PC-OPT: Optimal Offline Prefetching and Caching for Parallel I/O Systems

Mahesh Kallahalla, *Member*, *IEEE*, and Peter J. Varman, *Senior Member*, *IEEE*

Abstract—We address the problem of prefetching and caching in a parallel I/O system and present a new algorithm for parallel disk scheduling. Traditional buffer management algorithms that minimize the number of block misses are substantially suboptimal in a parallel I/O system where multiple I/Os can proceed simultaneously. We show that in the offline case, where a priori knowledge of all the requests is available, PC-OPT performs the minimum number of I/Os to service the given I/O requests. This is the first parallel I/O scheduling algorithm that is provably offline optimal in the parallel disk model. In the online case, we study the context of global L-block lookahead, which gives the buffer management algorithm a lookahead consisting of L distinct requests. We show that the competitive ratio of PC-OPT, with global L-block lookahead, is $\Theta(M - L + D)$, when $L \leq M$, and $\Theta(MD/L)$, when $L > M$, where the number of disks is D and buffer size is M.

Index Terms—Parallel I/O systems, caching, prefetching, scheduling, buffer management, competitive ratio, algorithms, online algorithm, offline algorithm.

1 Introduction

The I/O system is a critical bottleneck for many modern data-intensive applications. Parallel I/O systems consisting of multiple disks have the potential to improve I/O performance by performing multiple concurrent I/Os. However, it is a challenging problem to successfully exploit the higher available bandwidth to reduce application I/O latency.

Caching and prefetching are fundamental techniques used to improve I/O performance by exploiting the main-memory buffer present in I/O systems. Blocks can be cached in the I/O buffer so that future requests to the same data are serviced from main memory instead of accessing the much slower disk. Additionally, the I/O buffer can be used to facilitate prefetching. Prefetching refers to the process of reading data from disks prior to their being requested by the computation. While a read progresses on one disk, concurrent reads can be started on other disks to prefetch data that are required later. The prefetched data are held in the I/O buffer till needed. This overlap of data fetches reduces the I/O time significantly.

This paper deals with the problem of prefetching and caching in parallel I/O systems. The aim is to exploit the parallelism provided by multiple disks to reduce the average read latency seen by an application by using appropriate buffer management techniques. The prefetching algorithm must schedule reads carefully so that the most useful blocks are fetched in advance. To exploit locality effectively, the caching mechanism must retain the most valuable blocks in the buffer when the need for eviction arises. The interrelation between prefetching and caching decisions makes designing effective buffer management policies challenging. Applying these techniques to parallel I/O systems is fundamentally different from that in systems with a single disk [7], [17], [22]. Intuitively appealing greedy prefetching policies that are suitable for single-disk systems can have poor worst-case and average-case performance for multiple disks [7], [21]. Similarly, traditional caching strategies do not account for parallelism in the I/O accesses and, consequently, may not perform well in a multiple-disk system. In this paper, we will present a new priority-controlled greedy algorithm, PC-OPT, for optimizing prefetching and caching decisions in a parallel I/O system.

The model of the I/O system used to analyze our algorithm is based on the Parallel Disk Model [25]: The I/O system consists of D independent disks that can be accessed in parallel and a buffer of capacity M blocks through which all disk accesses occur (See Fig. 1). The computation requests data in *blocks*; a block is the unit of disk access. When an I/O is initiated on one disk, blocks can be concurrently fetched from other disks in the same parallel I/O step. The computation waits for data from the I/O system only when the data are not available in the buffer.

The ideal measure of performance of an I/O system is the time it takes to service a set of I/O requests. In a multiple-disk system, this depends on the time for each disk access and the concurrency among the different disks. The former depends upon parameters like the geometry of the disk, load on the disk, unit of data fetch, the layout of the data, and the disk-head scheduling algorithm. In practice, parameters such as unit of data fetch, data layout, and physical geometry are optimized to perform I/O from a single disk efficiently. Additionally, by issuing I/O requests to a disk in batches, the prefetching algorithm can provide the individual disk-head schedulers with considerable scope for optimizing the disk access times. Our prefetching

- *M. Kallahalla is with Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304. E-mail: mahesh_kallahalla@hp.com.*
- *P.J. Varman is with the Department of Electrical and Computer Engineering, Rice University, Houston, TX 77251. E-mail: pjv@ece.rice.edu*

Manuscript received 5 July 2001; revised 19 Oct. 2001; accepted 23 Oct. 2001. For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114468.

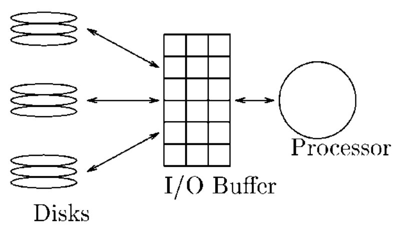

Fig. 1. Parallel disk model

and caching algorithm attempts to maximize the disk concurrency by minimizing the number of parallel I/O steps. In each parallel I/O step, up to D blocks (at most one from each disk) are fetched from the I/O subsystem. Note that this is different from the *total* number of I/Os performed on all disks as an I/O on one disk may be overlapped with an I/O on another disk. An empirical study [12] discusses the practical usefulness and limitations of the idealized parallel disk model.

In the parallel disk model, the computation is implicitly assumed to be very much faster than I/O. This is a reasonable characterization of I/O-bound applications where the main performance bottleneck is the I/O subsystem. In these situations, the I/O time dominates the computational time and, hence, the benefit of overlapping computation and I/O is small compared to the potential benefit of overlapping I/Os on different disks. Theoretically, a block is consumed in zero time so that the total elapsed time is just the I/O time.

The I/O trace of a computation is characterized by a *reference string*, which is an ordered sequence of I/O requests made by the computation. In serving a reference string, the buffer manager (which makes the prefetching and caching decisions) determines which blocks to fetch and when to fetch them so that the computation can access the blocks in the order specified by the reference string.

If the application requests are known only when the data are immediately required, then only speculative prefetching is possible. In order to prefetch accurately, some amount of information about future requests is essential. This information about future accesses is embodied in the idea of lookahead. We define and work with *global lookahead*, which provides the prefetching algorithm a window of future requests. Global lookahead is an abstraction of the lookahead available when applications provide hints to the prefetcher [22] or when a speculative execution of the application provides a window into potential future I/O requests [11]. Lookahead is also naturally available in applications like video retrieval [13], broadcast servers [4], and database systems. Our notion of lookahead is similar to *strong lookahead* [2], introduced to study caching in sequential I/O systems. We consider both the *online* situation, which uses bounded lookahead, as well the *offline* situation, which uses knowledge of the entire reference string to construct the I/O schedule.

Specifically, this paper investigates the problem of generating an I/O schedule for a given read-only reference string. The I/O schedule specifies the blocks to be fetched in each I/O step and the blocks to be evicted from the buffer, subject to the conditions that, in any I/O step, never more than one block is fetched from a disk and the number of blocks in the buffer is no more than M. In the offline case, the algorithm has a priori knowledge of the entire reference string. In the online case, it only has knowledge of past references and references in the lookahead. The lookahead consists of the next L distinct references beyond the current reference. The goal of the scheduling algorithm is to generate a schedule that performs the smallest number of parallel I/Os with the available information.

