ELSEVIER

Theoretical Computer Science 268 (2001) 119–131

Theoretical Computer Science

www.elsevier.com/locate/tcs

# Online paging and file caching with expiration times

Tracy Kimbrel

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA*

Accepted May 2000

**Abstract**

We consider a paging problem in which each page is assigned an expiration time at the time it is brought into the cache. The expiration time indicates the latest time that the fetched copy of the page may be used. Requests that occur later than the expiration time must be satisfied by bringing a new copy of the page into the cache. The problem has applications in caching of documents on the World Wide Web (WWW). We show that a natural extension of the well-studied least recently used (LRU) paging algorithm is strongly competitive for the uniform retrieval cost, uniform size case. We then describe a similar extension of the recently proposed Landlord algorithm for the case of arbitrary retrieval costs and sizes, and prove that it is strongly competitive. The results extend to the loose model of competitiveness as well. 

*Keywords:* Online algorithms; Competitive analysis; Paging; Caching; World Wide Web

## 1. Introduction

In the online *paging* problem, a sequence of requests for pages is presented one at a time to an online algorithm. The algorithm is provided with a *cache* which can hold up to $k$ pages for some integer $k$. The goal is to minimize the number of page faults, that is, the number of times that a request is made to a page that is not present in the cache. Once the cache is full, the algorithm must choose a victim page to evict from the cache each time there is a page fault, so that there will be room for the currently requested (faulting) page.

Previous work [11] has considered the related *weighted caching* problem, in which the cost to bring a page into the cache may vary from page to page. The goal is to minimize the total cost of all page faults. Recent work [6, 15, 3] has extended this

*E-mail address:* kimbrel@watson.ibm.com (T. Kimbrel).


PII: S0304-3975(00)00263-2

model to the case in which page sizes may vary; that is, each page requires some (integer) number of cache locations, rather than just one. Irani [6] considered the case in which the costs are uniform and the case in which the cost to fetch a page is proportional to its size. Young [15] and Cao and Irani [3] considered the general case of arbitrary costs and sizes. This allows the modelling of network bandwidth costs, access latency, etc. in a distributed information retrieval system such as the World Wide Web (WWW).

In this paper, we extend these results to allow each page to be assigned an expiration time at the time it enters the cache. Once the expiration time has passed, the page must be discarded and a new copy must be fetched to satisfy any later requests. This models the behavior of many popular WWW *proxy caches* [5, 10, 12], caches that are placed in a network between users and WWW servers to speed the users' observed response time. Our model corresponds to the time to live (TTL) cache consistency mechanism employed in these current systems. TTL provides a weak consistency guarantee; for instance, it may guarantee that a delivered copy of an item was current at least as late as some fixed amount of time before the delivery time.

A recent empirical study [9] considered the TTL mechanism as well as two others, *invalidation* and *polling*. In an invalidation scheme, WWW servers keep track of caches which contain copies of a document, and notify the caches when documents are modified. In a polling scheme, the caches poll the servers to validate that documents in the cache are not stale. Via simulations of these mechanisms, this study concludes that invalidation can provide strong consistency guarantees with little additional overhead when compared to the weaker TTL-based mechanism. However, another study [4] concludes that a variant of TTL is the best consistency mechanism to use in the WWW, and current WWW caching systems employ TTL-based consistency [5, 10, 12].

## 2. Model and overview of results

Our model generalizes the classical paging model. Each time a page is brought into the cache, it is assigned a *time to live* (TTL), measured in the number of (future) requests served. Equivalently, we will think of the induced *expiration time* measured in terms of the number of requests served since the beginning of the request sequence. Once the expiration time passes, the page must be discarded and, if the page is referenced again, a new copy must be brought into the cache.

In real applications, the time required to serve a request for a page that is present in the cache may be very different from the time to serve a cache miss. The time may also depend on the size of the page and other factors, such as the locations of the cache and the server in a network. Our use of a single request as the unit of time is for notational convenience only. Our algorithms require only that the requests and expiration times are linearly ordered in time. Thus, our algorithms apply and achieve the stated performance without the assumption of unit time per request. In fact, our algorithms do not use the advance knowledge of the expiration times, and thus, our

results hold even in the case in which, like page requests, expirations arrive online but are known in advance to the offline adversary.

