Basic Research in Computer Science

On Competitive On-Line Paging with Lookahead

Dany Breslauer

BRICS Report Series **RS-95-50**

ISSN 0909-0878 **September 1995**

See back inner page for a list of recent publications in the BRICS
Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk

BRICS publications are in general accessible through WWW and
anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

On Competitive On-Line Paging with Lookahead

Dany Breslauer*

Abstract

This paper studies two methods for improving the competitive efficiency of on-line paging algorithms: in the first, the on-line algorithm can use more pages; in the second, it is allowed to have a lookahead, or in other words, some partial knowledge of the future. The paper considers a new measure for the lookahead size as well as Young's resource-bounded lookahead and proves that both measures have the attractive property that the competitive efficiency of an on-line algorithm with k extra pages and lookahead l depends on $k+l$. Hence, under these measures, an on-line algorithm has the same benefit from using an extra page or knowing an extra bit of the future.

1 Introduction

The paging problem models a virtual memory computer system with $\mathcal{K}$ pages of fast memory and $\mathcal{N}-\mathcal{K}$ pages of slow memory. The system has to serve a sequence of page requests. To serve a request, the requested page must be in fast memory. If the page is not in fast memory, a *page fault* occurs, and the page has to be brought into fast memory by evicting one of the pages already in the fast memory and replacing it by the requested page. The paging problem is that of deciding which page to evict from the fast memory. The performance of a paging algorithm is measured as the number of page faults it makes. We denote the number of page faults made by a paging algorithm $\mathcal{A}$ on the sequence of pages σ by $\mathcal{C}_{\mathcal{A}}(\sigma)$.

Belady [2] gave an optimal algorithm for the paging problem. His algorithm evicts the page in fast memory that will not be requested for the longest time. This algorithm is *off-line* in the sense that it requires knowledge of the future requests. In contrast, a paging algorithm is said to be *on-line* if its decision of which page to evict does not depend on future requests.

*Basic Research in Computer Science, Centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark.
Partially supported by the ESPRIT Basic Research Action Program of the EC under contract #7141 (ALCOM II).

In *competitive analysis* of on-line paging algorithms, the performance of an on-line algorithm $\mathcal{A}$ is compared to that of an optimal off-line algorithm OPT. Algorithm $\mathcal{A}$ is said to be h-competitive if $\mathcal{C}_{\mathcal{A}}(\sigma) \leq h \cdot \mathcal{C}_{OPT}(\sigma) + a$, for all request sequences σ and some fixed constant a. Sleator and Tarjan [9] showed that for any on-line paging algorithm $\mathcal{A}$ with $\mathcal{K} + k$ pages (of fast memory), there exist request sequences σ with $\mathcal{C}_{OPT}(\sigma)$ arbitrarily large, such that,

$$\mathcal{C}_{\mathcal{A}}(\sigma) \geq \frac{\mathcal{K} + k}{k + 1} \cdot \mathcal{C}_{OPT}(\sigma).$$

They also showed that there exist on-line paging algorithms $\mathcal{A}$, such that for any request sequence σ,

$$\mathcal{C}_{\mathcal{A}}(\sigma) \leq \frac{\mathcal{K} + k}{k + 1} \cdot \mathcal{C}_{OPT}(\sigma) + \mathcal{K}.$$

In other words $\frac{\mathcal{K}+k}{k+1}$ is the best competitive ratio achievable by an on-line paging algorithm with $\mathcal{K} + k$ pages when compared to an optimal off-line paging algorithm with $\mathcal{K}$ pages.

If the on-line paging algorithms are allowed to make random choices, the number of page faults $\mathcal{C}_{\mathcal{A}}(\sigma)$ is defined to be the *expected* number of page faults. When a randomized on-line paging algorithm with $\mathcal{K}$ pages is confronted with the *oblivious adversary*, an adversary that knows the behavior of the algorithm, but can not see its random internal state, it has been shown by Fiat et al. [4] that the best attainable competitive ratio is at least $\mathcal{H}(\mathcal{K})$, where $\mathcal{H}(\mathcal{K}) = 1 + \frac{1}{2} + \cdots + \frac{1}{\mathcal{K}}$ are the *harmonic numbers.* McGeoch and Sleator [8] gave a randomized on-line paging algorithm that is $\mathcal{H}(\mathcal{K})$-competitive, improving a $2 \cdot \mathcal{H}(\mathcal{K})$-competitive algorithm given by Fiat et al. [4].

