Tight Competitive Ratios for Parallel Disk Prefetching and Caching[*]

Wing-Kai Hon
Dept. of Computer Science
National Tsing-Hua University
wkhon@cs.nthu.edu.tw

Rahul Shah
Dept. of Computer Science
Louisiana State University
rahul@cs.purdue.edu

Peter J. Varman
Dept. of ECE
Rice University
pjv@rice.edu

Jeffrey Scott Vitter
College of Science
Purdue University
jsv@purdue.edu

ABSTRACT

We consider the natural extension of the well-known single disk caching problem to the parallel disk I/O model (PDM) [17]. The main challenge is to achieve as much parallelism as possible and avoid I/O bottlenecks. We are given a fast memory (cache) of size M memory blocks along with a request sequence $\Sigma = (b_1, b_2, ..., b_n)$ where each block b_i resides on one of D disks. In each parallel I/O step, at most one block from each disk can be fetched. The task is to serve Σ in the minimum number of parallel I/Os. Thus, each I/O is analogous to a page fault. The difference here is that during each page fault, up to D blocks can be brought into memory, as long as all of the new blocks entering the memory reside on different disks. The problem has a long history [18, 12, 13, 26]. Note that this problem is non-trivial even if all requests in Σ are unique. This restricted version is called *read-once*. Despite the progress in the offline version [13, 15] and read-once version [12], the general online problem still remained open. Here, we provide comprehensive results with a full general solution for the problem with *asymptotically* tight competitive ratios.

To exploit parallelism, any parallel disk algorithm needs a certain amount of lookahead into future requests. To provide effective caching, an online algorithm must achieve $o(D)$ competitive ratio. We show a lower bound that states, for lookahead $L \leq M$, any online algorithm must be $\Omega(D)$-competitive. For lookahead L greater than $M(1 + 1/\epsilon)$, where ϵ is a constant, the tight upper bound of $O(\sqrt{MD/L})$ on competitive ratio is achieved by our algorithm SKEW. The previous algorithm tLRU [26] was $O((MD/L)^{2/3})$ competitive and this was also shown to be tight [26] for an LRU-based strategy. We achieve the tight ratio using a fairly different strategy than LRU. We also show tight results for randomized algorithms against oblivious adversary and give an algorithm achieving better bounds in the resource augmentation model.

[*]Supported by NSF CCF–0621457, NSF CNS–0541369.

Categories and Subject Descriptors

F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous

General Terms

Algorithms

Keywords

Online algorithms,Competitive Analysis,Parallel Disk Model

1. INTRODUCTION

Parallel disks provide a cost-effective way of speeding up I/O performance in applications that use massive amounts of data. The main challenge is to achieve as much parallelism as possible and avoid I/O bottlenecks. The parallel disk model (PDM) [17] is a popular model for measuring the I/O complexity of problems when data are allowed to be on multiple disks. In each parallel I/O step, one block from each disk can be read (or written) simultaneously. That is, if D is the number of disks, then one I/O step can read or write up to D blocks, as long as they fall on different disks. The objective is to perform a given task in the minimum number of parallel I/O steps. As an application designer, the task is to arrange and access data across the disks so as to have as much parallelism and avoid bottleneck situations where one may need to access many blocks from the same disk. Thus, the designer has the freedom to place the data on appropriate disks. At the systems level, the task of I/O controller is to schedule I/Os and maximize throughput. The fundamental problem in this model is caching/prefetching: Given an ordered sequence $\Sigma = (b_1, b_2, ..., b_n)$ 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. Each block in the request sequence Σ has an associated disk number in $\{1, .., D\}$. To serve the read request for a particular block b_i, the block must be in main memory. If it is not present, an I/O must be done to fetch it. During this I/O step, we might choose to prefetch some blocks from other disks so that when they are requested in future they are already in memory. Minimizing I/Os in such a way using parallelism and some future knowledge is called *prefetching*. After the block's read request is served, we may still want to keep the block in the memory in case any future request comes for that block. Since the buffer is of limited size, determining which blocks to keep in memory and which to throw away so as to make room for new requests is called the *caching* problem. A restricted case of the problem where all the requests in Σ are unique is called *read-once*. This version only involves prefetching. When the request sequence is known in advance, the problem is *offline* and then the objective is to compute the minimum I/O schedule. On the other hand, when there is no future knowledge or limited future knowledge, the problem is *online* and then the objective is to achieve competitive performance as compared to the best offline algorithm.

Prefetching is needed to take advantage of parallelism, and 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 next L blocks in the request sequence. However, for the general problem, an adversary can nullify the advantage of lookahead by repeating the same block L times consecutively. In this case, the lookahead does not provide much information. To overcome this problem, many different definitions of lookaheads are considered in the (single-disk) paging literature [6, 21, 2]. We shall assume the definition provided by [6] 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.

1.1 Previous work and our results

Single disk paging algorithms have a long history in computer science. For the offline problem, Belady's MIN replacement policy [27] achieves the minimum number of page faults. In the online problem, the algorithm needs to achieve the best competitive ratio. Some fundamental results were given by [1, 4, 7, 27]. The variants which use lookahead [6] and use extra memory [21, 1] have also been considered. The adaptation of LRU with lookahead $LRU(L)$ achieves the competitive ratio of $M-L$ and the adaptation of MARK, $MARK(L)$ achieves competitive ratio of $2H_{M-L}$ [6] (Note $H_k = \sum_{i=1}^{k} 1/i$). When the online algorithm is given K extra blocks of memory, the competitive ratio of LRU (without lookahead) becomes $(M+K)/(1+K)$.

The parallel disk generalization of the problem also has a long history [18, 12, 13, 15, 26]. This was first formulated in [18]. They gave $\Theta(\sqrt{D})$ bounds for the read-once version when lookahead $L = M$. Later, [12] generalized this to get a tight competitive ratio for all ranges of lookahead for the read-once version. For the general problem of caching/prefetching on parallel disk, the optimum offline algorithm was given by [15, 13]. The first online solution was given by [26] based on *thresholding*. However, the competitive ratio was not tight against the lower bound. Also, it only considered the lookahead range of $L > M(1 + 1/\epsilon)$. As noted by [26], the algorithm tLRU cannot achieve a competitive ratio better than $O((MD/L)^{2/3})$. The main difficulty is to devise a caching strategy that favors the blocks from the bottleneck disks to be kept in the cache. However, the notion of bottleneck disks was not captured very accurately. Here, we devise the analytic structures like potential function to capture this notion and devise a caching policy in accordance with this potential function. We also obtain the new lower bounds and tight upper bounds for other ranges of lookahead: $L = M$ and $L < M$. For the $L = M$ case, although our lower bound shows that the competitive ratio cannot be better than $\Omega(D)$, we show that if the online algorithm is allowed extra memory, competitive ratio of $O(\sqrt{D})$ can be achieved. Table 1 shows the spectrum of results on this problem, previous as well as current.