At this point, the reader may wonder why a new algorithm is needed at all. Previously known algorithms could be used, simply by treating a request for a page that has expired as though it were a request for a new page. However, it is not a priori clear that previous algorithms achieve the same performance under the new model. It is conceivable that a competitive online algorithm would need to take into account the expiration time of a page in making its replacement decisions, or even that the offline adversary could force a greater competitive ratio by taking advantage of the weak consistency guarantee of the TTL mechanism. These concerns will be made explicit below in Section 3.

Each page $p$ is associated with a cost $cost(p)$ that is paid each time $p$ is fetched into the cache. Each page $p$ also has a positive integer size $size(p)$. We use $k$ to denote the size of the fast memory, or cache. At any time, the cache can hold a set of pages whose sizes sum to $k$ or less.

We will refer to a request's index in the sequence of requests as the *time* of the request. We make the natural assumption that if the same page is fetched at times $t_1$ and $t_2$, where $t_2 > t_1$, then the expiration time assigned at time $t_2$ is at least as great as that assigned at $t_1$. That is, we assume that the expiration time (of a single page) is a monotonically nondecreasing function of the time at which the page is fetched. This function may differ for different pages.

We analyze the online problem of paging with expiration times in the standard competitive analysis framework [13, 8]. See Borodin and El Yaniv [2] for background on competitive analysis of online algorithms.

**Definition.** An algorithm $A$ is *c-competitive* if there is some constant $d$ such that for any input $x$,

$$cost_A(x) \leqslant c\, cost_{\mathrm{opt}}(x) + d,$$

where $cost_A(x)$ is the cost incurred by algorithm $A$ on input $x$ and $cost_{\mathrm{opt}}(x)$ is the optimal (offline) cost on $x$. The *competitive ratio* of $A$ is the infimum over $c$ such that $A$ is $c$-competitive. If $A$ is $c$-competitive and no online algorithm is $c'$-competitive for $c' < c$, algorithm $A$ is said to be *strongly competitive*.

The well-known lower bound of $k$ on the competitive ratio of any deterministic algorithm for paging applies directly to the problem of paging with expiration times, in both cases of uniform costs and sizes and of arbitrary costs and sizes. In Section 3, we note some differences between the current problem and the classical paging problem. In Section 4, we introduce an algorithm for the uniform cost and size case that we call *least recently used with lazy updates* ($LRULU$) and show that it is $k$-competitive. In Section 5, we extend this to the case of arbitrary costs and sizes and present an algorithm that we call *landlord with lazy updates* ($LLU$). The results are extended to Young's loose model of competitiveness [14, 15] in Section 6. In Section 7, we consider

the implementation costs of the algorithms, and in Section 8 we present directions for further research.

## 3. Comparison with classical paging

In this section, we give simple examples to illustrate two counterintuitive properties of the problem at hand.

First, we show that the optimal offline policy does not always use an old copy of a page as long as is allowed by the expiration time. That is, it can be the case that the optimal algorithm discards a page before it expires, even though it could use the page to satisfy a future request without faulting. Suppose that the cache size $k$ is two, and consider the request sequence *abcabacb*. Suppose that pages $b$ and $c$ are long-lived, so that their expiration times are of no concern, but that page $a$ expires four requests after it is brought into the cache. Thus, $a$ will expire after its second reference, and a new copy must be brought into the cache. In effect, the sequence $abcaba'cb$ must be served, where $a'$ is a page distinct from $a$. However, another option is to discard $a$ after its first reference, and to fetch a new copy into the cache on the second reference. In this case, the sequence that must be served is $abca'ba'cb$. Using the well-known MIN algorithm of Belady [1], which always evicts the page that is not referenced for the longest time among all pages in the cache, it is easy to verify that the optimal number of faults on the second sequence is five, whereas the optimal number of faults on the first is six.