Off-line and on-line algorithms are two extremes. An obvious generalization that comes to mind is that of algorithms that are allowed to see some part of the future request sequence before making their decision of which page to evict. Such algorithms are called *on-line algorithms with lookahead.* Unfortunately, by allowing an algorithm to see the next l requests, for any finite constant l, an on-line algorithm does not gain any advantage in the worst case, since any request sequence $\sigma = \sigma_1, \ldots, \sigma_p$ can be replaced by the request sequence $\sigma_1^l, \ldots, \sigma_p^l$ in which each request is repeated l times, thus, hiding the future requests from the algorithm. Empirically, however, on-line algorithms benefit from using such a lookahead and several authors have suggested other models for the evaluation of on-line algorithms in which this lookahead makes a difference [3, 7, 6, 10].

Young [11] suggested an alternative measure for the lookahead size which he calls *resource-bounded lookahead.* He showed that there exists an on-line paging algorithm with resource-bounded lookahead l that has competitive ratio[1] $\max\left\{2 \cdot \frac{\mathcal{K}+k}{k+l+1}, 2\right\}$, and that no such algorithm can achieve a competitive ratio

[1]Lookahead l in our notation corresponds to lookahead $l + 1$ in Young's notation.

smaller than $\frac{\mathcal{K}+k+l}{k+l+1}$. Young [11, 12] also gave a randomized on-line paging algorithm with $\mathcal{K}$ pages that is $2 \cdot (\ln \frac{\mathcal{K}}{l} + 1)$-competitive and showed that no such algorithm can attain a competitive ratio that is smaller than $\ln \frac{\mathcal{K}+l}{l} - \ln\ln \frac{\mathcal{K}+l}{l} - \frac{2}{l}$.

Albers [1] defined another measure for the lookahead size that she calls *strong lookahead*. Albers showed that under this measure, there exists an on-line paging algorithm with $\mathcal{K}$ pages and strong lookahead l, $0 \leq l \leq \mathcal{K} - 2$, that has competitive ratio $\mathcal{K} - l$ and that this competitive ratio is the best possible. Albers also considered randomized paging algorithms with $\mathcal{K}$ pages and strong lookahead l, $0 \leq l \leq \mathcal{K} - 2$, and proved that there exists an algorithm that is $2 \cdot \mathcal{H}(\mathcal{K} - l)$-competitive and that no algorithm is better than $\mathcal{H}(\mathcal{K} - l)$-competitive.

In this paper we define a new measure for the lookahead size, which we call *natural lookahead*. We show that under this measure there exists a deterministic on-line paging algorithm with $\mathcal{K} + k$ pages and *natural lookahead* l that has competitive ratio $\frac{\mathcal{K}+k+l}{k+l+1}$ and that this is the best attainable competitive ratio. The same tight competitive bounds are shown also for Young's resource-bounded lookahead. Notice that these bounds have the attractive property that the competitive ratio is a function of $k + l$. Thus, under these measures for the lookahead size, an on-line algorithm obtains the same benefit from using an extra page or knowing an extra bit of future requests.

The paper is organized as follows. Section 2 introduces the basic terminology and concepts. Section 3 shows that natural lookahead can be simulated by extra pages and Section 4 gives the competitive bounds with natural and resource-bounded lookaheads. Concluding remarks are given in Section 5.

2 Paging algorithms

Throughout the paper we compare the performance of an optimal off-line paging algorithm OPT to that of an on-line algorithm $\mathcal{A}$. The off-line algorithm OPT has $\mathcal{K}$ fast memory pages. We adopt the notation $\mathcal{A}[k, l]$ for an on-line algorithm $\mathcal{A}$ with lookahead l and k extra fast memory pages ($\mathcal{K} + k$ fast memory pages in total). An on-line algorithm without lookahead is denoted by $\mathcal{A}[k]$. The overall number of fast and slow memory pages is denoted by $\mathcal{N}$.

There are few paging algorithms that we use in this paper. Belady's optimal off-line algorithm, denoted as algorithm OPT hereafter[2], or OPT_h when it has h fast memory pages, evicts the page in fast memory that *will* not be requested for the longest time. The on-line algorithm $LRU[k]$, which stands for *least recently used*, evicts the page in fast memory that *was* not requested for the longest time[3]. We only consider here *demand paging* algorithms that exchange pages between the fast and the slow memories only when necessary.