A similar problem has also been considered by Karlin et al. [5, 22, 25] and Albers et al. [9, 23, 24] in stall-model which is more general than PDM. In this model, 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 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 work is focussed on approximation algorithms for the offline problem. Some of the strategies used for the offline algorithm are similar to those used in PDM model. In stall-model this problem is hard even in offline settings while for PDM this can be optimally solved. Our online results (especially the lower bounds) on PDM will have implication for this more general model as well. Here is the summary of our new results in this paper:

- We present a deterministic algorithm SKEW which is the first algorithm to achieve the tight competitive ratio of $O(\sqrt{MD/L})$ for lookahead $L > M(1 + 1/\epsilon)$. This improves upon the previous tLRU which was $O((MD/L)^{2/3})$ competitive. This is tight against the lower bound and thus we close the gap.

- For lookahead $L < M - D$, we show a lower bound of $\Omega(D)$ for the competitive ratios of randomized algorithms, as well as those with extra memory. We show upper and lower bounds of $\Theta(D \log(1 + (M - L)/D))$ for randomized algorithms against an oblivious adversary.

- For lookahead $L = M$, we show a lower bound of $\Omega(D)$. This indicates that simple LRU or Supervisor [13] is tight. We also show that when the online algorithm is allowed to have $2M$ extra memory than that of the offline adversary, competitive ratio of $O(\sqrt{D})$ can be achieved.

In Section 2, we re-cap some previous results that will be building blocks for our algorithms in this paper. Section 3 shows bounds for $L < M$ cases and also shows a randomized upper bound. In Section 4, we give the algorithm for $L = M$ when an online algorithm is allowed to have $3M$ memory as compared to M for the offline adversary. We also give a lower bound of $\Omega(D)$ when no extra memory is allowed. In Section 5, we present our main algorithm SKEW. We conclude in Section 6.

2. PRELIMINARIES

Each block in the request sequence Σ has an associated disk number in $\{1, .., D\}$. At the point when request for

Lookahead	Results (Competitive Ratio)	paper
$L = \infty$	optimal offline	duality[15], Supervisor[13]
$L \geq M(1+1/\epsilon)$	$\Omega(\sqrt{MD/L})$	[12]
	$O(MD/L)$	Supervisor[13]
	$O((MD/L)^{2/3})$	tLRU [26]
	$\mathbf{O}(\sqrt{\mathbf{MD/L}})$	SKEW [this paper]
$L = M$	$\mathbf{\Omega(D)}$	[this paper]
	$O(D)$	LRU, Supervisor [13]
$L = M$, $3M$ online memory	$\Omega(\sqrt{D})$	adaptation of [18]
	$\mathbf{O}(\sqrt{\mathbf{D}})$	[this paper]
$L < M - D$	$\Omega(M-L)$	adaptation of [6]
$L < M - D$, with extra memory	$\mathbf{\Omega(D)}$	[this paper]
$L < M - D$, randomized	$\mathbf{\Theta(D\log(1+(M-L)/D))}$	[this paper]

Table 1: Comparison of results

block b_i is served, the algorithm can see the strong lookahead of size L which consist of blocks $(b_{i+1}, b_{i+2}, ..., b_k)$ where k is the largest index such that $(b_{i+1}, b_{i+2}, .., b_k)$ consists of at most L distinct blocks. Lookahead gives information to the algorithm, not only about which block to maintain in fast memory (caching) while making room for the new pages which come, but also about which blocks to prefetch into the fast memory using parallel I/Os before they are ready to be accessed.

The competitive ratio in general assumes that both offline and online algorithms have same amount of fast memory M. We shall also consider the case when online algorithm is allowed twice (or thrice) as much memory and we shall achieve tight competitive ratios in terms of complexity. Our algorithms and analyses are *phase-wise*. A *phase* consists of contiguous subsequence of Σ such that the subsequent phase consists of lookahead sequence at the end of previous phase. Our algorithms will use the lookahead in only phase-wise sense (rather than in sliding window sense).

2.1 Optimum offline algorithm

We shall use the optimum offline algorithm for the problem given by duality principle [15] as a building block. The algorithm works by looking Σ in the reverse order Σ^R and treating this as a write problem where blocks are issued into fast memory to be written on their respective disks; at most one block can be written to each disk in an I/O step; also, the latest instance of any block has to be available in either fast memory or on the disk. The solution is: in each I/O step, write to as many disk as possible and for a particular disk write that block whose next request in Σ^R is the latest. This algorithm when seen as a reverse process gives optimum offline read (caching) algorithm. Given an initial fast memory block set S, this algorithm can also be modified to produce optimum read schedule by generating a write schedule for $\Sigma^R S$ with an additional liberty that blocks in S need not be written on the disk. We shall use this algorithm, separately in every phase.

2.2 Lower bound of $\Omega(\sqrt{D})$

Since read-once is a particular case of general read-many, the lower bounds on the competitive ratio for the read-once case also apply to our general case [18]. 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 following: 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 [18]. 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 the lower bound $\Omega(\sqrt{MD/L})$ on the competitive ratio [12].

2.3 Request sequence where LRU does bad

When designing single disk online algorithms with lookahead, LRU turns out to be the best deterministic policy [6]. In the case of parallel disks, however, this is not exactly true. Consider the following example [19]: 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 $O(D)$. We need to somehow avoid the blocks from disks containing very few blocks from being stored in the cache memory so that cache can be effectively used to store blocks from disks which can create I/O bottlenecks. One simple strategy is *thresholding* [26] where during processing of the phase, only the blocks in excess of threshold t from each disk are stored. That is up to t blocks in the phase from each disk are not required to be in caching storage, when the phase processing is over. We shall see this further in Section 2.4.

