

ASP: Adaptive Online Parallel Disk Scheduling

Mahesh Kallahalla and Peter J. Varman

ABSTRACT. In this work we address the problems of prefetching and I/O scheduling for read-once reference strings in a parallel I/O system. We use the standard parallel disk model with D disks and a shared I/O buffer of size M . We design an on-line algorithm ASP (Adaptive Segmented Prefetching) with ML -block lookahead, $L \geq 1$, and compare its performance to the best on-line algorithm with the same lookahead. We show that for any reference string the number of I/Os done by ASP is with a factor $\Theta(C)$, $C = \min\{\sqrt{L}, D^{1/3}\}$, of the number of I/Os done by the optimal algorithm with the same amount of lookahead.

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 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 availability of cost-effective multiple-disk storage systems [CLG⁺94] provides an opportunity to improve the I/O performance through the use of parallelism. However it remains a challenging problem to use the increased disk bandwidth effectively and reduce the I/O latency of an application.

The parallel I/O system is modeled using the intuitive parallel disk model introduced by Vitter and Shriver [VS94]: the I/O system consists of D independently-accessible disks and an associated I/O buffer with a capacity of M blocks, 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 I/O up to D blocks, at most one 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. 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. The measure of performance of the system is the number of I/Os required to service a given reference string. In this

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

To appear in the Proceedings of DIMACS Wkshp. on External Memory Algorithms and Visualization 1998.

work we consider *read-once* reference strings in which each block is read exactly once. Such reference strings arise naturally in database operations such as external merging [BGV96, PSV94] (including carrying out several of these concurrently), joins and real-time retrieval and playback of multiple streams of multimedia data.

I/O parallelism is obtained by *prefetching* blocks from the idle disks in parallel with the block currently requested by the computation. These prefetched blocks are buffered until required. In order to prefetch accurately, the I/O scheduling algorithm needs to have some knowledge regarding future accesses. [BKVV97] introduced the notion of *M-block lookahead* to model this information. An algorithm having this form of lookahead knows the sequence of next M blocks beyond the currently referenced block. [BKVV97] showed that any algorithm having M -block lookahead can require $\Omega(\sqrt{D})$ times as many I/Os as the optimal *off-line* algorithm. They also presented a simple algorithm, NOM, which achieves this bound.

We are interested in designing on-line algorithms which can use ML -block lookahead, where $L \geq 1$. One straightforward approach to use ML -block lookahead is to be greedy: on every I/O fetch, from each disk, the next block in the lookahead not present in the buffer. However, such an aggressive policy can require $\Omega(D)$ times more I/Os than the optimal on-line algorithm (when $L \geq 2$). On the other hand NOM, which is greedy only within M requests, can be shown to require $\Theta(C)$, $C = \min\{L, D^{1/2}\}$, times as many I/Os as the optimal on-line algorithm using ML -block lookahead. The fact that NOM uses only the next M requests is intrinsic to the algorithm and there does not seem to be any way to generalize it to make use of the additional lookahead.

The main result of this paper is a new prefetching and scheduling algorithm called ASP (Adaptive Segmented Prefetching), with improved on-line performance. ASP uses ML -block lookahead to schedule I/Os. The number of I/Os performed by ASP is within a factor $\Theta(C)$, $C = \min\{\sqrt{L}, D^{1/3}\}$, of the optimal on-line algorithm with ML -block lookahead. The only other result we are aware of for scheduling read-once reference strings in the parallel disk model is the algorithm RBP whose competitive ratio was shown to be $\Theta(C)$, $C = \max\{\sqrt{D/L}, D^{1/3}\}$ [KV98]. The relation between RBP and ASP is further discussed in Section 4.2.

Classical buffer management which primarily deals with optimizing buffer evictions has been studied extensively in sequential I/O models [Bel66, ST85, BGV95]. In the parallel disk model of this paper, a randomized caching and scheduling algorithm using M -block lookahead was presented in [Var98]. Using a distributed buffer configuration, in which each disk has its own private buffer, [VV96] presented an optimal off-line I/O scheduling algorithm. An interesting alternative measure of performance is the elapsed or stall time that includes the time required to consume a block as an explicit parameter. Off-line approximation algorithms for a single-disk and multiple-disk systems in this model were addressed in [CFKL95] and [KK96] respectively. Recently a polynomial time optimal algorithm for the single disk case was presented in [AGL98].

