Tight Bounds for Prefetching and Buffer Management Algorithms for Parallel I/O Systems

Peter J. Varman, *Senior Member, IEEE*, and Rakesh M. Verma

Abstract—The I/O performance of applications in multiple-disk systems can be improved by overlapping disk accesses. This requires the use of appropriate prefetching and buffer management algorithms that ensure the most useful blocks are accessed and retained in the buffer. In this paper, we answer several fundamental questions on prefetching and buffer management for distributed-buffer parallel I/O systems. First, we derive and prove the optimality of an algorithm, P-min, that minimizes the number of parallel I/Os. Second, we analyze P-con, an algorithm that always matches its replacement decisions with those of the well-known demand-paged MIN algorithm. We show that P-con can become fully sequential in the worst case. Third, we investigate the behavior of on-line algorithms for multiple-disk prefetching and buffer management. We define and analyze P-lru, a parallel version of the traditional LRU buffer management algorithm. Unexpectedly, we find that the competitive ratio of P-lru is independent of the number of disks. Finally, we present the practical performance of these algorithms on randomly generated reference strings. These results confirm the conclusions derived from the analysis on worst case inputs.

Index Terms—Parallel I/O, algorithms, prefetching, caching, buffer management, competitive ratio, multiple-disk systems, external memory.

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 [6], [10] provides an opportunity to improve the I/O performance through the use of parallelism. However, it remains a challenging problem to exploit the increased disk bandwidth to reduce the I/O latency of an application. Effective use of parallel I/O systems requires buffer management algorithms that carefully coordinate the prefetching and caching policies.

Prefetching is a powerful technique that can be used to reduce the I/O latency seen by an application. Prefetching refers to reading a block of data from disk before it is actually required by the computation. In a parallel I/O system, prefetching can be used to overlap the operations of the disks. When the computation demands a disk-resident block of data, anticipatory reads on the other disks can be done concurrently to prefetch blocks that will be referenced

later. Prefetched blocks are held in the disk buffers until needed. The overlap in the reads can potentially decrease the number of I/Os by a factor equal to the number of disks. However, reading a block from a disk requires evicting some other block from the buffer to make space for the new block. This may trigger an additional I/O in the future in case the evicted block is referenced again. The buffer manager must choose between the relative benefit of increasing disk parallelism by prefetching now, against the increase in the number of I/Os due to suboptimal eviction choices. The actual disk parallelism achievable depends on the distribution of accesses to the disks, the size of the disk buffers and the buffer management policies.

Several different I/O organizations have been proposed for parallel I/O systems [18], [9]. A recent taxonomy of storage architectures is presented in [10]. Two logical configurations representing extreme points on the spectrum can be identified, based on centralized or distributed organizations of the I/O system. A traditional multiple-disk server represents the canonical globally-shared buffer organization. In such a system, the disks are attached to a centralized server using the system I/O bus, and caching is performed centrally at the server. However, such a scheme has limitations for scalability, as both the bus and the server are potential bottlenecks. Distributed storage architectures consisting of multiple disks interconnected by a high-speed network, such as the Fiber Channel Arbitrated Loop (FC-AL), are an emerging trend [10], [12], [15] directed towards distributed operation of the disks. In a typical configuration [10], [15] I/O requests are made to the server which forwards the I/O request to the appropriate disk using the storage network. The disk then fetches the data and communicates it directly back to the client. Each

- P.J. Varman is with the Department of Electrical and Computer Engineering, MS 366, Rice University, Houston, TX 77005. Email: pjv@rice.edu.
- R.M. Verma is with the Computer Science Department, 4800 Calhoun Road, University of Houston, Houston, TX 77204. E-mail: rmverma@cs.uh.edu.

Manuscript received 26 Jan., 1998; revised 01 Dec., 1998.

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 106207.

network-attached disk controls the layout and caching of its data, which is performed locally at that disk. This is similar to the distributed buffer model of parallel I/O used in this paper. A third possibility is to simulate a globally shared buffer organization on the physically distributed buffers. The potential gain is better buffer usage by smoothing local load imbalances between disks. The cost is much greater server complexity required to perform central coordination of the distributed cache, more sophisticated processing ability at the individual disks, and increased communication requirements. Cooperative caching in the environment of a network of workstations consisting of powerful processors with attached disks and executing applications directly was studied in [7].

In this paper, we describe and analyze buffer management algorithms for a distributed I/O architecture. The I/O system consists of several I/O nodes connected together by a communication network. An I/O node consists of a disk and associated disk buffer. Data from a disk is cached in the local buffer of that I/O node. The data for the computation is spread out among the disks in units of *blocks*. A block is the unit of retrieval from a disk. In each I/O at most one block from each disk can be read and stored in the disk's I/O buffer. The computation is characterized by a *reference string*, consisting of the ordered sequence of blocks that the computation accesses. A block must be brought into the I/O buffer before it can be delivered to the computation.

The buffer management algorithm is responsible for prefetching and caching decisions that are necessary to service the reference string. Its goal is to service the reference string in the smallest number of I/Os. To do this, it must decide when to initiate an I/O, from which disks to prefetch in an I/O, and from which blocks to evict the corresponding buffers. In order to prefetch, the buffer management algorithm must have some knowledge of the future accesses to be made by the computation [17]. This is referred to as *lookahead*. Without lookahead, only the block currently demanded by the computation can be fetched, and the disks will operate serially. By using the lookahead, prefetches from other disks can be made concurrently with the demanded block. Lookahead is also beneficial for caching by allowing the buffer manager to make better eviction decisions. We consider two scenarios: in an off-line situation, the entire reference string is available to the buffer management algorithm at the start of the computation; in an on-line situation, only partial knowledge of the future reference string (see Section 2.1) is known to the buffer manager at any time.

In this paper, we address several fundamental issues dealing with prefetching and caching in a distributed-buffer parallel I/O system. We present three algorithms (P-min, P-con, and P-lru), and study their performance using analysis and simulation. We define and prove the optimality of a greedy off-line algorithm, P-min, that minimizes the number of parallel I/Os. Secondly, we show that P-con, an algorithm that attempts to optimize the number of I/Os on each disk in isolation (by mimicking the eviction decisions of the optimal sequential algorithm), can have very poor parallel performance. The analysis of P-con and the simulation results for randomly generated data (see

Section 6) underscore the importance of prefetching even when it necessitates making imperfect eviction decisions.

The P-min and P-con algorithms analyzed in this paper are generalizations of the aggressive and conservative policies proposed by Cao et al. [5] for sequential I/O. However, while aggressive is suboptimal in the single disk I/O model of [5], P-min is proven to be the optimal algorithm in the distributed-buffer parallel I/O model. It is interesting to note that for a single disk, both P-min and P-con degenerate to the optimal single disk algorithm, MIN [4]. However, in the parallel case, P-min is optimal while P-con can be suboptimal by a factor proportional to the number of disks.