2.4 An online algorithm with $2M$ lookahead and $2M$ memory

Here, we shall describe the algorithm which uses twice as much memory as offline optimum[1] and achieves the com-

[1]This is in the spirit of the resource-augmentation model. Our algorithm in Sec 4.2 also uses resource-augmentation.

petitive ratio of $O(\sqrt{D})$. This algorithm uses thresholding technique of [26] and forms a basic building block for our algorithms in Section 4.2 and in Section 5. Our algorithm A has a memory buffer of size $2M$ and it manages it in two components: a processing 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. For simplicity, we shall assume that the lookahead for online algorithm A is $2M$. This can be generalized to case where lookahead is $M + L$ to yield $O(\sqrt{MD/L})$ competitive ratio. We use a phase-wise (recall that phase consists of consequent sequence of request given by lookahead at the start of the phase) approach, so our algorithm only uses the lookahead as it exists at the beginning of each phase. Note that since lookahead is $2M$, the optimum offline algorithms at least does M/D I/Os during the phase. This algorithm exploits the fact that in every phase, it can allow $O(M/\sqrt{D})$ I/Os which are accounted by M/D I/Os that O does in the phase.

Superphase.
The algorithm is history-aware and can run the optimum offline algorithm on the history of the sequence seen so far to have an estimate of how many I/Os the optimum offline algorithm O has done. The algorithm works in superphases. Superphase r consists of contiguous sequence of phases, following the last phase in superphase $r - 1$. The superphase r ends at phase k such that k is a minimum number for which request sequence in phase $1, 2, .., k$ requires at least $rM/\sqrt{D}$ I/Os. At the end of each superphase, the algorithm flushes all the blocks in its caching storage C.

Notation.
Let the current superphase consist of phases $j, j + 1, ..., k$. Let L_j denote lookahead for phase j. Consider a set $Q_{a,b}$ of disk blocks from request sequence in phases $L_a \cup L_{a+1} \cup \ldots \cup L_b$. For any ordered request sequence Σ, let $\Sigma|^t$ denote the set of blocks consisting of t least recently used blocks on each disk as seen from the end of Σ. We call this the *carry set* of Σ. Let $\Sigma|_t = \Sigma - \Sigma|^t$. Let width of Σ be maximum number of blocks occurring on any disk in Σ. Now, our algorithm A maintains $Q_{j,l}|_{M/\sqrt{D}}$ in its caching memory C at the end of phase l. Note that since O does less than $M/\sqrt{D}$ I/Os over the phases $j, j + 1, .., k - 1$, the size of $Q_{j,k-1}|_{M/\sqrt{D}}$ is less than M and always fits in C.

Phase processing.
During processing of the phase l for $l < k$, A first updates C (looking at lookahead for the phase) to contain exactly $Q_{j,l}|_{M/\sqrt{D}}$. Then, while processing the phase, it uses its memory space P. It uses optimum offline algorithm over $L_l - (Q_{j,l}|_{M/\sqrt{D}})$. All the requests for blocks in $Q_{j,l}|_{M/\sqrt{D}}$ can be served from C. For processing phase k, the algorithm A flushes the cache memory C and uses only P to run the optimal offline algorithm on L_k.

Analysis.
Let C_i be the cache of A at the end of phase i. Let *prefetch cost* be I/O cost of getting any block in C_i for the first time in cache memory C during the superphase. Let the cost of getting any block in C_i for the second time onwards (from carry set) be called *swapping cost.* Let the cost of processing phase using P be called *processing cost.* The total prefetch cost over all the phases in the superphase is at most M. The swapping is from the carry set whose width is no more than $M/\sqrt{D}$ in each phase. So swapping cost is bounded by $M/\sqrt{D}$. Once C is updated, the processing is no more than $2M/\sqrt{D}$ I/Os because O can process the phase in less than $M/\sqrt{D}$ I/Os and A can simulate initial memory condition of O within no more than $M/\sqrt{D}$ I/Os. The sum of the swapping cost and the processing cost during each phase is no more than $3M/\sqrt{D}$ I/Os. The offline optimum O does at least M/D I/Os in each phase and does at least $M/\sqrt{D}$ I/Os over the superphase. The cost of processing the last phase k is no more than M plus the cost of O for the phase k. Thus, A is $O(\sqrt{D})$ competitive. This can be generalized to the case where lookahead is L to achieve the competitive ratio of $O(\sqrt{MD/(L - M)})$ by adjusting the width of the carry set to be no more than $\sqrt{M(L - M)/D}$ and redefining the superphase to make O do at least $O(\sqrt{M(L - M)/D})$ I/Os.

THEOREM 1. *The online algorithm A which uses $2M$ memory and $2M$ lookahead is $O(\sqrt{D})$ competitive when compared with the optimum offline algorithm having memory size M. When the lookahead is $L > M$ this competitive ratio becomes $O(\sqrt{MD/(L - M)})$.* □

3. NEW RESULTS FOR $L < M$

A simple lower bound.
First, we show lower bounds for online algorithms with lookahead $M - D + 1$. We shall consider randomized algorithms as well as algorithms that are allowed extra memory. Let S be any set of $M - D$ blocks. Let R be an infinite sequence of all distinct blocks that is striped equally on all D disks. That is, the the ith block R_i in R is from the disk i mod D. Now the request sequence consists of S alternating with a block from R. That is, $\Sigma = SR_0SR_1SR_2...$. The ith phase has lookahead SR_i. Any online algorithm will have to do at least one I/O in every phase. It has no way of knowing what block R_i is in advance. The optimal offline algorithm will do one I/O every D phases. It maintains set S in memory and fetches next D blocks from R simultaneously. Thus, regardless of the power online algorithm may have due to randomization or extra memory, any lookahead less than $M - D$ can be effectively reduced to no lookahead by the adversary. Thus, if the lookahead is bounded by $M - D + 1$, we have a lower bound of $\Omega(D)$ on the competitive ratio.

Randomized upper-bound.
Here, we show that the adaptation of the MARK(L) algorithm [6] is $O(D \log(1 + (M - L)/D)$ competitive for $L < M - D$. The caching strategy is exactly as in [6]. Note that the definition of phase in this and next subsection is as defined by the marking algorithm [7]. Thus, the algorithm works in phases as defined by marking algorithms. At the beginning of each phase it evicts as many unmarked pages as are new in the lookahead. After this point, it exactly mimics marking[7] algorithm.

THEOREM 2. *The algorithm MARK(L) is $O(D \log(1 + (M - L)/D)$ competitive.*

PROOF. The proof is almost the same as in [6], except for the parallel disk implications. The intuition of the proof hinges on the fact that if during the phase an adversary has loaded k clean (new) pages, the amortized competitive ratio of marking during the phase is $2H_{M/k}$ and subsequently of MARK(L) is $2H_{(M-L)/k}$.