The second counterintuitive observation relates to the number of phases of a request sequence, defined according to the usual analysis of the LRU algorithm. Each phase of a request sequence is a maximal-length substring of $k$ requests for distinct pages. The first phase begins with the first request, and ends just before the request to the $(k+1)$th distinct page since the beginning of the phase. The next phase begins with that request, and subsequent phases are defined similarly.

We will say that a page is *updated* when an algorithm delivers a particular copy of a page (i.e., uses a cached copy to satisfy a request) for the last time. The intuitive notion we would like to capture is that the algorithm has decided to stop using an old copy of the page and that a new copy should be used to satisfy the next request for the page. The effect is as if the next request for that page is for a different page. We thus modify the definition of phases as follows: we consider requests to a page that has been updated to be requests for a different page. Note that without expiration times, the phases are a property of the request sequence only, and not of any page replacement policy, whereas in the new problem with expiration times, the phase decomposition and the number of phases may depend on the choice of the algorithm.

A page in the cache must be updated no later than its expiration time. However, as we have seen, it may be advantageous to update a page earlier than its expiration time. It would seem that a natural strategy to minimize the number of phases of a request sequence would be to update pages only when they expire. The surprising fact is that

the number of phases of a request sequence can be larger for a policy that discards pages only when they expire than for a policy that sometimes updates pages earlier than is required by the expiration time. Consider the sequence *abacaca*, and again suppose the cache size $k$ is two. Suppose that pages $b$ and $c$ are long-lived, but the lifetime of $a$ is only five requests. The sequence $abacaca'$, corresponding to a lazy update policy, has three phases: *aba*, *cac*, and $a'$. The sequence $abaca'ca'$, which corresponds to a policy that updates $a$ earlier than necessitated by the expiration time, has only two phases: *aba* and $ca'ca'$; a similar remark applies to the sequence $aba'ca'ca'$.

Some applications of the TTL consistency mechanism will preclude these anomalies. For instance, if the exact time at which a page will be modified (at the server in the Web environment, say) is known in advance, the TTL can be set accordingly. In the first example above, there would be no advantage to fetching a new copy of $a$ to satisfy its second request, since the new copy would be known to be identical to the old copy, and would expire at the same time. A similar observation can be made for the second example. This application is a special case of the TTL mechanism and thus our results hold in this case as well as in the more general case. In this special case, there is no need for a new algorithm, since a modified page can simply be treated as if it were a different page, and previously known algorithms can be used. We note that current Web proxy caching mechanisms [5, 10, 12] use the TTL mechanism in the more general sense, for instance, by assigning default TTL values to documents that do not have them assigned a priori by the server.

## 4. The LRULU algorithm for paging with expiration times

LRULU is an adaptation of the well-known *least recently used* (LRU) algorithm. LRULU replaces the page least recently used among all pages in its cache, and updates pages lazily, i.e., only when it is necessary based on the expiration time. Note that a page is updated implicitly when it is evicted from the cache. By lazy update, we mean that LRULU never overrides the LRU replacement rule in order to evict a page that has been referenced more recently, but has an earlier (future) expiration time, than the least recently used page. Of course, once a page has expired, the effect is as if subsequent references are to a different page, so the expired page may as well be evicted. We assume that both LRULU and the optimal offline algorithm OPT evict any pages that have expired or, in the case of OPT, any page that it has elected to update before its expiration time.

We have seen that lazy update can be sub-optimal. Nonetheless, we will show that LRULU is (strongly) $k$-competitive. We have also seen that lazy update can result in a phase decomposition with more phases than nonlazy update. Thus, we cannot adapt the usual phase-based proof (see, for example, [7]) that LRU is $k$-competitive for paging to show competitiveness of LRULU, since the optimal phase decomposition may differ from that of LRULU. Instead, we adapt the alternate proof of [8] based on a potential function.