The performance of on-line buffer management algorithms is analyzed in the framework of competitive analysis. In our context, the competitive ratio is the worst case (over all possible reference strings) ratio of the number of I/Os done by the on-line algorithm to the number of I/Os needed by the optimal off-line algorithm P-min, to service the same reference string. We define local lookahead that permits future knowledge of only the next access on each disk, and analyze the performance of P-lru, a parallel on-line adaptation of the traditional Least Recently Used (LRU) buffer management algorithm, with local lookahead. The competitive ratio of P-lru is shown to be m , the size of each local buffer, and surprisingly, independent of the number of disks. This implies that P-lru reaps the benefits of increased parallelism that P-min gets with an increasing number of disks, while the performance penalty for suboptimal caching due to incomplete future knowledges does not increase. Finally, simulation experiments using synthetic reference strings were performed to study the performance of the algorithms in typical situations.

The rest of the paper is organized as follows: In Section 1.1, we summarize related work. Section 2 discusses the prefetching and caching problem for parallel I/O systems, followed by formal definitions and a summary of the main results. In Section 3, we show that algorithm P-min performs the minimal number of I/Os, and is therefore optimal in the distributed-buffer model. In Section 4, we derive a tight upper bound on the worst case performance of the P-con algorithm. In Section 5, we analyze the worst case performance of the on-line algorithm P-lru. Simulation results that complement and validate the analysis are presented in Section 6. We conclude with a discussion in Section 7.

1.1 Related Work

The formal study of buffer management algorithms for sequential I/O (also known as the paging problem), was pioneered by Belady [4] and Sleator and Tarjan [19]. The sequential paging algorithm MIN, based on the longest-forward-distance eviction policy, was shown to minimize the number of page faults [4]. Competitive analysis of on-line paging algorithms was studied in [19]. It was shown that no deterministic on-line paging algorithm can be better than m -competitive, where m is the size of the main-memory buffer, and that LRU achieves the best possible competitive ratio. These fundamental results have been expanded to include models that allow different forms of lookahead [1], [8], whereby the algorithm has access to

some portion of the future request sequence in determining the block to evict. These results deal exclusively with demand-fetched I/O, since in the sequential disk model considered, there is no benefit to be gained by prefetching.

The use of prefetching from a single disk to overlap CPU and I/O operations, was addressed in [5]. As mentioned earlier, two off-line algorithms called *aggressive* and *conservative* were analyzed, and bounds on the elapsed time relative to the optimal prefetching algorithm were derived. In a single disk system without prefetching, there is no overlap between the disk and the CPU; in this case, the elapsed time is within a factor two of that achievable by the optimal prefetching algorithm. In the model of [5], the *aggressive* algorithm has a worst case ratio between 1 and 2 over optimal prefetching, depending on the relative speeds of the disk and CPU, and the size of the buffer. The issues involved in prefetching in a multiple-disk parallel I/O context were, however, not considered in that paper.

An alternate parallel disk model based on multiple disks sharing a global buffer [21] has been studied from the viewpoint of buffer management. Patterson et al. [17], Kimbrel and Karlin [13], Barve et al. [2], Varman [20], and Kallahalla and Varman [11] have presented and analyzed different buffer management algorithms under varying assumptions. In the TIP model [17], Patterson et al. define a cost/benefit model to compare the relative cost of reclaiming a buffer by eviction against the benefit obtained by prefetching a block into that buffer. This is used to guide the decisions of the buffer manager. The approximations made in the model favor the case of shallow prefetching within a small lookahead window and the case when the requests within the lookahead are distributed randomly across the disks. Kimbrel and Karlin [13] considered a more general model with complete off-line knowledge of the reference string and arbitrary load distributions across the disks. They presented a sophisticated approximation algorithm called reverse-aggressive, and analyzed the worst case ratio of the elapsed time relative to the optimal algorithm. Within realistic ranges of technology-related parameters for CPU and disk speeds, typical buffer sizes, and a small number of disks, the approximation ratio was shown to be small. A useful special class of read-once reference strings where each block was accessed exactly once was considered in [2]. On-line algorithms for different models of lookahead were presented and optimal and near-optimal bounds were demonstrated. In [20], randomized algorithms for read-once reference strings in a global shared-buffer model were presented and analyzed. Recently, in [11], an on-line algorithm using dynamic, randomized relocation of the blocks for a general reference string was proposed and analyzed.

Parallel prefetching algorithms in the context of specific applications were discussed in [14], [16], [3]. In terms of analysis, the shared-buffer model presents a challenging optimization problem for buffer management. At present, there is no known optimal off-line algorithm for general reference strings besides the brute-force search method. In contrast, in this paper we present an optimal off-line algorithm for general reference strings for the distributed buffer model.

2 PRELIMINARIES

In this section, we define the buffer management problem for parallel I/O, starting with an intuitive description. The computation references the blocks on the disks in an order specified by the *reference string* Σ . When a block is referenced, the buffer for that disk is checked; if the block is present in the buffer, it is directly returned to the computation, which then proceeds to reference the next block in Σ . If the referenced block is not present in the disk buffer, then an I/O (known as a *demand* I/O) for the missing block is initiated from that disk. If only demand I/Os were initiated, then the other disks in the system would idle while this block was being fetched. However, every demand I/O at a disk provides a *prefetch* opportunity at the other disks, which may be used to read blocks that will be referenced in the near future. For example, consider a two disk system holding blocks (a_1, a_2) and (b_1, b_2) on disks 1 and 2, respectively. If $\Sigma = a_1, b_1, b_2, a_2$, then a strictly demand I/O would require four nonoverlapped I/Os to fetch the blocks. A better strategy is to overlap reads using prefetching. During the demand I/O for block a_1 , the second disk could concurrently prefetch b_1 ; after a_1 and b_1 have been consumed, a demand I/O for block b_2 will be made concurrently with a prefetch of block a_2 . The number of parallel I/Os in this case is now two.

While prefetching increases the I/O parallelism, the problem is complicated by the bounded size of the buffers. For every block read from a disk, some previously fetched block in the corresponding buffer must be evicted. For prefetch blocks, the eviction decision is being made earlier than is absolutely necessary, since the computation can continue without the prefetched block. These early eviction choices can be much poorer than eviction choices made later, since as the computation proceeds, a better candidate for eviction may become available. Of course, once a block becomes a demand block, then an eviction cannot be deferred. Poor eviction choices result in a greater number of I/Os, as these prematurely discarded blocks may have to be fetched repeatedly into the buffer. There is, therefore, a tradeoff between the I/O parallelism that can be achieved by prefetching, and the increase in the number of I/Os required due to suboptimal evictions.

2.1 Definitions

The distributed-buffer system has d disks and a buffer of size m blocks at each disk. The reference string Σ is the order in which blocks are requested by the computation. The subsequence of Σ consisting of blocks from *disk* i will be denoted by Σ_i .

