# Competitive Parallel Disk Prefetching and Buffer Management[†]


Rakesh Barve[‡]
rbarve@cs.duke.edu
Dept. of CS
Duke University
Durham NC 27708

Mahesh Kallahalla[§]
kalla@rice.edu
Dept. of ECE
Rice University
Houston TX 77251

Peter J. Varman[§]
pjv@rice.edu
Dept. of ECE
Rice University
Houston TX 77251

Jeffrey Scott Vitter[¶]
jsv@cs.duke.edu
Dept. of CS
Duke University
Durham NC 27708



**Abstract**

We provide a competitive analysis framework for online prefetching and buffer management algorithms in parallel I/O systems, using a read-once model of block references. This has widespread applicability to key I/O-bound applications such as external merging and concurrent playback of multiple video streams. Two realistic lookahead models, global lookahead and local lookahead, are defined. Algorithms NOM and GREED based on these two forms of lookahead are analyzed for shared buffer and distributed buffer configurations, both of which occur frequently in existing systems. An important aspect of our work is that we show how to implement both the models of lookahead in practice using the simple techniques of forecasting and flushing.

Given a $D$-disk parallel I/O system and a globally shared I/O buffer that can hold upto $M$ disk blocks, we derive a lower bound of $\Omega(\sqrt{D})$ on the competitive ratio of *any* deterministic online prefetching algorithm with $O(M)$ lookahead. NOM is shown to match the lower bound using global $M$-block lookahead. In contrast, using only local lookahead results in an $\Omega(D)$ competitive ratio. When the buffer is distributed into $D$ portions of $M/D$ blocks each, the algorithm GREED based on local lookahead is shown to be optimal, and NOM is within a constant factor of optimal. Thus we provide a theoretical basis for the intuition that global lookahead is more valuable for prefetching in the case of a shared buffer configuration whereas it is enough to provide local lookahead in case of the distributed configuration. Finally, we analyze the performance of these algorithms for reference strings generated by a uniformly-random stochastic process and we show that they achieve the minimal expected number of I/Os. These results also give bounds on the worst-case expected performance of algorithms which employ randomization in the data layout.



[†] A preliminary version of this paper has appeared in the Proceedings of the Fifth Annual Workshop on I/O in Parallel and Distributed Systems

[‡] Supported in part by an IBM graduate fellowship. Part of this work was done while the author was visiting Lucent Technologies, Bell Laboratories, Murray Hill, NJ.

[§] Supported in part by a grant from the Schlumberger Foundation and by the National Science Foundation under grant CCR–9704562.

[¶] Supported in part by the National Science Foundation under grant CCR–9522047 and by Army Research Office MURI grant DAAH04–96–1–0013. Part of this work was done while the author was visiting Lucent Technologies, Bell Laboratories, Murray Hill, NJ.

# 1 Introduction

The increasing imbalance between the speeds of processors and I/O devices has resulted in the I/O subsystem becoming a bottleneck in many applications. The use of multiple disks to build a parallel I/O subsystem has been advocated to enhance I/O performance and system availability [3], and most current high-performance systems incorporate some form of parallel I/O.

Prefetching is a powerful technique to reduce the I/O latency seen by an application. This is particularly true in a parallel I/O system where prefetching can be effectively used to obtain parallelism in disk access, so that the disks are most efficiently used. To fully exploit this potential, it is important to design and implement prefetching and buffer management algorithms that ensure that the most useful blocks are fetched and retained in the I/O buffer.

We consider a parallel I/O system consisting of $D$ independent disks that can be accessed in parallel [12]. The data for the computation is spread out among the disks in units of blocks. A block is the unit of access from a disk. As far as I/O is concerned, the computation is characterized by a reference string consisting of an ordered sequence of blocks that the computation accesses. In general, the reference string corresponding to a computation can consist of an arbitrary interleaving of reference strings of several concurrent applications. For the computation to successfully access a data-block, it should be resident in the internal memory of the computer system. By serving a reference string, we refer to the act of carrying out a series of I/O operations that make it possible for the computation to access blocks in the order specified by the reference string.

A recent study [6] focussed on the off-line problem of serving an arbitrary but fully known reference string of blocks spread across $D$ parallel, independent disks using parallel prefetching in conjunction with page replacement [1]. The authors presented and analyzed a very clever but somewhat complicated approximation algorithm for this problem. However, the practical issue of devising an online algorithm in the framework of competitive analysis [9] for the same problem was not addressed.

The performance of parallel versions of LRU and MIN [2] was analyzed in [11]. Modeling a distributed parallel I/O system, with independent disks and a partitioned I/O buffer, they defined a parallel version of MIN, and showed that it is optimal. The performance of online algorithms in a more tightly coupled system where the buffer can be shared by the different disks was not considered.

In this paper we present a competitive analysis framework for parallel prefetching algorithms on parallel disk systems for a restricted family of reference strings. In contrast to the requirement [6] of knowing a priori the entire reference string exactly, our parallel prefetching approach is based on models of bounded lookahead that are easily realizable in practice.

Our restricted family of reference strings are called *read-once* consumption sequences, in which all references are read-only and no block is read more than once. Such read-once reference strings arise very naturally and frequently in I/O-bound applications running on parallel disk systems: external merging and mergesorting (including carrying out several of these concurrently [13]) and real-time retrieval and playback of multiple streams of multimedia data, such as compressed video and audio.

Since no block is referenced more than once, it would seem that we only need to be able to fetch blocks in the order of their appearance in the reference string, in order to design an optimal prefetching algorithm. When the I/O buffer can hold $M$ blocks, a prefetching algorithm that is allowed a lookahead of $M$ blocks into the reference string would know, at each point, the next memory-load to fetch and can easily fetch blocks in the order of their appearance in the reference string.

Counter to intuition, in the parallel model the information provided by a lookahead of $M$ is

[1] Note that replacement decisions are necessitated by the fact that the I/O buffer can hold only some fixed number, say $M$, of pages.

insufficient to prefetch accurately. 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 to make the performance of these algorithms optimal.

As illustration, consider a system consisting of 3 disks with an I/O buffer of capacity 6. Assume that blocks labeled $A_i$ (respectively $B_i$, $C_i$) are placed on disk 1 (respectively 2, 3), and that the reference string is $A_1A_2A_3A_4B_1C_1A_5B_2C_2A_6B_3C_3A_7B_4C_4C_5C_6C_7$. Say that a parallel I/O is initiated only when the referenced block is not present in the buffer. The schedule in Figure 1 is one obtained by always fetching in the order of the reference string. At step 1, blocks $B_1$ and $C_1$ are prefetched along with the $A_1$. At step 2, $B_2$ and $C_2$ are prefetched along with $A_2$. At step 3, there is buffer space for just 1 additional block besides $A_3$, and the choice is between fetching $B_3$, $C_3$ or neither. Fetching in the order of $\Sigma$ means that we fetch $B_3$; continuing in this manner we obtain a schedule of length 9. In an alternative schedule, Figure 2, which does not always fetch in order, at step 2 disk 2 is idle (even though there is buffer space) and $C_2$ which occurs later than $B_2$ in $\Sigma$ is prefetched; similarly, at step 3, $C_3$ which occurs even later than $B_2$ is prefetched. However, the overall length of the schedule is 7, better than the schedule that fetched in the order of $\Sigma$.

| Disk 1 | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | | |
|---|---|---|---|---|---|---|---|---|---|
| Disk 2 | $B_1$ | $B_2$ | $B_3$ | | $B_4$ | | | | |
| Disk 3 | $C_1$ | $C_2$ | | | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |

Figure 1: Scheduling in order

| Disk 1 | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ |
|---|---|---|---|---|---|---|---|
| Disk 2 | $B_1$ | | | | $B_2$ | $B_3$ | $B_4$ |
| Disk 3 | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | $C_6$ | $C_7$ |

Figure 2: Scheduling out of order

It is unclear as to how I/Os ought to be scheduled on a parallel I/O system. The first step in this direction would be to know bounds on the achievable performance of scheduling policies knowing the next memory load of requests, and how these bounds may be achieved. We obtain the interesting result that there are read-once reference sequences such that *any parallel prefetching algorithm with a bounded lookahead of* $M$ incurs $\Omega(\sqrt{D})$ times as many parallel I/O operations as does the optimal off-line prefetching algorithm that knows the entire sequence. Using novel techniques, we go on to show that a simple prefetching algorithm called NOM that uses the bounded $M$-block lookahead to fetch blocks from a disk in the order of their appearance in the reference string never incurs more than $O(\sqrt{D})$ times the number of parallel I/O operations required by the optimal off-line prefetching algorithm. Thus, $\Theta(\sqrt{D})$ is a tight fundamental bound on the performance of bounded-lookahead parallel prefetching relative to optimal off-line parallel prefetching.

