An Improved Parallel Disk Scheduling Algorithm*

Mahesh Kallahalla Peter J. Varman
Deptartment of Electrical and Computer Engineering
Rice University
Houston TX 77251
E-mail: {kalla,pjv}@rice.edu

Abstract

We address the problems of prefetching and I/O scheduling for read-once reference strings in a parallel I/O system. Read-once reference strings, in which each block is accessed exactly once, arise naturally in applications like databases and video retrieval. Using the standard parallel disk model with D disks and a shared I/O buffer of size M, we present a novel algorithm, Red-Black Prefetching (RBP), for parallel I/O scheduling. The number of parallel I/Os performed by RBP is within $O(D^{1/3})$ of the minimum possible. Algorithm RBP is easy to implement and requires computation time linear in the length of the reference string. Through simulation experiments we validated the benefits of RBP over simple greedy prefetching.

1. Introduction

Modern applications like multimedia servers, seismic databases, and visualization and graphics need access to large data sets that reside on external storage. The high data access rates demanded by such applications has resulted in the I/O subsystem becoming a significant performance bottleneck. The problem is exacerbated by the advent of multiprocessing systems that 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 adoption of multiple-disk arrays and distributed storage networks [14], provides an opportunity to improve I/O performance through the use of parallelism. The problem is to effectively use the increased storage bandwidth and reduce the I/O latency.

In this paper we address the problem of scheduling I/Os in a parallel I/O system. We use the intuitive parallel disk model introduced by Vitter and Shriver [18]: the I/O system consists of D independently-accessible disks and a shared I/O buffer of capacity M blocks. The data for the application is stored on the disks in blocks; a block is the unit of access from a disk. In each I/O up to D blocks, at most one from each disk, can be read from 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 it accesses. A block should be present in the I/O buffer before it can be accessed by the computation. Serving the reference string requires developing an I/O schedule to provide the computation with blocks in the order specified by the reference string. The measure of performance is the number of I/Os required to service the given reference string.

We concentrate on a class of computations characterized by read I/O accesses to *distinct* data blocks [3]. Streamed-data applications like multimedia retrieval and playback, scientific vector processing, database merging and filtering [2, 12], string processing and flow visualization for instance, largely involve such data-access patterns. For such *read-once* reference strings there is no benefit in caching blocks that have been referenced: we are guaranteed that they will never be requested again. Surprisingly, even with this simplification the "natural" intuitive schedules can be shown to be suboptimal.

Prefetching is a well-known technique to hide I/O latency in single-disk systems. It is particularly attractive in the context of parallel I/O as it provides a mechanism to exploit disk parallelism. Parallelism can be obtained by prefetching blocks from idle disks in parallel with I/Os on the disks fetching immediately needed data. Such prefetched data is cached in the I/O buffer so that a future access to it can be serviced directly from the I/O buffer without any disk access. Performing accurate rather than speculative prefetching requires knowledge of future I/O accesses of the application. There has been considerable recent work on how to gather such information from applications using program analysis and programmer provided hints [6, 11, 13].

In a single-disk I/O system prefetching can be used

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

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. (a) Greedy Schedule

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. (b) An Improved Schedule

to reorder requests so as to optimize the seek time [19], or to overlap computation and I/O. In the parallel I/O case prefetching presents a higher level scheduling problem as to which disks should be accessed in an I/O. As an illustration consider the following example of servicing a read-once reference string. Let the system consist of 3 disks and a buffer of capacity 6 blocks. Let the blocks labeled A_i (respectively B_i, C_i) be placed on disk 1 (respectively 2, 3), and the reference string be $A_1A_2A_3A_4B_1C_1A_5B_2C_2A_6B_3C_3A_7B_4C_4C_5C_6C_7$. For purposes of this example let us say that an I/O is initiated only when the referenced block is not present in the I/O buffer. Figure 1 (a) shows an I/O schedule constructed by a simple greedy algorithm that always fetches blocks in the order of the reference string, and maximizes the disk parallelism at each step. At step 1, blocks A_1 B_1 and C_1 are fetched concurrently in one I/O. When block A_2 is requested, blocks A_2, B_2 and C_2 are fetched in parallel in step 2. In 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 greedily in the order of the reference string means that we fetch B_3; continuing in this manner we obtain a schedule of length 9.

