Online Algorithms for Prefetching and Caching on Parallel Disks*

Rahul Shah
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907
rahul@cs.purdue.edu

Peter J. Varman
Dept. of ECE
Rice University
Houston, TX 77005
pjv@rice.edu

Jeffrey Scott Vitter
Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907
jsv@purdue.edu

ABSTRACT

Parallel disks provide a cost effective way of speeding up I/Os in applications that work with large amounts of data. The main challenge is to achieve as much parallelism as possible, using prefetching to avoid bottlenecks in disk access. Efficient algorithms have been developed for some particular patterns of accessing the disk blocks. In this paper, we consider general request sequences. When the request sequence consists of unique block requests, the problem is called *prefetching* and is a well-solved problem for arbitrary request sequences. When the reference sequence can have repeated references to the same block, we need to devise an effective *caching* policy as well. While optimum offline algorithms have been recently designed for the problem, in the online case, no effective algorithm was previously known. Our main contribution is a deterministic online algorithm *threshold-LRU* which achieves $O((MD/L)^{2/3})$ competitive ratio and a randomized online algorithm *threshold-MARK* which achieves $O(\sqrt{(MD/L)\log(MD/L)})$ competitive ratio for the caching/prefetching problem on the parallel disk model (PDM), where D is the number of disks, M is the size of fast memory buffer, and $M + L$ is the amount of lookahead available in the request sequence. The best-known lower bound on the competitive ratio is $\Omega(\sqrt{MD/L})$ for lookahead $L \geq M$ in both models. We also show that if the deterministic online algorithm is allowed to have twice the memory of the offline then a tight competitive ratio of $\Theta(\sqrt{MD/L})$ can be achieved. This problem generalizes the well-known paging problem on a single disk to the parallel disk model.

1. INTRODUCTION

The parallel disk model (PDM) [19] is one of the most popular models for measuring the I/O complexity of problems when data are allowed to be on multiple disks. In each I/O step, a block from each disk can be read (or written) simultaneously. That is, if D is the number of disks, in one I/O step we can read or write as many as D blocks if these blocks happen to fall on different disks. A fundamental problem in this model is caching/prefetching: Given an ordered sequence of read requests and a main memory buffer of size M, the problem is to generate the I/O schedule to serve these requests in the minimum number of parallel I/Os. At each I/O step, at most one block from each disk is read into the buffer. The read request for the block can be served from the buffer. At the moment when a read request for a particular block is issued, an I/O step must be done to fetch that block into memory buffer from its corresponding disk if the block is not already in the memory buffer. During this I/O step, we might choose to prefetch some blocks from other disks, so that when the blocks are requested they are already in memory, thus avoiding I/O. This is called the *prefetching* problem for parallel disks. After the block's read request is served, we may still want to keep it in the memory buffer in case any future read request for the same block occurs in the read sequence. Determining which blocks to keep in the buffer is known as the *caching* problem.

For minimizing the number of I/Os in the parallel disk model, caching and prefetching have to be considered simultaneously. In the special case where all the blocks in the request sequence are unique, which we call the *read-once* version of the problem, no caching is required and the only problem is prefetching. The more general *read-many* version involves caching. If the full request sequence is known in advance (i.e., if future references are known), the problem becomes *offline*. In the *online* problem we consider in this paper, future requests are unknown to the algorithm or perhaps are known for only some limited lookahead. We consider competitive ratio as the measure of effectiveness of our online algorithms.

Prefetching is needed to take advantage of parallelism. In order to do prefetching effectively, an algorithm needs a certain amount of lookahead into the request sequence. In the read-once case, this lookahead can be in the form of the next L blocks in the request sequence. However, in the read-many case, an adversary can nullify any advantage of lookahead by repeating the same block L times consecutively. In this case, the lookahead doesn't provide much information. To overcome this problem, many different definitions of looka-

*Supported in part the Army Research Office through grant DAAD19–03–1–0321, by the National Science Foundation through Grants CCR–9877133 and 0105565, and an IBM research grant.

SPAA'04, June 27–30, 2004, Barcelona, Spain.

heads are considered in the (single-disk) paging literature [20, 1, 8]. We shall assume the definition provided by [1] of *strong lookahead*. The algorithm has a strong lookahead of size L if, at any given time, it can see up to the number of references into the future that are sufficient to have L distinct blocks in the lookahead string. We shall use the term "lookahead" to implicitly mean strong lookahead.

Lookahead of at least M blocks is required in order to achieve a competitive ratio achieving full parallelism. When lookahead is less than $M-D$, the lower bounds for the single disk case (which is a particular case of our problem) are $\Omega(D)$, and hence meaningful parallelism cannot be achieved. Hence, whenever convenient, we shall denote the lookahead by $M + L$. The challenge is to achieve a competitive ratio sublinear in D.

1.1 Previous and related work

Paging (caching) algorithms have a long history in computer science. Given a sequence of n pages (or blocks) to be read and a memory of size M, which pages should be kept in fast memory so as to have the minimum number of page faults? When the request sequence is known in advance, Belady's MIN replacement policy [6] achieves a minimum number of page faults. When future requests are not known to the algorithm, the problem becomes online. The competitive ratio measures how best any online algorithm without knowledge of future requests compares with the one with full future knowledge. See [18, 14, 1, 10, 7] for some fundamental results for competitive online paging.

Sleator and Tarjan [18] showed that LRU and FIFO replacement policies achieve a competitive ratio of M, which is the best possible for any online algorithm in a deterministic setting. Albers [1] showed that when the online algorithm has a strong lookahead of L the competitive ratio can be lowered to $M - L$ for $L \leq M - 2$. This algorithm is based on LRU. Several other definitions of lookahead are considered in the literature, like the resource-bounded lookahead of Young [20] or the natural lookahead of Breslauer [8]. The competitive ratio when the online algorithm has more resources than the offline algorithm is also considered. If the online algorithm has $M + K$ fast memory, then the competitive ratio is $(M + K)/(1 + K)$, when compared with the offline algorithm with memory size M [18, 8]. Under randomized setting against an oblivious adversary a competitive ratio $2H_M$ can be achieved by the marking algorithm[10, 7] which is asymptotically tight. A tight competitive ratio of H_M was obtained by the partition algorithm of McGeoch and Sleator [17].

When considering the problem on the parallel disk model (PDM), lookahead is a must in order to effectively prefetch blocks. The problem of minimizing page faults translates to the problem of minimizing the number of parallel I/Os. This problem is interesting even when the request sequence consists of distinct pages (read-once). Here, in order to exploit parallelization, we need to prefetch some blocks into fast memory using parallel I/Os. This problem is called prefetching. When the pages can repeat in the sequence, then caching (paging) comes into play. In the particular case when the number of disks is one, this problem reduces to the paging problem described above.

