

Improving Parallel-Disk Buffer Management using Randomized Writeback*

Mahesh Kallahalla

Peter J. Varman

Department of Electrical and Computer Engineering

Rice University

Houston TX 77251

E-mail: {kalla,pjv}@rice.edu

Abstract

We address the problems of I/O scheduling and buffer management for general reference strings in a parallel I/O system. Using the standard parallel disk model with D disks and a shared I/O buffer of size M , we study the performance of on-line algorithms that use bounded global M -block lookahead. We introduce the concept of write-back whereby blocks are dynamically relocated between disks during the course of the computation. Write-back allows the layout to be altered to suit different access patterns in different parts of the reference string. We show that any bounded-lookahead on-line algorithm that uses purely deterministic policies must have a competitive ratio of $\Omega(D)$. We show how to improve the performance by using randomization, and present a novel algorithm, RAND-WB, using a randomized write-back scheme. RAND-WB has a competitive ratio of $\Theta(\sqrt{D})$, which is the best achievable by any on-line algorithm with only global M -block lookahead. If the initial layout of data on the disks is uniformly random, RAND-WB has a competitive ratio of $\Theta(\log D)$.

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-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 I/O system is modeled using the intuitive parallel disk model introduced by Vitter and Shriver [18]: the I/O system consists of D independently-accessible disks and an associated I/O buffer with a capacity of M blocks that 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 (written) from (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, 8, 16]. 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 regarding 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 using prefetching in a single-disk system was addressed in [6], and off-line approximation algorithms were presented and analyzed. In the context of parallel I/O studied in this paper, several new issues (discussed in Section 2) 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 a parallel I/O system using bounded lookahead was addressed. Fundamental bounds on the performance of

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

algorithms were presented for an important 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 [17] for a distributed-buffer parallel I/O model in which each disk has its own private buffer. In an alternative stall model of parallel I/O, a generalization of the sequential model of [6] to multiple disks and a shared buffer, an approximate off-line algorithm for I/O scheduling and buffer management was presented in [10]. 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 paper we study the on-line I/O scheduling problem for read-many reference strings in the framework of competitive analysis. We use the competitive ratio (defined formally in Section 2.1) as the measure of performance of an on-line algorithm. Informally, this ratio measures how well a given on-line algorithm compares with the optimal off-line algorithm; the off-line algorithm has access to the entire reference string and constructs its schedule using some off-line optimization strategy. We introduce the concept of *write-back*, whereby blocks are dynamically relocated between disks during the course of the computation. We show that buffer management algorithms which 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 non-trivial to do in an online manner. We show that *any* scheduling algorithm with bounded lookahead that uses deterministic rather than randomized buffer management and I/O scheduling policies can in the worst case require $\Theta(D)$ times as many I/Os as the optimal off-line algorithm.

We use *randomization* to improve the performance of buffer management and scheduling decisions. In particular, we design an on-line buffer management and scheduling algorithm, RAND-WB, whose competitive ratio matches that of the *best on-line* algorithm that uses 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. This is significantly better than the $\Theta(D)$ competitive ratio of deterministic methods.

The rest of the paper is organized as follows. Issues in parallel I/O and basic definitions are presented in Section 2. In Section 3 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. An algorithm using randomized write-back that achieves a competitive ratio of $\Theta(\sqrt{D})$ is presented in Section 4.

2. Performance Issues in Parallel I/O

In the sequential I/O model, the measure of performance is the total number of blocks accessed from the disk. However, for parallel I/O a better measure is the number of parallel I/Os performed, as more than one block can be fetched in a single I/O operation. The potential for overlapped accesses raises new issues that make the problem of optimizing the number of parallel I/Os challenging.

Prefetching: It is well known that early fetching cannot reduce the number of I/Os needed¹ in the single-disk model [16]. In a parallel I/O system fetching a block only when it is requested by the computation 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. Prefetching for multiple disks for specific applications has been studied in [2, 11, 12, 13, 14], 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. In the parallel model fetching blocks in order of their appearance in the reference string can be inefficient. For instance, consider the example of Figure 1 which assumes $D = 3$ 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 $\Sigma = A_1 A_2 A_3 A_4 B_1 C_1 A_5 B_2 C_2 A_6 B_3 C_3 A_7 B_4 C_4 C_5 C_6 C_7$. Figure 1 (a) shows the I/O schedule obtained by always fetching in the order of the reference string. At step 1, blocks

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

(a)

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		

(b)

Figure 1. Schedules for reference string Σ .