Figure 1 (b) presents an alternative schedule for the same reference string. At step 2 disk 2 is idle (even though there is buffer space) and C_2 that occurs later than B_2 in the reference string 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 greedily in the order of the reference string. The underlying optimization problem is to find a schedule that services the reference string in the least number of I/Os with the given I/O buffer.

The greedy prefetching algorithm mentioned has poor performance requiring, in the worst-case, $\Omega(D)$ times more I/Os than the optimal [3]. The worst-case performance can be improved to a ratio of $\Theta(\sqrt{D})$ by using the on-line algorithm NOM described in [3]. While NOM was shown to have the best possible ratio among all on-line algorithms with lookahead of M blocks, there appears no way to generalize it to do better with increased lookahead.

The main contribution of this paper is an I/O scheduling algorithm, Red-Black Prefetching (RBP), for read-once reference strings in a parallel I/O system. Algorithm RBP is easy to implement and requires computation time linear in the length of the reference string. Furthermore in the schedule generated the I/Os for any single disk are naturally aggregated, thereby allowing the system to exploit lower-level optimizations at the level of the disk controller. In addition RBP has superior worst-case performance: analysis shows that the worst-case number of I/Os performed by RBP is within $\Theta(D^{1/3})$ of the optimal.

The rest of the paper is organized as follows. A brief summary of related work is presented in Section 2. Algorithm RBP is described in Section 3. An outline of the analysis bounding the number of I/Os done by RBP relative to the optimal is presented in Section 4. Finally, an empirical evaluation of the algorithm is presented in Section 5.

2. Related Work

Algorithmic study of parallel I/O for read-once reference strings initially concentrated on particular applications such as external merging and sorting [2, 12, 18]. In [3] a general framework for studying scheduling and buffer management algorithms for read-once reference strings was introduced, and fundamental bounds were demonstrated.

Read-many reference strings where a block can be accessed repeatedly have been the focus of classical buffer management [4, 15]. These works primarily deal with the problem of caching and the impact of different eviction policies on the number of I/Os. In [5] a new model for integrated prefetching and caching was introduced. In this model the time to consume a block is an explicit parameter, and the elapsed time is the performance metric. Off-line approximation algorithms for a single-disk and multiple-disk systems in this model were addressed in [5] and [9] respectively. Recently a polynomial time optimal algorithm for the single disk case was discovered in [1]. In the parallel disk model used in this paper, a randomized caching and scheduling algorithm with bounded expected performance was presented in [8]. Using a distributed buffer configuration, in which each disk has its own private buffer, [17] presented an optimal off-line I/O scheduling algorithm to minimize the number of I/Os.

There has also been empirical work in improving the performance of parallel I/O systems through the use of prefetching. Several recent studies have also looked at improving the performance of prefetching and caching by using knowledge of the application's access patterns [6, 11, 13]. The benefits of performing collective I/O and efficient methods to implement it in a distributed I/O system were discussed in [7, 10].

3. Algorithm RBP

In this section we describe the I/O scheduling algorithm Red-Black Prefetching (RBP). As indicated before,

- Partition the I/O buffer into two parts, a *red buffer* of size M_r and a *black buffer* of size $M_b = M - M_r$. Each buffer will only be used to hold blocks of that color.
- A block of $phase(j)$ at depth i is colored red if the width of $phase(j)$ at depth i is less than the threshold width W, $w_i < W$; else it is colored black.
- On a request for block b, one of the following actions are taken:
- If b is present in either the red or the black buffer, service the request and evict block b from the corresponding buffer.
- If b is not present in either buffer then
  - Begin a batched-I/O operation as follows: If b is red (black), fetch the next M_r red (respectively M_b black) blocks beginning with b in order of reference, into the red (respectively black) buffer. These blocks are fetched with maximal parallelism.
  - Service the request and evict block b from the buffer.

Figure 2. Algorithm RBP

the reference string, $\langle r_0, r_1, r_2, r_3, \ldots, r_{N-1}\rangle$, denotes the ordered sequence of I/O accesses made by the computation. We consider read-once reference strings where all the blocks accessed are distinct; i.e., $r_i \neq r_j$ if $i \neq j$.

The details of the algorithm are presented in Figure 2. RBP labels each block of the reference string either red or black. The I/O buffer is also partitioned into red and black portions to hold blocks of the corresponding color. I/Os for each color are scheduled greedily and independently in such a way as to collectively reduce the total number of I/Os. Labeling of the blocks is the important part of the algorithm.