Motivated by the above results, in this paper we study online parallel prefetching algorithms for read-once sequences in several models varying in parallel disk configuration and the nature of lookahead available to the algorithm. Last but not least, we identify practical situations in which our models of lookahead are applicable and in fact, can be efficiently implemented using techniques such as forecasting and flushing [1].

Precise descriptions of I/O performance metrics, lookahead models, and parallel disk configurations are given in section 1.1. Our parallel prefetching algorithms NOM and GREED are described

in section 1.2. In section 2, we discuss practical situations in which lookahead may not be readily available. In section 3, we state and prove upper and lower bounds on *competitive ratios* for the shared buffer configuration for both forms of bounded lookahead. Section 4 gives similar results for the distributed buffer configuration. We consider the performance of our parallel disk prefetching and buffer management schemes in a probabilistic setting in section 5. In section 6 we describe how to implement the two forms of lookahead by using simple and practical techniques such as flushing and forecasting.

## 1.1 Model and Main Results

We consider the standard PDM (parallel disk model) consisting of $D$ parallel disks with an associated I/O buffer capable of holding $M$ blocks ($M \geq 2D$) [12], for parallel I/O performance. In each parallel I/O step, up to $D$ blocks, at most one from each disk, may be read concurrently into the buffer. Note that the parallel prefetching algorithm decides the disks from which blocks are to be prefetched, weighing the parallelism obtainable against the buffer space occupied by the blocks which are read. We measure the performance of a parallel prefetching algorithm on a reference string $\Sigma$ by counting the number of parallel I/O operations required to serve that reference string. Hence, we shall use the abbreviated term "I/O" to refer to a "parallel I/O step".

In the targeted applications (video servers and external merging), 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. 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 [5].

We consider only read-once reference strings in which each block appears exactly once. In order to enable prefetching we consider two natural models of bounded lookahead in this paper: *Global $M$-block* lookahead permits the prefetcher to know precisely the $M$ references in the reference string immediately following the last reference. In *local lookahead* only one block (the next reference missing in the buffer) from each disk is known to the prefetcher, beyond what is present in the buffer.

**Global $M$-Block Lookahead:** Let $\Sigma = r_1, r_2, \cdots r_N$, and suppose that the last block referenced is $r_i$. An I/O scheduling algorithm has *global $M$-block lookahead* if it knows the next $M$ blocks in $\Sigma$, $r_{i+1}, r_{i+2}, \cdots r_{i+M}$.

**Local Lookahead:** An I/O scheduling algorithm has *local lookahead* if it knows for each disk the next block in $\Sigma$ that is not in the buffer.

We consider two natural configurations of the parallel disk system, modeling commonly found I/O architectures. We refer to these as the *distributed buffer* configuration and the *shared buffer* configuration respectively, and are illustrated in Figure 3.

**Distributed Buffer:** In this configuration each disk has a local, private buffer of $M/D$ blocks. A disk's buffer is used exclusively for holding blocks read from that disk, and cannot be used to buffer blocks of other disks.

**Shared Buffer:** In this configuration there is a common buffer of $M$ blocks that is shared globally among all the disks.

For read-once sequences, we consider both a *worst-case model* wherein each block of the read-once sequence 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.

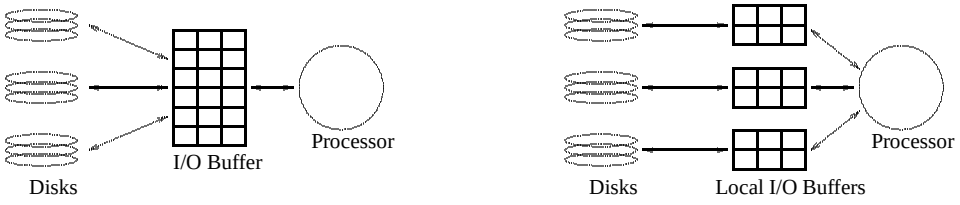


Figure 3: Two configurations of the I/O buffer

Below, we state our bounds on the I/O performance of the online parallel prefetching algorithms NOM and GREED that respectively employ global $M$-block lookahead and local lookahead on the two parallel disk configurations we mentioned earlier. We express the I/O performance of these online algorithms in terms of competitive ratios in the worst-case model.

**Definition 1** 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, $T_{\rm A}(\Sigma)$, that $A$ requires to serve $\Sigma$ is no more than $c_A T_{\rm OPT}(\Sigma) + b$, where $b$ is a constant and $T_{\rm OPT}(\Sigma)$ is the number of I/O operations required by an optimal off-line algorithm to serve $\Sigma$.

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

- In the worst-case model, the competitive ratio of parallel prefetching algorithms using only global $M$-block lookahead, running in the shared buffer configuration, is at least $\Omega(\sqrt{D})$. NOM has a competitive ratio of $\Theta(\sqrt{D})$ and is thus optimal among all algorithms using global $M$-block lookahead.

- In the worst-case model, the competitive ratio of algorithms using only local lookahead, running in the shared buffer configuration is at least $\Omega(D)$. GREED has a competitive ratio of $\Theta(D)$ and is thus optimal among all algorithms using local lookahead.

- In the worst-case model, GREED has a competitive ratio of $1$ for the distributed buffer configuration, and is hence optimal among all algorithms (online and off-line). On the other hand, NOM has a competitive ratio of a constant $c > 1$, and is hence near-optimal.

- For stochastically generated reference strings of length $N$, NOM incurs the *minimum* expected number of I/Os, namely $\Theta(N/D)$, in both the shared and distributed buffer configurations working with a buffer of size $M = \Omega(D \log D)$; whereas GREED requires a buffer of size $M = \Omega(D^2)$ and $M = \Omega(D \log D)$ respectively in the two configurations to achieve the same I/O performance.

## 1.2 Prefetching Algorithms

All the algorithms we consider generate a *valid schedule*; that is, in the resulting schedule a block must be present in the buffer before it is consumed and the number of blocks present in buffer must never exceed the buffer size. For the shared buffer this means that there are at most $M$ blocks in the buffer at any time; in the case of distributed buffer there are never more than $M/D$ buffered blocks from any disk. We say that a valid schedule is *normal* if each parallel I/O contains a *demand block*; that is, the block which is to be consumed next, thereby necessitating that I/O. Finally, the optimal

algorithm (OPT) generates an *optimal schedule* which minimizes the total number of parallel I/Os among all valid schedules. Note that the optimal algorithm may be an off-line algorithm.

We define scheduling algorithms NOM and GREED, that make use of $M$-block and local lookahead respectively. Both these algorithms do not evict a block once it has been fetched into the I/O buffer, till a request for that block has been serviced. Also, as these algorithms service read-once reference strings, once a request for a block has been serviced, the requested block is evicted from the buffer.

The performance of these algorithms are analyzed in section 3 for the shared buffer configuration, and in section 4 for the distributed buffer configuration.

**NOM:** algorithm uses global $M$-block lookahead to build a normal schedule as follows: on every parallel I/O it fetches a block from each disk that has an unread block in the current global $M$-block lookahead, provided there is space in the (local) buffer.

As the depth of lookahead used by NOM is $M$, or one memory-load, there will always be free buffer space for the unread blocks in the shared buffer configuration. However, in the distributed buffer configuration, some local buffers may be full, and no reads from the associated disks can occur.

**GREED** : algorithm uses local lookahead to build a normal 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 (local) buffer. In the distributed buffer configuration, if there is no buffer space in some local buffer then no block is read from that disk. In the shared buffer configuration, 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 NOM and GREED algorithms consider the reference string

$$\Sigma = A_1 A_2 A_3 A_4 B_1 B_2 B_3 B_4 D_1 D_2 C_1 C_2 B_5 B_6 A_5 A_6$$

The letter denotes the disk from which the block is requested and the subscript denotes the block index within the disk. Let $M = 8$.

The following is the schedule generated by NOM for the shared buffer configuration.

| | | | | | | |
|---|---|---|---|---|---|---|
| Disk A | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
| Disk B | $B_1$ | $B_2$ | $B_3$ | $B_4$ | $B_5$ | $B_6$ |
| Disk C | | | | $C_1$ | $C_2$ | |
| Disk D | | $D_1$ | $D_2$ | | | |

During the I/O for $A_1$ the lookahead window extends up to (and including) $B_4$. As this window does not include any blocks from disks $C$ and $D$, no blocks are prefetched from those disks. For the second I/O the lookahead window extends until $D_1$, causing it to be prefetched. Similarly, during the fourth I/O the lookahead window includes $C_1$, which is then prefetched. From the schedule above it can be seen that NOM requires a total of six I/Os.