The buffer management algorithm PC-OPT presented in this paper is an *optimal offline* prefetching and caching algorithm in the parallel disk model. That is, when PC-OPT has a priori knowledge of the entire reference string, it generates a schedule of minimal length. This is the first optimal offline scheduling algorithm for the parallel disk model that we are aware of. In an online scenario, PC-OPT uses available lookahead to make prefetching and caching decisions dynamically. Specifically, we show that the competitive ratio of PC-OPT is $\Theta(M - L + D)$ when $L \leq M$ and $\Theta(MD/L)$ when $L > M$, where D is the number of disks, M is the size of the I/O buffer, and L is the number of distinct references in each lookahead window. The competitive ratio is the worst-case ratio of the number of I/Os performed by the online version of PC-OPT that has access to only the references in the lookahead, to the minimum number of I/Os required if the entire reference string were known in advance. Finally, we would like to point out that, though the running time of the buffer management algorithm is of secondary importance, PC-OPT has a running time of $O(N \log M)$ to schedule a sequence of N I/O references, where M is the size of the I/O buffer.

The rest of the paper is organized as follows: The difficulty of prefetching and caching in a multiple-disk context is illustrated informally in Section 2. A brief survey of related work is presented in Section 3. The buffer management algorithm PC-OPT is described in Section 4. In Section 5, we prove formally that PC-OPT constructs the minimal length schedule in the off-line case. In Section 6, we present bounds on the performance of PC-OPT as a function of the available lookahead. The paper concludes with Section 7.

2 Limitations of Traditional Prefetching and Caching

In this work, we consider the I/O buffer to be a shared resource, capable of buffering blocks from any disk. In contrast, in a distributed buffer model, each disk has its own private buffer; optimal buffer management algorithms for a distributed-buffer model were presented and analyzed in [24]. With a shared buffer, it is possible to dynamically allocate buffer space unevenly to different disks to meet the changing load on individual disks. However, this freedom in allocating buffer space makes the buffer management problem more difficult. The buffer manager has to judiciously decide on questions like how much buffer to allocate for prefetching and how much for caching, which blocks to prefetch and which blocks to cache. For instance, to utilize available bandwidth, it may appear attractive to keep many disks busy during an I/O. However, prefetching

Disk A	a_4	a_1	a_2	a_3
Disk B	b_3			
Disk C	c_2			

(a)

Disk A	a_4	a_1
Disk B	b_3	b_1
Disk 3	c_2	c_1

(b)

Fig. 2. Influence of replacement strategy on I/O schedule length. (a) Using MIN as eviction policy. (b) Optimizing for multiple disks.

too eagerly can fill up the buffer with blocks which may not be used till much later in the computation. Such blocks have the adverse effect of choking the buffer and reducing the parallelism in fetching more immediate blocks. In earlier works [7], [15], we showed that, even when the problem is reduced to just prefetching decisions, it is not trivial: In this case, when each block is accessed exactly once, the problem of deciding which blocks to fetch in an I/O is non-intuitive. In the general case considered here, the buffer manager has to deal with the additional complexity of deciding which blocks to cache and which to evict, in conjunction with prefetching. As it turns out, these decisions are interrelated. A good prefetching and caching algorithm needs to cooperatively decide how much buffer space to allocate for prefetching in a particular I/O and which blocks ought to be prefetched then.

Traditional caching policies for buffer management have concentrated on minimizing the total number of disk accesses. However, these algorithms do not generalize to the problem of optimizing parallel I/Os. For instance, the offline caching algorithm MIN [8], which evicts the block that will be next requested farthest in the future, is known to minimize the number of sequential I/Os in a single-disk I/O system. But, as the following example illustrates, using MIN as the caching policy in a parallel I/O system can potentially serialize otherwise fully parallelizable accesses.

Consider a system with three disks and a buffer of size six. At some point during the computation, let the buffer contain the blocks a_1, a_2, a_3, b_1, b_2, and c_1, where blocks a_i, b_i, and c_i are from disks A, B, and C, respectively. Let the remainder of the reference string, which is fully known to the scheduler, be:

$$\langle a_4, b_3, c_2, a_4, b_3, b_2, b_1, c_1, a_1, a_2, a_3 \rangle$$

The next three request are to blocks a_4, b_3, and c_2, all of which are not present in the buffer. Since all these can be fetched in parallel, three blocks are evicted to fetch these blocks.[1] The actual set of blocks that are evicted determines the overall length of the I/O schedule.

Fig. 2a shows the schedule generated by an algorithm which uses MIN to service this reference string. The farthest referenced blocks, a_1, a_2, and a_3, are evicted, requiring three I/Os to fetch them back. On the other hand, evicting blocks a_1, b_1, and c_1, instead results in a schedule of length two, shown in Fig. 2b.

The example highlights the need for the caching algorithm to explicitly account for I/O parallelism in addition to issues in traditional sequential caching. This example can be generalized to prove that irrespective of the

1. Even if we fetched these blocks one at a time, the same conclusion holds.

Disk A	a_1	a_2	a_3	a_4	a_5	a_6	a_7		
Disk B	b_1	b_2	b_3		b_4				
Disk C	c_1	c_2			c_3	c_4	c_5	c_6	c_7

Fig. 3. Greedy-in-order schedule.

prefetching policy, any algorithm that uses MIN for block replacement may require $\Omega(D)$ times more parallel I/Os than the optimal, where D is the number of disks. It should be noted that this is the worst possible dilation of the schedule length due to ignoring I/O parallelism.

Even when there are no caching decisions to be made, the problem of deciding when any block ought to be fetched is challenging. When the reference string consists of requests to all distinct blocks, caching is a non-issue as there is no benefit in retaining a block in the buffer after its reference. As the following example shows, even in this case, creating an optimal schedule is nontrivial. The problem of optimally scheduling such read-once reference strings was considered in [15].

As before, let the I/O system have $D = 3$ and $M = 6$. Let the blocks labeled a_i (respectively, b_i, c_i) be placed on disk A (respectively, B, C) and the (fully known) reference string be:

$$\langle a_1, a_2, a_3, a_4, b_1, c_1, a_5, b_2, c_2, a_6, b_3, c_3, a_7, b_4, c_4, c_5, c_6, c_7 \rangle.$$

Fig. 3 shows the I/O schedule constructed by an intuitive greedy algorithm that always fetches blocks in the order of the reference string and maximizes the disk parallelism at each I/O step. In Step 1, blocks a_1, b_1, and c_1 are fetched concurrently in one I/O. Block a_1 is consumed immediately (in zero time). Next, block a_2 is requested and blocks a_2, b_2, and c_2 are fetched in parallel in Step 2. Subsequently, the buffer contains five blocks: a_2, b_1, b_2, c_1, and c_2. Now, when a_3 is requested, an I/O needs to be done to fetch it. However, there is buffer space for only one additional block besides a_3 and the choice is between fetching b_3, c_3, or neither. Fetching greedily in the order of the reference string means that we fetch b_3. Continuing in this manner, we obtain a schedule of length nine.

Fig. 4 presents an alternative schedule for the same reference string. The first two steps in the schedule are identical to the previous case. In Step 3, c_3 that occurs after b_3 is prefetched and, in Step 4, c_4 is fetched by evicting b_2 even though c_4 is referenced only after b_4. However, by doing so the overall length of the schedule is reduced to seven, better than the previous schedule.

The above example indicates that optimizing the number of parallel I/Os cannot be done based solely on local heuristics such as "at any time keep as many disks busy as possible" or "fetch blocks only if they are within the next buffer load of blocks to be requested." The algorithm, PC-OPT, presented later in this paper, makes optimal offline caching and prefetching decisions to schedule the sequence

Disk A	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Disk B	b_1	b_2			b_2	b_3	b_4
Disk C	c_1	c_2	c_3	c_4	c_5	c_6	c_7

Fig. 4. Minimal I/O length schedule.

of requests in its lookahead. It carefully delays prefetches so that the increased available buffer space can be used for deeper prefetching, while simultaneously caching only those blocks that occupy buffer space for a small time between repeated references.

