INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeib Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
RICE UNIVERSITY

Prefetching and Buffer Management for Parallel I/O Systems

by

Mahesh Kallahalla

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Peter J. Varman, Chair
Associate Professor in Electrical and Computer Engineering

Moshe Y. Vardi
Karen Ostrum George Professor in Computational Engineering
Chair, Department of Computer Science

Willy Zwolnepoel
Noah Harding Professor of Computer Science
Professor of Electrical and Computer Engineering

Houston, Texas

September, 2000
To Appa, Amma, and Putti.
Prefetching and Buffer Management for Parallel I/O Systems

Mahesh Kallahalla

Abstract

In parallel I/O systems the I/O buffer can be used to improve I/O parallelism by improving I/O latency by caching blocks to avoid repeated disk accesses for the same block, and also by buffering prefetched blocks and making the load on disks more uniform. To make best use of available parallelism and locality in I/O accesses, it is necessary to design prefetching and caching algorithms that schedule reads intelligently so that the most useful blocks are prefetched into the buffer and the most valuable blocks are retained in the buffer when the need for evictions arises.

This dissertation focuses on algorithms for buffer management in parallel I/O systems. Our aim is to exploit the high parallelism provided by multiple disks to reduce the average read latency seen by an application. The thesis is that traditional greedy strategies fail to exploit I/O parallelism thereby necessitating new algorithms to make use of the available I/O resources.

We show that buffer management in parallel I/O systems is fundamentally different from that in systems with a single disk, and develop new algorithms that carefully decide which blocks to prefetch and when, together with which blocks to retain in the buffer. Our emphasis is on designing computationally simple algorithms for optimizing the number of I/Os performed. We consider two classes of I/O access patterns, read-once and read-often, based on the frequency of accesses to the same data. With respect to buffer management for both classes of accesses, we identify fundamental bounds on performance of online algorithms, study the performance of intuitive strategies, and present randomized and deterministic algorithms that guarantee higher performance.
Acknowledgments

First, and foremost, I would like to heart-fully thank Dr. Varman for being an extraordinary advisor over the past five years. I cannot over-emphasize the support and guidance that he has provided, and am very grateful for that. I would also like to thank him for his time and patience during the marathon brain-storming sessions where most of the results were worked out; and for the lunches and dinners during these sessions.

Thanks are also due to Dr. Vardi and Dr. Zwaenepoel for being on my committee. I would especially like to thank them for their support in my job hunt, which was the final motivation to complete this dissertation.

I also thank Partha, Karthick, Suman, Srikrishna, and Mukkavalli for being great friends. Thanks also to my friends in and out of Rice – their lively company definitely made life outside school something to look forward to.

Finally, I would like to dedicate this thesis to my parents and sister for their unceasing support and for encouraging me to make my decisions.
Contents

Abstract iii
Acknowledgments iv
List of Illustrations vii

1 Introduction 1
1.1 Motivation ...................................................... 1
  1.1.1 Potential solutions ........................................... 2
  1.1.2 Prefetching and caching .................................... 3
1.2 System model .................................................. 4
1.3 Problem addressed ........................................... 6
1.4 Organization of this dissertation .............................. 7
1.5 Summary of results .......................................... 8
1.6 Related work .................................................... 13

2 System model 17
2.1 Model of I/O system and computation ....................... 17
2.2 Model of Lookahead ........................................ 19
2.3 Problem specification ......................................... 21
2.4 Performance metrics ........................................ 21
  2.4.1 Competitive ratio ........................................ 22
  2.4.2 Online ratio ............................................. 22

3 Scheduling read-once reference strings 24
3.1 Limits of lookahead ........................................... 27
3.2 Performance of intuitive algorithms .......................... 34
  3.2.1 Performance of PHASE .................................... 37
3.2.2 Performance of GREED .................................. 43
3.3 Exploiting a randomized layout .......................... 46
3.4 Optimal read-once parallel disk scheduling .......... 47
    3.4.1 Algorithm L-OPT ..................................... 48
    3.4.2 Analysis of L-OPT .................................... 52
3.5 Comparing on-line algorithms .......................... 62
    3.5.1 Comparing GREED, PHASE, and L-OPT .......... 67
3.6 Simulation study .......................................... 69

4 Scheduling read-often reference strings ................. 74
    4.1 Limits of lookahead .................................... 78
    4.2 Performance of intuitive algorithms ............... 79
    4.3 Exploiting a randomized layout...................... 80
    4.4 Optimal read-often scheduling ....................... 85
        4.4.1 Algorithm SUPERVISOR ......................... 85
        4.4.2 Performance of SUPERVISOR ................. 89

5 Conclusion .................................................. 102
    5.1 Directions for future research ...................... 102
        5.1.1 Deadline driven scheduling .................. 103
        5.1.2 Incorporating system constraints ............ 104
        5.1.3 Scheduling independent applications .......... 105
        5.1.4 Extended lookahead models .................. 105

Bibliography .................................................. 106
Illustrations

2.1 Model of the I/O system .................................................. 18
2.2 Global Lookahead ......................................................... 20

3.1 Greedy-in-order schedule ............................................. 25
3.2 Minimal I/O length schedule ........................................ 25
3.3 Illustration of Loaded and Balanced phases ..................... 29
3.4 Schedule generated by PHASE ....................................... 36
3.5 Schedule generated by GREED ....................................... 36
3.6 Reference string used to lower bound GREED ................. 45
3.7 Algorithm L-OPT ......................................................... 50
3.8 Routine to update priorities ......................................... 51
3.9 Reference string used to lower bound online scheduling algorithms 64
3.10 Performance of algorithms vs. percentage of hot-spotting blocks.
    Buffer = 250 block .................................................. 71
3.11 Performance of algorithms vs. percentage of hot-spotting blocks.
    Buffer = 1000 blocks .............................................. 72

4.1 Influence of replacement strategy on I/O schedule length .... 75
4.2 Classification of blocks in a phase .................................. 82
4.3 Algorithm SUPERVISOR ............................................. 87
4.4 Routine to assign priorities ......................................... 88
Chapter 1

Introduction

This dissertation addresses issues in utilizing multiple-disk I/O systems and develops new algorithms for optimizing I/Os on such parallel I/O systems. Results presented here show that buffer management in parallel I/O systems is fundamentally different from that in systems with a single disk. The thesis is that traditional greedy strategies, based on sequential algorithms, fail to exploit I/O parallelism in parallel I/O systems, thereby necessitating new algorithms to make best use of the available I/O resources. The I/O prefetching and caching algorithms presented here achieve a high performance by carefully managing the scheduling of prefetches and the allocation of buffer space between prefetched and cached blocks. Our emphasis is on designing algorithms, with low computational complexity, for optimizing the number of I/Os performed. In this dissertation we consider two classes of I/O access patterns – read-once and read-often – based on the frequency of repeated accesses to the same data. With respect to prefetching and caching strategies for both classes of accesses, we first identify fundamental bounds on performance of online algorithms, study the performance of intuitive sequential buffer management strategies, and then present randomized and deterministic algorithms that provide higher performance guarantees.

1.1 Motivation

The I/O subsystem is a critical bottleneck for many modern data-intensive applications such as geographical and seismic databases, streaming video, and visualization tools. The increasing imbalance between the speeds of processors and secondary storage devices has further aggravated the bottleneck. Due to
advances in computer architecture and parallel computing, the speed at which data can be processed has been rising at a fast pace. On the other hand, device-level advances in I/O systems, have been less dramatic, further aggravating the imbalance between the speed of I/O and computation. On the positive side, the size of the memory which is used to match the speed of the disk and the speed of the processor has been increasing, driven primarily by the decreasing cost of random access memory. Still, the increasing size of many modern workloads makes it prohibitively expensive to buffer the entire data set in main memory.

Parallel I/O systems have the potential to alleviate the I/O bottleneck. Parallel I/O systems based on multiple disks have been proposed to increase I/O performance and system availability [13], and current high-performance systems incorporate some form of parallel I/O. Higher performance can be obtained by concurrently accessing data from multiple disks. However it is a challenging problem to successfully exploit the higher bandwidth to reduce application I/O latency. For instance, our results indicate that traditional prefetching and caching strategies grossly under-utilize available I/O bandwidth and do not scale well, thereby leading to excessive I/O service time.

1.1.1 Potential solutions

One strategy to decrease the execution time of I/O intensive applications is to hide the I/O latency behind some computation. However, the achievable benefit is then limited by the amount of computation that can be performed in parallel with I/O. Especially in parallel I/O systems, there is greater scope for improvement in performance by overlapping I/O operations on multiple disks. This allows the latency of an I/O operation to be hidden behind the latency of another I/O. In a multiple disk system exploiting I/O parallelism better utilizes the available disks, which would otherwise idle while one I/O request is being serviced.

An intuitive mechanism to derive I/O parallelism is to distribute each block, the data access unit, across all disks in the system. However, this scheme has limited scalability with the number of disks since the unit of access from each
individual disk needs to be at least of a minimum size to amortize disk seek
time and rotational latency. Hence, to obtain high parallelism either the block
size or the buffer size must increase in proportion to the number of disks. This
increases the buffer requirement of an application. This form of speculatively
fetching additional data implicitly assumes that data located close together on
disk would be needed together by the application. This could lead to inefficient
usage of I/O resources in applications like databases and video-servers, where
the data access pattern during run-time cannot be determined when data is
created.

1.1.2 Prefetching and caching

Prefetching provides another technique to improve I/O parallelism, and hence
I/O performance, without artificially increasing the block size. While a read
progresses on one disk, reads can be started concurrently on other disks to
fetch data that would be required later. These prefetched blocks are cached in
the I/O buffer, from which future I/O requests can be serviced much faster as
main memory typically has orders of magnitude lower access time.

The I/O buffer itself has traditionally been used to cache frequently accessed
blocks, so that each access for a block does not incur a disk access. Thus, in a
parallel I/O system, the I/O buffer can be used to improve I/O performance
in two ways (a) improve I/O parallelism by allowing prefetching so that the
latency of multiple disk accesses can be hidden behind a single I/O, and (b)
 improve I/O latency by caching blocks to avoid repeated disk accesses for the
same block.

To make best use of available parallelism and locality in I/O accesses, it is
necessary to design and implement prefetching and caching algorithms that
schedule reads intelligently so that the most useful blocks are prefetched into
the buffer and the most valuable blocks are retained in the buffer when the
need for evictions arises. Thus, the performance of the overall I/O subsystem
critically depends on how effectively the I/O buffer is managed, and how effec-
tively blocks are prefetched.
A form of simple prefetching used in practice is to prefetch consecutive data blocks from a stream, with the aim of reducing the average seek time. In the parallel I/O model, by treating this larger unit of fetch as a block, the gains from reduced average access time can be combined with the performance benefits of disk parallelism. But again, fetching such additional data speculatively has drawbacks: the additional data that is prefetched could waste buffer space and disk bandwidth if the data is not required till much later. If the buffer size is fixed, increasing the block size effectively decreases the total number of blocks that the buffer can hold, and could correspondingly affect the I/O parallelism that can be obtained. For a fixed size of the I/O buffer, there is a tradeoff between the benefits of a larger block size and the achievable I/O parallelism, with the latter dominating at practical buffer sizes [18].

Additionally, in recent years there has been a move in storage technology away from centralized data repositories to loosely coupled storage networks. This trend makes it even more important to exploit coarse-grained I/O parallelism, beyond traditional techniques like data striping. In short, the demands of modern applications and current computing trends motivate us to design and evaluate new techniques for storage management which can efficiently make use of the capabilities of modern I/O architectures.

In this dissertation we focus on techniques to improve parallel I/O performance of read-intensive applications. Specifically, the emphasis is on prefetching and caching algorithms that exploit disk parallelism to reduce application I/O latency. The aim is to exploit the high I/O bandwidth provided by multiple disks, by using appropriate buffer management to reduce the average read latency seen by an application.

1.2 System model

The abstract model of the I/O system, which we use to analyze our algorithms, is based on the Parallel Disk Model [36]: the I/O system has $D$ independent disks that can be accessed in parallel, and an I/O buffer through which all disk
accesses occur *. The sequence of data block accesses made by the computation is called the reference string. In serving a reference string, the prefetching and caching algorithm, or the buffer management algorithm, decides 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 measure of performance in the model is the number of parallel I/Os that are issued.

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 consider global lookahead which provides a subsequence of future requests †.

In this dissertation we consider two categories of I/O access patterns: read-once and read-often. Read-once reference strings are characterized by accesses to distinct blocks. This form of accesses is representative of workloads in which repeated accesses to any piece of data is infrequent. More general workloads are characterized by read-often reference strings in which the same block can be accessed several times during the course of the computation.

Read-once reference strings model workloads of I/O-bound applications such as database query processing, external merging and merge-sorting, and multimedia streaming applications such as video-servers. Such reference strings also provide a mechanism to study prefetching in isolation without the associated problem of caching in read-often reference strings. Caching is not an issue in such read-once reference strings — once a block has been referenced it can be evicted since it is never referenced again. This makes read-once reference strings easy to schedule in single disk systems; all we need to do is fetch in the order of the reference string, on demand. It is interesting that even with the simplification of read-once behavior the problem of I/O scheduling is non-trivial

---

* A distributed buffer configuration, where each disk has a local buffer that is not shared among the disks was studied in [18], Varman96

† Local lookahead, which gives the next block to be accessed from any disk was studied in [18, 35]
in the multiple disk case.

In the case of read-often reference strings, traditional paging policies for buffer management have concentrated on minimizing the total number of I/O requests that need to go to disk. However with multiple disks, the total number of page faults is not a sufficient criterion due to I/O parallelism. For instance, the paging policy longest future distance (MIN) [7] has been shown to minimize the total number of page faults in a single disk system. However, as we shall show later, it does not optimize the number of (parallel) I/Os in the parallel disk case.

1.3 Problem addressed

The problem addressed here 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. The only conditions that a valid schedule should satisfy are that at any time no more than one block is fetched from any one disk, and the number of blocks in the buffer is never more than $M$. 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. Thus, 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 prefetching and caching algorithm is to generate a schedule that performs the smallest number of parallel I/Os with the available information.

Since the I/O buffer is a shared resource, it is capable of buffering blocks from any disk, thereby allowing buffer space to be allocated 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 and dynamically allocate buffer space among blocks fetched from different disks. It has to de-
cide on questions like "how much buffer to allocate for prefetching and how much for caching", "which blocks to prefetch", and "which blocks to cache".

To make use of the higher bandwidth available 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 effects of both polluting the buffer and causing unnecessary I/Os for blocks which may be evicted, as well as choking the buffer and reducing the parallelism in fetching more immediate blocks. Later, in Chapter 3 we will show that the problem even when reduced to just prefetching decisions is interesting: even in the case when each block is accessed only once, the problem of deciding what blocks to fetch in an I/O is non-intuitive. A good prefetching and buffer management algorithm ought to co-operatively decide how much buffer space to allocate for prefetching in a particular I/O and which blocks ought to be prefetched then.

1.4 Organization of this dissertation

This dissertation is organized as follows. A summary of the main contributions of this thesis is presented in Section 1.5. A discussion of related work follows in Section 1.6. We first introduce and formalize the model of the I/O system, model of lookahead, and performance metrics in Chapter 2. Chapter 5 presents a summary of our main results and avenues for further work in this area. Overall in this dissertation, we first analyze the fundamental limitations arising due to limited lookahead is, look at simple algorithms based on intuitive generalizations of sequential algorithms, and then design and analyze algorithms optimized for parallel I/O systems. A large portion of the technical matter in this dissertation involves formal proofs. To improve the readability, we briefly summarize the main idea of the proof in the paragraph which precedes the proof.

In Chapter 3 we present our results on scheduling read-once reference strings. We show fundamental bounds on the performance of online algorithms
with global lookahead in Section 3.1. We introduce and study the performance of two intuitive greedy prefetching algorithms, PHASE and GREED, in Section 3.2. In Section 3.3 we consider the performance of prefetching algorithms when the data layout is randomized. The optimal off-line prefetching algorithm for read-once scheduling, L-OPT, is presented and analyzed in Section 3.4. In Section 3.5 we compare the performance of online algorithms with the same amount of lookahead and show there there is no one algorithm which is guaranteed to always perform the minimal number of I/Os for all reference strings. Finally, in Section 3.6 we empirically evaluate L-OPT using synthetically generated reference strings.

In Chapter 4 we present our results on prefetching and caching for read-often reference strings. In Section 4.1 we present fundamental limitations on the information available through a given amount of lookahead. In Section 4.2 we examine the performance of algorithms which use MIN as the caching algorithm and show that they can potentially serialize all accesses. Again, when data blocks are distributed across all disks randomly, with uniform probability, we show in Section 4.3 that simple prefetching and caching can provide high performance. In Section 4.4 we present and analyze a deterministic optimal off-line buffer management algorithm.

1.5 Summary of results

This section summarizes the main contributions of this work. Briefly our contribution can be categorized into results for read-once reference strings and read-often reference strings. In both situations, we have first developed a model of the system incorporating the main features of the problem, performed studies to show limitations of existing techniques, and then developed and analyzed new algorithms to fully exploit the underlying I/O system.

We use $M$ to denote the size of the I/O buffer (in blocks), $D$ to denote the number of disks in the system, and $L$ to denote the length (in blocks) of the lookahead window available to the buffer management algorithm. To measure
the performance of online algorithms we use the \textit{competitive ratio}, which is the worst case ratio of the number of I/Os done by the online algorithm to the number of I/Os needed by the optimal off-line algorithm to service the same reference string.

\textbf{Read-once reference strings}

The problem of scheduling read-once reference strings highlights the issue of prefetching, without any interference from caching. We analyze generalizations of single-disk greedy algorithms and show that there can be significant loss in parallelism when they are used in a parallel I/O system. We then present an optimal algorithm, L-OPT, for read-once scheduling. In the off-line case when the entire reference string is known to the algorithm, we show that L-OPT is in fact optimal; that is, it generates the minimal length I/O schedule. To summarize, our main results on scheduling read-once reference strings are as follows.

\textit{Limits of lookahead}

We show that the competitive ratio of any online prefetching algorithms serving read-once reference strings with global $L$-block lookahead is at least

\[
\begin{align*}
\Omega(\sqrt{MD/L}) & \quad \text{for } L > M \\
\Omega(\sqrt{D}) & \quad \text{for } M \geq L \geq M/\sqrt{D} \\
\Omega(M/L) & \quad \text{for } M/\sqrt{D} \geq L > M/D \\
\Omega(D) & \quad \text{for } M/D \geq L
\end{align*}
\]

To put these results in perspective, note that for scheduling read-once reference strings, the competitive ratio of a simple sequential scheme that performs one I/O per access is $D$.

An important consequence of this result is that it quantifies the maximum potential benefit that can be obtained by having increased lookahead. Since the relation is non-linear, it shows a tradeoff between the cost of getting additional
lookahead and the additional benefit that it can provide. For instance, when $L = MD$ the competitive ratio of L-OPT is $O(1)$, and hence at most constant-factor improvements can be obtained by increasing the lookahead beyond this.

**Performance of intuitive algorithms**

We consider two intuitive greedy algorithms, PHASE and GREED which make use of global lookahead. Both are greedy prefetching algorithms that fetch from as many disks as possible in an I/O, but PHASE limits its prefetching depth by preferring earlier referenced blocks. We show that when the lookahead is shallow, $L < M$, the competitive ratio of both PHASE and GREED match the lower bounds shown before. However, PHASE cannot make use of additional lookahead beyond $M$, and the performance of GREED deteriorates to $\Theta(D)$ when the lookahead is more than $2M$.

Thus by limiting the damage due to greedy prefetching, PHASE can be effective when the lookahead is shallow. However, inherently these algorithms cannot use additional lookahead, larger than the size of memory, to improve performance.

**Optimal algorithm**

We design an online prefetching algorithm L-OPT [20] using $L$-block lookahead, $L \geq 1$. L-OPT uses the currently available lookahead to determine blocks to fetch on the next I/O. It uses a novel priority assignment scheme to determine the most valuable blocks to fetch and keep in the buffer. As the lookahead window advances and new information about requests are made available, the priorities of blocks are dynamically updated to incorporate the latest information.

When the lookahead $L$ comprises the entire reference string, L-OPT is the optimal off-line algorithm; that is, it produces the minimal length I/O schedule for the given reference string. This is the first provably optimal polynomial-time algorithm for scheduling read-once reference strings.
In the online case when the lookahead does not encompass the entire reference string we show that L-OPT has a competitive ratio within a constant factor of the best possible with the given amount of lookahead. Hence L-OPT is comparable to the best online algorithm with the same amount of lookahead.

**Comparing online algorithms**

We compare the schedules generated by online algorithms that have the same amount of lookahead. Ideally we would like to have an online algorithm that is optimal; that is, one which, for all reference strings, generates the smallest length schedule among all algorithms with the same amount of lookahead. When the length of lookahead is less than the buffer size, we show that L-OPT is the optimal online algorithm. However, we show the interesting result that *no such online optimal algorithm exists* when the lookahead exceeds the memory size. We show that the worst-case dilation of L-OPT with respect to the smallest length online schedule is 2.

