Journal of Algorithms **36,** 152–181 (2000)
doi:10.1006/jagm.2000.1089, available online at http://www.idealibrary.com on IDEAL®

Competitive Parallel Disk Prefetching and Buffer Management[1]

Rakesh Barve[2]

Department of Computer Science, Duke University, Durham, North Carolina 27708
E-mail: rbarve@cs.duke.edu

Mahesh Kallahalla[3] and Peter J. Varman[3]

Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77251
E-mail: kalla@rice.edu; pjv@rice.edu

and

Jeffrey Scott Vitter[4]

Department of Computer Science, Duke University, Durham, North Carolina 27708
E-mail: jsv@cs.duke.edu

Received October 5, 1998

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

[1] A preliminary version of this paper has appeared in the Proceedings of the ACM Fifth Annual Workshop on I/O in Parallel and Distributed Systems.
[2] 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.
[3] Supported in part by a grant from the Schlumberger Foundation and by the National Science Foundation under Grant CCR-9704562.
[4] 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.

0196-6774/00 $35.00

occur frequently in existing systems. An important aspect of our work is that we show how to implement both of 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 up to 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 the case of a 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.

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 computer systems incorporate some form of parallel I/O.

Prefetching is a powerful technique used to reduce the I/O latency seen by an application. In a parallel I/O system prefetching can be employed to obtain parallelism in disk accesses and to better utilize the available I/O bandwidth. To fully exploit this potential, it is necessary to design prefetching and buffer management algorithms that ensure 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 are 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. 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 makes it possible for the computation to access blocks in the order specified by the reference string.

A recent study [6] focused on the off-line problem of serving an arbitrary but fully known reference string of blocks spread across D independent disks, using parallel prefetching in conjunction with page replacement.[5] 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 tightly-coupled configuration, where the buffer can be shared by the different disks, was not considered.

In this paper we present a competitive analysis framework for prefetching algorithms on parallel disk systems for a restricted family of reference strings. In contrast to the requirement of knowing a priori the entire reference string, our parallel prefetching approach is based on models of bounded lookahead that are easily realizable in practice. Our restricted family of access sequences are called *read-once* reference strings. In such a reference string all accesses are read-only and no block is referenced more than once. Read-once reference strings arise naturally and frequently in I/O-bound applications like external merging and mergesorting (including carrying out several of these concurrently [13]) and in 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 an effective prefetching algorithm need only fetch blocks in the order of their appearance in the reference string. Given an I/O buffer of size M blocks, a prefetching algorithm that can look ahead M blocks into the reference string would always know the next memory-load of accesses and could fill up the buffer with maximal parallelism. Counter to this intuition, we show that in the parallel model the information provided by a lookahead of M is 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 for optimal prefetching.

[5] Note that replacement decisions are necessitated by the fact that the I/O buffer can hold only a fixed number of pages.

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

FIG. 1. Scheduling in order.

As an illustration, consider a system consisting of three 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, Σ, is $A_1 A_2 A_3 A_4 B_1 C_1 A_5 B_2 C_2 A_6 B_3 C_3 A_7 B_4 C_4 C_5 C_6 C_7$. Say that a parallel I/O is initiated only when the referenced block is not present in the buffer. The schedule in Fig. 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 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 Σ means that we fetch B_3; continuing in this manner we obtain a schedule of length 9. In an alternative schedule (Fig. 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 Σ 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 Σ.

It is unclear how best to schedule I/Os on a parallel I/O system. The first step in this direction is to determine bounds on the achievable performance of scheduling policies that know the next memory-load of references and the methods to achieve these bounds. We obtain the interesting result that *any parallel prefetching algorithm with a bounded lookahead of M* may need $\Omega(\sqrt{D})$ 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 M-block lookahead to fetch blocks from a disk in the order of their appearance in the reference string never requires more than $O(\sqrt{D})$ times the number of parallel I/O operations required by the optimal off-line prefetching algorithm. Thus,

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

FIG. 2. Scheduling out of order.

$\Theta(\sqrt{D})$ is a tight fundamental bound on the performance of bounded-lookahead parallel prefetching relative to the optimal off-line algorithm.

Motivated by the above results, in this paper we study online parallel prefetching algorithms for read-once sequences in several models with differing parallel-disk configurations and lookahead assumptions. Finally, we identify practical situations in which our models of lookahead are applicable and 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 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. *System Model*

In this section we present a precise description of the parallel I/O system models, models for lookahead, and performance metrics.

Model of I/O System. We consider the standard parallel disk model [12] consisting of D parallel disks with an associated I/O buffer capable of holding M blocks, $M \geq D$. A block must be present in the buffer before a request for it can be serviced. In each parallel I/O step, up to D blocks, at most one from each disk, may be read concurrently into the buffer. Since our emphasis in this paper is on using prefetching to improve I/O parallelism, we ignore the time taken by the computation to process data. Thus, the performance of a prefetching algorithm, on a reference string Σ, is the number of parallel I/O operations required to serve that reference string. We shall use the abbreviated term "I/O" to refer to a "parallel I/O step."

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.

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].

Buffer configurations. We consider two natural configurations of the parallel disk system, modeling commonly used I/O architectures. These will be referred to as the *distributed buffer* configuration and the *shared buffer* configuration and are illustrated in Fig. 3.

Distributed Buffer. In this configuration each disk has a local, private buffer of C blocks, such that the total memory in the system $M = CD$. 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 the shared buffer this means that there are at most M blocks in the buffer at any time; in the case of a distributed buffer there are never more than M/D buffered blocks from any disk.

Models of lookahead. We consider only ready-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. These models are motivated by the target streaming applications and how lookahead can be implemented in such situations (details are presented in Section 6).