Computation occurs in *rounds*, with each round consisting of an I/O *phase* followed by a *computation phase*. In the I/O phase of the round, a parallel I/O is initiated and some number of blocks, at most one from any disk, are selected to be read. For each selected disk, a block in the corresponding disk buffer is chosen for replacement. When all new blocks have been read from the disks, the I/O phase ends and the computation phase begins. In the computation phase, the CPU consumes zero or more blocks that are present in the buffer in the order specified by Σ . If at any point the next block of Σ is not present in a buffer, then the round ends

and the next round begins. The block whose absence forced the I/O is known as a *demand block*; blocks that are fetched together with the demand block are known as *prefetch blocks*. An I/O phase may also be initiated before the computation requires a demand block. In this case, all the blocks fetched are prefetch blocks. We will also refer to the I/O phase of a round as an I/O time step.

Definition 1. An I/O schedule with length T is a sequence $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_T$, where $\mathcal{F}_k$ is the set of blocks, at most one from each disk, fetched by the parallel I/O at time step k . The length of a schedule is the number of I/O time steps required to complete the computation.

Definition 2.

- In a valid schedule, a block must be present in the buffer before it can be consumed, and no buffer should hold more than m blocks.
- An optimal schedule is a valid schedule with minimal length among all valid schedules.
- A normal schedule is a valid schedule in which each $\mathcal{F}_k$, $1 \leq k \leq T$, contains a demand block.
- A sequential schedule is a valid schedule in which the blocks from each disk i are fetched in the order of Σ_i .

At the start of a round, let U_i denote the block of Σ_i that will be referenced next and is not currently in the buffer of disk i . Define a *min-block* of disk i to be the block in disk i 's buffer with the longest forward distance to the next reference.

Definition 3. A *P-min* schedule $S = \langle \mathcal{F}_k, k = 1, \dots, T \rangle$ is a normal, sequential schedule in which at I/O step k , $U_i \in \mathcal{F}_k$, unless all blocks in disk i 's buffer are referenced before U_i . If $U_i \in \mathcal{F}_k$, then replace the min-block of disk i with U_i .

Definition 4. A *P-con* schedule $S = \langle \mathcal{F}_k, k = 1, \dots, T \rangle$ is a normal, sequential schedule in which at every I/O step k , $U_i \in \mathcal{F}_k$, provided the min-block of disk i is now the same as the min-block if U_i were fetched on demand at the next request to U_i . If $U_i \in \mathcal{F}_k$, then replace the min-block of disk i with U_i .

Definition 5. A *P-lru* schedule $S = \langle \mathcal{F}_k, k = 1, \dots, T \rangle$ is a normal, sequential schedule in which at every I/O step k , $U_i \in \mathcal{F}_k$, unless all blocks in disk i 's buffer are referenced before U_i . If $U_i \in \mathcal{F}_k$, then from among the blocks in the buffer whose next reference is not before that of U_i choose the least recently used block and replace it with U_i .

Notice that all the schedules defined above are normal. That is, in every I/O step, one disk is performing a demand fetch and the rest either perform a prefetch or idle. Also, P-min uses a greedy prefetching strategy, and will prefetch the next required disk-resident block, unless all the blocks in that buffer will be referenced before the new block. Note that this greedy prefetching may require making suboptimal replacement choices that may result in an increase in the number of I/Os done by that disk. However, P-min will be shown to have the minimal parallel I/O time. In contrast, P-con attempts to minimize the number of I/Os done by any

Round	Disk 1	Disk 2	CPU
1	a_1 /–	b_1 /–	a_1
2	a_2 /–	b_2 /–	a_2
3	a_3 /–	b_3 /–	a_3, b_1
4	a_4 / a_3	b_4 / b_1	$a_4, a_1, a_2, b_2, b_3, b_4$
5	a_3 / a_4	b_1 / b_4	$a_3, b_1, b_2, b_3, a_1, a_2, a_3, b_1$
6	a_4 / a_3	–/–	a_4, a_1, a_2

Fig. 1. P-min Schedule for ω .

single disk. For each disk i , P-con performs the same sequence of fetches as the MIN algorithm on the sequence Σ_i , except that a fetch may be done at a possibly earlier time. However, while minimizing the number of I/Os done by a disk, P-con can end up serializing these accesses, and perform significantly worse than the optimal algorithm.

Finally, P-lru makes its replacement decision based on the past history of accesses, and the references in the lookahead window up to the next block not in the buffer. Note that this lookahead is needed by the prefetching algorithm to identify the block to prefetch on the next parallel I/O.

Example. Below, an example with $d = 2$ and $m = 3$ is presented. Let the blocks on disk 1 (2) be $a_i, i = 1, \dots, 4$ (respectively $b_i, i = 1, \dots, 4$). Assume that the reference string ω is given by:

$$\omega = a_1 a_2 a_3 b_1 a_4 a_1 a_2 b_2 b_3 b_4 a_3 b_1 b_2 b_3 a_1 a_2 a_3 b_1 a_4 a_1 a_2.$$

Figs. 1, 2, and 3 show the P-min, P-con, and P-lru schedules for the example reference string ω . The entries in the second and third columns indicate the blocks that are fetched and replaced from that disk at that round. Bold and italic faced blocks indicate a demand block and a prefetch block, respectively. The entry s/t in the figures means that block s is fetched and block t is replaced.

In Fig. 1 after the first three steps, the buffers of both disks have been filled and blocks a_1, a_2, a_3 , and b_1 have been consumed. At step 4, demand block a_4 is fetched from disk 1, replacing the current min-block a_3 , whose next forward reference occurs later than a_1 and a_2 . At the same step, the next unread block of disk 2, b_4 , is prefetched from disk 2, replacing b_1 . The next six blocks of ω are present in the buffers and are consumed directly, after which a demand is made for block a_3 . Continuing in this way, the entire sequence is completed in six I/O steps.

The schedule created by P-con in servicing the same reference string is shown in Fig. 2. Note that at step 4, no block is fetched from disk 2 by P-con. This is because the only candidate for replacement at this time (the current min-block) is block b_1 ; however, if one were to defer fetching b_4 until it became a demand block, the min-block would be b_3 . Since these two replacement choices differ, b_4 is not fetched at step 4.

The schedule created by P-lru is shown in Fig. 3. Note that in step 5, P-lru does not perform a prefetch. This is because the next required block not in the buffer of disk 2 is b_1 , but all of the blocks currently in that buffer (b_2, b_3 , and b_4) are accessed before the next reference to b_1 .

Round	Disk 1	Disk 2	CPU
1	$\mathbf{a_1}/-$	$b_1/-$	a_1
2	$\mathbf{a_2}/-$	$b_2/-$	a_2
3	$\mathbf{a_3}/-$	$b_3/-$	a_3, b_1
4	$\mathbf{a_4}/a_3$	$-/-$	a_4, a_1, a_2, b_2, b_3
5	a_3/a_4	$\mathbf{b_4}/b_3$	b_4, a_3, b_1, b_2
6	$-/-$	$\mathbf{b_3}/b_4$	b_3, a_1, a_2, a_3, b_1
7	$\mathbf{a_4}/a_3$	$-/-$	a_4, a_1, a_2

Fig. 2. P-con Schedule for ω .