3 Related Work

Classical buffer management in the single-disk model deals primarily with optimizing eviction decisions to minimize the number of I/Os performed. The well-known buffer management algorithm MIN [8] is the optimal offline algorithm to construct a minimum-length schedule for a single disk system. Online buffer management for sequential IO systems in the framework of competitive analysis was first studied in [23], followed by studies using extended models which incorporated lookahead [2], [9] or randomization [14], [19]. An analysis of paging algorithms with strong lookahead (similar to global lookahead defined here) was presented in [2] for the single disk situation. Recent formal work on using prefetching to overlap computation with I/O in single disk systems was introduced by [10], where they showed that aggressive prefetching can reduce total elapsed time. A linear programming approach to optimizing the total time in such a stall model was presented in [3].

There has been relatively little work dealing with prefetching and buffer management in the parallel setting, though there has been work on developing I/O-efficient algorithms for specific applications (e.g., [1], [5], [6], [21]).

The problem of online scheduling of read-once reference strings in a multiple-disk system was first addressed in [7]. A lower bound of $\Omega(\sqrt{D})$ on the competitive ratio of *any* prefetching algorithm with M-block lookahead was established and a greedy prefetching algorithm was shown to attain the bound. However, the algorithm could not make use of lookahead beyond M. An offline optimal algorithm, L-OPT, for read-once reference strings was presented in [15]. As an online algorithm with L-block lookahead, L-OPT was shown to have a competitive ratio of $\Theta(\sqrt{MD/L})$, $L \geq M$, and to match the lower bound established in the paper. However, L-OPT did not exploit caching for general reference strings containing repeated accesses to the same set of blocks.

For read-once reference strings, analysis of algorithms exploiting a randomized data placement was studied in [6], [7], [20], [21]. In a generalization of the stall model to parallel disks, [17] designed a sophisticated offline approximation algorithm to service read-many reference strings. In the stall model, the measure of performance is the total elapsed time including both I/O and computation. The model assumes constant I/O and computation times for all blocks and is parameterized by F, the ratio of block I/O time to block computation time. They showed that their algorithm, Reverse Aggressive, is near optimal for typical system parameters (memory size and computation time). I/O scheduling in a distributed buffer configuration where each disk has a private buffer, was considered in [24]. They presented an optimal off-line algorithm PMIN, a generalization of MIN, and showed that the algorithm is optimal for this buffer configuration. Empirical studies on using prefetching, based on hints provided to the system, to improve parallel I/O performance include [11], [22]. The underlying greedy prefetching heuristic is restricted to performing shallow prefetching and preventing prefetched blocks from choking the buffer and affecting the LRU-based caching. An empirical evaluation of prefetching and caching in parallel file systems was presented in [18].

4 Algorithm PC-OPT

In this section, we introduce our parallel I/O scheduling algorithm PC-OPT. The *reference string* is the sequence of I/O requests, $\Sigma = \langle r_1, \ldots, r_N \rangle$, made by the computation, with possibly several references to the same block. Each block resides on a specified disk from which it is to be fetched. The problem is to generate a schedule for a given reference string with the available lookahead information. The I/O schedule specifies the blocks to be fetched in each I/O and the blocks to be evicted, subject to the conditions that, in any I/O, never more than one block is fetched from a disk and the number of blocks in the buffer is no more than M. In the offline case, the algorithm has a priori knowledge of the entire reference string, while, in the online case, it only has knowledge of past references and references in the lookahead. The goal of the scheduling algorithm is to generate a schedule that performs the smallest number of parallel I/Os with the available information.

PC-OPT is a priority-controlled greedy scheduling algorithm. It is made up of two components: the *prefetching* module and the *priority assignment* module. The latter assigns all references in the current lookahead a *priority*. The priority of a block on disk reflects the urgency with which it should be fetched; the priority of a block currently in the buffer indicates how worthwhile it is to cache rather than to evict that block. The prefetching module uses the current priority assignments to determine which blocks to fetch in the current I/O and which blocks to evict. The prefetching and priority assignments modules are described in detail below.

4.1 Prefetching Module

The prefetching module of PC-OPT is presented in Fig. 5. The specification makes use of the the following definitions:

Definition 1.

- *The current lookahead is denoted by the sequence of references $\mathcal{L} = \langle r_i, r_{i+1} \ldots, r_j \rangle$. The block accessed in reference r is denoted by block(r).*
- *For a reference r, the disk from which block(r) needs to be fetched is denoted by $disk(r)$: We shall use the same notation $disk(b)$ to refer to the disk on which block b occurs: The meaning will be clear from the context.*
- *Every reference r in the current lookahead has a priority, denoted by $priority(r)$. We shall use the same notation to denote the priority of a block b that is present in the buffer.*
- *The priority of a block in the buffer is defined as the priority of the next reference in $\mathcal{L}$ to that block. When there is no reference to that block in the lookahead, the block is assigned priority i-MAX, where i is the index*

```
Algorithm PC-OPT: Prefetching Module

On a request $r$, algorithm PC-OPT takes the following actions.

If block(r) is present in the buffer then
    no I/O is necessary before servicing the request
If block(r) is not present in the buffer then
    Update the priorities of references in the lookahead by calling
        the Priority Assignment module
    Update the priorities of blocks in the buffer
    An I/O is initiated to fetch blocks in $H \cap B^+$ evicting the
    lowest priority blocks in $B - B^+$ as necessary, where
        $B$ is the set of blocks present in the buffer
        $H$ is the maximal set of (up to) $D$ blocks, such that if $b \in H$
            then $b$ has the highest priority among all blocks from
            disk(b) in the lookahead but not present in the buffer
        $B^+$ is the maximal set of (up to) $M$ blocks with the highest
            priority in $H \cup B$;
The request $r$ is then serviced
```

Fig. 5. Algorithm PC-OPT: prefetching module.

of the most recent past reference to that block. Here, MAX is a large positive constant greater than the length of the reference string. This priority assignment simply implements a Least-Recently-Used (LRU) eviction policy for blocks that are no longer in the lookahead.

- *Ties in the priorities are handled as follows: A tie between a cached block and a disk-resident block is resolved in favor of the cached block. A tie between two-disk resident blocks or two buffer-resident blocks is resolved in favor of the block occurring earlier in the reference string.*

The blocks in the buffer that are not in the lookahead have the lowest priorities among all blocks and are among the first to be evicted from the buffer. Blocks in the buffer that are also in the lookahead have the same priority as the immediately following reference to the block. PC-OPT is a greedy algorithm in that it attempts to fetch a block from as many disks as possible in every I/O, subject to the constraint that it does not fetch a block if it requires evicting a block with higher priority from the buffer.

The prefetching module of PC-OPT performs I/Os only on demand; that is, only when the referenced block is not present in the buffer. By doing so, PC-OPT can defer its decisions till the latest time and make use of the largest lookahead available. In an I/O, the prefetcher attempts to fetch an uncached block from each of the D disks. The candidate block from a disk is the highest-priority reference from that disk that has not been serviced and is not in the buffer; the set H identifies the candidate blocks. Among these blocks and the blocks currently in the buffer (B), we would like to have the M blocks with the largest priority (B^+) present in the buffer following the I/O. The blocks with the lowest priorities are evicted to fetch the blocks in H that are among the M highest-priority blocks ($H \cap B^+$).

PC-OPT eagerly schedules prefetches and attempts to utilize as many I/O slots as it can during a parallel read. This allows it to utilize future lookahead information as soon as it is available and avoid situations where the decision to leave an I/O slot idle is rendered obsolete (and wasteful) by fresh lookahead information. However, aggressive greedy prefetching has to be tempered since it can fill up the buffer and hinder useful caching and further prefetching. The dynamically assigned priorities allows PC-OPT to recover from counterproductive prefetching decisions, causing it to reuse those prefetch buffers for more useful blocks.