The reference string is partitioned into contiguous sequences of length M. Each such sequence is called a *phase*; that is, the ith phase, $i \geq 0$, denoted by $phase(i)$, is the sequence $\langle r_{Mi}, r_{Mi+1}, \ldots r_{M(i+1)-1}\rangle$, of M consecutive blocks of the reference string.

By definition, a phase consists of one memory-load of data. As the size of the I/O buffer is M, it is possible to fetch all the blocks of a phase using the fewest possible number of I/Os. This can be achieved by a simple greedy strategy which, in every I/O, fetches from every disk the next block in the phase. The number of I/Os required is just the maximum number of blocks from any single disk in that phase. Such a greedy strategy to service the reference string would optimize the number of I/Os *in each phase* as described above. However, the algorithm fails to exploit parallelism *across* phases. This results in a dilation of the schedule by a factor of $\Theta(\sqrt{D})$ over algorithms which exploit inter-phase parallelism [3].

Our approach therefore attempts to use inter-phase parallelism to reduce the number of I/Os. We do this by introducing a coloring scheme that partitions the blocks of a phase into two sets based on a threshold width W. The blocks in one set are colored *red* and those in the other set are colored *black*. Intuitively, the red blocks of a phase span a small (less than W) number of disks and form a narrow portion of a phase, while the black blocks constitute a wide portion.

To determine the color of a block we first determine the *depth* of the block. The depth of a block is one more than the number of blocks that are requested before it, from the same disk, in the same phase. For instance if a block b requested from disk d in $phase(i)$ is of depth k, then there are $k-1$ blocks in $phase(i)$ that are requested from disk d before b. Note that the number of I/Os that are needed to service a phase is directly given by the maximum depth of any block in that phase.

Blocks are colored based on the *width* of blocks at a particular depth. The width of a block is given by the number of blocks in that phase that have the same depth. Finally, the coloring is decided as follows: If the width of a block is less than W then it is colored *red* else it is colored *black*.

Figure 3 illustrates the coloring scheme with an example. Consider a system with 5 disks and an I/O buffer of capacity 16 blocks. Let the sequence of blocks in some phase be $phase(p) = \langle a_1 a_2 b_1 c_1 d_1 b_2 b_3 c_2 e_1 e_2 c_3 a_3 e_3 a_4 a_5 c_4\rangle$. Figure 3(a) shows the location of blocks on the disks. Figure 3(b) shows the depth of corresponding blocks; the number on the left hand side indicates the width assigned to all blocks that have a particular depth. For instance, as there are 4 blocks that have a depth 3, the width of all those blocks is 4. Finally, assuming a threshold width $W = 3$, all blocks that have a width less than 3 are colored red (in Figure 3(c) this corresponds to the grey portion), and all other blocks are colored black.

Intuitively the red blocks correspond to those blocks of a

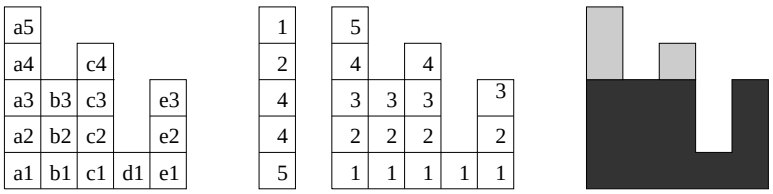


Figure 3. Illustration of definitions.

phase that span a few disks. If we attempt to service these requests in a phase-by-phase fashion, we may be unnecessarily sequentializing I/Os. A better alternative would be to parallelize I/Os for such blocks across phases instead. This is the intuition underlying RBP.

RBP logically partitions the I/O buffer into two parts: the *red* part of size M_r and the *black part* of size M_b, $M_r + M_b = M$. Each portion of the buffer is used to only hold blocks of that color. If the requested block is present in either of the buffers the request is serviced and the block immediately evicted from the corresponding buffer. If an I/O is required to fetch the requested block, a batched I/O is issued simultaneously on all or a subset of disks. All blocks fetched in the batched I/O are of the same color as the requested block. If this block is colored red (black), then the batched I/O issued is such that M_r (M_b) consecutive red (black) blocks starting from the requested block are fetched.

