

Optimal prefetching and caching for parallel I/O systems *

Mahesh Kallahalla[†]
Hewlett-Packard Labs
1501 Page Mill Rd.
Palo Alto CA, 94304
maheshk@hpl.hp.com

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

Abstract

We address the problem of prefetching and caching in a parallel I/O system and present a new algorithm for optimal parallel-disk scheduling. Traditional buffer management algorithms that minimize the number of I/O disk accesses, are substantially suboptimal in a parallel I/O system where multiple I/Os can proceed simultaneously.

We present a new algorithm SUPERVISOR for parallel-disk I/O scheduling. We show that in the off-line case, where apriori knowledge of all the requests is available, SUPERVISOR 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 on-line case, we study SUPERVISOR in 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 SUPERVISOR, 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 .

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 I/O latency. For instance, results presented in this paper show that traditional caching strategies can under-utilize available bandwidth and thereby do not scale well, leading to excessive I/O service time. We need to design new algorithms for managing parallel I/O

*Supported in part by the National Science Foundation under grant CCR-9704562 and a grant from the Schlumberger Foundation.

[†]This work was done while the author was a student at the Dept. of ECE, Rice University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA July 4-6 2001, Crete Island, Greece

Copyright 2001 ACM 0-89791-88-6/97/05 ...\$5.00.

resources, with the explicit intention of exploiting I/O parallelism.

Prefetching and caching are intuitive techniques for improving I/O performance by utilizing the main-memory I/O buffer present in I/O systems. Blocks can be cached in the I/O buffer so that future requests to the same data can be serviced from main memory instead of needing to access (the much slower) disk. Additionally, the I/O buffer can also be used to enable prefetching, a crucial mechanism that can significantly increase parallel I/O performance. While a read progresses on one disk, concurrent reads can be started on other disks to prefetch data that are required later. These prefetched blocks are held in the I/O buffer till needed.

This paper deals with prefetching and caching algorithms for parallel I/O systems. Our 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. To exploit I/O parallelism we need to design prefetching algorithms that schedule reads carefully, so that the most useful blocks are fetched in advance. To exploit locality in I/O requests, we need to design caching strategies that retain the most valuable blocks in the buffer when the need for eviction arises. The combined objectives of using the multiple disks and using the common I/O buffer to hold prefetched and cached blocks make the problem of designing prefetching and caching algorithms for parallel I/O interesting. Prefetching and caching in parallel I/O systems is fundamentally different from that in systems with a single disk, thereby necessitating new algorithms to handle it [?, ?]. We will present a new priority-controlled greedy algorithm, SUPERVISOR, 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 [?]: the I/O system consists of D independent disks that can be accessed in parallel, and a buffer of capacity M through which all disk accesses occur. The computation requests data in *blocks*; a block is the unit of disk access. 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. The computation waits for data from the I/O system only when the data is not available in the buffer. Additionally when an I/O is initiated on one

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

(a) Using MIN as eviction policy

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

(b) Optimizing for multiple disks

Figure 1: Influence of replacement strategy on I/O schedule length

disk, blocks can be concurrently fetched from other disks. The number of parallel I/Os that are issued is the measure of performance in this model.

In this work we consider the I/O buffer to be a shared resource, capable of buffering blocks from any disk. Since the buffer is shared by all disks it is possible to allocate buffer space unevenly to different disks to meet the changing load on different disks. However, this freedom in allocating buffer space makes the buffer management problem more difficult. The buffer management algorithm 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 make use of the available bandwidth, it may seem preferable to have a large number of disks busy during an I/O. However excessive prefetching may fill up the shared buffer with prefetched blocks, which may not be used till much later. Such blocks have the adverse effect of not only causing unnecessary I/Os for blocks which may be evicted, but also choking the buffer and reducing the parallelism in fetching more immediate blocks. In an earlier work [?], we showed that even when the problem is reduced to just prefetching decisions, it is not trivial: even in the case when each block is accessed only once, the problem of deciding which blocks to fetch in an I/O is non-intuitive. In the general case considered in this paper the buffer manager has to deal with the additional complexity of deciding which blocks to cache and which to evict, in addition to 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.

If the application requests are known only when the data is 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*, an intuitive form of lookahead, which provides to the prefetching algorithm sub-sequences of future requests. Global lookahead is an abstraction of the lookahead available when applications provide hints to the prefetcher [?], or when a speculative execution of the application provides a window into potential future I/O requests [?]. Our notion of lookahead is a similar to *strong lookahead* [?], introduced to study caching in sequential I/O systems. We consider both online algorithms which use bounded lookahead, as well as off-line algorithms which use apriori knowledge of the entire reference string to construct the I/O schedule.

