Real-time integrated prefetching and caching

Peter Sanders* Johannes Singler* Rob van Stee†

September 26, 2012

Abstract

The high latencies for access to background memory like hard disks or flash memory can be reduced by caching or hidden by prefetching. We consider the problem of scheduling the resulting I/Os when the available fast cache memory is limited and when we have real-time constraints where for each requested data block we are given a time interval during which this block needs to be in main memory. We give a near linear time algorithm for this problem which produces a feasible schedule whenever one exists. Another algorithm additionally minimizes I/Os and still runs in polynomial-time.

For the online variant of the problem, we give a competitive algorithm that uses lookahead and augmented disk speed. We show a tight relationship between the amount of lookahead and the speed required to get a competitive algorithm.

1 Introduction

A classical technique for dealing with memory hierarchies is prefetching and caching. Prefetching hides access latencies by loading pages into fast memory before they are actually required [7, 10, 13, 15]. Caching avoids I/Os by holding pages that are needed again later [3, 5, 8, 14]. Since both techniques compete for the same memory resources, it makes sense to look at the integrated problem [2, 6, 11, 9, 12]. Interestingly, there is very little theoretical work on the real-time setting of this problem. We are only aware of [7] which covers parallel disk prefetching in a read-once setting without caching. This is astonishing, since real-time properties are essential for more and more important applications such as games, virtual reality, graphics animations, or multimedia. Memory hierarchies get more and more important for these applications since larger and larger data sets are considered and since mobile devices have only very limited fast memory whereas the bulk of their memory is flash memory that is accessed in pages.

As a concrete (simplified) example, consider a flight simulator. Externally stored objects could be topographical data, textures, etc. At any particular time, a certain set of objects is required in order to play out the right screen content and sound without delays. A demo run could be preplanned resulting in an offline version of the problem. User interactions will result in an online problem where we do not completely know the future. However, we may be able to compute a *lookahead* containing a superset of the required objects for a certain amount of time.

In Section 2, we propose to model real time aspects by associating a time window with each request during which it needs to be in cache. As before we do prefetching and caching, and as soon as a fetch for a page starts, this page occupies one slot in the cache. Time windows are a flexible abstraction of

*Karlsruhe Institute of Technology, D-76128 Karlsruhe, Germany. {sanders,singler}@kit.edu.

†Max Planck Institute for Computer Science, Saarbrücken, Germany. vanstee@mpi-inf.mpg.de. Research supported by German Research Foundation (DFG). Work performed while the author was at the Karlsruhe Institute of Technology, Germany.

several effects occurring in practice. They efficiently model the amount of time a data block is needed for processing and it is also possible to require several blocks to be available concurrently. The only simpler model we could think of would use unit time windows. But then we would need many repeated requests to model longer time windows. This would lead to exponentially longer problem descriptions in the worst case.

We first prove some generally useful properties of the problem in Section 3. In Section 4 we then introduce algorithm EAGER-LFD that is closely modeled after Belady's algorithm [3] for offline caching. This algorithm runs in linear time and finds a feasible schedule whenever one exists. However, surprisingly and in contrast to the non-real-time situation, EAGER-LFD is not *I/O-optimal*, that is, it may perform many more fetches than necessary. In Section 5 we solve this problem using a different "semi"-greedy approach (LAZY-LFD). LAZY-LFD uses the frequently used basic trick to build the schedule backward in time [12, 9]. Apart from this, however, it is a new algorithm. Its main invariant is that it uses as little space as possible at all times and in order to achieve that, it has to move previously scheduled requests. The algorithm therefore has quadratic worst case performance.

We additionally consider an online version of our problem in Section 6. In our online model, algorithms have a certain amount of lookahead, input arrives incrementally, and a partial solution needs to be determined without knowledge of the remaining input. We say that an algorithm has a *lookahead* of ℓ if at time t, it can see all requests for pages that are required no later than at time $t + \ell$. In the *resource augmentation* model, the online algorithm has more resources than the offline algorithm that it is compared to. There are several ways to give an online algorithm more resources in the current problem: it can receive a larger cache, a faster disk (so that fetches are performed faster), more lookahead, or a combination of these.

We show that competitive algorithms are possible using resource augmentation on the speed *and* lookahead, and we provide a tight relationship between the amount of resource augmentation on the speed and the amount of lookahead required.

Section 7 concludes with a short summary and some possible future questions.

More Related Work

In the model introduced by Cao et al. [6] and further studied by Kimbrel and Karlin [12] and Albers et al. [2], the requests are given as a simple sequence without an explicit notion of time. It is assumed that serving a request to a page residing in cache takes one time unit, and fetching a page from disk takes F time units. When a fetch starts and the cache is full, a page must be evicted. If a page is not in cache when it is required, the processor must wait (stall) until the page has been completely fetched. The goal is to minimize the processor stall time.

Thus in this model, pages have implicit deadlines in the sense that each page should be in cache exactly one time unit after the previous request. However, when the processor incurs stall time, these implicit deadlines are shifted by the amount of stall time incurred. Additionally, this model does not cover the cases where many pages are required in a small time interval and conversely, where more time may elapse between two successive requests. Nor is it possible to model the case where a page is required over a certain time interval.

Albers [1] considers the impact of lookahead in the classical non-real-time situation. She shows that in order to be useful in a worst case sense, lookahead has to be measured in terms of the number of distinct pages referred to in the lookahead. This can be a problem in practice since very long lookahead sequences might be required if some blocks are accessed again and again. In our real-time setting the situation is different and very natural — we can measure lookahead in terms of time.

We therefore believe that, while these previous results are interesting in their own right, our new model also has practical merit.

2 Problem Definition

We consider the problem of prefetching pages into a cache of fixed size k. The request sequence σ serves as input and consists of pairs $\sigma_i := (p_i, [d_i, e_i))$ denoting a page and the interval in which it must reside in the cache. The page ids p_i are not necessarily different. The d_i are the *deadlines*, the e_i are the *evict times*. At time e_i, page p_i may (but does not have to) be evicted. Without loss of generality we may assume the input is sorted such that $d_1 \leq \ldots \leq d_n$ for $n = |\sigma|$. Each *fetch* (transferring a page to the cache) takes time 1. The earliest possible fetch time is $t = 0$.

The output is given by a sequence $f_1, \ldots, f_n$ of fetch times for the corresponding requests and implicitly defines a *schedule*. One cache slot is occupied by p_i in the time interval $[f_i, e_i)$, and possibly longer. Multiple requests of the same page can be served by the same fetch if there is enough room in the cache. That is, for any given page p there may be several index sets I in a schedule such that $p_i = p$ and $f_i = f$ for all $i \in I$ and some value f. In such a case, the page p resides in the cache from time f until time $\max_{i \in I} e_i$.

