Optimal Read-Once Parallel Disk Scheduling*

Mahesh Kallahalla
kalla@rice.edu

Peter J. Varman
pjv@rice.edu

Deptartment of Electrical and Computer Engineering
Rice University
Houston TX 77251

Abstract

We present an optimal algorithm, L-OPT, for prefetching and I/O scheduling in parallel I/O systems using a read-once model of block reference. The algorithm uses knowledge of the next L block references, L-block lookahead, to schedule I/Os in an on-line manner. It uses a dynamic priority assignment scheme to decide when blocks should be prefetched, so as to minimize the total number of I/Os.

The parallel disk model of an I/O system is used to study the performance of L-OPT. We show that L-OPT is comparable to the *best on-line* algorithm with the same amount of lookahead; the ratio of the length of its schedule to the length of the optimal schedule is within a constant factor of the best possible. Specifically, we show that the competitive ratio of L-OPT is $\Theta(\sqrt{MD/L})$ which matches the lower bound on the competitive ratio of any prefetching algorithm with L-block lookahead. In addition we show that when the lookahead consists of the entire reference string, L-OPT performs the minimum possible number of I/Os; hence L-OPT is the *optimal off-line* algorithm. Finally, using synthetic traces we empirically study the performance characteristics of L-OPT.

1 Introduction

Prefetching is a powerful and well-studied technique to reduce the I/O latency seen by an application. It is especially useful in a multiple-disk I/O system because it provides a mechanism for exploiting I/O parallelism. By prefetching data from idle disks concurrently with data demanded immediately by the application, I/O latency of the prefetched data can be hidden. Such prefetched data is held in the I/O buffer until required by the application. Since the amount of data that can be prefetched is limited by the size of the I/O buffer, a prefetching algorithm should carefully decide what data to prefetch and retain in the buffer.

We address the problem of prefetching and I/O scheduling in a multiple-disk parallel I/O system. The prefetching and I/O scheduling algorithms are developed using the parallel disk model [13] of I/O. In this model the I/O system consists of D independently accessible disks with a shared I/O buffer of capacity M blocks. In each I/O up to D blocks, at most one from each disk, can be read in parallel. The application is characterized by a *reference string* which consists of the ordered sequence of blocks that the application needs. The central problem is to service the I/O requests in the order specified by the reference string with the minimal number of I/Os.

We concentrate on *read-once* reference strings in which each I/O reference is to a *distinct* block. Such reference strings arise naturally in streamed applications, for instance in real-time playback of multiple multimedia data streams and external merging. Additionally read-once reference strings are also a good approximation of reference strings where accesses to the same block are infrequent. Finally, read-once reference strings provide a foundation to develop and study prefetching algorithms for parallel I/O without the additional complications introduced by caching, since there is no benefit in retaining a block in the buffer once it has been referenced. Interestingly, even with this simplification, the problem of scheduling a read-once reference string in the optimal number of I/Os has been open. In [3] it was shown that making prefetching choices based solely on the position of the block in the reference string may result in an excessive number of I/Os. The algorithm L-OPT that we present uses a different adaptive priority scheme to decide which blocks to prefetch and retain in the buffer.

To motivate the problem let us consider the following example. Let the I/O 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 A (respectively B, C), and the reference string be $\langle a_1\, a_2\, a_3\, a_4\, b_1\, c_1\, a_5\, b_2\, c_2\, a_6\, b_3\, c_3\, a_7\, b_4\, c_4\, c_5\, c_6\, c_7\rangle$. 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 the I/O schedule constructed by a greedy algorithm that always fetches blocks in the order of the reference string, and maximizes the disk parallelism at each I/O step. In the first step 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. Subsequently the buffer contains 5 blocks: a_2, b_1, b_2, c_1, and c_2. Next when a_3 is requested an I/O needs to be done to fetch it. However, there is buffer space for only one 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 generated by our algorithm L-OPT. The first two I/O steps in the I/O schedule are identical to the previous case. In step 3, c_3 that occurs after b_3 is prefetched; and in step 4 c_4 is fetched

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

Disk A	a_1	a_2	a_3	a_4	a_5	a_6	a_7		
Disk B	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 A	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Disk B	b_1	b_2			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): Schedule by L-OPT

by evicting b_2 even though c_4 is referenced only after b_4. However, by doing so the overall length of the schedule is reduced to 7, better than the schedule that fetched greedily in the order of the reference string. This schedule is of minimal length for this reference string because there are 7 blocks requested from a single disk.

The above example illustrates the problem of I/O scheduling in a parallel I/O system. Even though block c_4 is only requested after b_2, b_3, and b_4, giving preference to it results in a shorter I/O schedule. As an interesting aside, at the time when a_4 is fetched and b_2 evicted, b_2 is within the next buffer load of blocks to be requested; but nevertheless it is still preferable to evict it and fetch c_4 instead. Optimizing the total number of I/Os cannot be done based solely on simple local heuristics such as "at any time keep as many disks busy as possible" or "fetch blocks only if they are within the next buffer load of blocks to be requested".

By definition a prefetching algorithm should be able to fetch blocks before they are actually required by the computation. To do so accurately the algorithm should have some knowledge of future requests. We model this lookahead information through *Global Lookahead* [3]. If the reference string is $\langle b_0, b_1, \cdots, b_N \rangle$, immediately after referencing block b_i an algorithm with global L-block lookahead knows the sequence of next L blocks to be referenced, $\langle b_{i+1}, b_{i+2}, \cdots b_{i+L} \rangle$. As the computation progresses global L-block lookahead therefore provides a fixed-size rolling window of future requests. There has been substantial interest in obtaining lookahead information for prefetching from applications using combinations of programmer hints and program analysis [9, 11].

We refer to prefetching algorithms which have access to a bounded amount of lookahead as *on-line*. In contrast an *off-line* algorithm constructs a schedule based on a priori knowledge of the entire reference string. We measure the performance of an on-line algorithm using the *competitive ratio* [7], which is the worst case ratio of the number of I/Os done by the algorithm to that done by the optimal off-line algorithm.

Recent formal work on prefetching in single-disk systems have focussed on overlapping I/O with computation [4]. A stall model of sequential I/O incorporating computation time was introduced by [4]. They considered the combined problem of prefetching and caching and showed how aggressive prefetching can reduce the total I/O time. In the same model, formulating the caching problem as an integer program, a polynomial time solution was presented by [1].

Initial formal work on parallel I/O focussed mainly on designing algorithms and data-structures for particular applications; for example external sort [2, 10, 13, 14]. The more general problem of prefetching in the parallel disk model was studied by [3]. The special case when the lookahead spans the next buffer-load, that is $L = M$, was considered; it was show that the competitive ratio of any algorithm with global M-block lookahead is $\Omega(\sqrt{D})$, and a simple algorithm matching the bound was presented. However having lookahead equal to the size of the buffer is intrinsic to the algorithm and there does not seem to be any generalization which uses extended lookahead. Using M-block lookahead, [5] presented a randomized dynamic data relocation scheme to service read-many reference strings where the same block may be referenced more than once.

Other studies have looked at the problem of prefetching and caching read-many reference strings in alternative parallel I/O models. [12] presented an optimal off-line algorithm in a distributed buffer configuration, a variant of the parallel disk model, where each disk has its private buffer. [1, 8] considered the problem in a parallel version of the stall model. [8] designed a sophisticated off-line approximation algorithm to service read-many reference strings and showed that it is near optimal for typical system parameters (memory size and computation time).