Note that within a particular color blocks are fetched in the order they appear in the reference string. Hence if a block of a particular color is missing from the buffer, there are no blocks of that color in the buffer at that time. This ensures that we can always issue the batched I/Os for M_r (or M_b) blocks is described above. The total number of I/Os is no more than the maximum number of blocks fetched from any single disk.

The algorithm leaves open the choice of the threshold width W and the amount of buffer allocated to the two buffers. In Theorem 1 we show that the ratio of the number of I/Os done by RBP to the total number of I/Os done by the optimal algorithm can be bounded by $\Theta(D^{1/3})$, by taking the threshold width as $D^{1/3}$ and dividing the buffer equally between red and black. In practice, further improvements (by constant factors) can be achieved by choosing the sizes of the red and block buffers adaptively, based on the characteristics of the reference string. Section 5 presents an empirical study of the sensitivity of RBP to the buffer partition sizes.

4. Analysis

In this section we outline the analysis of RBP. For reasons of space, we give only an intuitive idea behind the proofs. We shall compare the number of I/Os done by RBP to that of the optimal off-line algorithm, denoted by OPT. OPT produces a minimal length schedule for any reference string. Interestingly, though we do not know OPT, we can get tight bounds on the performance of RBP relative to it.

The measure of performance that we use in this discussion is the approximation ratio: in the current context this is the worst case ratio over all reference strings, of the number of I/Os done by RBP to those done by OPT. In other words it is the worst case dilation in the schedule generated by RBP compared to that generated by OPT. The bound on the approximation ratio of RBP is given by the following theorem.

Theorem 1 *The approximation ratio of RBP is bounded by* $\Theta(D^{1/3})$.

Proof : The I/Os for RBP can be divided into I/Os done to fetch black blocks and the I/Os to fetch red blocks. We break our analysis into two parts. We show in Lemma 2 that the ratio of the number of I/Os done by RBP for the red blocks to the total number of I/Os done by OPT is $O(\sqrt[4]{DW})$ when $M_r = M/2$. We then show in Lemma 4 that the ratio of the number of I/Os done by RBP for the black blocks to the total number of I/Os done by OPT is $O(\sqrt{D/W})$ when $M_b = M/2$. Together, we then note that the approximation ratio, is bounded above by $O(D^{1/3})$ by choosing $W = D^{1/3}$. Finally, the lower bound is shown by constructing an example reference string for which the ratio is $\Omega(D^{1/3})$. □

In order to bound the ratio of the number of I/Os done by RBP for red blocks to the total number of I/Os done by OPT, we divide the reference string into *swaths*. Each swath is a minimal set of contiguous phases, such that there are at least M red blocks in it. As a phase has at most M blocks, it has at most M red blocks, and hence a swath contains at most $2M - 1$ red blocks. By noticing that RBP issues a batched I/O for $M_r = M/2$ blocks at a time we get the following lemma.

Lemma 1 *The number of I/Os done by RBP for red blocks in the* j*th swath is at most* $4H_r(j)$*, where* $H_r(j)$ *is the maximum number of red blocks on a single disk in that swath.*

The above lemma provides an upper bound on the number of I/Os done by RBP in each swath. Next we bound the

number of I/Os done by OPT in the phases that make up a swath. To do so we introduce the notions of *peak* and *useful* block, that intuitively keep track of the number of I/Os that still need to be done by OPT and the number of I/Os that OPT has saved compared to RBP respectively. The peak of a swath is the maximum number of red blocks of that swath from any single disk, that are not present in the I/O buffer. The number of useful blocks prefetched by OPT keeps track of the difference between the sum of peaks of all future swaths seen by OPT and RBP respectively. From the preceding bound on the number of I/Os needed by RBP the number of useful blocks can be seen as a measure of the number of I/Os OPT has saved compared to RBP.

The notions of peak and useful blocks play a central role in bounding the number of I/Os done by OPT. We consider a sequence of contiguous swaths such that the number of useful blocks prefetched by OPT in that sequence is at most $2M$. In each such *super-swath* we show that the ratio of the number of red I/Os done by RBP is within $O(\sqrt[4]{DW})$ of the total number of I/Os done by OPT.

The intuition behind the proof is that if OPT prefetched a lot of useful blocks for a single swath then it must have done a large number of I/Os to fetch them. On the other hand if OPT did not prefetch many useful blocks then its advantage over RBP is limited. This could still result in a high approximation ratio if the number of I/Os done by OPT itself were small. But then by noting that the width of red blocks is at most $D^{1/3}$, we show that a super-swath will include a substantial number of phases, and hence OPT will need to do sufficient number of I/Os to ensure the bound. The formalization of these ideas to get the desired bound is involved and the details are omitted from this paper. Lemma 2 gives the required bound concerning I/Os done by RBP for the red blocks of a reference string.