The rest of this paper is organized as follows. In Section 2 we introduce some notation and definitions. In Section 3 we present the algorithm ASP. The analysis is done in two parts: in Section 4.1 we bound its performance for lookahead $L \leq D^{2/3}$, while the case of $L > D^{2/3}$ is analyzed in Section 4.2.

2. Definitions

The I/O system is modeled by the Parallel Disk Model [VS94] with D disks and a buffer of capacity M blocks, $M \geq 2D$. The sequence of accesses to the I/O system is modeled by a reference string which is the ordered sequence of blocks accessed by the computation. The reference string is partitioned into *phases*, where each phase corresponds to a buffer-load of I/O requests. We quantify the performance of an on-line algorithm by comparing it with the optimal on-line algorithm that has the same amount of lookahead. This motivates the definition of *on-line ratio*, which is a measure of how effectively a given algorithm uses the lookahead available to it.

DEFINITION 2.1. Let the reference string be $\Sigma = \langle r_0, r_1, \dots, r_{N-1} \rangle$.

- The i th phase, $i \geq 0$, $phase(i)$, is the substring $\langle r_{iM}, \dots, r_{(i+1)M-1} \rangle$.
- If the last block referenced is r_i , then an on-line scheduling algorithm has *ML-block lookahead* if it knows the substring $\langle r_{i+1}, \dots, r_{i+ML} \rangle$.
- Let \mathcal{C}_L be the set of all algorithms with ML -block lookahead. For any algorithm $A \in \mathcal{C}_L$, let $T_A(\Sigma)$ denote the number of I/Os needed by A to service Σ . The *optimal on-line algorithm* with ML -block lookahead is the algorithm $OPT \in \mathcal{C}_L$, such that $T_{OPT}(\Sigma) \leq T_A(\Sigma)$, for every Σ .
- An on-line scheduling algorithm $\mathcal{A} \in \mathcal{C}_L$ has an *on-line ratio* of C_A if there is a constant b such that for any reference string Σ , $T_{\mathcal{A}}(\Sigma) \leq C_A T_{OPT}(\Sigma) + b$.

The performance of on-line algorithms has traditionally been studied through competitive analysis [ST85]. The competitive ratio, the usual measure of performance in this framework, is the worst case ratio of the number of I/Os needed by the on-line algorithm to that required by the optimal off-line algorithm to service a reference string. The competitive ratio of an algorithm with a certain amount of lookahead is influenced by two factors: unknown information about the reference string beyond the lookahead, and the inability of the algorithm to effectively exploit information available in the lookahead. The competitive ratio does not differentiate between these two factors and hence cannot distinguish algorithms for which the contribution of the former factor dominates. A complementary measure of performance, the comparative ratio, was introduced in [KP94]. This measure tries to quantify the performance loss of an on-line algorithms due to the unknown portion of the reference string beyond the lookahead. In contrast the on-line ratio measures how well the on-line algorithm exploits the information available in the lookahead.

3. Adaptive Segmented Prefetching

In this section we present the algorithm ASP, that constructs a schedule for a read-once reference string. ASP partitions the reference string into *segments*: a segment is a sequence of contiguous phases. The I/O schedule for each segment is generated by an algorithm THIN. The schedule for the overall reference string is obtained by concatenating the individual schedules. ASP uses a dynamic programming method to adaptively partition the reference string into segments. This procedure is presented later. We first present the algorithm THIN that generates the schedule for a given segment.

THIN colors each block of a segment either red or black. The buffer is also partitioned into a red buffer and a black buffer, each of size $M/2$. When a requested block is not present in the buffer, a batched I/O is initiated. If the block is red then the next $M/2$ red blocks are fetched into the red buffer. Similar action is

taken when the block is black. The coloring of the blocks by THIN is based on the following definitions.

DEFINITION 3.1. Within a phase, a block on some disk has a *depth* k if there are $k - 1$ blocks from that disk referenced before it in that phase. The set of blocks in a phase with the same depth is called a *stripe*. The *width* of a stripe is the number of blocks in that stripe. The maximum depth of any block in a phase is called the *max-depth* of that phase.