This problem has been studied extensively. When lookahead is M, Barve et al. [5] gave the first $O(\sqrt{D})$ competitive algorithm for the read-once problem. Following this, the read-once problem was optimally solved for all ranges of lookahead ($L < M$ and $L > M$) by Kallahalla and Varman [12]. They also gave the first optimal offline algorithm for the read-once case. Despite the optimal progress on prefetching, any effective online caching policy was not known on PDM. Kallahalla and Varman [13] and Hutchinson et al. [11] independently gave the optimal offline strategies for the caching/prefetching problem (i.e., read-many), which can be seen as a parallel generalization of Belady's MIN policy and the optimal solution for the read-once (prefetch only) problem. The only online algorithm, which was given by Kallahalla and Varman [13], is $O(D)$-competitive when lookahead is a constant multiple of M. The read-once problem (prefetch) has a tight competitive ratio of $\Theta(\sqrt{MD/L})$, given by [12].

A similar problem has been well considered by Karlin et al. [9, 16, 15] and Albers et al. [3, 2, 4] in a more general setting than PDM. In those papers, two factors are considered: the time needed by the application to process an in-memory block and the time needed to fetch a block from disk into fast memory. The problem is then to minimize total time or other similar metrics. This model tends to PDM in the limit as CPU speed increases relative to the disk latency. Most of the work is focused on approximation algorithms for the offline problem. We note that although some strategies used in the offline case for this model (like *aggressive, reverse aggressive and conservative*) are strikingly similar to those used in the optimal PDM algorithms (like *greedy write, duality, lazy prefetching, prudent prefetching*), the analysis (such as in [16]) tends be more suboptimal in the limit as their model approaches PDM. A striking result of the PDM work is that a similar framework to those in [9, 16, 15, 3, 2, 4] with proper augmentation and analysis can indeed provide optimal algorithms in PDM. While the more general problem remains hard, on PDM, this is a well-solved problem in offline case and in the case where request sequence is read-once (prefetching).

1.2 Our results

Our main contributions are summarized as follows:

1. For the read-once case, we present a simplified algorithm which achieves the asymptotically tight competitive ratio of $O(\sqrt{MD/L})$.

2. For the read-many case, under deterministic setting, we present the parallel disk generalization of LRU called *threshold-LRU* (or tLRU), which achieves a competitive ratio of $O((MD/L)^{2/3})$ for lookahead $M+L$. This improves upon the previous best of $O(MD/L)$.

3. For the read-many case, under a randomized (oblivious adversary) setting, we present a parallel generalization of the MARK algorithm called *threshold-MARK* (or tMARK). This achieves the expected competitive ratio of $O(\sqrt{(MD/L)\log(MD/L)})$ for lookahead $M + L$.

4. For the read-many case, under deterministic setting, we show that the asymptotically tight competitive ratio of $O(\sqrt{MD/L})$ can be achieved if the online algorithm (tLRU) is allowed twice the memory of the offline adversary. This is asymptotically tight when $L > (1 + 1/\epsilon)M$ for any constant ϵ.

Problem	Results (Competitive Ratio)	Lower bound	Algorithm
Offline read-once	optimal	–	L-OPT[12], duality+FIFO[11]
Offline read-many	optimal	–	Supervisor[13], duality+MIN[11]
Online read-once	$O(\sqrt{D})$, $L = M$	$\Omega(\sqrt{D})$	NOM [5]
	$O(\sqrt{MD/L})$	$\Omega(\sqrt{MD/L})$	L-OPT[12],threshold [this paper]
Online read-many	$O(MD/L)$, $L \geq M$	$\Omega(\sqrt{MD/L})$	Supervisor [13]
	$O((MD/L)^{2/3})$, $L \geq (1+1/\epsilon)M$	$\Omega(\sqrt{MD/L})$	tLRU [this paper]
Rand. Online r-many	$O(\sqrt{(MD/L)\log(MD/L)})$, $L \geq (1+1/\epsilon)M$	$\Omega(\sqrt{MD/L})$	tMARK [this paper]

Table 1: Caching and Prefetching Results on PDM

Our algorithms are built up on the work of [18, 1, 12, 11, 10].

In Section 2, we formulate the problem and describe known results which are essential building blocks of our algorithm. In Section 3, we present four algorithms with competitive analyses. We conclude with open problems and issues in Section 4.

2. PRELIMINARIES

2.1 Problem setting and formulations

Our problem setting consists of a request sequence Σ of n blocks (not necessarily distinct), a lookahead window of size L, buffer memory (shared) of size M, and D disks. The request sequence is an ordered sequence of n pages (or blocks). Each block is associated with its particular disk. The request sequence can be seen as a series of read requests for particular blocks. At the time when the read request is issued, if the requested block is in the buffer, it is accessed there; otherwise, an I/O step must be performed to fetch the block from its corresponding disk into the buffer. The I/Os are parallel; in each I/O step, at most D blocks can be fetched, one from each disk.

The central task is to schedule the I/Os so that we can exploit maximum parallelization and minimize the number of I/O steps (or I/Os for short) required to serve the request sequence Σ. As noted earlier, prefetching blocks is a necessity. To prefetch effectively, the scheduling algorithm is provided with the lookahead which, at the point when a particular block b in Σ is referenced, consists of the request sequence consisting of the next L distinct blocks after b.

When Σ consists exclusively of unique blocks (i.e., with no repetitions), it is called a *read-once* sequence. Otherwise, it is called a *read-many* sequence. Given a request sequence Σ, the problem is to construct a minimum-I/O read schedule. The solution in the *offline* setting, when the lookahead is infinite (i.e., at least as long as the request sequence), appears in [12, 13, 11], which is briefly described in section 2.3. In the *online* case, when lookahead is smaller than the request sequence, the scheduling algorithm may not achieve an I/O schedule as short as that of the offline problem.

Under the deterministic settings, the maximum (over the entire request sequence) ratio of the length of schedule obtained by an online algorithm A with bounded lookahead L to the length of optimal offline schedule, is the *competitive ratio* obtained by the online algorithm A. The competitive ratio obtained by any algorithm forms an upper bound for the competitive ratio of the problem. If for a given online algorithm for a problem there is some request sequence such that the length of the schedule generated is at least c times that of the offline optimum, then c is the lower bound for the competitive ratio of the problem. In a randomized setting, this competitive ratio is the ratio of the expected length of the online algorithm's schedule to that of the the optimum offline's schedule. Randomized online algorithms have better competitive ratios for many problems under the oblivious adversary model, where the request sequence is predetermined.

The competitive ratio in general assumes that both the offline and online algorithms use the same amount of buffer space, namely, M. We also consider the case when the online algorithm has buffer space $2M$, and we compare its performance to that of the offline algorithm with buffer space M. We achieve a competitive algorithm for this case.

All our algorithms are phase-wise. A phase is a contiguous subsequence of the full request sequence $\Sigma = (b_1, b_2, \ldots, b_n)$. The phases partition Σ into contiguous groups of blocks. The size of the phase is the same as the lookahead. That is, the first phase consists of the blocks $(b_1, b_2, \ldots, b_j)$ where j is the minimum index such that $(b_1, b_2, \ldots, b_j)$ consists of L distinct blocks. This is also the lookahead available to the algorithm initially. Phase i consists of blocks $(b_k, \ldots, b_l)$, where b_{k-1} is the last block of phase $i-1$ and l is the least index such that $(b_k, \ldots, b_l)$ consists of L distinct blocks. During the schedule, the I/Os for phase i start after the I/O in which the last block of phase $i-1$ is fetched and referenced and end with the I/O when all blocks of the phase i are referenced.