An example of the schedule created by PC-OPT is shown in Fig. 6. The buffer is assumed to be of size $M = 6$. The reference string and the priorities of the references are given in Fig. 6a. In the next section, we will show how these

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	a_1	a_2	a_3	b_1	b_2	c_1	a_4	b_3	c_2	a_4	b_3	b_2	b_1	c_1	a_1	a_2	a_3
Priority	5	4	3	4	3	3	2	2	2	2	2	3	1	1	1	4	3

(a)

Step	1	2	3	4	5
Disk A	a_1	a_2	a_3	a_4	a_1
Disk B	b_1	b_2		b_3	b_1
Disk C	c_1	c_2			c_1
Evict	-	-	a_1	b_1, c_1	a_4, b_3, c_2

(b)

Fig. 6. Schedule created by PC-OPT for given reference string. (a) Reference String. (b) Schedule created by PC-OPT.

Blocks	a_1	a_2	b_1	b_2	c_1	c_2
Priority	1	4	4	3	3	2

(a)

Block	a_2	a_3	b_1	b_2	c_1	c_2
Priority	4	3	1	3	1	2

(b)

Block	a_2	a_3	a_4	b_2	b_3	c_2
Priority	4	3	-90	-88	-89	-91

(c)

Fig. 7. Buffer block priorites immediately before I/Os steps 3, 4, and 5. (a) Prior to referencing a_3 at index 3. (b) Prior to referencing a_4 at index 7. (c) Prior to referencing b_1 at index 13.

priorities are computed; here, we assume they are given. Beginning with an empty buffer, the first two I/Os greedily fetch the two highest-priority blocks from each of the three disks. The third I/O is done when the demand block a_3 at index 3 is requested. Block a_1 is evicted from the buffer and a_3 is fetched. The priorities of the cached blocks that justify this decision are shown in Fig. 7a. Each block in the buffer has the priority of its next reference; hence, a_1, which is next referenced at index 15, has a priority of 1, and a_2, which is next referenced at index 16, has a priority of 4. Of all the cached blocks, a_1 has lowest priority and is evicted to fetch a_3. Block b_3 (referenced at index 8), which is also a candidate to be fetched, has priority 2; this is not more than the priority of any block currently in the buffer and is hence not fetched. The next I/O is required just prior to reference a_4 at index 7. The priorities of blocks in the buffer at this time are shown in Fig. 7b. Once again, all blocks in the buffer have the priority of their next reference. The set of candidate blocks to be fetched in this I/O are a_4 and b_3, both of which have priority 2. The two lowest-priority blocks in the buffer (b_1 and c_1) have priority 1 and are therefore evicted in favor of a_4 and b_3. Finally, the last I/O is required just prior to referencing b_1 at index 13. At this time several cached blocks are no longer in the remaining lookahead. These blocks (a_4, b_2, b_3, and c_2) are assigned priority i-MAX, where i is the index of their latest reference. This is shown in Fig. 7c, where MAX has been assumed to be 100. Candidate blocks a_1, b_1, and c_1 are fetched by evicting blocks c_2, a_4, and b_3 from the buffer. Note that the blocks not in the lookahead are evicted in LRU order.

Finally, we discuss the relation between disk-head scheduling at the individual disks and the schedule created by PC-OPT. Rather than present one parallel request at a time to the disk subsystem, PC-OPT can batch the requests so that, at any time, the next M blocks in its schedule (beyond the last reference) are requested from the disks. The individual disks can reorder the requests in their queues to optimize access times. This decoupling of the schedules allows lower-level access-time optimizations to be done independently of the higher-level load-balancing schedule of PC-OPT.

4.2 Priority Assignment Module

The priority assignment module, the heart of PC-OPT, assigns priorities to each reference in the lookahead. The priority of a block is a measure of how important it is to have that block in the buffer. With respect to prefetching, a low priority indicates that an I/O for that block can be delayed and, with respect to caching, a low priority indicates that the block can be evicted from the buffer.

The I/O buffer is used both to cache blocks since their previous reference and to hold prefetched blocks till they are referenced. PC-OPT attempts to set the priority of every block as low as possible, subject to certain constraints. It employs two principles in assigning the priorities: 1) Prefetches for blocks should occur close to their references so that prefetched blocks do not wastefully occupy buffer space and 2) avoid caching a block if there is any later free I/O slot available which can be used to fetch the block.

Among the candidates of blocks to cache, PC-OPT caches the block which occupies buffer space for a smaller duration. Hence, if, at some time, two blocks are required in the buffer, PC-OPT prefers to cache the block whose previous reference is closer to the current time as this would reduce the buffer pressure between the two previous accesses. For prefetching, PC-OPT assigns priorities so that prefetches can be delayed till close to the actual reference. These choices in ordering blocks to be fetched or evicted are made by PC-OPT through its priority assignment scheme.

Since at most one block can be fetched from one disk in an I/O, at any time all blocks on the same disk have distinct priorities. However, two blocks from different disks can have the same priority, indicating that, at the current time, both of these are equally preferable. Also, there can be at most M blocks cached in the buffer at any time. Hence, if a reference r has priority p, then at most $M-1$ distinct references that occur after r can be assigned a priority higher than $p-1$. This is necessary to guarantee that there is always buffer space available to fetch the demand block for which the computation is waiting. Finally, a priority needs to be assigned to those blocks that are present in the buffer to allow them to be compared to blocks that can be prefetched. A block in the buffer has the priority of the next reference to that block. If a block is no longer in the lookahead, we assign it a low priority so that it is evicted before blocks in the lookahead; if several blocks are no longer in the lookahead, their relative priorities are chosen so that they are evicted in least-recently-used (LRU) order.

The priority assignment module examines maximal-length subsequences of the lookahead consisting of M distinct references, called *phases*. It then assigns priorities to certain references in the phase, one distinct reference from each disk. The priority assignment can be understood by considering the first phase, *phase*(1): This consists of the largest subsequence of the lookahead that includes the last reference and has at most M distinct references. All references which are assigned the smallest priority should belong to *phase*(1); otherwise, there will be some reference such that M or more blocks referenced after it have a higher, or same, priority. Which among these blocks should have the lowest priority? First, we can assign the lowest priority to at most one distinct reference from each disk. Additionally, between two blocks from the same disk, we prefer assigning this priority to the block whose previous reference outside this subsequence is earlier. This is to

Routine to assign priorities to references

This routine is used to assign priorities to references $\mathcal{L} = \langle r_j, \ldots, r_k \rangle$. Priorities are assigned only to specific references; the priority of any other reference is taken to be the same as that of the previous reference in $\mathcal{L}$ to the same block.

```
Initialize lowestPriority to 1, all other counts to 0, and sets to $\phi$.
For $i$ from $k$ down to $j$
    Let previous($r_i$) be the index of the previous reference to block($r_i$),
        or $-i$ if that index is less than $j$
    If there is $r$ in $\mathcal{P}$[disk($r_i$)] such that block(r) = block($r_i$) then
        Replace $r$ in $\mathcal{P}$[disk($r_i$)] with $r_i$ and key previous($r_i$)
    else
        If numberOfBlocksPlaced = $M$ then
            For each disk $d$ such that $\mathcal{P}[d]$ is not empty
                Assign priority(r) ← lowestPriority,
                    where $r$ is the reference with the smallest key in $\mathcal{P}[d]$
                Remove $r$ from $\mathcal{P}[d]$;
                Decrement numberOfBlocksPlaced
            Increment lowestPriority
        Insert $r_i$ into $\mathcal{P}$[disk($r_i$)] with key previous($r_i$)
        Increment numberOfBlocksPlaced
```

Fig. 8. Algorithm PC-OPT: priority assignment.

indicate that, between the two blocks, we would rather not cache this block: assigning it a lower priority indicates that we prefer caching other blocks over this one.