Figure 1 illustrates the definitions above. Note that there can be at most D blocks in a stripe. The max-depth of $phase(i)$ is the minimum number of I/Os required for $phase(i)$ if no blocks of that phase have been fetched prior to the start of $phase(i)$. Each stripe present in the buffer at the start of a phase guarantees that the phase can be serviced in one I/O less than its max-depth. The width of a stripe indicates how much buffer space needs to be allocated to reduce the number of I/Os by one: the narrower a stripe is the lesser space it needs for the same benefit. We use \sum_{η} to denote the sum over all k , such that $phase(k)$ belongs to a segment η .

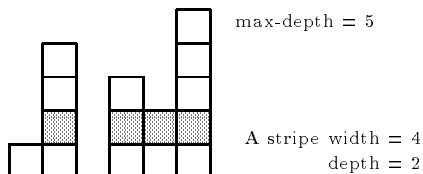


FIGURE 1. Illustration of a stripe

The blocks of a segment are classified as red or black depending on the widths of the stripes in the segment. Red blocks belong to stripes that are no wider than the stripe of any black block. Thus the red blocks of a phase span a small number of disks while the black blocks span a larger number. The details of algorithm THIN are presented in Figure 2. THIN is a building block which is used by ASP to generate the overall I/O schedule.

DEFINITION 3.2. If h_k is the maximum number of red blocks from a single disk in $phase(k)$ and R the maximum number of red blocks from a single disk in segment η , then the *benefit* of THIN in that segment is defined as $B_{\text{THIN}}(\eta) = \sum_{\eta} h_k - R$.

If the red blocks of η were fetched on a phase-by-phase basis, the number of I/Os required would be at least $\sum_{\eta} h_k$. The number of I/Os done by THIN to fetch these blocks in a batched manner is proportional to R .

LEMMA 3.3. *The number of I/Os done by THIN in a segment η is at most $T_{\text{THIN}}(\eta) \leq 3(\sum_{\eta} d_k - B_{\text{THIN}}(\eta))$, where d_k is max-depth of $phase(k)$.*

PROOF. Since a stripe has at most D blocks, the total number of red blocks in a segment is at most $M + D - 1$. In each batched-I/O THIN fetches $M/2$ red blocks; hence at most 3 batched-I/Os are required to fetch all the red blocks of a segment. Since the maximum number of red blocks from any single disk in the segment is R , in each batched-I/O operation for red blocks THIN performs no more than R I/Os.

By a similar argument at most 2 batched-I/Os are required to fetch the black blocks of a phase. Since the maximum number of red blocks from a single disk in $phase(k)$ is h_k , then the maximum number of black blocks from a single disk in

Algorithm THIN takes as input a segment $\eta = \langle \text{phase}(i) \dots \text{phase}(i') \rangle$, and generates an I/O schedule to service this segment.

- Partition the I/O buffer into two parts, *red* and *black*, each of size $M/2$. Each half of the buffer will only be used to hold blocks of that color.
- Order the stripes in a segment in increasing order of their width, breaking ties by giving priority to stripes which occur in earlier phases and, within a phase to a stripe with a larger depth. Choose the minimal number of stripes using the above ordering such that the total number of blocks in these stripes is at least M : color all these blocks red. All other blocks are colored black.
- On a request for block b , one of the following actions is 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 as follows: If b is red (black), fetch the next $M/2$ red (respectively black) blocks beginning with b in order of reference, into the red (respectively black) buffer. These $M/2$ blocks are fetched with maximal parallelism.
 - * Service the request and evict block b from the buffer.

FIGURE 2. Algorithm THIN

$\text{phase}(k)$ is $d_k - h_k$. Hence the maximum number of I/Os performed in each of the two batched-I/Os is at most $d_k - h_k$. Therefore, the total number of I/Os done by THIN in a segment is $T_{\text{THIN}}(\eta) \leq 2 \sum_{\eta} (d_k - h_k) + 3R$. The desired inequality now follows from the definition of B_{THIN} . \square

Algorithm ASP partitions the reference string into segments and schedules each segment independently using THIN. The total number of I/Os done by ASP is therefore $\sum T_{\text{THIN}}(\eta)$, where the sum is taken over all segments η in Σ . From Lemma 3.3 minimizing $\sum T_{\text{THIN}}(\eta)$ translates to maximizing $\sum B_{\text{THIN}}(\eta)$ as $\sum d_k$ over all phases in the reference string is independent of the partitioning.