2.2 Lower bounds and motivating examples

Since read-once is a particular case of read-many, the lower bounds on the competitive ratio for the read-once case also apply to the read-many case. The lower bound for the read-once case when the lookahead is of size M is $\Omega(\sqrt{D})$. Intuitively, this can be visualized as follows: Consider the alternating sequence of good phases and bad phases. A good phase consists of M requests striped equally on each disk. A bad phase consists of $M/\sqrt{D}$ requests on one particular disk, called the bad disk, and other requests striped equally. Consider a series of $\sqrt{D}$ such good and bad phases. An offline algorithm can prefetch all the blocks on bad disks (in their respective bad phases) during the first bad phase, while the online algorithm has no idea what these blocks are. Hence, the offline algorithm does $(M/\sqrt{D} + M/D)$ I/Os in the first pair of phases and then $2M/D$ in the remaining $2\sqrt{D} - 2$ phases, while the online algorithm incurs $M/\sqrt{D}$ I/Os in every bad phase. For the formal proof and illustration see [5].

This lower bound can be easily extended to the case when the lookahead L is greater than M. In this case, $M/\sqrt{D}$ is replaced by $\sqrt{ML/D}$, and hence there are $\sqrt{MD/L}$ pairs of phases, thus giving a lower bound of $\Omega(\sqrt{MD/L})$ on the

competitive ratio [12]. In our case we assume a lookahead of $L+M$, making this lower bound $\Omega(\sqrt{MD/(M+L)})$. This is the same as $\Omega(\sqrt{MD/L})$ in the asymptotic sense when $L \geq M/\epsilon$, for any constant ϵ.

When designing single disk online algorithms with lookahead, LRU turns out to be the best deterministic policy [1]. In the case of parallel disks, however, this is not exactly true. Consider the following example: Let the lookahead be $2M$. Let Σ_1 be the request sequence of $3M$ distinct references all striped equally on D disks. Let Σ_2 consist of M distinct references on disk 1. Let all the requests in Σ_2 be distinct from those in Σ_1. Now, the request sequence consists of many (possibly infinite) repetitions of $\Sigma_1\Sigma_2$. Simple LRU will fault $M+3M/D$ times on each repetition, while if we only cache the blocks in Σ_2, we fault $M+3M/D$ times for first occurrence but roughly only $3M/D$ times in each subsequent repetition. Thus, LRU can be off by a factor of $O(D)$. (This is basically the essence of the algorithm in [13].) However, some good combination of parallelization with LRU might give better bounds.

2.3 Optimal offline algorithms and duality

In this section, we discuss optimal offline algorithms known in literature. We shall use these algorithms as subroutines in our online solutions. We additionally show how to compute an optimal read schedule for a given sequence of accesses, assuming that we already have some of the required blocks in memory. The notion of duality and the proofs of optimality can be found in [11].

2.3.1 Read-Once case

The algorithm for achieving a minimum number of parallel read I/Os can be obtained by duality [11]. The duality reduces the read problem on the request sequence Σ to the write problem on the request sequence Σ^R, which has a simple greedy optimal solution. The greedy solution (see Theorem 2 of [11]) involves the repetition of the two steps

1. Insert as many blocks as possible into buffer to make it full.

2. Write blocks to as many disks as possible in a single I/O step.

until all the blocks are issued, after which we finally flush all the blocks remaining in the buffer onto the disks. For each disk, it uses a FIFO discipline for writing. By duality, this greedy writing translates to lazy prefetching for the read-once problem, giving an optimal number of read I/O steps for the request sequence. For illustrative examples of how simple heuristics like greedy reading perform worse than lazy prefetching, we refer the reader to [11].

2.3.2 Read-Many case

When blocks are no longer distinct, they may be retained in the buffer even after they are accessed, for referencing them in the future without future I/Os. The corresponding write-many problem is formulated as follows: Write the blocks in Σ^R onto the disk so that the latest instance of each block is either on its assigned disk or in the buffer pool during the schedule. The final instance of each block must be written to its assigned disk. We may not need to write the intermediate instances of blocks onto disk as long as we have them in the buffer.

The solution is that, in each write step, instead of following FIFO to write a block onto disk, the block that is written (among the blocks for that disk) is the one whose next reference in the string is the latest. This is like Belady's MIN heuristic [6, 11] but done in the disk wise sense. The reverse of this gives the optimal read schedule (see Theorem 4 of [11]).

2.3.3 Read-Many case with initial memory condition

Our algorithm will be a combination of the optimal algorithm above within a phase and an efficient caching policy across phases. We consider a slight modification of the optimal algorithm in the read-many case that will be useful later on. The problem is: Given an initial memory buffer containing a set S of $\leq M$ blocks and given the request sequence Σ to read, what is the optimal schedule? Note that in the previous case, the initial memory S is empty.

By duality, this problem can be done by the write schedule on $\Sigma\hat{S}$ where $\hat{S}$ is any arbitrary ordering of blocks in S and the final instance of blocks in S need not be written to the disk.

3. ALGORITHMS

In this section, we describe our online algorithm for integrated caching and prefetching. Before we describe our final algorithm, we start with two simple algorithms (Sections 3.1 and 3.2), which will lead us towards the final algorithm in Section 3.3. All our algorithms are *phase-wise* algorithms as compared with priority-controlled sliding window algorithms in literature [12, 13]. By phase-wise, we mean that the request sequence is broken into phases and we treat the request sequence one phase at a time. The phase is as big as the lookahead.

3.1 Read-once case

In this section, we consider the read-once case, in which every block appears only once in the request sequence, as in [12]. We describe a simple phase-wise algorithm which achieves the same competitive ratio as [12] but keeps the analysis simpler using a potential function.

Let the lookahead size be L and the request sequence be $\Sigma = (b_0, b_1, \ldots, b_{n-1})$. We shall assume n to be an integral multiple of the lookahead size L. We break up the request sequence into phases of size L, where phase i consists of blocks $\Sigma_i = (b_{iL}, b_{iL+1}, \ldots, b_{(i+1)L-1})$. There are a total of n/L phases $\Sigma_0, \ldots, \Sigma_{n/L-1}$.

Our algorithm A will treat each phase independently. For each phase, the algorithm will be able to see the entire request sequence for that phase at the beginning of the phase. The algorithm A empties its memory before starting the new phase. It then finds the best possible I/O schedule for each phase, using the optimal duality-based scheduling algorithm [11]. We shall compare the competitiveness of this algorithm with the optimal offline algorithm O that knows the entire request sequence in advance.