For a given schedule, call a request σ_i *primary* if there is no previous request which also uses fetch time f_i. Non-primary requests are called *free*. A feasible schedule must satisfy $\forall i \in \{1, \ldots, n\} : f_i + 1 \leq d_i$ to match the real-time requirements. The fetch time sequence also implicitly defines a *cache content* which is $\{p_i : f_i \leq t < e_i\}$ at time t. A feasible schedule must have a cache content of size at most k at all times. Finally, no more than one page can be fetched at any one time — $\forall i, j : |f_i - f_j| \geq 1 \vee p_i = p_j$.

In our model, we have no additional assumptions on the tasks, e.g. regarding periodicity, or dependencies between tasks (pages). Most of this paper is concerned with the case of a single disk (implying that exactly one page can be downloaded in one time unit), though in Section 6 we also explore the case of parallel disks.

3 Problem Properties

To simplify the analysis, we first prove that there exist I/O-optimal schedules (i.e., with the optimal number of fetches) with nice structural properties.

Definition 1 ***(FIFO property)*** *A schedule satisfies the* FIFO Property *if its fetch times for primary requests form an increasing sequence.*

Definition 2 ***(BUSY property)*** *A schedule satisfies the* BUSY Property *if, when it has a fetch at time f for a page that is next required at time d, it defines fetches at all times $f + i$ for $i = 1, 2, \ldots, \lceil d - f \rceil - 1$.*

Definition 3 ***(LFD property)*** *A schedule satisfies the* LFD Property *if, whenever a page is evicted, this page is one that is needed again the latest.*

Lemma 1 *Any feasible schedule for request sequence σ can be transformed to satisfy the FIFO, BUSY, and LFD properties without increasing the number of fetches.*

Proof: Consider an arbitrary feasible schedule. We can make three local updates:

1. If we have a violation of the FIFO-property with primary requests σ_i and σ_j with $i < j$ yet $f_i > f_j$, we have the following situation: $d_i \leq d_j$ by definition of the input, $f_i \leq d_i - 1$ by definition of a feasible schedule. Overall, $f_j < f_i \leq d_i - 1 \leq d_j - 1$. We now swap f_i and f_j (for any later free request with $f_k \in \{f_i, f_j\}$, we update f_k accordingly). The schedule remains feasible – all deadlines are met including those of free accesses to p_i or p_j which all have deadlines $\geq d_i$. The disk is busy exactly the same times as before – only accessing different pages. Similarly, the cache occupancy remains the same at all times.

2. If a fetch for page p ends at time t_1, but page p is first requested at time $t_2 > t_1$, and moreover no new fetch starts until time $t_3 > t_1$, we can move this fetch for p forward until it ends at time $\min(t_2, t_3)$.

3. If we have a violation of the LFD-property with pages p_i and p_j with $e_i < e_j$ yet at time e_i page i is needed again before page j, then we just swap these evictions. The cache occupancy remains the same and the contents of the cache outside the interval $[e_i, e_j]$ remain unchanged. If page j is requested at some point $t \in [e_i, e_j]$, then by assumption page i is requested in the interval (e_i, t), so must also be fetched again in that interval since it was evicted at time e_i. We simply replace this fetch by a fetch of page j in the new schedule.

It now follows that if we cannot make any local improvement to a feasible schedule, it satisfies all three properties. □

4 Algorithm EAGER-LFD

In this section we present an algorithm for the real-time integrated prefetching and caching problem, called EAGER-LFD. It is derived from the Longest Forward Distance (LFD) algorithm known for the non-real-time problem [4, page 35], i.e., whenever it evicts a page, it chooses one that is needed again the latest.

For this algorithm, we consider only the case that all deadlines and evict times are integers. In return, EAGER-LFD will output integer fetch times. This is almost without loss of generality: If the integer condition is not fulfilled, all times fetch times and deadlines can be rounded down and all evict times can be rounded up. To maintain feasibility in this case, one additional cache slot is needed to hold the page that is currently being fetched. Thus the integrality condition can be relaxed at the cost of requiring a very small resource augmentation in the cache size.

EAGER-LFD (refer to Algorithm 1 for pseudo-code) simulates the development of cache content over time. It maintains a set $\mathcal{C}$ of cache slots s for which it stores the page $\mathsf{page}(s)$ associated with the slot, the time $\mathsf{time}(s)$ this page is fetched, and the time $\mathsf{evict}(s)$ when it can be evicted again. When request $(p_i, [d_i, e_i))$ is processed, the easy case is when p_i is already stored in some cache slot s. In this case, no I/O is needed and we can simply set f_i to $\mathsf{fetch}(s)$ and reserve the slot for request i by extending $\mathsf{evict}(s)$ to cover e_i. If p_i is not cached, it is fetched as early as possible, i.e., immediately if there are pages in cache that can be evicted or at the first point in time a page becomes evictable. If this time is too late to finish fetching p_i before its deadline d_i, the instance is rejected as infeasible. Among the pages p eligible for eviction, the one with the longest forward distance is chosen, i.e., the one maximizing the next deadline $\min\{d_j : j > i \wedge p_j = p\}$.

Clearly, if Algorithm EAGER-LFD does not fail explicitly (in line 12), it will produce a feasible schedule. Moreover, it produces a schedule with the FIFO property since it schedules primary requests in the same order as the deadlines. However, we have to prove that a feasible schedule is actually found whenever possible.

Algorithm 1 EAGER-LFD

```
1: $t := 0$                                                    \\everything is planned until time $t$
2: fill cache $\mathcal{C}$ with empty slots $s$ all having $\mathsf{page}(s) = \bot$, $\mathsf{fetch}(s) = \mathsf{evict}(s) = 0, \mathsf{next}(s) = \infty$
3: for $i := 1$ to $n$ do                                       \\build schedule incrementally
4:   if $\exists s \in \mathcal{C} : \mathsf{page}(s) = p_i$ then                \\$p_i$ is in cache
5:     $f_i := \mathsf{fetch}(s)$
6:     $\mathsf{evict}(s) := \max(\mathsf{evict}(s), e_i)$                  \\reserve $s$ for request $i$.
7:     $\mathsf{next}(s) :=$ deadline of next request to $p_i$ or $\infty$ if no next request
8:   else
9:     $t := \max(t, \min\{\mathsf{evict}(s) : s \in \mathcal{C}\})$           \\earliest possible fetch time
10:    if $d_i < t + 1$ then                                    \\deadline $d_i$ will be missed
11:      fail
12:    choose the slot $s \in \mathcal{C}$ with $\mathsf{evict}(s) \le t$ which maximizes $\mathsf{next}(s)$   \\LFD rule
13:    $\mathsf{page}(s) := p_i$; $\mathsf{evict}(s) := e_i$; $f_i := \mathsf{fetch}(s) := t$
14:    $t := t + 1$
```

Theorem 1 ***(*EAGER-LFD *is feasible)*** *Algorithm* EAGER-LFD *is a real-time offline prefetching algorithm which computes a feasible schedule if there exists one.*

Proof: Consider any request sequence σ and any feasible schedule S for σ (specified by fetch times $f_1, \dots, f_n$ as discussed in Section 2). Wlog we can assume that the schedule has the FIFO property (see Lemma 1). We show that algorithm EAGER-LFD computes a feasible schedule S_{lfd} in this case. We prove that S can be transformed to S_{lfd}. The proof is inductive showing that a feasible FIFO schedule S_i using the same fetch times as S_{lfd} for steps $1, \dots, i$ and having the same cache content at time $\max_{j \le i} f_j$. can be transformed into a feasible FIFO schedule S_{i+1} using also the same fetch time and cache content for step $i+1$. In the following, when we change a fetch time t for a primary request to t', we also change the free requests that use fetch time t to t'.