Traditional paging 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 off-line caching algorithm MIN [?], 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 3 disks and a buffer of size 6. 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 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. The actual set of three blocks determines the overall length of the I/O schedule.

Figure 1 (a) 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 3 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 2, shown in Figure 1 (b).

The example highlights the point that in parallel I/O systems, caching algorithms need to consider I/O parallelism in addition to issues in traditional sequential caching. This example can be generalized to prove that irrespective of the prefetching policy, any algorithm that uses MIN for page replacement can perform $\Omega(D)$ 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 possible ignoring I/O parallelism. The algorithm SUPERVISOR, presented later in this paper, makes optimal caching decisions in the off-line case. 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.

Even if there are no caching decisions to be made, the problem of deciding when any block ought to be fetched is quite interesting. For instance, 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. [?] considered the problem of scheduling such read-once reference strings. As this scheduling is pertinent to this paper, let us intuitively illustrate the problem of I/O scheduling without interference from caching.

Consider the following example of an I/O system with 3 disks and an I/O buffer of capacity 6. Let the blocks labeled a_i (respectively b_i, c_i) be placed on disk A (respectively B, C), and the reference string be

$$\langle a_1 a_2 a_3 a_4 b_1 c_1 a_5 b_2 c_2 a_6 b_3 c_3 a_7 b_4 c_4 a_8 c_5 c_6 c_7 \rangle$$

Figure 2 shows the I/O schedule constructed by an in-

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

Figure 2: Greedy-in-order schedule

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

Figure 3: Minimal I/O length schedule

tuitive greedy algorithm that always fetches blocks in the order of the reference string, and maximizes the disk parallelism at each I/O step. In the first step blocks a_1 , b_1 , and c_1 are fetched concurrently in one I/O. When block a_2 is requested, blocks a_2 , b_2 , and c_2 are fetched in parallel in step 2. Subsequently the buffer contains 5 blocks: a_2 , b_1 , b_2 , c_1 , and c_2 . Next 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 9.

Figure 3 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 7, better than the previous schedule.

The above example illustrates that optimizing the total number of I/Os in a parallel I/O system cannot be done based solely on simple 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 buffer management algorithm SUPERVISOR presented in this paper, is an optimal off-line prefetching and caching algorithm in the parallel disk model. That is, when SUPERVISOR has apriori knowledge of the entire reference string, it generates a schedule of minimal length. This is the first optimal scheduling algorithm for the parallel disk model that we are aware of. In an online scenario, SUPERVISOR uses available lookahead to make prefetching and caching decisions dynamically. Specifically, we show that the competitive ratio of SUPERVISOR is $\Theta(M - L + D)$ when $L \leq M$, and $\Theta(MD/L)$ when $L > M$, where D is the number of disks, M the size of the I/O buffer, and L the number of distinct references in each lookahead window.

The rest of the paper is organized as follows. A brief survey of related publications is presented in Section 1.1. We formally introduce the problem in Section 2. Details of our scheduling algorithm are presented in Section 3. In Section 4 we present bounds on the performance of SUPERVISOR both in the online case, as a function of the available lookahead, as well as in the off-line situation.

1.1 Related work

Classical buffer management in the single-disk model deals primarily with optimizing eviction decisions to minimize the number of I/Os performed. Online buffer man-

agement for sequential IO systems in the framework of competitive analysis was first studied in [?], followed by studies using extended models, which incorporated lookahead [?, ?] or randomization [?, ?]. An analysis of paging algorithms with strong lookahead (similar to global lookahead defined here) was presented in [?] for the single disk situation. Recent formal work on using prefetching to overlap computation with I/O in single disk systems was introduced by [?], where they showed that aggressive prefetching can reduce total elapsed time. [?] presented an integer programming approach to optimizing the total time in such a stall model.

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. [?, ?, ?, ?]). For read-once reference strings, analysis of algorithms exploiting a randomized data placement was studied in [?, ?, ?, ?]. In a generalization of the stall model to parallel disks, [?] designed a sophisticated off-line approximation algorithm to service read-many reference strings. They showed that the algorithm, reverse-aggressive, is near optimal for typical system parameters (memory size and computation time). [?] considered I/O scheduling in a distributed buffer configuration, where each disk has a private buffer. They presented algorithm PMIN, a generalization of MIN, and showed that the algorithm is optimal for this buffer configuration. Other empirical studies on using prefetching, based on hints provided to the system, to improve parallel I/O performance include [?, ?]. The underlying greedy prefetching heuristic need to be curtailed to perform shallow prefetching to prevent prefetched blocks from choking the buffer and affecting the LRU based caching.