THEOREM 1. *The algorithm A is $O(\sqrt{MD/L})$ competitive.*

PROOF. Let M_i be the memory of O at the beginning of the phase i. We shall define the potential Φ_i at the beginning of phase i (we can break the execution of O into phases also– a phase end when O serves the last block in that phase) to

reflect this memory state of O as follows: Let M_i^d be the number of blocks in M_i that come from disk d. Then,

$$\Phi_i = \sum_{d=1}^{D} \max(M_i^d - \sqrt{ML/D}, 0)$$

We call this potential function Φ a threshold potential function, where the threshold size t is set to be equal to $\sqrt{ML/D}$. One important point is that at most $\sqrt{MD/L}$ disks can contribute a positive quantity to this potential function. (Otherwise, since each such disk must have at least $\sqrt{ML/D}$ blocks in memory, the total number of blocks in memory would be more than M.) Let o_i be the number of I/Os that O does in the phase i. (Phase i starts after the last I/O of phase $i-1$ and ends with the I/O in which the last block of phase i is fetched.) Let a_i be the number of I/Os done by our algorithm in phase i. The memory of our algorithm A is empty before the start of the phase, and given an empty starting memory, A is optimal in phase i. However, O may do fewer I/Os in phase i because it has already prefetched some blocks in the memory before the beginning of phase i.

Let's consider the case $o_i < a_i$. It follows that the contents M_i of memory include $a_i - o_i$ blocks accessed in phase i from some disk, say, disk d. (Otherwise A can obtain a shorter schedule in phase i by prefetching the blocks of M_i referenced in phase i and then simulating O's schedule restricted to phase i). By the read-once property, at the end of phase i, we can assume that O deletes from memory all the blocks used in phase i. In particular, M_{i+1} loses at least $a_i - o_i$ blocks that belonged to disk d. During phase i, O can prefetch at most o_i from each disk to be used in later phases.

The net effect of these changes in memory content is that during phase i the potential function decreases by at least $a_i - o_i - \sqrt{ML/D}$ and increases by at most $o_i\sqrt{MD/L}$. Therefore, we have

$$\Phi_{i+1} \leq \Phi_i - (a_i - o_i - \sqrt{ML/D}) + o_i\sqrt{MD/L}.$$

Summing up over all the phases and using the fact that $\Phi_0 = \Phi_{N/L+1} = 0$, we get

$$\sum_{i=0}^{n/L} a_i \leq \sum_{i=0}^{n/L} o_i(1 + \sqrt{MD/L}) + \sum_{i=0}^{n/L} \sqrt{ML/D}.$$

The last term on the right-hand side sums up to $n\sqrt{M/LD}$. Since O does at least n/D I/Os over the course of the entire algorithm, we have $\sum_{i=0}^{n/L} o_i\sqrt{MD/L} \geq (n/D)\sqrt{MD/L} = n\sqrt{M/LD}$. Substituting this into the previous equation, we get

$$\sum a_i \leq \sum o_i(1 + 2\sqrt{MD/L}).$$

This implies that A is $O(\sqrt{MD/L})$ competitive. □

The threshold potential function Φ captures two main ideas in order to achieve competitive ratio of $\sqrt{MD/L}$:

1. The online algorithm A can have $\sqrt{ML/D}$ I/Os in each phase "for free" without violating the desired competitiveness ratio. The cost of these I/Os is paid by the fact that O on average spends at least L/D I/Os in each phase. Based on these free I/Os we shall develop our threshold-LRU algorithm, where the threshold t is set to be equal to number of free I/Os allowed in each phase.
2. In each phase, only $\sqrt{MD/L}$ disks can contribute significantly (i.e., more than $\sqrt{ML/D}$ to the difference between O and A).

These observations will motivate our approach for the read-many case where caching along with prefetching will play a significant role in the competitive analysis.

3.2 An algorithm for read-many case with separate caching and prefetching storage

Here, we assume that our algorithm A has a memory buffer of size $2M$: a prefetching storage space P of size M and a caching storage space C of size M. We shall compare this algorithm with the optimal offline algorithm O that uses a memory of size M. We assume that the lookahead for the online algorithm A is $M + L$, by which we mean that the algorithm can look ahead into the request sequence until it finds $M + L$ *distinct* block references. The actual number of page references in the lookahead can be arbitrarily larger than $M + L$ because of repetitions. As before, we use a phase-wise approach, so our algorithm only uses the lookahead as it exists at the beginning of each phase. We shall achieve the competitive ratio of $O(\sqrt{MD/L})$.

Because we may need to access a given block multiple times in the read-many sequence, it is important to design some policy for caching the blocks already visited. Otherwise, as seen in examples earlier, the competitive ratio can be arbitrary. We cannot afford to have an empty memory state between phases as in the previous subsection. Our algorithm, called threshold-LRU (tLRU), will be motivated from the two observations of the previous subsection and the competitiveness of LRU algorithms for the single disk case [18, 1]. Here, we shall set the threshold size $t = \sqrt{ML/D}$. In each phase, the main idea is to cache blocks from disks that are above this threshold. That is, during the phase, we cache only from disks that have more than t new blocks, and we leave aside (i.e., do not cache) t blocks from each such disk.

Our algorithm A works as follows: Let C_i be the caching memory of our algorithm at the start of phase i. Let L_i be the lookahead (which shows the request sequence up to the end of phase i). We first update the LRU queue positions of the blocks in C_i according to L_i. That is, we move the blocks in $L_i \cap C_i$ to the bottom of the LRU eviction queue. We order them by LRU order as seen at the end of phase i. As in the previous section, let M_i be the set of memory blocks in the memory of O at beginning of the phase i.

At the beginning of each phase, we first prefetch the blocks from $L_i - C_i$ to arrive at new caching memory state C_{i+1}, with the following caveats: While doing this, we only prefetch the blocks from those disks on which $L_i - C_i$ has at least $\sqrt{ML/D}$ blocks. Each such disk is called *contributing disk*. Also while arriving at C_{i+1}, we do not prefetch all the blocks from contributing disks but we leave out those $\sqrt{ML/D}$ blocks on every contributing disk whose last occurrence at the end of phase i is the earliest. These blocks are called *left-out blocks*. To make space for these new incoming blocks we replace the blocks in C_i in LRU order as seen from the end of phase i.

If all these new blocks don't fit into C, then it means that $C_{i+1} \subseteq L_i$ but $L_i - C_{i+1}$ had more than $\sqrt{ML/D}$ blocks on

some disk. In this case we can be sure that any algorithm will incur at least $\sqrt{ML/D}$ I/Os in this phase. We simply try to hold as many blocks as we can in C_{i+1} by LRU order as seen from the end of phase i, in this case.

We denote by x_i the number of blocks prefetched by A at the beginning of phase i. At the end of this process, we refer to the resulting configuration of cache as C_{i+1}. We denote the number of I/Os required by A to do this prefetching stage as p_i, the *prefetching cost*. Note that $p_i \leq x_i \leq M$.

In order to process the request sequence, we use a static cache consisting of the blocks in C_{i+1} which have already been prefetched. For those blocks in the request sequence that are not in C_{i+1}, we serve the request sequence by using the optimal duality-based algorithm [11], using separate memory storage P and using the blocks in $L_i - C_{i+1}$ as the lookahead.

We denote the number of I/Os required by A to do the duality scheduling of $L_i - C_{i+1}$ as the *serving cost* s_i. Let the number of I/Os done by the optimal algorithm O in this phase be o_i.

LEMMA 1. *Either* $o_i > \sqrt{ML/D}$ *or* $s_i \leq o_i + \sqrt{ML/D}$.