In this paper we present an on-line prefetching algorithm L-OPT using L-block lookahead, $L \geq 1$. L-OPT uses the currently available lookahead to determine blocks to fetch on the next I/O. It uses a novel priority assignment scheme to determine the most valuable blocks to fetch and keep in the buffer. As the lookahead window advances and new information about requests are made available, the priorities of blocks are dynamically updated to incorporate the latest information. We show that for any lookahead $L \geq M$, L-OPT has a competitive ratio of $\Theta(\sqrt{MD/L})$. Furthermore, by generalizing a lower-bound result from [3], we show that the competitive ratio of *any* on-line algorithm with global L-block lookahead is $\Omega(\sqrt{MD/L})$; hence L-OPT is comparable to the best on-line algorithm with L-block lookahead. An additional consequence of this result is that it quantifies the maximum potential benefit that can be obtained by having increased lookahead. For instance when $L = MD$ the competitive ratio of L-OPT is $O(1)$, and hence at most constant-factor improvements can be obtained by increasing the lookahead beyond this. We also analyze the case when $L < M$. In this case L-OPT becomes a simple greedy prefetching algorithm. We bound its competitive ratio and show that it matches the lower bound up to constants in this range as well.

Another important feature of L-OPT is its performance as an *off-line* algorithm. When the lookahead L comprises the entire reference string, L-OPT is the *optimal* off-line algorithm; that is, it produces the minimal length I/O schedule for the given reference string. This is the first provably optimal polynomial-time algorithm for scheduling read-once reference strings in the parallel disk model. The previous best algorithm had a worst-case approximation ratio of $\Theta(D^{1/3})$ [6].

The rest of the paper is organized as follows. Section 2 presents the details of algorithm L-OPT. Theorem 1 establishes the competitive ratio of L-OPT in the entire range of lookahead. A corollary of Theorem 2 show that L-OPT is the optimal off-line algorithm. Section 4 presents a simulation study of L-OPT on synthetically generated reference strings.

2 Algorithm L-OPT

In this section we present the details of algorithm L-OPT. L-OPT is a *priority-controlled* greedy algorithm that uses L-block lookahead. In every I/O it fetches one block each from as many disks as possible, while ensuring that the buffer always contains the blocks with the currently highest priorities. L-OPT assigns a priority to each block in the lookahead using a novel priority assignment scheme. The priorities are used in deciding which blocks to prefetch in an I/O.

At any instant the priority of a block is a reflection of how urgently that block must be fetched. The lower the priority of a block, the later it can be fetched. As the reference string is serviced, additional blocks are revealed and L-OPT dynamically updates the

Algorithm L-OPT

On a request for a block, b_i, algorithm L-OPT takes the following actions.

If b_i is present in the buffer then no I/O is necessary.
If b_i is not present in the buffer then
  To accommodate the blocks to be read in, evict the blocks in $B_i - B_i^+$.
  Initiate an I/O to fetch the blocks in $H_i \cap B_i^+$.
Service the request for block b_i.
Call $\text{UpdatePriority}(i + L)$.

Figure 2: Algorithm L-OPT

priority of the blocks in the lookahead. Consequently some blocks which were prefetched earlier when their relative priorities were high, could now be less important and may need to be evicted to make space for higher priority blocks.

Next we shall describe the algorithm. The following definitions will be used to determine the most appropriate blocks to fetch from disks and those that need to be evicted from the buffer to accommodate the fetched blocks. We shall use the term "current lookahead" to mean the set of all blocks in the reference string which are less than L blocks away from the next block to be referenced.

Definition 1.

- Let $\Sigma = \langle b_0, b_1, \cdots \rangle$ denote the reference string. If b_i is a block requested in the current lookahead, let $\text{disk}(b_i)$ denote the disk from which it needs to be fetched and let $\text{priority}(b_i)$ be the block's priority.
- At the instance when b_i is referenced let B_i denote the set of blocks which are in the current lookahead and present in the buffer.
- When b_i is referenced let H_i be the maximal set of (up to) D blocks, such that if $b \in H_i$ then priority of b is the largest among all blocks from $disk(b)$ in the lookahead but not present in the buffer.
- Let B_i^+ be the maximal set of (up to) M blocks with the highest priorities in $H_i \cup B_i$; in the case of ties the block occurring earlier in Σ is preferred.

Algorithm L-OPT is detailed in Figure 2. The priority function that we define will ensure that for any two blocks from the same disk, the block referenced earlier has the higher priority. In one I/O at most one block is fetched from each disk. As we would like to fetch from each disk the highest priority block, H_i is the set of blocks that we would like to fetch in the next I/O. However, because of the current contents of the buffer we may not be able to fetch and accommodate all the blocks in H_i. At this time we use the priorities of blocks to decide which blocks to be fetch. Among the blocks present in the buffer and those that can be fetched in the next I/O we would like the buffer to contain the blocks with highest priority: B_i^+ is the desired state of the buffer following the I/O. Algorithm L-OPT performs an I/O and evicts appropriate blocks to guarantee that this is the case. Additionally, after servicing each reference, a new block is revealed to the algorithm by the lookahead; L-OPT updates priorities to correspond to the modified lookahead.

Definition 2.

- Let $\text{backlog}(b_i) = |\{b_j : j \geq i \text{ and } \text{priority}(b_j) \geq \text{priority}(b_i)\}|$ be the number of blocks in the current lookahead with priority at least that of b_i but referenced no later than b_i.
- Let $\text{smallest}(p)$ be the earliest referenced block in the current lookahead with priority p. $\text{smallest}(p) = \min_i\{b_i : \text{priority}(b_i) = p\}$.

The operation UpdatePriority, given in Figure 3, is used to incrementally determine the priorities of all blocks in the lookahead. The central idea behind the priority assignment scheme is to set the priority of a block as low as possible. However since the priorities of blocks directly influence the order in which blocks are fetched certain conditions need to be satisfied. It can be seen that fetching blocks from a disk out of order of the reference string cannot decrease the number of I/Os needed. Hence the priority assignment scheme should guarantee that blocks from the same disk are assigned priorities in order. Secondly if there are no additional blocks added to the lookahead, blocks are fetched according to the current priorities. Hence it is not permissible for any block to be assigned a priority such that more than $M - 1$ blocks referenced after it have higher priority. Our priority assignment scheme maintains the following two invariants:

1. No two blocks on the same disk have the same priority, and the priority of a block is always higher than other blocks from the same disk referenced after it
2. For any block b in the lookahead with priority p there are less than M blocks occuring after b in the reference string with priority at least p

After a block b_i is serviced the lookahead window shifts by one block: an additional block b_{i+L} becomes visible. The *UpdatePriority* operation assigns the lowest possible priority (of 1) to the new block and updates the priorities of the remaining blocks. The underlying intuition is that if there are no further blocks in the reference string then the new block can be fetched in the last I/O and can therefore be given the lowest priority. However this may result in two blocks from the same disk having priority 1. The first call to *IncreasePriority* corrects this problem by increasing the priorities of blocks on this disk till they are all unique. Increasing the priorities of the blocks may trigger a *buffer capacity violation*. That is, it may result in some block violating invariant 2. The inner two while loops of *UpdatePriority* check the priority assignments for buffer capacity violations and correct them by correspondingly increasing the priority of some blocks. The data structure $\mathcal{P}_{i+L}$ maintains the priorities of blocks in the lookahead just before block b_i is referenced.