**Randomized layout**

We consider the case when the data layout is randomized; that is, each block has an equal probability of being on any disk. Under such conditions, the expected number of I/Os performed by PHASE to service a reference string of length $N$ is $O(N/D)$, with a lookahead of size $\Omega(D \log D)$ and buffer of size $\Omega(D \log D)$. Note that the $N/D$ is the minimum number of I/Os that any scheduling algorithm needs to do to fetch $D$ blocks. Also randomization can be achieved by explicitly assigning blocks to disks randomly, with uniform probability. Thus, PHASE can extract full parallelism from the I/O system if the data being read in is randomized.

**Read-often reference strings**

Scheduling read-often reference strings further complicates the buffer management problem because of caching; now the buffer management algorithm needs
to choose among referenced blocks to evict from the buffer. Since evicted blocks may need to be fetched later, while making eviction decisions it is important to consider I/O parallelism in fetching the blocks later. To summarize, our main results on scheduling read-often reference strings are as follows.

Limits of lookahead

We show that the competitive ratio of any online prefetching algorithms serving read-often reference strings with global $L$-block lookahead is at least

$$\Omega(\max\{M - L, \sqrt{D}\}) \quad \text{for} \quad L \leq M$$
$$\Omega(\sqrt{MD/L}) \quad \text{for} \quad L > M$$

Note that in the case of scheduling read-often reference strings, the competitive ratio could be worse than $D$ because of bad caching decisions.

Again, this result highlights the benefit that can be gained by having additional lookahead. When the lookahead is less than the buffer size $M$, the effect of caching dominates. When additional lookahead is available, the benefit is the same as in the case of read-once scheduling.

Performance of intuitive algorithms

In the case of sequential I/O, it is known that an off-line algorithm that evicts the block which is referenced farthest in the future is optimal [7]. However such a caching strategy does not consider the load on the disks when these evicted blocks may need to be fetched. We show that any prefetching algorithm that uses MIN as the buffer replacement policy may sequentialize I/Os substantially and have a competitive ratio of $\Omega(D)$. Thus using MIN can result in ignoring all parallelism available in the reference string.

Optimal algorithm

We present a new integrated prefetching and caching algorithm SUPERVISOR. Intuitively, SUPERVISOR fetches blocks from as many disks as possible in an
I/O, while ensuring that the buffer contains the blocks with the highest priority. Each block in the lookahead is assigned a priority, which is a measure of how much the fetching of that block can be delayed. \textsc{Supervisor} is based on two intuitive ideas: (a) do not waste I/O slots if possible so that additional lookahead can be effectively utilized and (b) issue prefetches to blocks close to their references so that prefetched blocks do not wastefully occupy buffer space.

We show that \textsc{Supervisor} is the optimal off-line algorithm for servicing read-often reference strings; that is it performs the least number of I/Os when it has a-priori knowledge of the entire reference string. This is also the first polynomial time provably off-line optimal algorithm for prefetching and caching in the parallel I/O case.

\textit{Randomized layout}

As in the read-once case, we also consider the situation when the layout of data is randomized. We consider a simple algorithm \textsc{Phase-LRU}, a greedy prefetching algorithm together with Least-Recently-Used (LRU), and show that the expected number of I/Os performed by it is within a factor $O(\log D / \log \log D)$ of the optimal when the lookahead is at least the buffer size, and the buffer size $O(D \log D)$. Thus a randomized data layout not only simplifies prefetching, by making the load on disks uniform, but can also aid caching by minimizing the penalty due to erroneous eviction decisions.

\section{1.6 Related work}

Classical buffer management in the single-disk model deals with optimizing eviction decisions to minimize the number of I/Os needed to service a given reference string. The MIN algorithm [7] is the optimal off-line algorithm to handle page replacement in the single disk case. Intuitively, the MIN algorithm always replaces the page in the buffer which is to be accessed farthest in the future. Online buffer management was studied in [33], where the competitive ratio of online page replacement policies, LRU and FIFO were shown
to be $\Theta(M)$, where $M$ is the size of the I/O buffer. Extended models incorporating lookahead (and extra buffers) were studied in [3, 8, 1, 9, 16, 26, 34] in the sequential I/O case. The problem there is to determine which blocks need to be replaced in an online manner, with the measure being the competitive ratio. It has been shown that the online page replacement strategies like LRU and FIFO belong to a general class of marking algorithms which have a competitive ratio of $M$, with no lookahead. An analysis of marking algorithms with strong lookahead (similar to global lookahead defined here) was presented in [3] for the single disk situation, wherein it was shown that with strong lookahead $L$, the competitive ratio of LRU is $M - L$, when $L < M - 1$. Analysis of online paging algorithms using undirected access graphs to model locality of reference was presented in [1, 16], and it was shown that the competitive ratio of LRU is always less than that of FIFO in the presence of locality. [26] introduced a new measure of performance which attempts to capture the power of one form of information over another. A few results presented in this work use a variant of this measure, which we call the online ratio. This measure characterizes the performance of one online algorithm in relation to all other online algorithms with the same amount of lookahead. In [15] a lower bound on the performance of randomized algorithms for buffer management in the sequential I/O case was presented, which was matched to a constant factor, by randomizing the replacement decisions of the marking algorithm. A matching optimal randomized paging algorithm was presented in [27].

Recent formal work on prefetching in single-disk systems have focused on overlapping I/O with computation [10]. A stall model of sequential I/O incorporating computation time was introduced by [10]. They considered the combined problem of prefetching and caching and showed how aggressive prefetching can reduce the total elapsed time. In the same model, formulating the caching problem as an integer programming program, a polynomial time solution was presented by [4].

There has been relatively little work dealing with prefetching and buffer management in the parallel setting. There has been much work on improv-
ing the performance of external sorting using parallel I/O in the Parallel Disk Model [2, 31, 32, 29, 5]. [29] analyzed external merging using a greedy prefetching scheme to fetch from each disk. The Markov analysis from [29] can be used to show that a greedy prefetching scheme with randomized layout can get $\Theta(D)$ parallelism using a buffer of size $\Omega(D^2)$. [5] presented a external merge-sort algorithm using multiple disks. They achieved optimal I/O performance using a randomized layout which stripes data starting from a randomly chosen disk. These results are closest to our result on scheduling read-once reference strings using a randomized data layout. In contrast the work here deals with generalized reference strings.

In a generalization of the stall model to parallel disks, [23, 22] focused on the off-line problem of serving read-many reference strings. They designed a sophisticated off-line approximation algorithm called reverse aggressive to service read-many reference strings. Reverse-aggressive is shown to be near optimal for typical system parameters (memory size and computation time): within a factor $DF/M + 1$ of the optimal schedule length, where $F$ is the ratio of computation time to I/O time. [35] considered scheduling read-many reference strings 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 in this buffer configurations.

[28] addressed request distribution in a network-server where the system is organized so as to have a buffer which is partitioned among all the different back-end nodes and data is replicated. In this context they problem of distributing streams of requests so as to balance the load at the back-end data servers. Their distribution policy is based on allocating streams to servers to maximize data reuse, while changing a stream's server when the number of streams that the server is servicing increases beyond a threshold.

In a more empirical setting, the TIP model of parallel I/O [30] defines a cost/benefit model to compare the relative cost of reclaiming a buffer by eviction against the benefit obtained by prefetching a block into that buffer. This is used to guide the decisions of the buffer manager. The approximations made
in the model favor the case of shallow prefetching within a small lookahead window, and the case when the requests within the lookahead are distributed randomly across the disks. Other empirical studies on using prefetching, with information disclosed by the application, to improve system performance have been done in [11, 24, 25]. Recently, [12] studied how to automatically derive lookahead information, for use in prefetching, by speculatively executing the computation while it is actually stalled for an I/O.
Chapter 2

System model

In this chapter we shall precisely specify the model of I/O and computation used in this dissertation, the model of lookahead, and the performance metrics which are used to characterize the algorithms.

2.1 Model of I/O system and computation

The Parallel Disk Model [36] forms the basis of the abstract model of the I/O system used in this study. In this model the I/O system consists of \( D \) independent disks that can be accessed in parallel. The I/O system has a buffer through which all disk accesses occur. The I/O buffer is a shared resource, capable of buffering blocks from any disk. The architecture of the I/O system is illustrated in Figure 2.1.

The computation reads data from the disks in blocks; a block being the unit of disk access. The capacity of the I/O buffer is \( M \) blocks. The I/O trace of a computation is characterized by a reference string, which is an ordered sequence of blocks requested by the computation. For the computation to successfully access a data block from the I/O system, it should be resident in the I/O buffer. In serving a reference string the prefetching and caching algorithm decides 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.

Note that this model does not impose any restrictions on the timing of I/O requests relative to the computation process, other than requiring a block to be present in the buffer before its request is serviced. However, all of the scheduling algorithms that we define service as many requests from the buffer as possible, initiating an I/O only when the requested block is not present in the buffer.
The block which causes the I/O to be initiated is called the demand block.

In this study we shall use the number of parallel I/Os as the measure of performance. If the number of disks is \( D \), then in each parallel I/O up to \( D \) blocks (at most one from each disk) can be 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.

The ideal measure of performance of a parallel I/O system is the time it takes to service a set of I/O requests. The absolute time taken depends on the number of I/Os performed to service the requests; the time for each I/O depends upon parameters like the geometry of the disk, load on the system, disk scheduling algorithm, unit of data fetch, the layout of the data, and the interconnect between the disks and main memory. Recent advances in storage interconnection networks help achieving a much higher bandwidth between disks and main memory than traditional bus based interconnects; thereby allowing a large number of disks to be accessed simultaneously and avoid serializing access at the interconnect. 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 and hence, the best scope for improvement in the overall execution time is to reduce the number of I/Os performed. However, it must be noted that high level I/O scheduling, in the sense of optimizing the number of I/Os, and individual disk scheduling, in the sense of optimizing disk access time, are very interdependent. The actual order in which blocks are accessed from disk
influences the disk access time, which in turn could influence the higher level scheduling decision.

The parallel I/O model used in this work implicitly assumes that I/Os are issued synchronously and have the same latency. This approximation provides an intuitive yet powerful metric to design and analyze parallel I/O algorithms. It is more valid in I/O intensive applications, which typically aggregate I/Os to perform I/O for large blocks. This aggregation is done to amortize the seek time over a larger amount of data, wherein the access time can be approximated well by the transfer time. Especially in parallel I/O systems with a large number of disks, the model helps us to concentrate on formally designing suitable algorithms that achieve high parallelism, without complications introduced by variable block access times. In a more realistic application, where the I/O times are not constant, the algorithms developed here may need to be suitable tweaked to achieve higher performance. For instance, in [14] the authors used an algorithm whose ideas are based on L-OPT developed here to optimize the number of simultaneous video streams that a video-on-demand server can support. Though the I/O times for all blocks were not the same, substantial improvements were demonstrated over traditional scheduling algorithms.

As discussed before, we work with two forms of reference strings, read-once and read-often. A read-once reference string consists of accesses to unique blocks, while blocks may be repeatedly accessed in read-often reference strings. In the case of read-often reference strings, we shall call two references distinct if they accesses different blocks.

### 2.2 Model of Lookahead

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. Note that this lookahead information is in addition to any information regarding that past which the algorithm may decide to keep.

We define and work with global lookahead, an intuitive form of lookahead.
Global lookahead provides to the prefetching algorithm a subsequence of the reference string. As the computation progresses global lookahead therefore provides a rolling window of future requests. We consider both online algorithms which use bounded lookahead as well as off-line algorithms which use a-priori knowledge of the entire reference string to construct the I/O schedule.

The performance of an online scheduling algorithm clearly depends on the amount of information in the lookahead, and how frequently the lookahead is updated. The algorithms that we design in this dissertation can accommodate lookahead that is available in variable sized batches and which is available at arbitrary instances during the computation.

However, to analyze and quantify the performance of online algorithm we need a more precise definition of the extent of lookahead and when it is available. For our analysis we consider the lookahead to be available in constant sized batches, which is available as a rolling window, being updated after every reference.

**Definition 1** An algorithm is said to have global \textit{L-block lookahead} if at any time it knows a portion of the reference string, starting from the next reference and including accesses to \textit{L} distinct blocks; that is if the lookahead is \( \mathcal{L} = \langle r_i, \ldots, r_j \rangle \) then the number of references to distinct blocks in \( \mathcal{L} \) is \( L \).

This is a natural definition of lookahead and is similar to that of strong lookahead [3], which was used in the context of online caching algorithms for sequential systems. \( L \)-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.

Figure 2.2 illustrates global 4-block lookahead. When the next block to be accessed by the computation is B1, the lookahead window starts from B1 and consists of 4 distinct references B1, C1, A1, and A2.

Note that in the case of read-once reference string, global $L$-block lookahead means that the prefetching algorithm is provided with the next $L$ blocks to be referenced by the computation.

2.3 Problem specification

The problem is to generate a schedule for a given reference string with the available lookahead information. The reference string $\Sigma$ is a sequence of I/O requests, and each requested block resides on a specified disk from which it is to be fetched. The I/O schedule specifies the blocks to be fetched in each I/O and the blocks to be evicted. The conditions that a valid schedule should satisfy are 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.

2.4 Performance metrics

We consider both a worst-case model wherein each block of the reference string may be requested from any arbitrary disk and a stochastic model wherein each block is requested, independent of the others, from a randomly chosen disk. In
the worst-case model, we express the I/O performance of online algorithms in
terms of the competitive ratio and online ratio.

In the stochastic model, we express I/O performance in terms of the expected value of the total number of parallel I/O operations required as a function of $N$, the length of the read-once reference string.

### 2.4.1 Competitive ratio

The online algorithms considered in this work are analyzed in the framework of competitive analysis. We shall use the competitive ratio [21] as the measure. 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 it is a worst-case measure it attempts to isolate the effect of decisions made by the online algorithm from inherent features of the input data.

**Definition 2** An online parallel prefetching algorithm $A$ is said to have a competitive ratio of $c_A$ if for any read-once reference string $\Sigma$, the number of I/O operations, $T_A(\Sigma)$, that $A$ requires to serve $\Sigma$ is no more than $c_A T_{\text{OPT}}(\Sigma) + b$, where $b$ is a fixed constant and $T_{\text{OPT}}(\Sigma)$ is the number of I/O operations required by an optimal off-line algorithm to serve $\Sigma$.

### 2.4.2 Online ratio

The competitive ratio of an algorithm with a certain amount of lookahead is influenced by two factors: unknown information about the reference string beyond the lookahead, and the inability of the algorithm to effectively exploit information available in the lookahead. The competitive ratio does not differentiate between these two factors and hence cannot distinguish algorithms for which the contribution of the former factor dominates. A complementary measure of performance, the comparative ratio, was introduced in [26]. This measure tries to quantify the performance loss of an online algorithms due to the unknown portion of the reference string beyond the lookahead.
We also quantify the performance of an online algorithm by comparing it with the optimal length schedule that can be generated by some online algorithm that has the same amount of lookahead. This motivates the definition of online ratio. In contrast to the comparative ratio, the online ratio measures how well the online algorithm exploits the information available in the lookahead.

**Definition 3** Let $C_L$ be the set of all algorithms with $L$-block lookahead. For any algorithm $A$ in $C_L$, let $T_A(\Sigma)$ denote the number of I/Os needed by $A$ to service $\Sigma$. Let ON-OPT($\Sigma$) be the minimal length schedule for $\Sigma$, that can be generated by any algorithm in $C_L$, and let its length be $T_{ON-OPT}(\Sigma)$. An online scheduling algorithm $A \in C_L$ has an online ratio of $c_A$ if there is a fixed constant $b$ such that for any reference string $\Sigma$, $T_A(\Sigma) \leq c_A T_{ON-OPT}(\Sigma) + b$. 
Chapter 3

Scheduling read-once reference strings

In this chapter we present online and off-line prefetching algorithms for read-once reference strings. Read-once reference strings, where all accesses are to unique blocks, arise frequently in I/O-bound applications like external merging and merge-sorting (including carrying out several of these concurrently [37]) and in real-time retrieval and playback of multiple streams of multimedia data, such as compressed video and audio. Since there is no benefit in holding a block in buffer once it is referenced by the computation, read-once reference strings provide a natural scenario which highlights the problem of prefetching in a parallel I/O system without the interference of caching.

Since no block is referenced more than once, it would seem that an effective prefetching algorithm need only fetch blocks in the order of their appearance in the reference string. This would agree with the intuition that we would like to keeps blocks buffered in the buffer for only a short duration, and thus would prefer to prefetch a block required earlier over one that is referenced later. Counter to this intuition, we show that in the parallel I/O model deeper prefetching can help substantially. In fact, in certain cases the optimal off-line algorithm does not follow the policy of fetching blocks in the order of their appearance in the reference string: at times it needs to prefetch blocks that are referenced much later in the future, before blocks on some other disk that are about to be referenced in the immediate future. An important corollary is that information beyond the next memory-load of references is necessary for optimal prefetching.

To intuitively illustrate the problem of prefetching in a parallel I/O system, 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 $(a_1 a_2 a_3 a_4 b_1 c_1 a_5 b_2 c_2 a_6 b_3 c_3 a_7 b_4 c_4 c_5 c_6 c_7)$.

Figure 3.1 shows the I/O schedule constructed by a simple 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 3.2 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 schedule that fetched greedily in the order of the reference string. This schedule is of minimal length for this
reference string because there are 7 blocks requested from a single disk. This is
the schedule generated by algorithm L-OPT, the optimal read-once scheduling
algorithm, presented in this dissertation.

The above example illustrates the problem of I/O scheduling in a parallel
I/O system. Even though block $c_4$ is only requested after $b_2$, $b_3$, and $b_4$, giving
preference to it results in a shorter I/O schedule. As an interesting aside, at
the time when $a_4$ is fetched and $b_2$ evicted, $b_2$ is within the next buffer load
of blocks to be requested; but nevertheless it is still preferable to evict it and
fetch $c_4$ instead. Optimizing the total number of I/Os 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".

In this chapter we present interesting and non-intuitive results on scheduling
I/Os on a parallel I/O system. The first results are fundamental bounds on
the performance achievable by having a certain amount of lookahead. These
bounds provide the baseline which we would like the algorithms that we design
to match. Using novel techniques, we go on to show that a simple prefetching
algorithm called PHASE that fetches blocks from a disk in the order of their
appearance in the reference string can match the best competitive ratio when
the lookahead is less than $M$. However, even with this depth of lookahead,
the best competitive ratio achievable is $\Omega(\sqrt{D})$. Though PHASE matches this
bound, it cannot make use of additional lookahead to get higher performance.

However, we show that close to optimal parallelism can be obtained by
PHASE, in a special case, when the accessed blocks are distributed randomly,
with uniform probability, across all the disks. When the layout is thus ran-
domized, we show that PHASE can get an expected parallelism of $\Theta(D)$ with
a lookahead and buffer of just $\Omega(D \log D)$. Note that since there are $D$ disks in
the system, $D$ is the maximum parallelism that can be obtained.

To make use of additional lookahead beyond $M$, we present L-OPT, an optimal
algorithm for read-once parallel disk scheduling. In the off-line case, when
the entire reference string is known in advance, L-OPT generates an optimal
length I/O schedule. Moreover, applying it as an on-line algorithm, we show that L-OPT has competitive ratio within a constant factor of the best possible. As an aside, we also show that any on-line algorithm, with lookahead $L$, on some reference string will perform more I/Os than the absolute minimum possible with lookahead $L$. This indicates that there is no optimal on-line algorithm possible. Finally we present a simulation study which highlights and demonstrates the predictions made by the analysis of L-OPT.

The rest of this chapter is organized as follows. In Section 3.1, we show fundamental lower bounds on competitive ratios for algorithms using global lookahead. Intuitive parallel prefetching algorithms PHASE and GREED are described and analyzed in Section 3.2. The performance of these algorithms, when the layout is randomized, is described in Section 3.3. The algorithm L-OPT is presented in Section 3.4, followed by its analysis and empirical study in Section 3.6. In Section 3.5 we present results of comparing different online algorithms for read-once scheduling.

### 3.1 Limits of lookahead

We analyze the performance of algorithms with global $L$-block lookahead, which at any time, have knowledge of the next $L$ future accesses. The following theorem presents the main result of this section.

**Theorem 1** The competitive ratio of any on-line algorithm with global $L$-block lookahead is

\[
\begin{align*}
\Omega(\sqrt{MD/L}) & \quad \text{for } L > M \\
\Omega(\sqrt{D}) & \quad \text{for } M \geq L \geq M/\sqrt{D} \\
\Omega(M/L) & \quad \text{for } M/\sqrt{D} \geq L > M/D \\
\Omega(D) & \quad \text{for } M/D \geq L
\end{align*}
\]

These non-intuitive bounds are primarily due to the fact that in the shared buffer configuration, knowledge of the reference string beyond the next lookahead, can be used to perform more effective prefetching, even when the looka-
head is as large as \( M \) or the next memory load. Since this is a lower bound on the competitive ratio of any on-line algorithm with \( L \)-block lookahead, it quantifies the maximum potential benefit that an algorithm can derive with \( L \)-block lookahead.