2.2 Summary of Results

The three buffer management algorithms P-min, P-con, and P-lru are analyzed and tight bounds on the number of I/Os required are derived. The formal results from the analysis are summarized below, followed by a discussion of their significance. These analytical results were complemented with a simulation study using synthetically generated reference strings to study the performance of the algorithms for typical inputs. The conclusions drawn from the analysis for worst case inputs were validated by the simulations.

1. The schedule created by P-min minimizes the number of parallel I/Os. P-min is, therefore, the optimal buffer management algorithm in the distributed-buffer model (see Theorem 1).
2. For any $\epsilon > 0$, there exists a reference string such that the ratio between the lengths of the schedules created by P-con and P-min is $(d - \epsilon)$ (see Theorem 2). That is, the disk accesses of P-con can be almost completely serialized.
3. Using local lookahead, the competitive ratio of P-lru is m , independent of the number of disks (see Theorem 4). Using an analysis similar to that in [19] it can be shown that this is the best ratio achievable by any on-line algorithm with this lookahead.

The optimality of a simple greedy prefetching algorithm like P-min is interesting. P-min always prefetches from a disk if the block to be prefetched is required earlier than some block currently in the disk's buffer. By so doing, each disk potentially commits to performing a compensating I/O later to refetch the evicted block. Since the compensating I/Os on the disks are independent, and each disk will attempt to greedily prefetch once again during the compensating I/O of another disk (thereby generating further commitments for compensating I/Os), it may appear that an optimal algorithm should judiciously refuse some prefetching opportunities. The results show that this is not the case and the greedy prefetching of P-min does result in the minimum possible number of parallel I/Os.

The comparison of P-con and P-min highlights the tension between making optimal caching decisions and the benefits of prefetching. P-con guarantees that no disk will ever need to perform an unnecessary compensating I/O, i.e., its eviction decisions ensure that each disk taken in isolation makes the smallest number of disk accesses possible. However, refusing these prefetching opportunities actually results in much poorer overall performance. The worst case dilation in the length of the schedule is almost d times that of the optimal greedy schedule of P-min.

Round	Disk 1	Disk 2	CPU
1	$\mathbf{a_1}/-$	$b_1/-$	a_1
2	$\mathbf{a_2}/-$	$b_2/-$	a_2
3	$\mathbf{a_3}/-$	$b_3/-$	a_3, b_1
4	$\mathbf{a_4}/a_1$	b_4/b_1	a_4
5	$\mathbf{a_1}/a_2$	$-/-$	a_1
6	$\mathbf{a_2}/a_3$	$-/-$	a_2, b_2, b_3, b_4
7	a_3/a_4	$\mathbf{b_1}/b_2$	a_3, b_1
8	$-/-$	$\mathbf{b_2}/b_3$	b_2
9	$-/-$	$\mathbf{b_3}/b_4$	b_3, a_1, a_2, a_3, b_1
10	$\mathbf{a_4}/a_1$	$-/-$	a_4
11	$\mathbf{a_1}/a_2$	$-/-$	a_1
12	$\mathbf{a_2}/a_3$	$-/-$	a_2

Fig. 3. P-lru Schedule for ω .

Simulation results show that P-con's performance degrades with increased number of disks for randomly distributed access patterns, as well.

The benefit of prefetching even at the expense of making suboptimal caching decisions is also implicit in the behavior of P-lru. Like P-min, P-lru greedily prefetches from a disk whenever the prefetched block is referenced earlier than some block in that disk's buffer. Its corresponding eviction decisions may be much worse than P-min, since for any disk it has no knowledge of the request sequence beyond the prefetched block. The analysis shows that in the worst case the number of I/Os done by P-lru relative to P-min is independent of the number of disks. This behavior is also validated by the simulation results where P-lru's performance is seen to be relatively insensitive to the number of disks.

The independence of P-lru's competitive ratio from d suggests that as the number of disks increases, P-lru also gains the benefits of greedy prefetching reaped by P-min. The penalty for its poorer eviction decisions, in contrast, does not increase with the number of disks; as a consequence, P-lru will tend to do better with larger numbers of disks. Although P-lru's worst case competitive ratio is m (due to poor eviction choices similar to the serial LRU algorithm), the performance for random data is much closer to that of P-min. The similarity in the caching decisions made by traditional LRU and P-lru suggests that in many practical situations (where LRU is found to be acceptable), the performance impact of poor caching decisions of P-lru will be much smaller than the worst case ratio suggests, while the parallelism gains due to greedy prefetching will scale upwards with more disks.

3 OPTIMALITY OF P-MIN

In this section, we show that P-min requires the minimal number of parallel I/O steps among all valid schedules. For the proof, we show how to transform an optimal schedule OPT with length L , into a P-min schedule with the same length. This shows that P-min performs the optimal number of parallel I/Os.

Definition 6. Schedules α and β are said to match for time steps $[T]$, if for every t , $t \in \langle 1, \dots, T \rangle$, the blocks fetched and replaced from each disk in the two sequences are the same.

Lemma 1. Assume that α is a valid schedule of length W . Let γ be another schedule which matches α for $[T - 1]$. After the I/O at time step T , the buffers of α and γ for some disk i differ in one block: specifically, α has block V but not block U , and γ has block U but not block V . Assume that V is referenced after U in the reference string following the references at time $T - 1$. We can construct a valid schedule β of length W , such that β and α match for $[T - 1]$ and β and γ match at time step T .

Proof. Let $T + \delta, \delta > 0$, be the first time step after T at which α discards block V or fetches block U (or does both). Construct schedule β as follows: β matches α for time steps $[W]$ except at time steps T and $T + \delta$.

At T , β fetches and replaces the same blocks as γ . At $T + \delta$, one of the following must occur:

- α fetches a block $Z \neq U$ and discards block V : then β will also fetch block Z , but will discard block U .
- α fetches block U and discards block $Z, Z \neq V$: then β will fetch block V and will also discard block Z .
- α fetches block U and discards block V : then β does not fetch or discard any block.

In all three cases above, following the I/O at $T + \delta$, both β and α will have the same blocks in the buffer. Since β fetches and replaces the same blocks as α for all time steps $t \geq T + \delta + 1$, the buffers of α and β will be the same for all time steps after the I/O at $T + \delta$.

At each time step $t, 1 \leq t \leq W$, β will consume the same blocks as done by α at t . We show that β is a valid schedule by showing that every block consumed by β is present in the buffer.

For time steps $t, t \leq T - 1$ or $t \geq T + \delta$, there is nothing to show, since α and β have the same buffer. During the time interval $[T, T + \delta - 1]$, only blocks $X \neq U, V$ are requested, because U is not in the buffer maintained by α , and V is requested later than U . Since the buffers of α and β agree except on $\{U, V\}$, X can also be consumed by β at the same time step. Since α is a valid schedule, all consumptions of β are also valid. Hence, β is a valid schedule. $\square$

Theorem 1. P -min is an optimal schedule.