DEFINITION 3.4. Algorithm ASP partitions the reference string into disjoint segments $\langle \eta_1, \dots, \eta_n \rangle$ such that $\sum_{j=1}^n B_{\text{THIN}}(\eta_j)$, is maximized over all possible partitions of the reference string. The I/O schedule is then constructed by concatenating the schedules generated by THIN for each segment η_j .

A dynamic programming approach can be used to find the partitioning. Consider a graph where node (i, j) , $0 \leq i \leq j < L$, denotes that $\text{phase}(j)$ is the last phase of the i th segment. An edge from node $(i-1, k)$ to node (i, j) indicates that the i th segment is from $\text{phase}(k+1)$ to $\text{phase}(j)$. The weight of this edge, $w_{i,j,k}$, is the benefit of THIN for this segment. Associated with node (i, j) is $b_{i,j}$ where $b_{i,j} = \max_{i-1 < k < j} \{b_{i-1,k} + w_{i,j,k}\}$. The entries $b_{i,L-1}$, give the maximum benefit possible for a partitioning into $i+1$ segments. Hence the maximum benefit is $\max_{0 \leq i < L} \{b_{i,L-1}\}$. A straightforward implementation of the above dynamic programming algorithm has complexity $O(ML^3 + ML \log(ML))$. This can be improved to $O(ML + DL^2 + L^3)$ by working with a concise representation of each

phase consisting of D buckets, one for each possible width. We omit the details of the scheme for brevity.

4. Analysis of algorithm ASP

In this section we present tight bounds on the on-line ratio of ASP. Let OPT denote the optimal on-line algorithm with ML -block lookahead. First, note that any I/O schedule can be transformed into another schedule of the same length, or less, in which a block is never evicted before it has been referenced. Hence we shall implicitly assume this property for all the schedules considered in this paper.

DEFINITION 4.1. An I/O is said to be performed in *phase*(i) if the next block to be referenced is in *phase*(i). The *start* of a phase (respectively segment) refers to the first reference of that phase (segment). An *inter-segment* block is one, which is fetched in a segment different from the one in which it is referenced. Similarly an *inter-phase* block is one which is fetched in a phase different from the one in which it is referenced.

The bounds of ASP are presented in two ranges: $L \leq D^{2/3}$ and $L > D^{2/3}$. In the lower range the on-line ratio is shown to be $\Theta(\sqrt{L})$ in Theorem 4.9. An on-line ratio of $\Theta(D^{1/3})$ is shown in Theorem 4.11 for $L > D^{2/3}$.

THEOREM 4.2. *The on-line ratio of ASP is $\Theta(\sqrt{L})$ when $L \leq D^{2/3}$, and $\Theta(D^{1/3})$ when $L > D^{2/3}$.*

To simplify the analysis, in the subsequent sections we implicitly assume that the length of the reference string is ML . This is justified because of the following observations. Let the reference string $\Sigma = \langle l_1, l_2, \dots, l_n \rangle$, where each l_i is a substring of Σ with ML requests. Since OPT has ML -block lookahead, during l_i OPT might prefetch a block from l_{i+1} , but never beyond l_{i+1} . In contrast, consider a schedule generated by independently scheduling each l_i optimally. It is easy to show that such a schedule is within a factor of 2 of OPT. Hence the on-line ratio of an algorithm A for arbitrary reference strings, can be computed to within a factor 2 by computing the on-line ratio of A over reference strings of length ML .

4.1. Analysis of ASP for $L \leq D^{2/3}$. In the range $L \leq D^{2/3}$ we proceed as follows. We define a weaker form of ASP, called Fixed Segment Prefetching (FSP). FSP *statically* partitions the reference string into fixed-size segments and, like ASP, services each segment independently using THIN. Since the segments used by FSP is just one of the many partitions considered by ASP, the number of I/Os required by ASP is no more than that required by FSP. Hence we bound the on-line ratio of ASP by bounding the on-line ratio of FSP.

DEFINITION 4.3. Algorithm FSP partitions the reference string Σ into \sqrt{L} segments of equal length, $\Sigma = \langle \eta_0, \dots, \eta_{\sqrt{L}-1} \rangle$. Each η_j thus consists of \sqrt{L} consecutive phases. The I/O schedule for Σ is constructed by concatenating the schedules generated by THIN for each segment η_j .