The algorithm to assign priorities is described in Fig. 8. The variable *numberOfBlocksPlaced* keeps count of the number of distinct references in the subsequence scanned. A phase is complete when this count reaches M and the next reference differs from those currently in the sequence. For each disk d, the distinct references that are encountered are maintained in a priority queue $\mathcal{P}[d]$. There is one entry in $\mathcal{P}[d]$ for each distinct reference from disk d. If several references are to the same block, then only the earliest (leftmost) reference is stored in the priority queue. The references in any $\mathcal{P}[d]$ are ordered by the index of the previous occurrence of that block outside the phase (the key field in Fig. 9). The current lowest assignable priority is tracked by *lowestPriority*. Once a phase has been identified, the reference with the smallest key from each disk is assigned this priority. All other references in this phase that access the same block are also assigned the same priority.

The priority assignment module of PC-OPT has low time complexity. The amortized time complexity of the priority assignment routine is $O(\log M)$ per reference using any standard priority queue for implementing the $\mathcal{P}$ array. One element is inserted, deleted, or updated in the priority queue to account for one reference. The values for previous (r) can be initialized in one pass over all the references whose priorities need to be assigned. The overhead of assigning priorities depends on the frequency of additional lookahead being available as priorities of blocks in the lookahead change only when the lookahead window is extended.

An example illustrating the priority assignment is shown in Fig. 9. The reference string is that used in the example of Fig. 6. Fig. 9a illustrates *phase*(1); it consists of the last M distinct references with indexes 12 through 17. The references belonging to *phase*(1) are shown in bold. The field *key* for each reference in the phase is the index of the previous reference to the same block outside this phase. Hence, for instance, c_1 has a key field of 6 since the previous reference to c_1 outside the phase is at index 6 in the reference string. In the inner *for-loop*, the blocks from each disk with the lowest key are assigned the lowest priority (currently 1). Among the references a_1, a_2, and a_3 on disk A, a_1 has the smallest key (previous reference is earliest), and hence, is assigned priority 1. Similarly, b_1 and c_1 are the references with the smallest key values from the other two disks and are also assigned priority 1.

Fig. 9b shows the second phase of the example above; this is when the else condition is entered for the second time. The second phase includes the largest suffix of M distinct references, excluding the references which have previously been assigned priorities. At this time, $\mathcal{P}$ consists of the earliest (leftmost) reference of each distinct block in the phase. They are keyed by the index of the earliest reference to that block outside the phase. Hence, for instance, $\mathcal{P}$ would hold the reference to a_4 at index 7 rather than at index 10. Note that a reference r at index i is assigned a key of $-i$ if there are no previous references to *block*(r) outside this phase. Thus, blocks a_4, b_3, and c_2 have negative priorities. Again, in the inner *for-loop* of the algorithm, all the blocks in $\mathcal{P}$ with the smallest keys are assigned the lowest priority (currently 2) and the priorities are extended to all references to the same blocks. Consequently, references at indexes 7, 8, and 9 which have the smallest key values from each disk are assigned priority 2. Since the algorithm assigns the same priority to all other references to the same block in the phase, references a_4 and b_3 at indexes 10 and 11 are assigned the same priorities as the corresponding earlier references. Fig. 9c, Fig. 9d, and Fig. 9e show the remaining phases and the final assignment of priorities to all references.

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	a_1	a_2	a_3	b_1	b_2	c_1	a_4	b_3	c_2	a_4	b_3	$\mathbf{b_2}$	$\mathbf{b_1}$	$\mathbf{c_1}$	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$
Key												5	4	6	1	2	3
Priority													**1**	**1**	**1**		

(a)

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	a_1	a_2	a_3	b_1	b_2	c_1	$\mathbf{a_4}$	$\mathbf{b_3}$	$\mathbf{c_2}$	$\mathbf{a_4}$	$\mathbf{b_3}$	$\mathbf{b_2}$	*	*	*	$\mathbf{a_2}$	$\mathbf{a_3}$
Key							-7	-8	-9	-	-	5				2	3
Priority							**2**	**2**	**2**	**2**	**2**		1	1	1		

(b)

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$	$\mathbf{b_1}$	$\mathbf{b_2}$	$\mathbf{c_1}$	*	*	*	*	*	$\mathbf{b_2}$	*	*	*	$\mathbf{a_2}$	$\mathbf{a_3}$
Key	-1	-2	-3	-4	-5	-6						-				-	-
Priority			**3**		**3**	**3**	2	2	2	2	2	**3**	1	1	1		**3**

(c)

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	$\mathbf{a_1}$	$\mathbf{a_2}$	*	$\mathbf{b_1}$	*	*	*	*	*	*	*	*	*	*	*	$\mathbf{a_2}$	*
Key	-1	-2		-4												-	
Priority		**4**	3	**4**	3	3	2	2	2	2	2	3	1	1	1	**4**	3

(d)

INDEX	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Reference	$\mathbf{a_1}$	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Key	-1																
Priority	**5**	4	3	4	3	3	2	2	2	2	2	3	1	1	1	4	3

(e)

Fig. 9. Priority assignment example. (a) Phase(1). (b) Phase(2). (c) Phase(3). (d) Phase(4). (e) Phase(5).

5 Offline Optimality of PC-OPT

In this section, we analyze the performance of PC-OPT in terms of the length of the schedule it generates. We show that, when, PC-OPT has knowledge of the entire reference string Σ in advance, it generates a schedule of minimal length, that is, it is an *optimal offline* I/O scheduling algorithm. Note that, in this case, the entire reference string is in the initial lookahead of PC-OPT, and, hence, the priorities are all assigned at once. To do the analysis in this case, we start by showing some properties of the schedule generated by PC-OPT.

Property 1. *If PC-OPT is given the entire reference string at the start, the number of I/Os done by PC-OPT to service the reference string is given by the highest priority of any reference in the reference string.*

Our analysis will concentrate on showing that the maximum priority of any block equals the length of OPT, the optimal offline schedule. This will be used together with Property 1 to show that the length of the schedule generated by PC-OPT is of the same length as OPT. We shall start by characterizing the blocks which have a priority p. We begin by formally defining a *phase* that was used in describing the algorithm in Section 4.

Definition 2.

- *Initially, let Σ_1 equal Σ. Define phase(i) to be the largest subsequence of Σ_i, including the last reference in Σ_i, such that the total number of distinct blocks in phase(i) is no more than M.*
- *Define $\Sigma_{i+1} = \Sigma_i - earliest(i)$, where $earliest(i)$ is defined as follows: For each disk d, consider the set of all blocks from disk d referenced in $phase(i)$. Among these, let b be the block whose previous reference outside $phase(i)$ is earliest in the past. Then, $earliest(i)$ consists of all references to block b in $phase(i)$.*

From the above definition and the specification of the priority assignment routine, $phase(i)$ is the sequence of references from among which the references which have a priority i are selected. Also, the references in $earliest(i)$ are exactly the ones with priority i.

Property 2. *The priority of all references in $earliest(i)$ is i.*

From Property 2, we can estimate the maximum priority of any reference by counting the number of "earliest" sets that can be formed from the reference string. Without any loss in generality, we assume that, in OPT, no block that is fetched is evicted before at least one reference to it is serviced. This will simplify the comparison of the schedule generated by PC-OPT and OPT. Let the length of OPT be T_{OPT}. For integer k, $1 \leq k \leq T_{OPT}$, we denote by R_k the expression $T_{OPT} - k + 1$. We denote the jth I/O in a schedule by I_j.