Case $i = 1$: For S_1 we simply set $f_1 := 0$. Due to the FIFO property this is simply the old value of f_1 or all other fetch times are ≥ 2 so that the schedule remains feasible.

Case σ_{i+1} is free in S_i: By the induction hypothesis EAGER-LFD also has page p_{i+1} in cache and hence σ_{i+1} will also be a free request for it. Let σ_j, $j \le i$ denote the last primary request of page p_{i+1} in S_i. By the induction hypothesis, f_j has the same value as in the schedule produced by EAGER-LFD. Hence, $f_{i+1} = f_j$ already has the right value for the induction step.

Case σ_{i+1} is primary in S_i: Let f' denote the fetch time EAGER-LFD would use for f_{i+1}.

Case $f' = f_{i+1}$: There is nothing to show.

Case $f' > f_{i+1}$: EAGER-LFD selects the earliest possible time at which a page can be evicted as its next fetch time f'. Hence, $f' > f_{i+1}$ implies that S_i evicts a page p_j with $j \le i$ which is not evictable for EAGER-LFD at time f_{i+1}, i.e., $f_{i+1} < e_j$. But this is impossible since S_i behaves like EAGER-LFD for requests $\sigma_1, \dots, \sigma_i$.

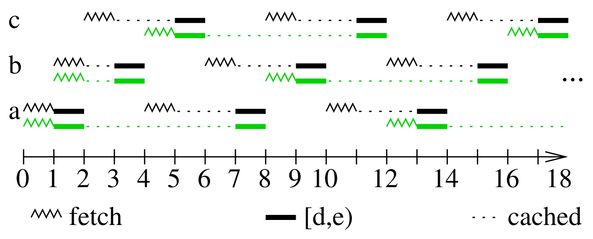

Figure 1: A periodic request sequence where EAGER-LFD (top) needs asymptotically twice the number of I/Os as the optimal schedule (below).

Case $f' < f_{i+1}$: In this case, since S_i is a FIFO schedule, S_i leaves the disk idle between $\max_{j \le i} f_j + 1$ and f_{i+1}, in particular at time f'. However, EAGER-LFD may evict a page p_j (with $j > i+1$) from the cache which is still needed later. In this case, we set $(f_{i+1}, f_j) := (f', f_{i+1})$, again obtaining a feasible schedule with the same cache content after f_{i+1}. Note that this may introduce a new primary fetch for σ_j. If the page evicted by EAGER-LFD is not needed later, we only make the modification $f_{i+1} := f'$. □

Theorem 2 *Algorithm* EAGER-LFD *needs at most twice the number of I/Os than an optimal algorithm. On the other hand, for every $i \ge 1$ there are request sequences with $|\sigma| = 2i$ where* EAGER-LFD *needs $2i$ I/Os whereas an optimal algorithm needs only $i+1$ I/Os.*

Proof: The first part of the theorem can be seen from the proof of Theorem 1 which shows how to convert an optimal schedule to an LFD schedule introducing at most one additional fetch for each primary request of the optimal schedule.

For the second part, consider $k = 2$ and any even size prefix of the sequence $\sigma_0\sigma_1\sigma_2\cdots$ with $\sigma_j = \langle(a, [6j+1, 6j+2)), (b, [6j+3, 6j+4)), (c, [6j+5, 6j+6))\rangle$ which periodically accesses the three pages a, b, and c – one every other step. EAGER-LFD will compute the fetch time sequence $\langle 0, 1\rangle L_1 \cdots L_{i-1}$ with $L_j = \langle 2i, 2i+2\rangle$ – using one fetch for each request. However, the sequence $\langle 0, 1\rangle F_1 \cdots F_{i-1}$ with $F_i = 4i, 4i-4$ is a feasible schedule that needs only $i+1$ I/Os. Figure 1 illustrates this situation. □

A major advantage of EAGER-LFD is that it can be implemented to run very efficiently. The exact running time depends on the machine model and certain assumptions on the representation of the input. In particular, we obtain deterministic linear time, if the page ID's, deadlines, and evict times are integers that are polynomial in $n = |\sigma|$:

In a preprocessing step we calculate for each request σ_i the deadline of the next request for the same page. This can either be done by stably sorting σ by page id, or, for general page IDs, using a hash table of size $O(n)$. Using radix sort, this is possible in linear time. Using this sort (or hash) operation, we can also rename the page ids to use integers in the range $1..n$. This allows us during the main loop to keep a lookup table telling us which pages are in cache.

A (minimum) priority queue stores the evict times of the cache slots while a (maximum) priority queue stores the next deadlines of the evictable slots. The min-priority queue supports the minimum operation in Line 9 and the max-priority queue supports the maximum operation in Line 12. Both priority queues have integer keys and are monotone, i.e., their minimum/maximum is monotonically increasing/decreasing. This allows a simple and efficient implementation. We can actually get the priority queue operations down to amortized constant time by not using the actual times but only their rank in a sorted order of evict times / deadlines as keys for the priority queues. This way, simple bucket queues yield the desired result. The required ranks can be determined by another preprocessing step using radix sort.

Without any assumptions on the inputs we get $O(n \log n)$ execution time using sorting or expected time $O(n \log n)$ using hashing. If σ does not fit into internal memory, we can nevertheless obtain an optimal schedule since the preprocessing operations can be done using I/O-efficient sorting and the main loop – using hashing – needs only space $O(k)$. Note that all these optimizations can also be used for the non-realtime LFD algorithm.

5 Algorithm LAZY-LFD

In this section we present the algorithm LAZY-LFD for the real-time integrated prefetching and caching problem. This algorithm works by working backwards from the last deadline. For each new request, it modifies the existing schedule to maintain I/O-optimality. While the schedule is being constructed, our algorithm keeps track of a value t which is the time at which the earliest fetch of the current schedule starts.