PROOF. Let M_i be the memory of O before the phase i. If $o_i \leq \sqrt{ML/D}$, then $L_i - M_i$ has no more than $\sqrt{ML/D}$ blocks on any disk. This also means that all the blocks in L_i (excluding left-out blocks) prefetched by A can be maintained in C_{i+1}. Our algorithm A will not have to throw away any blocks which are in L_i as a part of replacement policy. Hence, $L_i - C_{i+1}$ has no more than $\sqrt{ML/D}$ blocks on any disk. One way for A to serve the requests is to prefetch blocks in $(M_i \cap L_i) - C_{i+1}$ into prefetching storage P using at most $\sqrt{ML/D}$ I/Os and then to mimic the behavior of O in phase i. Since A uses the optimal duality-based schedule, it always does at least this well. Hence, we get $s_i \leq o_i + \sqrt{ML/D}$. □

LEMMA 2. $s_i \leq 2o_i\sqrt{MD/L}$.

PROOF. Since the lookahead is $M+L$, we have $o_i > L/D$. If $o_i \leq \sqrt{ML/D}$, then by Lemma 1 we get the required inequality. If $o_i > \sqrt{ML/D}$, then our algorithm (during processing the phase) can mimic the initial memory condition of O using at most M I/Os and then continue processing as O. However, our algorithm being optimal in the phase means $s_i \leq o_i + M$. Since $o_i > \sqrt{ML/D}$, we get the identity. □

LEMMA 3. *For every sequence of phases* $i, i+1, \ldots, j$, *in which* $x_i + x_{i+1} + \cdots + x_j \geq M$, *the optimal offline algorithm* O *does at least* $\sqrt{ML/D}$ *I/Os.*

PROOF. Consider the memory M_i of O at the beginning of phase i. Let $L = L_i \cup L_{i+1} \cup \ldots \cup L_j$. Let $x = x_i + x_{i+1} + \cdots + x_j$. Since $x \geq M$ and we use LRU as a policy to evict blocks, L consists of at least M different blocks that our algorithm A paid for as a prefetch cost plus at least $\sqrt{ML/D}$ more blocks (the left-out blocks) on each contributing disk that algorithm A did not prefetch. Hence, whatever the state of M_i, there is at least one disk such that $L - M_i$ has more than $\sqrt{ML/D}$ blocks. Thus, O has to do at least $\sqrt{ML/D}$ I/Os. □

The total cost of our algorithm is $a_i = p_i + s_i$. We can assume that O initially does at least $\sqrt{ML/D}$ I/Os for A to start paying prefetch costs. Lemma 3 indicates that the amortized prefetch cost is never more than $2\sqrt{MD/L}$ times the cost for optimal algorithm. This is because every minimal sequence of phases where $x_i + x_{i+1} + \cdots + x_j \geq M$ satisfies $x_i + x_{i+1} + \cdots + x_j \leq 2M$. Thus the I/O cost of A is at most $4\sqrt{MD/L}$ times that of O.

THEOREM 2. *The competitive ratio of tLRU, with lookahead* $L + M$ *and buffer memory* $2M$, *is* $O(\sqrt{MD/L})$ *when compared against optimal offline algorithm having buffer memory* M.

When $L > M/\epsilon$, the $O(\sqrt{MD/L})$ upper bound on the competitive ratio is tight, with a similar lower bound within a constant factor. In the particular case when $L = M$ (i.e., lookahead of size $2M$), the threshold t becomes $M/\sqrt{D}$ (instead of $\sqrt{ML/D}$) and we get the competitive ratio of $O(\sqrt{D})$ (instead of $O(\sqrt{MD/L})$).

3.3 The online algorithm for integrated caching-prefetching

In this section, we give our main algorithm for the read-many case. Unlike the last section, we no longer assume separate memory storage for prefetching and caching. Our online algorithm will only have one buffer of size M. We shall build upon the algorithm in the previous subsection, the crux of the new algorithm being how to carefully separate the buffer into caching and prefetching parts while maintaining the competitive ratio. We may not be able to preserve all the blocks we have cached because we have to make space for servicing the phase. The service cost will be higher if less memory space is allowed for servicing. The idea, then, is to maintain as many cached blocks as possible without penalizing the service cost too much.

Our competitive ratio is not as tight as we obtained in the previous subsection, mainly because of the following caching scenario: Consider comparing an online algorithm that uses LRU with the algorithm that knows the request sequence in advance and can change all its blocks using just one page fault. If k is the memory size, then LRU is k-competitive in this scenario. This is because by the time LRU faults k times, if the benchmark algorithm has yet to incur any page faults, then LRU's memory becomes exactly the same as that of the benchmark algorithm. This fact is what our previous result uses.

However, in our case at hand, we add one more constraint, namely, changing memory size. Consider comparing LRU with a similar algorithm but now with the new constraint that memory size can shrink or grow at any time by any amount. Both the algorithms use the same memory size at all times, but the memory size may be variable. In addition, the benchmark algorithm can load/change as many pages as it wants using just one page fault. The question arises: *Is LRU still k-competitive?* The answer is *no!* Every time the available memory is permanently reduced, LRU can be cheated into making a whole new sequence of page faults. However, the memory can permanently shrink only k times. This makes LRU $O(k^2)$-competitive.

When the caching memory has to be shared with processing, some phases might need more processing memory. In this case, the algorithm may not be able to carry all the blocks it cached to the next phase (i.e., it may have to drop some). In this case, the emphasis of algorithm is still on carrying as many blocks as possible.

However, this doesn't mean that all the advantage is lost. We can now have the threshold size $t = M/(MD/L)^{1/3}$ and achieve an algorithm which is $O((MD/L)^{2/3})$ competitive when the lookahead is $M + L$.

The processing of the phase algorithm tLRU works as follows: Update C_i to $C_i^{'}$ (we call it $C_i^{'}$ here and not C_{i+1}) as in the previous algorithm using prefetching. Order $C_i^{'}$ by LRU as seen at the end of phase i. That is, the block of $C_i^{'}$ whose last access is the latest appears last. Let C_i^L be the subset of $C_i^{'}$ consisting of those t blocks from each disk whose last occurrence is the earliest as seen at the end of phase i. In case a disk has fewer than t blocks in $C_i^{'}$, C_i^L contains all the blocks from that disk. Let $D_i = C_i^{'} - C_i^L$. The blocks in D_i are ordered by their last occurrence in L_i (the latest block first). Suppose that, given the initial memory condition $C_i^{'}$, algorithm A takes y I/Os to process the phase using duality. Then, we define an ordered sequence P_i as the largest prefix of D_i such that the lookahead L_iP_i can be served by A in at most $y + t$ I/Os.

During the servicing part of the phase, we run the duality based algorithm with L_iP_i as lookahead (and also the request sequence) and $C_i^{'}$ as the initial memory condition. The blocks in P_i become part of C_{i+1} at the end of the phase. If $P_i = D_i$, then we do t more I/Os to restore the blocks in C_i^L into C_{i+1}. Else, we carry the t extra blocks from each disk from $C_i^{'} - P_i$ chosen by the LRU order. In this case we call phase i as a *downsizing phase*.