At the start of computation the first L blocks, $b_0, \cdots, b_{L-1}$, are known to L-OPT. The priorities of these blocks are initialized by calling $\text{UpdatePriority}(i)$ successively with $i = 0, \cdots, L - 1$. Initially $\mathcal{P}_0$ is taken to be empty.

Let us illustrate the working of the UpdatePriority operation with an example. Consider a system of $D = 4$ disks with buffer

UpdatePriority and IncreasePriority Operations

The $\mathbf{UpdatePriority(j)}$ operation works to update the priorities of blocks in $\mathcal{P}_j$ as follows:

if there is a block a on $\mathrm{disk}(b_j)$ such that $\mathrm{priority}(a) = 1$ then
    call $\mathrm{IncreasePriority}\,(1, a)$
assign $\mathrm{priority}\,(b_j) \leftarrow 1$
assign $p \leftarrow 1$
while there is a block $a \in \mathcal{P}_j$ such that $\mathrm{priority}\,(a) = p$ do
    while $\mathrm{backlog}(\mathrm{smallest}\,(p)) > M$ do /* buffer capacity violation */
        call $\mathrm{IncreasePriority}\,(p, \mathrm{smallest}\,(p))$
    increment p
update $\mathcal{P}_{j+1} \leftarrow \mathcal{P}_j \cup \{b_j\} - \{b_{j-L}\}$

The $\mathbf{IncreasePriority(p, b)}$ operation works as follows:

If there is a block $a \in \mathcal{P}$ such that $\mathrm{priority}(a) = p + 1$ and $\mathrm{disk}(b) = \mathrm{disk}(a)$ then
    call $\mathrm{IncreasePriority}\,(p + 1, a)$
assign $\mathrm{priority}\,(b) \leftarrow p + 1$

Figure 3: Operations to update priorities

Block $\longrightarrow$	a_1	b_1	c_1	c_2	d_1	a_2	b_2	c_3	c_4
Initial Priorities	2	2	3	2	1	1	1	1	
After $\mathrm{IncreasePriority}\,(1, c_3)$	2	2	4	3	1	1	1	2	1
After $\mathrm{IncreasePriority}\,(1, d_1)$	2	2	4	3	2	1	1	2	1
After $\mathrm{IncreasePriority}\,(2, a_1)$	3	2	4	3	2	1	1	2	1
After $\mathrm{IncreasePriority}\,(2, b_1)$	3	3	4	3	2	1	1	2	1

Figure 4: Operations performed during $\mathrm{UpdatePriority}\,(8)$

of size $M = 4$ and lookahead $L = 9$. Let the reference string be $\Sigma = \langle a_1\, b_1\, c_1\, c_2\, d_1\, a_2\, b_2\, c_3\, c_4 \rangle$, where a block a_i (b_i, c_i, and d_i) is from disk A (B, C, and D).

Table 4 illustrates the various operations performed during $\mathrm{UpdatePriority}\,(8)$. $\mathrm{UpdatePriority}\,(8)$ is called to initialize the priority of c_4 and update the priorities of all blocks up to c_3. In the figure, each row gives the priorities of blocks in Σ after the operation given in the first column is performed. Blocks in the table are arranged in the order of appearance in Σ. Let us assume for now that the initially priorities of blocks are as in the first row.

Because there is a block c_3 from disk C with priority 1, $\mathrm{priority}\,(c_4)$ is assigned 1 after an $\mathrm{IncreasePriority}\,(1, c_3)$ operation. The resultant priority assignment is presented in the second row. The blocks whose priorities changed are shaded. Note that only the priorities of blocks on disk C are incremented. Next we perform buffer violation checks at each priority. Starting with blocks with priority 1, we see that the $\mathrm{smallest}(1) = d_1$ and $\mathrm{backlog}(d_1) = 5$ which is greater than $M = 4$. Hence $\mathrm{IncreasePriority}\,(1, d_1)$ is called. As d_1 is the only block from that disk in the reference string this only increases its priority by 1. Following this $\mathrm{smallest}(1) = a_2$ and $\mathrm{backlog}(a_2) = 4$ which is not a buffer capacity violation. Hence we next consider blocks with priority 2. Now $\mathrm{smallest}(2) = a_1$ and $\mathrm{backlog}\,(a_1) = 6$. Hence an $\mathrm{IncreasePriority}\,(2, a_1)$ is performed following which $\mathrm{smallest}\,(2) = b_1$ and $\mathrm{backlog}(b_1) = 5$. Thus an $\mathrm{IncreasePriority}\,(2, b_1)$ operation is performed, after which there are no more buffer capacity violations.

A complete picture of the changes in block priorities, as additional blocks in the lookahead are included, is presented in Table 5. Like in the previous illustration, the entries correspond to priorities of blocks. Blocks whose priorities changed by the operation in the first column are shaded.

From the above examples we can observe that the priority of any block increases by at most 1 during an UpdatePriority operation. Moreover following an UpdatePriority operation the backlog of any block is at most M. We shall now formalize these observations and show that L-OPT correctly generates a schedule to service a given reference string.

First note that after a block's priority is initialized to 1, its priority is changed only during IncreasePriority operations. It can be seen that IncreasePriority operation maintains the following invariant which was introduced informally before.

Claim 1 *If b_i and b_j are two blocks from the same disk and present in the lookahead then $\mathrm{priority}\,(b_i) > \mathrm{priority}\,(b_j)$ if and only if $i < j$; hence, all blocks on a disk are assigned unique priorities.*

The following Lemma shows that L-OPT does construct a valid schedule. The length of the schedule constructed by L-OPT will be analyzed in Section 3.

Block $\longrightarrow$	a_1	b_1	c_1	c_2	d_1	a_2	b_2	c_3	c_4
After $\mathrm{UpdatePriority}\,(0)$	1								
After $\mathrm{UpdatePriority}\,(1)$	1	1							
After $\mathrm{UpdatePriority}\,(2)$	1	1	1						
After $\mathrm{UpdatePriority}\,(3)$	1	1	2	1					
After $\mathrm{UpdatePriority}\,(4)$	2	1	2	1	1				
After $\mathrm{UpdatePriority}\,(5)$	2	2	2	1	1	1			
After $\mathrm{UpdatePriority}\,(6)$	2	2	2	1	1	1	1		
After $\mathrm{UpdatePriority}\,(7)$	2	2	3	2	1	1	1	1	
After $\mathrm{UpdatePriority}\,(8)$	3	3	4	3	2	1	1	2	1

Figure 5: Evolution of Priorities

Lemma 1 *Given any reference string L-OPT generates an I/O schedule to service it.*

Proof : We shall show that whenever a requested block is not present in the buffer, it is fetched in the next I/O. This guarantees that L-OPT generates an I/O schedule which services the reference string.

Let the requested block be b_i. As b_i is the earliest required block from $\mathrm{disk}(b_i)$, by Claim 1 it has the highest priority among all blocks from that disk; hence $b_i \in H_i$. By Lemma 2 $\mathrm{backlog}(b_i) \leq M$. Because b_i is the first block in the lookahead there are no more than $M - 1$ other blocks with higher priority occurring after it in the reference string. Hence $b_i \in B_i^+$. These conditions guarantee that b_i is fetched in the following I/O. □

The following lemma is proved inductively using Lemma 3 and observing that it holds after $k = 0$.