For the same reference string the schedule generated by GREED is as follows.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Disk A | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | | $A_6$ | |
| Disk B | $B_1$ | $B_2$ | | | $B_3$ | $B_4$ | $B_5$ | $B_6$ |
| Disk C | $C_1$ | $C_2$ | | | | | | |
| Disk D | $D_1$ | $D_2$ | | | | | | |

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.

## 2 Practical Issues concerning Lookahead

In this section, we consider local lookahead in the context of two distinct types of parallel disk data layout strategies for applications such as external merging and video servers that generate read-once consumption sequences. It may be observed that both the above-mentioned applications involve sequentially retrieving data blocks from multiple streams laid out on disk. Fundamental difficulties [12, 8, 1] arise from the fact that (except in special circumstances) the different streams are "consumed" at varying, dynamically changing rates. Local lookahead can play a key role in implementing prefetching and buffer management in such circumstances.

Local lookahead refers to being able to tell, for each disk, at any point of time, which disk-resident block will be referenced the earliest. In the "run on a disk" scheme analyzed in [8], it is possible to obtain a direct implementation of local lookahead using simple prediction techniques [7, 1]. This can be achieved without requiring any information to be implanted in the data blocks, as in more sophisticated data layout schemes [1]. This is the case in certain existing database systems [8] or in video servers with each video clip stored entirely on a disk.

However, there are certain algorithmic advantages to having the streams striped across the $D$ disks during merging or merge sorting, as pointed out in [1]. Existing and proposed video servers generally either stripe video clips across disks in a round robin fashion or employ more sophisticated forms of striping [10]. The video server in [10] uses independently chosen *random permutations* as orderings of the $D$ disks in which to place successive groups of $D$ contiguous blocks of a clip. (Such randomized striping helps prevent extended durations of time in which an I/O hot spot moves from one disk to the next in cyclic order because disk blocks from several video clips have active portions co-located on the same disk, getting consumed at uniform rates. The random permutation ensures that the hot spot does not move in synchrony from one disk to the next and so on.) In these situations, local lookahead does not come for free and involves picking out, for each disk, one block from the set of next blocks of all the streams on that disk. It is in these circumstances that the forecasting data structure [1] can be fruitfully employed to implement local lookahead with negligible preprocessing overhead, as we discuss in section 6.2.

## 3 Shared Buffer Configuration

In the shared buffer configuration a globally shared buffer is used to cache blocks fetched in an I/O. Since the buffer is shared by all disks, there is no specific portion of the buffer allocated to any particular disk as in the distributed buffer configuration. Hence it is possible to allocate buffer space unevenly to different disks. This allows the initiation of prefetches even on disks from which already a lot of blocks have been prefetched and buffered, which is not possible in the distributed buffer configuration.

This choice in allocating buffer space to different disks makes prefetching and buffer management difficult and challenging. The buffer management algorithm has to judiciously allocate buffer space among blocks fetched from different disks. In order to service the reference string with the least number of I/Os, the number of disks busy during each I/O ought to be maximized. 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 choking the buffer and reducing the parallelism in fetching more immediate blocks. But, counter to intuition, it is *not* always better to prefer fetching a block just because that block is required earlier than another. Such situations are presented and used to give a lower bound on the performance of algorithms using global $M$-block lookahead in section 3.1. Hence a good prefetching and buffer management algorithm ought to co-operatively decide how much buffer space to allocate for a particular I/O and which blocks ought to be (pre)fetched in a particular I/O, so that the entire reference string can be serviced in the least number of I/Os.

In this section we study the on-line version of the above problem, wherein the entire reference

string is not available to the algorithm. Instead the algorithm is allowed only the limited knowledge, of future requests, given by references in the lookahead.

## 3.1 Global $M$-block Lookahead

We first study the performance of algorithms which, at any time, have knowledge of the next memory-load of future accesses, or the next $M$ references. This is interesting since at any instant the buffer can hold at most $M$ blocks; hence algorithms which have such knowledge might intuitively keep the buffer filled with *immediately* important blocks and thereby be expected to perform very well. This is true in the limited sense that global $M$-block lookahead is more useful than local lookahead. Global $M$-block lookahead gives information regarding the relative order of reference of blocks from disks, which can be effectively used to perform prefetches cleverly.

However, surprisingly, we shall show that *any* algorithm that uses only global $M$-block lookahead is fundamentally limited to have a competitive ratio of at least $\Omega(\sqrt{D})$. This non-intuitive bound is primarily due to the fact that in the shared buffer configuration, knowledge of the reference string beyond the next $M$ blocks, can be used to perform more effective prefetching.

### 3.1.1 Lower Bound

Given any online parallel prefetching algorithm that employs $M$-block lookahead, we show how to construct a *nemesis* reference string $\Sigma$, that forces the online algorithm to perform $\Omega(\sqrt{D})$ times the number of I/Os incurred by the optimal off-line algorithm OPT on $\Sigma$.

As discussed before, global $M$-block lookahead provides information regarding the next memory-load of data. Hence, we consider the performance of these algorithms in a sequence of block references, each of length $M$. This intuition is naturally captured by the concept of *phase*.

**Definition 2** Consider a read-once reference string $\Sigma$ that consists of references $r_0, r_1, r_2, r_3, \ldots$. The string $\Sigma$ is said to consist of a sequence of phases $\Sigma = phase(0), phase(1), \ldots$, where $phase(i)$ consists of the sequence $\langle r_k \rangle$, $iM \leq k < (i+1)M$, of references, for $i \geq 0$.

By the above definition, when the first block of a phase is referenced, an algorithm with $M$-block lookahead knows all the blocks (and their order of reference) in that phase. As the computation proceeds, the lookahead window includes blocks from the subsequent phase as well. Hence, in general the lookahead window can span more than one (at most two) phase.

Let $c(i,d)$ denote the number of blocks from disk $d$ that are referenced in $phase(i)$. Note that $c(i,d)$ depends only upon the reference string and is independent of the scheduling algorithm.

**Definition 3** Consider any parallel prefetching algorithm $\mathcal{A}$. Let $p_{\mathcal{A}}(i,d)$ be the number of prefetched blocks from disk $d$ in the buffer at the start of $phase(i)$. Define the *dominant peak* in $phase(i)$ as $dom_{\mathcal{A}}(i) = \max_d \{c(i,d) - p_{\mathcal{A}}(i,d)\}$. The minimum number of I/Os that $\mathcal{A}$ needs to make in $phase(i)$ is given by $dom_{\mathcal{A}}(i)$.

We will use the following notation during our analysis. Let $T_{\mathcal{A}}$ be the total number of I/Os used by $\mathcal{A}$ to service $\Sigma$ and let $T_{\mathrm{OPT}}$ be the number of I/Os taken by OPT to service $\Sigma$. We use $\mathcal{D}$ to denote the set of the $D$ parallel disks.

In order to facilitate the presentation of our lower bound proof we define *good* and *bad* phases.

**Definition 4** A phase, $phase(i)$, is called a *good* phase if the constituent $M$ blocks, $\langle r_k \rangle$, $iM \leq k < (i+1)M$, are striped in a round-robin manner across the set $\mathcal{D}$ of $D$ disks.

A *bad* phase, $phase(i)$, with *bad disk* parameter $d_i$ consists of blocks $\langle r_k \rangle$, $iM \leq k < (i+1)M$, laid out such that the first $M - M/2\sqrt{D}$ blocks $\langle r_k \rangle$, where $iM \leq k < (i+1)M - M/2\sqrt{D}$, are striped in a round-robin manner across the set $\mathcal{D} - \{d_i\}$ of $D-1$ disks and the remaining $M/2\sqrt{D}$ blocks $\langle r_k \rangle$, where $(i+1)M - M/2\sqrt{D} \leq k < (i+1)M$, all originate from the *bad* disk $d_i$.

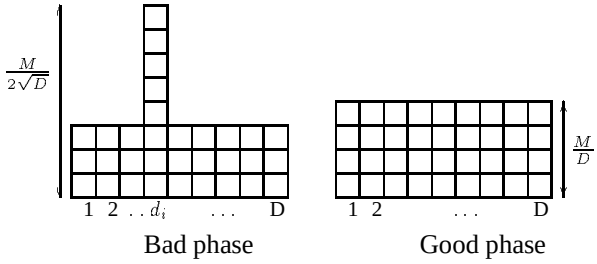


Figure 4: Illustration of Bad and Good phases

Figure 4 illustrates the distribution of blocks on different disks in the two kinds of phases.