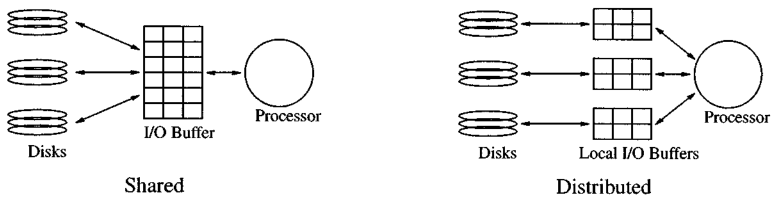

FIG. 3. Two configurations of the I/O buffer.

Global M-Block Lookahead. Let the reference string $\Sigma = r_1, r_2, \dots, r_N$, 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 Σ, $r_{i+1}, r_{i+2}, \dots, r_{i+M}$.

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

Performance metrics. 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. In the worst-case model, we express the I/O performance of online algorithms in terms of competitive ratios.

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 Σ the number of I/O operations, $T_A(\Sigma)$, that A requires to serve Σ 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 Σ.

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.

1.2. *Prefetching Algorithms*

We define scheduling algorithms NOM and GREED, which make use of M-block and local lookahead respectively. Neither of these algorithms will evict a block that has been fetched into the I/O buffer until a request for that block has been serviced. Also, as these algorithms service read-once reference strings, a block is evicted from the buffer immediately after it is referenced.

The performance of these algorithms is analyzed in Section 3 for the shared buffer configuration and in Section 4 for the distributed buffer configuration.

NOM: NOM uses global M-block lookahead to build 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 M-block lookahead, provided there is space in the buffer.

As the depth of lookahead used by NOM is M, the shared buffer configuration will always have free buffer space for the unread blocks.

However, in the distributed buffer configuration, some local buffers may be full, and no reads from the associated disks can occur.

GREED: GREED uses local lookahead to build a schedule as follows: on every parallel I/O it fetches the next block not in the buffer from each disk provided that there is space available in the 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 is not enough buffer space to fetch the next unread block from each disk, then only the requested 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 includes) 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 for the shared buffer configuration 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 at least 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 Σ in eight I/Os.

2. SUMMARY OF MAIN RESULTS

• 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 within a constant factor of the optimal prefetching algorithm 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 within a constant factor of the optimal prefetching algorithm 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 $c > 1$.

• 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 the 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.

3. SHARED BUFFER CONFIGURATION

In the shared buffer configuration a globally shared buffer is used to cache all blocks fetched from the disks. Unlike the distributed-buffer configuration, there is no specific portion of the buffer allocated to any particular disk, and hence it is possible to allocate buffer space unevenly to different disks. This choice in allocating buffer space makes the problem of prefetching and buffer management difficult. The buffer management algorithm has to judiciously partition the buffer among blocks from different disks based on their anticipated usefulness.

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, overly aggressive prefetching may fill up the buffer with prefetched blocks, which may not be referenced until much later. Such blocks have the adverse effect of choking the buffer and reducing the parallelism in fetching more immediate blocks. On the other hand, it is *not* always better to fetch a near block in preference to a block that is required later. Such situations are identified and used in lower bounding the performance of algorithms using global M-block lookahead in Section 3.1. A good prefetching and buffer management algorithm should decide how much buffer space to allocate for a particular I/O and which blocks to fetch in that I/O.

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 access to only the limited window 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. Since the buffer can hold at most M blocks, such algorithms can keep the buffer filled with *immediately* useful blocks and therefore may intuitively be expected to perform very well. 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 demonstrates that in the shared buffer configuration deep prefetching, beyond the next M references, can significantly improve performance.

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 Σ, which forces the online algorithm to perform $\Omega(\sqrt{D})$ times the number of I/Os incurred by the optimal off-line algorithm OPT on Σ.

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 Σ that consists of references $r_0, r_1, r_2, r_3, \ldots$. The string Σ is said to consist of a sequence of phases $\Sigma = \text{phase}(0), \text{phase}(1), \ldots,$ 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 of the blocks (and their order of reference) in that phase. As the computation proceeds, the lookahead window includes blocks from the following phase as well. Hence, in general, the lookahead window can span more than one but at most two phases.

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 $\mathscr{A}$. Let $p_{\mathscr{A}}(i, d)$ 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 $\text{dom}_{\mathscr{A}}(i) = \max_d\{c(i, d) - p_{\mathscr{A}}(i, d)\}$, the maximum number of blocks from phase(i) that need to be fetched from any single disk.

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

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

We will use the following notation during our analysis. Let $T_{\mathscr{A}}$ be the total number of I/Os used by $\mathscr{A}$ to service Σ, and let T_{OPT} be the number of I/Os taken by OPT to service Σ. We use $\mathscr{D}$ to denote the set of D parallel disks. To facilitate the presentation of our lower bound proof we define *balanced* and *loaded* phases.

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

A *loaded phase*, phase(i), with a *hot disk* d_i consists of blocks $\langle r_k \rangle$, $iM \le k < (i + 1)M$, laid out so that the first $M - H$ blocks $\langle r_k \rangle$, where $iM \le k < (i + 1)M - H$, are striped in a round-robin manner across the set $\mathscr{D} - \{d_i\}$ of $D - 1$ disks and the remaining H blocks $\langle r_k \rangle$, where $(i + 1)M - H \le k < (i + 1)M$, all originate from the *hot* disk d_i. H is an integer parameter whose value we shall set later.

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

Note that if no blocks from a loaded phase were prefetched prior to the beginning of the phase, 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

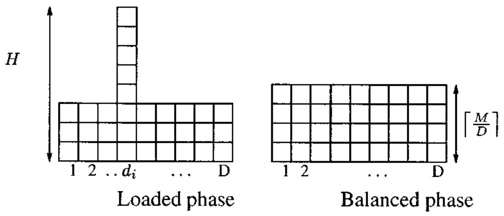

FIG. 4. Illustration of loaded and balanced phases.

substantial number of blocks for the next loaded phase, forcing it to perform $\Omega(H)$ I/Os in every such 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/O's.

