For the base case of the induction, we let s_0 be a sequence containing k_0 special requests **x**. For the inductive step we do the following. For $i = 0, 1, 2, \ldots$ let $k_{i+1} = \lceil k_0 (1 + 1/(4c))^i \rceil$ and let s_{i+1} be obtained from s_i by choosing any $k_{i+1} - k_i$ special requests **x** (including the first one) in s_i, replacing each unchosen **x** with a regular request not occurring elsewhere in s_i, and then appending two copies of the modified string.

For example, if $k_0 = 4$ and $k_1 = 5$, then $s_0 = \mathbf{xxxx}$ and $s_1 = \mathbf{x123x123}$.

We let the final sequence s be any s_i such that $k_i > n$. This describes the construction. The basic useful properties of s are the following:

Lemma 2 *(1) Each s_i has length $k_0 2^i$ and references k_i distinct items.*

(2) Any item r introduced in the ith inductive step (building s_{i+1}) has periodicity $k_0 2^i$ in s. That is, for some j with $1 \leq j \leq k_0 2^i$, the positions in s at which r is requested are $j, j + k_0 2^i, j + 2 \cdot k_0 2^i, j + 3 \cdot k_0 2^i, \ldots$.

(3) For each i, each length-$k_0 2^i$ contiguous subsequence of s references k_i distinct items.

Proof: Properties (1) and (2) above are easy to verify by induction. Property (3) follows from properties (1) and (2). In particular, in each length-$k_0 2^i$ contiguous subsequence of s, each item of periodicity $k_0 2^j$ (for $j \leq i$) is requested 2^{j-i} times, and each other request is to an item of periodicity larger than $k_0 2^i$ that is requested only once in the subsequence. Since each length-$k_0 2^i$ contiguous subsequence has this structure, each such subsequence references the same number of distinct items as the string s_i — that is, k_i distinct items.   ◇

Using these properties, we show the following:

Lemma 3 *Suppose n is larger than $4c/(1-\delta)$. Using any cache size k such that $k_0 \le k \le n$, the fault rate of* FWF *on s is more than c times that of* LRU.

Proof: In the construction of s_{i+1} from s_i, we were careful to leave the *first* special request $\mathbf{x}$ in s_i alone. This ensures that each k_i-phase of s is of length $k_0 2^i$ and starts with a symbol of periodicity greater than $k_0 2^i$.

From these properties it is easy to calculate the fault rates of FWF using a cache of size k_i on s. The fault rate of FWF is $k_i/(k_0 2^i)$ — each k_i-phase has length $k_0 2^i$ and causes k_i faults.

The fault rate of LRU can be calculated using the following observation. LRU with a cache of size k_i faults on exactly those items of periodicity greater than $k_0 2^i$. This is because LRU evicts an item r exactly when there have been k_i other distinct items requested since the last request to r, and we know (property (3)) that between two requests of any item r with periodicity $k_0 2^j$ there are $k_j - 1$ distinct items (other than r) requested.

We can count the frequency of requests to items with periodicity greater than $k_0 2^i$ as follows. Consider any contiguous subsequence of length $k_0 2^{i+1}$. Let a and b be the first and second half of the subsequence, respectively (each of a and b has length $k_0 2^i$). We know that there are k_i distinct items requested in a, and k_{i+1} distinct items requested in ab. But the items requested in b that are not requested in a are exactly the items of periodicity greater than $k_0 2^i$. Thus, there are $k_{i+1} - k_i$ such items in b. As each is requested exactly once in b, the frequency of such requests (and the fault rate of LRU with a cache of size k_i) is $(k_{i+1} - k_i)/(k_0 2^i)$.

Thus, for any i, using a cache of size k_i, the ratio of the fault rate of FWF to that of LRU is

$$k_i/(k_{i+1} - k_i).$$

An easy calculation (using the assumption $n > 4c/(1-\delta)$) shows this is at least $2c$.

What about any k such that $k_i \le k \le k_{i+1}$ for some i? We know that FWF faults k times in each k-phase. The number of k-phases is at least the number of k_{i+1}-phases, i.e. at least $|s|/(k_0 2^{i+1})$. Thus, the fault rate is at least $k_i/(k_0 2^{i+1})$ — half the fault rate of FWF with a cache of size k_i. For LRU, the fault rate with a cache of size k is at most the fault rate with a cache of size k_i. Together these facts imply that (for any k such that $k_i \le k \le k_{i+1}$ for some i), using a cache of size k, the ratio of the fault rate of FWF to that of LRU is at least half the ratio when using a cache of size k_i. Thus, the ratio is greater than c. ⋄

To finish the proof of Theorem 4, we need to show that the fault rate of FWF remains above ϵ for all k such that $k_0 \le k \le n$. Reasoning as in the previous proof, the fault rate of FWF with such a cache size k is at least $k_i/(k_0 2^{i+1})$ for some i where $k_i \le n$. So we need to show $k_i/(k_0 2^{i+1}) \ge \epsilon$ if $k_i \le n$. In fact, we show the stronger result that $1/2^{i+1} \ge \epsilon$.