The phase ends when there are no unmarked pages to remove and new phase begins with all unmarked pages. Consider an arbitrary phase. A page is called *stale* if it is unmarked but was marked in the previous phase, and *clean* if it is neither stale nor marked. Let c be number of clean pages and s be number of stale pages requested in the phase. Note that $c + s = M$. Accounting for parallelism, OPT has an amortized cost of at least $\lceil c/2D \rceil$ during the phase. For bounding the expected cost of MARK(L), let s_1 be the number of stale pages contained in the lookahead at the beginning of the phase and let $s_2 = s - s_1$. Let c_1 be the number of clean pages in the lookahead and $c_2 = c - c_1$. Then, following the analysis of [6], the expected cost is equal to

$$\frac{c}{M-s_1} + \frac{c}{M-s_1-1} + .. + \frac{c}{M-s_1-s_2+1} = c(H_{M-s_1} - H_c).$$

Note that $c - L + s_1 = c_2$, i.e., $M - s_1 = M - L + c_1 \leq M - L + c$. If $c > D$, this is bounded by $c(H_{M-L+D} - H_D)$, giving the desired competitive ratio. Else, the cost is bounded by $D(H_{M-L+D} - H_D)$ and this also gives the desired competitive ratio. □

Randomized lower-bound.

Here, we present a lower bound of $\Omega(D \log(1 + (M - L)/D))$ on the competitive ratio for any randomized paging algorithms. The result is captured in the following theorem, and whose proof is very similar to that presented by [7].

THEOREM 3. *Let A be any randomized paging algorithm for D disks with memory size of M and lookahead of L pages. Each disk has at least one page, and we assume that the total number of pages in the disks is at least $M + D$. Then, the competitive ratio of A is $\Omega(DH_{1+(M-L)/D})$ against an oblivious adversary.*

PROOF. Let N be the total number of pages in all disks. We show an oblivious adversary can construct a sequence that will force the claimed competitive ratio on A. For each $j = 1, 2, \ldots, N$, the adversary maintains the probability p_j that the jth page is not in A's memory. The calculation of p_j is possible, since adversary knows the probability distribution used by A.

Our sequence of requests for A is composed of an arbitrarily large number of phases. A phase is defined as follows: phase 0 is the empty sequence; for every $i \geq 1$, phase i is the maximal sequence following phase $i-1$ that contains at most M distinct page requests. Our request sequence will fix a particular $L - 1$ pages in the memory at the end of the previous phase, so that the first $L - 1$ requests in every lookahead in the current phase are the same. Then, the expected cost for A to load the jth page (when seen as the Lth request in the lookahead) is p_j.

Let $L' = L-1$. Each phase will be divided into $k = M-L'$ subphases. The purpose of each subphase is to fix a new page in the request sequence. We apply similar strategy as in Theorem 4.4 in [27], so that at the ith subphase, the expected cost of A is at least $(N - M)/(N - L' - i + 1)$. Thus, the cost of one phase is at least

$$\begin{aligned} &\frac{N-M}{N-L'} + \frac{N-M}{N-L'-1} + \cdots + \frac{N-M}{N-M+1} \\ &\geq \frac{D}{M+D-L'} + \frac{D}{M+D-L'-1} + \cdots + \frac{D}{D+1} \\ &\qquad = \Theta(DH_{1+(M-L)/D}). \end{aligned}$$

On the other hand, the cost of the oblivious adversary is one I/O in each phase. Thus, the competitive ratio follows. □

4. NEW RESULTS FOR $L = M$

4.1 Lower Bound

Here, we present our lower bound of $\Omega(D)$ on the competitive ratio of any online algorithms that uses memory of size M (same as adversary). This bound works even for lookahead as large as $M + cM/D$ for any constant c. We show that this bound also similarly holds for the competitive ratio of any randomized algorithm against the oblivious adversary.

We fix an arbitrarily chosen *useful set* U of $3M/2$ blocks with $3M/2D$ blocks on each of the disks. Our request sequence will consists of phases. A superphase consists of $2D$ consecutive phases. At the end of each superphase the adversary O has M/D blocks on each disk. In each phase, the request sequence consists of some requests from U and possibly some others from junk set J. The junk set consists of blocks totally striped across all disks and consists of possibly infinite number of blocks that never repeat in the request sequence.

Within a superphase, the phases are classified into alternating odd and even phases (D of each). The odd phases are downsizing phases, while the even phases are corrective phases. Let M_i be the memory of O and A_i be that of A before the start of phase i. In odd phase $2i - 1$, the request sequence consists of: (1) all the blocks of M_{2i-1} except those from disk d_i that were present in M_1, (2) $M/2D$ equally striped blocks from $U - M_1$ and (3) $M/2D$ equally striped blocks from J. The first two components of the lookahead (request sequence) are repeated many (possibly infinite) times followed by the junk requests once. Thus, to process this phase, any algorithm must have thrown away at least $M/2D$ blocks from disk d_i that were present in M_1. O throws away blocks after looking at A's choices. It keeps the blocks A has thrown away and shows them in the next phase. Although A has a sliding window lookahead, it has still no way of knowing the blocks which are going to appear in the next phase before it throws. In the even phase $2i$, the request sequence consist of blocks in M_{2i} repeated many times. Thus, in this phase A will have to correct its thrown away blocks from the previous phase. Note that $A_{2i+1} = M_{2i+1} = M_{2i}$. At the end of $2D$ phases, the superphase ends. The memory of O again consists of M blocks from U, with M/D from each disk. Thus, a new superphase can begin.

Now let's analyze the I/O cost for O and A. In the odd phase $2i-1$, O incurs M/D^2 I/Os for the new blocks coming in. A also incurs the same cost. In the even phase O pays no cost, while A incurs $M/2D$ I/Os. Thus summing over all phases, O's cost is M/D I/Os while A's cost is $M/D + M/2$ I/Os. Thus, the competitive ratio of A is at least $D/2$

which is $\Omega(D)$. In the case of randomized algorithms against the oblivious adversary, A's expected cost in even phases is $M/4D$ I/Os, which still gives the lower bound of $\Omega(D)$. Thus, we get the following theorem:

THEOREM 4. *When the lookahead for an online algorithm is the same as the size of the cache blocks, the competitive ratio of any deterministic online algorithm is lower bounded by $\Omega(D)$. The same lower bound holds for any randomized online algorithm against the oblivious adversary.* □

4.2 Online algorithm with M lookahead and $3M$ memory

This algorithm works over the algorithm in Section 2.4 as follows: Allocate $2M$ memory for the the processing part P and M for the caching part C. Collect up to $2M$ distinct blocks in P, then consider it as a single phase and update C as in the algorithm in Section 2.4. The main idea here comes from combining thresholding with the analysis of [12].