**Theorem 1.** *LRULU is k-competitive for paging with expiration times.*

**Proof.** We define a potential function $\Phi$ as follows. First, we define for each page $p$ the value $a(p)$ based on the states of $p$ in LRULU's cache and the optimal algorithm OPT's cache. If $p$ is not present in LRULU's cache, then $a(p)=0$. $a(p)=0$ also in the case that $p$ is in both caches, but has an earlier expiration time in LRULU's cache than in OPT's. If $p$ is in both caches and the expiration time in LRULU's cache is at least as great as that in OPT's, or $p$ is present in LRULU's cache but not in OPT's, $p$ is said to be a *good* page. $a(p)$ is based on the position of $p$ in LRULU's queue, *counting only the good pages*, where $k$ is the value at the MRU (most recently used) end of the queue. That is, the most recently used good page is assigned $k$, the next most recently used good page is assigned $k-1$, and so on. If there are $k' \leqslant k$ good pages in LRU's queue, they are assigned values in the range $[k-k'+1, k]$.

The potential function $\Phi$ is

$$\sum_{p \in S} (k - a(p))$$

where $S$ is the set of pages present in OPT's cache.

We show that for each request served,

$$cost(\text{LRULU}) + \Delta\Phi \leqslant k\, cost(\text{OPT}),$$

where $\Delta\Phi$ is the change in the potential function. First, we show that the fault costs incurred by the algorithms satisfy this relation. Next, we show that updates forced by expiration for either algorithm, and any earlier updates by OPT, satisfy the relation as well. $k$-competitiveness of LRULU follows by a standard argument (see, for example, [7]). We assume that OPT, like LRULU, fetches pages into the cache on demand only, i.e., only at the time of a request for a page that is not present in the cache. It is easy to show that there is such an optimal algorithm.

For each request, we first allow OPT to satisfy the request, then we allow LRULU to satisfy the request. We show that the relation holds at each step.

- OPT serves a request: If the requested page $p$ is in OPT's cache, both algorithms' costs are zero, and $\Phi$ is unchanged. If $p$ is missing from OPT's cache, OPT's cost is one and LRULU's is zero, so we must show that $\Delta\Phi \leqslant k$. The change in $\Phi$ due to OPT's eviction (if any) is negative or zero. Regardless of the value of $a(p)$ before and after OPT's fetch, $p$'s contribution to $\Phi$ can increase by at most $k$. OPT may receive a later expiration time than LRULU has for $p$, if $p$ is in LRULU's cache, so that $p$ ceases to be a good page. Some number of other pages' $a(\cdot)$ values may increase by one; this can only cause $\Phi$ to decrease.
- LRULU serves a request: OPT's cost is zero. If the requested page $p$ is in LRULU's cache, LRULU's cost is zero, and $\Phi$ is unchanged. If $p$ is missing from LRULU's cache, LRULU's cost is one, so we must show that $\Phi$ decreases by at least one.

  First, suppose that the page $q$ evicted (if any) by LRULU is not good, i.e., that $a(q)=0$. This eviction does not alter $\Phi$. The currently requested page $p$ is already

in OPT's cache, since OPT has already served the request. LRULU receives an expiration time at least as great as the expiration time that OPT has for $p$; thus, the fetch of $p$ by LRULU decreases $\Phi$ by $k$. $\Phi$ increases by at most $k'$, where $k' \leqslant k-1$ is the number of good pages before the fetch of $p$, due to their changes in position in LRULU's queue. Thus $\Delta\Phi = k' - k \leqslant -1$.

Now, suppose $q$ is good, i.e., that $a(q) > 0$. Since $p$ is in OPT's cache, there must be at least one page $r$ in LRULU's cache that is not in OPT's cache. Let $k' \leqslant k$ be the number of good pages. Note that $r$ is good, but that $r$ does not contribute positively to $\Phi$ since it is not in OPT's cache. We consider two cases:

- If $q$ is in OPT's cache, the change in $a(q)$ from $k-k'+1$ to zero results in an increase in $\Phi$ of $k-k'+1$. $p$ receives an expiration time in LRULU's cache at least as great as the time it has in OPT's cache. Thus, $a(p)$ changes from zero to $k$, contributing a decrease of $k$ in $\Phi$. $r$ does not contribute to $\Phi$, so at most $k'-2$ good pages each contribute an increase of one due to their shifts in position in LRULU's queue. Putting these together, $\Delta\Phi \leqslant k-k'+1-k+k'-2 = -1$.
- If $q$ is not in OPT's cache, then the eviction does not change $\Phi$. Again, $a(p)$ changes from zero to $k$, contributing a decrease of $k$ to $\Phi$. At most $k'-1$ good pages each contribute an increase of one due to their shifts in position in LRULU's queue. Putting these together, $\Delta\Phi \leqslant -k+k'-1 \leqslant -1$.