To bound the on-line ratio of FSP we proceed in two steps. We first define a weaker form of OPT, called OPT*, which performs $O(\sqrt{L})$ times as many I/Os as OPT. We then show that FSP does $O(1)$ times as many I/Os as OPT*. The construction of OPT* is described in Figure 3. OPT* is constructed from OPT by splitting each I/O done by OPT into I/Os performed currently for the present

phase, I/Os performed at the start of other phases of the same segment, and I/Os performed at the start of other segments. By assumption, no block fetched by OPT is evicted before it is referenced. Hence we can safely delay an I/O for a block without overflowing the buffer at the new time it is fetched. The total number of segments is \sqrt{L} and each segment has \sqrt{L} phases. Hence, each I/O of OPT is dilated to at most $2\sqrt{L}$ I/Os in OPT*.

LEMMA 4.4. *The number of I/Os done by OPT* to service Σ is within a factor $2\sqrt{L}$ of the number of I/Os done by OPT to service Σ .*

- Partition the reference string into \sqrt{L} segments $\langle \eta_1, \dots, \eta_{\sqrt{L}} \rangle$, such that each segment has \sqrt{L} contiguous phases. Consider the blocks fetched by OPT in some I/O in *phase*(i) of segment η_j .
- For blocks belonging to *phase*(i), an I/O is performed by OPT* at the current time.
- For blocks belonging to *phase*(i') in η_j , $i \neq i'$, an I/O is performed by OPT* just prior to the start of *phase*(i').
- For blocks belonging to $\eta_{j'}$, $j \neq j'$, an I/O is performed by OPT* just prior to the start of $\eta_{j'}$.

FIGURE 3. Construction of OPT* from OPT

THEOREM 4.5. *The on-line ratio of FSP is $O(\sqrt{L})$.*

PROOF. The schedule OPT* has the following properties

1. There are no inter-segment blocks.
2. There are at most M inter-phase blocks within any segment, and all of them are fetched at the start of the segment.

By construction, the segments of OPT* match those of FSP. Since FSP uses THIN to schedule each segment the number of I/Os done by FSP and THIN in a segment are equal. In Lemma 4.8 we use property 2 to show that the number of I/Os performed by THIN in any segment is within a constant factor of the number of I/Os performed by OPT* in the same segment. By property 1 the number of I/Os done by OPT* is the sum of the number of I/Os done by it in each segment. Hence the number of I/Os needed by FSP is within $O(1)$ of OPT*. The theorem then follows from Lemma 4.4. \square

We next define the notion of *useful blocks*, which is a measure of the value of the blocks that OPT* prefetches at the start of the segment for some phase in that segment. This will be used in comparing the number of I/Os done by OPT* and FSP in a segment.

DEFINITION 4.6. A total of u_i *useful blocks* are said to be prefetched by OPT* for *phase*(i) in segment η_j , if after the prefetches at the start of η_j , the maximum number of blocks of *phase*(i) that are yet to be fetched from a single disk is $d_i - u_i$, where d_i is the max-depth of *phase*(i).

Note that if no useful blocks were fetched by OPT* for *phase*(i), then OPT* *must* do at least as many I/Os as the max-depth of that phase. Hence the number

of useful blocks fetched for a phase is the reduction in the number of I/Os that OPT* needs to do in that phase. In addition if u_i useful blocks are fetched for a phase, the buffer must contain at least as many blocks as the u_i thinnest stripes of that phase. This gives a handle on the buffer space occupied by u_i useful blocks.

CLAIM 4.7. *The maximum number of useful blocks prefetched in a segment equals the maximum number of stripes in that segment such that the total number of blocks in them is at most M .*

LEMMA 4.8. *The number of I/Os done by THIN in segment η_j , $0 \leq j < \sqrt{L}$, is within $O(1)$ of the number of I/Os done by OPT* for η_j .*

PROOF. Let the number of I/Os performed by OPT* at the start of segment η_j , to fetch the useful blocks, be I_{OPT^*} . Let the total number of useful blocks fetched by OPT* for phases in η_j be U_{OPT^*} . Let $H = \sum_{\eta_j} d_i$, be the sum of the max-depths of the phases of η_j . By Definition 4.6, the total number of I/Os done by OPT* to service η_j is

$$(4.1) \quad T_{\text{OPT}^*} = H - U_{\text{OPT}^*} + I_{\text{OPT}^*}$$

By Claim 4.7 U_{OPT^*} is less than the maximum number of stripes in that segment such that the total number of blocks in them is at most M . From the definition of THIN, red blocks belong to stripes which have the least width and number at least M . Hence U_{OPT^*} is bounded by the total number of stripes containing red blocks.