Subsuperphase.

Within each superphase (as defined in Section 2.4) we define subsuperphases. Again, subsuperphases form a consecutive sequence of phases, the next starting at the end of previous subsuperphase. The sequence of phases $i, .., j$ forms a subsuperphase if $Q_{i,j-1}$ has less than $2M$ distinct blocks and $Q_{i,j}$ has at least $2M$ distinct blocks.

Phase Processing.

For phases $i, .., j-1$ in the subsuperphase, collect all the blocks in P. At phase j, treat $Q_{i,j}$ as the lookahead and use the algorithm in Section 2.4 to update C.

Analysis.

During processing, the blocks maintained in C and P incur no cost. Again, we shall prove $O(\sqrt{D})$ competitive ratio. The only additional cost over the algorithm in Section 2.4 is when the block enters freshly in P. Now within a subsuperphase, consider subsubsuperphase (sssphase for short) as contiguous collection of $\sqrt{D}$ phases. Let set O_1 be the set of new requests appearing in the current sssphase. Let O_2 be the subset of O_1 that was fetched by O in sssphases before (not during) the previous one. Then, to carry O_2 across the previous sssphase ($\sqrt{D}$ phases) without it appearing in previous sssphase, O pays at least $|O_2|/D$ I/Os every phase (because lookahead is M and $|O_2|$ blocks from memory of O are occupied) and thus it has paid the I/O cost of at least $|O_2|/\sqrt{D}$ during the previous sssphase. Thus cost of fetching O_2 for A during this sssphase is justified. The cost for the remaining subset $O_1 - O_2$ is paid by A over $\sqrt{D}$ phases. In each of these phases A never pays more than optimal cost required to fetch $O_1 - O_2$. Thus, this cost is no more than $\sqrt{D}$ times the cost O paid to acquire these during this and previous sssphase. Thus, the cost of A during this sssphase is at most $2\sqrt{D}$ times the cost of O during this and previous sssphase.

THEOREM 5. *The online algorithm A that uses $3M$ memory and M lookahead is $O(\sqrt{D})$-competitive when compared with the optimum offline algorithm having memory size M.*

□

5. ONLINE ALGORITHM SKEW

In this section, we describe our main algorithm that achieves the competitive ratio of $O(\sqrt{MD/(L-M)})$ for the lookahead size L. Unlike earlier algorithms, this algorithm uses no extra memory; and hence, it is a real competitive algorithm. Note that this bound is asymptotically tight for the ranges of lookahead greater than $(1+1/\epsilon)M$ for any constant $\epsilon > 0$. Again, we shall describe it for lookahead $L = 2M$ for simplicity and achieve $O(\sqrt{D})$ as the competitive ratio. The parameters of the algorithm can be adjusted appropriately for more general lookahead. We shall use the basic terminology and framework from Section 2.4 to derive our algorithm here.

Downsizing.

Since we do not assume separate caching and processing memory here (like Section 2.4), we may not be able to carry all the blocks in the caching part as we would do in the previous algorithms. Consider a set S consisting of M/D blocks on D disks and assume that this set S appears repeatedly (infinitely) many times within a given lookahead. Then, during processing this phase, all the cached blocks before this phase have to be given up to process this sequence of requests. In general, the crux of our strategy is to push as many blocks as possible through the phase while incurring not too many extra I/Os. Such phases are called downsizing phases since they require the cache size to be downsized. However, pushing as many blocks as possible during downsizing may not be enough. For example, if the blocks are pushed in LRU order (as in [26]), the competitive ratio can be much higher than $O(\sqrt{D})$. Every time a downsize occurs during the superphase, the online algorithm can be pushed into repetitive set of *corrective* I/Os until it determines to a good extent which set of blocks were thrown away by O during the downsize.

5.1 An abstract problem (AP)

Motivation.

Consider the following variant of PDM caching problem with downsizing. Let $K = \sqrt{D}$ be the memory size of O and A. In each phase i, A is shown some set of blocks L_i ($0 < |L_i| \leq K$, i.e., lookahead size can vary from 1 to K and can be different for each phase and is adversary's choice) as the request and then given a number $k_i \leq K$ such that only k_i blocks (from those in memory before the phase) can be carried forward by A (and same applies to O). Additionally, A (and not O) is allowed to carry forward 1 block per disk for free. There is no I/O cost for throwing away a block (for both A and O). How many I/Os can O make A do without itself incurring any I/Os? What is the competitive ratio? Is it $O(K)$ or $O(K^2)$?

If A was not allowed the extra liberty of carrying 1 block per disk, the answer would be $O(K^2)$. Let's say O has acquired K blocks (all on different disks) incurring an I/O. Next, it shows a block in each phase (with $k_i = K$), thus make A do K I/Os. After this sequence, it sets $k_i = K - 1$. This forces both O and A to drop a block. Then it repeats for $K - 1$ phases (with $k_i = K - 1$) and shows the block not in A's memory, forcing further $K - 2$ I/Os. Now, it sets $k_i = K - 2$ and so on. Thus, it can force $K(K+1)/2$ I/Os. At the end of this sequence O can do an I/O, load K new blocks and start over again, forcing a competitive ratio of $O(K^2)$.

However, with the liberty for A to carry 1 block free per each disk, A's cost in the above example is zero. Even in this case, O can still force somewhere close to $O(K^2)$ I/Os without it doing an I/O, if initially O had all the K pages on the same disk (rather than all on different disks). Assume that initially both O and A have all the same blocks and all belong to the same disk. Now O can set $k_i = K - 2$. This will force O to throw 2 blocks and A to throw 1 block. Now, keeping $k_i = K - 2$, O can force $K - 2$ I/Os of A in the phases to come by showing the block not in A's memory. After A has identified which block was thrown away by O, O sets $k_i = K - 3$ and repeats. Thus O could make A do $(K-1)(K-2)/2$ I/Os. Yet, O *cannot* really force the competitive ratio of $O(K^2)$ because to revert to its initial memory configuration O must do K I/Os (and not just 1). Thus, we could say to force more corrective I/Os on A, O lost some of the "potential" stored in its memory configuration. The competitive ratio in this case can be proved to be $O(K)$. Let $d_1 > d_2 > ... > d_K$ be the number of blocks in the memory M_i of O, the potential of O is defined as $\sum d_j^2$. Note that with an I/O of O, the potential can only increase by at most $2K$. Furthermore we shall show that if A was forced to do x corrective I/Os, then O's potential loss is at least x. Note that in above two examples O's initial potential was K and K^2 respectively. To gain the potential of K^2 it must have done $O(K)$ I/Os initially.