Note that if no block from a bad phase were to be prefetched prior to the beginning of the phase, at least $M/2\sqrt{D}$ I/Os need to be performed to serve the requests in that phase. We will force any online algorithm to get into a situation where its limited lookahead prevents it from prefetching a substantial number of blocks for the next bad phase.

The blocks referenced in good phases are striped across all $D$ disks. This guarantees that all the requests can be serviced with exactly $M/D$ fully parallel I/Os, provided that the number of free blocks in the buffer at the start of the phase is at least $D$.

Given any deterministic online algorithm $\mathcal{A}$ with a bounded lookahead of $M$ blocks, we show below how to construct a nemesis reference string from good and bad phases, depending on $\mathcal{A}$'s prefetching decisions.

**Definition 5** We construct a reference string $\eta$ of $2\sqrt{D}M$ references such that the nemesis string $\Sigma$ is obtained by repeating the string $\eta$ an arbitrary number of times. The reference string $\eta$ consists of a sequence of $2\sqrt{D}$ phases, $phase(1), \ldots, phase(2\sqrt{D})$, such that odd-numbered phases $phase(2k-1)$, with $1 \leq k \leq \sqrt{D}$, are bad and even-numbered phases $phase(2k)$, with $1 \leq k \leq \sqrt{D}$, are good.

The first bad phase, $phase(1)$, has a bad disk parameter of 1. The bad disk parameter of every subsequent bad phase is dependent on $\mathcal{A}$'s prefetching decisions and is chosen as follows: For $k > 1$, let $B_k$ denote the set of bad disk parameters corresponding to all bad phases $phase(2k'-1)$ with $k' < k$, occurring prior to $phase(2k-1)$. Let $G_k$ denote the set $\mathcal{D} - B_k$ of $D - k + 1$ disks not in $B_k$. It is possible that on account of $\mathcal{A}$'s prefetching, one or more future disk blocks[2] are already in the I/O buffer at the end of $phase(2k-3)$. The disk $d_k \in G_k$ such that among all the disks in the set $G_k$, $d_k$ has the smallest number of future blocks in algorithm $\mathcal{A}$'s buffer at the end of $phase(2k-3)$, is chosen to be the bad disk parameter of $phase(2k-1)$.[3]

**Theorem 1** *The competitive ratio of any deterministic online algorithm having bounded global $M$-block lookahead is at least $\Omega(\sqrt{D})$.*

**Proof :** We shall show that the reference string $\Sigma$ defined above is such that $T_{\mathcal{A}}/T_{\mathrm{OPT}}$ is $\Omega(\sqrt{D})$. In Lemma 1 we show that $\mathcal{A}$ will incur $\Omega(M)$ I/Os for every instance of the substring $\eta$, defined above, in the nemesis string $\Sigma$. On the other hand we show in Lemma 2 that there exists a normal schedule $\mathcal{S}$ that incurs only $\Theta(M/\sqrt{D})$ I/Os corresponding to every instance of the substring $\eta$. Hence the theorem follows. □

Intuitively, the subsequence $\eta$ is constructed by alternating bad phases with good phases. Bad phases are constructed to have a large number ($M/2\sqrt{D}$) blocks requested from a single disk, and

[2] By future disk blocks we mean blocks that get referenced some time in the future with respect to the present point in time.

[3] This is a valid construction as $\mathcal{A}$ can see only $M$ blocks ahead in the reference string and so cannot make any prefetching decisions depending on $phase(2k-1)$ prior to the end of $phase(2k-3)$.

the rest of the blocks striped across all the remaining disks. Hence these phases can cause a large number of I/Os if no blocks are prefetched from the disk which has the "peak". Good phases are designed to hide the skewed disk block distribution of bad phases from the bounded lookahead algorithm, while not permitting "free" prefetching opportunities as the next bad phase is discovered.

It may be noted that the reference string $\eta$'s disk blocks are distributed so as to classify the $D$ disks into two sets $\mathcal{D}_1$ and $\mathcal{D}_2$ of sizes $\sqrt{D}$ and $D - \sqrt{D}$ respectively: each disk of $\mathcal{D}_1$ has exactly $\mathcal{K} + \Delta$ blocks of $\eta$ originating from itself, while each disk of $\mathcal{D}_2$ has exactly $\mathcal{K}$ blocks originating from itself; where $\mathcal{K} = O(M/\sqrt{D})$ and $\Delta = M/2\sqrt{D} - (M - \frac{M}{2\sqrt{D}})/(D-1)$ additional blocks are requested from a bad disk in a bad phase.

We force the online algorithm $\mathcal{A}$ to incur approximately $\Delta$ I/Os for every consecutive pair of phases of $\eta$ thus resulting in a cost of $\Omega(M)$ for $\mathcal{A}$. On the other hand, we show that it is possible to design an optimal off-line schedule $\mathcal{S}$ that fetches $\Delta$ future blocks from $\sqrt{D} - 1$ disks of the set $\mathcal{D}_1$ in the first phase itself thereby leaving an evenly balanced disk block placement for subsequent phases. We show that $\mathcal{S}$ incurs no more than $O(M/\sqrt{D})$ I/Os in doing so.

The following lemmas formalize the above intuition.

**Lemma 1** *Algorithm $\mathcal{A}$ incurs at least $T_{\mathcal{A}}$ I/Os to service $\eta$, where $T_{\mathcal{A}} = \Omega(M)$.*

**Proof :** For $1 \leq k \leq \sqrt{D} - 1$, consider the $k$th bad phase, $phase(2k-1)$, in $\eta$. Let the next bad phase, $phase(2k+1)$, have bad disk parameter $d_{k+1}$. By construction, $d_{k+1}$ is chosen such that it has not been the parameter for any previous phase, and to which the least number of blocks have been prefetched by $\mathcal{A}$. Since at most $M$ prefetched blocks can be in the buffer at the end of $phase(2k-1)$, the number of prefetched blocks from disk $d_{k+1}$ in the buffer at the end of $phase(2k-1)$ is at most $M/(D-k)$. During $phase(2k)$, one I/O is required for each block of $phase(2k+1)$ that $\mathcal{A}$ chooses to prefetch from disk $d_{k+1}$. Hence if during $phase(2k)$, $\mathcal{A}$ prefetches $n_k$ blocks from disk $d_{k+1}$ for $phase(2k+1)$, it must perform at least $n_k$ I/Os in $phase(2k)$. The total number of blocks from disk $d_{k+1}$ that could have been prefetched at the start of $phase(2k+1)$ is no more than $M/(D-k) + n_k$, and so the total number of I/Os done by $\mathcal{A}$ in $phase(2k+1)$ is at least $M/2\sqrt{D} - M/(D-k) - n_k$. The total number of I/Os done during $phase(2k)$ and $phase(2k+1)$ combined is therefore at least $M/2\sqrt{D} - M/(D-k)$. Hence the number of I/Os done by $\mathcal{A}$ to service $\eta$ is

$$T_{\mathcal{A}} \geq \frac{M}{2\sqrt{D}} + \sum_{1 \leq k \leq \sqrt{D}-1} \left( \frac{M}{2\sqrt{D}} - \frac{M}{D-k} \right)$$

Hence $T_{\mathcal{A}} = \Omega(M)$. □

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 bad phase, the off-line schedule prefetches blocks from bad disks of all future bad phases thus reducing the number of I/Os that need to be performed in future bad phases to $O(M/D)$. It exploits the fact that good phases can be serviced with full parallelism (needing $M/D$ I/Os) with just a small amount of storage ($D$). By prefetching into only $M/2$ memory blocks, the schedule leaves $M/2 \geq D$ blocks free to get full parallelism in the good phases. Hence no blocks need to be prefetched for the good phases.

**Lemma 2** *A normal schedule $\mathcal{S}$ can be constructed, that incurs at most $T_{\mathrm{S}} = \Theta(M/\sqrt{D})$ I/Os to service $\eta$.*

**Proof :** We construct a schedule $\mathcal{S}$ to service $\Sigma$ by running the following algorithm on it. As before, let $\eta$ have bad disk parameter $d_k$, corresponding to $phase(2k-1)$, for $1 \leq k \leq \sqrt{D}$ and let $\tau = (M - \frac{M}{2\sqrt{D}})/(D-1)$.

- In $phase(1)$ of $\eta$, we prefetch as follows:

– During the first $\tau$ I/Os we fetch only blocks required in the same phase.
– During the remaining $M/2\sqrt{D} - \tau$ I/Os of $phase(1)$, we prefetch $M/2\sqrt{D} - \tau$ blocks from each one of the disks $d_k$, for $2 \leq k \leq \sqrt{D}$, that constitute the bad disk parameters of the remaining $\sqrt{D} - 1$ bad phases. The $M/2\sqrt{D} - \tau$ blocks prefetched from disk $d_k$, for $2 \leq k \leq \sqrt{D}$, are precisely *the farthest*[4] $M/2\sqrt{D} - \tau$ *disk blocks of* $phase(2k-1)$.

