Randomized Parallel Prefetching and Buffer Management*

Peter J. Varman

Department of Electrical and Computer Engineering
Rice University, Houston TX 77005, USA

Abstract. We show that deterministic algorithms using bounded lookahead cannot fully exploit the potential of a parallel I/O system. Randomization can be used to significantly improve the performance of parallel prefetching and buffer management algorithms. Using randomization in the data layout and a simple prefetching scheme, we show that a read-once reference string of length N can be serviced in $\Theta(N/D)$ parallel I/Os in a D-disk system. For the case of read-many reference strings we introduce a novel algorithm using randomized write-back with a competitive ratio of $\Theta(\sqrt{D})$. In contrast, we show that deterministic write-back results in a competitive ratio of at least $\Omega(D)$.

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 the 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 mechanisms.

The parallel I/O system is modeled using the intuitive parallel disk model: the I/O system consists of D independently-accessible disks and an associated I/O buffer with a capacity of M blocks. In the standard parallel-disk model introduced by Vitter and Shriver [16], the buffer is shared by all the disks and is therefore called the *shared buffer model*. In contrast, in the *distributed buffer model* [15], each disk has a private buffer and all I/O from a disk is between the disk and its private buffer. In this work we deal exclusively with the shared buffer model. The data for the computation is initially stored on the disks in blocks. A block is the unit of access from a disk. In each parallel I/O at most

* Supported in part by NSF Grant CCR-9704562 and a grant from the Schlumberger Foundation.

one block from each disk can be read into the buffer. 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. In general, the reference string corresponding to a computation can consist of an arbitrary interleaving of reference strings of several concurrent applications. A data block should be present in the buffer memory 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* performed to service a given reference string.

The parallel-disk model generalizes the usual single-disk model (which we will also refer to as the sequential model) used extensively in the study of paging [4, 14]. In the sequential model there is a single disk and associated buffer. The measure of performance is the total number of I/Os performed. However, in the parallel-disk model several new issues arise that make the problem of optimizing the number of parallel I/Os challenging.

- **Prefetching**: In the sequential model blocks are fetched on-demand. It is well known that early fetching cannot reduce the number of I/Os needed[2] in the single-disk model [14]. 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 be able to look ahead in the reference string, beyond the last referenced block.
- **Choice of block to be fetched 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 $\Sigma = A_1A_2A_3A_4B_1C_1A_5B_2C_2A_6B_3C_3A_7B_4C_4C_5C_6C_7$. Figure 1 (a) shows the I/O schedule 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 Σ means that we fetch B_3; continuing in this manner we obtain a schedule of length 9. In Figure 1 (b), at step 2 disk 2 is idle (even though there is buffer space) and C_2 which occurs later than B_2 in Σ is prefetched; similarly, at step 3, C_3 which occurs even later than B_2 is prefetched. However, the overall length of the schedule is 7, better than the schedule that fetched in the order of Σ.
- **Replacement Policy**: The issue of choosing a block to evict is complicated in the parallel I/O context because of the use of prefetching. There is a

[2] Prefetching may however help in overlapping CPU and I/O operations [6].

tension between the desire to increase the parallelism by prefetching and to delay the fetch as late as possible to obtain the best possible candidate for eviction.

– **Data Placement**: In order to obtain good performance accesses should be spread out evenly among the disks, both spatially as well as temporally. In applications where data sets and accesses are dynamically generated, or there a multiplicity of conflicting data access patterns that must be satisfied, the problem of data placement is critical to performance.

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)

Fig. 1. Schedules for reference string Σ.

A restricted family of reference strings called *read-once* strings in which all blocks are read-only and no block is read more than once was introduced in [3]. Such 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. A more general form of access pattern is one where the accesses are still read-only, but there is no restriction placed on the number of times a block is referenced. Such a reference string will be called a *read-many* reference string. The main difference between the problems of serving read-once reference strings and read-many reference strings is that in the read-many case buffer management plays an important role in determining the performance. In the read-once case a block could be evicted from the buffer as soon as it was 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. Furthermore the choice of block to evict is influenced by the potential parallelism with which it can be read again.

In this paper 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 bounded lookahead (defined formally in Section 1.1) must perform a significantly larger number of I/Os, in comparison with the optimal offline algorithm that has access to the entire reference string. We analyze several prefetching schemes based on a randomized data placement, and present a simple prefetching algorithm that performs an optimal (up to constants) expected number of I/Os. For the case of general read-many strings we show that a natural class of deterministic buffer management algorithms must perform at least $\Omega(D)$ times as many I/Os as the optimal. We present a randomized algorithm that uses a novel randomized write-back scheme, which performs an

expected number of I/Os that is within a factor of $\Theta(\sqrt{D})$ of the optimal offline deterministic algorithm.