LEMMA 4. *Either $o_i > t$ or $s_i \leq o_i + 3t$.*

PROOF. Same as Lemma 1. From Lemma 1, we have $y \leq o_i + t$ and our service cost here is at most $y + 2t$. □

LEMMA 5. *Consider any sequence of phases $i, i+1, ..., j$, where $x_i + x_{i+1} + .. + x_j \geq M$ and there is no downsizing phase among these where our algorithm is forced to lose some blocks which appear during these phases. Then O must do at least t I/Os over these phases.*

PROOF. Same as lemma 3. Here, $L = L_i \cup L_{i+1} \cup .. \cup L_j$ consists of at least M distinct blocks plus at least t blocks on each disk which was contributing disk during one of these phases. Hence, whatever M_i was, O must see at least t new blocks on one of the disks. □

A downsizing phase is called *effective* downsizing phase if it allows fewer blocks to pass through than the previous effective downsizing phase l or it occurs after more than z pages are accessed as a part of cache (i.e., in some C_i) after the phase l, where z is the number of blocks allowed to pass through by the phase l. Note that algorithm tLRU need note be aware whether a downsizing phase is effective or not. This is only for analysis.

LEMMA 6. *Let i be a downsizing phase allowing z blocks. Let $i + 1, ..., j$ be the sequence of phases where there is no effective downsizing phase and $x_{i+1} + .. + x_j \geq z$. Then, O must do at least t I/Os over the phases $i, .., j$.*

PROOF. If i allows z blocks to pass through, O has no more than z old blocks when it enters phase $i + 1$, unless it does some I/Os during the phase i to carry over the blocks. O has to serve at least z blocks plus t blocks per each of the contributing disks. Hence, O must do at least t I/Os (this includes I/Os in phase i) over these phases. □

Consider a maximal sequence of phases $i, ..., j$ where O does at most $t/2$ I/Os. Let's call this sequence a *superphase*. Now $L = L_i \cup ... \cup L_j$ consists of at most M blocks plus $t/2$ blocks on each disk. There can be at most $2M/t$ contributing disks in this superphase. Hence, there are at most $2M$ blocks altogether in a superphase which occur on any of the contributing disks, and there are at most M block fetches O can do during the superphase using only $t/2$ I/Os involving the blocks on the contributing disks.

During the effective downsizing phase, if A loses some blocks from the current superphase, then we shall show that O must lose at least $t/2$ blocks from the contributing disks of the current superphase. Thus, within a superphase, there cannot be more than $4M/t$ effective downsizing phases. On each subsequence of the superphase where there is no effective downsizing, A can do at most $2M$ prefetching I/Os. Hence, the total prefetch cost of A within a phase is $\leq 8M^2/t$. Again, choosing $t = M/(MD/L)^{1/3}$, the prefetch cost is bounded by $8M(MD/L)^{1/3}$. The cost of O is at least $M/(2(MD/L)^{1/3})$ (i.e., $t/2$) during phases $i, ..., j, j+1$. The cost of O in phase $j+1$ can be counted twice so that it pays for the previous superphase as well as the new superphase, which starts with $j + 1$. This gives a competitive ratio of $O((MD/L)^{2/3})$ on prefetch cost. As seen earlier, the competitive ratio for the service cost is $O(t/(L/D))$, since o_i is at least L/D in any phase. Again choosing $t = M/(MD/L)^{1/3}$, this competitive ratio becomes $O((MD/L)^{2/3})$.

What remains to be proven is that during effective downsizing, where A loses some blocks from the current superphase, O loses at least $t/2$ blocks from the superphase. Consider the processing of the phase. Let $X = L_i \cap C_i^{'}$, $Y = (C_i^{'} - L_i) \cap L$ and $Z = C_i^{'} - L$. We have $C_i^{'} = X \cup Y \cup Z$, and the blocks in $X \cup Y$ appear before the blocks in Z in the reverse LRU order. Since P_i is the largest prefix in the reverse LRU order, it prefers the blocks in $X \cup Y$ over Z, and there is no other subset $W \subseteq (X \cup Y) - C_i^L$ such that $|W| > |P_i|$ and that (W) can be maintained (without ever losing the blocks from W, during the processing of the phase) at the end of the phase using less than t I/Os during the phase. This uses the LRU property of P_i and the fact that if such a W existed then the blocks of W could be exchanged for the blocks occurring later to make $|P_i|$ larger. Ultimately, after restoring t more blocks from each disk, if our algorithm loses a few blocks in $X \cup Y$, then any other algorithm which does less than $t/2$ I/Os (i.e., it cannot restore more than $t/2$ blocks per disk) loses at least $t/2$ blocks in $X \cup Y$ on some disk. This finishes the argument and yields the following theorem.

THEOREM 3. *The competitve ratio of tLRU, with lookahead $L + M$, is $O((MD/L)^{2/3})$.*

In the particular case where lookahead is $2M$, the threshold t becomes $M/D^{1/3}$ and we get the competitive ratio of $O(D^{2/3})$.

3.3.1 Tight Example

In this section, we show that the competitive ratio of threshold-LRU is indeed $\Theta((MD/L)^{2/3})$. We shall begin with the case $L = 2M$. Let the threshold size of the algorithm be t. We shall first describe the sequence where this tight bound occurs. Let us say we have $1 + M/12t + D/2$ disks in total. Disks 1, 2, ..., $M/12t$ contain $6t$ blocks la-

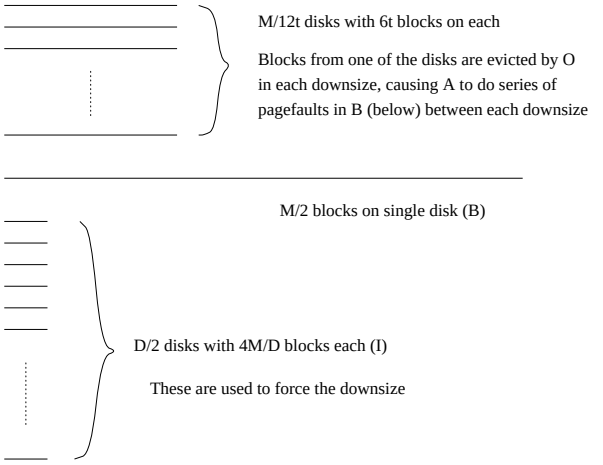

Figure 1: Tight Example

beled $b_{i,j}$ where $1 \leq i \leq M/12t$ and $1 \leq j \leq 6t$. Let B consist of the set of $M/2$ blocks on disk $M/12t+1$ ordered arbitrarily. Let $B(i,j)$ denote the ordered sequence of blocks $i, i+1, .., j$ from B. The remaining $D/2$ disks each contain $4M/D$ blocks ordered in round-robin fashion across the $D/2$ disks. Let's call this set I, the set of idle blocks. We have $|I| = 2M$. We define $I(i,j)$ similarly to $B(i,j)$

Now let's say phase 1 contains L_1 as $I(1,M), B, b_{1,1}, .., b_{1,6t}, ..., b_{M/12t,1}, .., b_{M/12t,6t}$. Then O does at most $M/2 + M/D$ I/Os while A spends $M/2 - t$ as the prefetch cost. A prefetches blocks $B(t+1, M/2), b_{1,t+1}, ..., b_{1,6t}, ..., b_{M/12t,t+1}, .., b_{M/12t,6t}$. So this set is C_1. Let the final memory state M_1 of O be equal to C_1. $|M_1| = |C_1| = 11M/12 - t$.