In [?], we studied the more restricted problem of scheduling read-once reference strings in a deterministic setting in the context of global lookahead. We developed an algorithm L-OPT that generates the smallest length I/O schedule in the offline case, and has the best possible competitive ratio (among all algorithms with the same lookahead) for any lookahead. The algorithm SUPERVISOR, developed in this paper, generalizes these ideas to the problem of scheduling read-many reference strings. Later, at the end of Section 3, we present a discussion on the relation between the problems of scheduling read-once and read-many reference strings, and the two algorithms we developed.

2. Problem specification

The *reference string* is a sequence of I/O requests, $\Sigma = \langle r_1, \dots, r_N \rangle$, with several references possibly 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 at any time no more than one block is fetched from any one disk, and the number of blocks in the buffer is no more than M . In the off-line case the algorithm has information 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.

3. Algorithm Supervisor

In this section we introduce our parallel I/O scheduling algorithm, SUPERVISOR. SUPERVISOR is a priority controlled greedy scheduling algorithm that fetches blocks from as many disks as possible in an I/O, while ensuring that the buffer contains the blocks with the highest priority. SUPERVISOR assigns a priority to each reference in the lookahead. The priority of a block is a measure of how much the fetching of a block can be delayed: 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.

Intuitively, SUPERVISOR greedily schedules I/Os and attempts to utilize all I/O slots available during a parallel read. This is done to allow SUPERVISOR to utilize future lookahead information as soon as it is available and avoid situations where it unnecessarily wasted slots in previous I/Os. However, aggressive greedy prefetching has to be tempered by the fact that doing so could cause the buffer to fill up and prevent useful caching and further prefetching. We use two mechanisms to prevent this from happening: (a) issue prefetches for blocks close to their references so that prefetched blocks do not wastefully occupy buffer space, (b) avoid caching a block if there is any later free I/O slot available which can be used to fetch the block.

The I/O buffer is used both to cache blocks since their previous reference and to hold prefetched blocks till they are referenced. Among possible candidates of blocks to cache, we would like to cache the block which would occupy buffer space for a smaller duration. Hence the question is: given that at some time we would like two blocks in the buffer, which of these is it preferable to have cached, and which should be fetched now? We prefer caching the block whose previous reference is closer to the current time, as this would reduce the buffer pressure between the two previous accesses. When prefetching, we prefer to issue prefetches for blocks close to where they are actually referenced. These choices in ordering blocks to be fetched or evicted are made by SUPERVISOR through its priority assignment scheme.

Given the priorities of references in the lookahead, the prefetching engine of algorithm SUPERVISOR determines the blocks to be fetched in any I/O. This part of the algorithm is presented in Figure 4. The details of the priority assignment scheme are presented in Figure 5. These specifications make use of the the following definitions.

- The current lookahead is denoted by $\mathcal{L} = \langle r_i, \dots, r_j \rangle$. $block(r)$ is used to denote the block accessed in reference 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 is assigned 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. In this case the priority of the block is defined as the priority of the next reference to that block; when there is no reference to that block in the lookahead, a low priority, $-i$, is assigned to that block, where i is the index of the last reference to that block (in the past).

The prefetching engine of SUPERVISOR performs I/Os only on demand; that is, it performs I/O only when the referenced block is not present in the buffer. By doing so SUPERVISOR can defer its decisions till the latest time and make use of the largest lookahead available. When an I/O is to be performed, the blocks from each disk with the highest priority (H) are potential candidates to be fetched. Among these blocks and the blocks 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 lower priorities are evicted if there is not enough space free in the buffer.

The priority assignment routine, the more interesting part of SUPERVISOR, assigns priorities to each reference in the lookahead. The details of the assignment routine are presented in Figure 5. The priority assignment scheme attempts to set the priority of every block as low as possible, subject to system constraints. First, since at most one block can be fetched from one disk in an I/O, at any time each block from the disk has a unique priority. 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 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. Among those blocks which do not occur again, we use LRU (least recently used) policy to assign priorities.