Given any deterministic online algorithm $\mathscr{A}$ with a bounded lookahead of M blocks, in the following definition we show how to construct a nemesis reference string from balanced and loaded phases, depending on $\mathscr{A}$'s prefetching decisions.

DEFINITION 5. Let P be an integer parameter less than D whose value we shall set later. We construct a reference string η of $2PM$ references such that the nemesis string Σ can be obtained by concatenating strings constructed in the same manner as η an arbitrary number of times. The reference string η consists of a sequence of $2P$ phases, phase(1), ..., phase($2P$), such that odd-numbered phases phase($2k-1$), with $1 \leq k \leq P$, are loaded phases and even-numbered phases, phase($2k$) with $1 \leq k \leq P$, are balanced phases.

Each loaded phase has a different hot disk: the first loaded phase, phase(1), has disk 1 as the hot disk. The hot disk of every subsequent loaded phase is dependent on $\mathscr{A}$'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 phase($2k-1$). Let G_k denote the set $\mathscr{D} - B_k$ of $D - k + 1$ disks not in B_k. It is possible that on account of $\mathscr{A}$'s prefetching, one or more future blocks[6] are already in the I/O buffer at the end of phase($2k-3$). Among the disks in G_k, the disk d_k that has the smallest number of future blocks in algorithm $\mathscr{A}$'s buffer at the end of phase($2k-3$) is chosen to be the hot disk of phase($2k-1$).[7]

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 Σ defined above is such that $T_{\mathscr{A}}/T_{\mathrm{OPT}}$ is $\Omega(\sqrt{D})$. In Lemma 1 we show that $\mathscr{A}$ will incur at least $PH - P\lfloor M/(D-P)\rfloor$ I/Os for every instance of the substring η. On the other hand, we show in Lemma 2 that if $PH \leq M - D$ and $H > M/D$, then there exists a schedule $\mathscr{S}$ that incurs no more than $H + (2P - 1)\lceil M/D\rceil$ I/Os for every instance of the substring η. By assigning $P =$

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

[7] This is a valid construction as $\mathscr{A}$ 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*($2k-3$).

$\lceil\sqrt{D}/3\rceil$ and $H = \lfloor(M - D)/P\rfloor$, we have $T_{\mathscr{A}} = \Omega(M)$ and $T_{\text{OPT}} = O(M/\sqrt{D})$ provided $M \geq 2D$, whence the theorem 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 $T_{\mathscr{A}}$ and T_{OPT}. ▮

Intuitively, the subsequence η is constructed by alternating loaded phases with balanced phases. Loaded phases are constructed to have a large number ($H = \Theta(M/\sqrt{D})$) of blocks requested from a single disk, and the rest of the blocks striped across all of the remaining disks. 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 η partitions the D disks into a set $\mathscr{D}_1$ of P disks that are hot in some phase, and the rest of the $D - P$ disks into a set $\mathscr{D}_2$. We force the online algorithm $\mathscr{A}$ to incur about $H - M/(D - P)$ I/Os for every consecutive pair of phases of η, thus resulting in a cost of about $HP - MP/(D - P)$ I/Os for $\mathscr{A}$. On the other hand, we show that it is possible to design an optimal off-line schedule $\mathscr{S}$ that fetches HP future blocks from P disks of the set $\mathscr{D}_1$ in the first phase itself, thereby leaving an evenly balanced disk block placement for subsequent phases. Thus, $\mathscr{S}$ incurs only about $H + (2P - 1)(M/D)$ I/Os overall. The following lemmas formalize the above intuition.

LEMMA 1. *Algorithm $\mathscr{A}$ incurs at least $T_{\mathscr{A}} \geq HP - P\lfloor M/(D - P)\rfloor$ I/Os to service η.*

Proof. For $1 \leq k \leq P$, consider the kth loaded phase, phase$(2k - 1)$, in η. Let the next loaded phase, phase$(2k + 1)$, have d_{k+1} as the hot disk.

By Definition 5, 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 $\mathscr{A}$ at the end of phase$(2k - 1)$. Because the buffer capacity is M, at most M prefetched blocks can be in the buffer at the end of phase$(2k - 1)$, and specifically the number of prefetched blocks from disk d_{k+1} in the buffer at the end of phase$(2k - 1)$ is at most $\lfloor M/(D - k)\rfloor$. During phase$(2k)$, one I/O is required for each block of phase$(2k + 1)$ that $\mathscr{A}$ chooses to prefetch from disk d_{k+1}. Hence if during phase$(2k)$ $\mathscr{A}$ prefetches n_k blocks from disk d_{k+1} for phase$(2k + 1)$, it must perform at least n_k I/Os in phase$(2k)$. Thus, the total number of blocks from disk d_{k+1} that could have been prefetched by the start of phase$(2k + 1)$ is no more than $\lfloor M/(D - k)\rfloor + n_k$. So, by Claim 1 the total number of I/Os done by $\mathscr{A}$ in phase$(2k+1)$ is at least $H - \lfloor M/(D-k) \rfloor - n_k$. The total number of I/Os done during phase$(2k)$ and phase$(2k+1)$ combined is therefore at least $H - \lfloor M/(D-k) \rfloor$. The number of I/Os done by $\mathscr{A}$ to service η is thus

$$T_{\mathscr{A}} \geq H + \sum_{1 \leq k < P} \left(H - \lfloor M/(D-k) \rfloor \right).$$

This quantity is at least $HP - P\lfloor M/(D-P) \rfloor$. ∎

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, thus reducing the number of I/Os that need to be performed in future loaded phases to $O(M/D)$. This exploits the fact that balanced phases can be serviced with full parallelism (needing about M/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 2. *A schedule $\mathscr{S}$ can be constructed that incurs at most $T_S \leq H + (2P-1)\lceil M/D \rceil$ I/Os to service η, if $HP \leq M - D$ and $H > M/D$.*

Proof. We construct a schedule $\mathscr{S}$ to service η by running the following algorithm on it. As before, in phase$(2k-1)$ let d_k be the hot disk, for $1 \leq k \leq P$, and let $\tau = \lceil (M-H)/(D-1) \rceil$.