[2] Often refered to in the literature as algorithm MIN.

[3] The results obtained in this paper for algorithm LRU can be similarly derived for ap-

We shall use Young's [11] general lookahead model. In this model, a paging algorithm has a lookahead queue whose content can be examined freely (time and space are not an issue here). An on-line paging algorithm may either service the current page request at the head of the queue (if there is one) or expose an additional request and place it at the end of the queue (if there is one). The three different lookahead definitions mentioned in the introduction differ in their measure of the lookahead size.

Young's [11] defined an on-line paging strategy to have *resource-bounded lookahead* l if it will never incur more than $l+1$ page faults on the requests in the lookahead queue. (Grove [5] considers a similar lookahead definition.) Notice that with this definition, the lookahead queue depends on the algorithm's past behavior (which pages it currently has in fast memory) and its future behavior (when the algorithm is to decide if it can extend its lookahead, it must consider the precise number of page faults it will make on the requests it already sees in the lookahead queue.)

Albers [1] defined an on-line paging strategy to have *strong lookahead* l if it never has more than $l+1$ distinct page requests in the lookahead queue. Notice that with this definition, the lookahead queue at each step may be defined *independently* of the algorithm, to contain always $l+1$ distinct page requests (fewer if the request sequence has ended).

We define an on-line paging strategy to have *natural lookahead* l if at no time it has in the lookahead queue more than $l+1$ distinct page requests that are not currently in fast memory. Notice that with this definition, the lookahead queue depends on the algorithm's past behavior (which pages it currently has in fast memory), but not on its future behavior. We argue in Section 3 that this seems to be the most natural definition of the three lookahead measures mentioned above.

It would be perhaps more intuitive to define the resource-bounded and natural lookahead measures to exclude the current request. Namely, an on-line algorithm has resource-bounded lookahead l if it will not fault more than l times on the requests it sees in the lookahead queue, excluding the current request; similarly, an on-line algorithm has natural lookahead l if it never has more than l distinct requests in the lookahead queue that are different from the current request and from the pages currently in its fast memory. If we measure the lookahead size only when the current request is for a page that is not in fast memory, these more intuitive definitions coincide with the original definitions. Notice that if the current request is for a page that is already in fast memory, the on-line algorithm does not need to evict any page and it may continue to serve the next requests without consulting the lookahead queue.

The natural lookahead closely resembles the strong lookahead. In fact, under both measures, the lookahead queue can be loaded regardless of which paging

propriate lookahead versions of other on-line paging algorithms, such as first-in first-out and flush-when-full.

strategy is being used, since the lookahead does not depend on the future behavior of the algorithm. This is obvious for the strong lookahead where more pages can be loaded until the lookahead queue contains $l+1$ distinct pages. With the natural lookahead, however, one needs to be more careful: pages can be loaded into the lookahead queue until it contains $l+1$ distinct pages that are not currently in fast memory; when some page is being evicted, since we only consider demand paging algorithms, one page is brought into fast memory and one is evicted, so the number of distinct pages in the lookahead queue which are not in fast memory is still bounded by $l+1$.

An interesting observation is that the natural lookahead is *longer* than both the resource-bounded lookahead and the strong lookahead in the sense that it always allows to see at least as many future requests. If $l \leq \mathcal{K}+k-1$, then the resource-bounded lookahead may also be longer than the strong lookahead.

The lookahead version of algorithm LRU is defined as follows:

> After the lookahead queue is loaded, the algorithm evicts the page that *will* not be requested for the longest time by consulting the lookahead queue. If there is more than one such page, namely, if there are two or more fast memory pages that are not requested within the requests in the lookahead queue, the algorithm applies the least recently used rule to these pages and evicts the page that *was* not requested for the longest time.

As mentioned above, under the strong and the natural lookahead measures, the lookahead queue can be loaded regardless of which algorithm is being used. Under the resource-bounded lookahead measure, the lookahead queue of algorithm $LRU[k,l]$ can be loaded so that $LRU[k,l]$ will fault on exactly $l+1$ pages in the lookahead queue (or fewer if the request sequence has ended): if at some point $LRU[k,l]$ decides that it would not fault on a page in its lookahead queue, it must keep that page in its fast memory from before; $LRU[k,l]$ never decides to evict such a page because of some page that is requested later in the lookahead queue.