The routine examines subsets of the lookahead consisting of M distinct references and then assigns priorities to one block from each disk. The intuition behind the priority assignments can be illustrated by considering the largest subsequence of the lookahead including the last reference and having at most M distinct references. All blocks which are assigned the smallest priority should belong to this set. Otherwise there will be some reference such that M or more blocks referenced after it have a higher, or same, priority. The question we then ask is: 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 indicate that between these blocks we would rather not cache that block, since a lower priority indicates that we prefer

Algorithm SUPERVISOR

On a request r , algorithm SUPERVISOR 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
  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  $H \cup B$ ;
      in the case of ties the block occurring earlier in  $\Sigma$  is selected
  The request  $r$  is then serviced
```

Figure 4: Algorithm SUPERVISOR

Routine to assign priorities to references

This routine is used to assign priorities to references $\langle r_j, \dots, r_k \rangle$. 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 to the same block.

Initialize `lowestPriority` to 1, all other counts to 0, and sets to ϕ .

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 $\mathcal{P}[\text{disk}(r_i)]$ such that `block(r) = block(r_i)` then

Replace r in $\mathcal{P}[\text{disk}(r_i)]$ 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 $\mathcal{P}[\text{disk}(r_i)]$ with key `previous(r_i)`

Increment `numberOfBlocksPlaced`

Figure 5: Routine to assign priorities

caching other blocks over this one.

Priorities are assigned starting from the lowest priority (1). At any time, a subset of the references in the lookahead consisting of M distinct references is used to decide the blocks that are assigned a particular priority. Specifically, the variable `numberOfBlocksPlaced` keeps count of the number of blocks in the subset, triggering a priority assignment when it reaches M . While scanning the references in the lookahead, SUPERVISOR first generates a local ordering of blocks from the same disk, indicating which block among the blocks from this disk is to be assigned the smallest priority. This ordering for each disk is kept in a priority queue $\mathcal{P}[\cdot]$. There is one entry in $\mathcal{P}[d]$ for each distinct reference from disk d . The references in any $\mathcal{P}[d]$ are ordered based on the index of the previous occurrence of that block outside the subsequence being used to decide this priority. The current lowest assignable priority is tracked by `lowestPriority`. When M blocks in the lookahead to which priorities have not been assigned are examined, the counter `numberOfBlocksPlaced` reaches M , and from each $\mathcal{P}[d]$ the block with

the earliest previous reference (smallest key) is assigned the current lowest priority.

Algorithm SUPERVISOR 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 change only when the lookahead window is extended.

In [?] we had presented an algorithm L-OPT for scheduling read-once reference strings. In that paper the reference string does not contain repeated accesses to the same block, implying that there is no use in retaining a block in the buffer once its request has been serviced. Algorithmically L-OPT and SUPERVISOR share a similar structure: both are priority controlled greedy algorithms. Both algorithms have

a similar “greedy” component because both use the same model of lookahead, and they attempt to not waste I/O slots. They differ significantly in the way the priorities are assigned. In the read-once case it can be seen that from any disk, a block occurring earlier than another has a higher priority. This observation simplifies the priority assignment in the read-once case as a new block that becomes visible due to extending the lookahead is always assigned the lowest priority; the resulting effect is then cascaded to the priorities of rest of the blocks. However no such ordering can be decided in the read-often case, resulting in a relatively more complex priority assignment scheme. But if all references were to distinct blocks, SUPERVISOR would generate the same schedule as L-OPT.

4. Analysis

In this section we shall analyze the performance of SUPERVISOR in terms of the length of the schedule it generates. We shall show that when SUPERVISOR has knowledge of the entire reference string in advance it generates a schedule of minimal length; that is, it is an *optimal off-line* I/O scheduling algorithm. In the case when SUPERVISOR has only a limited view of future requests we present bounds on the ratio of the number of I/Os done by SUPERVISOR to that of the optimal off-line while serving the same sequence of requests.

We shall present only the main lemmas and an outline of the proofs here: the details are presented in the appendix. Given a reference string Σ , let OPT denote the minimal-length I/O schedule to service Σ .

Offline situation

