

# 1 RANDOMIZED PARALLEL PREFETCHING AND BUFFER MANAGEMENT

Mahesh Kallahalla  
Peter J. Varman

Department of Electrical and Computer Engineering  
Rice University  
Houston TX 77005 \*

**Abstract:** There is increasing interest in the use of multiple-disk parallel I/O systems to alleviate the I/O bottleneck. Effective use of I/O parallelism requires careful coordination between data placement, prefetching and caching policies. We address the problems of I/O scheduling and buffer management in a parallel I/O system. Using the standard parallel disk model with  $D$  disks and a shared I/O buffer of  $M$  blocks, we study the performance of on-line algorithms that use bounded lookahead.

We first discuss algorithms for read-once reference strings. It is known (see [3]) that any deterministic prefetching algorithm with either global  $M$ -block or local lookahead, must perform a significantly larger number of I/Os than the optimal off-line algorithm. We discuss several prefetching schemes based on a randomized data placement, and present a simple prefetching algorithm that is shown to perform the minimum (up to constants) expected number of I/Os.

For general read-many reference strings, we introduce the concept of *write-back* whereby blocks are relocated between disks during the course of the computation. We show that any on-line algorithm with bounded lookahead using deterministic write-back and buffer management policies must have a competitive ratio of  $\Omega(D)$ . We therefore present a randomized algorithm, RAND-WB, that uses a novel randomized write-back scheme. RAND-WB attains a competitive ratio of  $\Theta(\sqrt{D})$ , which is the best achievable by any on-line algorithm with only global  $M$ -block lookahead.

\*Research partially supported by the National Science Foundation under grant CCR-9704562 and a grant from the Schlumberger Foundation.

## 1.1 INTRODUCTION

Continuing advances in processor architecture and technology have resulted in the I/O subsystem becoming the bottleneck in many applications. The problem is exacerbated by the advent of multiprocessing systems which can harness the power of hundreds of processors in speeding up computation. Improvements in I/O technology are unlikely to keep pace with processor and memory speeds, causing many applications to choke on I/O. The increasing availability of cost-effective multiple-disk storage systems [7] provides an opportunity to improve the I/O performance through the use of parallelism. However it remains a challenging problem to effectively use the increased disk bandwidth to reduce the I/O latency of an application. Effective use of I/O parallelism requires careful coordination between data placement, prefetching and caching policies.

The parallel disk model of Vitter and Shriver [21] consists of  $D$  independently accessible disks and an associated I/O buffer with a capacity of  $M$  blocks. The buffer is shared by all the disks. The data for the computation is stored on the disks in blocks; a block is the unit of access from a disk. In each parallel I/O up to  $D$  blocks, at most one from each disk, can be read from or written to the I/O subsystem. From the viewpoint of the I/O, the computation is characterized by a *reference string* consisting of the ordered sequence of blocks that the computation accesses. A block should be present in the I/O buffer before it can be accessed by the computation. Serving the reference string requires performing I/O operations to provide the computation with blocks in the order specified by the reference string. In this model the measure of performance of the system is the number of parallel I/Os required to service a given reference string.

Classical buffer management has been studied extensively in a sequential I/O model [1, 4, 5, 6, 9, 16, 19]. These works primarily deal with developing efficient buffer management algorithms for a single-disk system, by optimizing decisions regarding the blocks to be evicted from the buffer. The use of information about future accesses, *lookahead*, to improve the eviction decisions made by on-line algorithms for single-disk systems was studied in [1] and [5], using different models of lookahead. The overlap of cpu and I/O operations in a single-disk system was addressed in [6], and off-line approximation algorithms were presented and analyzed. In the scenario of parallel I/O studied here, several new issues (discussed in Section 1.1.1) arise, precluding any straightforward extensions of the algorithms for single-disk systems to the parallel situation. In [3] the question of designing on-line prefetching algorithms for parallel I/O systems using bounded lookahead was addressed. Fundamental bounds on the performance of algorithms were presented for an important, albeit restricted, class of reference strings called read-once reference strings. However, the problem of general reference strings in which blocks can be repeatedly accessed, called read-many reference strings, was not considered. For read-many reference strings an optimal off-line buffer management and scheduling algorithm was presented in [20] for a distributed-buffer parallel I/O model in which each disk has its own private buffer. An interesting alternate measure of perfor-

mance that has been proposed is the elapsed or stall time which includes the time required to consume a block as an explicit parameter [6, 12]. An off-line approximation algorithm for parallel I/O scheduling and buffer management in this model was presented in [12]. However, so far the question of devising an on-line algorithm with bounded lookahead for general read-many reference strings in a parallel I/O model has not been addressed.

In this article we study the on-line I/O scheduling problem in the framework of competitive analysis. In this framework, the measure of performance of an on-line algorithm is the competitive ratio (defined formally in Section 1.2). Informally, this ratio measures how well a given on-line algorithm performs compared to the optimal off-line algorithm; the latter has access to the entire reference string and constructs its schedule using some off-line optimization strategy. We first discuss algorithms for read-once reference strings, followed by consideration of general read-many reference strings. We introduce the concept of *write-back*, whereby blocks are dynamically relocated between disks during the course of the computation. Using global  $M$ -block lookahead (defined in Section 1.2) and randomized write-back, we design an on-line buffer management and scheduling algorithm, RAND-WB, whose competitive ratio matches that of the *best on-line* algorithm which uses only the same amount of lookahead. For worst-case reference strings, the expected number of I/Os performed by RAND-WB is shown to be within  $\Theta(\sqrt{D})$  times the number of I/Os done by the optimal off-line algorithm. In contrast *any* scheduling algorithm with the same lookahead, that uses only deterministic rather than randomized policies is shown to require  $\Omega(D)$  times as many I/Os as the optimal off-line algorithm in the worst case.

The rest of the article is organized as follows. In Section 1.1.1 we intuitively discuss some basic differences in parallel and sequential I/O. Formal definitions of the parallel disk and lookahead models, and the terms used are presented in Section 1.2. We summarize the main results of this work in Section 1.2.1. Bounds on algorithms for read-once reference strings and the use of randomization in this context are presented in Section 1.3. Read-many reference strings are considered in Section 1.4. In Section 1.4.1 we introduce the concept of write-back and derive a lower bound of  $\Omega(D)$  on the competitive ratio of any deterministic algorithm using global  $M$ -block lookahead. Finally, an algorithm which uses randomized write-back and achieves a competitive ratio of  $\Theta(\sqrt{D})$  is presented in Section 1.4.2.

### 1.1.1 Performance Issues in Parallel I/O

In the sequential I/O model, the measure of performance is the total number of I/Os performed. However in the parallel I/O case the appropriate measure is the total number of parallel I/Os performed, as more than one block can be fetched in parallel. The potential for overlapped accesses raises new issues that make the problem of minimizing the number of parallel I/Os challenging.

**Prefetching.** In the sequential model blocks are fetched on-demand; that is, an I/O for a block is initiated only when the block is requested by the computation. It is well known that early fetching cannot reduce the number of I/Os needed<sup>1</sup> in the single-disk model [19]. In a parallel I/O system doing all I/Os only on-demand is wasteful of the available I/O bandwidth, since only one block will be fetched in any I/O operation. Disk parallelism can be obtained by prefetching blocks from disks that would otherwise idle, concurrently with a demand I/O. In order to prefetch accurately, the computation must therefore be able to look ahead in the reference string, beyond the last referenced block. Parallel prefetching for specific applications has been studied in [2, 8, 14, 15, 17], for example.