Classical deterministic buffer management algorithms for the sequential I/O model were presented in [4, 14]. In sequential systems randomization was effectively employed to improve the eviction decisions made by on-line buffer management algorithms [8, 12]. The use of lookahead information to improve the eviction decisions in single-disk systems has been studied in [1, 5] using different models of lookahead. In the parallel model, lookahead is needed in order to prefetch effectively as well. The study of lookahead models for parallel I/O systems was introduced in [3] for read-once reference strings. An approximate offline deterministic prefetching and buffer management algorithm for the stall model of parallel I/O was presented in [11]. An optimal offline deterministic algorithm for the distributed buffer parallel I/O system was presented in [15].

1.1 Definitions

The notions of read-once and read-many reference strings and lookahead, which we introduced informally in the previous section, are defined below.

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.

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. For a read-once reference string this lookahead spans exactly the next M blocks in the reference string. An I/O scheduling algorithm has *local lookahead* if for each disk it knows the references till the next block from that disk not currently in the buffer.

In this paper we design algorithms which perform a parallel I/O only on demand; that is, an I/O is initiated only when there is a block requested by the computation that is not present in the buffer. We use the competitive ratio as a measure of performance of our algorithms.

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 that A requires is within a factor C_A of the number of I/Os required by the optimal offline algorithm serving the same reference string. If A is a randomized algorithm then the expected number of I/Os done by A is considered.

2 Deterministic Bounds

In this section we show that deterministic strategies using bounded lookahead can have very poor performance when compared to offline algorithms. In Sections 2.1 and 2.2 we consider the cases of read-once and read-many reference strings respectively.

2.1 Read-Once Reference Strings

For read-once sequences, we consider a *worst-case model* wherein a deterministic placement algorithm is used to place each block of the read-once sequence on a disk. Fundamental bounds on the performance of such online algorithms with bounded lookahead for read-once reference strings were derived in [3].

Since no block is referenced more than once, it would seem that we only need to be able to fetch blocks in the order of their appearance in the reference string, in order to design an optimal prefetching algorithm. Since 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 and can easily fetch blocks in the order of their appearance in the reference string. Counter to intuition, we obtained the interesting result (Theorem 4) 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 offline prefetching algorithm that knows the entire sequence. The reason for this is that in certain cases the optimal offline algorithm does not follow the policy of fetching blocks in the order of their appearance in the reference string: at times it needs to prefetch blocks from some disk that are referenced much later in the future, *before* blocks on some other disk that are about to be referenced in the immediate future.

Theorem 4. *[3] The competitive ratio of any deterministic online 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 of consumption of blocks originating from different disks. It turns out that this is a very powerful advantage for an adversary in the shared buffer configuration. The adversary can force a higher lower bound on the competitive ratio of such online algorithms. Theorem 5 below shows that for the shared buffer configuration *any* deterministic algorithm using only local lookahead can perform $\Omega(D)$ times as bad as the optimal off-line algorithm. Note that it is trivial to design 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.

Theorem 5. *[3] The competitive ratio of any deterministic online algorithm having local lookahead is at least $\Omega(D)$.*

In Section 3.1 we will present algorithms using randomized placement that perform an optimal number of parallel I/Os using either form of bounded lookahead.

2.2 Read-Many Reference Strings

In a read-many reference string a data block can be requested more than once by the application. Hence a situation may arise wherein a particular data layout

strategy may be favorable for data accesses occurring in some 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 at the next set of accesses. The driving intuition is to rearrange the layout so that blocks which are evicted from the buffer 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 was initially located, incurs the cost of writing out a block. But the gain in I/O parallelism by relocation can be used to offset the extra cost of performing a 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*.

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 existing buffer management algorithms. We shall show that, in the worst case, such algorithms may be completely ineffective in exploiting the parallelism available in the I/O subsystem.

Definition 6. Let the set of blocks in the I/O buffer after requests $r_1, r_2, \cdots r_k$ in the reference string are serviced be $\mathcal{B}_k$, and let the set of blocks in the lookahead window be $\mathcal{L}_k$. An algorithm is is said to be a *simple deterministic algorithm* (SDA) if at that time, the set of blocks that it prefetches, the set of blocks that it evicts, and the disks to which it writes back these evicted blocks can be uniquely determined by specifying $\mathcal{B}_k$, $\langle r_1, r_2, \ldots, r_k \rangle$ and $\mathcal{L}_k$.

The above definition hinges on 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 a SDA to make an unnecessarily large number of I/Os. We analyze the performance of simple deterministic algorithms with global M-block lookahead and show (Theorem 7) that the competitive ratio of such algorithms is $\Omega(D)$.