We shall first show that SUPERVISOR generates the minimal length schedule to service any reference string of length L . Note that in this case the entire reference string is in the initial lookahead of SUPERVISOR, and hence the priorities are all assigned at once. To do the analysis in this case we shall start by showing some properties of the schedule generated by SUPERVISOR.

Property 1 If SUPERVISOR is given the entire reference string apriori, the number of I/Os done by SUPERVISOR 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 is the same as the length of the optimal off-line schedule. This will be used together with Property 1 to show that the length of the schedule generated by SUPERVISOR is of the same length as OPT. We shall start by characterizing the blocks which have a priority p . Let us recursively define a *phase* as follows.

Definition 1

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

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

Property 2 The priority of all references in $\text{earliest}(i)$ 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. These lemmas will be used to show the following theorem that in the case when SUPERVISOR has apriori knowledge of the entire reference string, SUPERVISOR generates a schedule of the same length as OPT. Without any loss in generality we assume that in OPT no block is fetched and evicted before at least one reference to it is serviced. This will simplify the comparison of the schedule generated by SUPERVISOR and OPT. Let the length of OPT be T_{OPT} .

The theorem is proved by inductively showing that OPT can be repeatedly transformed into a series of T_{OPT} schedules, each derived from the previous one and of the same length as OPT, such that in the k th schedule all references fetched beyond the $(T_{\text{OPT}} - k)$ th I/O match those in the corresponding *earliest* set. Then the length of OPT, T_{OPT} , has to be at least that of the number of earliest sets that are possible in the given reference string. Correspondingly, the maximum priority of any block in the reference string is at most T_{OPT} , by Property 2, thereby completing the proof together with Property 1.

The inductive proof then shows how the schedule $\text{OPT}^*(k+1)$ can be constructed from previous schedule $\text{OPT}^*(k)$. In making this transformation, only the references in the $(T_{\text{OPT}} - (k+1) - 1)$ th I/O are changed from $\text{OPT}^*(k)$ to $\text{OPT}^*(k+1)$. The main idea used here is to compare the reference a fetched by OPT and the corresponding reference from the same disk, b , in $\text{earliest}(k+1)$. By definition of $\text{earliest}(k+1)$, the previous access to reference b is earlier than the previous access to block a . Hence at any time that b is present in the buffer, it can equivalently be replaced by a in the buffer. Additionally, since they are both fetched from the same disk, their I/Os can also be exchanged. Thus we argue that in the $(T_{\text{OPT}} - (k+1) - 1)$ th I/O, we can fetch b instead of a to match the requirements of the induction hypothesis. The detailed proof, presented in Section A.1 of the Appendix handles the various technical details and cases possible in effecting this transformation.

THEOREM 1. *Given a reference string Σ of length L , SUPERVISOR, with L block lookahead, performs the minimal number of I/Os to service Σ .*

Immediately, as a corollary, it can be seen that when the entire reference string is known to SUPERVISOR it generates the minimal length schedule to service it.

COROLLARY 1. *SUPERVISOR is the optimal off-line parallel I/O scheduling algorithm.*

Online situation

In this section, we shall characterize the performance of online algorithms using the competitive ratio [?] 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 off-line 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 at least 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 $\mathcal{L} = \langle r_i, \dots, r_j \rangle$ 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 [?], which was used in the context of on-line caching algorithms for sequential systems.¹

We shall next provide bounds on the performance of SUPERVISOR 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 $L > M$. The details of the proof are presented in Section A.2 of the Appendix.

THEOREM 2. *The competitive ratio of SUPERVISOR is*

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

To do the analysis, we partition the entire reference string into subsequences consisting of M distinct blocks called *phases*: a phase is a maximal length subsequence of the reference string consisting of references to at most M distinct blocks, with the first phase starting with the first reference. Let the total number of phases in the reference string be N . Next we categorize all references into stale and clean based on whether the same block has been accessed in the previous phase or not. Let the i th phase be denoted by $\text{phase}(i)$, and let the number of clean blocks in $\text{phase}(i)$ be c_i .