Lemma 2 *The ratio of the number of I/Os done by RBP to fetch the red blocks to the total number of I/Os done by OPT is* $O(\sqrt[4]{DW})$.

Next we bound the number of I/Os done by RBP to fetch black blocks of the reference string, relative to the total number of I/Os done by OPT. The outline of the analysis in this case parallels that used for red blocks except that the analysis proceeds on a phase-by-phase rather than a swath-by-swath basis.

Lemma 3 *The number of I/Os done by RBP for the black blocks of* $phase(i)$ *is at most* $2H_b(i)$*, where* $H_b(i)$ *is the maximum number of black blocks on a single disk in* $phase(i)$.

We define notions of peak and useful blocks with respect to phases and black blocks. An important observation is that if the number of useful blocks in the buffer is U, then the number of blocks in the buffer is at least UW. This is because only those blocks that are more than W wide are colored black. Hence if at some stage there are U useful blocks intended for a phase in the future, at each intermediate phase, there are UW blocks in the buffer. This in turn reduces the buffer available for OPT to service intermediate phases forcing OPT to perform a certain number of extra I/Os. By carefully accounting for these I/Os and the I/Os done to fetch useful blocks, we get the desired bound on the ratio of the number of I/Os done by RBP for black blocks to the total number of I/Os done by OPT.

Lemma 4 *The ratio of the number of I/Os done by RBP to fetch the black blocks to the total number of I/Os done by OPT is* $O(\sqrt{D/W})$.

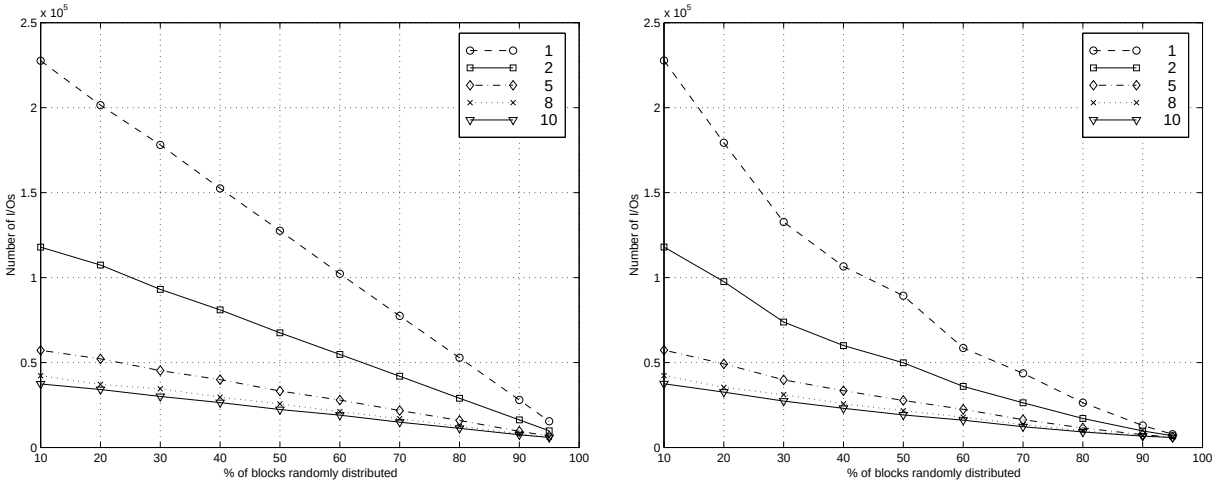
5. Simulation Results

In this section we present an empirical evaluation of the algorithm RBP. Synthetic reference strings were generated using a bursty model described below. We compare the performance of RBP with the natural greedy prefetching algorithm, and study the sensitivity of RBP to the buffer partitioning. The measure of performance is the number of parallel I/Os needed by the algorithm to service the reference string.

The implementation of the natural greedy algorithm is based on the algorithm NOM described in [3]. Intuitively it works as follows. It issues an I/O when a requested block is not present in the I/O buffer. On every such I/O the algorithm attempts to prefetch blocks as follows: from any disk it prefetches the earliest required block, provided the block lies within the next M references.