To illustrate the intuition behind the theorem consider the following simple example of an SDA. 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 current lookahead (b) perform striped write-back, writing the first block of any stripe to the first disk (c) evict blocks not in the current lookahead using the Least Recently Used (LRU) policy (d) prefetch all blocks in the lookahead window.

Consider the performance of the above algorithm while accessing a $3M \times D$ matrix in two stages: first in row major order and then in column major order. The matrix is assumed to be initially laid out on disks with the elements of each row striped across all D disks. Assume that $M = \Omega(D)$. Tracing through the algorithm we note that the I/Os during the first stage are completely parallelized. The write-back policy in this case does not perform any relocation. However,

when the matrix is accessed in the second stage there is very little disk parallelism; only a part of the next column is prefetched. The algorithm makes $3M$ I/Os during the first stage, and $3M + 2M(D-1)$ 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; its write-back policy is such that all blocks which originated on the same disk are now striped across all disks. The cost of writing back during the first stage is $3M$ I/Os. However the advantage gained is that all I/Os in the second stage can be completely parallelized. Hence the total number of I/Os performed by such an algorithm is $9M$ ($3M$ reads in each stage and $3M$ writes). The ratio is clearly $\Omega(D)$. The intuition behind this example can be generalized to give a lower bound on the competitive ratio of any simple deterministic algorithm.

Theorem 7. *[10] Any simple deterministic algorithm with global M-block lookahead has a competitive ratio of $\Omega(D)$.*

A special case of this theorem, when the algorithm does not do any write-back, is representative of prefetching and buffer management algorithms normally used in practice. The theorem indicates that in the worst case such algorithms can be completely ineffective in exploiting the I/O parallelism, even when substantial lookahead – one memory load – is provided to them.

3 Randomized Algorithms

In this section we present randomized algorithms for read-once and read-many reference strings with considerably better performance than their deterministic counterparts.

3.1 Read-Once Reference Strings

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 access pattern of 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 $1/D$. Different 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, have been discussed in [2, 3, 9] for instance. We define a simple prefetching algorithm GREED using local lookahead below.

Definition 8. 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: on every parallel I/O it fetches the next block not in buffer from each disk provided there is buffer space available in the logical buffer for that disk. If there is no buffer space for a particular disk then no block is read from that disk.

In Theorem 9 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 take at least N/D I/Os to fetch N distinct blocks, GREED is within a constant factor of the optimal.

Theorem 9. *[9] 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.*

It is interesting to consider the role of partitioning the shared buffer in GREED. It may appear that by static partitioning of the buffer the realizable parallelism is being unnecessarily curtailed. An alternative is to have a single buffer of M blocks and prefetch from all disks whenever there are at least D free blocks in the buffer. If there are less than D free blocks, then only the demand block is fetched. Denote this algorithm by GREED*. Surprisingly, Theorem 10 shows that the performance of GREED* which uses an unpartitioned shared buffer can be actually worse.

Theorem 10. *[13] To service a reference string of length N, GREED* incurs $\Theta(N/D)$ expected number of I/Os, provided the buffer is of size $M = \Omega(D^2)$.*

Finally, we consider an algorithm NOM that uses M-block global lookahead. NOM behaves like GREED* except that only blocks within the current lookahead are prefetched. By adapting the results from [2] NOM can be shown to have performance bounds similar to GREED.

Theorem 11. *[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.*

3.2 Read-Many Reference Strings

In this section we sketch an on-line algorithm, RAND-WB with global M-block lookahead, which uses randomized write-back to attain a competitive ratio of $\Theta(\sqrt{D})$ [10].

Definition 12. A block present in the buffer is said to be *marked* when the lookahead window includes a reference to that block; the block is said to be *unmarked* otherwise.

In order to specify the blocks to be prefetched in any I/O it is useful to determine, for each disk, the next block not in the buffer to be referenced by the computation; we shall call this the *leading block* from each disk. We use Δ to denote the set of leading blocks from all disks in the lookahead window. Finally, let $\mathcal{B}$ denote the set of all blocks in the buffer.

The total buffer available to the algorithm, $B = M + D$, is logically partitioned into a *main buffer* of size M and a *write-back buffer* of size D. The write-back buffer is used to buffer a block from each disk in order to perform the write-backs efficiently. The algorithm uses global M-block lookahead. The number of free blocks in the buffer is denoted by $F = B - |\mathcal{B}|$; and the number of free blocks in the write-back buffer is denoted by W.