We shall show the bounds in each region separately. The central idea in each case is the same: given any online parallel prefetching algorithm that employs \( L \)-block lookahead, we show that the competitive ratio is \( C \) by constructing a nemesis reference string \( \Sigma \), which forces the online algorithm to perform \( C \) times the number of I/Os incurred by the optimal off-line algorithm OPT on \( \Sigma \).

First, let us consider the case when \( L > M \). Let \( \mathcal{A} \) be an arbitrary prefetching algorithm that uses global \( L \)-block lookahead. We will use the following notation during our analysis. Let \( T_A \) be the total number of I/Os used by \( \mathcal{A} \) to service \( \Sigma \) and let \( T_{\text{OPT}} \) be the number of I/Os taken by OPT to service \( \Sigma \). We use \( D \) to denote the set of the \( D \) parallel disks.

Let us partition the reference string into phases, with the \( i \)th phase denoted by \( \text{phase}(i) \). The actual structure of each phase will be presented later. Let \( c(i, d) \) denote the number of blocks from disk \( d \) that are referenced in \( \text{phase}(i) \). Note that \( c(i, d) \) depends only upon the reference string and is independent of the scheduling algorithm.

**Definition 4** Let \( p_A(i, d) \) be the number of prefetched blocks from disk \( d \) in the buffer at the start of \( \text{phase}(i) \). Define the dominant peak in \( \text{phase}(i) \) as \( \text{dom}_A(i) = \max_d \{c(i, d) - p_A(i, d)\} \), the maximum number of blocks from \( \text{phase}(i) \) that need to be fetched from any single disk.

By the above definition, in \( \text{phase}(i) \) algorithm \( \mathcal{A} \) will need to fetch at least \( \text{dom}_A(i) \) blocks from a single disk, after beginning to service \( \text{phase}(i) \).

**Claim 1** The minimum number of I/Os that algorithm \( \mathcal{A} \) needs to make in \( \text{phase}(i) \) is given by \( \text{dom}_A(i) \).

Note that if no blocks from a phase were prefetched before beginning to service it, by Claim 1 at least \( H \) I/Os need to be performed to serve the requests.
in that phase. We will force the online algorithm into a situation where its limited lookahead prevents it from prefetching a substantial number of blocks for alternate phases, forcing it to perform $\Omega(H)$ I/Os in every such phase.

To facilitate the presentation of our lower bound proof we define balanced and loaded phases.

**Definition 5** A balanced phase $\text{balanced}(i)$ is a subsequence of the reference string of length $L$. The constituent $L$ blocks are striped in a round-robin manner across the set $D$ of all $D$ disks. A loaded phase $\text{loaded}(i)$ with hot disk $d_i$ consists of $H$ blocks laid out such that the constituent blocks all originate from the hot disk $d_i$. $H$ is an integer parameter whose value we shall set later.

Figure 3.3 illustrates the distribution of blocks on different disks in loaded and balanced phases.

We shall construct the nemesis reference string as a sequence of alternating loaded and balanced phases. By the above definition and the definition of global $L$-block lookahead, when the first block of a loaded phase is referenced, an algorithm with $L$-block lookahead knows all of the blocks (and their order of reference) in that loaded phase, and $L - H$ blocks in the next balanced phase. As the computation proceeds, the lookahead window includes additional
blocks, but while serving the references of a loaded phase, the lookahead never includes any blocks from the subsequent loaded phase.

The blocks referenced in balanced phases are striped across all $D$ disks. Hence if there are $D$ free blocks in the buffer at the start of the phase, then by reading one stripe at a time these $M$ requests can be serviced in about $M/D$ I/Os.

Given any deterministic online algorithm $A$ with a bounded $L$-block lookahead, in the following definition we show how to construct a nemesis reference string from balanced and loaded phases, depending on $A$'s prefetching decisions.

**Definition 6** Let $P$ be an integer parameter less than $D$ whose value we shall set later. We construct a reference string $\eta$ of $PH + (P - 1)L$ references such that the nemesis string $\Sigma$ can be obtained by concatenating strings constructed in the same manner, as $\eta$, an arbitrary number of times. The reference string $\eta$ is an alternating sequence of loaded and balanced phases, loaded(1), balanced(1), ..., balanced($P - 1$), loaded($P$)

Each loaded phase has a different hot disk: the first loaded phase, loaded(1), has disk 1 as the hot disk. The hot disk of every subsequent loaded phase is dependent on $A$'s prefetching decisions and is chosen as follows: For $k > 1$, let $B_k$ denote the set of hot disks corresponding to all loaded phases occurring prior to loaded($k$). Let $G_k$ denote the set $D - B_k$ of $D - k + 1$ disks not in $B_k$. It is possible that on account of $A$'s prefetching, one or more future blocks* are already in the I/O buffer at the end of loaded($k - 1$). Among the disks in $G_k$, the disk $d_k$ that has the smallest number of future blocks in algorithm $A$'s buffer at the end of loaded($k - 1$), is chosen to be the hot disk of loaded($k$). This is a valid construction as $A$ can see only $L$ blocks ahead in the reference string and so cannot make any prefetching decisions depending on loaded($k$) prior to the end of loaded($k - 1$).

*By future blocks we mean blocks that get referenced some time in the future with respect to the present point in time, when they are prefetched.
Lemma 1 The competitive ratio of any deterministic online algorithm having global L-block lookahead is at least $\Omega(\sqrt{MD/L})$ when $L > M$.

Proof: We shall show that the reference string $\Sigma$ defined above is such that $T_A / T_{OPT}$ is $\Omega(\sqrt{MD/L})$ when $L \geq M$. In Lemma 2 we show that $A$ will incur at least $PH - P[M/(D - P)]$ I/Os for every instance of the substring $\eta$. On the other hand we show in Lemma 3 that if $PH \leq M - D$, then there exists a schedule $S$ that incurs no more than $H + (P - 1)[L/D]$ I/Os for every instance of the substring $\eta$. By assigning $P = \lceil \sqrt{MD/9L} \rceil$ and $H = \lfloor (M - D)/P \rfloor$, we have $T_A = \Omega(M)$ and $T_{OPT} = O(\sqrt{MD/L})$ provided $M \geq 2D$, whence the lemma follows. When $M < 2D$, a similar construction and proof that uses $D' = D/2$ disks instead of $D$ disks can be used to give the same bounds on $T_A$ and $T_{OPT}$. □

Intuitively, the subsequence $\eta$ is constructed by alternating loaded phases with balanced phases. Loaded phases are constructed to have a large number ($H = \Theta(\sqrt{ML/D})$) of blocks requested from a single disk. Hence these loaded phases can cause a large number of I/Os if no blocks are prefetched from the hot disk. Balanced phases are designed to hide the skewed disk block distribution of loaded phases from the online algorithm, while not permitting “free” prefetching opportunities as the next loaded phase is discovered.

It may be noted that the reference string $\eta$ partitions the $D$ disks into a set $D_1$ of $P$ disks that are hot in some loaded phase, and the rest of the $D - P$ disks, into a set $D_2$. We force the online algorithm $A$ to incur about $H - M/(D - P)$ I/Os in every loaded phases of $\eta$, thus resulting in a cost of about $HP - MP/(D - P)$ I/Os for $A$. On the other hand, we show that it is possible to design an optimal off-line schedule $S$ that fetches $HP$ future blocks from $P$ disks of the set $D_1$ in the first loaded phase itself, thereby leaving an evenly balanced disk block placement for the rest of the reference string. Thus, $S$ incurs only about $H + (P - 1)[L/D]$ I/Os overall. The following lemmas formalize the above intuition.

Lemma 2 Algorithm $A$ incurs at least $T_A \geq HP - P[M/(D - P)]$ I/Os to service $\eta$. 
Proof: For $1 \leq k \leq P$, consider the $k$th loaded phase, $\text{loaded}(k)$, in $\eta$. Let the next loaded phase, $\text{loaded}(k+1)$, have $d_{k+1}$ as the hot disk.

By Definition 6, disk $d_{k+1}$ is chosen such that (a) it has not been the hot disk for any of the previous $k$ loaded phases, and (b) among the remaining $D - k$ disks, $d_{k+1}$ has the smallest number of blocks prefetched by $A$ at the end of loaded($k$). Because the buffer capacity is $M$, at most $M$ prefetched blocks can be in the buffer at the end of loaded($k$), and specifically the number of prefetched blocks from disk $d_{k+1}$ in the buffer at the end of loaded($k$) is at most \(\lceil M/(D - k) \rceil\). During the balanced phase after loaded($k$), balanced($k$), one I/O is required for each block of loaded($k+1$) that $A$ chooses to prefetch from disk $d_{k+1}$. Hence if during balanced($k$), $A$ prefetches $n_k$ blocks from disk $d_{k+1}$ for loaded($k+1$), it must perform at least $n_k$ I/Os in balanced($k$). Thus, the total number of blocks from disk $d_{k+1}$ that could have been prefetched by the start of loaded($k+1$) is no more than $\lceil M/(D - k) \rceil + n_k$. So, by Claim 1 the total number of I/Os done by $A$ in loaded($k+1$) is at least $H - \lceil M/(D - k) \rceil - n_k$. The total number of I/Os done during balanced($k$) and loaded($k+1$) combined is therefore at least $H - \lceil M/(D - k) \rceil$. The number of I/Os done by $A$ to service $\eta$ is thus

$$T_A \geq H + \sum_{1 \leq k < P} (H - \lceil M/(D - k) \rceil)$$

This quantity is at least $HP - P\lceil M/(D - P) \rceil$. $\square$

In the following lemma we show how to construct an off-line schedule that serves the same set of requests in much fewer I/Os. Essentially, during the I/Os for the first loaded phase, the off-line schedule prefetches blocks from hot disks of all future loaded phases so that no I/Os need to be done during any of the future loaded phases. This exploits the fact that balanced phases can be serviced with full parallelism (needing about $L/D$ I/Os) with just a small amount of storage (about $D$ blocks). When $HP \leq M - D$, by prefetching $HP$ blocks into memory, the schedule leaves at least $D$ blocks free; these are used to get full parallelism in the balanced phases. Hence blocks belonging to a balanced phase need only be fetched during the servicing of the phase.
Lemma 3 A schedule $S$ that incurs at most $T_S \leq H + (P - 1)[L/D]$ I/Os to service $\eta$ can be constructed, if $HP \leq M - D$.

Proof: We construct a schedule $S$ to service $\eta$ by running the following algorithm on it. As before, in loaded($k$) let $d_k$ be the hot disk, for $1 \leq k \leq P$.

- In loaded(1) of $\eta$, we prefetch as follows:
  - During the first $H$ I/Os of loaded(1), we prefetch $H$ blocks from each of the future hot disks $d_k$, $2 \leq k \leq P$, of the rest of the $P$ loaded phases.

- During each subsequent phase, we fetch blocks of that phase with full disk parallelism. Since at most $HP$ blocks have been prefetched, and $HP \leq M - D$, there are at least $D$ free blocks in the buffer to perform fully parallel I/Os.

Since $S$ has prefetched $H$ blocks from each of the disks $d_k$, for $2 \leq k \leq P$, the dominant peak in each of the $P - 1$ loaded phases following loaded(1) will be reduced to 0. Hence, $S$ will not incur any I/Os in each of these loaded phases.

As discussed previously, any balanced phase can be serviced in $[L/D]$ I/Os provided there are $D$ free buffer blocks. This is satisfied by the schedule. Hence, in any balanced phase $S$ will incur at most $[L/D]$ I/Os. Therefore, in servicing $\eta$, the total number of I/Os done by $S$ is

$$T_S = H + (P - 1)[L/D]$$

□

Lower bounds in other regions of lookahead can be got in a similar fashion, with appropriate values for $P$ and $H$. These results are proved in the following lemma.

Lemma 4 The competitive ratio of any deterministic online algorithm having global $L$-block lookahead is at least $\Omega(\sqrt{D})$ when $M \geq L > M/\sqrt{D}$, at least $\Omega(M/L)$ when $M/\sqrt{D} \geq L > M/D$, and at least $\Omega(D)$ when $L \leq M/D$. 
Proof: When \( M \geq L > M/\sqrt{D} \), the proof follows along similar lines as Lemma 1: we show that the reference string \( \Sigma \) given in Definition 6 is such that \( T_A/T_{\text{OPT}} \) is \( \Omega(\sqrt{D}) \), whenever \( M \geq L > M/\sqrt{D} \). By assigning \( P = \lceil \sqrt{D}/3 \rceil \) and \( H = \lfloor (M - D)/P \rfloor \), we have \( T_A = \Omega(M) \) and \( T_{\text{OPT}} = O(M/\sqrt{D}) \) provided \( M \geq 2D \), whence the lemma follows. When \( M < 2D \), a similar construction and proof that uses \( D' = D/2 \) disks instead of \( D \) disks can be used to give the same bounds on \( T_A \) and \( T_{\text{OPT}} \).

When \( M/\sqrt{D} \geq L > M/D \), by assigning \( H = L \) and \( P = \lceil (M - D)/H \rceil \), we have \( T_A = \Omega(M) \) and \( T_{\text{OPT}} = O(L) \) provided \( M \geq 2D \), whence the lemma follows. Again, when \( M < 2D \), a similar construction and proof that uses \( D' = D/2 \) disks instead of \( D \) disks can be used to give the same bounds on \( T_A \) and \( T_{\text{OPT}} \).

When \( L \leq M/D \), by assigning \( H = \lfloor M/D \rfloor \) and \( P = \lceil (M - D)/H \rceil \), we have \( T_A = \Omega(M) \) and \( T_{\text{OPT}} = O(M/D) \) provided \( M \geq 2D \), whence the lemma follows. Again, when \( M < 2D \), a similar construction and proof that uses \( D' = D/2 \) disks instead of \( D \) disks can be used to give the same bounds on \( T_A \) and \( T_{\text{OPT}} \).

\( \square \)

3.2 Performance of intuitive algorithms

The previous section presented fundamental bounds on the competitive ratio of algorithms with a given amount of lookahead. We would like to design a prefetching algorithm that makes best use of the available lookahead, and thus matches the lower bounds.

One simple prefetching strategy is to fetch greedily from as many disks as possible. The intuition is to maximize the parallelism that can be obtained in each I/O. In this section we look at two prefetching algorithms, PHASE and GREED, that are based on this idea. Both these algorithms fetch blocks that are in the lookahead so as to greedily maximize the parallelism in each I/O. However, their behavior is slightly different when the buffer gets filled: PHASE prefers to fetch and buffer blocks that are referenced earlier, while GREED attempts to continually maximize parallelism without performing any evictions.
like PHASE.

We shall show that when the lookahead is shallow (less than \( M \)) both these algorithms have a competitive ratio that matches the lower bounds of Section 3.1. When the lookahead is \( M \), they have a competitive ratio of \( \Theta(\sqrt{D}) \). However, neither of these algorithms can make use of additional lookahead to perform better. In fact, we show that GREED, by being too aggressive can fill up the buffer with blocks required only much later, and has a competitive ratio of \( \Theta(D) \) when the lookahead is more than \( 2M \). PHASE on the other hand forces its prefetching to be shallow and continues to have a competitive ratio of \( \Theta(\sqrt{D}) \) even when the lookahead is more than \( M \). Thus while both these greedy prefetching strategies are quite useful when the lookahead is less than the size of the I/O buffer, they cannot make use of additional lookahead to improve their performance.

Let us start by defining the algorithms PHASE and GREED. Both these algorithms evict a block from the buffer once a request for that block has been serviced.

**PHASE** builds a schedule as follows: on every parallel I/O it fetches a block from each disk that has an unread block in the current global lookahead, preferring a block which is referenced earlier. If necessary it may evict blocks in the buffer which are required later to create space for the blocks being fetched.

**GREED** builds a schedule as follows: on every parallel I/O it fetches the next block not in buffer from each disk provided there is space available in the buffer. If there are less than \( D \) free blocks when the I/O is made, then only the demand block is fetched.

To illustrate the functioning of PHASE and GREED algorithms consider the reference string \( (a_1a_2a_3a_4b_1b_2b_3b_4d_1d_2c_1c_2b_5b_6a_5a_6c_3) \). As before, the letter denotes the disk from which the block is requested and the subscript denotes the block index within the disk. Let the size of the buffer be 8, while the lookahead be the entire reference string, 16.
Figure 3.4 presents the schedule generated by PHASE for this reference string. During the I/O for \( a_1 \) and \( a_2 \) blocks from the other three disks are also fetched as the lookahead windows extends till the end of the reference string, and the buffer can accommodate 8 blocks. Since there is place for only 2 more blocks in the buffer, and \( b_3 \) is referenced before \( c_3 \), only \( b_3 \) is fetched in parallel with \( a_3 \). In the next I/O, \( b_4 \) is fetched in parallel with \( a_4 \), while evicting \( c_2 \) from the buffer as \( c_2 \) is the farthest referenced block in the buffer and \( b_4 \) is referenced before \( c_2 \). \( c_2 \) is again fetched back in the next I/O where it is the demand block, and \( c_3 \) in the last I/O in parallel with \( b_6 \). From the schedule above it can be seen that PHASE requires a total of six I/Os.

For the same reference string the schedule generated by GREED is shown in Figure 3.5. During the I/O for \( a_1 \), GREED prefetches blocks from all other disks as there are more than \( D = 4 \) free blocks. When \( a_3 \) is requested, GREED will have six prefetched blocks and hence no blocks are prefetched during the third I/O. Blocks are freed later, and when \( b_3 \) is requested there are only four prefetched blocks in the buffer; consequently, \( a_5 \) is prefetched with \( b_3 \). Thus, GREED services \( \Sigma \) in eight I/Os.
Our main results on the performance of PHASE and GREED are as follows

- The competitive ratio of PHASE is
  \[ \Theta(\sqrt{D}) \text{ for } L \geq M/\sqrt{D} \]
  \[ \Theta(M/L) \text{ for } M/\sqrt{D} \geq L > M/D \]
  \[ \Theta(D) \text{ for } M/D \geq L \]

- The competitive ratio of GREED is
  \[ \Theta(D) \text{ for } L \geq 2M \]
  \[ \Theta(\sqrt{D}) \text{ for } M \geq L \geq M/\sqrt{D} \]
  \[ \Theta(M/L) \text{ for } M/\sqrt{D} \geq L > M/D \]
  \[ \Theta(D) \text{ for } M/D \geq L \]

Thus both GREED and PHASE are the best algorithms when the lookahead is less than \( M \). The main difference between GREED and PHASE is that when the lookahead is more than \( M \), PHASE limits the prefetching depth to \( M \). It turns out that this is a very useful thing to do. By not limiting the prefetching depth, GREED could fill the buffer with blocks which would only be required much later in the future; this can cause substantial loss in I/O parallelism for intermediate references. GREED has a competitive ratio of \( \Theta(D) \) when \( L \geq 2M \).

### 3.2.1 Performance of PHASE

Theorem 1 presented bounds on the competitive ratio of any online prefetching algorithm using global \( L \)-block lookahead. In this section we derive a matching upper bound on the competitive ratio of PHASE whenever the lookahead is less than \( M \); that is, the competitive ratio of PHASE is

\[ \Theta(\sqrt{D}) \text{ for } L \geq M/\sqrt{D} \]
\[ \Theta(M/L) \text{ for } M/\sqrt{D} \geq L > M/D \]
\[ \Theta(D) \text{ for } M/D \geq L \]
First, the following claim ensures that while considering I/O schedules for read-once reference strings, it suffices to consider schedules in which a prefetched blocks is never evicted before it is referenced. The claim follows because we can cancel the prefetches for blocks that are evicted before the block is referenced with no increase in the number of parallel I/Os. Additionally, we can also assume that two blocks from the same disk are always fetched in the same order as they appear in the reference string. This can easily be guaranteed by interchanging the two I/Os in case they are fetched out of order; this will not increase the number of blocks present in the buffer at any time since the block occurring later will need to stay in the buffer at least till the block occurring first is consumed. We shall assume these properties for all schedules for read-once reference strings considered henceforth.

**Claim 2** Any I/O schedule can be transformed into another of same or smaller length such that the following properties hold (i) no block is evicted from the buffer till it is referenced (ii) if \( b_i \) and \( b_j \), \( i < j \), are two blocks fetched from disk \( d \) in the \( k \)th and \( l \)th I/O, respectively, then \( k < l \).

When the lookahead \( L \leq M \), PHASE prefetches everything in the lookahead. As \( L \leq M \), it can do so while still guaranteeing that the number of blocks in the buffer is always less than \( M \). Before we determine the competitive ratio of PHASE, let us first look at the number of I/Os done by PHASE when \( L \leq M \) in relation to other online algorithms with the same amount of lookahead. We show that among all algorithms with \( L \)-block lookahead, PHASE performs the minimal number of I/Os for all reference strings when the lookahead \( L \leq M \).

The bound follows by noting that in this range of lookahead PHASE fetches blocks as soon they are visible in the lookahead. It thus does not miss any prefetching opportunities. Moreover, in this range of lookahead it does not need to consider the problem of deciding which blocks to fetch among different blocks in lookahead.

**Theorem 2** When \( L \leq M \), the number of I/Os done by any on-line algorithm
with $L$-block lookahead is lower bounded by the number of I/Os done by PHASE to service the same reference string.

**Proof:** Without loss in generality we assume that the optimal online schedule, ON-OPT, satisfies the properties of Claim 2. The lemma can then be shown by proving that in the optimal schedule no block can be fetched earlier than the corresponding I/O in the schedule generated by PHASE.