Proof. Let Δ and Ω denote the schedules created by P -min and OPT algorithms, respectively. We successively transform Ω into another valid schedule that matches Δ and has the same length as Ω . This will show that the P -min schedule is optimal.

The proof is by induction. For the induction hypothesis, assume that at time step t , Ω has been transformed to a valid schedule Ω_t , which matches Δ at time steps $[t]$. We show how to transform Ω_t to Ω_{t+1} below.

We discuss the transformation for an arbitrary disk at time step $t + 1$. The same construction is applied to each disk independently. If Δ and Ω_t match at $t + 1$, then let Ω_{t+1} be the same as Ω_t . Suppose Δ and Ω_t differ at time step $t + 1$. Then one of the following three cases must occur at $t + 1$:

- Case 1: Δ fetches a block but Ω_t does not fetch any block.
- Case 2: Ω_t fetches a block but Δ does not fetch any block.
- Case 3: Δ and Ω_t fetch different blocks.

We consider each case separately.

- Case 1: Let Δ fetch block P and discard block Q at $t + 1$. Since Δ will not fetch a block if it is not needed before the one that has to be replaced, P will be referenced before Q . From the induction hypothesis, Δ and Ω_t have the same buffer at the start of time step $t + 1$. Hence, after the I/O at $t + 1$, Δ and Ω_t differ in one block: Δ has block P but not block Q , while Ω_t has Q but not P .

Using Lemma 1 with $\alpha = \Omega_t, \beta = \Omega_{t+1}, \gamma = \Delta, U = P, V = Q, T = t + 1$, we can construct valid schedule Ω_{t+1} that matches Ω_t at time steps $[t]$ and Δ at time $t + 1$. Hence, the induction hypothesis is satisfied for $t + 1$.

- Case 2: Since Δ does not fetch any block at time step $t + 1$, every block in the buffer at the start of time step $t + 1$ will be consumed before any block not currently in the buffer is referenced.

Since Δ and Ω_t have the same buffer at the start of time step $t + 1$, if Ω_t brings in a fresh block (P) at $t + 1$, it must discard some block (Q). Since Δ chose to retain block Q in preference to fetching block P , then either Q must be referenced before P , or neither P nor Q will be referenced again.

In the first case, using Lemma 1 with $\alpha = \Omega_t, \beta = \Omega_{t+1}, \gamma = \Delta, U = Q, V = P, T = t + 1$, we can construct Ω_{t+1} , a schedule that satisfies the induction hypothesis for $t + 1$.

In the second case, Ω_{t+1} is the same as Ω_t , except that at time step $t + 1$, Ω_{t+1} does not fetch any block. Since, the buffers of Ω_t and Ω_{t+1} agree on all blocks except P and Q , and these two blocks are never referenced again, all blocks consumed by Ω_t at a time step can also be consumed by Ω_{t+1} at that time.

- Case 3: Suppose that Δ fetches block P and discards block Q at $t + 1$, and Ω_t fetches block Y and discards block Z at $t + 1$. Assume that $Q \neq Z$, since otherwise the buffers of Ω_t and Δ differ in just the pair of blocks $\{P, Y\}$, and we can easily construct Ω_{t+1} as before by using Lemma 1 with $\alpha = \Omega_t, \gamma = \Delta, \beta = \Omega_{t+1}, U = P, V = Y, T = t + 1$.

By the induction hypothesis, Δ and Ω_t have the same buffer at the start of time step $t + 1$. Hence, after the I/O at $t + 1$, Δ and Ω_t differ in two blocks. Specifically,

$$\mathcal{B}(\Delta) = (\mathcal{B}(\Omega_t) - \{Y, Q\}) \cup \{P, Z\},$$

where $\mathcal{B}(\Theta)$ is the set of blocks in the buffer of schedule Θ .

Let $t + \delta, \delta > 1$, be the first time after $t + 1$ that Ω_t fetches or replaces a block $W \in \{P, Q, Y, Z\}$. It

Fig. 4. Performance of P-con vs. P-min.

can either discard block Q or Y , or fetch block P or Z , or some appropriate combination of these (see cases below), at $t + \delta$. Construct schedule Ω'_{t+1} as follows: Ω'_{t+1} matches Ω_t at all time steps except $t + 1$ and $t + \delta$. At $t + 1$, Ω'_{t+1} fetches P and discards Q , following the actions of Δ at this time step. Hence after the I/O at $t + 1$,

$$\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Y, Q\}) \cup \{P, Z\} = \mathcal{B}(\Delta).$$

At $t + \delta$, one of the following will occur:

- Ω_t fetches block $S \notin \{P, Z\}$ and discards Q : then Ω'_{t+1} also fetches S , but discards Z . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Y\}) \cup \{P\}$.
- Ω_t fetches P and discards Q : then Ω'_{t+1} fetches Y and discards Z . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = \mathcal{B}(\Omega_t)$.
- Ω_t fetches Z and discards Q : then Ω'_{t+1} does nothing at this step. After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Y\}) \cup \{P\}$.
- Ω_t fetches $S \notin \{P, Z\}$ and discards Y : then Ω'_{t+1} also fetches S , but discards P . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Q\}) \cup \{Z\}$.
- Ω_t fetches P and discards Y : then Ω'_{t+1} does not fetch any block at this time step. After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Q\}) \cup \{Z\}$.

- Ω_t fetches Z and discards Y : then Ω'_{t+1} fetches Q and discards P . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = \mathcal{B}(\Omega_t)$.
- Ω_t fetches P and discards block $S \notin \{Q, Y\}$: then Ω'_{t+1} fetches Y and discards block S . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Q\}) \cup \{Z\}$.
- Ω_t fetches Z and discards block $S \notin \{Q, Y\}$: then Ω'_{t+1} fetches Q and discards block S . After the I/O at $t + \delta$, $\mathcal{B}(\Omega'_{t+1}) = (\mathcal{B}(\Omega_t) - \{Y\}) \cup \{P\}$.

Consider the consumptions made by Ω_t at time steps T , $t + 1 \leq T \leq t + \delta - 1$.

Notice that in the reference string P must precede both Q and Y , and Z must precede Q . The constraints on P follow, since Δ fetches P and discards Q , and fetches P in preference to Y . The constraint on Z follows since Δ discards Q rather than Z .

We now show how to transform Ω'_{t+1} to Ω_{t+1} . If the buffers of Ω'_{t+1} and Ω_t are the same after the I/O at $t + \delta$, then let $\Omega_{t+1} = \Omega'_{t+1}$. Otherwise, the buffers must differ in either the pair of blocks $\{Q, Z\}$ or $\{Y, P\}$.

We will construct Ω_{t+1} by concatenating the prefix of Ω'_{t+1} between time steps 1 and $t + \delta - 1$ with a schedule β that will be constructed using Lemma 1, as described below.