It is interesting to consider the reason why a block is prefetched only if it lies within the next M references. An alternative aggressive I/O scheduling algorithm that may be considered is one that prefetches a block from all disks whenever there is space in the buffer to hold D blocks. Such an aggressive algorithm has the disadvantage of possibly prefetching too far into the future from a few disks. In turn this may fill the I/O buffer with blocks that will not be consumed for a long time, choking the buffer and resulting in low I/O parallelism. In the worst case such an aggressive algorithm can do $\Theta(D)$ times more I/Os than the optimal [3].

The sequence of references used to evaluate the algorithms was generated using a bursty model of data. In this model, the sequence of accesses are usually random; that is, they access any disk with uniform probability. But occasionally there is burst of I/O requests to a small set of k *hot-spot* disks. The set of hot-spot disks changes dynamically as the computation progresses. Specifically, the model takes two parameters (a) the fraction f of random blocks and (b) the number k of hot-spot disks. Data was generated in rounds, with each round consisting of M accesses. The



(a) Algorithm NOM (b) Algorithm RBP

Figure 4. Performance of NOM and RBP vs. percentage of random blocks, for different values of k

set of hot-spot disks in each round was chosen randomly from among the set of all disks. In a round, the first Mf blocks were randomly requested from all disks and then $M(1-f)$ blocks were requested randomly from among the k hot-spot disks.

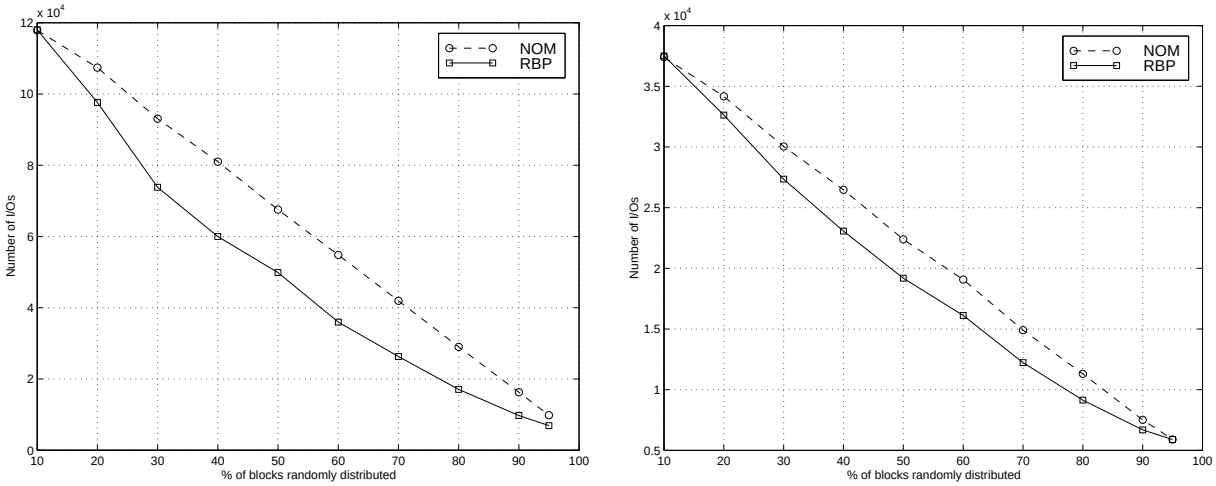
Both $f = 1$ and $f = 0$ represent extremes in the spectrum of possible access patterns. When $f = 1$ then the data is completely random so that any disk may be accessed in an I/O with equal probability. In this case the simple I/O scheduling policy that fetches greedily from all disks suffices to achieve good I/O parallelism. On the other hand the case when $f = 0$ represents the situation when in a phase the data accessed is from a small set of hot-spot disks. In this case, there is much less potential for I/O parallelism since data is concentrated on just a few disks; hence prefetching will not provide much benefit in this case. Other choices of f represent interesting access patterns where there is substantial latent I/O parallelism available in the reference string which can be exploited by a suitable prefetching algorithm. For our results presented we consider an I/O system consisting of 100 disks and an I/O buffer of 5000 blocks.