The proof is by induction on I/Os in the schedule ON-OPT. For the base case, consider any block $r$ fetched from disk $d$ in the first I/O in ON-OPT. By the definition of global $L$-block lookahead, $r$ must be among the first $L$ blocks in the lookahead. Additionally, by Claim 2, $r$ must be the first block referenced from disk $d$. Hence by the definition of PHASE, $r$ will have been fetched in the first I/O of PHASE. Now consider any block $r$ fetched from disk $d$ in the first I/O by PHASE. We shall argue that the I/O for $r$ in ON-OPT can be advanced to the first I/O. By the definition of PHASE, $r$ is the first block referenced from disk $d$ in the referenced string, and $r$ is among the first $L$ references. Let $r$ be fetched in the $i$th I/O in ON-OPT, since the lookahead $L \leq M$, the number of blocks in the buffer at any time before or including the $i$th I/O is at most $L - 1 < M$. Also by Claim 2, no block is fetched from disk $d$ in ON-OPT's first I/O. These two conditions – availability of buffer space and a free I/O slot – guarantee that we can move the I/O for $r$ up to the first I/O.

For the induction hypothesis let us assume that in ON-OPT the first $k$ I/Os can be made to match that of PHASE and still be a valid schedule.

Consider a block $r$ fetched from disk $d$ in the $(k+1)$st I/O in ON-OPT. Let the last block referenced after the first $k$ I/Os be $s$. Note that since ON-OPT's and PHASE's schedules match for the first $k$ I/Os, this is the last block referenced in both cases. Now arguing exactly as the in the base case, we can show that $r$ (being in the lookahead and the earliest block from disk $d$ not in the buffer) will be fetched in the $(k+1)$th I/O by PHASE. Also by a similar reasoning, we can show that if a block $r$ is fetched in the $(k+1)$th I/O in PHASE, we can move the I/O for $r$ up to the $(k+1)$th I/O in ON-OPT and still have a valid schedule.

Thus ON-OPT can be transformed to match PHASE in each I/O. Hence
PHASE has to generate the minimal length I/O schedule that can be generated by any online algorithm with  $L$-block lookahead, when $L \leq M$.  

However when the lookahead $L > M$, PHASE cannot make use of the additional lookahead, as it always prefers blocks which are referenced earlier. This is the reason why its performance does not improve when additional lookahead, beyond the next memory load is available to it.

Now we can present the competitive ratio of PHASE. Let OPT denote the optimal off-line schedule to service a reference string $\Sigma$. Intuitively, OPT benefits by prefetching blocks for several lookaheads simultaneously. So, the ratio is highest when OPT performs the fewest number of I/Os to get a certain amount of benefit over PHASE. The idea of the proof is to show that OPT cannot perform too few I/Os in gaining this benefit, since there is an inverse relation between the number of I/Os spent in prefetching and the number of references across which the benefit is spread. Since OPT should do at least $x/D$ I/Os per $x$ references, this indicates that minimizing the number of I/Os done by OPT requires balancing these two factors.

**Theorem 3** The competitive ratio of PHASE is

$$\Theta(\sqrt{D}) \quad \text{for} \quad L \geq M/\sqrt{D}$$

$$\Theta(M/L) \quad \text{for} \quad M/\sqrt{D} \geq L > M/D$$

$$\Theta(D) \quad \text{for} \quad M/D \geq L$$

**Proof**: For ease of presentation we assume that $M$ is a multiple of $L$ when $L \leq M$. When $L \leq M$ it may be noted that PHASE behaves like a natural greedy algorithm: on every I/O fetch from each disk the next block not in the buffer.

For PHASE it can be seen that if a requested block is not present in the buffer it is fetched in the very next I/O. Hence for a reference string of length $N$, the maximum number of I/Os done by PHASE is $N$. On the other hand the
minimum number of I/Os required to service a reference string of length \( N \) is \( N/D \). This gives the first bound on the competitive ratio of PHASE: \( O(D) \).

Stronger bounds can be shown when more lookahead is available. Let the reference string be \( \Sigma = \langle r_0, \cdots \rangle \). By Lemma 2, PHASE is the optimal online algorithm when \( L \leq M \). Hence, if the minimal number of I/Os needed to service the reference string \( \Sigma_i = \langle r_{iL}, \cdots, r_{(i+1)L-1} \rangle \) is \( T_i \), then the number of I/Os done by PHASE to service \( \Sigma \) is less than \( \sum_i T_i \), where the sum is taken over all non-overlapping \( L \) block subsequences of \( \Sigma \). As PHASE generates the optimal length I/O schedule for any sequence of \( L < M \) blocks, we shall use this to bound the benefit that an off-line algorithm can get by prefetching from multiple lookahead windows simultaneously.

Consider the optimal off-line scheduling algorithm OPT. First let us consider the schedule generated by OPT to service \( \Sigma \). Let us partition this schedule into sub-schedules. 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 \( \Sigma_i \) is fetched; let \( S_0 \) start with the first I/O. In effect, we have partitioned the schedule generated by OPT into sub-schedules, each servicing one lookahead-full of blocks.

Also let us consider subsequences of length \( M \). Let the number of I/Os done by OPT to service the string \( \Delta_j = \langle b_{jM}, \cdots, b_{(j+1)M-1} \rangle \) be \( I_j \). Let us consider an alternative partitioning of OPT's schedule into sub-schedules, \( R_j \) which starts with the I/O following the last I/O of sub-schedule \( R_{j-1} \) and end with the last I/O in which a block from \( \Delta_j \) is fetched; let \( R_0 \) start with the first I/O. This is thus a partition of OPT's schedule into sub-schedules, each of which is fetches blocks for a \( M \) block sub-sequence of the reference string.

First consider the case when \( L \leq M \). Let us consider \( \Sigma_i \), a subsequence of \( \Delta_j \). OPT's advantage over PHASE is its additional lookahead. Hence we shall characterize the benefit in terms of the number of blocks that it prefetches more than \( L \) blocks in advance. There are two kinds of such blocks that OPT prefetches for \( \Sigma_i \) (a) those that OPT prefetches in \( \Delta_j \) prior to \( \Sigma_i \) and (b) those that OPT prefetches prior to \( \Delta_j \) for \( \Sigma_i \). We will account for these two classes of
prefetched blocks separately.

Let $P_i$ denote the blocks from $\Sigma_i$ prefetched by OPT during I/Os of schedule $R_j$ but not $S_i$. $P_i$ denotes the blocks of $\Sigma_i$ that OPT prefetches before it starts fetching any block from $\Sigma_i$ on demand. Let the maximum number of blocks in $P_i$ which are from the same disk be $\alpha_i$. $\alpha_i$ represents the savings that OPT gets by prefetching blocks from $\Sigma_i$ while fetching blocks from $\Delta_j$.

Now $I_j$ is the minimal number of I/Os needed to service $\Delta_j$, and $T_i$ is the optimal number of I/Os to service $\Sigma_i$. Hence the only advantage that OPT could have by prefetching blocks in advance is captured by the blocks in $P_i$. Hence,

$$\sum_i T_i \leq \sum_j I_j + \sum_i \alpha_i$$

During I/Os of sub-schedule $R_j$ OPT fetches $\sum_{i : \Sigma_i \in \Delta_j} \alpha_i$ blocks. However there are at most $M/L$ $L$ block sequences in $\Delta_j$. Hence, by the pigeon-hole principle, there is some $P_k$ such that $\alpha_k \geq L/M \sum_{i : \Sigma_i \in \Delta_j} \alpha_i$. That is, while performing I/Os in $R_j$, OPT prefetches at least these many blocks for the same future lookahead window. As these blocks need to be fetched from the same disk, the number of I/Os in $R_j$ is at least $\max_i \alpha_i$. Hence $T_{OPT} \geq M/L \times \sum_i \alpha_i$.

The third bound is due to the blocks that OPT prefetches for $\Sigma_i$ in advance. Let $Q_{j,k}$ denote the blocks from $\Delta_j$ prefetched by OPT in the sub-schedule $R_k$. $Q_{j,k}$ denotes the blocks from $\Delta_j$ that OPT prefetches while fetching blocks from $\Delta_k$ on demand. Because the size of the buffer is $M$, $|Q_{j,k}| \leq M$. Let the maximum number of blocks in $Q_{j,k}$ which are from the same disk be $\beta_{j,k}$; let $\gamma_j = \sum_k \beta_{j,k}$. $\gamma_j$ represents the savings that OPT gets by prefetching blocks from $\Delta_j$ in advance. Let the number of I/Os done by OPT in the sub-schedule $R_k$ be $T_k$. Since all the $\beta_{j,k}$ blocks are fetched from the same disk, it immediately follows that $\beta_{j,k} \leq T_k$.

Now $I_j$ is the optimal number of I/Os needed to service $\Delta_j$. Hence the only savings in the number of I/Os done by OPT is due to the blocks in $Q_{j,k}$s.

$$\sum_j I_j \leq T_{OPT} + \sum_j \gamma_j$$
That is

\[ T_{\text{PHASE}} \leq T_{\text{OPT}} + \sum_i \alpha_i + \sum_j \gamma_j \]

The blocks in \( Q_{j,k} \) are present in the buffer while at least \((j - k - 1)M\) other requests between \( \tau_{(k+1)M} \) and \( \tau_{jM-1} \) are serviced. Because the buffer is of size \( M \), for each set of \( M \) such intermediate blocks OPT must do at least \( |Q_{j,k}| / D \) I/Os; these I/Os are attributable to \( Q_{j,k} \). Hence the number of I/Os done by OPT is at least \( T_{\text{OPT}} \geq \sum_j \sum_k (j - k - 1) \times \lceil |Q_{j,k}| / D \rceil \). Using the bounds \( \gamma_j = \sum_k \beta_{j,k}, \beta_{j,k} \leq T_j \) and \( |Q_{j,k}| \geq \beta_{j,k} \)

\[ T_{\text{OPT}} \geq \sum_j \frac{\gamma_j^2}{D T_j} - O((M + \gamma_j) / D) \]

The above bound, together with the simple bounds \( T_{\text{OPT}} \geq \sum_j T_j, T_{\text{OPT}} \geq \sum_j M / D \) and \( T_{\text{OPT}} \geq \sum_j \gamma_j / D \), implies that

\[ T_{\text{OPT}} = O(\sum_j \gamma_j \sqrt{D}) \]

We have shown earlier that \( T_{\text{OPT}} \geq M / L \times \sum_i \alpha_i \).

Thus we have that

\[ T_{\text{PHASE}} \leq T_{\text{OPT}} + \sum_i \alpha_i + \sum_j \gamma_j \]

while

\[ T_{\text{OPT}} = O(\sum_j \gamma_j \sqrt{D} + M / L \times \sum_i \alpha_i) \]

This gives the other two bounds on the competitive ratio.

When \( L > M \) each \( \Delta_i \) is a part of one lookahead. Hence we have \( T_{\text{PHASE}} \leq T_{\text{OPT}} + \sum_j \gamma_j \), thereby bounding the competitive ratio to \( O(\sqrt{D}) \).

\[ \Box \]

3.2.2 Performance of GREED

In this section, we consider the performance of GREED. When the lookahead is less than \( M \), GREED and PHASE have identical behavior: in every I/O both
fetch blocks greedily from as many disks as possible. Thus the competitive ratio of GREED is the same as that of PHASE when the lookahead is less than the buffer size.

From the analysis of PHASE, it can be seen that PHASE cannot make use of additional lookahead beyond $M$. This is primarily because it prefers to fetch and retain blocks which are referenced earlier. On the other hand GREED could fetch blocks beyond $M$ references in the future. The question that arises is whether this is useful or harmful.

In Lemma 5 below, we show that for $L > 2M$, GREED can perform $\Omega(D)$ times as many I/Os as the optimal off-line algorithm. Note that this is the worst possible competitive ratio for any algorithm that performs I/Os only on demand (that is, when the referenced block is not present in the buffer), as it then performs at most one I/O per block in the reference string. Hence, if the length of the reference string is $N$, the maximum number of I/Os that the algorithm can do is $N$; while the least number of I/Os that the optimal algorithm could do is $N/D$ (fetching $D$ blocks in each parallel I/O). Therefore, trivially the competitive ratio of GREED is $\Theta(D)$ for $L > 2M$. It is interesting that the competitive ratio of GREED actually degrades with additional lookahead. It turns out that the fact that GREED issues I/Os indiscriminately can be used by an adversary to cause GREED to clog up its I/O buffer and lose substantial I/O parallelism.

**Lemma 5** GREED has a competitive ratio of at least $\Omega(D)$ when $L > 2M$.

**Proof:** The proof is similar to that of Theorem 1; that is, we construct a reference string that can trick GREED into performing a large number of I/Os.

We shall prove the theorem by constructing a request sequence $\eta = \langle r_i \rangle$, which requires GREED to make $\Omega(M)$ I/Os. We shall also give a schedule that serves $\eta$ with $\Theta(M/D)$ I/Os. A reference string of arbitrary length can be obtained by concatenating strings constructed in exactly the same manner as $\eta$. The length of $\eta$ will be $2M$. 
Let $H = \lfloor 2M/D \rfloor$. The reference string $\eta$ is made of $D$ sequences of length $H$ each, such that the blocks in the $i$th sequence $\langle \tau_{(i-1)H}, \ldots, \tau_{iH-1} \rangle$ are all referenced from disk $i$. The string $\eta$ is illustrated in Figure 3.6.

During the first $H$ I/Os it is possible for an off-line algorithm to prefetch the first $HD/2$ blocks of $\eta$ as they are all lie on different disks, and $HD/2 = M$. Similarly the next set of $M$ blocks can also be fetched in $H$ I/Os. Thus we can construct a schedule that can service all references in $\eta$ in $2H$ I/Os.

Now consider the number of I/Os done by GREED to service $\eta$. After the first $H$ references are fetched, $M/(D-1) - 1$ blocks from the other $D-1$ disks are fetched into the buffer. The number of I/Os done by GREED before servicing $\tau_{H-1}$ is thus $H$. Since only $M/(D-1) - 1$ blocks have been fetched from the second set of $H$ blocks, another $H - M/(D - 1) - 1$ I/Os are done by GREED. Continuing, the number of I/Os done by GREED to fetch the $i$th set of $H$ blocks is $H - M/(D - i - 1) - 1$ I/Os. Hence the total number of I/Os done by GREED is

$$T_{\text{GREED}} \geq H + \sum_{i=2}^{D} \frac{H}{D-i} - 1 \geq HD - O(H \ln D)$$

This gives the result that the competitive ratio of GREED is $\Omega(D)$.
3.3 Exploiting a randomized layout

In this section we consider the parallel prefetching problem in probabilistic settings. In previous sections we considered serving arbitrary worst-case reference strings on parallel disk systems. A natural question that arises is one regarding the performance of parallel prefetching algorithms when the blocks in the reference strings originate from randomly chosen disks, or equivalently when the reference string is generated by a stochastic adversary. In this section we present results that indicate improved performance for the parallel prefetching algorithms in this setting, compared to the worst-case settings considered earlier.

Previously, parallel I/O prefetching has been studied in a probabilistic setting, in the context of external merging [5, 29]. In external merging, the problem is to merge a set of sorted sequences, also called runs, into one globally sorted sequence. Theorem 4 presents the results for parallel prefetching in the shared buffer configuration that may be proved using results from [5, 29].

In [5], the authors studied external merging when the individual runs are striped across all the disks, with each stripe starting from a randomly chosen disk. In this case they presented a method to implement global $M$-block lookahead and used it, in an algorithm very similar to PHASE, to perform prefetching. They showed that when the buffer is of size $\Omega(D \log D)$, the algorithm performs only $\Theta(N/D)$ I/Os to service a reference string of length $N$. To do the analysis, the authors posed the problem as a variant of the classical urn occupancy problem [17], which they called the dependent occupancy problem, since there is a strong dependency between the disks on which two consecutive blocks are located. Their results are also applicable to the case when the blocks can be assumed to originate on a disk chosen independently and with uniform probability (instead of being striped, starting from a randomly chosen disk), and lead to the performance bounds on PHASE against a stochastic adversary.

In [29] the authors considered a data layout where each run is located on a single disk. Using a greedy prefetching algorithm, they showed that $N$ blocks
can be fetched in $\Theta(N/D)$ I/Os if the buffer is of size $\Omega(D^2)$. This result directly translates to performance bounds on GREED, when each block originates from a disk chosen independently and with uniform probability.

**Theorem 4** To service stochastically generated read-once reference strings of length $N$, PHASE incurs the minimum expected number of I/Os, namely $\Theta(N/D)$, using a buffer and lookahead of size $\Omega(D \log D)$; while GREED needs a buffer and lookahead of size $\Omega(D^2)$ to attain that I/O bound.

### 3.4 Optimal read-once parallel disk scheduling

In this section we present an on-line prefetching algorithm L-OPT using $L$-block lookahead, $L \geq 1$. L-OPT uses the currently available lookahead to determine blocks to fetch on the next I/O. It uses a priority assignment scheme to determine the most valuable blocks to fetch and keep in the buffer. As the lookahead window advances and further information about requests are made available, the priorities of blocks are dynamically updated to incorporate the latest information. We show that for any lookahead $L \geq M$, L-OPT has a competitive ratio of $\Theta(\sqrt{MD/L})$. Since we have shown in Lemma 1 that the competitive ratio of any on-line algorithm with global $L$-block lookahead is $\Omega(\sqrt{MD/L})$, L-OPT has performance comparable to the best on-line algorithm with $L$-block lookahead. An additional consequence of this result is that it quantifies the maximum potential benefit that can be obtained by having increased lookahead. For instance when $L = MD$ the competitive ratio of L-OPT is $O(1)$, and hence at most constant-factor performance improvements can be obtained by increasing the lookahead beyond this. We also analyze the case when $L < M$. In this case L-OPT becomes a simple greedy prefetching algorithm. We bound its competitive ratio and show that it matches the lower bound up to constants in this range as well.

Another important feature of L-OPT is its performance as an off-line algorithm. When the lookahead $L$ comprises the entire reference string, L-OPT is the optimal off-line algorithm; that is, it produces the minimal length I/O
schedule for the given reference string. This is the first provably optimal polynomial-time algorithm for scheduling read-once reference strings in the parallel disk model. The previous best algorithm had a worst-case approximation ratio of $\Theta(D^{1/2})$ [19].

The rest of this section is organized as follows. Section 3.4.1 presents the details of algorithm L-OPT. Theorem 6 establishes the competitive ratio of L-OPT in the entire range of lookahead. A corollary of Theorem 5 shows that L-OPT is the optimal off-line algorithm. Section 3.6 presents a simulation study of L-OPT on synthetically generated reference strings.

### 3.4.1 Algorithm L-OPT

In this section we present the details of algorithm L-OPT. L-OPT is a priority-controlled greedy algorithm that uses $L$-block lookahead. In every I/O it fetches one block each from as many disks as possible, while ensuring that the buffer always contains the blocks with the currently highest priorities. L-OPT assigns a priority to each block in the lookahead using a novel priority assignment scheme. The priorities are used in deciding which blocks to prefetch in an I/O.

At any instant the priority of a block is a reflection of how urgently that block must be fetched. The lower the priority of a block, the later it can be fetched. As the reference string is serviced, additional blocks are revealed and L-OPT dynamically updates the priority of the blocks in the lookahead. Consequently some blocks which were prefetched earlier when their relative priorities were high, could now be less important and may need to be evicted to make space for higher priority blocks.

Next we shall describe the algorithm. The following definitions will be used to determine the appropriate blocks to fetch from disks and those that need to be evicted from the buffer to accommodate the fetched blocks. We shall use the term "current lookahead" to mean the longest length substring of the reference string starting from the next block to be accessed such that every block is in
the lookahead.

**Definition 7**

- Let $\Sigma = \langle b_0, b_1, \ldots \rangle$ denote the reference string. If $b_i$ is a block requested in the current lookahead, let $\text{disk}(b_i)$ denote the disk from which it needs to be fetched and let $\text{priority}(b_i)$ be the block's priority.

- Let the reference string be partitioned into *windows*, which are subsequences of the reference string. Let the $s$th window be denoted by $W_s$ and consist of the references $\langle b_i, \ldots, b_j \rangle$, such that $W_s$ is the lookahead when $b_i$ is referenced, and $b_{i-1}$ is the last reference of the previous window. The first window starts with $b_0$.

- At the instance when $b_i$ is referenced let $B_i$ denote the set of blocks which are in the lookahead and present in the buffer.

- When $b_i$ is referenced let $H_i$ be the maximal set of (up to) $D$ blocks, such that if $b \in H_i$ then priority of $b$ is the largest among all blocks from $\text{disk}(b)$ in the lookahead but not present in the buffer, with blocks from an earlier window chosen over those from a later window.

- Let $B_i^{\dagger}$ be the maximal set of (up to) $M$ blocks with the highest priority in $H_i \cup B_i$; in the case of ties the block occurring earlier in $\Sigma$ is chosen.