The theorem is proven by inductively showing that OPT can be repeatedly transformed into a series of schedules, each derived from the previous one and of the same length as OPT, such that, in the kth schedule, all references fetched in I_j, $j \geq R_k$, match those in the $earliest(j)$.

Theorem 1. *Given a reference string* Σ *of length L, PC-OPT, with L block lookahead, performs the least number of I/Os to service* Σ.

Proof. We shall prove the theorem by inductively showing that OPT can be repeatedly transformed into a series of schedules OPT*(k), k from $T_{\mathrm{OPT}}, \cdots, 1$, such that schedule OPT*(k) is the same length as OPT and any reference that is fetched by OPT*(k) in I_j, $j \geq k$, is in $earliest(R_j)$. The theorem will then follow due to Properties 1 and 2.

For the Induction Hypothesis (IH), assume that OPT* (k+1) and OPT are the same length and that all references r fetched by OPT*(k+1) in I_j, $j \geq k+1$, are in $earliest(R_j)$. We shall show how to construct OPT*(k) from OPT*(k+1) by changing only the blocks fetched in the kth I/O, I_k, or earlier. The blocks fetched in I_k are chosen to match those in $earliest(R_k)$, thereby reestablishing the IH.

For an arbitrary disk d, let p be the reference fetched in I_k by OPT*(k+1), and let q be the reference from the same disk with the smallest index in $earliest(R_k)$. If either reference does not exist, then we denote p or q by ϕ. For each disk independently, we change the block fetched in I_k of OPT*(k+1) to that belonging to $earliest(R_k)$.

There are four possible cases to consider in the proof.

Case 1: $p = \phi$

In OPT*(k), we will instead fetch q in I_k. By the IH, since q is in $phase(R_k)$ and hence not in $earliest(R_i)$ for any $i \geq k+1$, it is not fetched by OPT*(k+1) in any I_j, $j \geq k+1$. Therefore, q must be fetched by OPT*(k+1) in some I_j, $j < k$.

If q is referenced after I_k in OPT*(k+1), then it must already be in the buffer at the start of I_k. In this case, to get OPT*(k), we evict q immediately before I_k and fetch it back again immediately during I_k, thereby satisfying the IH.

If q was referenced before I_k in OPT*(k+1), then, in OPT*(k), we fetch q in I_k and reference it immediately after the I/O. All references in OPT*(k+1) that occurred between the reference to q and I_k are also deferred to the end of I_k. Since there are at most $M-1$ other blocks which occur after q in Σ that are fetched on or before I_k (these are from among the $M-1$ distinct references in $phase(R_k)$ different from q), there is enough buffer space to defer the references to all these blocks till after I_k.

Case 2: p does not belong to $phase(R_k)$

In OPT*(k) we cancel the I/O for p and instead fetch q in I_k. To show that we can cancel the I/O done by OPT*(k+1) for p in I_k and still get a valid schedule, we argue as follows. Either the index of p in Σ is less than the smallest index of any reference in $phase(R_k)$ or p belongs to $earliest(R_i)$ for some $i \geq k+1$. By the IH, the latter implies that p is fetched in I_j for some $j \geq k+1$. Hence, in this case, we do not need the I/O done for p in I_j and can cancel it while still maintaining a valid schedule. Next, we show that the first case is not possible. By the IH, none of the references in $phase(R_k)$ will be fetched in the I/Os beyond I_k. Hence, all M distinct references in $phase(R_k)$, different from p and referenced after p, would need to have been fetched by I_k. There can be at most M such references (including p) as the buffer capacity is M, an impossibility. This contradicts the hypothesis that OPT*(k) is a valid schedule. Once we cancel the I/O for p, then we use case 1 above to fetch q in I_k to get OPT*(k).

Case 3: p belongs to $phase(R_k)$ and, in OPT*(k+1), both p and q are referenced after I_k.

In OPT*(k+1), both $block(p)$ and $block(q)$ must be in the buffer following I_k since, by the IH, they are not fetched in any I/O beyond I_k. To construct OPT*(k), we fetch q instead of p in I_k and adjust earlier I/Os to ensure that p is in the buffer before I_k.

In OPT*(k+1), let $block(q)$ be last fetched in some I/O, I_q earlier than I_k, and cached till I_k. By the discussion above, such an I/O must occur. Suppose that $block(p)$ was last evicted at time t and not read back till I_k. If t is earlier than I_q, then, in OPT*(k), we fetch $block(p)$ instead of $block(q)$ in I_q. If t is later than I_q, then, instead of evicting $block(p)$ at t, we evict $block(q)$ instead. By the definition of $earliest(k)$, we are guaranteed that the last references to both $block(p)$ and $block(q)$ outside $phase(R_k)$ have been serviced at t and, so, both transformations are valid.

Case 4: p belongs to $phase(R_k)$ and, in OPT*(k+1), q is referenced earlier than I_k.

To construct OPT*(k), we fetch q instead of p in I_k, and adjust earlier I/Os to ensure that p is in the buffer before I_k.

Let q be referenced at time t in OPT*(k+1). At this time, OPT*(k+1) must have q in the buffer. By an argument identical to Case 3, we can transform OPT*(k+1) so that, at t, OPT*(k) has $block(p)$ instead in the buffer. Then, $block(q)$ will be fetched in I_k in OPT*(k) as required.

In OPT*(k), we can reference q only after its fetch at I_k. Hence, all references made by OPT*(k+1) between its reference of q and I_k are also deferred till immediately after I_k. Since there are at most $M-1$ blocks that occur after q in Σ and are fetched no later than I_k, there is always enough buffer available to do this.

Between t and I_k, we fetch the same blocks in OPT*(k) as in OPT*(k+1). Any block not in $phase(R_k)$ that is evicted in these I/Os by OPT*(k+1) is also evicted by OPT*(k), but no blocks that are referenced between q and p in Σ are evicted since these have not yet been referenced. Following the I/O at I_k, all the deferred references between t and I_k are made and blocks that had been evicted by OPT*(k+1) are also evicted.

Thus, OPT*(k+1) can be transformed into a valid schedule OPT*(k) such that the induction hypothesis holds for OPT*(k).

For the **base case,** $OPT^*(T_{OPT}+1)$ is initialized to OPT. Clearly, the two schedules have the same length and the second condition of the induction hypothesis holds vacuously. □

Immediately, it follows that, when the entire reference string is known to PC-OPT, it generates the minimal length schedule to service it.

Theorem 2. *PC-OPT is the optimal offline parallel I/O scheduling algorithm.*

6 Bounds on Online Performance of PC-OPT

In this section, we shall characterize the online performance of PC-OPT using the competitive ratio [16] as the metric. In this context, the competitive ratio is the ratio of the number of I/Os done by the online algorithm to the number of I/Os required by the optimal offline algorithm to schedule the same reference string. Though being a worst case measure, the competitive ratio attempts to isolate the effect of decisions made by the online algorithm from inherent features of the input data.