- In phase(1) of η, we prefetch as follows:

  —During the first τ I/Os we fetch only the blocks required in phase(1).

  —During the remaining $H - \tau$ I/Os of phase(1), we prefetch $H - \tau$ blocks from each of the future hot disks, d_k, $2 \leq k \leq P$, of the rest of the P loaded phases. The prefetched blocks from disk d_k are precisely *the farthest*[8] *$H - \tau$ blocks of phase*$(2k-1)$.

- During each subsequent phase, we fetch blocks of that phase with full disk parallelism. Since at most $(H-\tau)P$ 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.

[8] The i farthest blocks of a phase are the i belonging to that phase that are referenced farthest in the future, relative to all blocks of that phase.

Since $\mathscr{S}$ has prefetched $H - \tau$ 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 phase(1) will be reduced to τ. Hence, in each of these loaded phases, $\mathscr{S}$ will incur only τ I/Os.

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

$$T_S = H + P\lceil M/D \rceil + (P - 1)\tau.$$

When $H > M/D$, we have $\tau \leq \lceil M/D \rceil$ and hence $T_S \leq H + (2P - 1)\lceil M/D \rceil$. ▮

3.1.2. *Upper bound on the competitive ratio.* From Theorem 1, the competitive ratio of any online prefetching algorithm using global M-block lookahead is $\Omega(\sqrt{D})$. This raises the question as to whether we can design an algorithm that 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 derive a matching 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. Intuitively, OPT benefits by prefetching blocks for several phases simultaneously. So, the ratio is highest when OPT performs the fewest number of I/Os to get a certain amount of benefit over NOM. 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 phases across which the benefit is spread. Since OPT should do at least M/D I/Os per phase, this indicates that minimizing the number of I/Os done by OPT requires balancing these two factors.

The following claim 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. 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.

Claim 2. 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 Σ and a parallel prefetching algorithm $\mathscr{A}$ serving it. A phase of Σ is said to have been completed at the time the final block in that phase is referenced.

The sequence of consecutive I/Os made by $\mathscr{A}$ 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 referred to as *the I/Os incurred by* $\mathscr{A}$ *in* phase(i).

Consider the set $P_{l,h}$ of blocks of phase(l) that are yet to be read after completion of the hth 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.

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

DEFINITION 6. Consider the jth I/O, R_j, of OPT and let it be incurred in phase(i). It is possible that 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 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 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 Σ, we break it down into contiguous subsequences called *superphases*.

DEFINITION 7. The ith superphase, denoted by $\mathscr{S}_i$, $i \geq 0$, is defined to be the subsequence $\langle \text{phase}(t), \text{phase}(t+1), \ldots, \text{phase}(t+s) \rangle$ such that the following two conditions are satisfied: (a) phase$(t-1)$ belongs in $\mathscr{S}_{i-1}$ if $i > 0$, else phase$(t) =$ phase(0) if $i = 0$; (b) the number of useful blocks prefetched in $\mathscr{S}_i$ is at least M; and (c) the number of useful blocks prefetched in phases $\langle \text{phase}(t), \text{phase}(t+1), \ldots, \text{phase}(t+s-1) \rangle$ 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 ith superphase $\mathscr{S}_i$ of the reference string. Let $\gamma_i = |\mathscr{S}_i|$ be the number of phases in $\mathscr{S}_i$. Let the number of useful blocks prefetched by OPT in superphases before $\mathscr{S}_i$ for phases of $\mathscr{S}_i$ be α_i and the number of useful blocks prefetched by OPT in $\mathscr{S}_i$ for phases in $\mathscr{S}_i$ be β_i.

Since at most one block can be fetched from a single disk in an I/O, it follows that in one I/O at most one useful block *per phase* can be fetched in an I/O. Hence we have the following lemma.

LEMMA 3. *For an arbitrary but fixed phase, at most one useful block can be prefetched in an I/O operation.*

The following key lemma shows that to bound the competitive ratio of NOM by $O(\sqrt{D})$, it is enough to show that OPT incurs $\Omega(M/\sqrt{D})$ I/Os in a superphase. Let $I_{\text{NOM}}(i)$ and $I_{\text{OPT}}(i)$ be the number of I/Os done in superphase $\mathscr{S}_i$ by NOM and OPT respectively.

LEMMA 4. *NOM needs at most $2M$ more I/Os than does OPT to service $\mathscr{S}_I$; $I_{\text{NOM}}(i) < I_{\text{OPT}}(i) + 2M$.*

Proof. To prove the lemma we shall first show that the difference in the number of I/Os done by NOM and OPT in a superphase is at most $\alpha_i + \beta_i$. The proof is then completed by showing that both α_i and β_i are no more than M.

Consider an arbitrary phase(i). Let the maximum number of blocks that are requested from some disk, d, in phase(i), be H. First, the number of I/Os done by NOM in phase(i) is no more than H. Also, by the definition of useful blocks, if OPT prefetches m useful blocks for phase(i), then it needs to do at least $H - m$ I/Os to fetch the remaining blocks of that phase. Thus NOM incurs at most m more I/Os than OPT in a phase belonging to $\mathscr{S}_i$, where m is the number of useful blocks prefetched by OPT in previous phases for that phase. Thus, overall, NOM performs at most $\alpha_i + \beta_i$ I/Os more than OPT, since this is the total number of useful blocks prefetched by OPT for phases in the superphase.