The rest is algebra. In the following we will use the inequalities $1 + x \ge 2^x$ for $x \le 1$ and $1 - x \ge 2^{-2x}$ for $x \le 1/2$.

That $k_i \leq n$ implies that $i \leq 8\delta c$ by the following argument. (Each line follows from the line before it by the reason given.)

$$
\begin{array}{rcll}
k_i & \leq & n & \text{given} \\
(1-\delta)n(1+1/4c)^i & \leq & n & \text{definition of } k_i, \text{ and } x \leq \lceil x \rceil \\
2^{-2\delta}2^{i/4c} & \leq & 1 & \text{inequalities mentioned above} \\
i & \leq & 8\delta c & \text{algebra}
\end{array}
$$

Using this we will show $1/2^{i+1} \geq \epsilon$, which implies $k_i/(k_0 2^{i+1}) \geq \epsilon$.

$$
\begin{array}{rcll}
8\delta c & \leq & \log_2(1/2\epsilon) & \text{definition of } c \\
i & \leq & \log_2(1/2\epsilon) & i \leq 8\delta c \text{ (proven above)} \\
1/2^{i+1} & \geq & \epsilon & \text{algebra}
\end{array}
$$

This concludes the proof of Theorem 4.   ◇

We can modify FWF so that it doesn't evict all items from the cache at the beginning of the phase, but instead evicts the 0-credit items (those not yet request this phase) one at a time but pessimally — in the order that they will be next requested. The modified algorithm only evicts one page at a time, but, since it still incurs k faults per k-phase, the proof of Theorem 4 applies to the modified algorithm as well. The modified algorithm is also a special case of LANDLORD. Thus, the lower bound applies to LANDLORD even if LANDLORD is constrained to evict only as many items as necessary to handle the current request.

5   Further Directions

A main open question here seems to be to more tightly characterize the loose competitiveness of LRU. A reasonable goal would be to find a non-trivial lower bound or an upper bound better than the one implied in this paper. The latter would show that LRU is better than FWF in this model. It would also be nice to characterize the relative loose competitiveness of LRU and first-in-first-out (FIFO).

Another direction is to find a non-trivial lower bound for the randomized marking algorithm for paging. Finally, the lower bounds in this paper apply to particular on-line algorithms; what lower bounds can be shown for *arbitrary* deterministic on-line algorithms, or for *arbitrary* randomized on-line algorithms?

Acknowledgements

References

[1] *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997.

[2] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, April 1995.

[3] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems*, December 1997.

[4] Marek Chrobak, Howard Karloff, T.H. Payne, and Sundar Vishwanathan. New results on server problems. *SIAM Journal on Discrete Mathematics*, 4(2):172–181, May 1991.

[5] Amos Fiat and Anna R. Karlin. Randomized and multipointer paging with locality of reference. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 626–634, Las Vegas, Nevada, 29 May–1 June 1995.

[6] Amos Fiat, Richard M. Karp, Michael Luby, Lyle A. McGeoch, Daniel D. Sleator, and Neal E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.

[7] Amos Fiat and Ziv Rosen. Experimental studies of access graph based heuristics: Beating the LRU standard? In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 63–72, New Orleans, Louisiana, 5–7 January 1997.

[8] IEEE. *35th Annual Symposium on Foundations of Computer Science*, Santa Fe, New Mexico, 20–22 November 1994.

[9] Sandy Irani. Page replacement with multi-size pages and applications to Web caching. In ACM [1], pages 701–710.

[10] Sandy Irani, Anna R. Karlin, and Steven Phillips. Strongly competitive algorithms for paging with locality of reference. *SIAM Journal on Computing*, 25(3):477–497, June 1996.

[11] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In ACM [1], pages 654–663.

[12] Anna R. Karlin, Steven J. Phillips, and Prabhakar Raghavan. Markov paging (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science*, pages 208–217, Pittsburgh, Pennsylvania, 24–27 October 1992. IEEE.

[13] Elias Koutsoupias and Christos H. Papadimitriou. Beyond competitive analysis. In *35th Annual Symposium on Foundations of Computer Science* [8], pages 394–400.

[14] Carsten Lund, Steven Phillips, and Nick Reingold. IP over connection-oriented networks and distributional paging. In *35th Annual Symposium on Foundations of Computer Science* [8], pages 424–434.

[15] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[16] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.

[17] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, February 1985.

[18] Neal E. Young. Competitive paging and dual-guided algorithms for weighted caching and matching. (Thesis) Tech. Rep. CS-TR-348-91, Computer Science Department, Princeton University, October 1991.

[19] Neal E. Young. On-line caching as cache size varies. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1991.

[20] Neal E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11(6):525–541, June 1994.