$$(4.2) \quad U_{\text{OPT}^*} \leq \sum_{\eta_j} h_i$$

where h_i is the maximum number of red blocks from a single disk in *phase*(i). By Definition 3.2 if R is the maximum number of red blocks from a single disk in η_j , and the benefit of THIN in the segment is $B_{\text{THIN}}(\eta_j)$,

$$(4.3) \quad \sum_{\eta_j} h_i = B_{\text{THIN}}(\eta_j) + R$$

Also, in η_j there are at least R (red) blocks required from a single disk. Hence,

$$(4.4) \quad T_{\text{OPT}^*} \geq R$$

From Lemma 3.3 and Equations 4.1-4.4, the total number of I/Os done by THIN in η_j is at most

$$T_{\text{THIN}}(\eta_j)/3 \leq T_{\text{OPT}^*} + U_{\text{OPT}^*} - B_{\text{THIN}}(\eta_j) \leq T_{\text{OPT}^*} + R \leq 2T_{\text{OPT}^*}$$

□

The on-line ratio of ASP follows from Theorem 4.5 and the fact that the number of I/Os done by ASP is no more than the number of I/Os done by FSP. We can show that the bound is tight by constructing a reference string of length ML such that the bound is achieved. The proof of the lower bound is omitted for brevity.

THEOREM 4.9. *The on-line ratio of ASP with ML -block lookahead is $\Theta(\sqrt{L})$.*

The analysis above may suggest that FSP is comparable in performance to ASP. However with $L > D^{2/3}$, it will be shown in Section 4.2 that the on-line ratio of ASP is $\Theta(D^{1/3})$, while that of FSP can be shown to be $\Omega(\sqrt{L})$, for $D^{2/3} < L \leq D$. Even in the range $L \leq D^{2/3}$ there are reference strings for which ASP performs $\Omega(\sqrt{L})$ times less I/Os than FSP. The details of these constructions are omitted in this abstract.

4.2. Analysis of ASP for $L > D^{2/3}$. In this range of L , we shall bound the on-line ratio of ASP by comparing it to the algorithm RBP presented in [KV98]. We summarize the algorithm and its analysis in the current context in Figure 4. The main difference between ASP and RBP is the scheme used by RBP to color blocks. RBP uses a fixed threshold width to decide which blocks to color red, while THIN adapts to the structure of the reference string.

Partition the I/O buffer into two parts, *red buffer* and *black buffer*, each of size $M/2$. Each part will only be used to hold blocks of that color. The blocks of all stripes in the reference string with a width smaller than $D^{1/3}$ are colored red. All other blocks are colored black. Partition the reference string into segments of maximal length such that the total number of red blocks in each segment is at most M .

On a request for block B , one of the following actions is 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: (a) If B is red, fetch up to the next $M/2$ red blocks in the segment beginning with B in order of reference, into the red buffer. (b) If B is black, fetch up to the next $M/2$ black blocks in the phase beginning with B in order of reference, into the black buffer. These blocks are fetched with maximal parallelism.
 - Service the request and evict block B from the buffer.

FIGURE 4. Algorithm RBP

THEOREM 4.10 ([KV98]). *The ratio of the number of I/Os done by RBP to that done by OPT is $\Theta(D^{1/3})$, for $L \geq D^{2/3}$.*

To show that the on-line ratio of ASP is $O(D^{1/3})$ it is sufficient to show that the number of I/Os done by ASP is within a factor of $O(1)$ of the number of I/Os done by RBP. This follows from the way RBP partitions the reference string: At most M blocks are prefetched for phases in a segment, and in any I/O done in a segment no block is prefetched for a different segment. Hence, by a proof similar to that used in Lemma 4.8 we can show that in any segment the number of I/Os done by THIN is within a constant factor of the number of I/Os done by RBP. As ASP partitions the reference string into segments to minimize the total cost of THIN, the number of I/Os done by ASP is within a constant factor of the number of I/Os done by RBP. Like in the previous range of L , the bound above is tight. The lower bound, whose proof is omitted for brevity, can be shown by constructing a reference string for which the equality holds.