- During each subsequent phase, we prefetch blocks of that phase with full disk parallelism. Due to the prefetching carried out in the first phase, no bad phase now has more than $\tau$ blocks residing on any disk so even bad phases incur only $\Theta(M/D)$ I/Os. Since the I/O buffer can hold $M \geq 2D$ blocks, disk blocks prefetched in the first phase do not have to be evicted or flushed out to make space for subsequent processing.

Since $\mathcal{S}$ has prefetched $M/2\sqrt{D} - \tau$ blocks from each of the disks $d_k$, for $2 \leq k \leq \sqrt{D}$, the dominant peak in each of the $\sqrt{D} - 1$ bad phases following $phase(1)$ will be reduced to $\tau$. Hence in each of these bad phases, $\mathcal{S}$ will incur $\tau$ I/Os.

As discussed previously, any good phase can be serviced in exactly $M/D$ I/Os provided there are $D$ free blocks in the buffer. This is satisfied by the schedule. Hence, in any good phase $\mathcal{S}$ will incur exactly $M/D$ I/Os to fetch all blocks that are referenced in the same phase itself. Therefore, in servicing $\eta$, the total number of I/Os done by $\mathcal{S}$ is

$$T_{\mathrm{S}} = \frac{M}{2\sqrt{D}} + \frac{M}{D} \times \sqrt{D} + \tau \times (\sqrt{D} - 1)$$

And hence, $T_{\mathrm{S}} = \Theta(M/\sqrt{D})$. □

### 3.1.2 Upper bound on the Competitive Ratio

From theorem 1, the competitive ratio of any online algorithm using global $M$-block lookahead is $\Omega(\sqrt{D})$. This raises the question as to whether we can design an algorithm which can match this bound. We shall show in this section that a simple algorithm NOM can match the lower bound up to constant factors.

In this section we prove an upper bound on the ratio of the number of I/Os required by NOM to the number of I/Os required by the optimal off-line algorithm in the shared buffer configuration. The following lemma ensures that, while considering optimal algorithms that service read-once reference strings, it suffices to consider simple off-line prefetching algorithms that never evict prefetched blocks before they are referenced. We omit the simple proof here: we can cancel I/Os for blocks which are evicted before the block is referenced with no increase in the number of parallel I/Os.

**Lemma 3** *For every I/O schedule servicing a read-once reference string, there exists a schedule that performs the same or fewer number of I/Os and never evicts a prefetched block before it is referenced.*

Consider a reference string $\Sigma$ and a parallel prefetching algorithm $\mathcal{A}$ serving it. Any phase of $\Sigma$ is said to have completed at the time of consumption of the final block referenced in that phase.

The sequence of consecutive I/Os made by $\mathcal{A}$ between the I/O immediately following the completion of $phase(i-1)$ up to but not including the I/O immediately following the completion of $phase(i)$, is refered to as *the I/Os incurred by* $\mathcal{A}$ *in* $phase(i)$.

Consider the set $P_{l,h}$ of blocks of $phase(l)$ that are yet to be read after completion of the $h$-th parallel I/O done by OPT. We denote by $\Delta(l,h)$ the largest number of blocks of the set $P_{l,h}$ that need to be read from any single disk, taking all disks into consideration.

[4]The $i$ farthest blocks of a phase are the $i$ blocks belonging to that phase which get consumed farthest in the future, relative to all blocks of that phase.

We can now present the notion of a *useful block* that plays a key role in our analysis.

**Definition 6** Consider the $j$-th parallel I/O, $R_j$, of OPT and let it be incurred in $phase(i)$. It is possible that read $R_j$ prefetches blocks belonging to phases occurring after $phase(i)$. We say that a block $b$ referenced in $phase(k)$, where $k > i$, prefetched by read $R_j$ is a *useful block prefetched in* $phase(i)$ *for* $phase(k)$ if the following conditions hold : (a) $\Delta(k, j) = \Delta(k, j-1) - 1$; (b) among all blocks of $phase(k)$ prefetched by read $R_j$ block $b$ is the (unique) block to be referenced *farthest in the future* .

We next introduce the notion of a *superphase* with respect to the actions of the optimal prefetching algorithm OPT while servicing a reference string. Given a reference string $\Sigma$, we break it down into contiguous subsequences called *superphases*.

**Definition 7** The $i$th superphase, denoted by $\mathcal{S}_i$, $i \geq 0$, is defined to be the subsequence $\langle phase(t), phase(t+1), \ldots, phase(t+s)\rangle$ such that the following conditions are satisfied : (a) $phase(t-1)$ belongs in $\mathcal{S}_{i-1}$ if $i > 0$ else $phase(t) = phase(0)$ if $i = 0$ (b) the number of useful blocks prefetched in $\mathcal{S}_i$ is at least $M$ and (c) the number of useful blocks prefetched in phases $\langle phase(t), phase(t+1), \ldots, phase(t+s-1)\rangle$ is less than $M$.

In its essence, a superphase is a collection of a minimal number of contiguous phases in which at least $M$ useful blocks are prefetched.

Consider the $i$-th superphase $\mathcal{S}_i$ of the reference string. Let $\Gamma_i$ denote the set of phases of $\mathcal{S}_i$ such that for each $phase(j) \in \Gamma_i$, at least one useful block has been prefetched in some previous $phase(k)$, where $k < j$. Let $\Pi_i$ denote the set of all remaining phases of $\mathcal{S}_i$: that is, phases to which no useful block has been prefetched.

Let $\gamma_i = |\Gamma_i|$. Let the number of useful blocks prefetched by OPT in superphases before $\mathcal{S}_i$ for phases of $\mathcal{S}_i$ be $\alpha_i$ and the number of useful blocks prefetched by OPT in $\mathcal{S}_i$ for phases in $\mathcal{S}_i$ be $\beta_i$. Let $I_{\mathrm{NOM}}$ and $I_{\mathrm{OPT}}$ be the number of I/Os done in superphase $\mathcal{S}_i$ by NOM and OPT respectively.

The following lemmas follow directly from previous definitions.

**Lemma 4** *For a particular phase $phase(k)$, no two useful blocks can be prefetched in the same I/O operation.*

**Lemma 5** *NOM does not incur any more I/Os than OPT in phases belonging to $\Pi_i$.*

**Lemma 6** *NOM incurs at most $m$ more I/Os than OPT in a phase belonging to $\Gamma_i$, where $m$ is the number of useful blocks prefetched by OPT in previous phases for that phase.*

**Lemma 7** *The number of useful blocks prefetched by OPT in $\mathcal{S}_i$ for phases in $\mathcal{S}_i$ is $\beta_i < M$.*

**Proof :** Let $\mathcal{S}_i = \langle phase(t), phase(t+1), \ldots, phase(t+s)\rangle$. The lemma follows from the fact that no useful block prefetched in $phase(t+s)$ can be for for phases in $\mathcal{S}_i$. But by definition, the number of useful blocks fetched in the remaining phases of $\mathcal{S}_i$ is at most $M$. □

The following key lemma ensures that essentially it is enough for us to show that OPT incurs $\Omega(M/\sqrt{D})$ I/Os in a superphase.

**Lemma 8** $I_{\mathrm{NOM}} < I_{\mathrm{OPT}} + 2M$.

**Proof :** By definition, $\alpha_i \leq M$, and from Lemma 7, $\beta_i < M$. Applying Lemma 5 and Lemma 6 completes the proof. □

We shall now prove Theorem 2 considering the following mutually exclusive cases.