We shall show that our problem can be reduced essentially to this abstract problem. There are additional complexities like how to handle the uncertainty due to multiple (nested) downsizes simultaneously and how to handle selective downsizing which we show for the abstract problem in more formal settings below.

Settings.

More formally, consider following problem: O has a memory size of K, while A has memory size of K plus extra cache (called carry cache) of size K which can hold at most one block from each disk. The request sequence is shown to A in phases. In each phase, some blocks are shown to A (the number of blocks to be shown is adversary's choice). Additionally, a phase can also request a downsize. Each downsizing request is also accompanied with a set S^c of blocks which have to be retained after the downsize. Such a sequence runs from the start of the superphase till the end of a superphase (i.e., for some set of phases). During this we assume O does no additional I/Os. Thus, request sequence is $\mathcal{L}_1, \mathcal{L}_2, ...\mathcal{L}_j$. Each $\mathcal{L}_i$ consists of a triplet $< R_i, S_i^c, c_i >$, where R_i is a sequence of shown blocks, S_i^c is a subset of R_i denoting compulsory blocks which cannot be thrown away by A during the downsize, and c_i is a number called downsize parameter (or end capacity) which denotes number of allowed blocks for A and O to be carried after the phase i. Note that $K \geq c_1 \geq c_2 ... \geq c_j \geq 0$.

A's strategy and analysis.

Let M_i^o be the memory of O before phase i and M_i^a be that of A. Let the disk distribution of blocks in M_1^o be $d_1 > d_2 > ....d_K$. Then, the initial potential of O is defined as $\sum d_i^2$. For each disk of these disks in M_1^o, let $w_i = d_i$ be the weight of each block on that disk. Thus, the potential is nothing but the sum of weights of all blocks in M_i^o.

We shall show our solution of A's strategy using a stepwise gradation among three versions of the abstract problem: APv1, APv2 and APv3. APv3 is the actual abstract problem without any assumptions. APv2 is the particular case of APv3 where A is allowed some extra knowledge. APv1 is the particular case of APv2 where we assume there is only one downsizing phase throughout the superphase.

APv1: Let us, for this version, assume that A knows the initial weights w_i for each disk before the beginning of the superphase. Let us also assume that O and A start with the same memory configuration. That is, $M_1^a = M_1^o$. Additionally, we assume there is only one downsizing phase during the superphase. We shall show that A can be designed such that the number of corrective I/Os A needs to spend after this downsize is bounded by O's loss of potential during this downsize. After the downsize, A manages the blocks using the following 4-way partition: (1) Certain set C, (2) Uncertain set U (3) Carry Set E (4) Thrown-away set T. The sets C and U reside in the memory of A while E resides in the extra carry cache of A. Thus, E can have at most one block from each disk. At the downsizing phase i, let $K - c_i = \hat{D} + x$ (where $\hat{D}$ is number of disks covered by blocks in memory of A, barring those blocks in S_i^c, at phase i). Then, A ranks the disks by their decreasing weights. Next, A shifts 1 block from each disk into carry set E and it throws away the remaining x blocks from lowest ranked disks into thrown-away set T. The remaining blocks in the memory of A now consist of uncertain set U (it is unknown whether O has these blocks or not). The blocks from S_i^c are placed in certain set C. We make sure no block from S_i^c goes into E or T initially. Finally, for a particular disk, if there are no blocks in the uncertain set from disks below (in ranked order by weights) it, then its corresponding block in carry set E is shifted to T.

During subsequent phases, all the blocks in requested set R get moved to C. If the requested block comes from U, it is directly placed in C. If it comes from E, it is placed in C and in exchange one of the blocks from U, from the same disk, is moved to E. If the requested block comes from T, then it is charged as corrective I/O. In this case, the next block in U from the bottom of the queue (the queue ranks blocks by their decreasing weights, breaking ties arbitrarily) is moved to T and the requested block from T moves to C. After this, A again checks if U has no block below (in all the disks ranked lesser) some disk. If so, the carry set block from that disk is moved to T. Note that all the accesses to T result in corrective I/Os and ends up shifting a block from U in T. This shifting of blocks from U to T occurs according to the queue. This ranking by weights is done to ensure that A's guess of O's thrown away set has minimum possible weight. When the bottom of this queue shifts to an upward disk, a block from E on that disk gets shifted to T. We shall prove the following two facts:

Fact 1. *The number of access to T at any given instance is bounded above by the weight of T (i.e., sum of weights of blocks in T).*

Proof. The corrective I/Os happen in the increasing order of weights in queue, because that is how A replaces blocks in T. Every time our corrective trail moves to an upward disk, the size of T increases by one block from this disk which was moved from E to T. Since these carry set blocks (moved to T) have enough weight to pay for all the requests from their corresponding disk, the weight of T compensates for number of corrective I/Os so far on T. □

FACT 2. *At any point, the weight of T is bounded above by weight of T', the thrown away set of O.*

PROOF. This is true because the thrown away set of O is at least as big as that of A and moreover A chooses the minimum weighted blocks to throw away. □

APv2: Now, consider a more general case, where A knows O's initial ranking of the disks but not O's configuration; A starts with empty memory, and there could be multiple downsizes (not just one).

In this case, A employs the following modifications. It partitions T into $T_1, T_2, ..., T_j$ which are non-empty for each downsizing phase. Also U is maintained as $U_1 \subseteq U_2 \subseteq ... \subseteq U_j = U$. Each pair T_i, U_i maintains the best guess about what was thrown away and what is uncertain due to the downsizing in phase i. For simplicity, consider T_1, T_2. For the request in T_2 the next block can be from U_1. We maintain the invariant that thrown away blocks from U_1 to T_2 have more weight than those from $U_2 - U_1$ to T_1(this is to ensure Fact 3(2)). Thus, when a request from T_1 comes, we actually have to shift a block from T_2 to T_1 and then from U_1 to T_2. Also, when a carry set block from U_1 gets moved to T_1 a block from same disk in U_2 is moved to E. Also, in this case previously unseen requests can come (since A starts with empty memory). These are moved to certain set and are treated as those coming from T_0 (i.e., a block from U will get thrown away to include this new block in C). Thus, T_1 and subsequently (possibly) $T_2, T_3, ...$ can be modified due to this.