**Choice of blocks to fetch on an I/O.** In the sequential model blocks are always fetched strictly in order of the reference string. Interestingly, in the parallel model fetching blocks in order of their appearance in the reference string can be inefficient. For instance, consider the examples of Figures 1.1a and 1.1b which assume  $D = 3$  and  $M = 6$ . Assume that blocks labeled  $A_i$  (respectively  $B_i, C_i$ ) are placed on disk 1 (respectively 2, 3), and that the reference string  $\Sigma = A_1 A_2 A_3 A_4 B_1 C_1 A_5 B_2 C_2 A_6 B_3 C_3 A_7 B_4 C_4 C_5 C_6 C_7$ . For the example we assume that a parallel I/O is initiated only when the referenced block is not present in the buffer. The schedule in Figure 1.1a is 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 demand block  $A_1$ . 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 1.1b, 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 1.1a** Scheduling  $\Sigma$  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 1.1b** Scheduling  $\Sigma$  out of order

**Replacement Policy.** In the parallel I/O model, choosing a block to evict is complicated because of two reasons: the need for parallelism and the use of prefetching. It is well known that the replacement policy that evicts the block from buffer whose next reference is the farthest (known as the MIN algorithm [4]) minimizes the total number of I/Os done in the sequential model. In a parallel system this is not a sufficient criterion. The eviction decision is influenced by the potential parallelism with which blocks can be read again; that is, it may be better to evict a block even though it increases the total number of blocks fetched, if it permits greater parallelism. Secondly, there is an intrinsic tension between the need to increase parallelism by prefetching and the desire to delay the fetch as late as possible in order to obtain the best possible candidate for eviction. For instance consider the following example with  $D = 3$  and  $M = 6$ , where blocks  $A_i$  (respectively  $B_i, C_i$ ) are placed on disk 1 (respectively 2, 3). Suppose that at some point the buffer contains  $A_1, A_2, A_3, B_1, B_2, C_1$ , and the remainder of the reference string consists of the subsequence:  $\Sigma^* = A_4 B_3 C_2 A_4 B_3 B_2 B_1 C_1 A_1 A_2 A_3$ . Figure 1.2a shows the I/O schedule obtained by using the same policy as MIN (known to be optimal for a single-disk) to determine evictions. To fetch  $A_4$  the algorithm will evict  $A_3$  that is referenced later than all the other blocks in buffer.  $B_3$  and  $C_2$  are prefetched along with  $A_4$ , evicting blocks  $A_2$  and  $A_1$ <sup>2</sup>. The buffer now has blocks  $A_4 B_3 C_2 B_1 B_2 C_1$ , and the computation proceeds till block  $A_1$  is referenced. Three more I/Os, fetching blocks  $A_1, A_2$  and  $A_3$  respectively, are required to complete the schedule. In contrast Figure 1.2b shows a schedule that takes only 2 rather than 4 steps. This is obtained by evicting blocks  $A_1, B_1$  and  $C_1$  (instead of  $A_1, A_2$  and  $A_3$ ) at the first step. When the computation references  $B_1$  these blocks are read back in just one parallel I/O.

|        |       |       |       |       |
|--------|-------|-------|-------|-------|
| Disk 1 | $A_4$ | $A_1$ | $A_2$ | $A_3$ |
| Disk 2 | $B_3$ |       |       |       |
| Disk 3 | $C_2$ |       |       |       |

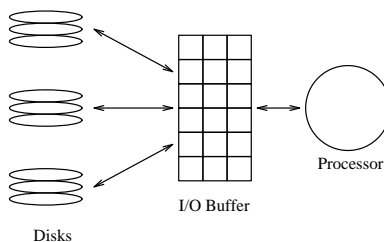
Figure 1.2a Schedule using MIN

|        |       |       |
|--------|-------|-------|
| Disk 1 | $A_4$ | $A_1$ |
| Disk 2 | $B_3$ | $B_3$ |
| Disk 3 | $C_2$ | $C_1$ |

Figure 1.2b Alternative Schedule

## 1.2 DEFINITIONS

We use the Parallel Disk Model [21] of a parallel I/O system. This model consists of  $D$  independently accessible disks and a shared I/O buffer capable of holding  $M$  ( $M \geq 2D$ ) data blocks. In one parallel I/O step up to  $D$  accesses, at most one on any disk, can proceed in parallel. The measure of performance is the number of parallel I/Os required to service a given sequence of I/O requests. This model uses the I/O time as the measure, and looks to the overlap of operations on different disks as the primary method of performance improvement. Accesses are modeled by a reference string. We consider both read-once and read-many reference strings. There is no restriction on the blocks referenced



**Figure 1.3** Parallel Disk Model with a Shared I/O Buffer

in read-many reference strings, in contrast to read-once reference strings where all the requests are to distinct blocks.

**Definition 1.** The ordered sequence of blocks read by the computation is called the *reference string*. In a *read-once* reference string all the references are to distinct blocks. In a *read-many* reference string any two references can be to the same data block.

Read-once reference strings arise naturally and frequently in I/O-bound applications running on parallel-disk systems: external merging and mergesorting (including carrying out several of these concurrently) and real-time retrieval and playback of multiple streams of multimedia data such as compressed video and audio. In a read-many reference string the accesses are still read-only, but there is no restriction placed on the frequency of accesses to a block. The main difference between the problems of serving read-once reference strings and read-many reference strings is that in the latter, buffer management plays an important role in determining the performance. In the read-once case a block can be evicted from the buffer as soon as it is referenced. However, in the read-many case a data block can be referenced several times, and the buffer manager may find it useful to retain it in the I/O buffer even after a request for it has been serviced. As noted earlier, the choice of block to evict is influenced by the potential parallelism with which it can be read again.

In order to perform accurate rather than speculative prefetching it is necessary to have some knowledge of the future requests to be made to the I/O system, beyond the current reference. This knowledge of future accesses is embodied in the notion of lookahead. In sequential I/O systems lookahead was used to help the buffer management algorithm in making eviction decisions [1, 5]. In parallel systems lookahead is needed for making prefetching decisions independent of its use in aiding evictions. In [3], two models for lookahead were defined in the context of read-once reference strings: global and local. Our algorithm RAND-WB uses global  $M$ -block lookahead defined below for read-many reference strings. Such a lookahead is also called  $M$ -block strong lookahead in the model of [1]. There has been substantial interest in obtaining such lookahead information for prefetching from applications using combinations of programmer hints and program analysis [18]. Intuitively,

global  $M$ -block lookahead gives the next buffer-load of distinct requests to the buffer management algorithm. This information can aid both prefetching and caching.

**Definition 2.** An I/O scheduling algorithm has *global  $M$ -block lookahead* if it knows the portion of the reference string containing the next  $M$  distinct blocks, beyond the current reference. For a read-once reference string this lookahead spans exactly the next  $M$  blocks in the reference string.

The second form of lookahead we consider, local lookahead, is particularly appropriate for read-once reference strings. This lookahead models stream data where each stream is associated with a separate disk. The data within each stream is accessed in a predictable fashion but the relative order of accesses across streams is unknown. In this model lookahead information is localized to each disk independently. For each disk we only know the sequence of accesses from that disk up to and including the first block from that disk which is not present in the buffer. In [2, 11] it is shown how this information can be obtained in an on-line fashion in applications like external merging and video servers, by implanting a small amount of information in the stored data blocks.

**Definition 3.** An I/O scheduling algorithm has *local lookahead* if, for each disk  $d$ , it knows the sequence of accesses from disk  $d$  up to the first block from  $d$  not present in the buffer.

As an example consider the an I/O system with  $D = 2$  disks, containing blocks  $A_i$  and  $B_i$  respectively, and a buffer of size  $M = 4$ . Suppose that at some stage, the buffer contains blocks  $A_1, B_1, B_2, A_2$  and the remainder of the reference string is  $\Sigma^* = A_2B_1B_2B_3A_3$ . Global  $M$ -block lookahead provides the sub-string  $A_2B_1B_2B_3$  to the algorithm, while local lookahead provides the two strings  $A_2A_3$  and  $B_1B_2B_3$  to the algorithm. In particular local lookahead does *not* provide any information regarding the relative order of blocks *across* disks.

Since the algorithms we consider use a bounded amount of lookahead we refer to them as on-line algorithms. In contrast, off-line algorithms base their decisions on the entire reference string. To quantify the performance of an algorithm we use the competitive ratio which is a measure of how well it performs relative to the optimal off-line algorithm.

**Definition 4.** An on-line parallel prefetching algorithm  $\mathcal{A}$  has a competitive ratio of  $C_A$  if for any reference string the number of I/Os that  $\mathcal{A}$  requires is within a factor  $C_A$  of the number of I/Os required by the optimal off-line algorithm serving the same reference string. If  $\mathcal{A}$  is a randomized algorithm then the expected number of I/Os done by  $\mathcal{A}$  is considered.