For the updates, we consider four cases. Both algorithms' costs are zero, so we need only to show that $\Phi$ does not increase.

- Suppose page $p$ expires for LRULU, but does not expire for OPT.
  LRULU must update $p$. We assume that LRULU evicts $p$ from its cache. Since $p$ does not expire for OPT, OPT has a later expiration time for $p$, or does not have $p$ in its cache. In both cases, there is no change in $\Phi$ due to $p$'s ejection from LRULU's cache, since $p$ is not good. In the latter case, some number of pages' $a(\cdot)$ values may increase by one; this can only result in a decrease in $\Phi$.
- Suppose page $p$ expires for OPT but does not expire for LRULU.
  OPT must update $p$; we assume that OPT evicts $p$ from its cache. The nonnegative contribution of $p$ to $\Phi$ is eliminated, thus only decreasing $\Phi$.
- Suppose OPT elects to update (evict) page $p$ before its expiration time.
  Again, the nonnegative contribution of $p$ to $\Phi$ is eliminated, thus only decreasing $\Phi$.
- Suppose page $p$ expires both for OPT and for LRULU.
  $p$'s nonnegative contribution to $\Phi$ is eliminated, and any changes of the positions of pages other than $p$ in LRULU's queue can only decrease $\Phi$. □

## 5. The LLU algorithm for the nonuniform size and cost case

In this section, we extend the result of the previous section to the case in which the sizes of the pages and the costs to retrieve them are arbitrary rather than uniform.

```
Evict any page q which has expired
if p is in the cache
  set credit(p) to cost(p)
Else
  Until there is room for p
    Let δ be min_{q∈cache} credit(q)/size(q)
    For each page q in the cache
      decrease credit(q) by δ size(q)
      If credit(q) = 0, evict q
  Bring p into the cache
  set credit(p) to cost(p)
```

Fig. 1. The LLU algorithm.

We adapt the Landlord algorithm of Young in the same way that LRU was adapted for the uniform size and cost case. It was recently shown [15] that the Landlord algorithm is $k$-competitive for paging with arbitrary costs and sizes. Our contribution is to allow the additional constraints corresponding to expiration times. We call the algorithm landlord with lazy updates, or LLU. LLU maintains a value $credit(p)$ for each page $p$ in its cache. On a request for a page $p$, LLU behaves as described in Fig. 1.

Following Young, we prove the following more general result concerning the case in which the optimal algorithm OPT is provided with a cache smaller than or the same size as that of LLU. $k$-competitiveness of LLU is a simple corollary. The more general result is used to show loose competitiveness of LLU and LRULU in Section 6.

**Theorem 2.** *LLU is $k/(k-h+1)$-competitive for paging with expiration times and arbitrary page sizes and retrieval costs, when LLU is provided with a cache of size $k$ and the optimal algorithm OPT is provided with a cache of size $h \leqslant k$.*

The proof closely parallels that of [15] for the same problem without expiration times.

**Proof.** We use the potential function

$$\Phi = (h-1)\sum_{p\in \mathrm{LLU}} goodcredit(p) + k\sum_{p\in \mathrm{OPT}} (cost(p) - goodcredit(p))$$

where $goodcredit(p)$ is equal to $credit(p)$ if $p$ has an expiration time in LLU's cache at least as great as in OPT's, or $p$ is not in OPT's cache, and zero otherwise. For $p$ not in LLU's cache, we define $goodcredit(p) = credit(p) = 0$. Notice that $\Phi = 0$ at the beginning of the request sequence when both caches are empty, and that $\Phi$ is always nonnegative since $goodcredit(p) \leqslant credit(p) \leqslant cost(p)$.

We consider in isolation each change in either LLU's state or OPT's state in reaction to a request. We will show that for each step,

$$(k-h+1)\,cost(\mathrm{LLU})+\Delta\Phi\leqslant k\,cost(\mathrm{OPT}),$$

where $\Delta\Phi$ is the change in the potential function.