Figure 4 shows the number of I/Os performed by NOM and RBP as the percentage of random blocks in the reference string is varied. Each line in the plot corresponds to a fixed number of hot-spot disks. For both NOM and RBP we can notice a decrease in the total number of I/Os performed as the percentage of random blocks increases. Intuitively this is due to the fact that as the percentage of random blocks increases, in any given segment of the reference string there are a lot more blocks that are uniformly distributed across all disks. In effect for RBP this means that the number of black blocks increases as the percentage of random blocks increases. On the other hand when the percentage of random blocks is low the number of red blocks is higher; that is, more blocks are concentrated on a few disks. This requires them to be serviced with much less parallelism and subsequently requires a larger number of I/Os.

As expected, the plot of NOM is linear with the percentage of random blocks, as the effective overall time can be approximated by the expression: $N(f/D + (1-f)/2k)$, where f is the fraction of random blocks, k the number of hot-spot disks, and N the total number of blocks in the reference string. Intuitively, the fN random blocks are fetched with close to full parallelism (D), while the rest are got with parallelism limited by the number of hot-spot disks. As a few blocks from the next round can be got in parallel with the I/Os on hot-spot disks, the effective parallelism for such blocks is doubled.

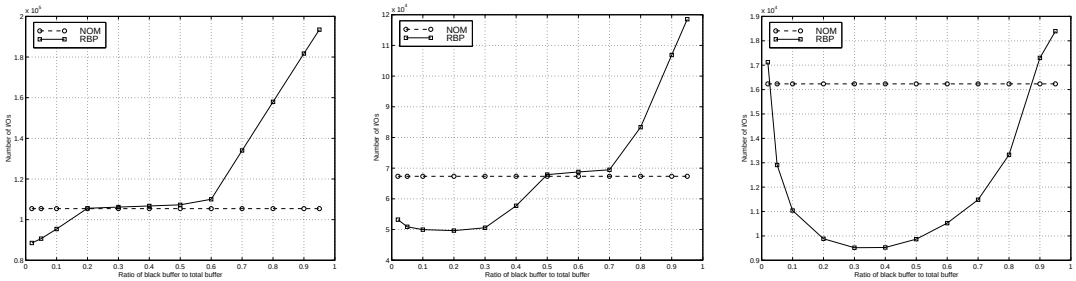
On the other hand RBP shows non-linear behavior as it exploits parallelism across burst periods through I/Os for red blocks. That is even if there is very little parallelism to be exploited during a single burst period, RBP exploits the fact that the hot-spot disks differ across burst periods. As the percentage of random blocks increases, the number of burst periods that RBP needs to look at to fetch the same number of red blocks increases. Hence the chance that some disk may be common across many burst periods increases, reducing the effectiveness of prefetching across burst periods. This is the reason why the slope of the plot for RBP decreases with increasing f.

From the same figure we can note that for a given percentage of random blocks, as the number of hot-spot disks increases, the number of I/Os done by both the algorithms decreases. This can be explained from the fact that for a given f, as k increases, the average parallelism during the burst period increases because the blocks are distributed over a larger number of disks. RBP tends to flatten out



(a) 2 hot-spot disks (b) 10 hot-spot disks

Figure 5. Comparison between NOM and RBP for a fixed number of hot-spot disks



(a) 20% random blocks (b) 50% random blocks (c) 90% random blocks

Figure 6. Sensitivity of RBP to the amount of black buffer, when $k = 2$

for large k as prefetching across bursts becomes less effective due to a higher probability of the sets of hot-spot disks across bursts overlapping. At this stage the behavior of RBP approaches that of NOM.

A closer comparison between the performance of RBP and NOM for a fixed number of hot-spot disks, is presented in Figure 5. For brevity, we present results for only two sample values of k: 2 and 10. Similar trends hold for other values of k. These illustrate two extremes in the structure of the reference string. When $k = 2$, the bursts are significantly serialized, while when $k = 10$ there is a reasonable amount of parallelism even within a burst.

In both cases, RBP out-performs NOM, though the difference is greater when $k = 2$. This is because for smaller k the references within a burst are essentially sequential, but across bursts there is substantial parallelism that is exploited by RBP.

Figure 6 shows the sensitivity of RBP to the partitioning of the buffer. It shows the variation in the number of I/Os done by RBP with the fraction of the buffer allocated to black blocks, when the percentage of random blocks is 20, 50 and 100 respectively. In all three cases the number of hot-spot disks was kept at 2. The horizontal line in all three cases is the number of I/Os done by NOM, that is, of course, independent of the way the buffer is partitioned.