The performance of an online scheduling algorithm depends on the amount of information in the lookahead and how frequently the lookahead is updated. For the analysis, we consider that, at any time, the algorithm has the next L distinct references in the lookahead. Formally, an algorithm is said to have *global L-block lookahead* if, at any time it knows a portion of the reference string, starting from the next reference to be accessed and including accesses to L distinct blocks, that is, if the lookahead is $\mathcal{L} = \langle r_i, \ldots, r_j \rangle$, then the number of distinct blocks in $\mathcal{L}$ is L. This is a natural definition of lookahead and is similar to that of strong lookahead [2], which was used in the context of on-line caching algorithms for single-disk systems.[2]

We shall next provide bounds on the performance of PC-OPT in the online case, where it only has L-block lookahead. We shall show the bounds separately in the two regions, $L \leq M$ and $L > M$. For the analysis, we partition the entire reference string into subsequences consisting of M distinct blocks, called *segments*: A segment is a maximal length subsequence of the reference string consisting of references to at most M distinct blocks, with the first segment starting with the first reference.

We categorize all references as either *stale* or *clean* based on whether or not the same block has been accessed in the previous segment. For the reference string in Fig. 6a, segment(1) consists of references 1 to 6 and segment(2) consists of references 7 through 14. The clean blocks of segment(2) are a_4, b_3, and c_2 and the stale blocks are b_2, b_1, and c_1. The main theorem below follows from Lemmas 3 and 4, which are proven below.

Let the ith segment be denoted by segment(i) and let the number of clean blocks in segment(i) be c_i. Let the total number of segments in the reference string be N.

Theorem 3. *The competitive ratio of PC-OPT is:*

$$\begin{array}{lll} \Theta(M - L + D) & when & L \leq M \\ \Theta(MD/L) & when & L > M \end{array}$$

Lemma 1. *The number of I/Os done by PC-OPT to fetch the first set of L distinct references of a segment, $L \leq M$, is at most equal to the number of clean blocks in that set.*

Proof. From the definition of the priority assignment scheme, it can be seen that 1) blocks in the buffer which are present in the lookahead have a higher priority than blocks which are not present in the lookahead, and 2) among the blocks in the buffer which are not present in the lookahead, a lower priority is assigned to a block whose past reference was earlier in the past. This indicates that, when blocks are evicted, the blocks in the buffer that are not in the lookahead and whose prior reference was earlier in the past are preferred.

Since there are exactly M distinct blocks in a segment, once a block has been fetched, it will not be evicted during that segment as there will always be some other block which is not in the lookahead but which is present in the buffer. Hence, at the end of a segment, if a block is present in the buffer of PC-OPT, then it is either a block that is referenced in that segment or it is a block which is a clean block from the first lookahead window, the first set of references comprised of L distinct requests, of the next segment.[3]

Consider the I/Os done by PC-OPT in segment(i). From the previous argument, the number of I/Os done by PC-OPT to service the first lookahead window is no more than the number of clean blocks in this lookahead. □

Lemma 2. *The total number of blocks fetched by OPT is at least half the total number of clean blocks in the reference string.*

Proof. We can show that the total number of blocks fetched by OPT is at least $\sum c_i/2$, using an analysis similar to one used to bound the competitive ratio of marking algorithms in [14]; we briefly repeat it here for convenience. Let n_i clean blocks of segment(i) be present in OPT's buffer at the start of segment(i). Hence, the number of blocks fetched by OPT in segment(i) is at least $c_i - n_i$. Also, since n_{i+1} clean blocks are present in OPT's buffer at the end of segment(i), OPT must have fetched at least n_{i+1} blocks in segment(i). Hence, the number of blocks fetched by OPT in segment(i) is at least $\max\{c_i - n_i, n_{i+1}\} \geq (c_i - n_i + n_{i+1})/2$. This, when summed over all segments, gives the result that OPT should have fetched at least $\sum c_i/2$ blocks and, hence, should have done at least $\sum c_i/2D$ I/Os. □

In the example of Fig. 6, at the end of segment(1) (step 3 of the schedule), the buffer contains $a_2, a_3, b_1, b_2, c_1, c_2$. Of the three clean blocks in segment(2), only one of them, (c_2), is present in the buffer at the start of segment (2). Hence, the first bound ($c_i - n_i$) states that at least $3 - 1 = 2$ blocks must be fetched in segment(2). For the second bound, note that two blocks (a_2 and a_3) are cached throughout segment(2), even though they are never referenced in that segment. These two blocks are clean blocks for segment(3) and are present in the buffer at the start of the segment. Carrying these two blocks across segment(2) gives the second bound, (n_{i+1}), that at least two blocks must be fetched in segment(2).

Lemma 3. *When $L \leq M$ the competitive ratio of PC-OPT is $O(M - L + D)$.*

Proof. Consider the I/Os done by PC-OPT in segment(i). From Lemma 1, the number of I/Os done by PC-OPT to service the first lookahead window is no more than the number of clean blocks in this lookahead. There are at most $M - L$

2. L-block lookahead is a slightly weaker form of lookahead than strong lookahead: Strong lookahead requires that the first reference of any lookahead window be to a block that is not present in the previous lookahead. Thus, for instance, several sets of L-block lookahead could correspond to only one strong lookahead window.

3. Sequential paging algorithms with this property have been previously referred to as marking algorithms [14].

other blocks referenced in the rest of the segment and, hence, the number of I/Os done by PC-OPT in the rest of the segment is at most $M - L$. Hence the total number of I/Os done by PC-OPT in segment(i) is at most $c_i + M - L$. We will next show that the total number of I/Os done by OPT is $\sum c_i/2D$, where the sum is taken over all segments in Σ, or $\Omega(N)$, where N is the total number of segments in the reference string; hence, the competitive ratio of PC-OPT will be $O(D + M - L)$.

From Lemma 2, the total number of blocks fetched by OPT is at least $\sum c_i/2$. Hence, OPT should have done at least $\sum c_i/2D$ I/Os. Additionally, in any set of two consecutive segments, there are a total of at least $M + 1$ distinct blocks that are referenced. Hence, OPT should do at least one I/O in every set of two segments, which gives the second bound of $\Omega(N)$ on the total number of I/Os done by OPT to service Σ. □

Lemma 4. *When $L > M$, the competitive ratio of PC-OPT is $O(MD/L)$.*

Proof. Let us divide the reference string Σ into subsequences $\mathcal{L}_1, \cdots, \mathcal{L}_n$, where $\mathcal{L}_1$ is the lookahead available to PC-OPT initially and $\mathcal{L}_i$ is the lookahead window available to PC-OPT after servicing the last reference of $\mathcal{L}_{i-1}$. Note that, from the definition of global L-block lookahead, there are L distinct references in each L_i.

Consider the optimal length schedule OPT to service Σ. Let us partition the schedule into subschedules, each consisting of a sequence of contiguous I/Os of the original schedule. Let the ith subschedule S_i start with the I/O following the last I/O of sub-schedule S_{i-1} and end with the last I/O in which a block from $\mathcal{L}_i$ is fetched; let S_0 start with the first I/O. In the case when $L > M$, it can be noted that each S_i consists of at least one I/O.

First, consider the case when $L \geq 2M$. OPT needs to do at least $(L - M)/D$ I/Os in each S_i as it could have at most M blocks in the buffer prior to scheduling L_i. Thus, in this case, if the number of lookaheads is N, $T_{OPT} = \Omega(NL/D)$. The number of I/Os done by PC-OPT to service any L_i is less than $2M$ more than the number of I/Os in S_i. Thus, $T_{PC-OPT} \leq T_{OPT} + 2NM$, thereby completing the proof for this case.