- OPT evicts page $p$. Since $goodcredit(p)\leqslant cost(p)$, $\Delta\Phi=-k(cost(p)-goodcredit(p))\leqslant 0$.
- OPT fetches page $p$. OPT's cost is $cost(p)$, so we must show $\Delta\Phi\leqslant k\,cost(p)$. If $p$ is in LLU's cache and OPT receives a later expiration time for $p$ than LLU has, $goodcredit(p)$ changes from $credit(p)\geqslant 0$ to zero, so

  $$\Delta\Phi=-(h-1)credit(p)+k\,cost(p)\leqslant k\,cost(p).$$

  Otherwise, $p$ is not in LLU's cache, or the expiration time received by OPT is the same as that LLU has. In either case, $goodcredit(p)=credit(p)$ both before and after OPT's fetch, so that

  $$\Delta\Phi=k(cost(p)-credit(p))\leqslant k\,cost(p).$$

- LLU decreases $credit(q)$ by $\delta$ for each $q$ in its cache. LLU decreases credits only on a fault on some page $p$ for which it does not have enough space in its cache. We assume that OPT has already satisfied the request, so that $p$ is in OPT's cache. Let $Q$ denote the set of pages $q$ in LLU's cache such that $goodcredit(q)\neq 0$, and let $R\subseteq Q$ denote the set of pages $r$ in OPT's cache such that $goodcredit(r)\neq 0$. For any set $S$ of pages, let $\|S\|=\sum_{s\in S} size(s)$. The decrease in credit for each page $q$ in LLU's cache is $\delta size(q)$. Thus

  $$\Delta\Phi=-\delta((h-1)\|Q\|-k\|R\|).$$

  $\Delta\Phi$ will be less than or equal to zero if we can show

  $$\frac{\|R\|}{\|Q\|}\leqslant\frac{h-1}{k}.$$

  Let $B$ denote the set of pages $b$ in both caches with $goodcredit(b)=0$, i.e., the "bad" pages that expire sooner for LLU than for OPT. Let $P$ denote the set of pages in OPT's cache that are not in LLU's; note $p\in P$. Finally, let $u_{\mathrm{LLU}}$ denote the amount of unused space in LLU's cache, and similarly let $u_{\mathrm{OPT}}$ denote the amount of unused space in OPT's cache. Given these definitions, we can write

  $$\|R\|=h-\|B\|-\|P\|-u_{\mathrm{OPT}}$$

  and

  $$\|Q\|=k-\|B\|-u_{\mathrm{LLU}}.$$

  Because $p\in P$ we have that $\|P\|>0$, and since $p$ does not fit in LLU's cache, we have that $u_{\mathrm{LLU}}<\|P\|$. From these it follows that $\|R\|/\|Q\|\leqslant(h-1)/k$ as needed.

- LLU evicts a page $p$. $credit(p) = 0$ when $p$ is evicted, so $\Delta\Phi = 0$.
- LLU brings page $p$ into the cache. LLU pays $cost(p)$, so we must show that $\Delta\Phi \leqslant -cost(p)(k-h+1)$. We can assume $p$ is in OPT's cache, since LLU fetches a page only when it is requested, and OPT must serve the request. LLU receives an expiration time for $p$ at least as late as the one OPT has, so $goodcredit(p)$ changes from zero to $cost(p)$. $\Delta\Phi$ is thus $(h-1)\,cost(p) - k\,cost(p) = -(k-h+1)\,cost(p)$, as needed.
- LLU does not fault on $p$ and sets $credit(p)$ to $cost(p)$. $p$ must be in OPT's cache. If $p$ has a later expiration time in OPT's cache than in LLU's, then $goodcredit(p) = 0$ both before and after the change to $credit(p)$, and $\Delta\Phi = 0$. Otherwise,

  $$\Delta\Phi = ((h-1)-k)\Delta credit(p) \leqslant 0,$$

  where $\Delta credit(p)$ is the nonnegative change in $credit(p)$.
- Page $p$ expires for LLU and is not in OPT's cache. $\Delta\Phi = -(h-1)credit(p) \leqslant 0$.
- Page $p$ expires for LLU and does not expire for OPT. In this case, $goodcredit(p) = 0$, so $\Delta\Phi = 0$.
- Page $p$ expires for OPT but not for LLU. $\Delta\Phi = k(credit(p) - cost(p)) \leqslant 0$.
- Page $p$ expires for both LLU and OPT. All of $p$'s nonnegative contribution to $\Phi$ is eliminated, so $\Delta\Phi \leqslant 0$. □