Let phase 2 consist of $L_2 = I(M + 5t + 1, 2M), B(t + 1, M/2), b_{1,t+1}, .., b_{1,6t}; [b_{2,t+1}, .., b_{M/12t,6t}; I(1, M/12+6t)]^M$. Here, the last part of the sequence in square brackets is repeated M times. This repetitive sequence consist of $M/2+t$ blocks. These blocks necessarily reside in the memory together. Hence, at least $4t$ blocks amongst $B(t+1, M/2), b_{1,t+1}, .., b_{1,6t}$ have to be dropped by O. O drops $b_{1,t+1}, .., b_{1,6t}$ while A following LRU order drops $B(t+1, 6t+1)$. However, since A is allowed t extra I/Os during the processing of the phase and t at the end of the phase to carry over $2t$ blocks, A ends up losing only $2t$ blocks which are $B(t+1, 3t)$.

We define $L_3 = I(M/2+3t+1, .., 2M), B(t+1, 3t), [b_{2,t+1}, .., b_{M/12t,6t}, I(1, M/12 + 6t)]^M$.

Then, by LRU order it does t I/Os, fetches $B(2t+1, 3t)$, and then after processing the phase it carries over $B(4t + 1, 6t)$ and drops $B(3t + 1, 4t)$. In next lookahead these dropped blocks occur along with $B(t+1, 2t)$. Thus, in each subsequent phase, A does t prefetching I/Os while O does at most $4M/D$ I/Os. This will go on till A does $M/2 - 3t$ I/Os in $M/2t - 3$ next phases.

Then, the next phase will again be similar to phase L_2 and cause further downsizing, where O will emit the blocks from disk 2 and the following $M/2t - 3$ phases A will do $M/2 - 3t$ prefetch I/Os. This repeats until we finally have blocks from only B in both C_i and M_i. At this point, O will do $5t$ I/Os and repeat the whole sequence all over again.

Over each run of the sequence A does at least $(M/2 - 3t)(M/12t)$ I/Os in $(M/2t-3+1)(M/10t)$ phases. Hence, O does at most $(M/2t-2)(M/12t)(4M/D)$ I/Os. If A chooses $t < M/D^{1/3}$, O (adversary) can still assume $t = M/D^{1/3}$ and if A chooses higher t its competitive ratio only gets worse. Putting $t = M/D^{1/3}$ we find that A does $\Omega(MD^{1/3})$ I/Os while O does $O(M/D^{1/3})$ I/Os. Thus, on this sequence A is $\Omega(D^{2/3})$ competitive.

When the lookahead is $M + L$, the adversary can choose t as $M/(MD/L)^{1/3}$. This shows that whatever the choice of threshold, algorithm tLRU is $\Omega((MD/L)^{2/3})$ competitive.

3.4 Randomized algorithm against oblivious adversary

In the traditional (single disk) paging models, algorithm MARK [10] (also [7] Sec 4.3) gives the expected competitive ratio of $2H_M$ using randomization against an oblivious adversary. In the parallel disk case, the prefetching lower bound also applies. Hence, we cannot hope to achieve drastic improvement over the deterministic (against adaptive offline adversary) algorithm. However, the effect of downsizing phases to the competitive ratio can be reduced by allowing randomization. The lower bound in this case is still $\Omega(\sqrt{MD/L})$. We shall give an algorithm *threshold* MARK (or tMARK) based on the Marking algorithm of Fiat et al. [10, 7] which achieves a near optimal competitive ratio of $O(\sqrt{(MD/L)\log(MD/L)})$.

The algorithm tMARK works exactly as that in section 3.3, except for the replacement policy. A block becomes marked either whenever it enters the cache (i.e., becomes a part of C_i), or when it is unmarked and is accessed in the current phase. When all the stored blocks are marked, all the marks are erased. This happens when C_i' is created from C_i during prefetching phase; subsequently all pages in C_{i+1} are unmarked. For replacement, we randomly remove from among unmarked blocks. A phase is called *effective downsizing* if it allows fewer blocks to pass through than the previous effective downsizing phase or if there were at least as many marked pages in the cache as allowed by the previous effective downsizing phase sometime between these two phases. Whenever an effective downsizing phase is encountered, all the marks are erased and as many blocks as possible are pushed through such a phase by LRU order. On other downsizing phases, blocks are pushed according to LRU ordering but their marks are left as they are. If all the blocks get marked, the marks erased. Note that, in contrast to tLRU in the previous subsection, tMARK is aware of effective downsizing.

The algorithm chooses $t = \sqrt{(ML/D)\log(MD/L)}$ as the threshold. Let $k = M/t$. Now, in any phase i, during the prefetching phase, if no disk has more than $2t$ blocks in $L_i - C_i$ then the I/Os during this prefetching part (to get C_i') are charged to "free I/Os" (since these are less than t I/Os, these will count as service cost) else these I/Os are charged as caching I/Os (i.e., charged to prefetch cost). Hence, in any phase if caching I/Os are charged, they involve caching at least t blocks. We shall compare our algorithm tMARK with optimal offline algorithm O.

LEMMA 7. *Let $i, i+1, ..., j$ be the sequence of phases such that (1) none of them is downsizing (2) in phase i all the blocks in C_i are unmarked and (3) in phase j all the blocks in C_j' get marked. Then, O does at least t I/Os over these phases.*

PROOF. In C_i no block is marked. The blocks appearing in $L_i \cup ... \cup L_j$ are marked as they are accessed. Since there

is no downsizing phase, no mark gets deleted until all the blocks are marked. No marked block is evicted until this point. Let $l \leq j$ be the first phase where all the blocks get marked. Then, $L_i \cup .. \cup L_l$ had M blocks plus at least t blocks on each disk. Thus, similar to lemma 5, O must do at least t I/Os during these sequence of phases. □

LEMMA 8. *Let $i, i+1, ..., j$ be the sequence of phases such that (1) i is effective downsizing allowing z blocks to pass through (2) none of the other phases are effective downsizing (3) in phase j at least z blocks in $C_j^{'}$ get marked. Then O must do at least t I/Os over these phases.*

PROOF. Similar to lemmas 6 and 7. □

For any set of blocks S, we define (similar to Section 3.1) $\Phi(S) = \sum_{d=1}^{D} \max(S^d - t, 0)$, where S^d is the number of blocks in S which belong to disk d. A block which enters a cache, i.e., becomes a part of some $C_l, i \leq j$, is called *new* over the phases $i, .., j$ if it was not a part of C_i.