**Lemma 9** *If $\gamma_i \geq \sqrt{D}$, then $I_{\mathrm{NOM}}$ / $I_{\mathrm{OPT}}$ is $O(\sqrt{D})$.*

**Proof :** The total number of blocks referenced in $\mathcal{S}_i$ is at least $M\gamma_i$, since each phase references $M$ blocks. Since at most $M$ blocks of these blocks could have been prefetched before the start of $\mathcal{S}_i$, at least $M\gamma_i - M$ blocks must be fetched in $\mathcal{S}_i$. This would require at least $(M\gamma_i - M)/D$ I/Os. Since $\gamma_i \geq \sqrt{D}$, $I_{\rm OPT} \geq M/\sqrt{D} - M/D$. Applying Lemma 8, we have $I_{\rm NOM}/I_{\rm OPT} = O(\sqrt{D})$. □

**Lemma 10** *If $\gamma_i < \sqrt{D}$ and $\beta_i \geq M/2$ then $I_{\rm NOM}/I_{\rm OPT}$ is $O(\sqrt{D})$.*

**Proof :** By the definition of $\beta_i$ and $\gamma_i$, there must be some phase in $\mathcal{S}_i$ such that at least $\beta_i/\gamma_i$ useful blocks were prefetched by OPT for that phase in previous phases of $\mathcal{S}_i$. It follows from Lemma 4, that at least $\beta_i/\gamma_i$ I/Os must have been incurred by OPT in superphase $\mathcal{S}_i$. Hence $I_{\rm OPT} \geq \beta_i/\gamma_i > (M/2)/\sqrt{D} = M/2\sqrt{D}$. Applying Lemma 8, we have $I_{\rm NOM}/I_{\rm OPT} = O(\sqrt{D})$. □

**Lemma 11** *Let $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ be two consecutive non-overlapping superphases. If $\gamma_i < \sqrt{D}$ and $\beta_i < M/2$, then the ratio of the sum of the number of I/Os in $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$ by NOM and OPT is $O(\sqrt{D})$.*

**Proof :** If $\gamma_{i+1} \geq \sqrt{D}$ or if $\beta_{i+1} \geq M/2$, then the amortized ratio over the two superphases is at most $O(\sqrt{D})$ by an analysis similar to that of Lemma 9 and Lemma 10.

The interesting case is when $\gamma_{i+1} < \sqrt{D}$ and $\beta_{i+1} < M/2$. In this case, at least $M/2$ useful blocks prefetched during $\mathcal{S}_i$ lie in the buffer of OPT at the end of superphase $\mathcal{S}_i$, since at most $M/2$ useful blocks prefetched in $\mathcal{S}_i$ were for phases in $\mathcal{S}_i$ ($\beta_i < M/2$). Now in $\mathcal{S}_{i+1}$, OPT prefetches at least $M$ additional useful blocks. Since the I/O buffer can hold at most $M$ blocks, at least $M/2$ of all the useful blocks that were prefetched in either $\mathcal{S}_i$ or $\mathcal{S}_{i+1}$ must necessarily be for phases of $\mathcal{S}_{i+1}$. But the number of phases for which useful blocks were fetched in $\mathcal{S}_{i+1}$ is $\gamma_{i+1} \leq \sqrt{D}$. Consequently there must be some phase in $\mathcal{S}_{i+1}$ for whom at least $M/2\sqrt{D}$ useful blocks were prefetched. Hence, invoking Lemma 4, OPT incurred at least $M/2\sqrt{D}$ I/Os during phases $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$. Now, applying Lemma 8, we again have $I_{\rm NOM}/I_{\rm OPT} = O(\sqrt{D})$ considering the two superphases together. □

**Theorem 2** *The competitive ratio of NOM is $O(\sqrt{D})$ and hence it is optimal among all algorithms using only global $M$-block lookahead.*

**Proof :** Partition $\Sigma$ into non-overlapping superphases as described previously. Lemmas 9, 10 and 11 show that either the ratio of the number of I/Os done by NOM to those done by OPT in a single superphase is $O(\sqrt{D})$ (Lemmas 9, 10), or that this bound is satisfied by the I/Os done in two consecutive superphases (Lemma 11). Hence the competitive ratio of NOM is $O(\sqrt{D})$. □

## 3.2 Local Lookahead

In this section, we consider the benefits of using pure local lookahead: that is, the prefetching algorithm has no access to any information regarding the relative order of consumption of blocks originating from different disks. It turns out that this is a very powerful advantage for the adversary in the shared buffer configuration. The adversary can force a higher lower bound on the competitive ratio of online algorithms based only upon local lookahead relative to that for online algorithms that can use global $M$-block lookahead.

In Theorem 3 below, we show for the shared buffer configuration that *any* algorithm using only local lookahead can perform $\Omega(D)$ times as bad as the optimal off-line algorithm. Note that this

is the worst possible competitive ratio for any algorithm which generates a normal schedule. This is because any algorithm which generates a normal schedule, initiates I/Os only on demand and hence performs at most one I/O per block in the reference string. Hence, clearly, if the length of the reference string is $N$, the most 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, a simple algorithm like GREED can easily match the bound.

The proof of the lower bound is similar to that of Theorem 1; that is, we construct a reference string that can fool any given algorithm $\mathcal{A}$ that uses only local lookahead into performing a large number of I/Os.

**Theorem 3** *Any algorithm using only local lookahead, in the shared buffer configuration, has a competitive ratio of at least* $\Omega(D)$.

**Proof :** Let $\mathcal{A}$ denote an arbitrary algorithm using only local lookahead. We shall prove the theorem by constructing a reference sequence $\eta = \langle r_i \rangle, 1 \leq i \leq 3M$, to service which $\mathcal{A}$ makes $\Omega(M)$ I/Os. We shall also give a schedule which serves $\eta$ with $\Theta(M/D)$ I/Os. A reference string of arbitrary length can be obtained by repeating $\eta$ as required.

Sequence $\eta$ is constructed depending upon the behavior of $\mathcal{A}$ till the previous I/O. This is a valid construction of the reference string as $\mathcal{A}$ has no knowledge of the relative order of reference of blocks across disks. By definition, local lookahead allows the algorithm knowledge of the order of reference from any *single* disk. Let $\alpha = 3M/D$. We construct $\eta$ as follows:

1. The first $3M/D$ blocks of $\eta$ are requested from disk 1.
2. Divide the next $M - 3M/D$ references into $D/3 - 1$ sets of $3M/D$ blocks. Let the $i$th, $1 \leq i < D/3$, set of $3M/D$ blocks be requested from a disk $d_i$, where $d_i$ satisfies the following conditions
   - No block from disk $d_i$ has been requested in $\eta$ till block $r_{i\alpha}$ has been requested.
   - If the number of blocks prefetched by $\mathcal{A}$ from disk $d$ at the instant when the reference for block $r_{i\alpha}$ has been serviced is $p_d$, then $p_{d_i} = \min_d\{p_d\}$; that is, $d_i$ is the disk from which the least number of blocks have been prefetched by $\mathcal{A}$ at the instant when block $r_{i\alpha}$ is referenced [5].
3. The last $2M$ requests are to blocks that are striped in a round-robin manner across all disks from which there have been no requests.

During the first $3M/D$ I/Os it is possible for an off-line algorithm to prefetch the first $M$ blocks of $\eta$ as they are all lie on different disks, and no disk has more than $3M/D$ blocks. The next $2M$ blocks can be fetched in $3M/D$ I/Os since the blocks are striped across $2D/3$ disks. Hence knowing $\eta$, we can construct a schedule which can service all references in at most $6M/D$ I/Os.

Now consider the performance of $\mathcal{A}$ while servicing $\eta$. After the first $3M/D$ references, $3M/D$ blocks from each of the disks $d_1, d_2, \ldots d_{D/3-1}$ are requested. By construction, at most $M/(D-i)$ blocks could have been prefetched for any $d_i$, when the $i$th set of 3M/D blocks are requested from disk $d_i$. Hence, the total time taken by $\mathcal{A}$ to service the first $M$ references of $\eta$ is at least

$$\begin{aligned} T_{\mathcal{A}} &\geq \frac{3M}{D} + \sum_{i=1}^{D/3-1} \left( \frac{3M}{D} - \frac{M}{D-i} \right) \\ &\geq M - M \left( H_{D-1} - H_{2D/3} \right) \\ &\geq M \left( 1 - \ln 3/2 \right) \\ &= \Omega(M) \end{aligned}$$

where $H_n$ is the $n$th Harmonic number.

Hence the competitive ratio of $\mathcal{A}$ is at least $\frac{\Omega(M)}{6M/D}$ which is $\Omega(D)$. □

[5] Note that this implies that at most $M/(D-i)$ blocks could have been prefetched from disk $d_i$ by $\mathcal{A}$.

# 4 Distributed Buffer Configuration

## 4.1 Local Lookahead

In the distributed buffer configuration there is no possibility of using free blocks from some other disk's local buffer. Intuitively the best we can do is to prefetch from a disk whenever possible. This, in fact, is the optimal algorithm in this configuration of the buffer (among all algorithms — online and off-line).

**Theorem 4** *In the distributed buffer configuration, GREED is the optimal algorithm, performing the least number of parallel I/Os.*

**Proof :** In [11] it was proved that an algorithm, P-MIN, minimizes the number of parallel I/Os in the distributed buffer configuration, when the reference string can have repetitions. When P-MIN is restricted to read-once reference strings it behaves like GREED. Hence GREED is optimal. □

## 4.2 Global $M$-block lookahead

From Theorem 4, algorithm GREED that uses only local lookahead, is optimal in the distributed buffer configuration. It is not difficult to construct a reference string $\Sigma$ for which any algorithm that uses only global $M$-block lookahead performs more parallel I/Os than GREED. We show however that NOM is near optimal; that is, its competitive ratio is $\Theta(1)$. To determine the competitive ratio of NOM we shall assume without loss of generality that OPT = GREED. In the following lemma we bound the performance of NOM in any phase.