Fig. 5. Performance of P-lru vs. P-min.

Let α and γ be, respectively, the schedules consisting of the suffixes of Ω_t and Ω'_{t+1} for time steps greater than or equal to $t + \delta$. If at the end of the I/O at $t + \delta$, the buffers of Ω_t and Ω'_{t+1} differ in $\{Y, P\}$, then let $U = P$ and $V = Y$; otherwise, if they differ in $\{Q, Z\}$, then let $U = Z$ and $V = Q$. Applying Lemma 1 with $T = 1$, we can construct the desired sequence β . Schedule Ω_{t+1} is obtained by concatenating the prefix of Ω'_{t+1} between 1 and $t + \delta - 1$ with β .

The consumptions of blocks in Ω_{t+1} are as follows: for time steps T , $1 \leq T \leq t + \delta - 1$, the consumptions are those of Ω_t , and for $T \geq t + \delta$ the consumptions are determined by β . All consumptions from 1 until t are valid, since Ω_t is a valid schedule, and Ω_{t+1} and Ω_t match for $[t]$. By construction, consumptions after $t + \delta$ onwards are valid. We need to show that Ω_{t+1} can consume the same blocks as Ω_t at time steps T , $t + 1 \leq T \leq t + \delta - 1$. Since Ω_t does not have P or Z in buffer at the end of the I/O at $t + 1$, it can consume P or Z only after the I/O at time $t + \delta$ or later. Also, since Q and Y must be consumed *after* P , none of the blocks P, Q, Y, Z can be consumed by Ω_t before the I/O at $t + \delta$. Since after the I/O at $t + 1$ the buffers of Ω_t and Ω_{t+1} agree except on $\{P, Q, Y, Z\}$, all blocks consumed by Ω_t between $t + 1$ and $t + \delta - 1$ can also be consumed by Ω_{t+1} at that time step.

This concludes the proof. $\square$

4 BOUNDS FOR P-CON

We begin with a simple proof of the upper bound for T_{P-con} . Let T_{MIN} denote the maximum number (over all disks i) of I/Os done by the sequential MIN algorithm to disk i for Σ_i .

Theorem 2. For any reference string, $T_{P-con} \leq dT_{opt}$.

Proof. We show that $T_{P-con} \leq dT_{MIN} \leq dT_{opt}$. The I/Os made by P-con to disk i for reference string Σ , are exactly the I/Os done by the sequential MIN algorithm to disk i for sequence Σ_i . Hence, the number of I/Os performed by any disk in P-con is bounded by T_{MIN} . At worst, none of the accesses of any of the disks can be overlapped whence the first inequality follows. Finally, the second inequality follows, since the optimal parallel time for d disks cannot be smaller than the minimal number of I/Os for a single disk. $\square$

Theorem 3. For $m \geq d$ and every $\epsilon > 0$, there exists a reference string such that $T_{P-con} \geq (d - \epsilon)T_{opt}$.

Proof. Construct the following four length- m sequences of blocks of disk i , where $B_i(j)$ refers to the j^{th} block on disk i .

$$\alpha_i = B_i(2m)B_i(1)B_i(2) \cdots B_i(m-1)$$

$$\beta_i = B_i(m), B_i(1)B_i(2) \cdots B_i(m-1)$$

Fig. 6. Performance with varying number of disks ($r = 10,000$).

$$\gamma_i = B_i(m)B_i(m+1)B_i(m+2) \cdots B_i(2m-1)$$

$$\delta_i = B_i(2m)B_i(m+1)B_i(m+2) \cdots B_i(2m-1).$$

Define a reference string Σ as follows, where $(u)^n$ means n repetitions of the sequence within the parentheses. We shall show that for Σ , $T_{P-con} \geq d(1 - 1/n)T_{opt}$.

$$\Sigma = (\alpha_1\beta_1\alpha_2\beta_2 \cdots \alpha_d\beta_d\gamma_1\delta_1\gamma_2\delta_2 \cdots \gamma_d\delta_d)^n.$$

In the first m I/Os, all m buffers of each disk are filled with the blocks of sequence $\alpha_i, i = 1, \dots, d$. In step $m+j, 1 \leq j \leq d$, an I/O is done to fetch $B_j(m)$ from disk j ; hence, at the end of step $m+d$, P-con has processed the subsequence $\alpha_1\beta_1\alpha_2\beta_2 \cdots \alpha_d\beta_d$ of Σ . At this point, the buffer of each disk i has the blocks $B_i(1), B_i(2), \dots, B_i(m)$, and P-con will begin processing γ_1 by referencing block $B_1(m)$ next.

The system enters a repetitive loop now as follows: in m consecutive steps blocks $B_1(m+1)$ through $B_1(2m)$ of disk 1 are fetched, followed by the same block numbers from disk 2, then from disk 3 and so on. At each of these time steps, exactly one of the disks (the disk with the request for the demand block) is active. The remaining disks will not prefetch any block as it will require them to evict a block different from the one they would replace at the time the prefetched block becomes a demand block. After md steps, each disk

would have fetched m blocks and consumed $2m$ blocks. Another cycle of md block fetches and $2md$ block references follows, serially from disk 1 through disk d , fetching blocks $B_i(1), \dots, B_i(m)$. The two sequences now alternate until the string is exhausted. Thus, the total I/O time required is $m + d + 2md(n-1) + md$ steps. Hence,

$$T_{P-con} = m + d + (2n-1)md.$$

It is not difficult to see that P-min can parallelize all accesses after step $m+d-1$, as well as the first m accesses. Therefore, T_{opt} is of length $2nm + d$, for $m \geq d$. Thus, the ratio of the two times T_{P-con}/T_{opt} is lower bounded by:

$$\frac{m + d + (2n-1)md}{2nm + d} \geq (d - d/n)$$

for $m \geq d$. By choosing n large enough, the ratio of the two times can be made arbitrarily close to d . $\square$

5 BOUNDS FOR P-LRU

In this section, we obtain an upper bound on the worst case performance of P-lru, and show that this bound is tight. Recall that P-lru makes its replacement decisions by examining the current blocks in the buffer and tracking which of them will be referenced before the next unread block. From those blocks whose next reference is not before

Fig. 7. Performance with varying number of disks ($r = 100$).

that of the next unread block, the least recently consumed block is chosen as the replacement candidate.

Lemma 2. *Let disk i be fixed but arbitrary. Let S be a contiguous subsequence of Σ which references m or less distinct blocks from disk i . Then in consuming S , none of these blocks will be fetched more than once by P-lru.*

Proof. For the sake of contradiction, assume that while consuming S , P-lru fetches block A , evicts it, and then fetches it again. At the time A is evicted it must have been referenced. Furthermore, each of the other $m - 1$ blocks in the buffer must either be referenced in the current lookahead (and hence are not candidates for eviction), or must have been referenced later than A (since A is least recently used among blocks not in the lookahead). Consequently, the block being fetched by evicting A is the $(m + 1)^{th}$ distinct reference in S , which is a contradiction. $\square$

Theorem 4. *For all reference strings, $T_{P-lru} \leq mT_{opt}$.*

Proof. We show that an arbitrary reference string that takes n time steps using the optimal algorithm P-min, can be done in mn time steps by P-lru. The proof is by induction on the time steps of P-min.