Let $\mathscr{S}_i = \langle \text{phase}(t), \text{phase}(t+1), \dots, \text{phase}(t+s) \rangle$. By the definition of useful blocks, no useful blocks can be prefetched for a phase while servicing that phase itself. Hence no useful block prefetched in phase($t + s$) can be for phases in $\mathscr{S}_i$. But by Definition 7, condition (c), the total number of useful blocks fetched in the other phases of $\mathscr{S}_i$ is less than M. Hence less than M useful blocks can be prefetched by OPT in $\mathscr{S}_i$ for phases in $\mathscr{S}_i$; therefore, $\beta_i < M$.

Since the buffer size is M, the total number of blocks prefetched for phases in $\mathscr{S}_i$ prior to that superphase is $\alpha_i \leq M$. We have already shown that $I_{\text{NOM}}(i) \leq I_{\text{OPT}}(i) + \alpha_i + \beta_i$, completing the proof. ∎

We shall now prove Theorem 2 by considering two mutually exclusive cases, whether γ_i, the number of phases for which OPT prefetches useful blocks, is greater or less than $\sqrt{D}$.

LEMMA 5. *If $\gamma_i \geq \sqrt{D}$, then $I_{\text{OPT}}(i) = \Omega(M/\sqrt{D})$.*

Proof. The total number of blocks referenced in $\mathscr{S}_i$ is at least $M\gamma_i$, since each phase references M blocks. Because the buffer size is M, at most M of these blocks could have been prefetched before the start of $\mathscr{S}_i$; hence, at least $M\gamma_i - M$ blocks must be fetched in $\mathscr{S}_i$. This would require at least $\lceil (M\gamma_i - M)/D \rceil$ I/Os. Given that $\gamma_i \geq \sqrt{D}$, $I_{\text{OPT}}(i) \geq M/\sqrt{D} - M/D$. ∎

LEMMA 6. *If $\gamma_i < \sqrt{D}$ and $\beta_i \geq M/2$, then $I_{\mathrm{OPT}}(i) = \Omega(M/\sqrt{D})$.*

Proof. By the definition of β_i, each of the β_i useful blocks is fetched for some phase in $\mathscr{S}_i$ in earlier phases of $\mathscr{S}_i$. Hence, there must be some phase in $\mathscr{S}_i$ such that at least $\lceil \beta_i/\gamma_i \rceil$ useful blocks were prefetched by OPT for that phase in previous phases of $\mathscr{S}_i$. It follows from Lemma 3 that at least $\lceil \beta_i/\gamma_i \rceil$ I/Os must have been incurred by OPT in $\mathscr{S}_i$. Hence, $I_{\mathrm{OPT}}(i) \geq \beta_i/\gamma_i > (M/2)/\sqrt{D} = M/2\sqrt{D}$. ∎

LEMMA 7. *Let $\mathscr{S}_i$ and $\mathscr{S}_{i+1}$ be two consecutive non-overlapping superphases. If $\gamma_i < \sqrt{D}$ and $\beta_i < M/2$, then the sum of the number of I/Os done by OPT in $\mathscr{S}_i$ and $\mathscr{S}_{i+1}$ is $\Omega(M/\sqrt{D})$.*

Proof. If $\gamma_{i+1} \geq \sqrt{D}$ or if $\beta_{i+1} \geq M/2$, then by an analysis similar to that of Lemma 5 and Lemma 6 it can be shown that over the two superphases OPT does at least most $\Omega(M/\sqrt{D})$ I/Os.

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 $\mathscr{S}_i$ are present in the buffer of OPT at the end of superphase $\mathscr{S}_i$, since at most $M/2$ useful blocks prefetched in $\mathscr{S}_i$ were for phases in $\mathscr{S}_i$ ($\beta_i < M/2$). Now, by Definition 7, in $\mathscr{S}_{i+1}$ 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 $\mathscr{S}_i$ or $\mathscr{S}_{i+1}$ must necessarily be for phases of $\mathscr{S}_{i+1}$. But the number of phases for which useful blocks were fetched in $\mathscr{S}_{i+1}$ is $\gamma_{i+1} \leq \sqrt{D}$. Consequently there must be some phase in $\mathscr{S}_{i+1}$ for which at least $\lceil M/2\sqrt{D} \rceil$ useful blocks were prefetched. Hence, invoking Lemma 3, OPT incurred at least $\lceil M/2\sqrt{D} \rceil$ I/Os during phases $\mathscr{S}_i$ and $\mathscr{S}_{i+1}$. ∎

THEOREM 2. *The competitive ratio of NOM is $O(\sqrt{D})$ and hence it is within a constant factor of the optimal algorithm using only global M-block lookahead.*

Proof. We divide the reference string into P contiguous nonoverlapping superphases; the sequence of phases beyond the last superphase (which do not form a superphase) will be referred to as the tail. Two situations are possible: (a) there is at least one superphase or (b) the reference string consists entirely of the tail.

In the first case, by Lemmas 5, 6, and 7 and the total number of I/Os done by OPT to service the entire reference string is at least $P/2 \times \Omega(M/\sqrt{D})$. On the other hand, by Lemma 4 the number of I/Os done by NOM is at most $(P+1)2M$ more than the number of I/Os done by OPT.[9] Hence the ratio of the number of I/Os done by NOM to those done by OPT is $O(\sqrt{D})$.

[9] Note that the bound in Lemma 4 holds for the tail also.