**Lemma 12** *To service a sequence of $|\Sigma| = M$ requests, $T_{\mathrm{NOM}} \leq T_{\mathrm{OPT}} + M/D$.*

**Proof :** Let $\mathrm{NOM(d,n)}$ (respectively $\mathrm{OPT(d,n)}$) be the number of blocks from disk $d$ that are in its buffer immediately after NOM (respectively OPT) has referenced $n$ blocks in $\Sigma$. Define a potential function $\Phi(n) = \max_d\{\mathrm{OPT(d,n)} - \mathrm{NOM(d,n)}\}$. Let $T_{\mathrm{NOM}}(n)$ and $T_{\mathrm{OPT}}(n)$ be the number of I/Os done by NOM and OPT respectively to service $n$ references.

Note that since $|\Sigma| = M$, all blocks in the reference string are in NOM's lookahead window. Hence, on every I/O NOM will prefetch a block from disk $d$, if there is a free block in the local buffer and there is some unbuffered block from that disk.

Using the above definitions, we shall first prove inductively, that

$$T_{\mathrm{NOM}}(n) \leq T_{\mathrm{OPT}}(n) + \Phi(0) - \Phi(n) \tag{1}$$

The hypothesis is true for $n = 0$ since by definition $T_{\mathrm{OPT}}(0) = T_{\mathrm{NOM}}(0) = 0$. Let equation 1 be valid for $n = k$; that is, $T_{\mathrm{NOM}}(k) \leq T_{\mathrm{OPT}}(k) + \Phi(0) - \Phi(k)$. While servicing the next reference, NOM and OPT perform at most one parallel I/O and consequently $|\Phi(k+1) - \Phi(k)| \leq 1$. Now four cases are possible for $n = k+1$, depending on how OPT and NOM service the $k+1$th request:

- Neither OPT nor NOM do an I/O: $\Phi(k+1) = \Phi(k)$ as the referenced block must have been in the both OPT's and NOM's buffer. Also, $T_{\mathrm{NOM}}(k+1) = T_{\mathrm{NOM}}(k)$ and $T_{\mathrm{OPT}}(k+1) = T_{\mathrm{OPT}}(k)$.
- Both OPT and NOM do an I/O: The potential cannot increase when NOM performs an I/O; hence $\Phi(k+1) = \Phi(k)$ or $\Phi(k+1) = \Phi(k) - 1$. Also, $T_{\mathrm{NOM}}(k+1) = T_{\mathrm{NOM}}(k) + 1$ and $T_{\mathrm{OPT}}(k+1) = T_{\mathrm{OPT}}(k) + 1$.
- NOM does an I/O but OPT does not: In this case $r_{k+1}$, from disk $d$, must have been in the OPT's buffer while no block from that disk is in NOM's buffer. Hence $\Phi(k) \geq 1$. Also in

this I/O, NOM *will* prefetch a block from a disk $k \neq d$ if there are any unbuffered blocks from that disk. On the other hand OPT consumed a block from the buffer of disk $d$. Hence $\Phi(k+1) = \Phi(k) - 1$. Also, $T_{\mathrm{NOM}}(k+1) = T_{\mathrm{NOM}}(k) + 1$ and $T_{\mathrm{OPT}}(k+1) = T_{\mathrm{OPT}}(k)$.

- OPT does an I/O but NOM does not: Since OPT performed an I/O, $\Phi(k+1) = \Phi(k) + 1$ or $\Phi(k+1) = \Phi(k)$. Also, $T_{\mathrm{NOM}}(k+1) = T_{\mathrm{NOM}}(k)$ and $T_{\mathrm{OPT}}(k+1) = T_{\mathrm{OPT}}(k) + 1$.

In all cases, $T_{\mathrm{NOM}}(k+1) \leq T_{\mathrm{OPT}}(k+1) + \Phi(0) - \Phi(k+1)$. Hence equation 1 holds for $n = k+1$. Also, by definition $T_{\mathrm{NOM}}(M) = T_{\mathrm{NOM}}$ and $T_{\mathrm{OPT}}(M) = T_{\mathrm{OPT}}$. Therefore $T_{\mathrm{NOM}} \leq T_{\mathrm{OPT}} + \Phi(0) - \Phi(M)$; proving that $T_{\mathrm{NOM}} \leq T_{\mathrm{OPT}} + \frac{M}{D}$. □

Finally a bound on the competitive ratio of NOM in the distributed buffer configuration is given by the following theorem. The proof mainly consists of showing that in one of two contiguous phases OPT does at least $M/2D$ I/Os.

**Theorem 5** *NOM has a competitive ratio of $\Theta(1)$, in the distributed buffer configuration.*

**Proof :** Consider two disjoint consecutive phases $phase(i)$ and $phase(i+1)$ of the reference string $\Sigma$.

First, if OPT does more than $M/2D$ I/Os in at least one of the two phases then we have,from Lemma 12, that $T_{\mathrm{NOM}}/T_{\mathrm{OPT}}$ for the two phases is $O(1)$.

Assume for the sake of contradiction that OPT does less than $M/2D$ I/Os in both phases. This implies that less than $M$ blocks were fetched in $phase(i)$ and $phase(i+1)$. But a total of $2M$ blocks are consumed in the same two phases. Since at most $M$ blocks could have been present in the buffer at the start of $phase(i)$ we have a contradiction.

The corresponding lower bound follows from Theorem 6. □

Finally, we show that in the distributed buffer configuration, global $M$-block lookahead is not as powerful as local lookahead. This is primarily due to the fact that global lookahead does not necessarily give information to prefetch on disks which have space in the local buffers. This causes disks to idle in spite of having free space in their local buffers. However the overall performance is not hit much, as shown by Theoremtheo:NOM-dist-buffer.

Let $\mathcal{A}$ denote any parallel prefetching algorithm with global $M$-block lookahead. We show below how to construct a nemesis reference string $\eta$ of length $2M$. A longer reference string can be obtained by repeating $\eta$ an arbitrary number of times.

**Definition 8** The reference string $\eta$ is made up of 2 phases,$phase(1)$ and $phase(2)$.

- In $phase(1)$, the first $2M/D$ blocks $\langle r_k \rangle$, where $0 \leq k < 2M/D$, are from disk 1. The rest of the $M(D-2)/D$ blocks $\rangle r_k \rangle$, where $2M/D \leq k < M$, are striped in a round-robin fashion across all disks except 1 and 2.

- In $phase(2)$, the first $2M/D$ blocks $\langle r_k \rangle$, where $M \leq k < M + 2M/D$, are striped across all disks except 2. The next $2M/D$ blocks $\langle r_k \rangle$, where $M + 2M/D \leq k < M + 4M/D$, are from disk 2. The rest of the blocks $\langle r_k \rangle$, where $M + 4M/D \leq k < 2M$, are striped across all disks except disk 2.

Figure 5 illustrates the set of requests made in the nemesis string $\eta$ described above when $M = 15$ and $D = 5$.

**Theorem 6** *There exist a reference string, of arbitrary length, for which the number of I/Os required by any algorithm using only global $M$-block lookahead is greater than the number of I/Os done by the optimal off-line algorithm.*

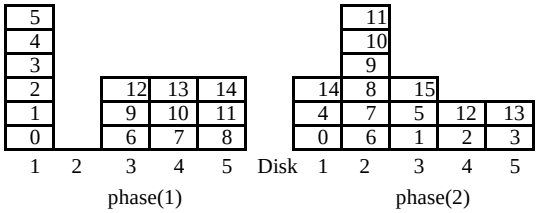


Figure 5: Example of $phase(1)$ and $phase(2)$ of $\eta$ for $M = 15$ and $D = 5$

**Proof :** We shall show that the reference string $\eta$ defined above is such that $T_{\mathcal{A}}/T_{OPT} > 1$. Lemma 13 shows that to service the string $\eta$ $\mathcal{A}$ makes at least $4M$ I/Os. On the other hand we show in Lemma 14 that the same sequence is scheduled by OPT (or GREED) in $3M + M/D$ I/Os. □

From the construction it may be noted that no block from disk 2, belonging to $phase(2)$ is revealed to algorithm $\mathcal{A}$ until it services the first $2M/D$ blocks in $phase(1)$. Due to this it is unable to exploit the parallelism across blocks from disks 1 and 2. The following lemma formalizes this intuition.