LEMMA 9. *Let $i, i+1, ..., j$ be the sequence phases such that there is no effective downsizing phase among them and all blocks C_i are unmarked. If $\Phi(L_i \cup .. \cup L_j) \leq |C_i|$ and $\Phi(L_i \cup .. \cup L_j - C_i) \leq p$ then the expected number of caching I/Os during this sequence of phases is $\leq O(p \ln k)$.*

PROOF. If $\Phi(L_i \cup ... \cup L_j - C_i) \leq p$ then there are at most p new blocks that can enter the cache of tMARK during these phases and at most p blocks in C_i are useless. Let $x \leq p$ be the number of new blocks entering the cache. The worst case for tMARK comes when all the requests to these x blocks precede the request to other $|C_i| - x$ blocks. Hence, we may delete some of the useful blocks from C_i. Following analysis of [7], the access to the jth block (in order they are first accessed) in $C_i \cap (L_i \cup .. \cup L_j)$ incurs fault with probability $x/(|C_i| - j + 1)$. Summing this up, the expected number of faults is at most $x(H_M - H_x + 1)$ where H_l is lth harmonic number. If $x \leq t$ then none of these faults are charged as caching I/Os. Hence, number of faults charged on caching I/Os is no more than $x(H_M - H_t + 1)$. This is less than $O(p(\ln(M/t) + 1))$ which is $O(p \ln k)$. □

THEOREM 4. *The competitive ratio of tMARK, with lookahead $M + L$, is $O(\sqrt{(MD/L)\log(MD/L)})$.*

PROOF. Consider a maximal sequence of phases $i, ..., j$ which form a superphase. That is O can do at most $t/2$ I/Os during these phases, regardless of the contents of its initial memory M_i. During each downsizing phase since our algorithm pushes as many blocks as possible in the LRU order, the only way O can have pushed more blocks through the downsizing phase is by spending I/Os, to carry the blocks forward. Since O does less than $t/2$ I/Os during these phases $\Phi(L_i \cup ... \cup L_j) \leq M$. Now, if there was a downsizing phase l during the superphase and if tMARK lost x blocks, then O must also lose x blocks unless it replaces some of the lost blocks by doing I/Os. In this case, it will have those many less I/Os allowed during the superphase. Hence, $\Phi(L_l \cup .. \cup L_j) - \Phi(L_{l+1} \cup .. \cup L_j) \geq x$. That is if we lost x blocks, the Φ value of remaining lookahead during the superphase also decreases by at least x.

Let phases $d_1 < ... < d_s$ be s effective downsizing phases during this superphase where number of blocks allowed through the phase d_l is less than those allowed during d_{l-1} for any l between 2 and s. Let $z_1, ..., z_s$ be the number of blocks lost during these downsizing phases. Let $p_1, p_2, ..., p_{s+1}$ be the number of *new* requests served by the tMARK during phases $(i..d_1), (d_1 + 1..d_2), ..., (d_s..j)$ respectively. Let P be the maximum number of new requests tMARK has to serve during the superphase in the worst case assuming no downsizing during the superphase. Then, if tMARK serves p_1 new request during phases $(i..d_1)$ the expected caching cost is $O(p_1 \ln k)$ (by lemma 9), for the part of superphase consisting of phases $d_1 + 1, ..., j$ the maximum number new requests can be $P - p_1 + z_1$. Summing this up we get, $p_{s+1} \leq P - \sum_{l=1}^{s} p_l + \sum_{l=1}^{s} z_s$. This means, the total expected caching cost $O(\ln k \sum_{l=1}^{s+1} p_l)$ is at most $O(\ln k(P + \sum_{l=1}^{s} z_l))$. Since both P and summation of x_l's are bounded above by M, we get that total expected caching cost paid by tMARK during the superphase is no more than $O(M \ln k)$.

Now, O pays at least $t/2$ I/Os during the phases $i, .., j+1$ (this will count cost of O for phase $j+1$ twice which is allowed in asymptotic sense). Hence, the competitive ratio is the maximum of $O((M/t)\log k)$ and $t/(M/D)$ (due to service cost). Recall that $k = M/t$ and by choosing $t = \sqrt{(ML/D)\log(MD/L)}$ we get the competitive ratio of $O(\sqrt{(MD/L)\log(MD/L)})$. □

4. CONCLUSIONS AND OPEN PROBLEMS

Ours is the first set of online algorithms that achieve good parallelism for the problem of caching/prefetching on parallel disks. This has remained elusive for a long time. There are many issues related to this model that remain open: The first is to close the gap between the lower and upper bounds. The second is to extend this result for lower ranges of lookahead. We discussed earlier that there is no hope for better results if lookahead is less than $M - D$. But this holds only for the deterministic case. We can hope that randomization will help. However, the main difficulty of thresholding remains. Any algorithm which falls under the *marking* category (see [7]) will not work without something like thresholding. In the lower set of lookaheads, fewer free I/Os are allowed and they may not sufficiently pay for thresholding. Ideas like adaptive thresholding (i.e., threshold as much as can be paid by free I/Os) are worth exploring. The third is to make the algorithms more practical. Many constant factors in our algorithms are purely for analysis and ease of presentation. In practice, we do not need separate prefetching and servicing phases. The algorithm can also be made to work with a sliding window of lookahead using a priority based mechanism. Finally, thresholding (i.e., purposely throwing away blocks from cache) doesn't have to be done explicitly. We can still keep track of thresholded blocks in the cache and give them lower priority. We believe that the results herein form an essential stepping stone for further progress (practical as well as theoretical) on this problem.

5. REFERENCES

[1] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18(3):283–305, 1997.

[2] S. Albers and M. Büttner. Integrated prefetching and caching in single and parallel disk systems. In *SPAA*, pages 109–117, 2003.

[3] S. Albers, N. Garg, and S. Leonardi. Minimizing stall time in single and parallel disk systems. In *In Proc. of*

30th Annual ACM Symp. on Theory of Computing (STOC 98), pages 454–462, 1998.

[4] S. Albers and C. Witt. Minimizing stall time in single and parallel disk systems using multicommodity network flows. In *RANDOM-APPROX*, 2001.

[5] R. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive parallel disk prefetching and buffer management. In *In Proc. of Fifth Workshop on I/O in parallel and Distributed Systems*, pages 47–56, Nov 1997.

[6] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

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

[8] D. Breslauer. On competitive online paging with lookahead. *TCS*, 290(1-2):365–375, 1998.

[9] P. Cao, E. W. Felton, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *In Proc. of the joint Intl. Conf. on measurement and modeling of computer systems*, pages 188–197, May 1995.

[10] A. Fiat, R. Karp, M. Luby, L. McGoech, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, Dec 1991.

[11] D. A. Hutchinson, P. Sanders, and J. S. Vitter. Duality between prefetching and queued writing with application to integrated caching and prefetching and to external sorting. In *ESA*, 2001.

[12] M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *In Proc. of Sixth ACM Workshop on I/O in Parallel and Distributed Systems*, pages 68–77, 1999.

[13] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel i/o systems. In *SPAA*, 2001.

[14] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[15] T. Kimbrel, P. Cao, E.W. Felten, A. R. Karlin, and K. Li. Integrated parallel prefetching and caching. In *SIGMETRICS*, 1996.

[16] T. Kimbrel and A. R. Karlin. Near optimal parallel prefetching and caching. In *FOCS*, pages 540–549, 1996.

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

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

[19] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys*, 33(2):209–271, June 2001.

[20] N. Young. Competitive paging and dual-guided on-line weighted caching and matching algorithms. In *Ph.D. thesis*. Princeton University, 1991. CS-TR-348-91.