Let $w(T_i)$ denote the weight of T_i and let T'_i be the set of blocks thrown away by O during downsizing phase i and let $w(T'_i)$ be its weight. Let $T' = \cup T'_i$. Now, similar to Fact 1, Fact 2, we can show:

FACT 3. *(1) The number of corrective I/Os on $T_i \leq w(T_i)$ (2) $\sum_{i=1}^{h} w(T_i) \leq \sum_{i=1}^{h} w(T'_i)$ for any $h \leq j$* □

It is easy to derive the following lemma from the above fact:

LEMMA 1. *During the superphase, the number of corrective I/Os of A is bounded by $w(T')$ and hence the number of corrective I/Os is no more than the loss of potential of A.* □

APv3: Now, we are ready to prove our real case where A does not have any estimate of O's initial weights. In this case the essence would be that A learns them as it sees the blocks. However, this could result in flipping of disk rankings in between. This could result in queues for T_1, T_2 opposing each other, since T_2 ranks disks differently that T_1. Hence, the solution above will not work. We need to re-rank the disks and do the subsequent correction for T_1 also. If this re-ranking happens too often, it could result in a lot of I/Os for keeping T_1 consistent. We shall give a fix for this situation using pseudoweights. A maintains pseudoweights for each disk. The pseudoweight p_i for disk i is always a power of two. It is the power of two just less than the number of blocks seen from disk i so far in the superphase. Thus, for any disk i, $p_i \leq w_i$, the actual weight of disk i. Using this we ensure that the rank of disk does not get updated too often and also that every time a disk changes its rank, there are enough new I/Os to amortize the cost of re-ranking.

Now, A orders disks in the decreasing order of their pseudoweights during any downsize. There is an additional complexity when the pseudoweight of a disk changes. The disk moves up in the ranking. At this situation, A makes amends for this, by changing all previous downsizing guesses. This is done to make sure that the disk rankings of each downsizing is consistent. Thus, when a disk moves up in the ranking, all the T_h's involving this disk could change. However, the number of I/Os required to do this, called re-ranking I/Os, is no more than the number of blocks on disk i seen so far (which is p_i). These I/Os can be attributed to the new incoming blocks seen on disk i between this and previous upgrade of p_i. Thus, number of re-ranking I/Os is no more than $2K$ over the entire superphase. We can modify Fact 3 to show that the number of corrective I/Os is bounded by $2p(T)$ which in turn is at most $2w(T)$.

Thus, we can conclude with the following lemma:

LEMMA 2. *The number of I/Os done by A during the superphase is no more than $3K$ plus twice the drop in potential of O, which is $\Psi_1 - \Psi_{j+1}$.* □

5.2 Algorithm SKEW

We are now ready to sketch our algorithm SKEW which has a lookahead of $2M$ blocks in each phase and achieves the competitive ratio of $O(\sqrt{D})$ for our *real* problem. Although there is a simpler variant of this algorithm (which is somewhat harder to analyse) we use the variant of SKEW which is more close to the abstract problem. For the simpler variant see the appendix. We shall show how this problem can essentially be reduced to the abstract problem situation. Again we shall set $t = M/\sqrt{D}$ as in Section 2.4. Intuitively, imagine each block in abstract problem to be equivalent to a chunk of t blocks (from same disk) here. Thus, there are $M/t = \sqrt{D} = K$ chunks to manage. Everything that does not fit into these chunks can be handled by swapping costs (as in Sec 2.4). We shall assume the same definition of superphase as in Section 2.4 here. Without loss of generality, let the superphase consist of phases 1 though j.

Firstly, the potential function changes slightly here. Let $d_1 \geq d_2 \geq ... \geq d_D$ be the disk distribution of M_1^o. Each block from disk i in M_1^o now has the weight equal to $\lfloor d_i/t \rfloor$ where $t = M/\sqrt{D}$. That is $w_i = \lfloor d_i/t \rfloor$. Note that $w_i < \sqrt{D}$. The potential function $\Psi_1 = \sum_{i=1}^{D} w_i d_i$. Note that only $\sqrt{D}$ disks can have nonzero weights.

The carry set E here can hold up to t blocks from each disk. Accesses to the carry set are considered free because of the $O(t)$ 'free I/Os' (covered by swapping cost) A gets in every phase. Also, now O could do at most t I/Os during the superphase (as opposed to no I/Os in the abstract problem). To compensate for the extra blocks O can imbibe into its memory during the superphase, we allow one more carry set E' for A. This set E' holds the least recently seen t blocks from each disk during the superphase. Thus, E' acts as a thresholded set and as before accesses (or swaps) on E' are paid by the free I/Os.

It remains to describe what happens during the processing of downsizing phase. Firstly, A estimates the minimum number of I/Os to process the phase, given its current memory state. Next, it allows t more I/Os to process the phase and checks if, by doing this, it can carry all the blocks it needs to in the next phase. Even after these t extra I/Os, if it is not possible to carry all the blocks, then it declares the downsizing phase. Once A has decided it is a downsizing phase, then it uses only t extra I/Os (not $2t$) and tries to carry as many blocks as possible. Out of these blocks, it tries to use one-to-one exchanges during phase processing to retain the set which prefers the block on higher ranked disks over the lower ones. If some block is not exchangeable i.e., removing it will reduce the number of blocks A carries forward, then such a block is called a compulsory block. The compulsory blocks end up in a certain set C after the downsizing. All others blocks in the memory of A go either in U, E, or T. Thus, compulsory blocks form the set S_i^c in the downsizing constraint.

Except from downsizing, when the lookahead of the phase is processed, the updates and transfers of blocks between U, C, E and T happen exactly in the same way as in the abstract problem. These four sets serve as a tracking mechanism for the best guess of what O did during various downsizes and also as analytic counters.