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 Σ means that we fetch B_3 ; continuing in this manner we obtain a schedule of length 9. In Fig-

¹Prefetching may however help in overlapping cpu and I/O operations [6].

ure 1 (b), at step 2 disk 2 is idle (even though there is buffer space) and C_2 that occurs later than B_2 in Σ is prefetched; similarly, at step 3, C_3 that 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 .

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 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 sufficient. 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 a tension between the need to increase parallelism by prefetching and the desire to delay the fetch as late as possible to obtain the best possible candidate for eviction. For illustration, 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 2(a) 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 evicts A_3 that

Disk 1	A_4	A_1	A_2	A_3
Disk 2	B_3			
Disk 3	C_2			

(a)

Disk 1	A_4	A_1
Disk 2	B_3	B_1
Disk 3	C_2	C_1

(b)

Figure 2. Schedules for reference string Σ^* .

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 ². 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 2(b) 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 I/O.

²Note that even if we did not prefetch B_3 or C_2 but fetched them on demand on the next 2 I/Os, the same eviction decisions would be made, so any difference in performance in this example is not due to suboptimal decisions caused by prefetching.

2.1. Definitions

We use the Parallel Disk Model [18] of a parallel I/O system. This model consists of D independently accessible disks and an associated 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 performed 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, rather than overlap of cpu and I/O operations. In the rest of the paper we shall use I/O to mean a *parallel I/O*.

Reference strings, introduced informally in the previous section, model the sequence of I/O accesses. In this paper we consider read-many reference strings where there is no restriction on the blocks referenced. This is contrast to *read-once* reference strings [3], where all the requests are to distinct blocks.

Definition 1 The sequence of read I/O requests 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.

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 window of future accesses embodies the notion of lookahead. In sequential I/O systems lookahead helps in the eviction decisions of the buffer management algorithm [1, 5]. In parallel systems lookahead is needed for making prefetching decisions [3] independent of its use in aiding evictions. Our algorithms use global M -block lookahead defined below for read-many reference strings. Such a lookahead is 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 [15]. 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.

Definition 3 An online parallel prefetching algorithm A has a competitive ratio of C_A if for any reference string the number of I/Os required by A is within a factor C_A of the number of I/Os required by the optimal off-line algorithm to serve the same reference string. If A is a randomized algorithm then the expected number of I/Os done by A is considered.

3. Lower Bound on Deterministic Algorithms

In a read-many reference string, data blocks 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 it was fetched from, as *write-back*. Write-back allows the location of a data block to be dynamically altered. However, there is only one copy of the block on the disks at any time.

We next introduce the notion of simple deterministic algorithms that captures the determinism inherent in most existing buffer management algorithms. An algorithm is said to be a *simple deterministic algorithm* (SDA) if at any instant 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 is a deterministic function of the reference string till that instant, the lookahead, and the contents of the buffer. The central idea in the above characterization is that a SDA uses *deterministic* policies. This determinism can be exploited by an adversary to generate reference strings which require a SDA to unnecessarily make a large number of I/Os.

As a simple illustration, consider a reference string that consists of accesses to an $M \times D$ matrix of blocks. The blocks are first accessed in row-major order and then in column-major order. Assume that the matrix is laid out in block row-major order, striped across all disks. An algorithm with only global M -block lookahead, that does not perform any write-back will significantly serialize the I/Os while accessing the columns. A better strategy would be to relocate the blocks while accessing the rows so that the accesses for the columns can also be parallelized. In this case the writes too can proceed with full parallelism and drastically reduce the number of I/Os required.

Even in a more general case, when a deterministic algorithm uses an arbitrary write-back policy along with more sophisticated prefetching and block replacement heuristics, there are reference strings for which the algorithm must serialize significantly. This intuition is formalized in Theorem 1 which gives a lower bound of $\Omega(D)$ on the competitive ratio of any simple deterministic algorithm.

3.1. Proof of Lower Bound

Let \mathcal{A} be an arbitrary SDA with global M -block lookahead. Based on the behavior of \mathcal{A} we construct a reference 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 detailed proofs follow.

Theorem 1 *Every simple deterministic algorithm with only global M -block lookahead has a competitive ratio of $\Omega(D)$.*

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 4 The reference string is partitioned into sub-strings called *phases* such that each phase (a) is of maximal length and (b) contains references to exactly M distinct blocks. The i th phase, $i \geq 1$, is denoted by $phase(i)$.

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.

Definition 5 The set of *clean blocks* in $phase(i)$ is the set of blocks in $phase(i)$ not requested in $phase(i-1)$. The set of *stale blocks* in $phase(i)$ is the set of blocks in $phase(i)$ requested in $phase(i-1)$. The set of *new blocks* in $phase(i)$ is the set of blocks in $phase(i)$ not requested in any, $phase(j)$, $1 \leq j < i$. The set of *reuse blocks* in $phase(i)$ is the set of clean blocks in $phase(i)$ that have been requested in some $phase(j)$, $j < i-1$.