The other case, when $L < 2M$, is simpler. By Lemma 1, the number of I/Os done by PC-OPT in the first lookahead of any segment is at most the number of clean blocks in that lookahead. However, when the lookahead is more than M, the entire segment is a part of the lookahead. Hence, the total number of I/Os done by PC-OPT in the reference string is no more than the total number of clean blocks in the reference string, $\sum_i c_i$. On the other hand, by Lemma 2, the total number of blocks fetched by OPT is at least half the total number of clean blocks in the reference string. Thus, the total number of I/Os done by OPT is at least $\sum_i c_i/2D$. This gives the result that the ratio of the number of I/Os done by PC-OPT to the number of I/Os done by OPT is at most $O(D)$ when $M < L < 2M$. □

7 Summary

In this paper, we addressed a generalization of the sequential paging problem [8] to a multiple-disk parallel I/O situation. In the parallel I/O model, the *total* number of I/Os is not an appropriate metric since multiple I/Os can proceed concurrently on different disks. The problem here is to optimize the number of *parallel I/Os*.

We argued that both caching and prefetching decisions need to be sensitive to disk parallelism and that there could be substantial loss in parallelism if traditional prefetching and caching algorithms are used. For instance, by using the MIN algorithm [8] for caching, we could sequentialize accesses that could otherwise be parallelized completely.

We used an intuitive model of lookahead, global lookahead, which provides information regarding a subsequence of future accesses to the online algorithm. Lookahead allows algorithms to perform accurate prefetching rather than speculate on future references. Lookahead information is also useful in making caching decisions.

We presented a new algorithm, PC-OPT, for optimizing prefetching and caching decisions in the parallel disk model. We showed that, in the offline case, where a priori knowledge of all the accesses is available, PC-OPT performs the minimum number of I/Os to service the I/O requests. This is the first optimal offline algorithm in the parallel disk model. In the online case, we show that the competitive ratio of PC-OPT, with this lookahead, is $\Theta(\max\{M - L, D\})$, when $L \leq M$, and $\Theta(MD/L)$, when $L > M$, where the number of disks is D and buffer size is M.

Acknowledgments

This research was suported in part by the US National Science Foundation under grants CCR-9704562 and CCR-0105565 and a grant from the Schlumberger Foundation. A preliminary version of this work appears in the *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, SPAA 2001. This work was done while Mahesh Kallahalla was with the Department of Electrical and Computer Engineering at Rice University.

References

[1] A. Aggarwal and J.S. Vitter, "The Input/Output Complexity of Sorting and Related Problems," *Comm. ACM,* vol. 31, no. 9, pp. 1116-1127, Sept. 1988.
[2] S. Albers, "On the Influence of Lookahead in Competitive Paging Algorithms," *Algorithmica,* vol. 18, no. 3, pp. 283-305, July 1997.
[3] S. Albers, N. Garg, and S. Leonardi, "Minimizing Stall Time in Single and Parallel Disk Systems," *J. ACM,* vol. 47, no. 6, pp. 969-986, Nov. 2000.
[4] D. Askoy and M. Franklin, "Rxw: A Scheduling Approach to Large Scale On-Demand Broadcast," *IEEE/ACM Trans. Networking,* vol. 7, pp. 846-861, 1999.
[5] L.M. Baptist and T.H. Cormen, "Multidimensional, Multiprocessor, Out-of-Core FFTs with Distributed Memory and Parallel Disks," *Proc. 11th Ann. ACM Symp. Parallel Algorithms and Architectures,* June 1999.
[6] R.D. Barve, E.F. Grove, and J.S. Vitter, "Simple Randomized Mergesort on Parallel Disks," *Parallel Computing,* vol. 23, no. 4, pp. 601-631, June 1996.
[7] R.D. Barve, M. Kallahalla, P.J. Varman, and J.S. Vitter, "Competitive Parallel Disk Prefetching and Buffer Management," *J. Algorithms,* vol. 36, no. 2, pp. 152-181, Aug. 2000.

[8] L.A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems J.*, vol. 5, no. 2, pp. 78-101, 1966.
[9] D. Breslauer, "On Competitive On-Line Paging with Lookahead," *Proc. 13th Ann. Symp. Theoretical Aspects of Computer Science,* pp. 593-603, Feb. 1996.
[10] P. Cao, E.W. Felten, A.R. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems,* pp. 188-197, May 1995.
[11] F. Chang and G.A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," *Operating Systems Design and Implementation (OSDI),* pp. 1-14, 1999.
[12] T.H. Cormen and M. Hirschl, "Early Experiences in Evaluating the Parallel Disk Model with the VIC^* Implementation," *Parallel Computing,* vol. 23, no. 4, pp. 571-600, 1997.
[13] O. Ertug, M. Kallahalla, and P. Varman, "Real-Time Parallel Disk Scheduling for VBR Video Servers," *Proc. Fifth Int'l Conf. Computer Science and Informatics,* Feb. 2000.
[14] A. Fiat, R. Karp, M. Luby, L. McGeoch, D.D. Sleator, and N.E. Young, "Competitive Paging Algorithms," *J. Algorithms,* vol. 12, no. 4, pp. 685-699, Dec. 1991.
[15] M. Kallahalla and P.J. Varman, "Optimal Read-Once Parallel Disk Scheduling," *Proc. Sixth ACM Workshop I/O in Parallel and Distributed Systems,* pp. 68-77, (see www.ece.rice.edu/pjv for a revised and expanded version), 1999.
[16] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator, "Competitive Snoopy Caching," *Algorithmica,* vol. 5, no. 3, pp. 79-119, Mar. 1988.
[17] T. Kimbrel and A.R. Karlin, "Near-Optimal Parallel Prefetching and Caching," *SIAM J. Computing,* vol. 29, no. 4, pp. 1051-1082, 2000.
[18] D.F. Kotz and C.S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors," *IEEE Trans. Parallel and Distributed Computing,* vol. 1, no. 2, pp. 218-230, 1990.
[19] L.A. McGeoch and D.D. Sleator, "A Strongly Competitive Randomized Paging Algorithm," *Algorithmica,* vol. 6, pp. 816-825, 1991.
[20] J.H.M. Korst, P. Sanders, and S. Egner, "Fast Concurrent Access to Parallel Disks," *Proc. SIAM Symp. Discrete Algorithms,* Jan. 2000.
[21] V.S. Pai and A. Schäeffer, and P.J. Varman, "Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging," *Theoretical Computer Science,* vol. 128, nos. 1-2, pp. 211-239, June 1994.
[22] R.H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating Systems Principles,* pp. 79-95, Dec. 1995.
[23] D.D. Sleator and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Comm. ACM,* vol. 28, no. 2, pp. 202-208, Feb. 1985.
[24] P.J. Varman and R.M. Verma, "Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 12, pp. 1262-1275, Dec. 1999.
[25] J.S. Vitter and E.A.M. Shriver, "Optimal Algorithms for Parallel Memory I: Two-Level Memories," *Algorithmica,* vol. 12, nos. 2-3, pp. 110-147, 1994.

Mahesh Kallahalla received the BTech degree from the Indian Institute of Technology, Chennai, in 1995. He received the MS and PhD degrees from Rice University, in 1997 and 2001, respectively. He is a researcher at Hewlett Packard Laboratories in Palo Alto, California. His research interests currently are in the general area of storage systems, including external memory algorithms, storage system security, and self-managing storage systems. He is a member of the IEEE, ACM, and SIAM.

Peter J. Varman received the BTech degree from the Indian Institute of Technology, Kanpur, and the MS and PhD degrees from the University of Texas at Austin, all in electrical engineering. He is currently on the faculty of Rice University. He has held visiting positions at IBM's T.J. Watson and Almaden Research Centers, Nanyang Technological University, Singapore, and Duke University. His research interests are in the areas of parallel computing, parallel I/O, resource scheduling algorithms, multimedia systems, and temporal and spatial databases. Dr. Varman is a senior member of the IEEE, an associate editor of the *IEEE Transactions on Computers*, and a member of the ACM and the New York Academy of Sciences.