THEOREM 4.11. *The on-line ratio of ASP is $\Theta(D^{1/3})$.*

We can show that there are reference strings for which ASP performs $\Omega(D^{1/3})$ times fewer I/Os than RBP. For the range $L \leq D^{2/3}$, it can be shown that RBP performs significantly more I/Os than ASP. Even if the best threshold width is chosen for a given L , we can show that there are reference strings of length ML , $L < D^{2/3}$, for which the number of I/Os done by RBP is at least $\Omega(C)$ times the number of I/Os done by OPT, where $C = \min\{L, (DL)^{1/5}\}$.

5. Conclusions

In this paper we addressed the problem of on-line scheduling of read-once reference strings in the parallel disk model using bounded ML -block lookahead. We presented a novel algorithm ASP and compared its performance with the best on-line algorithm with the same amount of lookahead, using the on-line ratio as the performance measure.

ASP partitions the reference string into segments of contiguous phases, and performs all prefetching within a segment only. Two auxiliary algorithms RBP and FSP, which also schedule I/Os on a segment-by-segment basis, were defined for bounding the performance of ASP. RBP defines segments based on a predetermined threshold width, while FSP defines segments of fixed length. In contrast, ASP adaptively chooses its segments to minimize the total cost of servicing them and is thereby able to match the best performance of both FSP and RBP. We showed that ASP has an online ratio of $\Theta(\sqrt{L})$ when $L \leq D^{2/3}$ and $\Theta(D^{1/3})$ when $L > D^{2/3}$.

References

- [AGL98] S. Albers, N. Garg, and S. Leonardi, *Minimizing Stall Time in Single and Parallel Disk Systems*, Proc. of STOC'98, 1998, pp. 454–462.
- [Bel66] L. A. Belady, *A Study of Replacement Algorithms for a Virtual Storage Computer*, IBM Systems Journal **5** (1966), no. 2, 78–101.
- [BGV95] R. D. Barve, E. F. Grove, and J. S. Vitter, *Application-Controlled Paging for a Shared Cache*, Proc. of FOCS'95, 1995, pp. 204–213.
- [BGV96] R. D. Barve, E. F. Grove, and J. S. Vitter, *Simple Randomized Mergesort on Parallel Disks*, Parallel Computing **23** (1996), no. 4, 601–631.
- [BKVV97] R. D. Barve, M. Kallahalla, P. J. Varman, and J. S. Vitter, *Competitive Parallel Disk Prefetching and Buffer Management*, Proc. of IOPADS'97, 1997, pp. 47–56.
- [CFKL95] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, *A Study of Integrated Prefetching and Caching Strategies*, Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, ACM, May 1995, pp. 188–197.
- [CLG+94] 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** (1994), no. 2, 145–185.
- [KK96] T. Kimbrel and A. R. Karlin, *Near-Optimal Parallel Prefetching and Caching*, Proc. of FOCS'96, 1996, pp. 540–549.
- [KP94] E. Koustoupias and C. H. Papadimitriou, *Beyond Competitive Analysis*, Proc. of STOC'94, 1994, pp. 394–400.
- [KV98] M. Kallahalla and P. J. Varman, *Red-Black Prefetching: An Approximation Algorithm for Parallel Disk Scheduling*, Proc. of FST&TCS'98, 1998, to appear.
- [PSV94] V. S. Pai, A. A. Schäffer, and P. J. Varman, *Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging*, Theoretical Computer Science **128** (1994), no. 1–2, 211–239.
- [ST85] D. D. Sleator and R. E. Tarjan, *Amortized Efficiency of List Update and Paging Rules*, Communications of the ACM **28** (1985), no. 2, 202–208.
- [Var98] P. J. Varman, *Randomized Parallel Prefetching and Buffer Management*, Parallel and Distributed Computing (1998), 363–372.
- [VS94] J. S. Vitter and E. A. M. Shriver, *Optimal Algorithms for Parallel Memory, I: Two-Level Memories*, Algorithmica **12** (1994), no. 2–3, 110–147.
- [VV96] P. J. Varman and R. M. Verma, *Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems*, Proc. of FST&TCS'96, vol. 16, 1996, pp. 454–462.