### 1.2.1 Summary of Results

We demonstrate quantitatively the benefits of randomization in prefetching and buffer management algorithms for multiple-disk parallel I/O systems. We consider the case of read-once and read-many reference strings separately. For the

read-once case it was shown in [3] that any deterministic prefetching algorithm with either global  $M$ -block or local lookahead must perform significantly more I/Os than the optimal off-line that has access to the entire reference string. We discuss several prefetching schemes based on a randomized data placement, and present a simple prefetching algorithm that is shown to perform the minimum (up to constants) expected number of I/Os. For the case of general read-many strings we introduce the concept of *write-back*, whereby blocks are dynamically relocated between disks during the course of the computation. We note that buffer management algorithms that perform no write-back can pay a significant I/O penalty when compared to algorithms that do. However deciding which blocks to relocate and how to move them is not easy to do in an on-line manner. We show that *any deterministic* buffer management and scheduling algorithm with global  $M$ -block lookahead must have a competitive ratio of  $\Omega(D)$ . That is, any strategy that is based solely on the bounded lookahead and the past behavior of the algorithm, can in the worst case serialize its disk accesses.

We use *randomization* to improve buffer management and scheduling decisions. In particular, we show that by employing randomization in the relocation, the competitive ratio can be improved to  $\Theta(\sqrt{D})$  rather than the  $\Omega(D)$  necessary for any deterministic strategy. We present a randomized algorithm, RAND-WB, that uses a novel randomized write-back scheme, and attains the lowest possible competitive ratio of  $\Theta(\sqrt{D})$ . As a corollary we show that if initially all the data blocks have been randomly placed on disks, the competitive ratio of RAND-WB is  $\Theta(\log D)$ .

### 1.3 READ-ONCE REFERENCE STRINGS

For read-once sequences, we first consider a worst-case model wherein a deterministic placement algorithm is used to place each block of the read-once sequence on a disk. Since no block is referenced more than once, and the 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. Yet, counter to intuition, we have the interesting result (Theorem 1) that there are read-once reference strings 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.

**Theorem 1** [3] *The competitive ratio of any deterministic on-line algorithm having global  $M$ -block lookahead is at least  $\Omega(\sqrt{D})$ .*

In the case of local lookahead the prefetching algorithm has no access to any information regarding the relative order in which blocks originating from different disks are consumed. It turns out that this is a very powerful advantage for an adversary who can force a much higher lower bound on the competitive ratio of such on-line algorithms. Theorem 2 shows that *any deterministic algorithm using only local lookahead can perform  $\Omega(D)$  times worse than the optimal off-line algorithm.* Note that for read-once strings it is trivial to de-

sign an algorithm with a competitive ratio of  $\Theta(D)$ , merely by performing all I/Os on demand. Thus in a deterministic setting, local lookahead is practically useless in speeding up the I/O.

**Theorem 2** [3] *The competitive ratio of any deterministic on-line algorithm having local lookahead is at least  $\Omega(D)$ .*

In order to improve the performance of prefetching algorithms using bounded lookahead, a randomized placement algorithm is employed. In a deterministic placement scheme the predictability of the decisions made by the prefetching algorithm can be exploited by an adversary to limit the performance significantly. By randomizing the placement it becomes difficult for the adversary to defeat the prefetching algorithm. It is possible to design simple prefetching algorithms which significantly improve the parallelism attainable over deterministic placement schemes. For our randomized algorithms we require that each block of the reference string be placed on any disk with uniform probability  $1/D$ . Implementations of such randomized placement schemes and associated data structures to obtain the desired lookahead in an on-line manner for applications like external merging and video servers, are discussed in [2] and [11], for instance.

#### Algorithm GREED

GREED uses local lookahead. Data is placed on disks so that each block independently has equal probability of being placed on any disk. The shared buffer is partitioned into  $D$  logical buffers of size  $M/D$  blocks each; each logical buffer is associated with a single disk. GREED builds a schedule as follows:

1. If the requested block is present in the buffer the request is serviced without any further action.
2. If the requested block is not present in the buffer a parallel I/O is initiated. The blocks fetched depends upon availability of buffer space.
  - (a) From each disk, the next block not in buffer is fetched provided there is buffer space available in the logical buffer for that disk.
  - (b) If there is no buffer space for a particular disk then no block is read from that disk.

Algorithm GREED uses a simple prefetching method based on local lookahead. In Theorem 3 we show that GREED performs  $\Theta(N/D)$  expected number of parallel I/Os to service a reference string of  $N$  blocks. Since any algorithm must require at least  $N/D$  I/Os to fetch  $N$  distinct blocks, GREED is within a constant factor of the optimal.

**Theorem 3** [11] *To service a reference string of length  $N$ , GREED using a buffer of size  $M = \Omega(D \log D)$ , incurs  $\Theta(N/D)$  expected number of I/Os.*

To bound the performance of GREED, it is useful to consider its behavior over a substring of the reference string. Define phase  $i$ ,  $phase(i)$ , to be the substring  $\langle b_j, b_{j+1}, \dots, b_k \rangle$  of the reference string, such that (a) 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 less than  $M/D$  blocks from any other disk. Note that  $phase(1)$  starts with block  $b_1$ . From this definition of a phase, it is easy to see that:

**Claim 1** *At most  $M/D$  I/Os are needed to service the requests in any phase.*

Hence to estimate the total number of I/Os done to service the reference string, we first estimate the total number of phases in the reference string. This 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,  $\mathcal{B}$ , 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 logical disk buffer with an urn of capacity  $C = M/D$ , and block references with throws of a ball, we see that  $\mathcal{B}$  is the average number of blocks referenced in a phase. The above problem has been studied in [10] to give the following expression for  $\mathcal{B}$ .

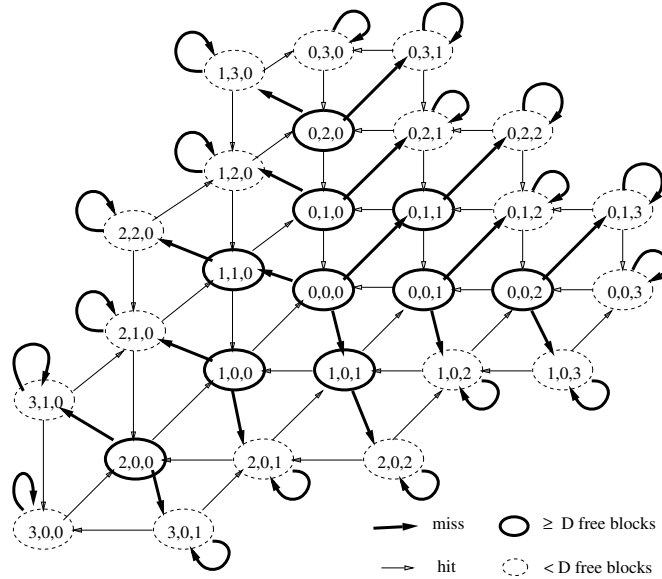
$$\mathcal{B} \geq D^{1-\frac{1}{C}} (C!)^{\frac{1}{C}} \Gamma(1 + C^{-1})$$

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  $\mathcal{B} = \Omega(CD/(D^{\frac{1}{C}}))$ . The expected number of phases is no more than  $N/\mathcal{B} + 1$  and each phase requires at most  $M/D$  I/Os. Hence the expected number of I/Os is  $O(ND^{\frac{1}{C}}/D)$ . When  $C = \Omega(\log D)$  the total number of I/Os is  $\Theta(N/D)$ , which is asymptotically optimal.

Note that GREED effectively partitions the shared buffer into  $D$  fixed parts of size  $M/D$  blocks each. It is interesting to consider the role of partitioning the shared buffer in GREED. It may appear that by statically partitioning the buffer the potential parallelism is being unnecessarily restricted. An alternative would be to maintain just a single buffer of  $M$  blocks and fetch from all disks whenever there are  $D$  or more buffer blocks available. If less than  $D$  blocks are available, then only the demand block is fetched. Denote this algorithm by GREED\*. Surprisingly, Theorem 4 shows that the performance of GREED\* which uses an unpartitioned shared buffer can be actually worse.