Algorithm L-OPT is detailed in Figure 3.7. The priority function that we define will ensure that for any two blocks belonging to the same window and from the same disk, the block referenced earlier has the higher priority. In one I/O at most one block is fetched from each disk. As we would like to fetch from each disk the highest priority block, $H_i$ is the set of blocks that we would like to fetch in the next I/O. However, because of the current contents of the buffer we may not be able to fetch and accommodate all the blocks in $H_i$. At this time we use the priorities of blocks to decide which blocks to be fetch. Among the blocks present in the buffer and those that can be fetched in the next I/O we would like the buffer to contain the blocks with highest priority: $B_i^{\dagger}$ is the
Algorithm L-OPT

On a request for a block \( b_i \), from window \( W_s \), algorithm L-OPT takes the following actions. Initially priorities are assigned to the blocks in the first window.

If \( b_i \) is present in the buffer then no I/O is necessary.
If \( b_i \) is not present in the buffer then
   - To accommodate the blocks to be read in, evict the blocks in \( B_i - B_i^+ \).
   - Initiate an I/O to fetch the blocks in \( H_i \cap B_i^+ \).
Service the request for block \( b_i \).
Update the priorities of blocks in \( W_{s+1} \) using any additional blocks revealed.

Figure 3.7: Algorithm L-OPT

desired state of the buffer following the I/O. Algorithm L-OPT performs an I/O and evicts appropriate blocks to guarantee that this is the case. Additionally, whenever a new lookahead window is revealed to L-OPT, it assigns priorities to the new blocks and also updates priorities of previous blocks of the future window to correspond to the modified lookahead. Note that priorities of blocks in the window that is currently being served are not changed during the course of the computation.

The priority assignment routine, given in Figure 3.8, is used to determine the priorities of blocks in a given piece (window) of the reference string. The central idea behind the priority assignment scheme is to set the priority of a block as low as possible. However since the priorities of blocks directly influence the order in which blocks are fetched certain conditions need to be satisfied. It can be seen that fetching blocks from a disk out of order of the reference string cannot decrease the number of I/Os needed. Hence the priority assignment scheme guarantees that blocks from the same disk are assigned priori-
Routine to assign priorities

The following routine is used to assign priorities to all references in the sequence \( r_1, r_2, \ldots, r_n \).

Initialize lowestPriority to 1, all other counts to 0, and sets to \( \phi \).

for \( i \) from \( n \) down to 1

if (lowestPriority > lowestPriorityOnDisk[disk(\( r_i \))] then
    assign lowestPriorityOnDisk[disk(\( r_i \))] \( \leftarrow \) lowestPriority
assign priority(\( r_i \)) \( \leftarrow \) lowestPriorityOnDisk[disk(\( r_i \))]
increment lowestPriorityOnDisk[disk(\( r_i \))]
increment blocksWithPriority[priority(\( r_i \))]
increment numberOfBlocksPlaced
if (numberOfBlocksPlaced = \( M \)) then
    decrement numberOfBlocksPlaced by
    blocksWithPriority[lowestPriority]
    increment lowestPriority

Figure 3.8: Routine to update priorities

ties in order. Secondly if there are no additional blocks added to the lookahead, blocks are fetched according to the current priorities. Hence it is not permissible for any block to be assigned a priority such that more than \( M - 1 \) blocks referenced after it have higher priority. Our priority assignment scheme maintains the following two invariants among blocks in the same window:

1. No two blocks on the same disk have the same priority, and the priority of a block is always higher than other blocks from the same disk referenced after it.
2. For any block \( b \) in the lookahead with priority \( p \) there are less than \( M \) blocks occurring after \( b \) in the reference string with priority at least \( p \).

A simple forecasting data structure such as one described in [5] can be used to maintain the list of blocks with highest priority on each disk, in both the current window being serviced and the future window. On a hit in the buffer, algorithm L-OPT does not need to do any book-keeping. When requested block is not present in the buffer the algorithm needs to find the set of blocks to fetch and the corresponding set of blocks to evict (from the buffer). If we have all the blocks in the buffer sorted in order of their priorities, then we can decide the blocks to fetch and evict in \( O(M + D) \) time. This is because there are at most \( D \) blocks that can be fetched in any I/O and we just need to find their position within the sorted list of blocks in the buffer. The ordered list of blocks in the buffer needs to be updated each time the additional lookahead is available because the priorities of even blocks that have been fetched can change. From the specification of the routine in Figure 3.8, it can be seen that the complexity of the priority assignment scheme itself is \( O(L) \), as only a constant number of operations are performed per block in the reference string. Note that at any time there are only two priority structures that are maintained by L-OPT; the first static one corresponds to the priorities of blocks in the window being serviced, and the other dynamic one corresponds to priorities of blocks in the next window, which is updated as additional blocks are revealed during the computation.

### 3.4.2 Analysis of L-OPT

In this section we shall analyze the performance of L-OPT both as an off-line as well as an online algorithm. First, we start off by showing that L-OPT does in fact generate a valid schedule for a reference string. We prove this by arguing that L-OPT will always always assign a high enough priority for the next block that the computation will access; this will guarantee that that block will definitely be fetched in the next I/O, and prevent deadlocks.
Our main result regarding L-OPT is that L-OPT is an optimal off-line read-once scheduling algorithm; that is if L-OPT is given apriori knowledge of the entire reference string, then it will generate the smallest length schedule possible. This is the first optimal off-line prefetching algorithm for the parallel I/O situation. Though this off-line situation may not arise often in practice, it provides a very useful benchmark for other online algorithms, as well as guiding data layout strategies.

In the online situation we show that L-OPT has the best competitive ratio possible with a given amount of lookahead; its competitive ratio is

\[ \Theta(\sqrt{MD/L}) \quad \text{for} \quad L > M \]
\[ \Theta(\sqrt{D}) \quad \text{for} \quad M \geq L \geq M/\sqrt{D} \]
\[ \Theta(M/L) \quad \text{for} \quad M/\sqrt{D} \geq L > M/D \]
\[ \Theta(D) \quad \text{for} \quad M/D \geq L \]

It thus makes best use of the available lookahead information. Again this is the first online algorithm whose competitive ratio matches the lower bound for all ranges of lookahead.

**L-OPT generates a valid schedule**

We first show that L-OPT generates a valid schedule to service a reference string. We do this by arguing that if the computation requests a demand block, then that block will definitely be fetched in the next I/O. Since L-OPT fetches the blocks with the highest priority, we shall argue that the demand block has an appropriately high priority. First, any block in the current window is chosen to be fetched over blocks from the next window. Hence in this argument we shall implicitly only compare the demand block to blocks from the current window. The following definition gives an indication of how many blocks referenced after a given block in the reference string can have a priority equal to or higher than it.

**Definition 8** Let \( \text{backlog}(b_i) = |\{ b_j : j \geq i \text{ and priority}(b_j) \geq \text{priority}(b_i) \}| \) be the
number of blocks in the same window as \( b_i \), with priority at least that of \( b_i \) but referenced no earlier than \( b_i \).

From Figure 3.8 showing the priority assignment scheme, it can be seen that each block in a window from a disk is assigned a unique priority based on the value of lowestPriorityOnDisk[\( d \)]. The value of lowestPriorityOnDisk[\( d \)] increases after any block is placed on disk \( d \) and may further increase the value of lowestPriority if it falls behind this value. Since blocks are inspected starting from \( r_n \) down to \( r_1 \), the priorities of earlier referenced blocks in the same window are higher.

**Property 1** If \( r_i \) and \( r_j \) are two blocks from the same disk and the same window then \( \text{priority}(r_i) > \text{priority}(r_j) \) if and only if \( i < j \); hence, all blocks from the same window on a disk are assigned unique priorities.

The next property relates the priority of a block to the number of blocks from the same window that occur after it in the reference string, but which have a higher priority. This is important because priorities are hints to the prefetching engine as to which blocks to fetch and retain in the buffer. If a reference \( r \) has a priority \( p \) and if more than \( M \) blocks occurring after that block have a higher priority, then the prefetching engine cannot fetch just based on priorities and still guarantee that \( r \) will be fetched into the buffer when the computation is waiting for it. Also note that blocks from a future window are not preferred over blocks from the current lookahead. This property of the priority assignment routine guarantees that such a situation will not arise.

From Figure 3.8 showing the priority assignment routine, every reference is assigned a priority lowestPriority or more. Now the variable numberOfBlocksPlaced keeps track of the number of blocks with a priority lowestPriority or higher. The value of numberOfBlocksPlaced is limited to \( M \), and we assign priorities starting from the last reference. Thus, when a reference \( r \) is assigned a priority \( p \), the number of blocks from the same window that occur after \( r \) in the reference string but which have a priority \( p \) or higher is at most \( M - 1 \).
Property 2 If the priority of a reference \( r \) is \( p \), then there are at most \( M - 1 \) blocks from the same window as \( r \) that are referenced after \( r \) but which have a priority higher than \( p - 1 \).

Using the two properties from above, we will next show that L-OPT generates a valid schedule to service a reference string. The proof argues that priorities are such that the demand block will always be fetched into the buffer in the next I/O.

Lemma 6 Given any reference string L-OPT generates an I/O schedule to service it.

Proof: We shall show that whenever a requested block is not present in the buffer, it is fetched in the next I/O. This guarantees that L-OPT generates an I/O schedule which services the reference string.

Let the requested block be \( r_i \). First note that \( r_i \) will be chosen to be fetched over any block from the next window. As \( r_i \) is the earliest required block from disk(\( r_i \)), by Property 1 it has the highest priority among all blocks from that disk in that window; hence \( r_i \in H_i \). By Property 2 \( \text{backlog}(r_i) \leq M \). Because \( r_i \) is the first block in the lookahead there are no more than \( M - 1 \) other blocks with higher priority occurring after it in the reference string. Hence \( r_i \in B_i^+ \). These conditions guarantee that \( r_i \) is fetched in the following I/O. \( \Box \)

L-OPT as an off-line algorithm

In this section we show that L-OPT performs the minimum number of I/Os of any schedule for a read-once reference string that is known in advance. We first present the central theorem of our analysis. It shows that L-OPT services reference strings of length \( L \) in the optimal number of I/Os. In the off-line case, the entire reference string corresponds to only one window, and hence in this discussion we shall implicitly assume that all blocks belong to only one window; this will simplify our presentation as now we do not need to relate priorities of blocks in different windows.
We shall first present conditions on the possible values for the priority of a block. This property will be useful in showing the relation between the priority of a block and the I/O in which it is fetched by OPT.

**Property 3** Let the priority of some reference \( r \) be \( \text{priority}(r) = p \). Then either (i) there is a block \( a \) from \( \text{disk}(r) \) such that \( \text{priority}(a) = p - 1 \) or (ii) number of blocks referenced after \( b \) with priority at least \( p - 1 \) is more than \( M \) or (iii) \( p = 1 \).

**Proof:** Let \( d = \text{disk}(r) \). Consider the iteration in which a priority is being assigned to \( r \). Either \( r \) is assigned a priority \( \text{lowestPriority} \) or \( \text{lowestPriorityOnDisk}[d] \) set previously.

Consider the case when \( r \) is assigned priority \( \text{lowestPriority} \). Two cases are possible: either \( \text{lowestPriority} \) is 1 or not. Consider the case when \( \text{lowestPriority} \) is not 1; the other case leads the condition that \( p = 1 \). As we had discussed previously, \( \text{numberOfBlocksPlaced} \) keeps track of the number of blocks which have been assigned a priority at least \( \text{lowestPriority} \). Blocks are assigned priorities starting from the last block, and \( \text{lowestPriority} \) is incremented only when the value of \( \text{numberOfBlocksPlaced} \) reaches \( M \). Thus, the number of blocks with priority \( p - 1 \), not including \( r \) should be at least \( M \).

Next consider the case when it is assigned \( \text{lowestPriorityOnDisk}[d] \). Whenever a block from disk \( d \) is assigned a priority the value of \( \text{lowestPriorityOnDisk}[d] \) is incremented, and incremented beyond the value of \( \text{lowestPriority} \). Thus the value of \( \text{lowestPriorityOnDisk}[d] \) is higher than \( \text{lowestPriority} \) if and only if some other block has been assigned a priority \( \text{lowestPriorityOnDisk}[d] - 1 \). This is the case when there is a block from the same disk with priority \( p - 1 \).

**Theorem 5** To service a reference string of length \(|\Sigma| < L\), the number of I/Os done by any algorithm is lower bounded by the number of I/Os done by \( L\text{-OPT} \); that is, \( L\text{-OPT} \) services any reference string up to length \( L \) in the optimal number of I/Os.
**Proof:** When the length of the reference string \( \Sigma \) is less than \( L \), L-OPT (a) assigns a priority to all blocks before performing any I/Os and (b) does not change priorities while generating the I/O schedule. Moreover, in any I/O L-OPT fetches blocks with the highest priority, if necessary, by evicting blocks of lower priority. Hence, if \( p_i \) \((p_{i+1})\) is the largest priority of any block which is yet to be fetched after the \( i \)th \((i + 1)\)th I/O, then \( p_{i+1} < p_i \). Therefore, the largest priority of any block in \( \Sigma \) is an upper bound of the number of I/Os done by L-OPT to service \( \Sigma \).

Let OPT be the optimal I/O schedule to service \( \Sigma \). Let the length of the optimal schedule be \( T_{OPT} \). Let block \( r_i \) be fetched in the \( I_i \)th I/O in the optimal schedule. We shall show that the priority of \( r_i \) is at most \( T_{OPT} - I_i + 1 \); the bound then follows by noting that the priority of any block is hence at most \( T_{OPT} \).

The inductive hypothesis used to prove the lemma is as follows. OPT can be transformed into another schedule \( \text{OPT}_k \), of the same length, satisfying the properties of Claim 2, such that the following properties hold: for all \( r_i, i < k \), \( I_i^k = I_i \) and for all \( i \geq k \), \( T_{OPT} - I_i^k + 1 = \text{priority}(r_i) \), where \( I_i^k \) denotes the I/O in which \( r_i \) is fetched in \( \text{OPT}_k \).

Consider the base case when \( k = |\Sigma| \). By the definition of priorities, \( \text{priority}(r_k) = 1 \). \( \text{OPT}_k \) can be got from OPT be simply delaying the fetch of \( r_k \) till the last I/O. As blocks from any disk are fetched in order, there can be no block fetched from \( \text{disk}(r_k) \) after \( r_k \) in OPT. Moreover, either \( r_k \) is fetched at \( T_{OPT} \) or there is some other block referenced before \( r_k \) that is fetched in step \( T_{OPT} \). \( \text{OPT}_k \) is still of length \( T_{OPT} \). Moreover, as the I/O for \( r_k \) only is delayed, the resultant schedule satisfies the properties of Claim 2.

We shall now present an inductive transformation from \( \text{OPT}_k \) to \( \text{OPT}_{k-1} \) such that the hypothesis holds. From Property 3, three cases are possible depending on the relation between \( \text{priority}(r_{k-1}) \) and \( T_{OPT} - I_{k-1}^k + 1 \).

**Case 1.** \( \text{priority}(r_{k-1}) > T_{OPT} - I_{k-1}^k + 1 \). We shall show that this is not possible. By Property 3 three cases are possible.
**Case 1a.** There is a block $r_l$ from disk($r_{k-1}$) such that $l > k - 1$ and priority($r_l$) = priority($r_{k-1}$) - 1. As $l \geq k$ by the induction hypothesis, priority($r_l$) = $T_{OPT} - I_l^k + 1$. By Claim 2 blocks from any disk are fetched in order. Hence

$$\text{priority}(r_{k-1}) - 1 = T_{OPT} - I_l^k + 1$$

$$< T_{OPT} - I_{k-1}^k$$

This contradicts the assumption that priority($r_{k-1}$) > $T_{OPT} - I_{k-1}^k + 1$.

**Case 1b.** If $B_{k-1} = \{r_i : \text{priority}(r_i) \geq \text{priority}(r_{k-1}) - 1 \text{ and } i > k - 1\}$, then $|B_{k-1}| > M$. Consider a block $r_n \in B_{k-1}$. As $n > k - 1$, by the induction hypothesis, priority($r_n$) = $T_{OPT} - I_n^k + 1$. As priority($r_n$) ≥ priority($r_{k-1}$) - 1, using the assumption priority($r_{k-1}$) > $T_{OPT} - I_{k-1}^k + 1$, $I_n^k \leq I_{k-1}^k$. Hence all the blocks of $B_{k-1}$ are fetched either earlier than or together with $r_{k-1}$ by OPT$_k$. This is not possible as $|B_{k-1}| > M$, while the size of the buffer is $M$.

**Case 1c.** priority($r_{k-1}$) = 1. We then have a contradiction to the assumption that priority($r_{k-1}$) > $T_{OPT} - I_{k-1}^k + 1$, because $I_{k-1}^k \leq T_{OPT}$.

**Case 2.** priority($r_{k-1}$) = $T_{OPT} - I_{k-1}^k + 1$. Then take OPT$_{k-1}$ to be the same as OPT$_k$.

**Case 3.** priority($r_{k-1}$) < $T_{OPT} - I_{k-1}^k + 1$. Correspondingly, let $I^* = T_{OPT} - \text{priority}(r_{k-1}) + 1$. We shall show that we can delay the I/O for $r_{k-1}$ till $I^*$ to get OPT$_{k-1}$. Because OPT$_{k-1}$ should be an I/O schedule and satisfy Property 2, this requires us to show that if $r_{k-1}$ is fetched in I/O $I^*$ then (i) the number of blocks referenced after $r_{k-1}$ but fetched before or in I/O $I^*$ is less than $M$ (ii) there are no blocks from disk($r_{k-1}$) referenced after $r_{k-1}$ fetched before or in I/O $I^*$.

By construction, backlog($r_{k-1}$) ≤ $M$. Hence $|\{r_i : i \geq k - 1 \text{ and } \text{priority}(r_i) \geq \text{priority}(r_{k-1})\}| \leq M$. By the inductive hypothesis, for all $r_i$, $i \geq k$, priority($r_i$) = $T_{OPT} - I_i^k + 1$. Hence $|\{r_i : i \geq k - 1 \text{ and } T_{OPT} - I_i^k + 1 \geq T_{OPT} - I^* + 1\}| < M$.

This proves that condition (i) is satisfied.

Condition (ii) is trivially satisfied if there are no blocks referenced after $r_{k-1}$ from disk($r_{k-1}$). Otherwise, let the next block requested from that disk be $r_l$. We need to show that $I_l^k > I^* = T_{OPT} - \text{priority}(r_{k-1}) + 1$. We know that
priority(r_{k-1}) > priority(r_l). By the induction hypothesis, priority(r_l) = T_{OPT} - I^k + 1. Hence priority(r_{k-1}) > T_{OPT} - I^k + 1 + 1.

This allows us to construct OPT_{k-1} as follows. Take schedule OPT_{k-1} to be exactly the same as OPT_k, except for block r_{k-1} which is such that I^{k-1}_{k-1} = T_{OPT} + 1 - priority(r_{k-1}). \square

As an immediate consequence of the above theorem, L-OPT is the optimal off-line algorithm

**Corollary 1** L-OPT is an optimal off-line I/O scheduling algorithm for read-once reference strings.

**L-OPT as an online algorithm**

In this section we shall prove bounds on the competitive ratio of L-OPT. Theorem 6 presents the main result. The theorem is proved for the range $L > M$ in Lemma 7 and for $L \leq M$ in Lemma 8.

**Theorem 6** The competitive ratio of L-OPT is

$$\Theta(\sqrt{MD/L}) \quad \text{for} \quad L > M$$

$$\Theta(\sqrt{D}) \quad \text{for} \quad M \geq L \geq M/\sqrt{D}$$

$$\Theta(M/L) \quad \text{for} \quad M/\sqrt{D} \geq L > M/D$$

$$\Theta(D) \quad \text{for} \quad M/D \geq L$$

From Definition 1, the lookahead reveals one block at a time at a distance $L$ away. Hence each window now consists of exactly $L$ references. Let us start by considering the case when the lookahead is more than the size of memory, $L > M$.

**Lemma 7** The competitive ratio of L-OPT is $O(\sqrt{MD/L})$, for $L > M$.

**Proof:** Let the reference string be $\langle r_0, \ldots, r_n \rangle$, and let $T_i$ denote the number of IOs done by L-OPT to service the $i$th window, $\langle r_{iL}, \ldots, r_{(i+1)L-1} \rangle$. L-OPT services blocks one lookahead window at a time. From Theorem 5, we know that
L-OPT will perform the minimal number of I/Os to service any single window, as the entire window is known to it at the start of the window. In this proof we shall bound the benefit OPT, the optimal off-line schedule, gets by prefetching blocks across windows.

Consider the schedule generated by the optimal algorithm OPT to service \( \Sigma \). 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 the sequence \( \langle r_{iL}, \cdots , r_{(i+1)L-1} \rangle \) 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.

Let \( P_{i,j} \) denote the blocks from the set \( \{ r_{jL}, \cdots , r_{(j+1)L} \} \) prefetched by OPT in the sub-schedule \( S_i \). Because the size of the buffer is \( M \), \( |P_{i,j}| \leq M \). Let the maximum number of blocks in \( P_{i,j} \) which are from the same disk be \( \alpha_{i,j} \). Let \( \beta_i = \sum j \alpha_{i,j} \). Let the number of I/Os done by OPT in the sub-schedule \( S_i \) be \( I_i \). It immediately follows that \( \alpha_{i,j} \leq I_i \).