Clearly, the first time step of both P-min and P-lru will be the same, since both P-min and P-lru will fetch one block from each disk and consume it as many times as needed.

For the induction hypothesis, assume that the consumptions made in the first t steps of P-min's schedule can be performed in no more than mt steps by P-lru. We will show that the consumptions of the first $t + 1$ steps of P-min's schedule can be performed by P-lru in $(t + 1)m$ steps. We do this by showing that the block consumptions made at time step $t + 1$ of P-min's schedule will be done within an additional m steps by P-lru.

Let U_i be the set of references made by P-min at time step $t + 1$ from disk i . Note that the size of U_i , $|U_i| \leq m$, since at most m distinct blocks can be consumed from any disk at a time step of P-min. Also note that even though P-min may consume the same block several times at this time step, P-lru will fetch this block only once. This follows from Lemma 2. Since there are at most m distinct references made at this time step, there will be at most m fetches by P-lru. Hence, within m time steps, all the blocks of U_i can be fetched and consumed. $\square$

Theorem 5. *The worst case bound of Theorem 4 is tight.*

Proof (sketch). We show the construction of Σ for two disks. The generalization to d disks is straightforward. In the reference string below, a_i (b_i), $i = 1, \dots, m + 1$ represent blocks from disks 1 and 2, respectively. Let

$$\Sigma = (a_1, \dots, a_m, b_1, a_{m+1}, a_1, \dots, a_{m-1}, b_2, \dots, b_{m+1}, a_m, b_1, \dots, b_m)^n.$$

Fig. 8. Performance with varying buffer size ($r = 5,000$).

It is easily seen that after the first m accesses, which are common to both P-lru and P-min, P-lru makes m accesses for every access of P-min. □

6 EXPERIMENTAL PERFORMANCE

A simulation program was developed to compare the performance of the different prefetching algorithms. Synthetic data traces were generated as follows: a parameter *range* determines the set of possible values that the blocks can take. A small value for *range* means that the reference string contains many references to any given block, while a large value for *range* means that it approximates a read-once or read-seldom reference string, where each block is accessed only a few number of times.

Two different data models were used in generating the sequence for each disk. In the *uniform model*, each block was independently assigned a random value in *range* with equal probability. In the *normal model*, the probability distribution for the blocks on a disk followed a normal distribution with mean equal to half the maximum *range* value, and standard deviation equal to a sixth of the *range* value. The reference string was obtained by randomly interleaving the individual sequences for each disk, i.e., the next block to be referenced can be from any disk with equal probability.

The parameters varied included the buffer size m at each disk, the number of disks d , and the *range* r . The results

were averaged over several runs for each data point and then reported. In each case, the average number of blocks from each disk in the reference string was kept fixed at 2,000. Several experiments were performed with both data distributions. The results for the uniform model are presented in the following discussion. The results that were observed with the normal distribution agree substantially with the results presented below and therefore are omitted.

When r becomes large, the reference string tends toward a read-once sequence. In a read-once reference string, no block is referenced more than once, so any block that has already been referenced is equally good as an eviction candidate. Hence, the eviction decisions of P-min, P-lru and P-con are equivalent. Furthermore, P-con becomes a greedy prefetching algorithm in this situation, since it can always find a block as good as P-min's to evict. Thus, all three algorithms should have identical performance in this situation. Note that the maximum number of distinct blocks on any disk gives a lower bound on the number of I/Os required.

Fig. 4 compares the algorithms P-con and P-min as the parameter *range* is varied, keeping d and m fixed at 50 and 20, respectively. The number of I/Os for P-min can be seen to initially increase rapidly and then reach a steady state. As the value of r increases, the number of distinct blocks on a disk increases until an overwhelming number of the blocks on a disk are unique. The steady state value above 2,000 illustrates the imperfect overlap and consequent loss

Fig. 9. Performance with varying buffer size ($r = 50$).

in parallelism by the P-min algorithm due to the limited buffer size.

The behavior of P-con in Fig. 4 is interesting. As r increases, it becomes less likely that the eviction choices for prefetch and demand-fetch of a block will be the same. Hence, P-con starts ignoring more and more prefetching opportunities, and the number of I/Os it requires grows. However, when the range gets sufficiently large compared to the number of references from a disk, the request sequence begins to resemble a read-seldom sequence. Many of the blocks in the buffer will not be referenced again, and all of these are candidates upon which both P-min and P-con can agree. Hence, P-con begins to avail of more prefetching opportunities, and the number of I/Os decreases until it matches that of P-min when the reference string becomes read-once.

Fig. 5 compares the number of I/Os performed by P-lru and P-min for the same data. Both P-lru and P-min are eager prefetching algorithms, and hence, differences in their performance will depend primarily on their caching behavior. Since P-lru makes its eviction decisions based on limited lookahead, its decisions are inferior to those of P-min, and its performance is consequently worse. As the reference string tends towards a read-seldom string, better caching provides diminishing returns, and consequently both algorithms converge.

Figs. 6 and 7 show the performance of the three algorithms as the number of disks varies, while keeping the number of blocks on each disk approximately constant. Two representative values of *range*, $r = 10,000$ and $r = 100$, were considered. For both values of *range*, the experiments clearly show the degradation in the performance of P-con as the number of disks increases. With these values of *range*, the number of I/Os performed by P-min is relatively constant, and hence, the ratio of the number of I/Os done by P-con to P-min increases (almost linearly in Figs. 6 and 7) with increasing numbers of disks. Interestingly, even with randomly distributed data, this reflects the worst case behavior predicted for this ratio in Theorem 3.

Figs. 8 and 9 show the variation in the performance of the three algorithms as the buffer size is varied for two values of *range*, $r = 5000$ and $r = 50$. These results illustrate that a buffer of moderate size is adequate in practice for achieving close to the best possible performance. As anticipated, P-lru performs somewhat worse than P-min over almost the entire range of m . However, its performance for randomly distributed data is far superior to its worst case performance. This is consistent with the behavior of sequential LRU, which performs well in practice despite having a poor competitive ratio. This observation, coupled with its scalability with d evidenced in Figs. 6 and 7, and its need

for only partial future information, suggests that P-lru is a practical algorithm in a distributed-buffer I/O organization.

Finally, observe that as the buffer size increases, P-con and P-min converge to the same values, whereas the I/Os for P-lru remain slightly higher than the optimal. As m increases, P-con improves its chances of finding a block to match P-min in its buffer and begins prefetching more aggressively. At that point, all three algorithms are prefetching greedily, but the limited lookahead of P-lru results in slightly worse performance due to its poorer eviction decisions.