**Theorem 4** [17] *To service a reference string of length  $N$ , GREED\* incurs  $\Theta(N/D)$  expected number of I/Os, provided the buffer is of size  $M = \Omega(D^2)$ .*

The theorem was proved by modeling the dynamics of the buffer using a Markov chain. Each state in the chain is a  $D$ -tuple  $[p_1, p_2, \dots, p_D]$ , where  $p_i$  is the number of prefetched blocks from disk  $i$  in the buffer. State transitions correspond to the buffer changes following a reference to a block. An example of the chain in a 3-disk system, with a buffer of size 5 is presented in Figure 1.4. Consider the state  $[0, 1, 1]$ . It has three outgoing transitions, each with probability  $1/3$



**Figure 1.4** The Markov chain with  $D = 3$  disks and  $M = 5$  buffer blocks.

since the next reference can be from any of the disks with equal probability. If the next block referenced is from disk 2 or 3 a transition is made to  $[0, 0, 1]$  or  $[0, 1, 0]$ , respectively. If the block referenced is from the first disk an I/O is required, since the first disk has no prefetched blocks. Since  $[0, 1, 1]$  has 2 free buffer blocks available for prefetch, the demand block from disk 1 is fetched along with prefetches from disks 2 and 3; this is shown by the transition leading to state  $[0, 2, 2]$ . However when the system is in state  $[2, 1, 0]$ , the buffer has only 1 free block available for prefetch. Hence in this case the reference of a block from the third disk results in only the demand block from disk 3 being fetched, forcing a transition back to  $[2, 1, 0]$ . This transition also has probability  $1/3$ . Finally, when in a state like  $[0, 0, 3]$  the system has a probability  $2/3$  of remaining in the same state since the next reference can be from either disk 1 or disk 2 with probability  $2/3$ .

Solving for steady-state probabilities of the states, the expected number of blocks fetched in any I/O can be shown to be [17]:

$$1 + \frac{D - 1}{2 - D + M [H_{M-1} - H_{M-D}]}$$

where  $H_k$  is the  $k$ th Harmonic number. This expression simplified when  $M = \Omega(D^2)$  gives Theorem 4.

Finally, we consider an algorithm, NOM, that uses global  $M$ -block lookahead. NOM behaves like GREED\* except that only blocks within the current  $M$ -block lookahead are prefetched. The proof of the theorem follows by using the analysis in [2] for a related problem. NOM has performance similar

to GREED for  $M = \Omega(D \log D)$ . However for smaller buffer sizes the relative order of performance of the algorithms considered is NOM followed by GREED followed by GREED\*.

**Theorem 5** [2] *To service a reference string of length  $N$ , NOM using a buffer of size  $M = \Omega(D \log D)$ , incurs  $\Theta(N/D)$  expected number of I/Os.*

## 1.4 READ-MANY REFERENCE STRINGS

In a read-many reference string, a single data block can be requested more than once by the computation. Hence a situation may arise wherein a particular data layout strategy may be favorable for data accesses occurring in one section of the reference string but unfavorable for accesses in other sections. One way to tackle this problem is to relocate data blocks dynamically so as to have a favorable data placement during the next set of accesses. The underlying intuition is to rearrange the layout so that blocks which are evicted may be fetched in parallel with other blocks in the future. Of course, writing a block out to a different disk, other than the one on which it currently resides, incurs the cost of writing out a block. But the gain in I/O parallelism as a result of this relocation can be used to offset the extra cost in performing the write. We refer to this action of writing an evicted block to a disk, different from the one from which it was fetched, as *write-back*.

### 1.4.1 Lower Bound on Deterministic Algorithms

Write-back allows the location of a data block to be dynamically altered as the requests in the reference string are serviced. However, there is only one copy of the block on the disks at any time. A block is said to *reside* on disk  $d$  if the only copy of the block in the I/O system is on disk  $d$ . We next introduce the notion of a simple deterministic algorithm (SDA) which captures the intuition behind most existing buffer management algorithms. We shall show that, in the worst case, such algorithms may be ineffective in exploiting the latent I/O parallelism.

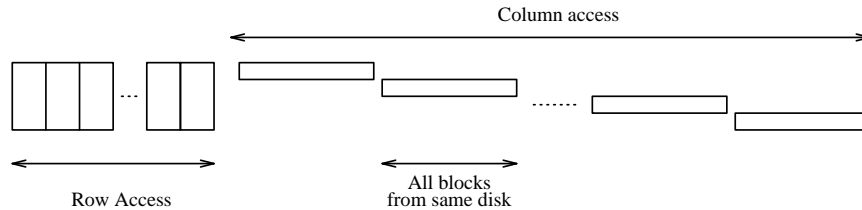
**Definition 5.** Let the set of blocks in the I/O buffer after the first  $k$  requests,  $\langle r_1, r_2, \dots, r_k \rangle$ , in the reference string are serviced be  $\mathcal{B}_k$ . At this stage, let the lookahead window be  $\mathcal{L}_k$ . An algorithm is said to be a *simple deterministic algorithm* (SDA) if at this time the set of blocks that it prefetches, the set of blocks that it evicts from the buffer and the disks to which it writes back these evicted blocks can be uniquely determined by specifying  $\mathcal{B}_k$ ,  $\langle r_1, r_2, \dots, r_k \rangle$  and  $\mathcal{L}_k$ .

The above definition underscores the fact that a SDA uses *deterministic* policies to decide which blocks to prefetch, which blocks to evict and where to write-back the evicted blocks. This determinism can be exploited by an adversary to generate reference strings which require the SDA to make a large number of I/Os. In this section, we first analyze the performance of simple

deterministic algorithms with global  $M$ -block lookahead and show that the competitive ratio of such algorithms is  $\Omega(D)$ .

As a simple illustration, let us consider an algorithm which does not perform write-back and uses deterministic buffer management policies. Let the algorithm have global  $M$ -block lookahead and base its decisions according to the following rules: (a) do not evict any block from the buffer that is in the current lookahead. (b) greedily prefetch blocks in the lookahead window.

Now consider the performance of the above algorithm while servicing the requests of an application which accesses a matrix of size  $3M \times D$  blocks in two stages: first in row major order and then in column major order. The matrix is assumed to be initially laid out on disk with the blocks of each row striped across all  $D$  disks. Assume  $M = \Omega(D)$ . Tracing through the algorithm we can note that the I/Os during the first stage are completely parallelized. However, in the second stage there is very little parallelism when the columns of the matrix are accessed; the first  $M - 1$  blocks of column  $i$  are prefetched along with the last  $M - 1$  blocks of column  $i - 1$ . Figure 1.5 depicts the structure of the entire reference string as seen by the algorithm. The algorithm makes



**Figure 1.5** Illustration of matrix accesses as seen by the deterministic algorithm

$3M$  I/Os during the first stage and  $3M + (2M + 1)(D - 1)$  I/Os in the second. Hence the total number of I/Os performed by the algorithm to service all the I/O requests is  $\Theta(MD)$ .

In contrast consider an algorithm which works just like the one presented except for the write-back policy; it writes back in such a way that all blocks which originated from the same disk are now striped across all disks. This can be done by writing back row  $i$ ,  $1 \leq i \leq 3M$ , in a stripe starting at disk  $(i - 1) \bmod D + 1$ . The cost of writing back during the row access stage is  $3M$  I/Os. However the benefit is that all future I/Os can be parallelized. Hence the total number of I/Os performed by this algorithm is  $6M$  (counting the reads and writes) in the first stage and  $3M$  in the last stage. The ratio is clearly  $\Omega(D)$ .

The on-line algorithm in the above example did not use any write-back. However even in a more general case, when the algorithm uses an arbitrary deterministic write-back policy along with more sophisticated prefetching and block replacement heuristics, we can construct a reference string for which the algorithm must serialize its I/Os significantly. This intuition is formalized in Theorem 6 which gives a lower bound on the competitive ratio of any simple de-

terministic algorithm. We shall prove the theorem by considering an arbitrary on-line algorithm  $\mathcal{A}$  using global  $M$ -block lookahead. Based on the behavior of  $\mathcal{A}$  we construct a reference string, the nemesis string, for which  $\mathcal{A}$  requires  $\Omega(MD)$  I/Os (Lemma 2). We then show an alternative off-line schedule which services the same reference string in  $\Theta(M)$  I/Os (Lemma 3), thereby showing that the ratio between the two is  $\Omega(D)$ .

The reference string constructed has two stages. In the first stage  $M(D+1)$  distinct blocks are referenced. In the second stage a total of  $2MD$  blocks are referenced; of these  $MD/2$  blocks were also referenced in the first stage. These  $MD/2$  blocks are chosen based on the write-back employed by algorithm  $\mathcal{A}$  so that it will be forced into sequentializing accesses to these blocks. Thus it will perform  $\Omega(MD)$  I/Os in the second stage (Lemma 2). On the other hand, the off-line algorithm reads in these  $MD/2$  blocks at the end of the first stage, and writes them out in a way that accesses to them can be parallelized completely. Overall the off-line algorithm will perform  $\Theta(M)$  reads and  $\Theta(M)$  writes (Lemma 3). The detailed proofs follow.