The blocks in \( P_{i,j} \) are present in the buffer while at least \((j - i - 1)L\) other requests between \( r_{(i+1)L} \) and \( r_{jL-1} \) are serviced. Because the buffer is of size \( M \), for each set of \( M \) such intermediate blocks OPT must do at least \( |P_{i,j}|/D \) I/Os; these I/Os are attributable to \( P_{i,j} \). Hence the number of I/Os done by OPT is at least

\[
T_{\text{OPT}} \geq \sum_i \sum_j [(j - i - 1)L/M] \times [\lfloor P_{i,j}/D \rfloor] \\
\geq \sum_i \sum_j [(j - i - 1)L/M] \times [\alpha_{i,j}/D] \quad \text{As } |P_{i,j}| \geq \alpha_{i,j} \\
\geq \sum_i \sum_{k=0}^{\lfloor \beta_i/I_i \rfloor} [kL/M] \times [I_i/D] \quad \text{As } \beta_i = \sum_j \alpha_{i,j} \text{ and } \alpha_{i,j} \leq I_i
\]

Thus we have that

\[
T_{\text{OPT}} \geq \sum_i [\beta_i/I_i]^2 L/M \times [I_i/D] - [\beta_i/I_i] \times [I_i/D]
\]

Using the bounds \( \beta_i = \sum_j \alpha_{i,j} \), \( \alpha_{i,j} \leq I_i \) and \( |P_{i,j}| \geq \alpha_{i,j} \)

\[
T_{\text{OPT}} \geq \sum_i \beta_i^2 L/MDI_i - O((L + \beta_i)/D)
\]
The above bound, together with the simple bounds $T_{\text{OPT}} \geq \sum_i I_i$, $T_{\text{OPT}} \geq \sum_i L/D$ and $T_{\text{OPT}} \geq \sum_i \beta_i/D$, implies that

$$T_{\text{OPT}} = O(\sum_i \beta_i \sqrt{L/MD})$$

From Lemma 5, $T_j$ is the minimal number of I/Os required to service the reference string $\langle r_{jL}, \cdots, r_{(j+1)L-1} \rangle$. Hence $T_j \leq I_j + \sum_i \alpha_{i,j}$. This implies that

$$\sum_j T_j \leq T_{\text{OPT}} + \sum_i \beta_i$$

\qed

In the lower range of lookahead, $L \leq M$, L-OPT behaves like a greedy prefetching algorithm that prefetches everything in the lookahead. In this case, the lookahead is smaller than the buffer size, hence L-OPT will never encounter a case where it needs to evict some blocks from the buffer to make space for some other blocks in the lookahead which have a higher priority. This situation is very similar to that of PHASE (and GREED) of Section 3.2. The competitive ratio of L-OPT is hence the same as that of PHASE for this range of lookahead.

**Lemma 8** When $L < M$, the competitive ratio of L-OPT is

- $\Theta(\sqrt{D})$ for $M \geq L \geq M/\sqrt{D}$
- $\Theta(M/L)$ for $M/\sqrt{D} \geq L > M/D$
- $\Theta(D)$ for $M/D \geq L$

As an aside, it is interesting to consider what influence the greedy prefetching engine has on L-OPT's performance. Note that the priorities could themselves be directly used to generate a schedule. However by fetching greedily, and fetching blocks from the next window we aim to avoid wasting I/O slots unnecessarily. It may happen that when additional lookahead is revealed, some priorities are advanced. The greedy prefetching engine can accommodate such a situation easily; a prefetching algorithm which followed the priorities strictly could fall out of track in these cases.
3.5 Comparing on-line algorithms

In this section we shall compare online algorithms; that is we will compare algorithms with the same amount of lookahead. To precisely characterize how well online algorithms use the lookahead that is available to them, we shall use the *online ratio* (Section 2.4) as the performance metric. Intuitively, the online ratio is the ratio of the number of I/Os done by the online algorithm to the number of I/Os needed by the minimal length schedule that can be generated by any online algorithm with the same amount of lookahead. This metric attempts to differentiate different online algorithms on the basis of how well they use the lookahead information that is available to them.

Given an input reference string it is useful to determine a lower bound on the minimal number of I/Os required by an on-line algorithm with $L$-block lookahead to service it. To do so we need a characterization of schedules which could have been generated by prefetching algorithms using $L$-block lookahead. Intuitively, in a schedule that has been generated by an on-line algorithm with $L$-block lookahead, in any I/O the blocks fetched are within $L + 1$ blocks of the last reference serviced.

**Definition 9** A schedule to service a reference string is said to have the $L$-property if among the blocks fetched in any I/O the block occurring farthest is within $L + 1$ blocks of the last request that was serviced.

In its essence, the above definition indicates that in an on-line schedule every block must be in the lookahead when an I/O for that block is initiated. From Definition 1 of global $L$-block lookahead, the lookahead window extends to at most $L - 1$ blocks beyond the next block to be referenced. Hence if the next block to be referenced is $a$, then all blocks that are fetched in the next I/O have to be within $L$ blocks of $a$.

For a given reference string $\Sigma$, let us consider the minimum length schedule ON-OPT that can be constructed by any online algorithm with $L$-block lookahead. ON-OPT should have the $L$-block property as otherwise some block outside the lookahead has been fetched in OPT. Also, the length of the shortest
schedule having the $L$-property is a tight lower bound on the length of any schedule that can be generated by any online algorithm to service the given reference string.

Ideally we would like to have an optimal deterministic online algorithm, which generates the smallest length I/O schedule for all reference string. Interestingly, it turns out that this is only possible when the lookahead is less than the buffer size, $L < M$. When $L > M$, we show in Theorem 7 that there is no single online algorithm that is guaranteed to generate the minimal length I/O schedule in comparison to all other schedules that can be generated by online algorithms. As part of the analysis, we will show that the online ratio of any online algorithm with $L$-block lookahead can be close to 2. While we show that the online ratio of L-OPT is upper bounded by 2 for all lookaheads, we leave it as an open problem to either tighten the bound on L-OPT's online ratio, or tighten the lower bound for any online algorithm.

In the case when the lookahead is less than the buffer size, $L < M$, the following lemma indicates that L-OPT is in fact the optimal online algorithm. In this range of lookahead we had shown, in Theorem 2, that PHASE generates the minimal length I/O schedule for all reference strings when compared to all other online algorithms with the same lookahead. As we have discussed before, L-OPT behaves exactly like PHASE in this range of lookahead. Hence the following lemma, from Theorem 2.

**Lemma 9** When $L < M$, L-OPT is the optimal online algorithm with $L$ block lookahead; that is, for any reference string there is no I/O schedule that could be generated, with only $L$ block lookahead, which is of smaller length than L-OPT's schedule.

The more interesting case is when the lookahead $L > M$.

**Theorem 7** There is no optimal online parallel I/O scheduling algorithm for $L > M$. That is, there is no online parallel I/O scheduling algorithm with $L > M$ block lookahead that generates an I/O schedule of the minimal length possible with $L$-block lookahead for all reference strings.
Lemma 10 There is no optimal online algorithm when the lookahead $L > M$; that is the online ratio of any online algorithm is greater than 1.

Proof: Let $A$ be an arbitrary online read-once scheduling algorithm. Let ON-OPT denote the minimal length I/O schedule satisfying the $L$ block property.

We shall show that $A$ has an online ratio of $X$ by constructing a nemesis reference string such that the number of I/Os done by $A$ is at least $X$ times the length of ON-OPT. The reference string, $\eta$ will constructed as an alternating sequence of around $L/D$ blocks all referenced from the same disk with a sequence of $L$ blocks striped in a round-robin fashion across all disks.

We shall define two kinds of sequences of requests loaded and balanced, similar to the definitions used in Section 3.1.

Definition 10 A balanced phase $\text{balanced}(i)$ is a subsequence of the reference string of length $L$. The constituent $L$ blocks are striped in a round-robin manner across $D - 1$ disks, all except disk 1. A loaded phase $\text{loaded}(i)$ with hot disk $d_i$ consists of $H$ blocks laid out such that the constituent blocks all originate from the hot disk $d_i$. $H$ is an integer parameter.

![Figure 3.9: Reference string used to lower bound online scheduling algorithms](image)

We shall construct the nemesis reference string $\eta$ as a sequence of alternating loaded and balanced phases. By the above definition and the definition of global $L$-block lookahead, when the first block of a loaded phase is referenced, an algorithm with $L$-block lookahead knows all of the blocks (and their order of reference) in that loaded phases, and $L - H$ blocks in the next balanced
phase. As the computation proceeds, the lookahead window includes additional blocks, but while serving the references of a loaded phase, the lookahead never includes any blocks from the subsequent loaded phase. The reference string \( \eta \) is illustrated in Figure 3.9.

**Definition 11** Let \( P \) be an integer parameter less than \( D \). We construct a reference string \( \eta \) of \((P+1)H+PL\) references such that the nemesis string \( \Sigma \) can be obtained by concatenating strings constructed in the same manner as \( \eta \) an arbitrary number of times. The reference string \( \eta \) is an alternating sequence of loaded and balanced phases, loaded(0), balanced(1), \ldots, balanced(P), loaded(P).

The hot-disk of the first loaded phase, loaded(0), is 1. The hot-disk of a subsequent loaded phase, loaded(\(i\)), is determined as follows. After servicing the last reference of loaded(\(i-1\)), \( A \) could have some blocks from the next balanced balanced(\(i\)) in the buffer. Now consider \( S \), the set of all disks which have not been a hot-disk for any previous loaded phase. Let \( d_i \) be a disk from \( S \), from which \( A \) has the minimum number of blocks, from that balanced phase, in the buffer. The hot-disk of loaded(\(i\)) is set to \( d_i \).

Let us first consider the number of I/Os done by \( A \) to service the reference string \( \eta \). \( A \) will perform \( H \) I/Os to fetch the blocks from the first loaded phase. Now consider any subsequent set of \( L + H \) references, balanced(\(i\)) and loaded(\(i\)). Let the hot-disk in loaded(\(i\)) be \( d_i \). In these set of \( H + L \) references, there are at least \([L/(D-1)] + H\) blocks referenced from disk \( d_i \). By construction, \( A \) could have prefetched at most \( M/(D-i) \) blocks from balanced(\(i\)). Hence the total number of I/Os done by \( A \) to service references in this set of references is at least \([L/(D-1)] + H - M/(D-i)\). Thus the total number of I/Os done by \( A \) is at least

\[
T_A \geq H(P+1) + P[L/(D-1)] - \sum_{i=1}^{P} M/(D-i) \\
\geq H(P+1) + P[L/(D-1)] - M(H_{D-1} - H_{D-P-1})
\]

where \( H_k = \sum_{i=1}^{k} 1/i \).

We shall next show that a schedule, which has the \( L \)-block property, can be generated that is of length \( H + PL/(D-1) \), whenever \( HP \leq M - D, P \leq D - 1, \)
and $H \leq (L - D)/D$. We shall construct a schedule ON-OPT that satisfies the $L$ block property as follows.

While fetching the first $H$ references of loaded(0) we fetch $H$ references from the next balanced phase, balanced(1), from all the disks which are hot in some loaded phase. This satisfied the $L$ property as blocks from balanced(1) are within $L$ references of blocks in loaded(0) when they are fetched. This can be done because all the hot-disk are chosen to be on different disks, and there is enough buffer space as $HP < M$. Now recursively, while fetching the references of balanced($i$), we fetch references from balanced($i + 1$) from all the hot disks of loaded phases beyond (and including) loaded($i$). We can fetch $H$ references from all such disks if there $H$ I/Os performed after the first $D$ blocks of balanced($i + 1$) are visible; that is $H \leq (L - H - D)/(D - 1)$. Thus the total number of I/Os done by ON-OPT is at most

$$T_{ON-OPT} \leq H + PL/(D - 1)$$

Setting $H = \lfloor L/(D - 1) \rfloor$ and $P = \lfloor (M - D)/H \rfloor$, the ratio of the number of I/Os done by $\mathcal{A}$ and ON-OPT is then

$$((2P + 1)\lfloor L/(D - 1) \rfloor - M(H_{D-1} - H_{D-P-1}))/((P + 1)(L/(D - 1))$$

which is at least $2 - ((L/(D - 1) + (2P + 1) + M \log((D - 1)/(D - P - 1)))/(M - D))$. \hfill \Box$

Finally, we shall show that the online ratio of L-OPT is 2.

**Lemma 11** The online ratio of L-OPT is upper bounded by 2.

**Proof.** From Theorem 5, the number of I/Os done by L-OPT to service a reference string $\Sigma = \langle r_0, r_1, \ldots, r_n \rangle$ is $\sum_i T_i$, where $T_i$ is the optimal number of I/Os needed to service the reference string $\Sigma_i = \langle r_{iL}, r_{iL+1}, \ldots, r_{(i+1)L-1} \rangle$. We shall complete the proof by arguing that any schedule, ON-OPT, to service $\Sigma$ that satisfies the $L$ block property will perform at least $\sum_i T_i/2$ I/Os.

Since ON-OPT satisfies the $L$-block property, the I/O in which a the last block of $\Sigma_{i-1}$, block $r_{iL-1}$, is fetched and the first I/O in which the first block of
\( \Sigma_{i+1}, \) block \( r_{(i+1)L} \), is fetched are distinct. Hence the number of I/Os done by ON-OPT can be lower bounded by counting the number of I/Os done by ON-OPT to fetch blocks in alternate \( \Sigma_i \). This gives us two bounds on the number of I/Os done by ON-OPT: \( T_{\text{ON-OPT}} \geq \sum_{\text{even}} T_i \) and \( T_{\text{ON-OPT}} \geq \sum_{\text{odd}} T_i \). Together this proves that \( T_{\text{ON-OPT}} \geq \sum_i T_i/2 \). \( \square \)

### 3.5.1 Comparing GREED, PHASE, and L-OPT

In this section we compare the three online algorithms described in this chapter: GREED, PHASE, and L-OPT. We consider the relative performance of to the other: we consider the worst case ratio of the number of I/Os done by one algorithm \( A \) to that done by the other, \( B \), while servicing the same reference string with the same amount of lookahead. We shall refer to this metric as the relative online ratio of \( A \) compared to \( B \).

When the lookahead is less than memory size, then we have shown in Theorem 2 and Lemma 9 that all three algorithms produce the same shortest length I/O schedule. The following table summarizes the ratios when \( L > M \); the entry in row \( A \) and column \( B \) represents the relative online ratio of algorithm \( A \) compared to algorithm \( B \).

<table>
<thead>
<tr>
<th></th>
<th>GREED</th>
<th>PHASE</th>
<th>L-OPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GREED</td>
<td>( - )</td>
<td>( \Theta(D) )</td>
<td>( \Theta(D) )</td>
</tr>
<tr>
<td>PHASE</td>
<td>( \Theta(\min{L/M, \sqrt{D}}) )</td>
<td>( - )</td>
<td>( \Theta(\min{L/M, \sqrt{D}}) )</td>
</tr>
<tr>
<td>L-OPT</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( - )</td>
</tr>
</tbody>
</table>

Table 3.1 : Comparing GREED, PHASE, and L-OPT when \( L > M \). The entry in row \( A \) and column \( B \) represents the relative online ratio of algorithm \( A \) compared to algorithm \( B \).

Note that since all the algorithms perform at most 1 I/O per request in the reference string, any entry in the table can never be greater than \( D \). In the same token any entry in the table can never be less than 1: if any two
algorithms are issued a reference string that contains references from only one
disk, both will do the same number of I/Os.

Let us first consider the relative online ratio of GREED compared to ei-
ther PHASE or L-OPT. Using the same reference string as in Lemma 5 we can
show that the worst case ratio of the number of I/Os done by GREED to ei-
ther PHASE or L-OPT is at least \( \Omega(D) \). Hence the entries in the first row of
Table 3.1.

Lemma 11 provides the relative online ratio of L-OPT compared to GREED
and PHASE.

The relative online ratio of PHASE is more interesting. First let us compare
PHASE to L-OPT. From Theorem 3, the competitive ratio of PHASE is \( \Theta(\sqrt{D}) \)
when \( L \geq M \). Hence, for any reference string, the worst case ratio of the
number of I/Os done by PHASE to those done by L-OPT can be at most \( O(\sqrt{D}) \).
This gives us one bound on the relative online ratio of PHASE compared to L-
OPT. When the lookahead is less than \( M \sqrt{D} \), it can be shown that the ratio is at
most \( O(L/M) \). The proof follows along the same lines as that used to show that
the competitive ratio of PHASE is \( \Theta(\sqrt{D}) \) in Theorem 3. The only difference
is that the lookahead poses an additional constraint on how well L-OPT can
optimize prefetching. Because both algorithms have a lookahead \( L \leq M \sqrt{D} \),
in the schedule generated by L-OPT no block can be prefetched greater than
\( \lfloor L/M \rfloor \) phases in advance. This gives us the second bound of \( O(M/L) \) on the
relative online ratio of PHASE compared to L-OPT. Both the bounds can be
shown to be tight by specializing the proof of Lemma 1, so that the the length
of the nemesis reference string is \( L \) and the optimal scheduling algorithm is
replaced by L-OPT, which performs the minimum number of I/Os to service a
sequence of \( L \) requests.

Interestingly the ratio of the number of I/Os done by PHASE to those done
by GREED can also be \( \Omega(\min\{L/M, \sqrt{D}\}) \). For simplicity, we shall illustrating
the lower bound for the case where the length of the lookahead is \( \Theta(M \sqrt{D}) \);
the lower bound for the other range of lookahead can be generated in the same
fashion.
The nemesis reference string consists of three kinds of reference strings.

- The *driver* is a sequence of \( M\sqrt{D}/2 \) references all striped in a round-robin fashion across the first \( D/2 \) disks.

- The *eraser* is a sequence of \( MD/(D - 1) \) references with the first \( M/(D - 1) \) references from disk 1, and the rest striped in a round-robin fashion across disks 1 through \( D/2 \).

- The *peak* is a sequence of \( M/\sqrt{D} \) requests all from the same disk; the disk will be assigned later.

The reference string is constructed to have a driver as the first sequence of requests, followed by alternating erasers and peaks, with the \( i \)th peak from disk \( D/2 + i \). There are a total of \( \sqrt{D} \) erasers and peaks.

It can be seen that PHASE performs \( O(M/D) \) I/Os to service the references in an eraser sequence, and evicts any prefetched blocks that it may have in the buffer at the start of the sequence. On the other hand, if GREED has \( D/2 \) free blocks in the buffer then it services the sequence in \( 2M/D - 1 \) I/Os, not evicting any prefetched blocks that it may have in the buffer.

Now both PHASE and GREED prefetch the blocks from subsequent peaks while servicing the first driver sequence. However, PHASE evicts all prefetched blocks while servicing intermediate eraser sequences, while GREED does not. As a result PHASE performs \( \Omega(M) \) I/Os to service the entire reference string, while GREED performs only \( O(M/\sqrt{D}) \) I/Os. This shows that the relative online ratio of PHASE compared to GREED is \( \Omega(\sqrt{D}) \), when the lookahead is \( \Omega(M\sqrt{D}) \).

### 3.6 Simulation study

In this section we present an empirical evaluation of algorithm L-OPT. We studied the performance of L-OPT using synthetically generated data based on a bursty access pattern. In order to help benchmark prefetching algorithms, we present a method to determine the minimal number of I/Os required by any
online algorithm with $L$-block lookahead to service a given reference string. We then compare the performance of this lower bound to the number of I/Os made by L-OPT and PHASE, described in [6]. We use the optimal off-line schedule length as the baseline.

Based on the definition of $L$-block property (Definition 9), one way to find a tight lower bound on the length of the online schedule is to find a minimal length schedule which services the given reference string and which has the $L$-property. A modification of L-OPT allows us to generate such a minimal length schedule. To do so we just need to modify the condition "if (numberOfBlock-\-sPlaced = M) then" to be "if (numberOfBlock-\-sPlaced = \text{min}(M, L)) then"

The proof that this generates a minimal length schedule having the $L$-property is similar to the proof of Lemma 5. Given a reference string, this method is used to derive a lower bound on the number of I/Os needed by any online algorithm with $L$-block lookahead to service it.

In this simulation study we model the reference string as a bursty sequence as next described. Blocks are accessed from any disk with uniform probability. Occasionally there is a burst of I/O requests to blocks from a small set of disks. These disks are referred to as hot-spot disks. The input parameters to the data model are: (a) the percent $f$, of blocks which occur on hot spot disks and (b) the number of hot spot disks $k$. Specifically, the probability that a block is on a hot-spot disk is $f/100$. The actual disk is randomly chosen from among the set of current hot-spot disks. The set of hot-spot disks changes during the reference string. At any instant the probability of the set of hot spot disks changing is proportional to the duration for which the set has remained unchanged. The new set of $k$ hot-spot disks is chosen at random from the set of all disks.

The motivation of the model is to capture a wide range of access patterns. When $f$ is small data is accessed randomly from all disks allowing almost all prefetching algorithm to perform equally well. When $f$ is close to 1, data is accessed sequentially from a small set of disks and again most algorithms will perform equally poorly due to the lack of parallelism to exploit. The intermediate range represents reference strings with substantial parallelism that can
be exploited by intelligent prefetching algorithms.