**Corollary 3.** *LLU is k-competitive for paging with expiration times and arbitrary page sizes and retrieval costs.*

## 6. Extension to the loose competitiveness model

Theorem 1 and Corollary 3 are easily extended to the loose competitiveness model of Young [15]. This model helps explain why many online algorithms such as LRU perform much better in practice than indicated by the worst-case bounds in the usual competitive analysis model. The model is motivated by two observations. First, only a small fraction of possible cache sizes may be "bad" for an algorithm on any particular sequence of requests. The parameter $\delta$ in the following definition captures this idea. Second, if the absolute cost incurred by an online algorithm is low enough, we need not worry about the competitive ratio. For instance, in paging, if the fault rate is less than the ratio of the fast memory retrieval time to the backing store retrieval time, then the total elapsed time of a program is at most twice the optimal elapsed time. The parameter $\varepsilon$ captures this notion.

**Definition.** An algorithm $A$ is $(\varepsilon, \delta)$-*loosely c-competitive* if, for any request sequence $r$ and any integer $n > 0$, at least $(1-\delta)n$ of the values $k \in \{1, 2, \ldots, n\}$ satisfy

$$cost_A(r,k) \leqslant \max\left(c\,cost_{\mathrm{OPT}}(r,k), \varepsilon \sum_{f \in r} cost(f)\right)$$

where $cost_A(r,k)$ denotes the cost incurred by algorithm $A$ on input $r$ with a cache of size $k$.

**Theorem 4.** *LLU and LRULU are* $(\varepsilon,\delta)$*-loosely c-competitive for* $c = e(1/\delta)\lceil \ln 1/\varepsilon \rceil$.

**Proof.** The result for LLU follows from our Theorem 2 and [15, Theorem 2]: *Every* $k/(k-h+1)$*-competitive algorithm is* $(\varepsilon,\delta)$*-loosely c-competitive for any* $0<\varepsilon,\delta<1$ *and* $c = e\,1/\delta\lceil \ln 1/\varepsilon \rceil$. Since a suitably defined version of Landlord (i.e., one that uses LRU to break ties) is a generalization of LRU, this applies to LRULU (and in fact to a similar extension of any *marking algorithm*; see, e.g., [2] for a definition) as well. □

## 7. Implementation costs

At first glance, it would appear that the LLU algorithm requires $\Omega(k)$ work per request, since on a cache miss, the credit value of every page in the cache must be updated. Cao and Irani [3] show how to implement the Landlord algorithm (which they called Greedy-Dual-Size) so that only $O(\log k)$ work is necessary per request. The first modification is that a page's credit is calculated as its cost divided by its size, rather than its cost. Pages' credits decrease uniformly rather than proportionally to their sizes. It is easy to see that this results in the same behavior as the algorithm as stated earlier. The other modification is that the credit of each page is offset by a time-dependent "inflation value". When a page's credit is calculated on a reference to the page, the current inflation value is added. On an eviction, the inflation value is increased to the credit of the page with the smallest credit (i.e., the evicted page), rather than decreasing the credit of every page. A page is evicted when the inflation value reaches the credit of the page. This way, the credit of each page needs to change only when the page is referenced, and the pages can be ordered by a priority queue keyed on their credits. The expiration times of pages can be stored in a priority queue as well, so that the overall implementation cost is $O(\log k)$ work per request.

## 8. Discussion and directions for further research

In this paper, we have extended recent results on paging with varying costs and sizes to handle an additional feature of World Wide Web caching mechanisms, namely that of data that expires and must be refreshed. The model for data expiration is derived from that used by several popular proxy caching systems [5, 10, 12]. This is the first theoretical treatment of this problem we are aware of.