In the case when the entire reference string is made up of just the tail, let the number of useful blocks prefetched by OPT be U. Following the proof of Lemma 4, we can show that the number of I/Os done by NOM in the reference string is at most U more than the number of I/Os done by OPT. By following the proofs of Lemmas 5 and 6 we can show that OPT does at least $\Omega(U/\sqrt{D})$ I/Os. Hence the competitive ratio of NOM is $O(\sqrt{D})$. ∎

3.2. *Local Lookahead*

In this section, we consider the effects of using pure local lookahead: that is, the prefetching algorithm has no access to any information regarding the relative reference order 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 compared 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 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, a simple algorithm such as 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 $\mathscr{A}$ 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 $\mathscr{A}$ denote an arbitrary algorithm using only local lookahead. We shall prove the theorem by constructing a request sequence $\eta = \langle r_i \rangle$, which requires $\mathscr{A}$ to make $\Omega(M)$ I/Os. We shall also give a schedule that serves η 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 η.

Sequence η is constructed depending upon the behavior of $\mathscr{A}$ until the previous I/O. This is a valid construction of the reference string as $\mathscr{A}$ has no knowledge of the relative reference order of blocks across disks. By definition, local lookahead allows the algorithm knowledge of the order of references from any *single* disk. Let α and H be two integer parameters such that $\alpha H \leq M$: we shall assign values to these later. The length of η will be HD.

1. The first H blocks of η are requested from disk 1.

2. Divide the next $(\alpha - 1)H$ references into $\alpha - 1$ sets of H blocks. Let the ith, $1 \leq i < \alpha$, set of H 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 η until the block r_{iH} is requested.

• If the number of blocks prefetched by $\mathscr{A}$ from disk d just before block r_{iH} is referenced 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 has been prefetched by $\mathscr{A}$ just before block r_{iH} is referenced.

3. The last $H(D - \alpha)$ 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 H I/Os it is possible for an off-line algorithm to prefetch the first $H\alpha$ blocks of η as they all lie on different disks and $\alpha H \leq M$. The next $(D - \alpha)H$ blocks can be fetched in H I/Os since the blocks are striped across $D - \alpha$ disks. Hence, knowing η, we can construct a schedule that can service all references in at most $2H$ I/Os.

Now consider the performance of $\mathscr{A}$ while servicing η. After the first H references, H blocks from each of the disks $d_1, d_2, \ldots, d_{\alpha-1}$ are requested. By construction, d_i is chosen from a set of $D - i$ disks such that $\mathscr{A}$ has prefetched the least number of blocks from it. Hence, at most $\lceil M/(D - i) \rceil$ blocks could have been prefetched for d_i when the ith set of H blocks is requested. Hence, the total time taken by $\mathscr{A}$ to service the first $H\alpha$ references of η is at least

$$\begin{aligned} T_{\mathscr{A}} &\geq H + \sum_{i=1}^{\alpha-1} \left(H - \left\lfloor \frac{M}{D - i} \right\rfloor \right) \\ &\geq H\alpha - M(\mathscr{H}_{D-1} - \mathscr{H}_{D-\alpha}) \\ &\geq H\alpha - M \ln D/(D - \alpha), \end{aligned}$$

where $\mathscr{H}_n$ is the nth Harmonic number.

Now setting $\alpha = \lfloor D/3 \rfloor$ and $H = \lfloor M/\alpha \rfloor$, $T_{\mathscr{A}} \geq M(2/3 - \ln 3/2) = \Omega(M)$. This gives the result that the competitive ratio of $\mathscr{A}$ is $\Omega(D)$. ∎

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 I/Os.*

Proof. In [11] it was proved that an algorithm, P-MIN, minimizes the number of 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 as GREED. Hence GREED is optimal. ▌

4.2. *Global M-Block Lookahead*

From Theorem 4, the algorithm GREED, which uses only local lookahead, is optimal in the distributed buffer configuration. We show in Theorem 6 that every algorithm that uses global M-block lookahead has a nemesis string for which it performs more I/Os than GREED. However, we first show that NOM is near optimal (Theorem 5); that is, its competitive ratio is $\Theta(1)$. In the following lemma we bound the performance of NOM in any phase.

LEMMA 8. *To service a sequence of $|\Sigma| = M$ requests, the number of I/Os done by NOM is at most M/D more than the number of I/Os done by OPT.*

Proof. Without loss of generality we assume that OPT is the algorithm GREED. Let NOM(d, n) (respectively, 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 Σ. Let us define a potential function

$$\Phi(n) = \max_d \{\mathrm{OPT}(\mathrm{d},\mathrm{n}) - \mathrm{NOM}(\mathrm{d},\mathrm{n})\}.$$

Note that because the size of any buffer is M/D, $\Phi(n) \le M/D$. 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.

Using the above definitions, we shall first prove inductively that

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

The hypothesis is true for $n = 0$ since $T_{\mathrm{OPT}}(0) = T_{\mathrm{NOM}}(0) = 0$. Let Eq. (1) be valid for $n = k$; that is, $T_{\mathrm{NOM}}(k) \leq T_{\mathrm{OPT}}(k) + \Phi(0) - \Phi(k)$.

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 does an I/O. In this case, the referenced block must be in the buffer as no I/O is done for that block by either NOM or OPT. Hence, $\Phi(k + 1) = \Phi(k)$. Also, $T_{\mathrm{NOM}}(k + 1) = T_{\mathrm{NOM}}(k)$ and $T_{\mathrm{OPT}}(k + 1) = T_{\mathrm{OPT}}(k)$.
- Both OPT and NOM do an I/O. Since $|\Sigma| = M$, all blocks in the reference string are in NOM's lookahead window. Hence, in this I/O NOM will prefetch a block from a disk if there is a free block in the local buffer and there is some unbuffered block from that disk. Hence the potential cannot increase. Also, since at most one block can be fetched from any disk, the potential can also not decrease by more than one. Therefore, either $\Phi(k + 1) = \Phi(k)$ 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. Note that NOM fetches blocks from any disk in the order in which they occur in Σ. Hence, in this case r_{k+1}, from disk d, must have been in OPT's buffer while *no block* from that disk is in NOM's buffer; therefore, $\Phi(k) \geq 1$. 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 can fetch at most one block from a disk, either $\Phi(k + 1) = \Phi(k) + 1$ 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 Eq. (1) holds for $n = k + 1$. Also, $T_{\mathrm{NOM}}(M) = T_{\mathrm{NOM}}$ and $T_{\mathrm{OPT}}(M) = T_{\mathrm{OPT}}$. Therefore $T_{\mathrm{NOM}} \leq T_{\mathrm{OPT}} + \Phi(0) - \Phi(M)$; proving that $T_{\mathrm{NOM}} \leq T_{\mathrm{OPT}} + \frac{M}{D}$, since $\Phi(n) \leq M/D$. ∎

Finally, a bound on the competitive ratio of NOM in the distributed buffer configuration is given by the following theorem. The proof consists mainly 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 Σ.