The following lemma holds under all lookahead measures defined above.

Lemma 2.1 *If algorithm $LRU[k,l]$ faults twice on requests to the same page, then the part of the request sequence between these two page faults contains requests for at least $\mathcal{K}+k+1$ distinct pages.*

Proof: Let p be the page that is faulted twice. Consider the page request when p is evicted, after the first page fault on p. If algorithm $LRU[k,l]$ can see in the lookahead queue the request causing the second page fault on p, then necessarily, it sees before that request $\mathcal{K}+k$ distinct requests for other pages. Otherwise, say it only sees in the lookahead queue requests for $h \leq \mathcal{K}+k-1$ distinct pages that are currently in the fast memory. Of the remaining $\mathcal{K}+k-h$ fast memory pages p has been requested least recently. Hence, since the last request for p, there have been $\mathcal{K} + k - h$ requests for distinct pages, which together with the current request and the h lookahead requests that are currently in fast memory we get $\mathcal{K} + k + 1$ distinct requests. $\Box$

Finally, we shall need the following lemma that allows us to compare the performance of algorithm OPT when it starts with different configurations of pages in its fast memory. We shall use the following notation: OPT_h^C refers to the optimal off-line algorithm OPT that starts with its $h = |C|$ fast memory pages containing the pages in C; $C_\sigma \subseteq C$ is the subset of pages in C that are requested somewhere within the sequence σ; we write $C \leq_\sigma^\phi D$, if for all but at most ϕ pages $d_i \in D_\sigma$, there exist distinct pages $c_i \in C$, such that c_i is requested for the first time in σ not later than the first request for d_i.

σ :	l	c	m	a	l	c	l	h	b	i	j	k	d	e	f	
C :		c		a					b				d	e	f	g
D :				a				h		i	j	k				

Figure 1: An example with $C \leq_\sigma^2 D$.

Lemma 2.2 *Let C and D be sets of pages initially in the fast memories of OPT_h^C and OPT_g^D and assume that $h \geq g$. Then, for any request sequence σ,*

$$\mathcal{C}_{OPT_h^C}(\sigma) \leq \mathcal{C}_{OPT_g^D}(\sigma) + |D_\sigma \setminus C|.$$

In addition, if $C \leq_\sigma^0 D$, then

$$\mathcal{C}_{OPT_h^C}(\sigma) \leq \mathcal{C}_{OPT_g^D}(\sigma).$$

Proof: Let $\sigma = \sigma_1, \ldots, \sigma_p$. Denote by C^n and D^n the sets of pages in the fast memories of OPT_h^C and OPT_g^D, respectively, after processing the requests $\sigma_1, \ldots, \sigma_n$ and let c_n and d_n denote the number of page faults made by OPT_h^C and OPT_g^D, respectively, while processing the requests $\sigma_1, \ldots, \sigma_n$. Define the potential function,

$$\Phi(n) = \min\{\phi \geq 0 \mid C^n \leq_{\sigma_{n+1}, \ldots, \sigma_p}^\phi D^n\}.$$

We prove inductively that,

$$c_n \leq d_n - \Phi(n) + \Phi(0).$$

The inductive claim holds vacuously when $n = 0$. There are four cases in proving the inductive step, all are easy to verify.

- If $\sigma_n \in C^{n-1} \cap D^{n-1}$, then $c_n = c_{n-1}$, $d_n = d_{n-1}$ and $\Phi(n) = \Phi(n-1)$.

- If $\sigma_n \in C^{n-1}$ and $\sigma_n \notin D^{n-1}$, then $c_n = c_{n-1}$, $d_n = d_{n-1} + 1$ and $\Phi(n) \leq \Phi(n-1) + 1$.

Notice that if $\sigma_n \notin C^{n-1}$, then by the definition of algorithm OPT, the nearest future request for the page evicted by OPT_h^C is not earlier than requests for the other $h-1$ pages in its fast memory. Therefore, since $h \geq g$, this eviction does not affect $\Phi(n)$.

- If $\sigma_n \notin C^{n-1} \cup D^{n-1}$, then $c_n = c_{n-1} + 1$, $d_n = d_{n-1} + 1$ and $\Phi(n) \leq \Phi(n-1)$.

- If $\sigma_n \notin C^{n-1}$ and $\sigma_n \in D^{n-1}$, then $c_n = c_{n-1} + 1$, $d_n = d_{n-1}$ and $\Phi(n) = \Phi(n-1) - 1$.