In the range when $L \leq M$, we show that the competitive ratio of SUPERVISOR is $O(M - L + D)$. The main idea of this proof is to note that after servicing all the references of a phase all the blocks in the buffer of SUPERVISOR are either references from that phase, or some blocks from the first lookahead of the subsequent phase. We use this to show that the number of I/Os done by SUPERVISOR in any phase is at most $M - L + c_i$, by showing that in the first lookahead of the phase, SUPERVISOR does at most c_i I/Os as it caches all other blocks from the previous phase. The proof is then completed by showing that the number of I/Os done by OPT in the same phase is at least $\max\{c_i/2D, 1\}$. This is done by showing that the total number of blocks that OPT needs to fetch is at least $\sum_i c_i$.

In the range $L > M$, we show that the competitive ratio is $O(MD/L)$. The proof is based on the fact that OPT could cache at most M blocks in the buffer to service references in a lookahead window. Hence if the lookahead consists of more than $2M$ references, then OPT will still have to do

¹ 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.

$\Omega(L/D)$ I/Os to service the lookahead, while the difference in the number of I/Os done by OPT and SUPERVISOR is at most $2M$ in that lookahead. When the lookahead is less than $2M$ then we show that the ratio is at most $O(D)$ by arguing that the total number of blocks fetched by OPT and SUPERVISOR are comparable.

We can show that the bounds on the competitive ratio are tight by constructing reference strings for which SUPERVISOR reaches the the upper bound.

5. Summary

In this paper we addressed a generalization of the sequential paging problem [?] 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 [?] 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 SUPERVISOR for optimizing prefetching and caching decisions in the parallel I/O model. We showed that in the off-line case, where apriori knowledge of all the accesses is available, SUPERVISOR performs the minimum number of I/Os to service the I/O requests. This is the first optimal algorithm in the parallel model. In the online case we show that the competitive ratio of SUPERVISOR, 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 .

6. References

- [1] 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.
- [2] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18(3):283–305, July 1997.
- [3] 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 Computer Science*, pages 454–462. ACM Press, May 1998.
- [4] L. M. Baptist and T. H. Cormen. Multidimensional, multiprocessor, out-of-core FFTs with distributed memory and parallel disks. In *Proc. of the 11th Annual ACM Symp. on Parallel Algorithms and Architectures*, June 1999.
- [5] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, June 1996.
- [6] R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive parallel disk prefetching and buffer

- management. *Journal of Algorithms*, 36(2):152–181, Aug. 2000.
- [7] L. A. Belady. A study of replacement algorithms for a virtual storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [8] D. Breslauer. On competitive on-line paging with lookahead. In *Proc. of the 13th Annual Symp. on Theoretical Aspects of Computer Science*, volume 1046 of *LNCS*, pages 593–603. Springer Verlag, Feb. 1996.
- [9] P. Cao, E. W. Felten, 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*, pages 188–197. ACM Press, May 1995.
- [10] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, 1999.
- [11] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, Dec. 1991.
- [12] M. Kallahalla and P. J. Varman. Optimal read-once parallel disk scheduling. In *Proc. of Sixth ACM Wkshp. on I/O in Parallel and Distributed Systems*, pages 68–77, Atlanta, GA, 1999. ACM Press.
- [13] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 5(3):79–119, Mar. 1988.
- [14] T. Kimbrel and A. R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. of Computing*, 29(4):1051–1082, 2000.
- [15] L. A. McGeoch and D. D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6:816–825, 1991.
- [16] J. H. M. K. P. Sanders, S. Egner. Fast concurrent access to parallel disks. In *Proc. of the SIAM Symposium on Discrete Algorithms*, Jan. 2000.
- [17] V. S. Pai, A. A. Schäffer, and P. J. Varman. Markov analysis of multiple-disk prefetching strategies for external merging. *Theoretical Computer Science*, 128(1–2):211–239, June 1994.
- [18] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles*, pages 79–95, Dec. 1995.
- [19] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, Feb. 1985.
- [20] 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.
- [21] J. S. Vitter and E. A. M. Shriver. Optimal algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

APPENDIX

A. Detailed proofs

A.1 Off-line optimality of Supervisor

THEOREM 1. *Given a reference string Σ of length L , SUPERVISOR, 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 $\text{OPT}^*(k)$, k from $T_{\text{OPT}}, \dots, 1$, such that schedule $\text{OPT}^*(k)$ is the same length as OPT, and for any reference r that is fetched in the i th I/O in $\text{OPT}^*(k)$, $i \geq k$, r is in $\text{earliest}(i)$. The theorem will then follow due to Properties 1 and 2.