First, if OPT does more than $M/2D$ I/Os in at least one of the two phases then we have, from Lemma 8, that $T_{\text{NOM}}/T_{\text{OPT}}$ for the two phases is $O(1)$.

Assume for the sake of contradiction that OPT does fewer than $M/2D$ I/Os in both phases. This implies that fewer 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 that 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 greatly affected, as shown by Theorem 5.

Let $\mathscr{A}$ denote any parallel prefetching algorithm with global M-block lookahead. We show below how to construct a nemesis reference string η of length $2M$. A longer reference string can be obtained by concatenating an arbitrary number of strings constructed in the same manner as η.

DEFINITION 8. The reference string η is made up of 2 phases, phase(1) and phase(2). Let H be an integer parameter greater than $2M/D$, whose value we shall set later.

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

Figure 5 illustrates the set of requests made in the nemesis string η described above when $M = 15$, $D = 5$, and $H = 6$.

THEOREM 6. *For any algorithm that uses only global M-block lookahead, there exists a nemesis reference string, of arbitrary length, for which the number of I/Os required by the algorithm is greater than the number of I/Os done by the optimal off-line algorithm.*

Proof. We shall show that the reference string η defined above is such that $T_{\mathscr{A}}/T_{\text{OPT}} > 1$. Lemma 9 shows that to service the string η, $\mathscr{A}$ makes

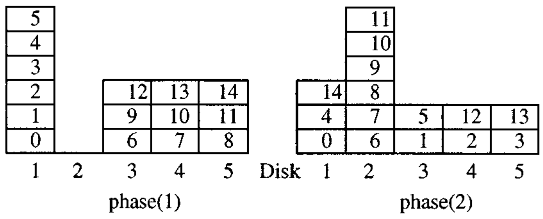

FIG. 5. Example of *phase*(1) and *phase*(2) of η for $M = 15$, $D = 5$, and $H = 6$.

at least $2H$ I/Os. On the other hand, we show in Lemma 10 that the same sequence is scheduled by OPT (or GREED) in $2H - M/D$ I/Os. ▌

From the construction it may be noted that no block from disk 2 belonging to phase(2) is revealed to algorithm $\mathscr{A}$ until it finishes serving the first H blocks in phase(1). Due to this it is unable to exploit the potential parallelism in accessing blocks from Disks 1 and 2. The following lemma formalizes this intuition.

LEMMA 9. *Algorithm $\mathscr{A}$ incurs at least $T_{\mathscr{A}} > 2H$ I/Os to service η.*

Proof. As the first H blocks in phase(1) are requested from disk 1, $\mathscr{A}$ incurs at least H I/Os to service the first H requests of phase(1). Now, until request r_{H-1} is serviced, the lookahead window does not extend past request r_{M+H-1}. Hence no block of phase(2) from disk 2 is prefetched, by $\mathscr{A}$, until the request r_H is serviced. Hence a total of at least H I/Os is required by $\mathscr{A}$ to service requests $\langle r_k \rangle$, where $H \le k < 2M$. Therefore a total of at least $2H$ I/Os is incurred by $\mathscr{A}$ to service η. ▌

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 $2H - M/D$ I/Os to service η.

LEMMA 10. *The optimal algorithm (GREED) incurs $2H - M/D$ I/Os to service η.*

Proof. To service requests in phase(1), OPT incurs H I/Os. As no block is requested from disk 2 in phase(1), M/D (as $H > M/D$) 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 $\max\{H - M/D, M/D\}$; therefore, the total number of I/Os incurred by GREED is $2H - M/D$, when $H > 2M/D$. ▌

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 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.

5.1. *Shared Buffer Configuration*

Previously, parallel I/O prefetching in the shared buffer configuration has been studied in a probabilistic setting, in the context of external merging [1, 8]. In external merging, the problem is to merge a set of sorted sequences, also called runs, into one globally sorted sequence. Theorem 7 presents the results for parallel prefetching in the shared buffer configuration that may be proved using results from [1, 8].

In [1], 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 NOM, 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 [4], 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), thereby leading to the performance bounds on NOM in the shared buffer configuration against a stochastic adversary.

In [8] the authors considered a data layout where each run is located on a single disk. In this case information about consecutive blocks in a run translates to having local lookahead. Using this form of lookahead and 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 in the shared buffer configuration, when each block originates from a disk chosen independently and with uniform probability.

THEOREM 7. *In the shared buffer configuration, to service stochastically generated read-once reference strings of length N, NOM incurs the minimum expected number of I/Os, namely $\Theta(N/D)$, using a buffer of size $\Omega(D \log D)$. GREED needs a buffer of size $\Omega(D^2)$ to attain that I/O bound.*

5.2. *Distributed Buffer Configuration*

Unlike the shared buffer model, in the distributed-buffer configuration buffers of one disk cannot be used to cache blocks from another disk. Normally this loss in sharing would result in underutilization of the buffer and would lead to a drop in performance. However, in the probabilistic setting, we show that the performance of NOM and GREED is at least as good as in the shared buffer configuration. In fact, because the distributed buffer caps the excessive prefetching that GREED may do from any disk, we show that GREED can attain full parallelism with a smaller buffer than in the shared buffer configuration.