In this study we compare the performance of L-OPT with PHASE, defined in Section 3.2. We consider two configurations of the I/O system and count the number of I/Os performed by each algorithm. Though this approach neglects other factors occurring in practical I/O systems (like disk latency, block size, bus contention, etc.) this allows controlled evaluations of prefetching algorithms. We consider a system of 50 disks with 250 and 1000 blocks of I/O buffer. The lookahead is fixed at 3000 blocks.

Figure 3.10 presents a plot of the number of I/Os done by the four algorithms as the percent of random blocks is varied. The number of I/Os of all the algorithms is normalized by the number of I/Os done by the optimal off-line algorithm. The I/O system has a buffer of size 250. The number of hot spot disks is fixed at 5 and 35, respectively, in Figures 3.10(a) and (b). Figure 3.11 presents a similar plot when the buffer is of size 1000 blocks.

When the buffer is of size 250, the lookahead window spans 12 buffer-loads of accesses. Due to this there is practically no difference between the number of I/Os needed by the the best schedule with the L-block property and the optimal off-line. The loss due to incomplete lookahead becomes evident only in the case
when the buffer is of size 1000.

In all cases the number of I/Os required by PHASE and L-OPT increase with the percentage of blocks on hot-spot disks and then decreases. When the percentage of blocks on hot-spot disks is low there is not much additional parallelism to be extracted by fetching blocks required much later. Hence both PHASE and L-OPT are close to the optimal. However as the percentage of such blocks increases, the optimal algorithm can exploit parallelism missed by both PHASE and L-OPT. However L-OPT performs much better than PHASE by extracting a substantial amount of parallelism from the lookahead window. Hence L-OPT follows the lower bound closely. As the percentage of blocks on the hot-spot disks increases further, most of the blocks are on the hot-spot disks and the situation is similar to that on the lower end. However the number of hot-spot disks is smaller than the total number of disks, and the actual disks in the set change. This is the reason why the curves is not symmetric: OPT can benefit more from the variations than both PHASE and L-OPT.

Finally as the number of hot spot disks increases the peaks of all curves shift to the right. This is because for a given percent of blocks that are on hot-spot disks, as the number of hot-spotting disks increases the number of
additional I/Os that the algorithms need to do to catch up with the optimal schedule is fewer. Hence the percentage of such blocks has to be higher to get the same ratio.

In summary, apart from the extremes L-OPT performs very close to the best achievable with the given amount of lookahead. The primary loss of performance for L-OPT compared to the optimal off-line is its bounded lookahead. The difference between it and PHASE is more pronounced when the amount of lookahead available is more compared to the amount of buffer size.
Chapter 4

Scheduling read-often reference strings

Read-often reference strings model a general I/O access pattern where each read is to a distinct block. In addition to facilitating prefetching, the I/O buffer in such a situation can be used to improves I/O latency by caching blocks to avoid repeated disk accesses for the same block. To use temporal locality together with I/O parallelism effectively, we also need to design caching strategies that retain the most valuable blocks in the buffer when the need for eviction arises.

In this chapter we consider the combined problem of prefetching and caching for parallel I/O systems. We will present a new priority controlled greedy algorithm, SUPERVISOR, for optimizing prefetching and caching decisions in a parallel I/O system.

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 higher 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 the previous chapter (Chapter 3) we have shown 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 what blocks to fetch in an I/O is non-intuitive. A good prefetching and caching algorithm ought to co-operatively decide how
much buffer space to allocate for prefetching in a particular I/O and which blocks ought to be prefetched then.

As before, 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, the only conditions being 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 on-line 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.

Traditional paging policies for buffer management have concentrated on minimizing the total number of requests that cause a disk access. However, with multiple disks, the total number of page faults is not a sufficient criterion due to I/O parallelism. For instance, let us consider the off-line caching algorithm MIN [7], which evicts the block that will be next requested farthest in the future. MIN is known to minimize the number of page faults in a sequential 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.

<table>
<thead>
<tr>
<th>Disk A</th>
<th>( a_4 )</th>
<th>( a_1 )</th>
<th>( a_2 )</th>
<th>( a_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk B</td>
<td>( b_3 )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Disk C</td>
<td>( c_2 )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Disk A</th>
<th>( a_4 )</th>
<th>( a_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk B</td>
<td>( b_3 )</td>
<td>( b_1 )</td>
</tr>
<tr>
<td>Disk 3</td>
<td>( c_2 )</td>
<td>( c_1 )</td>
</tr>
</tbody>
</table>

(a) Using MIN as eviction policy  
(b) Optimizing for multiple disks

Figure 4.1: Influence of replacement strategy on I/O schedule length

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 \( a_4, b_2, \) 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 4.1 (a) shows the schedule generated by an algorithm which uses MIN to service this reference string. Since the blocks \( a_1, a_2, \) and \( a_3 \) are the blocks to be referenced farthest in the future, these are the blocks chosen by MIN to be the candidates for eviction. However, as these three blocks are from the same disk, three I/Os need to be done to fetch them back later, thereby needing totally 4 I/Os to service the remainder of the reference string.

On the other hand, an algorithm which takes parallelism into consideration can generate the schedule shown in Figure 4.1 (b). Instead of evicting the MIN blocks, blocks \( a_1, b_1, \) and \( c_1 \) are evicted. These can later be fetched in one parallel I/O since all three are from different disks. This gives a shorter schedule, of length 2.

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. In the sequential I/O case the emphasis is only on maximizing the number of requests that do not need a disk access. As this example illustrates, not considering I/O parallelism can lead to additional unnecessary parallel I/Os. In fact, this example can be generalized to prove that any algorithm that uses MIN as the page replacement policy can perform \( D \) times more parallel I/Os than the optimal, where \( D \) is the number of disks: that is the worst possible dilation in the schedule due to not accounting for I/O parallelism. The algorithm \texttt{SUPERVISOR}, presented later in this paper, makes optimal caching decisions in the off-line case. It carefully delays prefetches so that more buffer space can be used for caching, and also carefully decides which blocks to evict so that they can be fetched back with high parallelism when maximizing parallelism when they need to fetched back later.

In this chapter we first present, in Section 4.1, fundamental bounds on the performance of algorithms having global \( L \)-block lookahead. When the looka-
head is smaller than $M - \sqrt{D}$, the effect of not being able to make effective caching decisions is dominant. The small lookahead can be exploited by an adversary to cause the algorithm make $\Omega(M - L)$ times more I/Os than the optimal off-line. Note that unlike for read-once reference strings, the competitive ratio of algorithms for scheduling read-often reference strings can be as bad as $M$; for instance the competitive ratio of LRU with no lookahead is $M$ in a sequential I/O system [33]. When the lookahead is more than $M - \sqrt{D}$ but less than $M$, the competitive ratio can be lower bounded by $\Omega(\sqrt{D})$, and by $\Omega(\sqrt{MD/L})$ when the lookahead is more than $M$. These bounds are the same as that for read-once reference strings, indicating that the scheduling problem is predominantly a prefetching problem in this range of lookahead.

MIN, which evicts the block to be referenced farthest in the future, is known to be optimal for sequential I/O systems. However, we show in Section 4.2 than any off-line algorithm that uses MIN to perform evictions could completely sequentialize accesses in a parallel I/O system.

In Section 4.3 we show how a randomized data layout can be used together with LRU and a simple greedy prefetching algorithm. We show that the expected number of I/Os performed by the scheme, PHASE-LRU, is within $O(\log D / \log \log D)$ of the optimal. Thus a randomized data layout not only simplifies prefetching, by making the load on disks uniform, but can also aid caching by minimizing the penalty due to erroneous eviction decisions.

The buffer management algorithm SUPERVISOR, presented in Section 4.4, 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 parallel disk model that we are aware of. Unlike the known MIN algorithm, which evicts pages that will not be requested for the longest time, SUPERVISOR is based on the intuition that it may be more useful to evict pages that can be fetched in parallel when needed. In an online scenario, SUPERVISOR uses available lookahead to make prefetching and caching decisions dynamically. Specifically, we show that the competitive ratio of SU-
PERVISOR is $O(L - M + D)$ when $L \leq M$, and $O(MD/L)$ 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.

### 4.1 Limits of lookahead

In this section we show that the competitive ratio of any online prefetching algorithms serving read-often reference strings with global $L$-block lookahead is at least

$$
\Omega(\max\{M - L, \sqrt{D}\}) \quad \text{for} \quad L \leq M
$$

$$
\Omega(\sqrt{MD/L}) \quad \text{for} \quad L > M
$$

Comparing these results to similar bounds for read-once reference strings (from Section 3.1), we can see that the bounds are different only when the lookahead is less than $M$. In this range of lookahead the effect of caching dominates; the small lookahead can be exploited by an adversary to cause the online algorithm to repeatedly make poor eviction decisions.

The bound when the lookahead $L > M$ are from Theorem 1. When the lookahead is less than $M$, the bound follows from the result for read-once reference strings and a lower bound from just due to caching, in a single disk I/O system.

**Lemma 12** The competitive ratio of any online algorithm scheduling read-often reference strings is $\Omega(\max\{M - L, \sqrt{D}\})$ when $L < M$.

**Proof:** The first bound of $\Omega(\sqrt{D})$ is from Lemma 4 for the competitive ratio of online read-once scheduling algorithms.

We shall show the other bound of $\Omega(M - L)$ by considering just a single disk. A similar bound was shown for online sequential paging algorithms with strong lookahead in [3]. Let $A$ be an online algorithm with $L$-block lookahead. We will assume, without any loss in generality, that in the schedule generated by $A$, it always performs an I/O only on demand; that is, it performs an I/O only when the requested block is not present in the buffer.
We will construct the nemesis reference string as a series of substrings \( \delta, \eta_1, \eta_2, \ldots \eta_N \), where \( \delta \) consists of \( M \) references and each \( \eta_i \) consisting of references to \( L \) blocks. The last \( L - 1 \) references of all the \( \eta_i \) will be the same, while the first block will be based on prior caching decisions made by \( \mathcal{A} \). \( \delta \) consists of references to \( M \) blocks \( b_1, \ldots, b_M \), in order.

The first \( 2L \) references in \( \eta_i \) is the string \( \sigma = \langle r, b_1, b_2, \ldots b_{L-1} \rangle \) repeated twice; we shall specify \( r \) later. If after servicing the first \( L \) references of \( \sigma \), all the blocks of \( \sigma \) are not present in the buffer, then the same string \( \sigma \) is repeated two more times. \( \eta_i \) is thus extended till at some time all blocks of \( \sigma \) are present in the buffer. At this time, let a block of \( \delta \) that is not present in the buffer be \( b_k \); \( b_k \) is chosen as the first reference of \( \eta_{i+1} \). \( b_{M+1} \) is chosen as the first reference of \( \eta_1 \).

From the construction, we can note that \( \mathcal{A} \) will do at least one I/O to service each \( \eta_i \). Thus the number of I/Os done by \( \mathcal{A} \) is at least \( \Omega(N) \), when \( N \) is sufficiently large. We shall show that the competitive ratio of \( \mathcal{A} \) is \( O(M - L) \) by showing that the same reference string can be serviced in \( O(N/(M - L)) \) I/Os.

We can service the reference string in \( O(N/(M - L)) \) I/Os as follows. Consider the set \( S_j \), which are the first references in \( \eta_{j(M-L)+1}, \ldots, \eta_{(j+1)(M-L)} \), where \( j \geq 0 \). We could cache these blocks together with \( b_1, \ldots, b_L \) at the beginning of \( \eta_{j(M-L)+1}, j \geq 0 \), and perform only one I/O every \( M - L \) \( \eta_i \)s. Thus we can service the entire reference string in \( O(N/(M - L)) \) I/Os.

\[ \square \]

4.2 Performance of Intuitive Algorithms

In the case of sequential I/O, it is known that an off-line algorithm that evicts the block which is referenced farthest in the future is optimal [7]. However, such a caching strategy does not consider the load on the disks when these evicted blocks may need to be fetched. In this section we shall show that any prefetching algorithm that uses MIN as the buffer replacement policy will sequentialize I/Os substantially and have a competitive ratio of \( \Omega(D) \). Thus using MIN can result in ignoring all parallelism available in the reference string.
**Lemma 13** Any algorithm that uses MIN as the eviction policy has a competitive ratio of $\Omega(D)$.

**Proof:** Let $A$ be an off-line scheduling algorithm that uses MIN as the eviction policy; that is, at any time if it needs to evict $k$ blocks from the buffer, then the $k$ chosen blocks are those whose next reference is farthest in the future.

We shall construct a reference string $\Sigma$ and a schedule $S$ to service it, such that the number of I/Os made by algorithm $A$ to service $\Sigma$ is $\Omega(D)$ times more than the length of $S$. The simple reference string $\Sigma$ is formed as a sequence of two strings $\alpha\beta\alpha\beta\alpha\ldots$: $\alpha$ is a sequence of $M$ blocks all referenced from disk 1; while $\beta$ is a sequence of $M$ references, which are accesses to blocks different from those in $\alpha$ and striped across all the disks. Let the number of $\alpha$ sequences be $a$, and $b = 2a$ be the number of $\beta$ sequences.

Now by caching all but $D$ blocks referenced in $\alpha$, each $\alpha$ subsequence can be serviced in just $D$ I/Os. The $D$ free buffer blocks can be used to prefetch one stripe of $\beta$ at a time, to service each $\beta$ subsequence in $M/D$ I/Os. Thus the number of I/Os needed to service the entire reference string is $M + (a - 1)D + bM/D$, which is $\Omega(aM/D)$ is $a$ is sufficiently large.

Now consider the functioning of $A$; specifically the blocks in the buffer after servicing the first of the two consecutive $\beta$ sequences. Since the buffer capacity is $M$, and there are $2M$ distinct blocks referenced in $\alpha$ and $\beta$, there can only be $M$ of these blocks in the buffer at this time. The next occurrence of any block of $\beta$ is earlier than any block of $\alpha$. Because $A$’s policy is to evict blocks whose next reference is farthest in the future, all the blocks in the buffer at this time should be those in $\beta$. Thus in each $\alpha$ subsequence, $A$ incurs $M$ I/Os. Hence the total number of I/Os done by it is $\Omega(aM)$. Hence the competitive ratio of $A$ is at least $\Omega(D)$.  

\[\square\]

### 4.3 Exploiting a randomized layout

In this section we consider the prefetching and caching problem in a probabilistic setting, where the data layout is randomized. Randomizing the data layout
allows the load on all disks to be uniformly balanced. As we have shown in Section 3.3, this can allow simple algorithms to get a high degree of parallelism for scheduling read-once reference strings, even with a small (Θ(D log D)) lookahead. In this section we shall show that even in the case of read-often reference a randomized data placement can help parallelize disk accesses, but more lookahead (Θ(M)) is necessary to perform efficient eviction decisions.

We shall show that a simple extension of PHASE, defined in Section 3.2, together with Least Recently Used (LRU) as the caching policy has an expected competitive ratio of Θ(log D / log log D) when the memory size is at least M = Ω(D log D) and the lookahead is M.

To schedule read often reference strings, we adapt algorithm PHASE to use LRU as follows.

**PHASE-LRU** orders blocks in the lookahead based on the index of their first appearance in the lookahead. In any I/O it fetches the earliest occurring block from all disks, evicting the block that is not in the lookahead, and which was referenced earliest in the past.

**Analysis**

In this section we shall analyze the performance of PHASE-LRU when L = M > Ω(D log D) and show that its competitive ratio is O(log D / log log D).

From the analysis of PHASE for read-once reference strings, we know that PHASE can fetch blocks from the disk with Θ(D) parallelism when the buffer is Ω(D log D). Here we will leverage that bound together with another result that we can make near optimal evictions (total number of blocks evicted) when the lookahead is more than the buffer size.

We analyze the performance of PHASE in sequences that consist of M distinct blocks called *phases*. Within a phase blocks are characterized as new, old or ancient depending on where they were referenced previously. This classification of blocks is illustrated in Figure 4.2.

**Definition 12**
The reference string is partitioned into subsequences of maximal length such that the number of distinct blocks, called phases, such that the number of distinct blocks in any phase is at most $M$. The first reference is in phase(1).

- The set of new blocks in phase($i$) is the set of blocks in phase($i$) not requested in any, phase($j$), $j < i$. The set of old blocks in phase($i$) is the set of blocks in phase($i$) requested in some phase($j$), $j \leq i - 1$. An old block is ancient if its last reference is in some phase($j$), $j \leq i - 2$.

We say that a block is prefetched for phase($i$) if the earliest future reference of that block is in phase($i$). We shall refer to an I/O as being for phase($i$) if the earliest block fetched in that phase is for phase($i$). We shall first bound the number of blocks that are not new, which PHASE-LRU may need to fetch in a phase.

**Lemma 14** The expected value of the total number of I/Os done by PHASE-LRU is at most $T_{\text{PHASE-LRU}} = O(\sum_i (a_i + n_i) \log D / (D \log \log D))$, where $a_i$ and $n_i$ are the number of ancient and new blocks, respectively, in phase($i$), and the sum is taken over all phases in the reference string.

**Proof:** We shall prove the lemma by considering separately the I/Os done by PHASE-LRU for new and old blocks. We shall show that because PHASE-LRU
has a lookahead of $M$, the total number of old blocks is at most $a_i + n_i$. Then we shall argue that due to the randomized layout, both the new blocks and the old blocks can be fetched with high parallelism.

We begin by counting the number of old blocks that PHASE-LRU needs to fetch in a phase. Consider the blocks in the buffer of PHASE-LRU after it services the last reference of $\text{phase}(i - 1)$. Let the number of ancient and new blocks of $\text{phase}(i)$ in the buffer at this time be $a$ and $n$, respectively. Consider evictions performed by PHASE-LRU in $\text{phase}(i - 1)$. PHASE-LRU evicts blocks that are not in the lookahead and uses LRU among the potential candidates. Hence all the $M$ blocks in the buffer at the end of $\text{phase}(i - 1)$ are either blocks belonging to $\text{phase}(i - 1)$ or blocks belonging to $\text{phase}(i)$. Now if PHASE-LRU evicts some block of $\text{phase}(i - 1)$ while servicing references from that phase, then for each such block that it evicted, it must have some block from $\text{phase}(i)$ in the buffer. Thus the number of blocks from $\text{phase}(i - 1)$ not present in the buffer at the end of $\text{phase}(i - 1)$ is $n + a \leq n_i + a$. Hence the maximum number of old blocks of $\text{phase}(i)$ referenced in $\text{phase}(i - 1)$ and not present in the buffer at the start of $\text{phase}(i)$ is at most $n + a \leq n_i + a$. This gives a bound on the total number of old blocks of $\text{phase}(i)$ that PHASE-LRU needs to fetch: $(a_i - a) + (a + n_i)$.

Now, if at the start of a phase there are a maximum of $b$ blocks from some disk referenced in that phase and not present in the buffer, then the number of read I/Os performed by PHASE-LRU in that phase is $b$. Now all the blocks in a phase are partitioned into new and old blocks. Hence the total number of I/Os done by PHASE-LRU in $\text{phase}(i)$ is upper bounded by the sum of the maximum number of new blocks on any single disk and the maximum number of old blocks on any single disk.

We can bound the number of I/Os needed by PHASE-LRU to fetch the remaining old blocks in any phase, by relating it to the classical occupancy problem [17]. Suppose that $m$ balls are randomly (uniform distribution) thrown into $n$ urns, what is the expected maximum number of balls in any urn? Let $C(m, n)$ denote the expected maximal occupancy when $m$ balls are thrown into
\( n \) urns. This leads to the result that during phase\((i)\), the expected value of the maximum number of old blocks that need to be fetched by PHASE-LRU from a single disk is \( C(a_i + n_i, D) = O((a_i + n_i) \log D/(D \log \log D)) \).

By the same token, the expected number of I/Os that PHASE-LRU performs to fetch new block of a phase is \( C(n_i, D) = O(n_i \log D/(D \log \log D)) \). \( \square \)

**Theorem 8** The expected competitive ratio of PHASE-LRU is \( \Theta(\log D/\log \log D) \).

**Proof:** In Lemma 14 we showed that the expected number of I/Os done by PHASE-LRU is at most \( T_{\text{PHASE-LRU}} = O(\sum_i (a_i + n_i) \log D/(D \log \log D)) \), where \( a_i \) and \( n_i \) are the number of ancient and new blocks, respectively, in phase\((i)\), and the sum is taken over all phases in the reference string. To complete the proof of the theorem we shall argue that the number of I/Os done by the optimal off-line algorithm is at least \( O(\sum_i (a_i + n_i) \log D/(D \log \log D)) \).

Let \( \text{OPT} \) denote the optimal off-line algorithm. Note that by any I/O schedule can be transformed into another schedule of the same or smaller length in which a block is never evicted before it has been referenced at least once since the last time it was fetched. Hence, we implicitly assume this property for \( \text{OPT} \).