Let us now construct a reference string η , consisting of $M(3D+1)$ references, which will be used to give a lower bound on the performance of \mathcal{A} . Reference strings of arbitrary length for which the proofs will follow can be constructed by repeating η . The details of the construction are presented in Figure 3. Figure 4 illustrates the structure of the constructed reference string as seen by algorithm \mathcal{A} .

With respect of the construction of Figure 3, we note the following. By counting the number of blocks which have been referenced exactly once till $phase(2i+D+1)$, and using the fact that there are at most M blocks in the buffer, we can show that at the end of $phase(2i+D+1)$ at least $M/2$ blocks, requested exactly once and not present in the buffer, reside on some disk. This will ensure that Λ_{2i+D+1} is well defined for all $0 \leq i < D$ thereby allowing the construction of the sequence η .

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

Lemma 2 *Algorithm \mathcal{A} performs $\Omega(MD)$ I/Os to service reference string η .*

1. The first $D + 1$ phases of the reference string η , consists of $M(D + 1)$ references to 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. The i^{th} , $0 \leq i < D$, set 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 given by the sequence Λ_{2i+D+1} , 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. Let d denote the disk with the maximum number of blocks from F_k not in the buffer. The sequence Λ_k , is defined as the ordered sequence of $M/2$ earliest referenced blocks in $A_{k,d} - B_{k,d}$.

Figure 3. Construction of reference string η

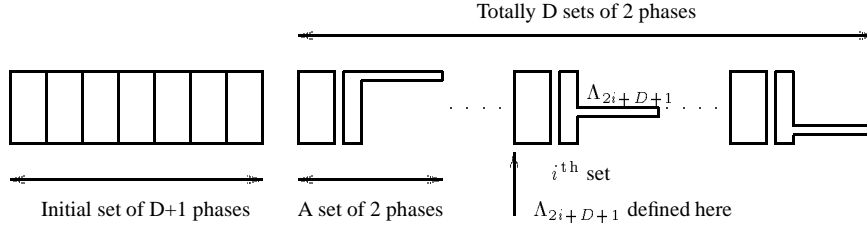


Figure 4. Structure of a worst case reference string for \mathcal{A}

Proof : To service the first $M(D + 1)$ requests algorithm \mathcal{A} performs at least $M(D + 1)/D$ I/Os. 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 a single 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 η , is $\Omega(MD)$. \square

In contrast, if all the blocks of the last $2D$ phases are written out striped across all disks the number of I/Os performed in each of these phases can be reduced to $O(M/D)$. This approach is used in Lemma 3 to develop a scheme which can service η in $\Theta(M)$ I/Os.

Lemma 3 η can be serviced in $\Theta(M)$ I/Os.

Proof : Let us consider an I/O schedule to service η based on the following two rules (a) within a phase I/Os are initiated only on demand; in parallel prefetch as many of the next D requests as allowed by buffer space (b) let Γ be the set of all blocks occurring in some Λ_{2k+D+1} , $0 \leq k < D$. At the end of $phase(D + 1)$, I/Os are performed to relocate

all blocks in Γ such that the $M/2$ blocks in each Λ_{2k+D+1} are uniformly distributed on all disks.

This schedule performs $(D + 1)M/D$ read I/Os till the end of $phase(D + 1)$. In each of the subsequent phases only M/D reads are required, as the blocks are uniformly distributed across all disks following the relocation. If $M \geq D^2$, the relocation can be performed in M reads and writes of Γ by reading blocks with full parallelism and writing out one stripe whenever D blocks are got from any set Λ_{2i+D+1} , $0 \leq i < D$. Interestingly, relocation can be done in $\Theta(M)$ I/Os even if $M \geq D$ by reducing it to an off-line load-balancing problem which can be solved using bipartite graph matching. \square

A special case, when the SDA 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 with a simpler construction suffices (the last $2MD$ requests can be constructed from the initial data layout). Hence 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.

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 that have already been relocated. Of these write-back those 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 all the disks starting the stripe from a randomly (uniform probability) chosen disk.
 - (c) Read in the blocks of L in one parallel I/O.

Figure 5. Algorithm RAND-WB

4. RAND-WB: A Randomized Algorithm

From the preceding discussion, *determinism* in the I/O scheduling algorithm results in poor performance. We address this problem through the use of randomization and 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].