Fetches are defined as late as possible, that is, just before the corresponding deadline or directly before the next fetch. For each new request (starting from the end), we check whether there is a later fetch for the same page which can be removed now. I.e., we check whether there is room in the cache to keep this page loaded between the evict time of the current request and the fetch time of the next request. This check is done in line 14 (after calculating the values next and full in line 5 and 10–13, respectively). We prove that these conditions are correct in Section 5.1; in particular, Lemma 4 is crucial for the correctness proof. Although it is perhaps not immediately obvious, as the name suggests LAZY-LFD also creates a schedule with the LFD property, as we will show in Lemma 2.

LAZY-LFD uses a subroutine which is called CreateFetch(i,t,DoEvict) and is defined in Algorithm 2.

Algorithm 2 CreateFetch(i, t, DoEvict)

```
1: if there is a slot $s \in \mathcal{C}$ which is free at time $t$ then
2:     $f_i := t$
3:     $\mathsf{fetch}(s) := t$
4:     if DoEvict then
5:         $\mathsf{evict}(s) := e_i$
6: if $t < 0$ or there exists a time at which there are at least $k + 1$ pages in cache then
7:     Output FAIL
8: Set $t := t - 1$
```

The algorithm itself is defined in Algorithm 3. It uses the following definitions. We also define a concept called *slack* which will be important later.

Definition 4 *The cache is* full *if there are no empty slots in the cache (recall that a page occupies a slot as soon as it starts being fetched) and moreover all pages in the cache are still needed at some point in the future. Pages that are evicted at time t are not taken into consideration to determine whether the cache is full or not at time t.*

Definition 5 *A* tight fetch *is a fetch which starts one time unit before the page is requested, i. e., at the latest possible time.*

Definition 6 *The* slack *of a fetch is the amount of time between the end of the fetch and the corresponding deadline.*

Algorithm 3 LAZY-LFD

```
1: Set $t := \infty$
2: for $i := n$ downto $1$ do
3:     if $t > d_i - 1$ then
4:         $t := d_i - 1$
5:     if $\exists j > i : p_j = p_i$ then
6:         next $:= \min\{f_j | p_i = p_j, j > i\}$
7:     else
8:         next $:= \infty$
9:     if next $= \infty$ then
10:        CreateFetch$(i, t,$ true$)$
11:    else
12:        if next $> d_i - 1$ then
13:            if $e_i \geq$ next or the cache is never full in $[e_i,$ next$]$ then
14:                full $:= \infty$
15:            else
16:                full $:=$ the first time the cache is full in $[e_i,$ next$]$.
17:            if full$\leq$next and there is a tight fetch which finishes in the interval [full, next] then
18:                CreateFetch$(i, t,$ true$)$
19:            else
20:                $f_j := -1 \, \forall j : f_j =$next            \\Remove the fetch of page $p_i$ which starts at time next
21:                $s :=$next$-1$
22:                while $\exists j : t < f_j \leq s$ do
23:                    $j := \arg\max\{f_j | t < f_j \leq s\}$            \\find last fetch in interval $(t, s]$
24:                    $s := f_j - 1$            \\make interval smaller
25:                    $m := \max(\min\{f_k | f_k > f_j\}, \min\{d_k | p_k = p_j, k > i\})$-1            \\latest possible fetch time
26:                    $f_k := m \, \forall k : f_k = f_j$
27:                $t :=$min$(\min\{f_j | f_j > 0, j > i\}, d_i) - 1$.
28:                CreateFetch(i,t,false)
29:                $f_j := t \, \forall j : f_j = -1$            \\Set fetch times that were set to -1 in line 20 correctly
```

Request	Page	d_i	f_i
1	a	2	0
2	e	2	1
3	d	3	2
4	c	5	3
5	b	5	4
6	a	5	-

Table 1: An example of the schedule produced by our algorithms for an input with $k = 3$. (Both algorithms produce the same schedule if ties are broken in the same way.) The first two columns contain the input. (For each request i, $e_i = d_i + 1$.) The third and fourth column lists the times at which these algorithms start to fetch these pages. A minus sign means that the page is not fetched (is already in the cache).

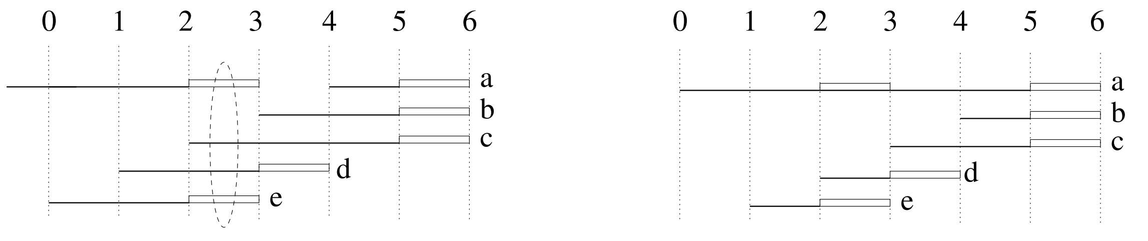

Figure 2: Suppose $k = 3$, and consider the input from Table 1. LAZY-LFD treats the requests in order of non-increasing deadline, starting from the end of the input sequence. Horizontal lines represent a cache location which is occupied by the page at the interval indicated by the line. The small rectangles indicate when each page is required to be in the cache. When LAZY-LFD gets to the request for page a at time 2, without the optimization in lines 20–26 it would create the infeasible schedule on the left: there are four pages in cache in the interval $[2, 3]$. However, LAZY-LFD removes the fetch for page a at time 4, the other fetches move forward by 1 time unit, and the resulting schedule is feasible.

Note that the condition in line 9 of Algorithm 3 need not be satisfied even though deadlines are sorted: if a fetch for a page p_i is defined a long time before d_i, p_i might be requested again between this fetch and time d_i. We give an example of the execution of LAZY-LFD and EAGER-LFD for $k = 2$ in Table 1.

5.1 Analysis of LAZY-LFD

For convenience, we consider fetches to take place during half-open intervals of the form $(f, f + 1]$.

Lemma 2 *The schedule of* LAZY-LFD *has the FIFO, BUSY, and LFD properties throughout the execution of* LAZY-LFD.

Proof: The FIFO property holds because requests are treated in order of non-increasing deadlines and fetches, once defined, are never reordered by LAZY-LFD.

We prove that the BUSY property is maintained by backwards induction, starting with the last request. For the last request, there is nothing to prove: if its deadline is at time d, its fetch is scheduled at time $d - 1$, at least initially (this fetch could be removed later).