7 DISCUSSION

The I/O performance of applications in multiple-disk systems can be improved by overlapping disk accesses. This requires the use of appropriate prefetching and buffer management algorithms that ensure the most useful blocks are accessed and retained in the buffer.

In this paper, we answered several fundamental questions on prefetching and buffer management for a distributed-buffer parallel I/O system. We derived and proved the optimality of a simple greedy prefetching algorithm P-min, that minimizes the number of parallel I/Os. P-min prefetches from a disk whenever the block to be prefetched is required earlier than some block in the buffer.

We analyzed a conservative prefetching algorithm, P-con, that sacrifices prefetching in order to make optimal caching decisions for each disk independently. It only prefetches when it can make the same eviction decisions as the optimal demand-paged sequential algorithm MIN. Both analytical and simulation results indicated that P-con has poor parallel performance that degrades in proportion to the number of disks. The results indicate that greater gains are possible by prefetching, compared to the benefit of making optimal caching decisions.

The importance of prefetching is also demonstrated in the behavior of the on-line algorithm P-lru analyzed in this paper. Because of its ignorance of accesses beyond the block to be prefetched, P-lru's eviction decisions are suboptimal than those of P-min. However, its performance relative to P-min is independent of the number of disks; it therefore gets all the benefits of increased parallelism that P-min gets with increasing the number of disks, while the penalty of suboptimal caching does not increase. Since in practice, LRU is a good caching algorithm with a single disk, the results imply that its parallel counterpart will have good scalable performance with increased I/O-system sizes.

ACKNOWLEDGMENTS

This research was partially supported by the National Science Foundation (NSF grants CCR-9704562, CCR-9303011, and CCR9732186) and by a grant from the Schlumberger Foundation. An earlier version of this paper appeared in the *Proceedings Conference on Foundations of Software Technology and Theoretical Computer Science*, 1996.

REFERENCES

- [1] S. Albers, "On the Influence of Lookahead in Competitive Paging Algorithms," *Algorithmica*, vol. 18, pp. 283–305, 1997.
- [2] R.D. Barve, M. Kallahalla, P.J. Varman, and J.S. Vitter, "Competitive Parallel Disk Prefetching and Buffer Management," *ACM Fifth Ann. Workshop I/O in Parallel and Distributed Systems*, pp. 47–56, 1997.
- [3] R.D. Barve, E.F. Grove, and J.S. Vitter, "Simple Randomized Mergesort on Parallel Disks," *Parallel Computing*, vol. 23, no. 4, pp. 601–631, 1997.
- [4] L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage," *IBM Systems J.*, vol. 5, no. 5, pp. 78–101, 1966.
- [5] P. Cao, E. Felten, A. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 188–197, 1995.
- [6] 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*, vol. 26, no. 2, pp. 145–185, 1994.
- [7] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson, "Cooperative Caching: Using Remote Client Memory to Improve File System Memory," *Proc. First Symp. Operating Systems Design and Implementation*, pp. 267–280, 1994.
- [8] D. Breslauer, "On Competitive On-line Paging with Lookahead," *Proc. 23rd Symp. Theoretical Aspects of Computer Science*, pp. 593–603, 1996.
- [9] J.M. del Rosario, A.N. Choudhary, "High Performance I/O for Massively Parallel Computers: Problems and Prospects," *Computer*, vol. 27, no. 3, pp. 59–68, 1994.
- [10] G. Gibson et al., "File-Server Scaling with Network-Attached Secure Disks," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, pp. 272–284, 1997.
- [11] M. Kallahalla, P. Varman, "Improving Parallel-Disk Buffer Management Using Randomized Writeback," *Proc. 1998 Int'l Conf. Parallel Processing*, pp. 270–277, 1998.
- [12] K. Keeton, D. Patterson, and J.M. Hellerstein, "A Case for Intelligent Disks (IDISKS)," *ACM SIGMOD Record*, vol. 27, no. 3, pp. 42–52, 1998.
- [13] T. Kimbrel and A.R. Karlin, "Near-Optimal Parallel Prefetching and Caching," *Proc. 37th IEEE Symp. Foundations of Computer Science*, pp. 540–549, Oct. 1996.
- [14] D.F. Kotz and C.S. Ellis, "Prefetching in File Systems for MIMD Multiprocessors," *IEEE Trans. Parallel and Distributed Computing*, vol. 1, no. 2, pp. 218–230, 1990.
- [15] G. Ma and A.L.N. Reddy, "An Evaluation of Storage Systems Based on Network-Attached Disks," *Proc. 1998 Int'l Conf. Parallel Processing*, pp. 278–285, 1998.
- [16] V.S. Pai, A.A. Schäffer, and P.J. Varman, "Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging," *Theoretical Computer Science*, vol. 12, pp. 211–239, 1994.
- [17] R.H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating System Principles*, pp. 79–95, 1995.
- [18] A.L.N. Reddy and P. Banerjee, "Design, Analysis, and Simulation of I/O Architectures for Hypercube Multiprocessors," *IEEE Trans. Parallel and Distributed Computing*, vol. 1, no. 2, pp. 140–151, 1990.
- [19] D. Sleator and R.E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Comm. ACM*, vol. 28, no. 2, pp. 202–208, 1985.
- [20] P. Varman, "Randomized Parallel Prefetching and Buffer Management," *Proc. Workshop Randomized Parallel Computing*, pp. 363–372, 1998.
- [21] J.S. Vitter and E.A.M. Shriver, "Optimal Algorithms For Parallel Memory 1: Two-Level Memories," *Algorithmica*, vol. 12, nos. 2–3, pp. 110–147, 1994.

Peter J. Varman received his BTech degree from the Indian Institute of Technology, Kanpur, and the MS and PhD degrees from the University of Texas at Austin, all in electrical engineering. He is currently on the faculty at Rice University. He has held visiting positions at IBM's T.J. Watson and Almaden Research Centers, and at Nanyang Technological University, Singapore. His research interests are in the areas of parallel computing, parallel I/O with

application to multimedia retrieval, database performance, and temporal and spatial databases. Dr. Varman is a senior member of the IEEE, as well as a member of the ACM and the New York Academy of Sciences.

Rakesh M. Verma received his BTech degree (with honors and a gold medal) in electronics engineering in 1984 from the Institute of Technology (Banaras Hindu University), Varanasi, India. He received the MS degree in 1985 and the PhD degree in 1989, both in computer science, from the State University of New York at Stony Brook, where he was a Catechismos fellow from 1988-1989. He is currently an associate professor in the Computer Science Department at the University of Houston. He was a visiting professor during the summer of 1995 and the academic year of 1996-1997 at the Center for Informatics Research (CRIN) in Nancy, France. His main research interests are in the areas of term rewriting and symbolic computation with applications to automated reasoning/verification, functional and equational programming, and computer algebra; storage and access structures for temporal and spatial databases with applications to scientific databases and geographic information systems; and parallel computing. Dr. Verma is a member of the ACM.