RAND-WB: On a request for a data block the following actions are taken:

1. If the requested block is present in the main or write-back buffer the request is serviced without any further action.
2. If the requested block is not present in either buffer a parallel I/O is initiated for all blocks in Δ. Some action is required, to create the necessary space for these blocks, if the number of free blocks in the buffer $F \leq |\Delta|$.
   - In this case, $\min(|\Delta| - F, D - W)$ unmarked blocks are moved from the main buffer to the write-back buffer and a write-back performed if the write-back buffer is full.
   - To perform the write-back the blocks are striped in a round robin fashion across all the disks starting the stripe from a randomly (uniform probability) chosen disk.
   - After performing the read all marked blocks in the write-back buffer are exchanged with some unmarked block in the main buffer.

The algorithm is based on the intuition that if blocks are relocated to randomly chosen disks then it is not possible for an oblivious adversary to generate a reference string for which requests to such relocated blocks are sequentialized. It can be noted that only the write-backs are randomized in the algorithm. No assumption is made regarding the initial placement of blocks on disks. In [10] we show that the ratio of the expected number of I/Os done by RAND-WB to those done by the optimal offline algorithm is given by Theorem 13 below.

Theorem 13. *The competitive ratio of the algorithm RAND-WB is* $\Theta(\sqrt{D})$.

4 Summary

In this paper we addressed the issues involved in designing prefetching and buffer management algorithms for multiple-disk parallel I/O systems. We showed that deterministic algorithms using bounded amounts of lookahead have poor parallel performance since useful parallelism is confined to their lookahead window. Randomization can be used to significantly improve the performance. Using randomization in the data layout and a simple prefetching scheme, we showed that a read-once reference string of length N can be serviced in $\Theta(N/D)$ parallel

I/Os, in a D-disk system. For the case of read-many reference strings we introduced a novel algorithm using randomized write-back with a competitive ratio of $\Theta(\sqrt{D})$. In contrast, we showed that deterministic write-back results in a competitive ratio of at least $\Omega(D)$.

References

1. Albers, S.: The influence of lookahead in competitive paging algorithms. Proc. 1st European Symp. on Algorithms, LNCS, Springer Verlag, (1993) 1–12
2. Barve, R. D., Grove, E. F., and Vitter, J. S.: Simple Randomized Mergesort on Parallel Disks. Parallel Computing, **23(4)** (1997) 601–631
3. Barve, R. D., Kallahalla, M., Varman, P. J., and Vitter, J. S.: Competitive Parallel Disk Prefetching and Buffer Management. Fifth Annual Workshop on I/O in Parallel and Distributed Systems, ACM (1997) 47–56
4. Belady, L. A.: A Study of Replacement Algorithms for Virtual Storage. IBM Systems Journal, **5** (1866) 78–101
5. Breslauer, D.: On Competitive On-Line Paging With Lookahead. Proc. Symp. Theoretical Aspects of Computer Science (1996) 593–603
6. Cao, P., Felten, E., Karlin, A., and Li, K.: A Study of Integrated Prefetching and Caching Strategies. Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems (1995) 188-197
7. Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson. D. A.: RAID: High-Performance Reliable Secondary Storage. ACM Computing Surveys **26(2)** (1994) 145–185
8. Fiat, A., Karp, R., Luby, M., McGeoch, L., Sleator, D. D., and Young, N. E: Competitive Paging Algorithms. J. Algorithms **12** (1991) 685–699
9. Kallahalla, M.: Competitive Prefetching and Buffer Management for Parallel I/O Systems Masters Thesis, Rice University (1997)
10. Kallahalla, M., and Varman, P. J.: Improving Competitiveness of Parallel-Disk Buffer Management Using Randomized Write-back. Tech. report, Dept. of Electrical and Computer Engineering, Rice University (1997)
11. Kimbrel, T., and Karlin, A.: Near Optimal Parallel Prefetching and Caching. 37th Ann. Symp. on Foundations of Computer Science (1996)
12. McGeoch, L. A., and Sleator, D. D.: A Strongly Competitive Randomized Paging Algorithm Algorithmica **6** (1991) 816–825
13. Pai, V. S., Schäffer, A. A, and Varman, P. J.: Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging. Theoretical Computer Science **128(1–2)** (1994) 211–239
14. Sleator, D. D., and Tarjan, R. E.: Amortized Efficiency of List Update and Paging Rules. Comm. ACM **28(2)** (1985) 202–208
15. Varman, P. J., and Verma, R. M.: Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. Proc. of the 1996 Symp. on Foundations of Software Tech. and Theoretical Computer Science, LNCS **1180** (1996)
16. Vitter, J. S., and Shriver, E. A. M.: Optimal Algorithms For Parallel Memory 1: Two-Level Memories. Algorithmica **12(2-3)** (1994) 110–147

This article was processed using the LaTeX macro package with LLNCS style