Lemma 2 *After* $\mathrm{UpdatePriority}\,(k)$ *the following two properties hold.*

- *The priority of any block has increased by at most one during* $\mathrm{UpdatePriority}\,(k)$.
- *For any block* $b \in \mathcal{P}_{k+1}$, $\mathrm{backlog}\,(b) \leq M$.

Lemma 3 *If after* $\mathrm{UpdatePriority}\,(k)$ *the properties in Lemma 2 hold then they hold after the next update,* $\mathrm{UpdatePriority}\,(k+1)$.

Proof : Two cases are possible depending on whether $k < L$ or $k \geq L$. We shall present an argument for the case $k \geq L$; a similar proof holds for the other range. We shall use $\mathrm{backlog}^*(b)$, $\mathrm{smallest}^*(p)$ and $\mathrm{priority}^*(b)$ to denote terms as defined in Definition 2 but considering priorities immediately after the previous update, $\mathrm{UpdatePriority}\,(k)$.

At most one IncreasePriority operation is performed prior to initializing the priority of b_{k+1} to 1. Hence at this time $\mathrm{backlog}\,(\mathrm{smallest}\,(1)) \leq \mathrm{backlog}^*(\mathrm{smallest}^*(1)) + 1$. Therefore, if after this $\mathrm{backlog}\,(\mathrm{smallest}\,(1)) \geq M$ the following $\mathrm{IncreasePriority}\,(1, \mathrm{smallest}\,(1))$ guarantees that the lemma is satisfied for all blocks in $\mathcal{P}_{k+1}$ which had priority 1. In addition the priority of any block has increased by at most 1.

Inductively let us assume that after the set of IncreasePriority operations, in the inner while loop, at priority p the lemma is satisfied by all blocks of priority p or lower in $\mathcal{P}_{k+1}$. As the priority of any block has increased by at most 1, only the following cases are possible:

Case a. There is only one block b such that $\mathrm{priority}\,(b) = p + 1$. By the induction hypothesis $\mathrm{backlog}^*(b) \leq M$ and the priority of any block has increased by at most 1. Hence $\mathrm{backlog}\,(b) \leq \mathrm{backlog}^*(b)$

Case b. $\mathrm{priority}^*(\mathrm{smallest}\,(p+1)) = p$.. The priorities of blocks have increased by at most 1 and $\mathrm{backlog}^*(\mathrm{smallest}^*(p)) \leq M$; hence $\mathrm{backlog}\,(\mathrm{smallest}\,(p+1)) \leq \mathrm{backlog}^*(\mathrm{smallest}^*(p))$.

Case c. $\mathrm{priority}^*(\mathrm{smallest}\,(p+1)) = p+1$. Then after performing $\mathrm{IncreasePriority}\,(p+1, \mathrm{smallest}\,(p+1))$, $\mathrm{smallest}\,(p+1)$ is at least 1 higher and hence $\mathrm{backlog}\,(\mathrm{smallest}\,(p+1))$ is at least 1 lower. The same three cases as above are now applicable again.

Hence in all the cases after the set of IncreasePriority operations performed within the inner while loop of UpdatePriority, at priority $p+1$, the induction hypothesis holds. □

A simple forecasting data structure as in [2] can be used to maintain the list of blocks with highest priority on each disk. On a hit in the buffer algorithm L-OPT does not need to do any other book-keeping apart from updating priorities. When requested block is not present in the buffer the algorithm needs to find the set of blocks to fetch and the corresponding set of blocks to evict. Since in an I/O at most D blocks could be fetched, this takes takes $O(M + D)$ time. A straightforward implementation of the lookahead window as a linear array could incur $O(L^2 D \log D)$ time for each UpdatePriority operation. This can be improved to $O(LD)$ by maintaining some additional state besides the reference string: from any disk the order in which blocks are accessed, the number of blocks with priority p or higher, and for every priority p the earliest referenced block with priority p. This is allowed by the property, from Lemma 2, that the priority of a block does not increase by more than 1 in one UpdatePriority operation.

3 Analysis

In this section we shall prove bounds on the competitive ratio of L-OPT. Theorem 1 presents the main result. In the process we shall also show, in Theorem 2, that when the length of the reference string matches the lookahead, L-OPT performs the minimal number of I/Os to service the reference string. Hence L-OPT is an optimal off-line algorithm. Theorem 1 is proved for the range $L > M$ in Lemma 7 and for $L \leq M$ in Lemma 8.

Theorem 1 *The competitive ratio of L-OPT is*

$$\begin{aligned} \Theta(\sqrt{MD/L}) &\quad \textit{for} \quad L > M \\ \Theta(\sqrt{D}) &\quad \textit{for} \quad M \geq L \geq M/\sqrt{D} \\ \Theta(M/L) &\quad \textit{for} \quad M/\sqrt{D} \geq L > M/D \\ \Theta(D) &\quad \textit{for} \quad M/D \geq L \end{aligned}$$

The following lemma shows that we do not need to consider the class of all possible optimal algorithms. Instead, we can, without loss in generality, assume that the optimal algorithm (i) does not evict a block that it fetched till it is referenced and (ii) fetches blocks from any disk in order. We shall assume this for all algorithms considered henceforth. The proof is omitted for brevity.

Lemma 4 *Any I/O schedule can be transformed into another of same or smaller length such that the following properties hold* (i) *no block is evicted from the buffer till it is referenced* (ii) *if b_i and b_j, $i < j$, are two blocks fetched from disk d in the kth and lth I/O, respectively, then $k < l$.*

During each UpdatePriority operation the priority of blocks change to incorporate the additional block. The following lemma presents conditions on the possible values for the priority of a block. It will be useful in showing the relation between the priority of a block and the I/O in which it is fetched by OPT.

Lemma 5 *Following an* $\mathrm{UpdatePriority}(i)$ *operation,* $0 \leq i < L$, *let* $\mathrm{priority}(b) = p$ *for some block b. Then either* (i) *there is a block a from* $\mathrm{disk}(b)$ *such that* $\mathrm{priority}(a) = p - 1$ *or* (ii) *number of blocks referenced after b with priority at least $p - 1$ is more than M or* (iii) $p = 1$.

Proof : Proof by induction. The lemma holds vacuously for $i = 0$. For the induction hypothesis let us assume the lemma holds after $\mathrm{UpdatePriority}(k)$.

During an UpdatePriority operation the priorities of blocks are increased only through the IncreasePriority operation. So we shall show that the lemma holds after $\mathrm{UpdatePriority}(k+1)$ by showing that the lemma holds after each intermediate IncreasePriority operation. Note that during an UpdatePriority operation the priority of a block is never reduced.

Consider the the first call to $\mathrm{IncreasePriority}(1, \mathrm{disk}(b_{k+1}))$ prior to assigning a priority of 1 to b_{k+1}. It is clear that for all blocks whose priorities have been increased condition (i) holds, while for the new block condition (iii) is true. For the rest of the blocks the lemma holds through the induction hypothesis.

The reason for performing an IncreasePriority operation later, at priority p, is that there was a block b such that $\mathrm{backlog}(b) > M$. Hence after that IncreasePriority operation condition (ii) holds for block b, while for the rest of the blocks whose priorities were increased condition (i) holds. The lemma is satisfied for the rest of the blocks through the inductive hypothesis. □

We next present the central theorem of our analysis. It shows that L-OPT services reference strings of length L in the optimal number of I/Os.