By induction, the BUSY property holds before request i is handled. If next $\leq d_i - 1$ in line 9, we do not change the schedule and are done by induction. This is also true if LAZY-LFD defines the new fetch one time unit before the earliest current fetch (by induction) or at time $d_i - 1$, without changing the rest of the existing schedule. This happens if the fetch is defined in line 10 or 18 of Algorithm 3, according to line 7 in Algorithm 2 and line 4 of Algorithm 3. In lines 20–26, the next fetch to p is removed and all or some earlier fetches are moved forward as far as possible, including the (currently) earliest one for p. Thus the BUSY property holds again.

Finally, we consider the LFD property. Note that evictions always take place immediately after a request for a page. The LFD property holds because whenever some page p_i is evicted, all pages that are needed again later than p_i but requested before have already been evicted[1]: the fact that p_i is evicted at time e_i means that the cache is full at some later point t_i (before p_i is needed again), and this means that a page requested before time e_i which is next needed after time t_i can also not be kept in the cache. □

[1]Here we describe the final schedule of LAZY-LFD. The actual decision to evict such a page will be taken later in the algorithm because it works backwards from the end of the input.

Lemma 3 *Suppose* LAZY-LFD *starts to fetch page p_i with deadline d_i at time t. Then the number of pages in the cache of* LAZY-LFD *at time $t+j$ is at least j for $j = 1, \ldots, \min(k, d_i - t)$.*

Proof: By the FIFO property, every page j loaded in the interval $[t, d_i]$ has $e_j \geq d_j \geq d_i$. By the BUSY property, LAZY-LFD keeps loading pages in this interval. By its definition, LAZY-LFD does not evict any page p_j until after time e_j (at the earliest). This proves the lemma. □

Theorem 3 *If there exists a feasible schedule,* LAZY-LFD *computes an I/O-optimal one.*

Proof: Let ALG(i) be the number of fetches that algorithm ALG defines to serve the sequence I_i consisting only of the requests $i, \ldots, n$. Denote an optimal algorithm by OPT, and without loss of generality assume it satisfies the FIFO and BUSY properties. We use a proof by induction.

Hypothesis: The schedule of LAZY-LFD is I/O-optimal for *any* input consisting of at most i requests that allows a feasible schedule, and there is no feasible schedule that fetches its first page later.

Base case: Consider an input consisting of a single request. LAZY-LFD defines a single fetch for it, at the last possible time. Thus, LAZY-LFD is I/O-optimal, and no schedule can start its first fetch later than LAZY-LFD.

Induction step: By the induction hypothesis, LAZY-LFD is optimal for I_{i+1} (We assume that a feasible schedule for the input exists, so it certainly exists for any subset of the input). Consider request i, for page p_i. We abbreviate p_i by p in this proof. We need to check the following properties for each fetch we create.

I/O-OPT. LAZY-LFD does not define more than the optimal number of fetches

LAZY. The first fetch cannot start later in any feasible schedule that satisfies the FIFO and BUSY property

FEASIBLE. The schedule created by LAZY-LFD is feasible

MIN. At any time, an algorithm which is feasible and I/O-optimal cannot keep less pages in its cache than LAZY-LFD.

Last fetch of p (Line 10): next $= \infty$, so the page is not fetched again after this time (i.e., $p \notin I_{i+1}$).

I/O-OPT. LAZY-LFD defines one extra fetch and any feasible schedule requires an extra fetch compared to input I_{i+1}.

LAZY. The first fetch for input I_{i+1} cannot start later in any schedule by induction, and the new fetch either starts immediately before it, or one time unit before p is due.

FEASIBLE. Suppose the new schedule is infeasible. If this is because we have defined a fetch before time 0, there is no feasible schedule by LAZY. Suppose it is because not all pages fit in the cache at some later time t. This means that at time t' a fetch of a page starts, but after this time the k other distinct pages loaded before time t' (in particular page p_i) are still required. Since MIN holds for I_{i+1}, k pages are in the cache of the optimal algorithm at time t for the input I_{i+1} (counting the one that starts getting fetched at time t'). Since page p_i is still required after time t', no feasible schedule exists. We conclude that FEASIBLE holds if the instance allows a feasible schedule.

MIN. Outside the interval $[t, e_i]$, nothing changes. For any $t' \in [t, e_i]$, the cache now contains one more page than before, namely p. The previous number of pages was minimal (for I_{i+1}) by induction and the first fetch cannot start later in any schedule by LAZY. Since $p \notin I_{i+1}$, MIN holds for $t' \in [d_i, e_i]$.

If $t = d_i - 1$, there is one page in cache during $[t, d_i)$ (recall that we count pages that are being fetched as pages in the cache), which is clearly minimal. Else, by Lemma 3 and the fact that MIN holds for the input I_{i+1}, the number of pages in the optimal cache *for the input* I_{i+1} is at least j at time $t + j + 1$ for $j = 1, \dots, \min(k, d_{i+1} - (t+1))$. Since $d_i \leq d_{i+1}$, adding one page which is not requested later ($p \notin I_{i+1}$) adds 1 to this number for all times $t + j$, $j = 1, \dots, d_i - t$. Therefore MIN holds.

New fetch for p (Line 18): I/O-OPT. We have the following lemma, which shows that LAZY-LFD can indeed evict page p at its evict time (and load it again later) and still be I/O-optimal.

Lemma 4 *Let a fetch of page p (request i) start at time t. Suppose that p is again fetched later, and that this happens for the first time at time* next. *Suppose there is at least one tight fetch in the interval* $[e_i, \text{next}]$*, and denote the last time at which such a tight fetch finishes by* tight. *If the cache is full at some point no later than* tight*, then* $\text{OPT}(i) = \text{OPT}(i+1) + 1$.

Proof: We call the fetch which is running at time full the *current* fetch. Generally, there are three sets of pages in the cache at time full:

1. k_1 pages required during the current fetch
2. k_2 pages already requested before, still needed after the current fetch
3. k_3 pages that are needed only after time full.

Of course, $k_1 + k_2 + k_3 = k$. Let us first consider the two easiest cases.

1. $k_2 = k_3 = 0$: k different pages have deadlines within an interval of length strictly less than 1 starting at time full. By assumption, all these pages are different from p. This means that no algorithm can have p in cache at time full, so I_i forces an extra fetch compared to I_{i+1}, and $\text{OPT}(i) = \text{OPT}(i+1)+1$.

2. $k_3 = 0, k_2 > 0$: All pages in the cache are either required during the current fetch, or before (these pages were kept in cache to save a fetch on them). This means that LAZY-LFD has saved k_2 accesses to the k_2 pages that were already required before this fetch. Suppose $\text{OPT}(i)$ keeps p in its cache throughout the interval $[e_i, \text{next}]$. Then at least one of these k_2 pages must be evicted by OPT, and later loaded again. However, LAZY-LFD has the optimal number of fetches for the sequence I_{i+1}. We have that $\text{OPT}(i)$ has one fetch more than LAZY-LFD for the requests in I_{i+1}. Thus $\text{OPT}(i) = \text{LAZY-LFD}(i+1) + 1 = \text{OPT}(i+1) + 1$.