**Proof of Lower Bound.** To aid in the analysis it is useful to define the notion of a *phase*, which is a sub-sequence of the reference string.

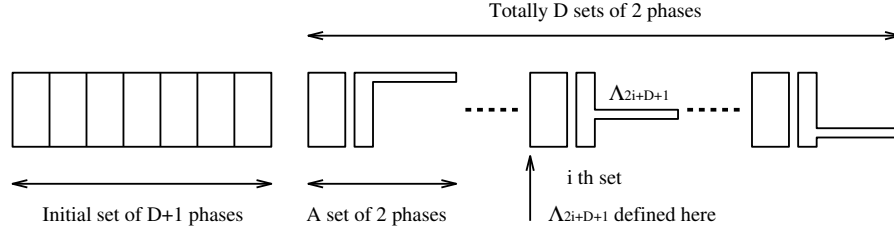
**Definition 6.** A phase is a maximal length sub-string of the reference string containing references to exactly  $M$  distinct blocks. The first request belongs to the first phase; and every request in the reference string belong to exactly one phase. The  $i$ th phase,  $i \geq 1$ , is denoted by  $phase(i)$ .

The *start of a phase* refers to the instant when the first block of the phase is referenced. Similarly, the *end of a phase* refers to the instant when the last block of a phase has been serviced. The following definitions will be useful in characterizing the set of blocks which are accessed in a phase. These definitions are based solely upon the reference string and are independent of the algorithm used to service it.

**Definition 7.** The set of *clean blocks*,  $C_i$ , in  $phase(i)$  is defined as the set of all blocks in  $phase(i)$  not requested in the previous phase,  $phase(i-1)$ . The set of *stale blocks*,  $S_i$ , in  $phase(i)$  is defined as the set of blocks in  $phase(i)$  requested in the previous phase,  $phase(i-1)$ . The set of *new blocks*,  $N_i$ , in  $phase(i)$  is defined as the set of blocks in  $phase(i)$  not requested in any prior phase,  $phase(j)$ ,  $1 \leq j < i$ . The set of *reuse blocks*,  $R_i$ , in  $phase(i)$  is defined as the set of clean blocks in  $phase(i)$  that have been requested in some previous phase,  $phase(j)$ , for  $j < i-1$ . The numbers of clean, stale, new and reuse blocks in  $phase(i)$  are denoted by  $c_i$ ,  $s_i$ ,  $n_i$  and  $r_i$  respectively.

Let  $\mathcal{A}$  denote a simple deterministic algorithm with global  $M$ -block lookahead. We describe the construction of a reference string, which will be used to give a lower bound on the performance of  $\mathcal{A}$ . Let us construct a reference string  $\eta$ , consisting of  $M(3D+1)$  references. Reference strings of arbitrary length for which the proofs will follow can be constructed by repeating  $\eta$ . Figure 1.6 il-

illustrates the structure of the constructed reference string as seen by algorithm  $\mathcal{A}$ . The details of the construction are as follows:



**Figure 1.6** Structure of a worst case reference string for  $\mathcal{A}$

#### Construction of nemesis reference string $\eta$

1. The first  $D + 1$  phases of  $\eta$  consist of  $M(D + 1)$  new blocks which are striped across all disks. Let  $F$  denote this set of  $M(D + 1)$  blocks.
2. The last  $2D$  phases are constructed in sets of two phases each. The  $i$ th,  $0 \leq i < D$ , set of two phases is constructed as follows.

- The first phase of the set,  $phase(2i + D + 2)$ , consists of  $M$  new blocks striped across all disks.
- The next phase,  $phase(2i + D + 3)$ , is made of two parts. The first part consists of  $M/2$  new blocks striped across all disks. The second part is a sequence of  $M/2$  blocks given by  $\Lambda_{2i+D+1}$ , where  $\Lambda_{2i+D+1}$  is determined as follows.

Let  $k = 2i + D + 1$ . Let  $F_k$  denote the set of all blocks from  $F$  which have been referenced exactly once till the end  $phase(k)$ . Let  $A_{k,j}$ , be the set of blocks from  $F_k$ , residing on disk  $j$  at the end of  $phase(k)$ ; let  $B_{k,j}$ , subset of  $A_{k,j}$ , be the set of all such blocks in the buffer. Then  $height(k) = \max_j \{|A_{k,j} - B_{k,j}|\}$ , is the maximum number of blocks from  $F_k$ , residing on the same disk and not in the buffer. A disk  $d$ , such that  $|A_{k,d} - B_{k,d}| = height(k)$  is called the *max disk*. The sequence  $\Lambda_k$ , can now be defined as the ordered sequence of  $M/2$  earliest referenced blocks in the set  $A_{k,d} - B_{k,d}$ , from a max disk,  $d$ .

First we show that for the reference string  $\eta$ ,  $height(2i + D + 1) \geq M/2$ , for all  $0 \leq i < D$ . This will ensure that  $\Lambda_{2i+D+1}$  is well defined for all  $0 \leq i < D$  thereby allowing the construction of the sequence  $\eta$  as above.

**Lemma 1** *With respect to algorithm  $\mathcal{A}$ , and the reference string  $\eta$ ,  $height(2i + D + 1) \geq M/2$ , for  $0 \leq i < D$ .*

**Proof :** In the first  $D + 1$  phases, a total of  $M(D + 1)$  distinct blocks are referenced exactly once. However at the end of  $phase(D + 1)$ , there are at most  $M$  blocks in the buffer; hence  $height(D + 1) \geq M$ . This allows the construction of  $\eta$  till  $phase(D + 3)$ .

Now, inductively, if  $\eta$  has been constructed till  $phase(2i + D + 1)$ ,  $0 \leq i < D - 1$ , we shall show that  $height(2i + D + 1) \geq M/2$ . This will then allow the construction of  $\eta$  till  $phase(2i + D + 3)$ . By construction and the inductive hypothesis,  $Mi/2$  blocks from  $F$  have been referenced twice at the end of  $phase(2i + D + 1)$ . Hence  $|F_{2i+D+1}| = M(D + 1) - Mi/2$ . In addition, there are at most  $M$  blocks in the buffer at the end of  $phase(2i + D + 1)$ . Hence  $|\Lambda_{2i+D+1}| \geq (M(D + 1) - M - MD/2)/D$ ; that is,  $height(2i + D + 1) \geq M/2$ .  $\square$

Now we shall count the number of I/Os performed by algorithm  $\mathcal{A}$  to schedule  $\eta$ . The lower bound on its performance is then shown by presenting a schedule which services  $\eta$  in  $\Omega(D)$  times fewer number of I/Os.

**Theorem 6** *Every simple deterministic algorithm with global  $M$ -block lookahead has a competitive ratio of  $\Omega(D)$ .*

**Proof :** In Lemma 2 we show that algorithm  $\mathcal{A}$  performs  $\Omega(MD)$  I/Os to service the reference string  $\eta$ . On the other hand Lemma 3 presents an I/O schedule which services the same reference string in only  $\Theta(M)$  I/Os. A worst case reference string for which the theorem holds can then be constructed by repeating  $\eta$  an arbitrary number of times.  $\square$

By the nature in which the last  $2D$  phases are constructed, in each set of two phases algorithm  $\mathcal{A}$  will need to fetch at least  $M/2$  blocks from a single disk, thereby incurring at least  $M/2$  I/Os.

**Lemma 2** *Every simple deterministic algorithm  $\mathcal{A}$ , performs  $\Omega(MD)$  I/Os to service the requests in reference string  $\eta$ .*

**Proof :** To service the first  $M(D + 1)$  requests algorithm  $\mathcal{A}$  performs at least  $M(D + 1)/D$  I/Os. We shall now show that the algorithm will perform  $\Omega(M)$  I/Os in each subsequent set of two phases,  $phase(2i + D + 2)$  and  $phase(2i + D + 3)$ , for all  $0 \leq i < D$ .

By construction, no block referenced in  $phase(2i + D + 3)$  is present in algorithm  $\mathcal{A}$ 's buffer at the end of  $phase(2i + D + 1)$ . In addition,  $M/2$  blocks in  $phase(2i + D + 3)$  are referenced from the same disk. In order to service these requests algorithm  $\mathcal{A}$  must perform at least a total of  $M/2$  I/Os in phases  $phase(2i + D + 2)$  and  $phase(2i + D + 3)$  combined, for each  $0 \leq i < D$ . Hence the total number of I/Os performed by algorithm  $\mathcal{A}$  to service the reference string  $\eta$ , is  $\Omega(MD)$ .  $\square$

In contrast, we could devise a scheme wherein all the blocks of the second stage could be written out striped across all disks. This would then reduce the number of I/Os performed in each of the phases of the second stage to

$O(M/D)$ . This approach is used in Lemma 3 to develop a scheme which can service the same reference string  $\eta$ , in  $\Theta(M)$  I/Os.