Now analogous to analysis of the abstract problem, we can show that the weight of thrown away set T_i compensates for the number of corrective I/Os on T_i. This is because for every disk which the corrective trail completes (i.e., disks lesser in weight than the one where the bottom of the queue is), t blocks from carry set (and from higher disk) are transferred to T_i. Thus, their collective weight is sufficient. The other facts and lemmas are applicable directly. Hence we derive following lemma:

Lemma 3. *During a superphase* $1, .., j$, *the number of I/Os of* A *is bounded by* $3M + 2(\Psi_1 - \Psi_{j+1}) + O(jM/\sqrt{D})$. $\square$

Now, using the definition of superphase, we can claim the following theorem:

Theorem 6. *The competitive ratio of* A *is* $O(\sqrt{D})$.

Proof. In every superphase, O spends at least $M/\sqrt{D}$ I/Os and it also spends at least M/D I/Os in every phase. If O does y I/Os then the gain in Ψ cannot be more than $2y\sqrt{D}$. Thus, the contribution of $2(\Psi_1 - \Psi_{j+1})$ term for A is no more than $4y\sqrt{D}$ in the amortized sense. Now, if there are x superphases and y_i is the number of O's I/O in each of them, then A does at most $O(xM + 4\sqrt{D}\sum_{i=1}^{x} y_i)$ I/Os in x superphases. Note that since each y_i is at least $M/\sqrt{D}$, the second term of the summation dominates. Thus, the cost of A is no more than $O(\sqrt{D})$ times the cost of O. This ensures the competitive ratio of $O(\sqrt{D})$. $\square$

Merely by adjusting $t = \sqrt{M(L-M)/D}$, we can generalize the above theorem:

Theorem 7. *For lookahead* $L > M(1 + 1/\epsilon)$, *there exists an online algorithm which is* $O(\sqrt{MD/L})$ *competitive for caching/prefetching on parallel disks.* $\square$

6. CONCLUSIONS

We give online algorithms which achieve optimal competitive ratios for the caching/prefetching problem on parallel disks. Our bounds show that to achieve effective parallelism one requires lookahead more than the memory size. Note that the lower bounds are based on some peculiar request sequences, which are less likely to occur in realistic workloads. The characterization of request sequences which can circumvent these lower bounds is an interesting future direction. Also, online algorithms which are instance optimal, or whose competitive ratios can be determined in terms of some function of request sequence, may also be meaningful.

7. REFERENCES

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

[2] N. Young. Competitive paging and dual-guided on-line weighted caching and matching algorithms. Ph.D. thesis, Princeton University, 1991. Available as CS Tech. Report CS-TR-348-91.

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

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

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

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

[7] 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.

[8] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116-1127, Sept. 1988.

[9] S. Albers, N. Garg and S. Leonardi. Minimizing stall time in single and parallel disk systems. *In Proc. of 30th Annual ACM Symp. on Theory of Computing (STOC 98)*, 454-462, 1998.

[10] J. S. Vitter and E. A. M. Shriver. Optimal algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3): 110-147, 1994.

[11] P. J. Varman and R. M. Verma. Tight bounds for prefetching and buffer management algorithms for parallel I/O systems. *IEEE trans. on Parallel and Distributed Systems*, 10:1262-1275, Dec. 1999.

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

[13] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O systems. *In Symposium on Parallelism in Algorithms and Architectures*, 2001.

[14] R. D. Barve, E. F. Grove and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601-631, June 1996.

[15] 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 European Symposium on Algorithms (ESA 2001)*, LNCS 2161, 2001.

[16] J. S. Vitter and D. A. Hutchinson. Distribution sort with randomized cycling. *In Symposium on Discrete Algorithms (SODA)*, 77-86, 2001.

[17] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys*, Vol. 33, 2:209-271, June 2001.

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

[19] A. Roy. Prefetching and caching with lookahead. *Manuscript*, 2001.

[20] M. Kallahalla and P. J. Varman. Red-black prefetching: An approximation algorithm for parallel disk scheduling. *In Foundations of Software Technology and Theoretical Computer Science*, LNCS 1530, 66-77, December 1998.

[21] D. Breslauer. On competitive online paging with lookahead. *TCS* 209(1-2), 365-375, 1998.

[22] T. Kimbrel and A. R. Karlin. Near optimal parallel prefetching and caching. *In Foundations of Computer Science (FOCS)*, 540-549, 1996.

[23] S. Albers and M. Buttner. Integrated prefetching and caching in single and parallel disk systems. *In Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 109-117, 2003.

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

[25] T. Kimbrel, P. Cao, E.W. Felten, A. R. Karlin and K. Li. Integrated parallel prefetching and caching. *In Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1996.

[26] R. Shah, P. J. Varman, and J. S. Vitter. Online algorithms for caching and prefetching on parallel disks. *In Symposium of Parallelism in Algorithms and Architectures*, 2004.

[27] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*, Cambridge University Press, 1998.

APPENDIX

A. SIMPLER VARIANT OF SKEW.

Here, we give a *simple* (to describe) variant of SKEW. This also achieves the desired competitive ratio. Let $t = M/\sqrt{D}$. Assume without loss of generality that the current superphase consists of the phases $l, ..., j$. Algorithm SKEW (simple) works as follows:

- Let I be the set of interest. The set I consists of all the blocks in $L_l \cup .. \cup L_j$ except for the first (in the order of their first appearance in the superphase) t blocks from each disk. Note that I is at most M blocks.
- At every phase SKEW maintains the weights for each disk in appearing in I. The weight for disk i is $\lfloor\lfloor d_i/t \rfloor\rfloor$, where d_i is the number of blocks in I from the disk i seen so far in the superphase and $\lfloor\lfloor x \rfloor\rfloor$ denotes the highest power of 2 less than x. All the blocks from the disk i have this weight.
- Let y be the number of I/Os O must have done before the start of this superphase. Then, at the end of each phase SKEW calculates the highest weighted subset of I that O can maintain (at the end of this phase) within $y + 2t$ I/Os and keeps this subset in its cache at the end of the phase. This set can be calculated by simulating O.

This algorithm is harder to analyse. One way to analyse this algorithm is by comparing it with the algorithm based on abstract problem.

B. FIGURE FOR APV2.

Here is an illustrative figure (Fig. 1) showing the book-keeping data structures in APv2 (and also in APv3). This shows the effect of nested downsizings.

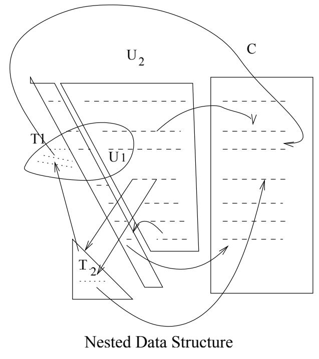

Figure 1: Maintaining C, U, E and T