If $k_3 > 0$, we have the following situation: the cache is full at time full $\in [e_i, \text{tight}]$, but some pages are already loaded solely in order to satisfy future requests (and not because they were requested before). If there is any time $t' \in [t+1, \text{tight}]$ at which the cache is full and case 1 or 2 above holds, we are done. Otherwise, we let full be the *last* time the cache is full in (t+1,next], and define the *current fetch* accordingly. If $k_3 > 0$, some pages are loaded that are needed only later.

Suppose that in the optimal schedule, p is in the cache throughout $[e_i, \text{next}]$, so definitely at time full. Then at least one of the pages that LAZY-LFD has in the cache at time full, say q, must be missing in the optimal cache, since the cache of LAZY-LFD is full. Page q is not required to be in the cache at time full, but there is a later need for it. If LAZY-LFD is saving an access on page q since q was requested before time full as well, we are done: the optimal schedule still needs to load q, and without the request for p there exists a schedule with one less fetch for q (namely, the one of LAZY-LFD), so $\text{OPT}(i) = \text{OPT}(i+1) + 1$. Otherwise, q is loaded only to satisfy a future request. In this case, consider the time starting from full.

Consider the k_3 pages that are loaded purely to satisfy request intervals starting after time full. Say that the last time such a page is first needed (after time full) is time t_1, and denote that page by p_1. (Possibly

$p_1 = q$.) At time t_1, we can make a similar division into sets as above. If any pages are in cache at time t_1 such that the request interval for which they were loaded has not yet started, we can find a time $t_2 > t_1$ where the last such page (say p_2) is first needed. We can continue this process until some time $t_\ell =: t^*$, which is defined as the earliest possible time after full such that all pages in cache were fetched to satisfy a request *at or before time* t^*. (Some of these pages might have been kept in cache to also satisfy later requests.)

We claim that LAZY-LFD fetches pages continuously in the interval $[\mathsf{full}, t^*]$. This follows by applying the BUSY property (Lemma 2) at time full, $t_1, t_2, \ldots$ successively until time t^* is reached. Precisely, *before* time full there is a fetch for p_1 which is next required at time t_1, and before time t_j $(j = 1, \ldots, \ell - 1)$ there is a fetch for p_{j+1} which is next required at time t_{j+1}.

Consider the last fetch that starts before time t^*. By definition of t^*, this fetch cannot be for a page that is requested only after t^*. Thus this fetch must in fact be tight (no slack): if it were not tight and yet, no other fetch starts after it and before time t^*, LAZY-LFD could (and would) have scheduled this fetch later by property LAZY and induction. So

$$t^* \leq \mathsf{tight} \leq \mathsf{next}.$$

Since the cache is not full in the interval $(\mathsf{full}, t^*]$ by definition of full, LAZY-LFD never fetches the same page twice during $(\mathsf{full}, t^*]$ (pages are only evicted between two successive requests to it if the cache is full between these requests by lines 13–14 and 17). All of these pages are needed in $[\mathsf{full}, t^*]$, including q. Denote the set of pages fetched by LAZY-LFD in $[\mathsf{full}, t^*]$ by S.

We now repeatedly apply the pigeonhole principle, using the fact that LAZY-LFD is loading pages continuously during $[\mathsf{full}, t^*]$. The optimal schedule must load q at some time *after* full. So it must load at least one page $p^{(1)}$ that LAZY-LFD loads in the interval $[\mathsf{full}, t^*]$ already before this interval, since there is no time to fetch $|S| + 1$ pages. This implies $p^{(1)}$ is not required during the fetch which runs at time full, and LAZY-LFD does not have the pages $\{p, p^{(1)}\}$ in its cache at time full. Therefore LAZY-LFD has yet another page $q^{(1)}$ in cache at time full that the optimal schedule does not, since its cache is full. We now repeat this reasoning: if LAZY-LFD saves a request on $q^{(1)}$ since it was already requested before, we immediately have $\text{OPT}(i) = \text{OPT}(i+1) + 1$ (the optimal schedule must still pay for $q^{(1)}$). Otherwise, we again find that the optimal schedule must load $q^{(1)}$ in the interval $[\mathsf{full}, t^*]$, leading to yet another page $p^{(2)}$ that it must load before the interval due to time constraints. Each such page $p^{(i)}$ implies an additional distinct page $q^{(i)}$ that LAZY-LFD has in its cache at time full which the optimal schedule does not, because the pages $p, p^{(1)}, \ldots, p^{(i)}$ are all in the cache of the optimal schedule at time full and the cache of LAZY-LFD is full. (Each time we find that LAZY-LFD does not have $p^{(i)}$ in its cache, because this page is requested in the interval $[\mathsf{full}, t^*]$ and we know that LAZY-LFD loads it after time full: if $p^{(i)}$ were already in the cache at time full, LAZY-LFD would never drop it since the cache is not full afterwards.)

Finally, after at most k steps we either run out of pages and find a contradiction, or we find a page that LAZY-LFD saves a fetch on by keeping it in cache and that the optimal schedule must pay for, implying that

$$\text{OPT}(i) = \text{LAZY-LFD}(i+1) + 1 = \text{OPT}(i+1) + 1.$$

This concludes the proof of Lemma 4. □

Lemma 4 immediately implies that I/O-OPT holds if LAZY-LFD creates a fetch in line 18.

LAZY. Since the optimal solution requires an additional fetch compared to the input $i + 1, \ldots, n$, and the first fetch could not start later for that input, LAZY holds.

FEASIBLE. The new schedule clearly does not violate deadlines. Thus if the new schedule is infeasible, no feasible schedule can exist by induction: either the cache capacity constraint is violated in all possible schedules, but then the instance does not admit a feasible schedule because the schedule starting from the next request was I/O-optimal and busy, or $t < 0$, but the next fetch already started as late as possible by LAZY.

MIN. Follows from Lemma 4, LAZY and Lemma 3.

Change in partial schedule: p fetched earlier, kept in cache (Line 28), $\mathsf{next} \leq e_i$**:** LAZY-LFD must keep p in cache until the end of the interval that it fetches it for, i.e., until time $e_i \geq \mathsf{next}$. LAZY-LFD sets $\mathsf{full} = \infty$ in Line 14 and continues to Line 20. We can remove the fetch at time next in line 20 since a new fetch will be created at time t in line 28, and I/O-OPT holds by induction. The new schedule also satisfies the FIFO and BUSY property. Thus if it is infeasible, no feasible schedule exists by induction and LAZY (for this smaller input).