**Lemma 3** *The reference string  $\eta$  can be serviced in  $\Theta(M)$  I/Os.*

**Proof :** Let us consider an I/O schedule to service reference string  $\eta$ , based upon the following two rules.

1. In any phase I/Os are initiated only on demand, simultaneously prefetching from as many of the remaining disks as allowed by buffer space.
2. Let  $\Gamma$  be the set of all blocks occurring in some  $\Lambda_{2i+D+1}$ ,  $0 \leq i < D$ . At the end of *phase*( $D + 1$ ), I/Os are performed to relocate all blocks in  $\Gamma$  such that the blocks in each  $\Lambda_{2i+D+1}$  are uniformly distributed on all disks.

During the first  $D + 1$  phases this schedule performs a total of  $(D + 1)M/D$  I/Os to read in the blocks. Now we shall show that the relocation can be performed by doing at most  $M$  reads and  $M$  writes. By doing so we can guarantee that in each of the subsequent phases only  $M/D$  reads are required, with a net cost of  $\Theta(M)$ .

If the buffer size  $M \geq D^2$ , the relocation can be easily performed in  $M$  reads and writes of  $\Gamma$  by reading blocks with full parallelism and writing out one stripe whenever we get  $D$  blocks from one set  $\Lambda_{2i+D+1}$ ,  $0 \leq i < D$ . Interestingly, relocation can also be done in  $\Theta(M)$  I/Os as long as  $M \geq D$  by reducing the problem of scheduling the I/Os to an off-line load-balancing problem which can be solved using bipartite graph matching.  $\square$

A special case of this theorem, when the algorithm does not do any write-back is representative of buffer management algorithms normally used in practice. In this case it is easy to see that the same proof holds. A simpler construction, with the last  $2MD$  requests being constructed from the initial placement of blocks on disk suffices. Theorem 6 indicates that in the worst case such algorithms are ineffective in exploiting the latent I/O parallelism even when substantial lookahead – one memory load – is provided to them.

#### 1.4.2 RAND-WB: An Efficient Randomized Algorithm

As seen from the preceding discussion, *determinism* in the I/O scheduling algorithm results in poor performance. We address this problem through the use of randomization. We present an on-line algorithm, RAND-WB, which uses randomized write-back in an attempt to parallelize repeated accesses to blocks. By doing so we show that its competitive ratio can be improved to  $\Theta(\sqrt{D})$ . In perspective, this is the best competitive ratio that is achievable by algorithms which have global  $M$ -block lookahead and a fixed initial layout of blocks on disks [3].

We begin by defining a few terms which help in the specification of the algorithm RAND-WB.

**Definition 8.** A block present in the buffer is said to be *marked* if it is referenced in the current phase; the block is said to be *unmarked* otherwise.

In order to specify the blocks to be prefetched in an I/O it is useful to determine, for each disk, the next block not in the buffer to be referenced in the same phase. Let  $L$  denote the set of these blocks.

#### Algorithm RAND-WB

Algorithm RAND-WB uses global  $M$ -block lookahead. Initially unmark all blocks in the buffer.

On a request for a data block the following actions are taken.

1. If the requested block is present in the buffer the request is serviced without any further action.
2. If the requested block is not present in the buffer a parallel I/O needs to be initiated for all blocks in the set  $L$ . Some action is required, to create the necessary space for these blocks.
  - (a) Choose any  $|L|$  unmarked blocks from the buffer, giving priority to blocks which have not been relocated. Of the blocks selected, write-back the blocks which have not been relocated as described in (b), after flagging them as relocated.
  - (b) To perform the write-back stripe the blocks in a round robin fashion across as many disks as necessary, starting the stripe from a randomly (uniform probability) chosen disk.
  - (c) Read in the blocks of  $L$  in one parallel I/O.

Intuitively, RAND-WB works as follows. On every I/O request the algorithm checks the buffer to see if the requested block is a hit, in which case the request can be serviced without any I/O. If the block is not in the buffer, an I/O is initiated and prefetches issued in parallel. However some buffer space needs to be freed to complete these I/Os. Since the buffer is of size  $M$ , and we prefetch blocks only in the current phase (size  $M$ ) there will be at least  $|L|$  blocks in the buffer which are not marked; these are candidates for eviction. However we need to write-back only those blocks which have not been relocated previously. Hence we try to choose such blocks whenever possible. The randomization in the choice of the first disk to perform a striped write guarantees that each block is effectively relocated to a randomly chosen disk. In the next section we analyze the performance of RAND-WB.

**Analysis of RAND-WB.** Let OPT denote the optimal off-line algorithm. It is easy to see that any I/O schedule can be transformed into another schedule

of the same length in which a block is never evicted before it has been referenced at least once since the last time it was fetched. Hence we implicitly assume this property for OPT.

At the start of  $phase(i)$ , let  $N_i$  be the maximum number of new blocks on a single disk in  $phase(i)$ . Similarly, let  $R_i$  be the maximum number of reuse blocks on a single disk in  $phase(i)$ . Note that while  $r_i$  depends only on the reference string,  $R_i$  also depends on the write-back policy of RAND-WB. To prove that the competitive ratio of RAND-WB is  $\Theta(\sqrt{D})$  we proceed as follows. In Claim 3 we argue that RAND-WB does not need to perform any I/Os to access the stale blocks of a phase. Lemma 4 shows that the total number of I/Os performed by RAND-WB in  $phase(i)$  is at most  $N_i + R_i$ . Lemma 5 shows that due to the use of randomization in the write-back the expected value of  $R_i$  is small. We then bound the benefit that OPT gets by prefetching new blocks. To do this we consider a sequence of phases, a super-phase, in which the number of I/Os that OPT can save by prefetching new blocks is no more than  $3M$ . We show that the number of I/Os done by OPT in a super-phase can be lower bounded by  $\Omega(M/\sqrt{D})$  (Theorem 7). Finally noting that the number of parallel writes done by RAND-WB is upper bounded by the number of parallel reads, the competitive ratio of RAND-WB is  $O(\sqrt{D})$ . The details of the proof follow.

We shall say that a block is *prefetched for phase(i)* if the earliest future reference of that block is in  $phase(i)$ . Similarly a block is said to be *from disk j* if it was last fetched from disk  $j$ .

**Definition 9.** At some instant let the number of new blocks in  $phase(i)$  which have been prefetched by OPT, from disk  $k$ , be  $\beta_{i,k}$ ; let  $\alpha_{i,k}$  be the total number of new blocks in  $phase(i)$  which reside on disk  $k$ . The *peak* of  $phase(i)$  is defined to be the maximum number of new blocks of  $phase(i)$  which need to be fetched from any single disk,  $\mathcal{P}_i = \max_k \{\alpha_{i,k} - \beta_{i,k}\}$ .

The peak is said to be *on disk d* if and only if  $\alpha_{i,d} - \beta_{i,d} = \mathcal{P}_i$ . The peak of a phase changes as OPT prefetches new blocks. The peak of  $phase(i)$  is a measure of the minimum number of I/Os that need to be done in  $phase(i)$  if no more blocks are prefetched for it.

**Definition 10.** In some I/O done by OPT in  $phase(i)$ , let  $\langle s_1, s_2, \dots, s_p \rangle$  be the new blocks prefetched for  $phase(i')$ ,  $i' > i$ , ordered by their occurrence in  $\Sigma$ . Then  $s_1$  is called a *useful block* if the peak of  $phase(i')$  following the I/O is (one) less than the peak prior to the I/O.

From the above definition, if a single phase has  $U$  useful blocks prefetched for it, then at least  $U$  blocks must have been fetched from a single disk, and hence at least  $U$  I/Os must have been done to fetch them. Of course, if the  $U$  useful blocks are for different phases then less than  $U$  I/Os may be sufficient to fetch them.

**Claim 2** *If  $U$  useful blocks are prefetched by OPT for any single phase, then OPT must have done at least  $U$  I/Os.*

We now cluster phases into super-phases based on the number of useful blocks prefetched by OPT in each cluster.