For the base case we will show how to construct $\text{OPT}^*(1)$ from OPT. Let $\text{OPT}^*(1)$ be initialized to OPT: we shall transform the last I/O of $\text{OPT}^*(1)$ so that the induction hypothesis holds.

Let p be the reference fetched by OPT in the last I/O from some arbitrary disk d and q be the reference from the same disk with the smallest index in $\text{earliest}(1)$. If no such reference exists, then we denote p and q by ϕ . But we will argue that if $q = \phi$ then $p = \phi$. This follows from the requirement that any reference should be present in the buffer before a request for it can be serviced. If $p \neq \phi$ then, since it is fetched in the last I/O, all references occurring after it should be present in the buffer after this I/O. There can at most be M such references (including p) as the buffer capacity is M . Hence p should be in $\text{phase}(1)$, and correspondingly in $\text{earliest}(1)$.

From the previous discussion if $q = \phi$ then in $\text{OPT}^*(1)$ we do not fetch anything from disk d in the last I/O, which will match what OPT fetches from disk d and hence is a valid schedule. Three other cases are possible depending upon the relation between the indices of p and q .

Case 1: p occurs earlier than q in Σ .

Note that by the definition of $\text{earliest}(1)$, the previous reference to q is earlier than the previous reference to p . In this case OPT fetches p in the last I/O. This means that q is either cached since its previous reference or it is fetched sometime after its previous reference.

If q is cached, then in $\text{OPT}^*(1)$ we set q to be fetched in the last I/O and arrange p to be cached instead. If q were fetched sometime after its previous reference, then in $\text{OPT}^*(1)$ we exchange the I/Os for p and q . These transformations will still lead to valid schedules because the previous reference to q is earlier than the previous reference to p .

Case 2: p occurs after q in Σ .

Again two cases are possible: either q is cached since its previous reference or it is fetched sometime after its previous reference. If q is cached, then in $\text{OPT}^*(1)$ we set q to be fetched in the last I/O and arrange p to be cached instead. If q were fetched sometime after its previous reference, then in $\text{OPT}^*(1)$ we exchange the I/Os for p and q . These transformations will still lead to a valid schedule because there are at most $M - 2$ blocks other than $\text{block}(p)$ and $\text{block}(q)$ referenced between the reference to p and q : both p and q are in $\text{phase}(1)$ as q is in $\text{phase}(1)$ by the definition of $\text{earliest}(1)$.

Case 3: $p = \phi$.

In this case, in $\text{OPT}^*(1)$, we set q to be fetched in the last I/O and not cache it (if it was cached in OPT) or cancel the previous I/O for it (if it was fetched after its previous reference). Again this transformation will lead to a valid schedule as the number of blocks referenced after q is at most $M - 1$, since q is in $\text{phase}(1)$.

Thus, OPT can be transformed into a valid schedule $\text{OPT}^*(1)$ such that all the references fetched in the last I/O of $\text{OPT}^*(1)$ are in $\text{earliest}(1)$; and the length of $\text{OPT}^*(1)$ is the same as that of OPT. For the induction hypothesis, we assume that this can be done up to $\text{OPT}^*(k)$; that is, $\text{OPT}^*(k-1)$ can be transformed into a valid schedule $\text{OPT}^*(k)$, of the same length, such that all references fetched in the i th I/O of $\text{OPT}^*(k)$, $i \leq k$, are in $\text{earliest}(i)$. We shall show that $\text{OPT}^*(k)$ can be transformed to $\text{OPT}^*(k+1)$ such that the hypothesis holds.

The transformation follows along the same lines as the base case. We take $\text{OPT}^*(k)$ and arrange the references in the $T_{\text{OPT}} - (k+1) + 1$ th I/O so that they match the earliest references from the corresponding disks in $\text{earliest}(k+1)$.

Again, let p be the reference fetched from some arbitrary disk d by OPT in the $T_{\text{OPT}} - (k+1) + 1$ th I/O and q be the reference from the same disk with the smallest index in $\text{earliest}(k+1)$. As in the base case we can argue that if $q = \phi$ then $p = \phi$. If $p \neq \phi$ then all references in Σ_{k+1} occurring after it should have been cached (By the induction hypothesis none of these references are fetched in later I/Os). Hence p should be in $\text{phase}(k+1)$, and correspondingly in $\text{earliest}(k+1)$, which means that $q \neq \phi$.

Hence, again if $q = \phi$ then we set $\text{OPT}^*(k+1)$ to fetch nothing from disk d in the last I/O, which will match what $\text{OPT}^*(k)$ fetches from disk d and hence is a valid schedule. Three other cases are possible depending upon the relation between the indices of p and q .

Case 1: p occurs earlier than q in Σ .

This case can be handled exactly like in the base case: If q is cached, then in $\text{OPT}^*(k+1)$ we set q to be fetched in the last I/O and arrange p to be cached instead. If q were fetched sometime after its previous reference, then in $\text{OPT}^*(k+1)$ we exchange the I/Os for p and q .

Case 2: p occurs after q in Σ .