Since $\Phi(p) \geq 0$ and $\Phi(0) \leq |D_\sigma \setminus C|$, we get that,

$$c_p \leq d_p - \Phi(p) + \Phi(0) \leq d_p + |D_\sigma \setminus C|,$$

and if $C \leq_\sigma^0 D$, then $\Phi(0) = 0$ and $c_p \leq d_p$. $\square$

3 Simulating lookahead

Young [11] observed that resource-bounded lookahead can be simulated by extra pages. We give next a slightly more general simulation for the natural lookahead. Notice that since the resource-bounded and the strong lookaheads are shorter than the natural lookahead, extra pages can also simulate these lookaheads. It is clear from the proof below, however, that the natural lookahead attains the limits of this simulation method. We therefore believe that the natural lookahead is a more natural measure for the lookahead size.

Lemma 3.1 *Let $\mathcal{A}[k,l]$ be an on-line paging algorithm. Then, for any integer $c \in \{1,\ldots,l\}$, there exists an on-line paging algorithm $\mathcal{B}[k+c, l-c]$, such that on any request sequence σ,*

$$\mathcal{C}_{\mathcal{B}[k+c,l-c]}(\sigma) \leq \mathcal{C}_{\mathcal{A}[k,l]}(\sigma).$$

Proof: Algorithm $\mathcal{B}[k+c, l-c]$ simulates algorithm $\mathcal{A}[k,l]$, mimicing natural lookahead l by maintaining lookahead $l-c$ together with c additional fast memory pages.

The simulating algorithm $\mathcal{B}$ maintains $\mathcal{K}+k$ pages that correspond to the $\mathcal{K}+k$ pages of algorithm $\mathcal{A}$. The additional c pages are used to simulate the lookahead. Initially, algorithm $\mathcal{B}$ keeps the $\mathcal{K}+k$ pages that contain the same pages that were originally in algorithm $\mathcal{A}$'s fast memory and loads the first c distinct page requests that are not in $\mathcal{A}$'s fast memory into the c additional pages it has.

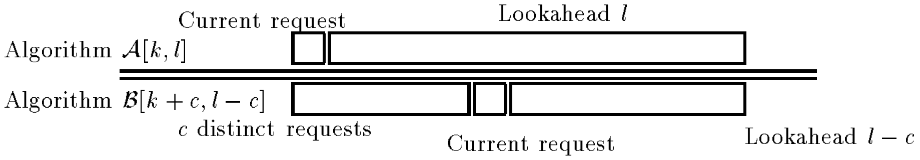

Figure 2: Simulating natural lookahead by additional pages: $B[k+c,l-c]$'s additional pages keep c distinct pages that are not currently in $A[k,l]$'s fast memory.

Next, observe that the lookahead queue of algorithm $\mathcal{B}$ contains $l-c+1$ distinct page requests that are different from the $\mathcal{K}+k+c$ pages in $\mathcal{B}$'s fast memory. These page requests together with the c distinct pages that are in $\mathcal{B}$'s additional fast memory pages form the $l+1$ distinct pages in the lookahead queue of algorithm $\mathcal{A}$. See Figure 2.

Algorithm $\mathcal{B}$ can then "evict" the same page that is evicted by algorithm $\mathcal{A}$ and "serve" $\mathcal{A}$'s "current" request by swapping the roles of the page containing the "current" request (one of the c additional pages) with one of the $\mathcal{K}+k$ pages containing the page to be evicted by $\mathcal{A}$.

If the page evicted by $\mathcal{A}$ is not requested before $\mathcal{B}$'s current request, then algorithm $\mathcal{B}$ may serve its own current request by evicting the page that was evicted by algorithm $\mathcal{A}$; load its lookahead queue until it contains $l-c+1$ distinct pages that are not currently in its fast memory (provided that the request sequence has not ended); and then, continue serving the future requests as long that they are for pages that are currently in its fast memory. It is clear that algorithm $\mathcal{B}$ makes at most as many page faults as algorithm $\mathcal{A}$ does, but somewhat in advance. □

4 Paging with lookahead

In this section we give tight lower and upper bounds on the competitive ratios attainable by algorithms with natural or resource-bounded lookaheads.