An important point shown by these results is that the performance of RBP is very sensitive to the amount of buffer allocated to the two colors. When the percentage of random blocks is 20%, only around 1% of the buffer needs to be allocated for black blocks. On the other hand even when the fraction of random blocks is 90%, around 60% of the buffer needs to be allocated for the red blocks. This asymmetry is because even when f is large and the bursts are small, there is still benefit to be got by prefetching across bursts. In fact it is still worthwhile to reserve a significant portion of the buffer for this purpose. This is due to the fact that when the accesses are uniform, the time that a prefetched block spends in the buffer before it is consumed is very small, and hence it is enough to reserve a relatively small buffer for black blocks. On the other hand even though there are fewer red blocks, once fetched into the buffer they are retained for a much longer time necessitating a larger portion

of the buffer to be reserved for such blocks.

In general, when too small a fraction of the buffer is reserved for the random blocks, the number of I/Os that RBP does for black blocks is very substantial. The situation is alleviated as more buffer is allocated for the black blocks. But beyond a threshold, the penalty of having insufficient buffer for red blocks causes an increase in the total number of I/Os. This is because the extra buffer for black blocks does not provide a compensating decrease in the number of I/Os for black blocks. Further, when the size of red buffer is enough to hold the current burst completely (at f), the performance of RBP matches that of NOM. When the red buffer is further decreased, even intra-burst parallelism cannot be exploited sufficiently and the number of red I/Os increases drastically. These results were further validated by experiments done by varying the number of hotspot disks. These results have been omitted for brevity.

Though the simulations considered in this study were based on a simple model of the I/O system, the basic results would carry over to a more detailed model of reasonable size. By scheduling I/Os across phases, RBP is able to exploit parallelism not evident to intuitive greedy algorithms. In addition, the batched I/O nature of RBP allows a lower-level disk scheduler to optimize the accesses based on the parameters of that particular device. A promising avenue for future experimentation is to consider a model that allows parallelism across disks in the I/O system, as well as takes into account the actual disk head and buffer optimizations available for a single disk.

6. Conclusions

In this paper we considered the problem of prefetching and I/O scheduling of read-once reference strings in a parallel I/O system. We presented a novel algorithm, Red-Black Prefetching (RBP), for this problem and bounded its approximation ratio. We showed that the number of I/Os performed by RBP is within $\Theta(D^{1/3})$ of the optimal schedule. The algorithm is easy to implement and computationally requires time linear in the length of the reference string. The previous best bound applicable to this problem had an approximation ratio of $\Theta(\sqrt{D})$. Simulation experiments have validated the benefits of RBP over simple greedy prefetching.

References

[1] S. Albers, N. Garg, and S. Leonardi. Minimizing Stall Time in Single and Parallel Disk Systems. In *Proc. of STOC'98*, 1998.

[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] 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.

[6] P. Cao, E. W. Feltin, A. Karlin, and K. Lee. Implementation and Performance of Application-Controlled File Caching. In *Proc. of OSDI'94*, pages 165–178, November 1994.

[7] J. M. del Rosario, R. Bordawaker, and A. Choudhary. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In *IPPS'93 Workshop on I/O in Parallel Computer Systems*, pages 56–70, 1993.

[8] M. Kallahalla and P. J. Varman. Improving Parallel-Disk Buffer Management using Randomized Writeback. In *Proc. of ICPP'98*, pages 270–277, August 1998.

[9] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *Proc. of FOCS'96*, pages 540–549. IEEE, October 1996.

[10] D. Kotz. Disk Directed I/O for MIMD Multiprocessors. *ACM Transactions on Computer Systems*, 15(1):41–74, February 1997.

[11] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Multiprocessor File Systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, 1993

[12] 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.

[13] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of ASPLOS'95s*, pages 79–95, December 1995.

[14] B. Phillips. Have Storage Area Networks Come of Age. *IEEE Computer*, 31(7): 10–12, July, 1998.

[15] 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.

[16] P. J. Varman. Randomized Parallel Prefetching and Buffer Management. Parallel and Distributed Computing, LNCS, 1388: 363–372, 1998.

[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, December 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.

[19] L. Q. Zheng and Per-Åke Larson. Speeding up External Mergesort. *IEEE Transactions on Knowledge and Data Engineering*, 8(2):322–332, April 1996.