Theorem 2 *To service a reference string of length $|\Sigma| < L$, the number of I/Os done by any algorithm is lower bounded by the number of I/Os done by L-OPT; that is, L-OPT services any reference string up to length L in the optimal number of I/Os.*

Proof : When the length of the reference string Σ is less than L, L-OPT (a) assigns a priority to all blocks before performing any I/Os and (b) does not change priorities while generating the I/O schedule. Moreover, in any I/O L-OPT fetches blocks with the highest priority, if necessary, by evicting blocks of lower priority. Hence, if p_i (p_{i+1}) is the largest priority of any block which is yet to be fetched after the ith ($i+1$th) I/O, then $p_{i+1} \leq p_i$. Therefore, the largest priority of any block in Σ is an upper bound of the number of I/Os done by L-OPT to service Σ.

Let OPT be the optimal I/O schedule to service Σ. Let the length of the optimal schedule be T_{OPT}. Let block $b_i \in \Sigma$ be fetched in the I_ith I/O in the optimal schedule. We shall show that the priority of b_i is at most $T_{\mathrm{OPT}} - I_i + 1$; the bound then follows by noting that the priority of any block is hence at most T_{OPT}.

The inductive hypothesis used to prove the lemma is as follows. OPT can be transformed into another schedule OPT_k, of the same length, satisfying the properties of Lemma 4 such that the following properties hold: for all b_i, $i < k$, $I_i^k = I_i$ and for all $i \geq k$, $T_{\mathrm{OPT}} - I_i^k + 1 = \mathrm{priority}(b_i)$, where I_i^k denotes the I/O in which b_i is fetched in OPT_k.

Consider the base case when $k = |\Sigma| - 1$. By definition $\mathrm{priority}(b_k) = 1$. OPT_k can be got from OPT be simply delaying the fetch of b_k till the last I/O. As blocks from any disk are fetched in order, there can be no block fetched from $\mathrm{disk}(b_k)$ after b_k in OPT. Moreover, because there is some other block referenced before b_k but fetched in step T_{OPT}, b_k cannot be referenced till after the last I/O. OPT_k is still of length T_{OPT}. Moreover, as the I/O for b_k is only delayed, the resultant schedule satisfies the properties of Lemma 4.

We shall now present an inductive transformation from OPT_k to OPT_{k-1} such that the hypothesis holds. Three cases are possible depending on the relation between $\mathrm{priority}(k-1)$ and $T_{\mathrm{OPT}} - I_{k-1}^k + 1$.

Case 1. $\mathrm{priority}(b_{k-1}) > T_{\mathrm{OPT}} - I_{k-1}^k + 1$. We shall show that this is not possible. By Lemma 5 three cases are possible.

Case 1a. There is a block b_l from $\mathrm{disk}(b_{k-1})$ such that $l > k-1$ and $\mathrm{priority}(b_l) = \mathrm{priority}(b_{k-1}) - 1$. As $l \geq k$ by the induction hypothesis, $\mathrm{priority}(b_l) = T_{\mathrm{OPT}} - I_l^k + 1$. By Property (b) blocks from any disk are fetched in order. Hence

$$\begin{aligned} \mathrm{priority}(b_{k-1}) - 1 &= T_{\mathrm{OPT}} - I_l^k + 1 \\ &\leq T_{\mathrm{OPT}} - I_{k-1}^k \end{aligned}$$

This contradicts the assumption that $\mathrm{priority}(b_{k-1}) > T_{\mathrm{OPT}} - I_{k-1}^k + 1$.

Case 1b. If $B_{k-1} = \{b_i : \mathrm{priority}(b_i) \geq \mathrm{priority}(b_{k-1}) - 1 \text{ and } i > k-1\}$, then $|B_{k-1}| > M$. Consider a block $b_n \in B_{k-1}$. As $n > k-1$, by the induction hypothesis, $\mathrm{priority}(b_n) = T_{\mathrm{OPT}} - I_n^k + 1$. As $\mathrm{priority}(b_n) \geq \mathrm{priority}(b_{k-1}) - 1$, using the assumption $\mathrm{priority}(b_{k-1}) > T_{\mathrm{OPT}} - I_{k-1}^k + 1$, $I_n^k \leq I_{k-1}^k$. Hence all the blocks of B_{k-1} are fetched either earlier than or together with b_{k-1} by OPT_k. This is not possible as $|B_{k-1}| > M$, while the size of the buffer is M.

Case 1c. $\mathrm{priority}(b_{k-1}) = 1$. This contradicts the assumption that $\mathrm{priority}(b_{k-1}) > T_{\mathrm{OPT}} - I_{k-1}^k + 1$, as $I_{k-1}^k \leq T_{\mathrm{OPT}}$.

Case 2. $\text{priority}(b_{k-1}) = T_{\text{OPT}} - I^k_{k-1} + 1$. Then take OPT_{k-1} to be the same as OPT_k.

Case 3. $\text{priority}(b_{k-1}) < T_{\text{OPT}} - I^k_{k-1} + 1$. Correspondingly, let $I^* = T_{\text{OPT}} - \text{priority}(b_{k-1}) + 1$. We shall show that we can delay the I/O for b_{k-1} till I^* to get OPT_{k-1}. Because OPT_{k-1} should be an I/O schedule and satisfy the properties of Lemma 4, this requires us to show that if b_{k-1} is fetched in I/O I^* then (i) the number of blocks referenced after b_{k-1} but fetched before or in I/O I^* is at most M (ii) there are no blocks from $\text{disk}(b_{k-1})$ referenced after b_{k-1} fetched before or in I/O I^*.

By construction, $\text{backlog}(b_{k-1}) \leq M$. Hence $|\{b_i : i \geq k-1 \text{ and } \text{priority}(b_i) \geq \text{priority}(b_{k-1})\}| \leq M$ By the inductive hypothesis, for all such b_i, $\text{priority}(b_i) = T_{\text{OPT}} - I^k_i + 1$. Hence $|\{b_i : i \geq k-1 \text{ and } T_{\text{OPT}} - I^k_i + 1 \geq T_{\text{OPT}} - I^* + 1\}| \leq M$. This proves that condition (i) is satisfied.

Condition (ii) is trivially satisfied if there are no blocks referenced after b_{k-1} from $\text{disk}(b_{k-1})$. Otherwise, let the next block requested from that disk be b_l. We need to show that $I^k_l > I^* = T_{\text{OPT}} - \text{priority}(b_{k-1}) + 1$. We know that $\text{priority}(b_{k-1}) \geq \text{priority}(b_l)$. By the induction hypothesis, $\text{priority}(b_l) = T_{\text{OPT}} - I^k_l + 1$. Hence $\text{priority}(b_{k-1}) \geq T_{\text{OPT}} - I^k_{k-1} + 1$.

This allows us to construct OPT_{k-1} as follows. Take schedule OPT_{k-1} to be exactly the same as OPT_k, except for block b_{k-1} which is such that $I^{k-1}_{k-1} = T_{\text{OPT}} + 1 - \text{priority}(b_{k-1})$. □

As an immediate consequence of the above theorem, L-OPT is the optimal off-line algorithm

Corollary 1 *L-OPT is an optimal off-line I/O scheduling algorithm for read-once reference strings.*

By the preceding analysis L-OPT services a reference string of length L in the optimal number of I/Os. The following lemma bounds the number of I/Os needed by L-OPT to service a reference string of arbitrary length.