Theorem 4.1 (Sleator and Tarjan [9]) *Given an on-line paging algorithm $\mathcal{A}[k]$, there exist request sequences σ, such that $\mathcal{C}_{OPT}(\sigma)$ is arbitrarily large, and,*

$$\mathcal{C}_{\mathcal{A}[k]}(\sigma) \geq \frac{\mathcal{K}+k}{k+1} \cdot \mathcal{C}_{OPT}(\sigma),$$

provided that the number of pages $\mathcal{N} \geq \mathcal{K}+k+1$.

As a simple consequence of the simulation in Lemma 3.1 and of the theorem above, we obtain the following corollary for natural lookahead. This corollary was first proved by Young [11] for resource-bounded lookahead.

Corollary 4.2 *Given an on-line paging algorithm $\mathcal{A}[k,l]$, there exist request sequences σ, such that $\mathcal{C}_{OPT}(\sigma)$ is arbitrarily large, and,*

$$\mathcal{C}_{\mathcal{A}[k,l]}(\sigma) \geq \frac{\mathcal{K}+k+l}{k+l+1} \cdot \mathcal{C}_{OPT}(\sigma),$$

provided that the number of pages $\mathcal{N} \geq \mathcal{K}+k+l+1$.

Proof: By Lemma 3.1, given any paging algorithm $\mathcal{A}[k,l]$, there exists an algorithm $\mathcal{B}[k+l]$, such that on any request sequence σ,

$$\mathcal{C}_{\mathcal{B}[k+l]}(\sigma) \leq \mathcal{C}_{\mathcal{A}[k,l]}(\sigma).$$

By Theorem 4.1, given any algorithm $\mathcal{B}[k+l]$, there exist request sequences σ on $\mathcal{N} \geq \mathcal{K}+k+l+1$ pages, with $\mathcal{C}_{OPT}(\sigma)$ arbitrarily large, such that,

$$\frac{\mathcal{K}+k+l}{k+l+1} \cdot \mathcal{C}_{OPT}(\sigma) \leq \mathcal{C}_{\mathcal{B}[k+l]}(\sigma) \leq \mathcal{C}_{\mathcal{A}[k,l]}(\sigma). \quad \square$$

We prove next that algorithm $LRU[k,l]$ with natural or resource-bounded lookahead has competitive ratio which is the best possible by the corollary above.

Theorem 4.3 *For any request sequence σ,*

$$\mathcal{C}_{LRU[k,l]}(\sigma) \leq \frac{\mathcal{K}+k+l}{k+l+1} \cdot \mathcal{C}_{OPT}(\sigma) + \mathcal{K}.$$

Proof: The proof given next is identical for the natural and the resource-bounded lookaheads. Partition the request sequence σ into phases $t_i\ \hat{t}_i$, $\sigma = t_1\ \hat{t}_1\ t_2\ \hat{t}_2\ \cdots\ t_m\ \hat{t}_m$, such that for $i = 2, \ldots, m$, t_i contains precisely $\mathcal{K}+k-1$ distinct requests that are different from the last request in $t_{i-1}\ \hat{t}_{i-1}$ and algorithm $LRU[k,l]$ makes exactly $l+1$ page faults on requests in $\hat{t}_i$. In the first phase $t_1\ \hat{t}_1$, if t_1 contains fewer than $\mathcal{K}+k$ distinct page requests, then $\hat{t}_1$ is empty, and otherwise, if t_1 contains $\mathcal{K}+k$ distincts request, then $LRU[k,l]$ faults at most $l+1$ times on requests in $\hat{t}_1$. The crucial property of this partition is that when $LRU[k,l]$ starts handling the requests in $\hat{t}_i$, it has in its lookahead queue all the requests in $\hat{t}_i$ on which it will fault.

The partition can be easily constructed by grouping the requests from the end of the request sequence towards its start repeating the following: the last part of the request sequence on which $LRU[k,l]$ faults $l+1$ times is $\hat{t}$; the preceding part of the request sequence that contains requests to $\mathcal{K}+k-1$ distinct pages is t; t is extended to include all the preceding requests to pages already in t, ensuring that the preceding request is not in t; t and $\hat{t}$ are removed from the end of the request sequence. The partitioning procedure above terminates when the construction of $t\ \hat{t}$ can not be completed since the beginning of the request sequence has been reached. Then, the remaining part of the request sequence is partitioned into t_1 that contains the first requests to $\mathcal{K}+k$ distinct pages (or fewer if there are not enough requests) and $\hat{t}_1$ that contains the remaining requests on which $LRU[k,l]$ makes no more than $l+1$ page faults.