Again two cases are possible: either q is cached since its previous reference or it is fetched sometime after its previous reference. If q is cached, then in $\text{OPT}^*(k+1)$ we set q to be fetched in the last I/O and arrange p to be cached instead. If q were fetched sometime after its previous reference, then in $\text{OPT}^*(k+1)$ we exchange the I/Os for p and q .

We shall prove that these transformations will still lead to a valid schedule by showing that the number of distinct blocks occurring between p and q that are fetched before p is fetched in $\text{OPT}^*(k)$ is at most $M-2$. Note that by the induction hypothesis in $\text{OPT}^*(k)$ all blocks not in Σ_{k+1} are fetched after the $T_{\text{OPT}} - k + 1$ th I/O. Moreover, since q is in $\text{phase}(k+1)$, so is p , indicating that the number of distinct blocks in Σ_{k+1} between q and p is at most $M-2$.

Case 3: $p = \phi$.

In this case, in $\text{OPT}^*(1)$, we set q to be fetched in the last I/O and not cache it (if it was cached in OPT) or cancel the previous I/O for it (if it was fetched after its previous reference). As in the previous case, this transformation will lead to a valid schedule as the number of blocks in Σ_{k+1} referenced after q is at most $M-1$. \square

A.2 Competitive ratio of Supervisor

LEMMA 1. *The number of I/Os done by SUPERVISOR to fetch the first set of L distinct references of a phase 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 (a) blocks in the buffer which are present in the lookahead have a higher priority than blocks which are not present in the lookahead, and (b) 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 phase, once a block has been fetched, it will not be evicted 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 phase, if a block is present in the buffer of SUPERVISOR, then it is either a block that is referenced in that phase, 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 phase².

Consider the I/Os done by SUPERVISOR in $\text{phase}(i)$. From the previous argument, the number of I/Os done by SUPERVISOR to service the first lookahead window is no more than the number of clean blocks in this lookahead. \square

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 [?]; we briefly repeat it here for convenience. Let n_i clean blocks of $\text{phase}(i)$ be present in OPT's buffer at the start of $\text{phase}(i)$. Hence the number of blocks fetched by OPT in $\text{phase}(i)$ is at least $c_i - n_i$. Also since n_{i+1} clean blocks are present in OPT's buffer at the end of $\text{phase}(i)$, OPT must have fetched at least n_{i+1} blocks in $\text{phase}(i)$. Hence the number of blocks fetched by OPT in $\text{phase}(i)$ is at least $(c_i - n_i + n_{i+1})/2$. This, when summed over all phases, 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. \square

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

PROOF. Consider the I/Os done by SUPERVISOR in $\text{phase}(i)$. From Lemma 1 the number of I/Os done by SUPERVISOR to service the first lookahead window is no more than the number of clean blocks in this lookahead. There are at most $M - L$ other blocks referenced in the rest of the phase, and hence the number of I/Os done by SUPERVISOR in the rest of the phase is at most $M - L$. Hence the total number of I/Os done by SUPERVISOR in $\text{phase}(i)$ 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 phases in Σ , or $\Omega(N)$, where N is the total number of phases in the reference string; hence the competitive ratio of SUPERVISOR will be $O(D + M - L)$.

²Sequential paging algorithms with this property have been previously referred to as marking algorithms [?].

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 phases 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 set of two phases, which gives the second bound of $\Omega(N)$ on the total number of I/Os done by OPT to service Σ . \square

LEMMA 4. *When $L > M$ the competitive ratio of SUPERVISOR is $O(MD/L)$.*

PROOF. Let us divide the reference string Σ into subsequences $\mathcal{L}_1, \dots, \mathcal{L}_n$, where \mathcal{L}_1 is the lookahead available to SUPERVISOR initially, and \mathcal{L}_i is the lookahead window available to SUPERVISOR 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 \mathcal{L}_i .

Consider the optimal length schedule OPT to service Σ . Let us partition the schedule into sub schedules, each consisting of a sequence of contiguous I/Os of the original schedule. Let the i th sub-schedule 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 \mathcal{L}_i . Thus in this case, if the number of lookaheads is N , $T_{\text{OPT}} = \Omega(NL/D)$. The number of I/Os done by SUPERVISOR to service any \mathcal{L}_i is less than $2M$ more than the number of I/Os in S_i . Thus $T_{\text{Super}} \leq T_{\text{OPT}} + 2NM$, 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 SUPERVISOR in the first lookahead of any phase is at most the number of clean blocks in that lookahead. However, when the lookahead is more than M , the entire phase is a part of the lookahead. Hence the total number of I/Os done by SUPERVISOR 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 SUPERVISOR to the number of I/Os done by OPT is at most $O(D)$ when $M < L < 2M$. \square