If *every* fetch between t and next gets postponed by 1 in line 26, LAZY and MIN hold by induction (the number of pages in cache has not increased at any time). If there is a fetch which gets postponed less, this can only happen because the deadline d_j for the corresponding page p_j is within 1 of the old end of the fetch. However, in this case, we again get two independent input sequences (before and after time d_j). Therefore, we can apply induction on the part of the input sequence ending at time d_j and conclude that LAZY and MIN hold.

Line 28, cache is never full during $[e_i, \mathsf{next}]$: We can certainly keep p in the cache without overloading it, and save an access. The schedule for the other pages may remain the same, or some fetches may now be moved (Line 26). Similarly, if the cache is only full **after** the last fetch with slack less than 1, all the fetches after that one can be postponed by 1 (after removing the access at time next) without violating any deadlines, meaning that we can save one cache slot in every time step, and we can save one access by keeping p in cache. An example of this situation can be seen in Figure 2.

In this case we do not increase the number of fetches in this step. Therefore, by induction the resulting schedule is I/O-optimal (I/O-OPT) and by the reasoning above it is feasible (FEASIBLE). Also, the first fetch in a feasible schedule cannot start later if the input becomes larger (LAZY) and the cache is not fuller at any time than it was before handling this request (MIN).

Line 28, all fetches in $[e_i, \mathsf{next}]$ **have slack at least 1:** There can be no interval in $[e_i, \mathsf{next}]$ in which no fetch takes place (because then the fetch immediately before that interval could be postponed, and LAZY-LFD would have done this). Thus at all times, some page is being fetched. In this case all fetches can be postponed by 1 as above after removing the access at time next. Clearly, the new schedule does not violate cache capacity constraints or deadlines. We can show that LAZY and MIN hold as in the previous case. If $t < 0$, then no feasible schedule exists. Else, FEASIBLE holds.

In all these cases, we find

$$\text{LAZY-LFD}(i) = \text{LAZY-LFD}(i+1) = \text{OPT}(i+1) = \text{OPT}(i)$$

(the last equality follows since $\text{OPT}(i+1) \leq \text{OPT}(i) \leq \text{LAZY-LFD}(i)$). So I/O-OPT holds if LAZY-LFD creates a fetch in line 28. This completes the proof of Theorem 3. □

When using LAZY-LFD in practice, e.g., for a soft real time system, it is a disadvantage that it fetches requests *as late as possible*. In contrast, EAGER-LFD in some sense fetches request *as early as possible* which is good since such a schedule is more resilient to fluctuations in fetch times[2]. Indeed, we can use a

[2]Which are very common for hard disks, whereas for SSDs – at least when only reading – it should be possible to have highly deterministic disk access delays.

combination of LAZY-LFD and EAGER-LFD that has the good features of both: For any feasible request sequence σ, first find an I/O-optimal schedule S with LAZY-LFD. Then build a new request sequence σ' that only contains the primary requests of σ for S but whose evict times are postponed to also cover the free requests. More formally,

$$\sigma' = \langle (p_i, [d_i, \max\{e_k : f_k = f_i\}) : \sigma_i \text{ is primary in } S\rangle .$$

Now apply EAGER-LFD to σ' thus moving fetches for σ' as far ahead as possible. Since σ' is feasible, EAGER-LFD will also find a feasible schedule. Moreover, this schedule will keep all pages long enough in cache to serve all the original requests in σ. Since it performs exactly $|\sigma'|$ fetches, this new schedule is also I/O optimal.

6 Online algorithms

In the pure online model, it is impossible for an online algorithm to handle the hard deadlines properly. In fact, we have the following lemma which shows that even lookahead does not help much.

Lemma 5 *Any finite amount of lookahead is insufficient by itself to provide feasible schedules.*

Proof: Let the lookahead be $n-1$ time units, and $k=2$ (for simplicity). Consider the following request sequence.

Page	a	b	x_1	x_2	...	x_n	a (or b)
Deadline	1	2	3	4	...	$n+2$	$n+2$

At time 2 the online algorithm needs to decide whether it removes page a or page b from its cache to fetch x_1. However, with a lookahead of $n-1$, it is impossible to know which page to evict. □

The explanation is that an online algorithm cannot handle more than 1 pages being requested per time unit on average, because it will need to decide which pages to evict and will inevitably make the wrong decisions. We see that the (single) disk is always busy in the above example. We therefore consider the resource augmentation model.

An option is to give the online algorithm a larger cache than the offline algorithm it is compared to. However, the example also shows that a larger cache does not really help: at some point a page must be evicted, and this will be the page on which the algorithm fails later.

We can also allow the online algorithm to fetch pages faster than the offline algorithm. We show in the following that this does allow for a competitive algorithm. In particular, we show that using a very simple algorithm, we can handle any sequence of requests which allows a feasible schedule as long as we have a lookahead of k and can fetch pages with twice the speed of the offline algorithm. That is, fetching a page costs only half a time unit for the online algorithm. Equivalently, we can also give the online algorithm the power to fetch two pages at the same time, by assuming that it has two parallel disks that both store all the data that is required.

In order to analyze our algorithm, we first consider offline algorithms for this problem. There exists an optimal offline algorithm with the following properties.

Assumption 1 *No pages are evicted during a fetch.*

It can be seen that evicting pages during fetches, instead of waiting until the end of the current fetch and then evicting them, cannot help an algorithm with respect to deadlines of later requests.

Assumption 2 *At most one page is evicted at the start of a fetch.*

Since at most one page can be loaded during a single fetch, it does not help to evict more than one page at the start of a fetch, since only one slot can be filled with this fetch anyway. On the other hand, it also does not harm to keep as many pages as possible in the cache, since at most one free slot is needed for any fetch.

Assumption 3 *Pages are evicted only at the start of fetches.*

Since at the beginning the cache is empty, and each fetch loads only one page, there is no need to evict pages at any other time.

Lemma 6 *The contents of the cache of the optimal offline algorithm change for at most one slot in any (half-open) interval of length 1.*

Proof: This follows immediately from the above assumptions. □

Lemma 7 *In an instance that allows a feasible solution, in an interval I of length strictly smaller than $i \in \mathbb{N}$, there cannot be more than $k + i - 1$ requests for distinct pages.*

Proof: At the deadline d of the last request in I, before any page is evicted, by the above assumptions k of the requested pages in I are in the offline cache. During any fetch, the configuration of the offline cache does not change. By Lemma 6 and Assumption 3, the configuration only changes at the end of fetches, and only by one page. Thus in I, the configuration can only change at most $i - 1$ times before the final fetch. This means that in total, at most $k + i - 1$ distinct pages are present in the cache during I (not all at the same time, but overall). This is then an upper bound for the number of pages that can be requested in a feasible problem instance. □

Remark 1 *In any interval $(0, t]$, at most $\lfloor t \rfloor$ distinct pages can be requested in a feasible instance.*

Consider a greedy algorithm for general k. See Figure 3. This algorithm simply loads pages in the order in which they are requested, as early as possible, and evicts pages that it does not see in its lookahead.

```
At each time $t$, do the following.

    1. If there is any page $p$ in the cache that is not currently needed and that is not visible in the looka-
       head, evict $p$.

    2. If a request for page $p$ becomes visible, schedule page $p$ to be loaded at the earliest time at or after
       $t$ at which there is a free slot in the cache.

    3. Load any pages that are scheduled to be loaded at time $t$.
```

Figure 3: A greedy online algorithm

Lemma 8 *Greedy with a speed of s (i.e., that needs only $1/s$ time units for each page fetch) and a lookahead of $k/(s - 1)$ creates a feasible schedule for each input for which a feasible schedule exists.*

Proof: We prove by induction on the number of requests that the greedy algorithm provides a feasible schedule, if one exists. Suppose the algorithm sees its first request at time t. Then this request has deadline $t+k/(s-1)$. If the algorithm fails at this point, there are at least $k+1$ pages required at time $t+k/(s-1)$, which means there is no feasible schedule.