In Theorem 8 we show that both NOM and GREED perform $\Theta(N/D)$ expected number of parallel I/Os to service stochastically generated reference strings of length N, using a buffer of size $\Omega(D \log D)$.

To bound the performance of these algorithms, it is useful to redefine a phase, phase(i), to be the substring $\langle b_j, b_{j+1}, \ldots, b_k \rangle$ of the reference string, such that (a) the block b_{j-1} occurs in phase phase($i-1$) and (b) if b_k is from disk d then phase(i) has exactly M/D blocks from disk d and fewer than M/D blocks from any other disk. Also, phase(1) starts with block b_1.

By this definition, all the blocks in a phase are within NOM's lookahead at the start of the phase. Hence NOM performs at most M/D I/Os to service a phase. Also, because there are at most M/D blocks referenced from any one disk, which is also the size of any single disk's buffer, GREED performs at most M/D I/Os to service a phase.

Claim 3. At most M/D I/Os are needed by NOM or GREED to service the requests in any phase.

Hence, to estimate the total number of I/Os done by either NOM or GREED to service the reference string, we just need to estimate the total number of phases in the reference string. This in turn can be done by counting the average number of blocks referenced in any phase. To find the expected number of blocks referenced in a phase, we consider the following urn problem: *Given D identical urns of capacity C each, find the average number of balls, $\mathscr{B}$, to be thrown before one urn gets filled, assuming that the probability of a ball falling in any urn is $1/D$.*

By associating each disk buffer with an urn of capacity $C = M/D$ and each block reference with the throw of a ball, we see that $\mathscr{B}$ is the average number of blocks referenced in a phase. The above problem has been studied in [4] to give the following expression for $\mathscr{B}$:

$$\mathscr{B} \geq D^{1-1/C}(C!)^{1/C}\Gamma(1 + C^{-1}).$$

Using $n! \geq (n/e)^n$ and the fact that $\Gamma(1 + C^{-1}) \geq 0.88$, for all positive C we get the desired result that $\mathscr{B} = \Omega(CD/(D^{1/C}))$. The expected number of phases is no more than $N/\mathscr{B} + 1$ and each phase requires at most M/D I/Os. Hence the expected number of I/Os is $O(ND^{1/C}/D)$. When $C = \Omega(\log D)$ the total number of I/Os is $\Theta(N/D)$.

THEOREM 8. *In the distributed buffer configuration, to service stochastically generated read-once reference strings of length N, both NOM and GREED incur the minimum expected number of I/Os, namely $\Theta(N/D)$, using a buffer of size $\Omega(D \log D)$.*

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*

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 reference strings. These applications involve sequentially retrieving data blocks from multiple streams laid out on disk. Fundamental difficulties [12, 8, 1] arise from the fact that the different streams are consumed at 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].

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]. Video servers generally either stripe video clips across disks in a round robin fashion or employ more sophisticated forms of striping [10] to mitigate the effects of hot-spots. In these situations, local lookahead does not come for free and involves picking out, for each disk, one block from the set of the 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.

Intuitively, each record in the sorted run and each frame in the video stream has 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 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 an 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 every block with a negligible increase in the occupied disk space. We refer the reader to [1] for details regarding the maintenance and use of the forecasting data structure during 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 of 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. Intuitively, the algorithm fetches the block with the earliest time-stamp from each of the D disks whenever there are at least D blocks free in the buffer. If not, let K be the number of free buffer blocks. Of the D candidate blocks, the algorithm fetches the K blocks with the earliest time-stamps from the disks; of the remaining $D - K$ blocks, it fetches a block only if there is a block in the buffer with a later time-stamp. In this case the buffered block with the later time-stamp is evicted.

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.

Whenever $|M_t| \leq M - D$, we read in all D blocks of the set S_t. When $|M_t| > M - D$, we flush 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. 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.

The flush operation by itself does not involve any I/O. Hence the forecasting data structure and the technique of flushing yield a simple and efficient implementation of global M-block lookahead for read-once reference strings. Overall, our definition of global M-block lookahead is motivated by the ease of implementing such lookahead using a local lookahead together with some global information across disks.

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 vital for prefetching.

REFERENCES

1. R. D. Barve, E. F. Grove, and J. S. Vitter, Simple randomized mergesort on parallel disks, *Parallel Computing* **23**, No. 4 (June 1997), 601–631.
2. L. A. Belady, A study of replacement algorithms for virtual storage, *IBM Systems J*. **5** (1966), 78–101.
3. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, RAID: High performance reliable secondary storage, *ACM Comput*. *Surveys* **26**, No. 2 (1994), 145–185.
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, Oct. 1996."
7. D. E. Knuth, "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, MA, 1973.
8. V. S. Pai, A. A. Schäffer, and P. J. Varman, Markov analysis of multiple-disk prefetching strategies for external merging, *Theoret. Comput. Sci.* **128**, Nos. 1–2 (June 1994), 211–239.
9. D. D. Sleator and R. R. Tarjan, Amortized efficiency of list update and paging rules, *Comm. ACM* **28**, No. 2 (Feb. 1985), 202–208.
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, February 1996," pp. 645–654.
11. P. J. Varman and R. M. Verma, Tight bounds for prefetching and buffer management algorithms for parallel I/O systems, *IEEE Trans. Parallel Distributed Systems* **10**, No. 12 (Dec. 1999), 1262–1275.
12. J. S. Vitter and E. A. M. Shriver, Optimal algorithms for parallel memory. I. Two-level memories, *Algorithmica* **12**, Nos. 2–3 (1994), 110–147.
13. W. Zheng and Per-Åke Larson, A memory-adaptive sort (MARSORT) for database systems, *in* "CASCON'96, Nov. 1996."