First let us count the number of I/Os done by \( \text{OPT} \) in fetching ancient blocks of the reference string. Of the \( a_i \) ancient blocks that are accessed in phase\((i)\) let \( \text{OPT} \) cache \( x_i \) since their previous reference and fetch \( a_i - x_i \). By definition, an ancient block in phase\((i)\) is not referenced in phase\((i-1)\). Hence the \( x_i \) cached references will cause at least \( x_i/D \) I/Os during phase\((i)\). Fetching the \( a_i - x_i \) blocks incurs an additional \( (a_i - x_i)/D \) I/Os. Hence at least \( \sum_i a_i/D \) I/Os are performed by \( \text{OPT} \) due to ancient blocks.

Moreover, there are a total of at least \( \sum_i n_i \) (new) blocks accessed in the reference string. Hence, in fetching these blocks, \( \text{OPT} \) needs to do at least \( T_{\text{OPT}} \geq \sum_i n_i/D \) I/Os.

Thus the total number of I/Os done by \( \text{OPT} \) is at least \( O(\sum_i (a_i + n_i)/D) \). \( \square \)
4.4 Optimal read-often scheduling

4.4.1 Algorithm SUPERVISOR

In this section we introduce our parallel I/O scheduling algorithm, SUPERVISOR. Intuitively, SUPERVISOR is a 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: if the priority is low then an I/O for that block can be delayed.

Intuitively, SUPERVISOR is based on two ideas: (a) do not waste I/O slots if possible so that additional lookahead can be effectively utilized (b) issue prefetches to blocks close to their references so that prefetched blocks do not wastefully occupy buffer space.

Note that in each I/O there are $D$ slots available to fetch up to $D$ blocks, one from each disk. The blocks to be fetched in an I/O are decided based on the lookahead at that instance. As additional lookahead could be available after performing this I/O, we would like to avoid situations where we unnecessarily wasted I/O slots in prior I/Os. By performing I/Os greedily we attempt to handle such cases.

The I/O buffer is used to both cache blocks, since their previous reference, and to hold prefetched blocks, till they are referenced. Hence we would like to issue prefetches to blocks close to their reference so that they occupy buffer space for a smaller duration. In the same token, among possible candidates of blocks to cache, we would like to cache the block which would occupy buffer space for a smaller duration. This would mean that if at some time we would like to have a certain set of blocks in the buffer, then the block which would have been fetched in the last I/O would be the block whose previous reference would have been the farthest in the past. These choices in ordering blocks to be fetched are made by SUPERVISOR through its priority assignment scheme.

The algorithm SUPERVISOR is specified in Figure 4.3. The details of the pri-
ority assignment scheme are presented in Figure 4.4. The following definitions are used in specifying the algorithm.

- The current lookahead is denoted by $L = \langle r_i, \ldots, r_j \rangle$. The block accessed in reference $r$ is denoted by $\text{block}(r)$.

- The disk from which $x$ needs to be fetched is denoted by $\text{disk}(x)$: we shall use the same notation when $x$ is either a reference or a block, the meaning will be clear from the context.

- The priority of a reference or a block $x$ is denoted by $\text{priority}(x)$. \texttt{SUPERVISOR} will assign priorities to each reference in the lookahead. In the case when $x$ refers to a block, the priority of $x$ is the priority of the next reference to that block, or $-i$, where $i$ is the index of the last reference to that block (in the past) if there is no reference to it in the lookahead.

The greedy prefetching engine of \texttt{SUPERVISOR} is presented in Figure 4.3. \texttt{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 \texttt{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 \texttt{SUPERVISOR}, assigns priorities to each block in the lookahead. The details of the assignment routine are presented in Figure 4.4. The priority assignment scheme attempts to set the priority of every block low. But priorities need to be assigned carefully since the priorities are hints, to the greedy prefetching engine, regarding the order in which to fetch blocks. Constraints imposed by the system translate to two conditions on priority assignments. 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
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 in the current lookahead

and 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 for block \( b_i \) is then serviced

Figure 4.3: Algorithm SUPERVISOR

a unique priority. Note that 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 \( r \_r \) 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 which
the computation is waiting for.

The priority mechanism also indirectly hints at blocks to be cached by spec-
ifying a higher priority to blocks in the buffer. The routine examines subsets of
Routine to assign priorities to references

This routine is used to assign priorities to references \( r_j, \ldots, r_k \). 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 \( \phi \).

For \( i \) from \( k \) down to \( j \)

- Let previous\( (r_i) \) be the index of the previous reference to the same block as \( r_i \), or \(-i\) if that index is less than \( j \)
- If there is \( r \) in \( P[disk(r_i)] \) such that block\( (r) = block(r_i) \) then
  - Replace \( r \) in \( P[disk(r_i)] \) with \( r_i \) and key previous\( (r_i) \)
- Else
  - If numberOfBlocksPlaced = \( M \) then
    - For each disk \( d \) such that \( P[d] \) is not empty
      - Assign priority\( (r) \leftarrow \) lowestPriority, where
        - \( r \) is the reference with the smallest key in \( P[d] \)
      - Remove \( r \) from \( P[d] \); Decrement numberOfBlocksPlaced
      - Increment lowestPriority
    - Insert \( r_i \) into \( P[disk(r_i)] \) with key previous\( (r_i) \)
    - Increment numberOfBlocksPlaced

Figure 4.4: Routine to assign priorities

the lookahead consisting of \( M \) distinct references and then issues priorities to one block from each disk. The intuition behind the priority assignments is well illustrated by considering the largest subsequence of the lookahead including the last references and having at most \( M \) distinct references. All blocks which are assigned the smallest priority, 1, should belong to this set. Otherwise there will be some reference such that more than \( M \) blocks referenced after it have a lower, 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, among 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 among these blocks we would rather not cache that block, since a lower priority indicates that we prefer caching other blocks over this one.

Priorities are assigned starting from the lowest priority. A subset of the references in the lookahead consisting of $M$ distinct references is used to decide which blocks are assigned a particular priority level. Specifically, the variable `numberOfBlocksPlaced` keeps count of the number of blocks in the subset, triggering a priority assignment when it reaches $M$. The current lowest assignable priority is tracked by `lowestPriority`. Each entry in the array $\mathcal{P}$ is a list of distinct references for the corresponding disk. These references are ordered by the index of the previous reference to that block outside the subsequence. The block with the earliest previous reference is assigned the current lowest priority.

Algorithm `SUPERVISOR` also 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. Note that the priorities are updated only when additional lookahead is available.

### 4.4.2 Performance of `SUPERVISOR`

The first question that arises regarding `SUPERVISOR` is to show that it does in fact generate a schedule to service a reference string. In essence the system poses three conditions that any valid schedule must satisfy: (a) the reference string must be serviced in order, (b) at most one block can be fetched from a disk in an I/O, and (c) the number of cached blocks at any time is at most $M$.

From the specification of the greedy prefetching engine (Figure 4.3), it can
be seen that conditions (b) and (c) are satisfied by \texttt{SUPERVISOR}. Additionally when the required block is present in the buffer \texttt{SUPERVISOR} does not perform any I/O. Hence to show that the reference string can be scheduled in order, we will argue that when the demand block is not present in the buffer, it will be fetched in the next I/O. Equivalently, we will show that in such a case the demand block has the highest priority among all blocks from that disk, and it is among the \( M \) highest priority blocks.

We start by noting that when a reference is assigned a certain priority, all distinct references occurring after it have either been assigned a priority (greater than the current value of \texttt{lowestPriority}), or are counted in \texttt{numberOfBlocksPlaced}, which at this time is \( M \). As we discussed before, the priorities, being hints to the prefetching engine, have to have this property.

\textbf{Property 4} \textit{If a reference \( r \) has a priority \( p \) then the number of distinct references occurring after \( p \) that are assigned a priority smaller than or equal to \( p \) is at most \( M \).}

Let the demand reference by \( r \). Since \( r \) is the earliest reference in the lookahead, from the algorithm's specification, \( r \) will have the highest priority among all references from that disk. From Property 4 there are at most \( M \) blocks in the lookahead with a higher or same priority. Hence \( r \) will be fetched in the next I/O.

In this section we shall analyze the performance of \texttt{SUPERVISOR} in terms of the length of the schedule it generates. We shall show that when \texttt{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 \texttt{SUPERVISOR} has only a limited view of future requests we present bounds on the ratio of the number of I/Os done by \texttt{SUPERVISOR} to that of the optimal off-line.

In this work, we shall characterize the performance of online algorithms using the competitive ratio [21] 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 our analysis we consider the lookahead to be available in constant sized batches, which is available as a rolling window, being updated after every reference. 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 \( L = \langle r_1, \ldots, r_j \rangle \) then the number of distinct blocks in \( L \) is \( L \). This is a natural definition of lookahead and is similar to that of strong lookahead [3], which was used in the context of on-line caching algorithms for sequential systems.*

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 \( \Sigma \), let \( \text{OPT} \) denote the minimal length I/O schedule to service \( \Sigma \).

We shall first show that \text{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 \text{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 \text{SUPERVISOR} .

**Property 5** If \text{SUPERVISOR} is given the entire reference string apriori, the number of I/Os done by \text{SUPERVISOR} to service the reference string is given by the highest priority of any reference in the reference string.

---

* 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.
Equivalently, this property indicates that among the references that are in the lookahead but not in the buffer, the references with the highest priority are fetched in the next I/O. Consider a reference \( r \) which has the highest priority, \( p \), among all blocks but not in the lookahead. Now the reference \( r \) occurs after the demand reference, \( r_d \), and the priority of \( r \) is higher than or the same as the priority of \( r_d \). But we know from Property 4 that the number of distinct references that occur after \( r_d \) but which have a higher, or same, priority is at most \( M \). Hence the number of distinct references that occur after \( r \) which have a priority higher than or the same as \( p \) is at most \( M \). Hence \( r \) will be fetched in the subsequent I/O.

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 5 to show that the length of the schedule generated by \texttt{SUPERVISOR} is of the same length as \texttt{OPT}. We shall start by characterizing the blocks which have a priority \( p \). Let us recursively define a \textit{phase} as follows.

- Let \( \Sigma_i \) be \( \Sigma \). We define \( \text{phase}(1) \) to be the largest subsequence of \( \Sigma \), including the last reference in \( \Sigma \), such that the total number of distinct blocks in \( \text{phase}(1) \) is no more than \( M \).

- Let \( \text{earliest}(i) \) be the maximal sized subset of \( \text{phase}(i) \) such that if \( r \) is in \( \text{earliest}(i) \) then there is a reference \( t \) in \( \text{phase}(i) \) (a) \( \text{block}(r) = \text{block}(t) \), and for all other references \( s \) in \( \text{phase}(i) \), which are from the same disk, the previous reference to the block(s) is later than that of block(t). Let \( \Sigma_{i+1} \) be \( \Sigma_i - \text{earliest}(i) \). \( \text{phase}(i + 1) \), is now defined as the largest subsequence of \( \Sigma_{i+1} \), including the last reference in \( \Sigma_{i+1} \) such that the total number of distinct blocks in \( \text{phase}(i + 1) \) is no more than \( M \).

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

\textbf{Property 6} \textit{The priority of all references in \( \text{earliest}(i) \) is \( i \).}
From Property 6, 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 $T_{\text{OPT}}$.

The theorem is proved by inductively showing that OPT can be repeatedly transformed into a series of $T_{\text{OPT}}$ 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, $T_{\text{OPT}}$, 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 $T_{\text{OPT}}$, by Property 6, thereby completing the proof together with Property 5.

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 $\text{OPT}^*(k)$. The main idea used here is to compare the reference $a$ fetched by OPT and the corresponding reference from the same disk $b$ in earliest($k+1$). By definition of 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 replace d 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 the appendix handles the various technical details and cases possible in affecting this transformation.

**Theorem 9** Given a reference string $\Sigma$ of length $L$, SUPERVISOR, with $L$ block lookahead, performs the least number of I/Os to service $\Sigma$. 
**Proof:** We shall prove the theorem by inductively showing that OPT can be repeatedly transformed into a series of schedules \( \text{OPT}^*(k) \), where \( k \) originates from \( T_{\text{OPT}} \), \( \cdots, 1 \), 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 5 and 6.

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). Note that either \( p \) or \( q \) can be \( \phi \) if no such reference exists. But we will argue that \( q = \phi \) 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 we set \( \text{OPT}^*(1) \) to fetch nothing 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 \( \Sigma \).

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 $\Sigma$.

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 block($p$) and block($q$) referenced between the reference to $p$ and $q$: both $p$ and $q$ are in phase(1) as $q$ is in phase(1) by the definition of 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 $M - 1 - q$ is in phase(1).

Thus, $\text{OPT}$ 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 earliest(1); and the length of $\text{OPT}^*(1)$ is the same as that of $\text{OPT}$. 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 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 earliest($k + 1$).

Again, let $p$ be the reference fetched from some arbitrary disk $d$ by $\text{OPT}$ 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 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 $\Sigma_{k+1}$ 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 phase($k + 1$), and correspondingly in 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 \( \Sigma \).

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 \( \Sigma \).

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 \( \Sigma_{k+1} \) are fetched after the \( T_{\text{OPT}} - k + 1 \) th I/O. Moreover, since \( q \) is in phase\((k + 1)\), so is \( p \), indicating that the number of distinct blocks in \( \Sigma_{k+1} \) 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 \( \text{OPT} \)) 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 \( \Sigma_{k+1} \) referenced after \( q \) is at most \( M - 1 \).

\[ \square \]

Immediately, as a corollary, it can be seen that when the entire reference string is known to \text{SUPERVISOR} it generates the minimal length schedule to service it.
Corollary 2  **SUPERVISOR is the optimal off-line parallel I/O scheduling algorithm.**

We shall next provide bounds on the performance of SUPERVISOR in the online case, where it only has $L$-block lookahead. In the online scenario we shall use the competitive ratio as the measure of performance: in this context the competitive ratio of an online algorithm $A$ is the worst case ratio of the number of I/Os done by $A$ to service a reference string $\Sigma$ to the number of I/Os needed by the optimal off-line algorithm to schedule the same reference string $\Sigma$. We shall show the bounds separately in the two regions $L \leq M$ and $L > M$.

**Theorem 10**  The competitive ratio of SUPERVISOR is

$$O(L - M + D) \quad \text{when} \quad L \leq M$$

$$O(MD/L) \quad \text{when} \quad L > M$$

To do the analysis, let us 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 $\text{phase}(i)$. 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 $O(L/D)$ I/Os to service the lookahead, while the difference in the number of I/Os done by OPT and SUPERVISOR is at most $M$ in that lookahead. When the lookahead is less than $2M$ then we show that the ratio is at most $O(D)$ since the total number of blocks fetched by OPT and SUPERVISOR are comparable due to the high lookahead.

**Lemma 15** 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 phase ↑.

↑Sequential paging algorithms with this property have been previously referred to as marking algorithms [15].
Consider the I/Os done by SUPERVISOR in 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 16** 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 [15]; we briefly repeat it here for convenience. Let \(n_i\) clean blocks of phase\((i)\) be present in OPT's buffer at the start of phase\((i)\). Hence the number of blocks fetched by OPT in 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 phase\((i)\), OPT must have fetched at least \(n_{i+1}\) blocks in phase\((i)\). Hence the number of blocks fetched by OPT in 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 17** When \(L \leq M\) the competitive ratio of SUPERVISOR is \(O(M - L + D)\).

**Proof:** Consider the I/Os done by SUPERVISOR in phase\((i)\). From Lemma 15 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 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 \(\Sigma\), or \(\Omega(N)\); hence the competitive ratio of SUPERVISOR will be \(O(D + M - L)\).

From Lemma 16 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 $\Sigma$. 

\[ \square \]

**Lemma 18** When $L > M$ the competitive ratio of SUPERVISOR is $O(MD/L)$.

**Proof:** Let us divide the reference string $\Sigma$ into subsequences $\mathcal{L}_1, \ldots, \mathcal{L}_n$, 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 $L_i$.

Consider the optimal length schedule OPT to service $\Sigma$. 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 $L_i$. 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 $L_i$ 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 15, 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 16, 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 $L > M$. □
Chapter 5

Conclusion

To summarize, in this dissertation we addressed the problem of prefetching and caching in parallel I/O systems. We considered two classes of I/O workloads – read-once and read-often – based on the frequency of repeated accesses to the same data. With respect to prefetching and caching strategies for both classes of accesses, we identified fundamental bounds on performance of online algorithms, studied the performance of intuitive strategies, and then presented randomized and deterministic algorithms that provide higher performance guarantees. Our results show that current algorithms have significant limitations in exploiting latent I/O parallelism. We then designed new prefetching and caching algorithms that achieve high performance by optimizing buffer usage and concurrency in I/O accesses. These schemes include an optimal online scheduling algorithm for read-once I/O, an optimal off-line integrated prefetching and caching algorithm, and algorithms that exploit a randomized data layout to guarantee near full parallelism with minimal buffer and lookahead.

5.1 Directions for future research

This research indicates that there is substantial scope for improving the I/O performance by exploiting I/O parallelism, and provides a firm foundation for future research in this direction. This section presents some potential directions for future research; the four areas introduced next are based on extensions of the models presented here to tailor algorithms for specific applications.
5.1.1 Deadline driven scheduling

For applications handling media data, such as video-servers, it is not sufficient to just optimize the total I/O time. These applications need the I/O server to provide time guarantees in data delivery. One major area of further research is to incorporate constraints and requirements of real-time data delivery in the parallel I/O system. Initial work in this area has been presented in [14]. In a broad sense the problem is to provide time based service guarantees in the I/O system to support media applications.

To intuitively introduce the issues, consider a video server that serves video streams on demand to clients. Since the amount of data involved is large, video servers need to incorporate a large number of disks. The video streams are laid out across multiple disks for both performance and reliability reasons – this allows a higher bandwidth access and also redundancy. To allow efficient access, video is first compressed, usually variable-bit-rate, and then stored in constant sized blocks. Each block of data then needs to be delivered to the client by a specific deadline, at which the processing and playback of that block start. Because of compression and blocking, the rate of data accesses from the disks changes during the course of the delivery of the video. However there is substantial lookahead information available in the structure of a video stream that can be used to support prefetching. For instance, lookahead can be got by using stored information regarding the playback duration of each block of video data. This could be used to facilitate intelligent stream scheduling and caching both at the client and at the server.

Specifically the problem is to maximize the number of simultaneous clients that the video server can support. In [14] we developed a framework to design algorithms for admission control and resource scheduling in the context of such deadline-constrained applications. There we showed, for instance, using the traditional Earliest Deadline First (EDF) policy for prefetching in a video-server results in significant underutilization of system resources and consequently leads to fewer concurrent clients being supported. We designed a new
scheduling algorithm that maximizes the number of concurrent streams that a video server can support while guaranteeing that all data is delivered to the client on time. One direction of further research is to develop algorithms which can maximize the bandwidth under relaxed service guarantees such as, for instance, percentage of data blocks missing the deadline.

In an abstract sense, the reference string in this case has the additional characteristic that it has a time-deadline. There is no use in delivering data after its deadline has expired, and the client needs to cache it if the server pushes the data to the client prior to the block's deadline. Additionally, since there is already an element of time (due to playback) introduced, and the playback time is not related to the I/O time, we can easily introduce variability in I/O time in the model.

5.1.2 Incorporating system constraints

The parallel I/O model used in this work models a tightly coupled centralized I/O architecture — there is enough bandwidth to read from all the disks simultaneously, and the server buffer is fully shared among all the disks. The algorithms developed might need to be customized or modified to account for more specific I/O architectures. Two such situations that arise naturally are (a) bandwidth limitations and (b) a distributed I/O architecture.

In some cases there may not be enough bandwidth to read from all the disks in parallel; this poses a limitation such as “a maximum of $D'$ disks can be accessed simultaneously”. The question then is to decide which disks to perform I/Os on.

In a more distributed architecture, each disk has its own local buffer, which can buffer blocks from that disk only, and the server has a shared buffer that can buffer blocks from any disk. The question then is to decide which blocks to fetch and cache in which buffer. The problem becomes more interesting in applications like a video server, where the clients can have their own private buffer too.
5.1.3 Scheduling independent applications

This dissertation addressed the problem of scheduling a single application to minimize its I/O time: there was only one reference string to be scheduled. In a more general case, the I/O system could be servicing multiple applications simultaneously, such as, for instance, a web server which is retrieving blocks to service multiple connections.

The main abstraction in this case is to note that there is no total ordering among all the requests. Within each application the blocks need to be delivered to the clients in order, but there is no ordering constraint among the requests from multiple clients. Several problems arise naturally based on the criterion for optimization, such as total I/O time or fairness.

In a preliminary study we have looked at the problem of scheduling multiple distinct read-once reference strings, with the aim of minimizing the overall I/O time. Our initial results lead us to hypothesize that this problem is NP-Hard. Immediate open problems include formally proving that this problem is NP-Hard and to develop efficient approximation algorithms.

5.1.4 Extended lookahead models

Most of this work has concentrated on accurate prefetching and caching based on deterministic lookahead. In other situations, statistical information about future accesses could be available. Such information could be, for instance, gathered by making measurements on the system, provided by the programmer, or incorporated by the system administrator. Some of the algorithmic and implementation challenges in obtaining and using this kind of information include: how much and what kind of information is required, and what are the appropriate statistical models? How can prefetching and caching algorithms use this information to provide quality of service guarantees in a parallel I/O system?
Bibliography