Consider a fetch (of page p) that finishes at time t and suppose the algorithm did not fail yet. Greedy plans to fetch the first requested page with deadline t or greater that is not in its cache. The only case in which this fails is if there exists a page q with deadline smaller than $t+\frac{1}{s}$ which is not in the cache of Greedy. This page was already visible at time $t+\frac{1}{s}-\frac{k}{s-1}$. Greedy must have been loading pages throughout the interval $[t+\frac{1}{s}-\frac{k}{s-1}, t]$, loading $s(\frac{k}{s-1}-\frac{1}{s}) = \frac{sk}{s-1}-1$ pages in this time. It was also loading a page immediately before time $t+\frac{1}{s}-\frac{k}{s-1}$, since otherwise it would have started to load q sooner. But this means that in the interval $[t+\frac{1}{s}-\frac{k}{s-1}, t+\frac{1}{s})$ there are $\frac{sk}{s-1}$ distinct pages requested (Greedy would not evict q at time $t+\frac{1}{s}-\frac{k}{s-1}$ anymore). However, by Lemma 7, there can be at most $k+\frac{k}{s-1}-1 = \frac{sk-s+1}{s-1}$ pages requested in an interval of length less than $\frac{k}{s-1}$, a contradiction. □

We now show a matching lower bound, showing a tight relationship between the amount of lookahead and the amount of resource augmentation on the speed that is required for an online algorithm to provide feasible schedules for feasible inputs.

Theorem 4 *An online algorithm with a disk of speed s, or s parallel disks of speed 1, needs at least a lookahead of $k/(s-1)$ in order to be able to create feasible schedules for all feasible inputs.*

Proof: Consider an online algorithm A. Assume that the amount of lookahead is $\ell < k/(s-1)$ and consider the following instance. It consists of k pages required at time k, followed by new distinct pages with deadlines at each time $k+i$ for $i=1,\dots,2k$. At time $3k-\ell$, A has at most k of the at least $2k$ pages with deadline no later than $3k-\ell$ in cache. We now add a request for k pages that A does not have in its cache, but that were already requested, at time $3k+1$.

The optimal offline solution is the following: first load the k pages requested at time k in the interval $[0,k]$. In each successive interval of length 1 until time $3k$, load one page and evict one page that will not be requested again. It can only happen once that there is no such page in cache, namely if all k pages in cache are requested at time $3k+1$ (which is the only time at which requests are repeated). In that case, evict an arbitrary page, and reload it in the interval $[3k, 3k+1]$. In all cases, this produces a feasible solution.

In this instance, in the interval $(3k-\ell, 3k+1]$, there are deadlines for $k+\ell$ distinct pages. Loading all these pages takes at least $(k+\ell)/s$ time for A. Thus we find as condition for ℓ such that A might create a feasible schedule that $\ell \geq (k+\ell)/s$, or $\ell \geq k/(s-1)$. □

If we give the online algorithm parallel disks instead of a faster disk, we get slightly different results because now loading $k+\ell$ distinct pages takes at least $\lceil (k+\ell)/s \rceil$ time, so the required lookahead may be slightly larger depending on k, s and ℓ.

7 Conclusions

We have introduced a model for real-time prefetching and caching that is simple, seems to model practically relevant issues, and allows fast and simple algorithms with useful performance guarantees in offline and online settings. Although previous work from non-real-time models provides useful ideas for algorithms, the situation in the real-time setting is often different (e.g., wrt to I/O optimality of LFD or how to measure lookahead). Hence, given the importance of real-time applications, we expect that more work will be done on this subject in the future.

One interesting open question is the case of parallel disks. Although the *hard real-time* case we currently consider is very important since hard real-time constraints are present in many safety critical systems (e.g. avionics), we could also look at *soft real-time* where the applications remains viable when some requests are missed but we want to minimize the number of missed requests (or the sum of importance weights given for the missed requests). Finally, it could be interesting to consider the case where a page is modified while it is in the cache and needs to be written back to disk.

Acknowledgment The authors would like to thank anonymous referees who pointed out some oversights in the proofs in a previous version of this paper.

References

[1] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18:283–305, 1997.

[2] Susanne Albers, Naveen Garg, and Stefano Leonardi. Minimizing stall time in single and parallel disk systems. *J. ACM*, 47(6):969–986, 2000.

[3] Laszlo A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.*, 5:78–101, 1966.

[4] Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[5] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *J. Comput. Systems Sci.*, 50:244–258, 1995.

[6] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS*, pages 188–197, 1995.

[7] Ö. Ertug, M. Kallahalla, and P. J. Varman. Real time parallel disk scheduling for vbr video servers. In *Proc. of Fifth Intl. Conf. On Computer Science and Informatics (CSI'00)*, Chennai, India, 2000.

[8] Amos Fiat and M. Mendel. Truly online paging with locality of reference. In *Proc. 38th Symp. Foundations of Computer Science (FOCS)*, pages 326–335. IEEE, 1997.

[9] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality Between Prefetching and Queued Writing. *SIAM Journal on Computing*, 34(6):1443–1463, 2005.

[10] M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *6th Workshop on Input/Output in Parallel and Distributed Systems*, pages 68–77, 1999.

[11] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O systems. In *13th Symposium on Parallel Algorithms and Architectures*, pages 219–228, 2001.

[12] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.

[13] P. Krishnan and Jeffrey Scott Vitter. Optimal prediction for prefetching in the worst case. *SIAM J. Comput.*, 27(6):1617–1636, 1998.

[14] Konstantinos Panagiotou and Alexander Souza. On adequate performance measures for paging. In *STOC '06: Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 487–496, New York, NY, USA, 2006. ACM Press.

[15] Jeffrey Scott Vitter and P. Krishnan. Optimal prefetching via data compression. *J. ACM*, 43(5):771–793, 1996.