**Lemma 13** *Algorithm, $\mathcal{A}$ incurs at least $T_{\mathcal{A}} > 4M/D$ I/Os to service $\eta$.*

**Proof :** As the first $2M/D$ blocks in $phase(1)$ are requested from disk 1, $\mathcal{A}$ incurs at least $2M/D$ I/Os to service the first $2M/D$ requests of $phase(1)$. Now till request $r_{2M/D-1}$ is serviced, the lookahead window does not extend past request $r_{M+2M/D-1}$. Hence no block of $phase(2)$ from disk 2 is prefetched till request $r_{2M/D}$ is serviced. Hence a total of at least $2M/D$ I/Os is required by $\mathcal{A}$ to service requests $\langle r_k \rangle$, where $2M/D \leq k < 2M$. Therefore a total of at least $4M/D$ I/Os is incurred by $\mathcal{A}$ to service $\eta$. □

From the above proof it can be seen that a greedy schedule which fetched the first $M/D$ blocks of $phase(2)$ from disk 2, can save $M/D$ I/Os. Formally we show in the following lemma that $GREED$ needs only $3M/D$ I/Os to service $\eta$.

**Lemma 14** *The optimal algorithm (GREED) incurs $3M + M/D$ I/Os to service $\eta$.*

**Proof :** To service requests in $phase(1)$, OPT incurs $2M/D$ I/Os. As no block is requested from disk 2 in $phase(1)$, $M/D$ blocks can be prefetched from disk 2 for $phase(2)$. Hence the total number of I/Os that need to be done in $phase(2)$ is $M/D$; therefore the total number of I/Os incurred by GREED is $3M/D$. □

# 5 Probabilistic Setting

In this section we consider the parallel prefetching and buffer management 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 rather when the reference string can be said to be 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.

The superior performance in probabilistic settings can be said to motivate the explicit randomized layout approach employed for multiple data streaming in [1, 10]. The same bounds that hold

for performance with respect to a stochastic adversary hold for worst-case expected performance with respect to randomization internal to the algorithm.

When the read-once reference string is such that each block may originate *independently* from any disk with uniform probability, the analysis uses results proved for the classic *urn occupancy* problem [4]. A complication arises while considering runs or videos that are striped, with each stream starting on a randomly chosen disk: this complication is related to the loss of probabilistic independence with respect to the disks from which successive blocks of the reference string may originate. While merging striped runs (streaming striped videos) such that the first block of each run (video) was placed on an independently chosen uniformly random disk, there exists a dependency among the disks from which contiguous blocks of the resulting reference string originate. In [1], the authors formulated and analyzed the *dependent occupancy* problem and proved bounds identical to the ones (up to lower order terms) for the classic occupancy problem [4].

Below we state the theorems pertaining to our models for parallel prefetching that may be proved using these results. Theorem 8 for the shared buffer configuration is from [8].

**Theorem 7** *To service stochastically generated read-once reference strings of length $N$, NOM incurs the minimum expected number of I/Os, namely $\Theta(N/D)$, in both the shared and distributed buffer configurations working with a buffer of size $\Omega(D \log D)$.*

**Theorem 8** *To service stochastically generated read-once reference strings of length $N$, GREED incurs the minimum expected number of I/Os, namely $\Theta(N/D)$. In the case of the distributed buffer configuration, it requires a buffer of size $\Omega(D \log D)$ to attain this I/O bound. In the case of the shared buffer configuration, it requires a buffer of size $\Omega(D^2)$ to attain that I/O bound.*

# 6 Practical Implementations of Lookahead

In this section, we describe the techniques of forecasting and flushing which make possible a practical implementation of local and global lookahead.

## 6.1 Implementing Local Lookahead

As we mentioned in section 2, in case of applications such as external merging, multimedia streaming, etc., the data streams are typically striped across the parallel disk system; sometimes even randomized striping is employed as in [1, 10]. In this section, we discuss how to use a forecasting data structure [1] to implement local lookahead under these circumstances.

In the applications of interest, each stream might be a sorted run of records that is expected to be merged or a compressed sequence of frames or some other multimedia data units that is expected to be played back. Intuitively, each record in the sorted run and each frame in the video stream have a certain natural *time-stamp* signifying when that record will be consumed; that is, merged or transmitted for display.

For instance, in external merging the key value of a record provides a natural time-stamp, since it determines when the record is consumed. Similarly, the time-stamp of a block of video is determined by the compression of the preceding frames.

Thus at any point of time during the parallel processing of multiple streams of data, the next block that should be prefetched from any disk is the block with the smallest time-stamp from the set of blocks resident on that disk, considering all streams having blocks on that disk. Therefore, implementing local lookahead involves implementing a simple, efficient mechanism to keep track of the block with the smallest time-stamp on each disk at all times.

In order to implement local lookahead, we follow the approach of implanting in each disk block of the stream, the value of the time-stamp of the next block of that stream that resides on the same

disk. This information can easily be implanted in each block of each stripe of each stream with negligible increase in occupied disk space. We refer the reader to [1] for details regarding the maintenance and use of the forecasting data structure during the streaming and estimates of the marginal memory requirements of such an approach.

### 6.2 Implementing Global Lookahead

The previous subsection makes possible an implementation of local lookahead and thus the algorithm GREED. In this subsection, we show how to combine local lookahead with the simple technique of flushing [1] to effectively implement global $M$-block lookahead and an algorithm that performs at least as well as NOM. The description in [1] provides the details of the algorithm, which we briefly sketch here.

At any time $t$ during the computation, let $M_t$ denote the set of blocks in the buffer and let $S_t$ denote $D$ blocks, each one being the block with the smallest time-stamp on one disk. It may be verified that the following algorithm incurs *no more* parallel I/O operations than does NOM and is optimal:

Whenever $|M_t| \leq M - D$, we read in all $D$ blocks of the set $S_t$. When $|M_t| > M - D$, we flush (empty the buffer) and read as required so as to ensure that *the $M$ blocks with the $M$ smallest time-stamps in the set $M_t \cup S_t$ are in the buffer immediately after completion of the read operation.*[6]

The flush operation by itself does not involve any I/O. Hence the forecasting data structure and the technique of flushing yield a simple, efficient implementation of global $M$-block lookahead for read-once consumption sequences.

## 7 Conclusions

In this paper we presented a competitive analysis framework for online parallel prefetching algorithms serving an important class of reference strings on parallel disk systems. Our prefetching algorithms are based on novel and practically realizable forms of bounded lookahead. We considered a variety of scenarios and presented upper and lower bounds for variants of the online problem that encompass many practical situations. Besides theoretically analyzing the problems at hand, we also discuss how to use simple techniques such as forecasting and flushing in order to implement the various forms of lookahead so vital for prefetching.

[6] A simple dynamic data structure to maintain the order of consumption of memory resident blocks may be used along with the forecasting data structure [1] to carry out these operations.

# References


[1] R. D. Barve, E. F. Grove, and J. S Vitter. Simple Randomized Mergesort on Parallel Disks. *Parallel Computing*, 23(4):601–631, June 1997.

[2] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage. *IBM Systems Journal*, 5:78–101, 1966.

[3] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High Performance Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[4] N. L. Johnson and S. Kotz. *Urn Models and Their Application: an Approach to Modern Discrete Probability Theory*. Wiley, New York, 1977.

[5] M. Kallahalla. *Competitive Prefetching and Buffer Management for Parallel I/O Systems.* Master's thesis, Rice University, May 1997.

[6] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *37th Annual Symposium on Foundations of Computer Science*, October 1996.

[7] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley Publishing Co., 1973.

[8] V. S. Pai, A. A. Schäffer, and P. J. Varman. Markov Analysis of Multiple-Disk prefetching Strategies for External Merging. *Theoretical Computer Science*, 128(1–2):211–239, June 1994.

[9] D. D. Sleator and R. R. Tarjan. Amoritzed Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[10] R. Tewari, D. M. Dias, R. Mukherjee, and H. M. Vin. High Availability in Clustered Multimedia Servers. In *Proceedings of the 12th International Conference on Data Engineering*, pages 645–654, February 1996.

[11] P. J. Varman and R. M. Verma. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Proceedings of 1996 Symposium on Foundations of Software Technology and Theoretical Computer Science*. LNCS 1180, Springer Verlag, December 1996.

[12] J. S. Vitter and E. A. M. Shriver. Optimal Algorithms for Parallel Memory, I: Two-Level Memories. In *Algorithmica*, 12(2-3):110-147, 1994.

[13] W. Zheng and Per-Åke Larson. A Memory-Adaptive Sort (MARSORT) for Database Systems. *CASCON'96*, November 1996.