**Definition 11.** A *super-phase* is a *minimal* collection of contiguous phases such that the total number of useful blocks prefetched by OPT in the collection is greater than  $2M$ . Each phase belongs to exactly one super-phase. The  $j$ th super-phase is denoted by  $S_j$ .

Let  $T_{\text{OPT}}(j)$  be the number of I/Os done by OPT, in the  $j$ th super-phase. Similarly, let  $T_{\text{RAND-WB}}(j)$  be the expected number of I/Os done by RAND-WB in the  $j$ th super-phase.

Algorithm RAND-WB never evicts any block that is used within the same phase. Hence once a block is fetched it remains in the buffer at least till the end of the phase. As a consequence we have:

**Claim 3** *In a phase, no I/O is required by RAND-WB to fetch stale blocks. In addition, RAND-WB reads in a block at most once in a phase.*

This leads to an important observation regarding the number of I/Os done by RAND-WB in a phase. If at the start of a phase there are a maximum of  $b$  blocks from some disk referenced in that phase and not present in the buffer, then the number of read I/Os performed by RAND-WB in that phase is  $b$ . We next show that the analysis of RAND-WB can be decoupled into counting the maximum number of new and reuse blocks of any phase.

**Lemma 4** *The total number of read I/Os done by RAND-WB in phase( $i$ ) is at most  $T_i \leq N_i + R_i$ .*

**Proof :** From Claim 3 no I/O is done by RAND-WB to fetch the stale blocks in *phase( $i$ )*. Hence, the total number of read I/Os done by RAND-WB in *phase( $i$ )*  $T_i$ , is equal to the maximum number of clean blocks on any single disk in *phase( $i$ )*. By definition, the number of clean blocks in *phase( $i$ )* is the sum of the number of new and reuse blocks in that phase. Hence the maximum number of clean blocks on any disk, in *phase( $i$ )* is at most  $N_i + R_i$ .  $\square$

Algorithm RAND-WB randomly relocates blocks which are evicted from the buffer. Such blocks when referenced later in the reference string are by definition, called reuse blocks. Hence, for RAND-WB, we can bound the expected value of the maximum number of reuse blocks from any single disk in a phase by relating it to the classical occupancy problem [10]. *Suppose that  $m$  balls are randomly (uniform distribution) thrown into  $n$  urns, what is the expected maximum number of balls in any urn?* Let  $C(m, n)$  denote the expected maximal occupancy when  $m$  balls are thrown into  $n$  urns. This translates to the problem of finding the expected maximal number of reuse blocks from a single disk in any phase, giving the following lemma.

**Lemma 5** *Let the number of reuse blocks in phase( $i$ ) be  $r_i$ . The expected value of the maximum number of reuse blocks from any disk that RAND-WB needs to fetch in phase( $i$ ) is at most  $R_i = C(r_i, D) = O(r_i \ln D/D)$ .*

Claim 3 states that RAND-WB does not perform any I/Os to fetch the stale blocks of a phase. Hence stale blocks cannot contribute to the difference in the number of I/Os done by OPT and RAND-WB in super-phase,  $S_j$ . Moreover, at most  $M$  blocks (one memory load) can be prefetched in earlier super-phases for phases belonging to super-phase  $S_j$ . This together with the fact that, by definition, at most  $2M$  useful blocks can be generated and consumed in the same super-phase, indicates that the the total number of useful blocks consumed in a super-phase is at most  $3M$ .

Finally, we note that the total number of writes performed by RAND-WB can be bounded by the number of reads. Hence it is enough to only count reads performed by RAND-WB in a super-phase. Putting the above arguments together, we obtain the following relation between the number of I/Os done by OPT and RAND-WB in a super-phase. We use  $\sum_{S_j}$  to indicate the sum over all  $i$  such that  $phase(i)$  belongs to  $S_j$ .

**Lemma 6** *The number of I/Os performed by RAND-WB in super-phase  $S_j$  is*

$$T_{\text{RAND-WB}}(j) \leq 2(T_{\text{OPT}}(j) + 3M + \sum_{S_j} \mathcal{C}(r_i, D))$$

where  $r_i$  is the number of reuse blocks in  $phase(i)$ .

Equipped by the above lemmas, we now bound the ratio of the number of I/Os done by RAND-WB and OPT – the competitive ratio.

**Theorem 7** *The competitive ratio of the algorithm RAND-WB is  $\Theta(\sqrt{D})$ .*

**Proof :** We partition the reference string into phases and then group phases into super-phases. We shall show that the ratio of the number of I/Os done by RAND-WB and OPT in any super-phase is at most  $O(\sqrt{D})$ .

We now derive a lower bound on the number of I/Os performed by OPT in a super-phase. Let  $S$  be an arbitrary super-phase of the reference string. By definition, a total of at least  $2M$  useful blocks are fetched by OPT in the super-phase. But as the buffer can hold at most  $M$  blocks, only  $M$  of these could have been prefetched for super-phases beyond  $S$ . Hence at least  $M$  useful blocks prefetched by OPT in the super-phase must have been consumed in the same super-phase.

For any  $phase(i)$  in the super-phase  $S_j$ , let the number of useful blocks prefetched by OPT in  $phase(i)$  for other phases in the same super-phase be  $\gamma_i$ . Let the number of such phases be  $n$ . Let the number of I/Os done by OPT in  $phase(i)$  to prefetch useful blocks be  $I_i$ . Let  $phase(i_k)$ , in super-phase  $S_j$ , be the  $k$ th phase for which a useful block is prefetched by OPT in  $phase(i)$ . Let  $\beta_k$  be the number of useful blocks prefetched by OPT in  $phase(i)$  for  $phase(i_k)$ . During one I/O by OPT in  $phase(i)$  at most one useful block could have been prefetched for  $phase(i_k)$ : the number of I/Os done by OPT to fetch useful blocks in  $phase(i)$  is  $I_i \geq \beta_k$ , for all  $1 \leq k \leq n$ .

The number of useful blocks prefetched in  $phase(i)$  for phases prior to and including  $phase(i_k)$  is  $\sum_{l=1}^k \beta_l$ . This implies that the number of (useful) blocks

occupying space in the buffer during  $phase(i_k)$  is at least  $\gamma_i - \sum_{l=1}^k \beta_l$ . Hence at least these many blocks referenced in  $phase(i_k)$ , are not present in the buffer at the start of  $phase(i_k)$ . Due to this at least  $(\gamma_i - \sum_{l=1}^k \beta_l)/D$  I/Os need to be done by OPT in  $phase(i_k)$ .

The total number of useful blocks prefetched in  $phase(i)$  for phases in  $S_j$  is  $\gamma_i = \sum_{l=1}^n \beta_l$ . Then the total number of I/Os in  $S_j$  caused by the reduced buffer space due to prefetched blocks is at least

$$\begin{aligned}
T_{\text{OPT}}(j) &\geq \sum_{S_j} \sum_{k=1}^n (\gamma_i - \sum_{l=1}^k \beta_l) / D \\
&= \sum_{S_j} \left( \sum_{k=1}^n \sum_{l=k+1}^n \beta_l / D \right) \\
&= \sum_{S_j} \left( \sum_{k=1}^n (\sum_{l=k}^n \beta_l - \beta_k) / D \right) \\
&= \sum_{S_j} \left( \sum_{k=1}^n k\beta_k / D - \gamma_i / D \right) \tag{1.1}
\end{aligned}$$

We know that  $\sum_{k=1}^n \beta_k = \gamma_i$  and also  $\beta_k \leq I_i$ .  $\sum_{k=1}^n k\beta_k$  is minimized when  $\beta_r \geq \beta_s$  whenever  $r < s$ . Therefore set  $\beta_k$  to its maximum value  $I_i$ , for as many of the initial  $\beta_k$ s as possible

$$\begin{aligned}
\sum_{k=1}^n k\beta_k &\geq \sum_{k=1}^{\gamma_i/I_i-1} kI_i \\
&\geq \frac{\gamma_i^2}{2I_i} - \frac{\gamma_i}{2}
\end{aligned}$$

Hence, from Equation 1.1, the total number of I/Os caused by prefetching and consuming useful blocks in the same phase can be bounded as follows.

$$T_{\text{OPT}}(j) \geq \sum_{S_j} \left( \frac{\gamma_i^2}{2I_i D} - \frac{3\gamma_i}{2D} \right) \tag{1.2}$$

The total number of I/Os performed by OPT in the super-phase is at least the sum of the number of I/Os to fetch useful blocks in the phase.