A block in the buffer is called *marked* if it is referenced in the current phase; else it is called *unmarked*. To specify the blocks to be prefetched in an I/O consider, for each disk, the next block not present in the buffer that is referenced in the same phase. Let L denote the set of these blocks. Let \mathcal{B} denote the set of all blocks in the buffer. Intuitively, RAND-WB works as follows. If the I/O request is a hit in the buffer it can be serviced without any I/O. If it is a miss, an I/O is initiated and prefetches issued in parallel. However some buffer space needs to be freed to complete these I/Os. Since 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. As we do not need to write-back blocks that have been relocated previously, we try to choose such blocks whenever possible. The randomization in the choice of the first disk to start a stripe guarantees that each block is effectively relocated to a randomly chosen disk. The specifics of algorithm RAND-WB are presented in Figure 5. In the next section we analyze the performance of RAND-WB.

4.1. Analysis of RAND-WB

Let OPT denote the optimal off-line algorithm. First

note that any I/O schedule can be transformed into another schedule of the same or smaller length in which a block is never evicted before it has been referenced at least once since the last time it was fetched. Hence we implicitly assume this property for OPT. We shall now prove that the competitive ratio of RAND-WB is $\Theta(\sqrt{D})$.

The following terms are defined with respect to the schedule created by OPT. 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 6 Let the set of new blocks in *phase(i)* be N_i .

- At some instant let the blocks in the buffer be \mathcal{B} . The *residual height* of *phase(i)* at that time is the maximum number of blocks residing on a single disk in the set $N_i - \mathcal{B}$.
- The number of *useful blocks* prefetched in *phase(j)* for *phase(i)* is the difference between the residual heights of *phase(i)* at the start and end of *phase(j)*.

If at the start of a phase, there are U useful blocks in the buffer for that phase, at least U I/Os need to have been done in the past to fetch them since at least U blocks need to have been fetched from a single disk. Of course, the I/Os to fetch useful blocks for different phases can be overlapped. Let $I_{\text{OPT}}(i)$ and $I_{\text{RAND-WB}}(i)$ be the number of I/Os done by OPT and RAND-WB, respectively, in the i^{th} phase. Let T_{OPT} be the total number of I/Os done by OPT.

Claim 1 *In a phase no I/O is done by RAND-WB to fetch stale blocks, and any block is read in at most once.*

This follows from the marking nature of RAND-WB. Hence, 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 can be decoupled into counting the number of I/Os done by the algorithm in servicing the new blocks and reuse blocks. Let H_i^n (H_i^r) denote the maximum number of new (reuse) blocks of $phase(i)$ on a single disk, at the start of $phase(i)$.

Lemma 4 *The number of read I/Os done by RAND-WB in $phase(i)$ is at most $I_{RAND-WB}(i) \leq H_i^n + H_i^r$.*

Proof : From Claim 1 no I/O is done by RAND-WB to fetch the stale blocks in $phase(i)$. Hence, the total number of I/Os done by RAND-WB in $phase(i)$ 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 $H_i^n + H_i^r$. \square

Algorithm RAND-WB randomly relocates blocks that are evicted from the buffer. Therefore, we can bound the number of I/Os that algorithm RAND-WB requires to fetch the reuse blocks in any phase, by relating it to the classical occupancy problem [9]. *Suppose that m balls are randomly (uniform distribution) thrown into n urns, what is the expected maximum number of balls in any urn?* Let $\mathcal{C}(m, n)$ denote the expected maximal occupancy when m balls are thrown into n urns. Let the number of reuse blocks in $phase(i)$ be r_i .

Lemma 5 *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 $\mathcal{C}(r_i, D) = O(r_i \log D/D)$.*

It can be noted that the total number of writes performed by RAND-WB is bounded by the number of reads. Hence it is enough to only count reads performed by RAND-WB. Thus, from Lemmas 4 and 5, if \sum_p indicates the sum over all i such that $phase(i)$ is in the reference string, we get:

Lemma 6 *If the total number of useful blocks prefetched by OPT is U ,*

$$\sum_p I_{RAND-WB}(i) \leq 2\left(\sum_p I_{OPT}(i) + U + \sum_p \mathcal{C}(r_i, D)\right)$$

Theorem 2 *The competitive ratio of RAND-WB is $\Theta(\sqrt{D})$.*

Proof : We shall prove the theorem by deriving a lower bound on the number of I/Os performed by OPT. Let the number of useful blocks prefetched by OPT in $phase(i)$ for n other phases be γ_i . Let the number of I/Os done by OPT in $phase(i)$ to prefetch these blocks be I_i . Let $phase(i_k)$, be the k^{th} phase for which a useful block is prefetched by

OPT in $phase(i)$. Let β_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 $I_i \geq \beta_k, 1 \leq i \leq n$.