Lemma 6 *The number of I/Os done by L-OPT to service a reference string $\Sigma = \langle b_0, b_1, \cdots, b_n \rangle$ is less than $2\sum_{i \geq 0} T_i$, where T_i is the number of I/Os done by L-OPT to service the reference string $\langle b_{iL}, b_{iL+1}, \cdots, b_{(i+1)L-1} \rangle$*

Proof : Proof by induction on the length of the reference string. For ease of presentation we assume that N is a multiple of L. Let $\Sigma_k = \langle b_0, \cdots, b_{(k+1)L-1} \rangle$. By definition, the number of I/Os required by L-OPT to service the reference string Σ_0 is equal to T_0. For the induction hypothesis let us assume that the number of I/Os required by L-OPT to service Σ_k be at most $I_k \leq 2\sum_{i=0}^{k} T_i$.

The reference string Σ_{k+1} is the string Σ_k followed by the string $\sigma = \langle b_{(k+1)L}, \cdots, b_{(k+2)L-1} \rangle$. The schedule generated by L-OPT to service Σ_{k+1} can be split into two parts: the first part P_1 starts at the first I/O and ends when all blocks from Σ_k are fetched and the second part P_2 is the rest of the I/O schedule. The length of the schedule for Σ_k is $S_1 + S_2$, where S_1 is the number of I/Os in P_1 and S_2 is the number of I/Os in P_2.

Because the blocks in σ form the last L blocks in the reference string, the number of I/Os in P_2 is upper bounded by $S_2 \leq T_{k+1}$.

Consider the priorities of blocks just prior to the last I/O of P_1. Let p_l be the priority of a block of Σ_k fetched in the last I/O of P_1. Let p_f be the largest priority of any block in σ. Hence the difference between the number of I/Os in P_1 and I_k is at most $p_f - p_l$.

Any block in σ is referenced after any block in Σ_k; hence there can be at most M blocks from σ fetched in I/Os of P_1. Hence if there is a block of σ on disk d with priority p fetched in P_1, there has to be block on disk d with priority $p-1$ fetched in P_1, unless $p = p_l$. Hence of the blocks in σ, if the maximum number of blocks from any one disk is H, $H \geq p_f - p_l$. Trivially, $T_{k+1} \geq H$; therefore $p_f - pl \leq T_{k+1}$. Hence $I_{k+1} \leq I_k + 2T_{k+1}$ □

Lemma 7 *The ratio of $\sum_{i \geq 0} T_i$ and the number of I/Os done by OPT to service a reference string is $O(\sqrt{MD/L})$, for $L > M$.*

Proof : Consider the schedule generated by the optimal algorithm OPT to service Σ. Let us partition the schedule into sub schedules, each consisting of a sequence of contiguous I/Os of the original schedule. Let the ith sub-schedule S_i start with the I/O following the last I/O of sub-schedule S_{i-1} and end with the last I/O in which a block from the sequence $\langle b_{iL}, \cdots, b_{(i+1)L-1} \rangle$ is fetched; let S_0 start with the first I/O. In the case when $L > M$ it can be noted that each S_i consists of at least one I/O.

Let $P_{i,j}$ denote the blocks from the set $\{b_{jL}, \cdots, b_{(j+1)L}\}$ prefetched by OPT in the sub-schedule S_i. Because the size of the buffer is M, $|P_{i,j}| \leq M$. Let the maximum number of blocks in $P_{i,j}$ which are from the same disk be $\alpha_{i,j}$. Let $\beta_i = \sum_j \alpha_{i,j}$. Let the number of I/Os done by OPT in the sub-schedule S_i be I_i. It immediately follows that $\alpha_{i,j} \leq I_i$

The blocks in $P_{i,j}$ are present in the buffer while at least $(j-i-1)L$ other requests between $b_{(i+1)L}$ and b_{jL-1} are serviced. Because the buffer is of size M, for each set of M such intermediate blocks OPT must do at least $|P_{i,j}|/D$ I/Os; these I/Os are attributable to $P_{i,j}$. Hence the number of I/Os done by OPT is at least $T_{\text{OPT}} \geq \sum_i \sum_j \lfloor (j-i-1)L/M \rfloor \times \lfloor |P_{i,j}|/D \rfloor$. Using the bounds $\beta_i = \sum_j \alpha_{i,j}$, $\alpha_{i,j} \leq I_i$ and $|P_{i,j}| \geq \alpha_{i,j}$

$$T_{\text{OPT}} \geq \sum_i \beta_i^2 L/MDI_i - O((L + \beta_i)/D)$$

The above bound, together with the simple bounds $T_{\text{OPT}} \geq \sum_i I_i$, $T_{\text{OPT}} \geq \sum_i L/D$ and $T_{\text{OPT}} \geq \sum_i \beta_i/D$, implies that

$$T_{\text{OPT}} = O(\sum_i \beta_i \sqrt{L/MD})$$

From Lemma 2, T_j is the minimal number of I/Os required to service the reference string $\langle b_{jL}, \cdots, b_{(j+1)L-1} \rangle$. Hence $T_j \leq I_j + \sum_i \alpha_{i,j}$. This implies that

$$\sum_j T_j \leq T_{\text{OPT}} + \sum_i \beta_i$$

□

The following theorem presents a lower bound on the competitive ratio of on-line algorithms with L-block lookahead: it quantifies the maximum potential benefit that an algorithm can derive with L-block lookahead. The theorem is proved by constructing a reference string which tricks a given on-line algorithm into performing substantially large number of I/Os when compared to the optimal. The construction proceeds similar to the proof of Theorem 1 in [3]; the details are omitted from this paper.

Theorem 3 *The competitive ratio of any on-line algorithm with L-block lookahead is $\Omega(\sqrt{MD/L})$ for $L > M$.*

In the lower range of lookahead, $L \leq M$, L-OPT behaves like a greedy prefetching algorithm that prefetches everything in the lookahead. As $L < M$ it can do so without ever violating the buffer condition.

Lemma 8 *When $L < M$, the competitive ratio of L-OPT is*

$$\begin{array}{lll} \Theta(\sqrt{D}) & \textit{for} & M \geq L \geq M/\sqrt{D} \\ \Theta(M/L) & \textit{for} & M/\sqrt{D} \geq L > M/D \\ \Theta(D) & \textit{for} & M/D \geq L \end{array}$$

Proof : When $L < M$ it may be noted that L-OPT behaves like a natural greedy algorithm: on every I/O fetch from each disk the next block not in the buffer. For ease of presentation we assume that M is a multiple of L.

For this algorithm it can be seen that if a requested block is not present in the buffer it is fetched in the very next I/O. Hence for a reference string of length N, the maximum number of I/Os done by L-OPT is N. On the other hand the minimum number of I/Os required to service a reference string of length N is N/D. This gives one bound on the competitive ratio of L-OPT: $O(D)$.

Stronger bounds can be shown when more lookahead is available. Let the reference string be $\Sigma = \langle b_0, \cdots \rangle$. By Lemma 9, if the minimal number of I/Os needed to service the reference string $\Sigma_i = \langle b_{iL}, \cdots, b_{(i+1)L-1} \rangle$ is T_i, then the number of I/Os done by L-OPT to service Σ is less than $\sum_i T_i$. Let the number of I/Os done by OPT to service the reference string $\Delta_j = \langle b_{jM}, \cdots, b_{(j+1)M-1} \rangle$ be I_j.