Our algorithm applies to the commonly used TTL-based cache consistency mechanism. It is easy to see that the previously proposed Landlord algorithm [15, 3] is $k$-competitive in the case of invalidation-based consistency. In that case, one can

simply treat requests to a page that has been invalidated as requests for a new page. The extra information, i.e., the fact that the invalidated page will never be referenced again, can only improve the performance of an online algorithm, and is known to the optimal offline algorithm in any case. We have not considered the case of polling-based consistency. An interesting question is the tradeoff between polling costs and cache miss costs.

In our model, we assume that any time a page expires, the full cost of retrieving it must be paid the next time it is requested. In the WWW, there is another option, based on the polling mechanism. The proxy can send an "if-modified-since" message to the server. If the requested page has not been modified since it was last retrieved, the server responds with a message indicating this fact and extending the page's time-to-live; otherwise, it sends the new version of the page. A replacement algorithm may benefit from keeping an expired page in the cache and revalidating it if the cost of retrieving the page is greater than that of retreiving a (short) revalidation message. Thus, our model is accurate for small WWW pages, for which the time to retrieve the page is comparable to the time to retrieve a revalidation message, but it is less accurate for large pages. Another direction for further research is to extend the model to encompass this feature of WWW caching mechanisms.

## Acknowledgements

The author wishes to thank Arun Iyengar for discussions relating to this work and for commenting on an earlier draft of this paper, the anonymous referees for several suggestions for improving the presentation, and Jayram Thathachar for pointing out an error in one of the proofs.

## References

[1] L.A. Belady, A study of replacement algorithms for virtual storage computers, IBM Systems J. 5 (2) (1966) 78–101.
[2] A. Borodin, R. El-Yaniv, Online Computation and Competitive Analysis, Cambridge University Press, Cambridge, 1998.
[3] P. Cao, S. Irani, Cost-aware WWW proxy caching algorithms, Proc. USENIX Symp. on Internet Technologies and Systems, Monterey, CA, December 8–11, 1997, pp. 193–206.
[4] J. Gwertzman, M. Seltzer, World-wide web cache consistency, Proc. 1996 USENIX Tech. Conf., San Diego, CA, January 22–26, 1996, pp. 141–151.
[5] D. Hardy, M. Schwartz, D. Wessels, Harvest User's Manual, http://harvest.transarc.com/afs/transarc.com/public/trg/Harvest/user-manual.
[6] S. Irani, Page replacement with multi-size pages and applications to web caching, Proc. 29th ACM Symp. on Theory of Computation, El Paso, TX, May 4–6, 1997, pp. 701–710.
[7] S. Irani, A. Karlin, Online computation, in: D. Hochbaum (Ed.), Approximation Algorithms for NP-Hard Problems, PWS Publishing Company, Massachusetts, 1997, pp. 521–564.
[8] A. Karlin, M. Manasse, L. Rudolph, D. Sleator, Competitive snoopy caching, Algorithmica 3 (1988) 79–119.
[9] C. Liu, P. Cao, Maintaining strong consistency in the world wide web, IEEETC: IEEE Trans. Comput. 47 (1998) 445–457.

[10] A. Luotonen, H.F Nielsen, T. Berners-Lee, CERN http User's Guide, http://www.w3.org/Daemon/User/Admin.html.
[11] M. Manasse, L. McGeoch, D. Sleator, Competitive algorithms for server problems, J. Algorithms 11 (1990) 208–230.
[12] O. Pearson, Squid User's Guide, http://cache.is.co.za/squid.
[13] D. Sleator, R.E. Tarjan, Amortized efficiency of list update and paging rules, Commun. ACM 28 (1985) 202–208.
[14] N.E. Young, The $k$-server dual and loose competitiveness for paging, Algorithmica 11 (6) (1994) 525–541.
[15] N.E. Young, On-line file caching, Proc. ACM-SIAM Symp. on Discrete Algorithms, San Francisco, CA, January 25–27, 1998, pp. 82–86.