Consider some phase $t_i\ \hat{t}_i$, for $i=2,\ldots,m$, and let p_i be the last page requested in the previous phase $t_{i-1}\ \hat{t}_{i-1}$. Then, t_i contains $\mathcal{K}+k-1$ distinct page requests that are different from p_i and $LRU[k,l]$ faults $l+1$ times on the requests in $\hat{t}_i$. By Lemma 2.1, $LRU[k,l]$ makes at most $\mathcal{K}+k-1$ page faults on requests in t_i and at most $\mathcal{K}+k+l$ page faults in the phase $t_i\ \hat{t}_i$.

Assume that while handling the requests in t_i algorithm OPT avoids making page faults on the first requests in $p_i\ t_i$ for s_i distinct pages, and let $\hat{s}_i$ denote the number of pages that were not requested in $p_i\ t_i$, but are still in OPT's fast memory when it is about to start handling the requests in $\hat{t}_i$. Then, algorithm OPT must have kept these $s_i+\hat{s}_i$ pages in fast memory since the previous phase. Since OPT has only $\mathcal{K}$ fast memory pages and one of these pages was used for p_i, we have that $s_i+\hat{s}_i\leq\mathcal{K}-1$. Hence, OPT must make at least $\mathcal{K}+k-s_i-1\geq k+\hat{s}_i$ page faults while handling the requests in t_i. We shall prove next that OPT makes at least $l-\hat{s}_i+1$ pages faults while handling the requests in $\hat{t}_i$, establishing that it makes at least $k+l+1$ page faults in the phase $t_i\ \hat{t}_i$.

Let C be the set of $\mathcal{K}+k$ pages that $LRU[k,l]$ has in its fast memory when about to start handling the requests in $\hat{t}_i$ and recall that at this point $LRU[k,l]$ has in its lookahead queue all the requests in $\hat{t}_i$ on which it will fault. Hence, by definition, $LRU[k,l]$ makes exactly $\mathcal{C}_{OPT^{C}_{\mathcal{K}+k}}(\hat{t}_i)$ page faults while handling the requests in $\hat{t}_i$. Let D be the set of $\mathcal{K}+k$ pages that are requested in $p_i\ t_i$ and let E be the set of $\mathcal{K}$ pages in OPT's fast memory when it is about to start handling the requests in $\hat{t}_i$. Then, clearly $|E_{\hat{t}_i}\setminus D|\leq\hat{s}_i$ and by the definition of algorithm $LRU[k,l]$, $C\leq^{0}_{\hat{t}_i} D$. Therefore, by Lemma 2.2,

$$l+1=\mathcal{C}_{OPT^{C}_{\mathcal{K}+k}}(\hat{t}_i)\leq\mathcal{C}_{OPT^{D}_{\mathcal{K}+k}}(\hat{t}_i)\leq\mathcal{C}_{OPT^{E}_{\mathcal{K}}}(\hat{t}_i)+\hat{s}_i,$$

and by definition, algorithm OPT makes $\mathcal{C}_{OPT^{E}_{\mathcal{K}}}(\hat{t}_i)\geq l-\hat{s}_i+1$ page faults while handling the requests in $\hat{t}_i$.

Similar analysis can be applied to the first phase, proving that $LRU[k,l]$ makes at most $\mathcal{K}$ more page faults than OPT while handling the requests in $t_1\ \hat{t}_1$. Putting this together with the fact that on each subsequent phase the ratio between the number of page faults made by $LRU[k,l]$ and OPT is at most $\frac{\mathcal{K}+k+l}{k+l+1}$, we get the desired bound. □

Remark. Observe that extra pages are strictly stronger than any lookahead since the number of page faults made by an optimal off-line algorithm with $\mathcal{K}$ pages can be arbitrarily large on request sequences that include $\mathcal{K}+1$ distinct pages, but there exist on-line algorithms with $\mathcal{K}+1$ pages that make at most $\mathcal{K}+1$ page faults on any such request sequence. If the number of pages $\mathcal{N}\leq\mathcal{K}+k+l$, then there exist on-line algorithms with natural lookahead which are as good as the optimal off-line algorithm since the lookahead queue contains the whole request sequence.