Consider the optimal schedule OPT. As in Lemma 7, let us partition the schedule into sub schedules. Let the ith sub-schedule, S_i, start with the I/O following the last I/O of sub-schedule S_{i-1} and end with the last I/O in which a block from Σ_i is fetched; let S_0 start with the first I/O. Let us also consider an alternative partitioning of the schedules into sub-schedules, R_j which starts with the I/O following the last I/O of sub-schedule R_{j-1} and end with the last I/O in which a block from Δ_j is fetched; let R_0 start with the first I/O.

Consider Σ_i which is a subsequence of Δ_j. Let P_i denote the blocks from Σ_j prefetched by OPT during I/Os of schedule R_j but not S_i. Let $Q_{i,j}$ denote the blocks from Δ_j prefetched by OPT in the sub-schedule R_i. Let the maximum number of blocks in P_i and $Q_{i,j}$ which are from the same disk be α_i and $\beta_{i,j}$ respectively. As L-OPT is the optimal algorithm to service a sequence of L blocks when its lookahead is L,

$$\sum_i T_i \leq \sum_j I_j + \sum_i \alpha_i$$

$$\sum_j I_j \leq T_{\mathrm{OPT}} + \sum_i \sum_j \beta_{i,j}$$

By an analysis similar to that of Lemma 7 we can show that

$$T_{\mathrm{OPT}} \geq \sum_i \sum_j \beta_{i,j} / \sqrt{D}$$

During I/Os of sub-schedule R_j OPT fetches $\sum_{i:\Sigma_i \in \Delta_j} \alpha_i$ blocks. However there are at most M/L schedules, Σ_i, in Δ_j. Hence, by the pigeon-hole principle, there is some P_k such that $\alpha_k \geq L/M \sum_{i:\Sigma_i \in \Delta_j} \alpha_i$. These blocks need to be fetched from the same disk and hence the number of I/Os in R_j is at least α_k. Hence $T_{\mathrm{OPT}} \geq M/L \times \sum_i \alpha_i$ □

Lemma 9 *When $L < M$, the number of I/Os done by any on-line algorithm with L-block lookahead is lower bounded by the number of I/Os done by L-OPT to service the same reference string.*

Without loss in generality we assume that the online schedule satisfies the properties of Lemma 4. Then lemma can then be shown by proving that in the optimal schedule no block can be fetched later than the corresponding I/O in the schedule generated by L-OPT. The details of the proof are omitted from this paper.

4 Simulation

In this section we present an empirical evaluation of algorithm L-OPT. We studied the performance of L-OPT using synthetically generated data based on a bursty access pattern. In order to help benchmark prefetching algorithms, we present a method to determine the minimal number of I/Os required by any on-line algorithm with L-block lookahead to service a given reference string. We then compare the performance of this lower bound to the number of I/Os made by L-OPT and NOM, described in [3]. We use the optimal off-line schedule length as the baseline.

Given an input reference string it is useful to determine a lower bound on the minimal number of I/Os required by an on-line algorithm with L-block lookahead to service it. To do so we need a characterization of schedules which could have been generated by prefetched algorithms using L-block lookahead. Intuitively, in a schedule which has been generated by an on-line algorithm with L-block lookahead in any I/O the blocks fetched are within $L+1$ blocks of the last reference serviced.

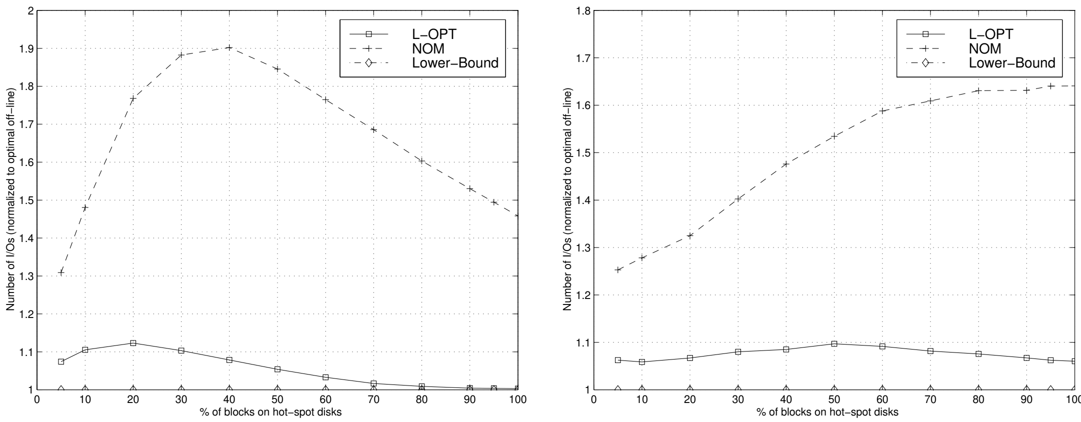
Definition 3. A schedule to service a reference string is said to have the L-property if among the blocks fetched in any I/O the block occurring farthest is within $L+1$ blocks of the last request that was serviced.

Based on the above definition, one way to find a lower bound on the length of the online schedule is to find a minimal length schedule which services the given reference string and which has the L-property. A modification of L-OPT allows us to generate such a minimal length schedule. To do so we modify the UpdatePriority routine. The inner while loop of UpdatePriority is changed to call $\mathrm{IncreasePriority}(p, \mathrm{smallest}(p))$ if for any two blocks b_i and b_j with priority p, $j - i + 1 > L$ in addition to calls to handle buffer capacity violations.

The proof that this generates a minimal length schedule having the L-property is similar to the proof of Lemma 2 and is omitted here. Given a reference string, this method is used to derive a lower bound on the number of I/Os needed by any on-line algorithm with L-block lookahead to service it.

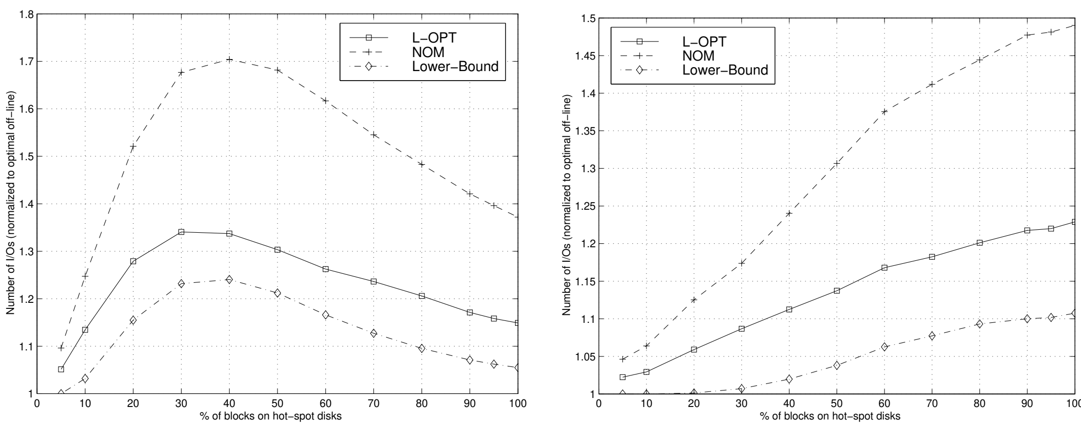
In this simulation study we model the reference string as a bursty sequence as next described. Blocks are accessed from any disk with uniform probability. Occasionally there is a burst of I/O requests to blocks from a small set of disks. These disks are referred to as *hot-spot* disks. The input parameters to the data model are: (a) the percent f, of blocks which occur on hot spot disks and (b) the number of hot spot disks k. Specifically, the probability that a block is on a hot-spot disk is $f/100$. The actual disk is randomly chosen from among the set of current hot-spot disks. The set of hot-spot disks changes during the reference string. At any instant the probability of the set of hot spot disks changing is proportional to the duration for which the set has remained unchanged. The new set of k hot-spot disks is chosen at random from the set of all disks.