$$T_{\text{OPT}}(j) \geq \sum_{S_j} I_i \tag{1.3}$$

Since a total of  $\gamma_i$  (useful) blocks are fetched in each phase, the number of I/Os performed in the super-phase must be at least

$$T_{\text{OPT}}(j) \geq \sum_{S_j} \gamma_i / D \tag{1.4}$$

Combining Equations 1.2, 1.3 and 1.4,

$$\begin{aligned} T_{\text{OPT}}(j) &\geq \max \left\{ \sum_{S_j} I_i, \sum_{S_j} \left( \frac{\gamma_i^2}{2I_i D} - \frac{3\gamma_i}{2D} \right), \sum_{S_j} \frac{\gamma_i}{D} \right\} \\ &\geq \sum_{S_j} \left( I_i + \frac{\gamma_i^2}{2I_i D} - \frac{3\gamma_i}{2D} + \frac{\gamma_i}{D} \right) / 3 \end{aligned}$$

Noting that  $I_i \geq \gamma_i/D$  we get

$$\begin{aligned} I_i + \frac{\gamma_i^2}{2I_i D} - \frac{\gamma_i}{2D} &\geq \frac{I_i}{2} + \frac{\gamma_i^2}{2I_i D} \\ &\geq \gamma_i / \sqrt{D} \end{aligned}$$

At least  $M$  useful blocks prefetched in the super-phase  $S_j$  are for phases in the same super-phase:  $\sum_{S_j} \gamma_i \geq M$ . Hence,  $T_{\text{OPT}}(j) \geq M/3\sqrt{D}$ .

By definition, a block which is a reuse block in  $\text{phase}(i)$  is not referenced in  $\text{phase}(i-1)$ . Hence it can be argued in a fashion similar to that of the useful blocks that at least  $\sum_{S_j} r_i/D$  I/Os are performed by OPT due to the reuse blocks – either they are in the buffer during  $\text{phase}(i-1)$ , in which case they occupy buffer space, or are fetched in  $\text{phase}(i)$  with full parallelism. Hence the total number of I/Os done by OPT in super-phase  $S_j$ , is at least

$$T_{\text{OPT}}(j) \geq \max \left\{ M/3\sqrt{D}, \sum_{S_j} r_i/D \right\}$$

From Lemma 6 and the preceding bound on  $T_{\text{OPT}}(j)$ ,  $T_{\text{RAND-WB}}(j)/T_{\text{OPT}}(j)$  is  $O(\sqrt{D})$ .

To lower bound algorithm RAND-WB one can construct a read-once reference string [3] for which RAND-WB will perform at least  $\Omega(\sqrt{D})$  times more I/Os than OPT. This therefore implies that the competitive ratio of RAND-WB is  $\Theta(\sqrt{D})$ .  $\square$

**Corollary 1** *If the initial data distribution is such that each block independently has probability  $1/D$  of being on any disk then the competitive ratio of RAND-WB is  $\Theta(\log D)$ .*

The proof follows from the fact that if the initial data layout is random, then the expected value of the maximum number of new blocks from a single disk in any phase parallels that for reuse blocks. In fact, in this situation RAND-WB does not need to rewrite evicted blocks, since the placement for the reuse blocks is already randomized.

## 1.5 CONCLUDING REMARKS

The use of multiple-disk parallel I/O systems to alleviate the I/O bottleneck is a practical reality. However it remains a challenging problem to effectively use the increased disk bandwidth to reduce the I/O latency of an application. Effective use of I/O parallelism requires careful coordination between data placement, prefetching and caching policies.

The parallel I/O system examined in this article is modeled using the intuitive parallel disk model consisting of independent disks sharing a common I/O buffer. The I/O accesses are modeled using a reference string which is the ordered sequence of blocks that the computation requires. We studied the I/O scheduling problem in such a system in the framework of competitive analysis. Two types of reference strings, read-once and read-many, were considered separately. Read-once reference strings, in which blocks are accessed only once, arise naturally in streamed applications like video servers. In contrast there is no restriction on the frequency of block accesses in read-many reference strings. In order to perform accurate prefetching the scheduling algorithm needs to be able to look ahead into the reference string. Based on the type of information available two forms of bounded lookahead, global  $M$ -block and local, were defined.

For the case of read-once reference strings it is known [3] that any deterministic prefetching algorithm using global  $M$ -block lookahead has a competitive ratio of  $\Omega(\sqrt{D})$ , and those using local lookahead must have a competitive ratio of  $\Omega(D)$ . Hence in the worst case on-line deterministic algorithms are significantly serialized. We therefore discussed several prefetching schemes based on randomized data placement, and presented a simple prefetching algorithm; the algorithm is shown to perform  $\Theta(N/D)$  expected number of I/Os, for a reference string of length  $N$  for  $M = \Omega(D \log D)$ .

For the case of general read-many strings we introduced the notion of *write-back*, whereby blocks are dynamically relocated between disks during the computation. We showed that any algorithm with global  $M$ -block lookahead, which uses deterministic write-back and buffer management policies must have a competitive ratio of  $\Omega(D)$ . In other words, any strategy that is based solely on the bounded lookahead and the past behavior of the algorithm, can in the worst case fail to exploit any I/O parallelism.

Using randomization we improved the performance considerably. We presented a randomized algorithm RAND-WB, that uses a novel randomized write-back scheme, and attains the lowest possible competitive ratio of  $\Theta(\sqrt{D})$ . As a corollary, if initially all the data blocks are randomly placed on disks, the competitive ratio of RAND-WB is  $\Theta(\log D)$ . This is the first study of on-line algorithms for read-many reference strings in the parallel disk model. Other recent works have dealt either with on-line algorithms for a restricted class of read-once reference strings, or with off-line approximation algorithms for read-many reference strings.

## Notes

1. Prefetching may however help in overlapping cpu and I/O operations [6].
2. Even if  $B_3$  or  $C_2$  were fetched on demand in the next 2 I/Os, the same eviction decisions would be made.

## References

- [1] S. Albers. The Influence of Lookahead in Competitive Paging Algorithms. In *1st Annual European Symposium on Algorithms*, volume 726, pages 1–12. LNCS, Springer Verlag, 1993.
- [2] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple Randomized Mergesort on Parallel Disks. *Parallel Computing*, 23(4):601–631, June 1996.
- [3] R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter. Competitive Parallel Disk Prefetching and Buffer Management. In *Fifth Annual Workshop on I/O in Parallel and Distributed Systems*, pages 47–56. ACM, November 1997.
- [4] L. A. Belady. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [5] D. Breslauer. On Competitive On-Line Paging with Lookahead. In *13th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1046 of LNCS, pages 593–603. Springer Verlag, February 1996.
- [6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 188–197. ACM, May 1995.
- [7] 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.
- [8] C. S. Ellis and D. Kotz. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, 1999.
- [9] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. E. Young. Competitive Paging Algorithms. *Journal of Algorithms*, 12(4):685–699, December 1991.
- [10] N. L. Johnson and S. Kotz. *Urn Models and Their Application: an Approach to Modern Discrete Probability Theory*. Wiley, New York, 1977.
- [11] M. Kallahalla. *Competitive Prefetching and Buffer Management for Parallel I/O Systems*. Masters Thesis, Rice University (1997).
- [12] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *37th Annual Symposium on Foundations of Computer Science*, pages 540–549. IEEE, October 1996.
- [13] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.

- [14] K. K. Lee, M. Kallahalla, B. S. Lee, and P. J. Varman. Performance Comparison of Sequential Prefetch and Forecasting Using Parallel I/O. In *Proceedings of IASTED PDCN Conference*, April 1997.
- [15] K. K. Lee and P. J. Varman. Prefetching and I/O Parallelism in Multiple Disk Systems. In *Proceedings 24th International Conference on Parallel Processing*, pages III:160–163, August 1995.
- [16] L. A. McGeoch and D. D. Sleator. A Strongly Competitive Randomized Paging Algorithm. *Algorithmica*, (6):816–825, 1991.
- [17] V. S. Pai, A. A. Schäffer, and P. J. Varman. Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging. *Theoretical Computer Science*, 128(1–2):211–239, June 1994.
- [18] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.
- [19] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.
- [20] 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*, volume 16. LNCS, Springer Verlag, December 1996.
- [21] J. S. Vitter and E. A. M. Shriver. Optimal Algorithms for Parallel Memory, I: Two-Level Memories. *Algorithmica*, 12(2-3):110–147, 1994.