5 Conclusion

We have introduced the natural lookahead and given tight bounds on the competitive performance of paging algorithms with natural and resource-bounded lookaheads. Unfortunately, the practical value of lookahead, whether it is Young's resource-bounded lookahead, Albers' strong lookahead or the natural lookahead, seems to be very small, since all three lookahead measures require the prediction of an arbitrarily long sequence of future request. On the other hand, in practice, it might be extremly easy to add extra fast memory pages. Our results establish the tight relation between these two methods for improving the performance of on-line paging algorithms with resource-bounded and natural lookaheads. The precise relation between extra pages and lookahead in randomized algorithms, and between extra pages and Albers' strong lookahead, remains to be explored.

6 Acknowledgments

We thank Susanne Albers and Adi Rosen for discussions and comments.

References

[1] S. Albers. The Influence of Lookahead in Competitive Paging Algorithms. In *Proc. 1st European Symposium on Algorithms*, number 726 in Lecture Notes in Computer Science, pages 1–12. Springer-Verlag, Berlin, Germany, 1993.

[2] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] S. Ben-David and A. Borodin. A New Measure for the Study of On-Line Algorithms. *Algorithmica*, 11:73–91, 1994.

[4] A. Fiat, R. Karp, M. Luby, L. McGeoch, D.D. Sleator, and N.E. Young. Competitive Paging Algorithms. *J. Algorithms*, 12:685–699, 1991.

[5] E.F. Grove. Online Bin Packing with Lookahead. In *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms*, pages 430–436, 1995.

[6] E. Koutsoupias. *On-Line Algorithms and the k-server Conjecture.* PhD thesis, Dept. of Computer Science and Engineering, University of California, San Diego, 1994.

[7] E. Koutsoupias and C.H. Papadimitriou. Beyond Competitive Analysis. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pages 394–400, 1994.

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

[9] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Comm. of the ACM*, 28(2):202–208, 1985.

[10] E. Torng. A Unified Analysis of Paging and Caching. In *Proc. 36th IEEE Symp. on Foundations of Computer Science*, 1995. To appear.

[11] N. Young. *Competitive Paging and Dual-Guided On-Line Weighted Caching and Matching Algorithms.* PhD thesis, Dept. of Computer Science, Princeton University, 1991.

[12] N. Young. On-Line Caching as Cache Size Varies. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 241–250, 1991.

Recent Publications in the BRICS Report Series

RS-95-50 Dany Breslauer. *On Competitive On-Line Paging with Lookahead*. September 1995. 12 pp.

RS-95-49 Mayer Goldberg. *Solving Equations in the λ-Calculus using Syntactic Encapsulation*. September 1995. 13 pp.

RS-95-48 Devdatt P. Dubhashi. *Simple Proofs of Occupancy Tail Bounds*. September 1995. 7 pp.

RS-95-47 Dany Breslauer. *The Suffix Tree of a Tree and Minimizing Sequential Transducers*. September 1995. 15 pp.

RS-95-46 Dany Breslauer, Livio Colussi, and Laura Toniolo. *On the Comparison Complexity of the String Prefix-Matching Problem*. August 1995. 39 pp. Appears in Leeuwen, editor, *Algorithms - ESA '94: Second Annual European Symposium proceedings*, LNCS 855, 1994, pages 483–494.

RS-95-45 Gudmund Skovbjerg Frandsen and Sven Skyum. *Dynamic Maintenance of Majority Information in Constant Time per Update*. August 1995. 9 pp.

RS-95-44 Bruno Courcelle and Igor Walukiewicz. *Monadic Second-Order Logic, Graphs and Unfoldings of Transition Systems*. August 1995. 39 pp. To be presented at CSL '95.

RS-95-43 Noam Nisan and Avi Wigderson. *Lower Bounds on Arithmetic Circuits via Partial Derivatives (Preliminary Version)*. August 1995. 17 pp. To appear in *36th Annual Conference on Foundations of Computer Science*, FOCS '95, IEEE, 1995.

RS-95-42 Mayer Goldberg. *An Adequate Left-Associated Binary Numeral System in the λ-Calculus*. August 1995. 16 pp.

RS-95-41 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. *Eta-Expansion Does The Trick*. August 1995. 23 pp.

RS-95-40 Anna Ingólfsdóttir and Andrea Schalk. *A Fully Abstract Denotational Model for Observational Congruence*. August 1995. 29 pp.