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 other phases is $\gamma_i = \sum_{l=1}^n \beta_l$. Then the total number of I/Os caused by the reduced buffer space due to prefetched blocks is at least

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

We know that $\sum_{k=1}^n \beta_k = \gamma_i$ and also $\beta_k \leq I_i$. Then $\sum_{k=1}^n k \beta_k$ is minimized when $\beta_r \geq \beta_s$ whenever $r < s$. Therefore each β_k is set to its maximum value: I_i .

$$\sum_{k=1}^n k \beta_k \geq \sum_{k=1}^{\gamma_i/I_i-1} k I_i \geq \frac{\gamma_i^2}{2I_i} - \frac{\gamma_i}{2}$$

Hence the total number of I/Os caused by prefetching and consuming useful blocks can be bounded as follows.

$$T_{OPT} \geq \sum_p \left(\frac{\gamma_i^2}{2I_i D} - \frac{3\gamma_i}{2D} \right)$$

The total number of I/Os performed by OPT is at least the sum of the number of I/Os to fetch useful blocks:

$$T_{OPT} \geq \sum_p I_i$$

Since a total of γ_i (useful) blocks are fetched in $phase(i)$, the number of I/Os performed must be at least

$$T_{OPT} \geq \sum_p \gamma_i / D$$

Combining the three bounds on T_{OPT}

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

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

$$I_i + \frac{\gamma_i^2}{2I_i D} - \frac{\gamma_i}{2D} \geq \frac{I_i}{2} + \frac{\gamma_i^2}{2I_i D} \geq \gamma_i/\sqrt{D}$$

Hence if a total of U useful blocks are prefetched by OPT: $\sum_p \gamma_i \geq U$. Hence, $T_{\text{OPT}} \geq U/3\sqrt{D}$.

Now, 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 $\sum_p r_i/D$ I/Os is 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, is at least

$$T_{\text{OPT}} \geq \max(U/3\sqrt{D}, \sum_p r_i/D)$$

By Lemma 6 and the preceding bound on T_{OPT} ,

$$T_{\text{RAND-WB}}/T_{\text{OPT}} = O(\sqrt{D})$$

By adapting a result in [3] a read-once reference string can be constructed, for which RAND-WB performs 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 number of I/Os required to fetch the new blocks in any phase parallels that required 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.

5. Conclusions

In this paper we studied the I/O scheduling problem for general read-many reference strings in a parallel I/O system. We introduced the concept of *write-back* whereby blocks are dynamically relocated between disks during the course of the computation. We showed that any algorithm with bounded lookahead, that uses deterministic write-back and buffer management policies 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 significantly serialize its disk accesses.

Using *randomization* we improved the performance of scheduling decisions. 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 $\Theta(\log D)$.

References

- [1] S. Albers. The Influence of Lookahead in Competitive Paging Algorithms. In *Proc. of ESA'93*, 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 *Proc. of IOPADS'97*, pages 47–56. ACM, 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 *Proc. of STOC'96*, volume 1046, LNCS, pages 593–603. Springer Verlag, Feb. 1996.
- [6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the Joint Int. Conf. on Measurement and Modeling of Comp. Sys.*, 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] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. D. Sleator, and N. E. Young. Competitive Paging Algorithms. *Journal of Algorithms*, 12(4):685–699, Dec. 1991.
- [9] N. L. Johnson and S. Kotz. *Urn Models and Their Application: an Approach to Modern Discrete Probability Theory*. Wiley, New York, 1977.
- [10] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *Proc. of FOCS'96*, pages 540–549. IEEE, Oct. 1996.
- [11] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, 1999.
- [12] K. K. Lee, M. Kallahalla, B. S. Lee, and P. J. Varman. Performance Comparison of Sequential Prefetch and Forecasting Using Parallel I/O. In *Proc. of PDCN'97*, Apr. 1997.
- [13] K.-K. Lee and P. J. Varman. Prefetching and I/O Parallelism in Multiple Disk Systems. In *Proc. of ICPP'95*, pages III:160–163, Aug. 1995.
- [14] 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.
- [15] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of ASPLOS'95s*, pages 79–95, Dec. 1995.
- [16] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, Feb. 1985.
- [17] P. J. Varman and R. M. Verma. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Proc. of FSTTCS'96*, volume 16, LNCS, Springer Verlag, Dec. 1996.
- [18] J. S. Vitter and E. A. M. Shriver. Optimal Algorithms for Parallel Memory, I: Two-Level Memories. *Algorithmica*, 12(2–3):110–147, 1994.