The motivation of the model is to capture a wide range of access patterns. When f is small data is accessed randomly from all disks allowing almost all prefetching algorithm to perform equally well. When f is close to 1, data is accessed sequentially from a small set of disks and again most algorithms will perform equally poorly due to the lack of parallelism to exploit. The intermediate range



(a) 5 Hot-Spot Disks (b) 35 Hot-Spot Disks

Figure 6: Performance of algorithms vs. percentage of hot-spotting blocks. Buffer = 250 block



(a) 5 Hot-Spot Disks (b) 35 Hot-Spot Disks

Figure 7: Performance of algorithms vs. percentage of hot-spotting blocks. Buffer = 1000 blocks

represents reference strings with substantial parallelism that can be exploited by intelligent prefetching algorithms.

In this study we compare the performance of L-OPT with algorithm NOM [3]. NOM is a natural generalization of a greedy policy: it always fetches a block if it is within M blocks of the missing requested block. We consider two configurations of the I/O system and count the number of I/Os performed by each algorithm. Though this approach neglects other factors occurring in practical I/O systems (like disk latency, block size, bus contention, etc.) this allows controlled evaluations of prefetching algorithms. We consider a system of 50 disks with 250 and 1000 blocks of I/O buffer. The lookahead is fixed at 3000 blocks.

Figure 6 presents a plot of the number of I/Os done by the four algorithms as the percent of random blocks is varied. The number of I/Os of all the algorithms is normalized by the number of I/Os done by the optimal off-line algorithm. The I/O system has a buffer of size 250. The number of hot spot disks is fixed at 5 and 35, respectively, in Figures 6(a) and (b). Figure 7 presents a similar plot when the buffer is of size 1000 blocks.

When the buffer is of size 250, the lookahead window spans 12 buffer-loads of accesses. Due to this there is practically no difference between the number of I/Os needed by the the best schedule with the L-block property and the optimal off-line. The loss due to incomplete lookahead becomes evident only in the case when the buffer is of size 1000.

In all cases the number of I/Os required by NOM and L-OPT increase with the percentage of blocks on hot-spot disks and then decreases. When the percentage of blocks on hot-spot disks is low there is not much additional parallelism to be extracted by fetching blocks required much later. Hence both NOM and L-OPT are close to the optimal. However as the percentage of such blocks increases, the optimal algorithm can exploit parallelism missed by both NOM and L-OPT. However L-OPT performs much better than NOM by extracting a substantial amount of parallelism from the lookahead window. Hence L-OPT follows the lower bound closely. As the percentage of blocks on the hot-spot disks increases further, most of the blocks are on the hot-spot disks and the situation is similar to that on the lower end. However the number of hotspot disks is smaller than the total number of disks, and the actual disks in the set change. This is the reason why the curves is not symmetric: OPT can benefit more from the variations than both NOM and L-OPT.

Finally as the number of hot spot disks increases the peaks of all curves shift to the right. This is because for a given percent of blocks that are on hot-spot disks, as the number of hot-spotting disks increases the number of additional I/Os that the algorithms need to do to catch up with the optimal schedule is fewer. Hence the percentage of such blocks has to be higher to get the same ratio.

In summary, apart from the extremes L-OPT performs very close to the best achievable with the given amount of lookahead. The primary loss of performance for L-OPT compared to the optimal off-line is its bounded lookahead. The difference between it and NOM is more pronounced when the amount of lookahead available is more compared to the amount of buffer size.

5 Conclusion

In this paper we have considered the problem of prefetching and I/O scheduling to service read-once reference strings in a parallel disk model. We presented an on-line algorithm L-OPT that uses L-block lookahead to decide the blocks to fetch in each I/O. L-OPT uses a novel priority scheme to determine which blocks in the lookahead are most useful. Priorities of blocks in the lookahead are dynamically updated as additional blocks are revealed; this allows L-OPT to closely track changes in the characteristics of the reference string and fetch the appropriate blocks. Alternatively, lookahead information may be availble in discrete chunks which could be of variable length. In such cases, L-OPT can schedule each window independently. The algorithm is simpler in this case because once the priorities of blocks in a lookahead window are computed, they need not be updated. Moreover, this also implies that L-OPT can avoid potentially wasteful I/Os: with a rolling lookahead window it may need to fetch a block only to evict it later in preference to a block of higher priority.

We presented bounds on the competitive ratio of L-OPT and showed that it is comparable to the best on-line algorithm with the same amount of lookahead. As a consequence we can quantify the maximum benefit of having increased lookahead. In addition, we also showed that when the lookahead matches the length of the reference string, L-OPT performs the minimal number of I/Os to service the reference string. This is the first optimal off-line I/O scheduling algorithm that we know of. Empirical analysis using synthetically generated reference strings shows that L-OPT is substantially better than previous known algorithms. The fact that L-OPT is the optimal off-line schedule to service a reference string can also be used as a benchmark in empirical evaluations of speculative prefetching heuristics. The same input trace of disk accesses when fed to L-OPT gives the minimal number of I/Os that any algorithm can do to service it, and hence provides a hard baseline.

References

[1] S. Albers, N. Garg, and S. Leonardi. Minimizing Stall Time in Single and Parallel Disk Systems. In *Proc. of 30th Annual ACM Symp. on Theory of Computer Science*, pages 454–462. ACM Press, May 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 the Fifth Workshop on I/O in Parallel and Distributed Systems*, pages 47–56. ACM Press, November 1997.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the Joint International Conf. on Measurement and Modeling of Computer Systems*, pages 188–197. ACM Press, May 1995.

[5] M. Kallahalla and P. J. Varman. Improving Competitiveness of Parallel-Disk Buffer Management using Randomized Writeback. In *Proc. of International Conf. on Parallel Processing*, pages 270–277, August 1998.

[6] M. Kallahalla and P. J. Varman. Red-Black Prefetching: An Approximation Algorithm for Parallel Disk Scheduling. In *Foundations of Software Technology and Theoretical Computer Science*, number 1530 in LNCS, pages 66–77. Springer-Verlag, December 1998.

[7] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive Snoopy Caching. *Algorithmica*, 5(3):79–119, March 1988.

[8] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *37th Annual Symp. on Foundations of Computer Science*, pages 540–549. IEEE, October 1996.

[9] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proc. First International Conf. on Parallel and Distributed Information Systems*, pages 182–189, December 1991.

[10] 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.

[11] R. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symposium on Operating Systems Principles*, pages 79–95, December 1995.

[12] P. J. Varman and R. M. Verma. Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems. In *Foundations of Software Technology and Theoretical Computer Science*, volume 16 of *LNCS*. Springer Verlag, December 1996.

[13] J. S. Vitter and E. A. M. Shriver. Optimal Algorithms for Parallel Memory, I: Two-Level Memories. *Algorithmica*, 12(2–3):110–147, 1994.

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