Prefetching in File Systems for MIMD Multiprocessors

DAVID F. KOTZ AND CARLA SCHLATTER ELLIS

Abstract—The problem of providing file I/O to parallel programs has been largely neglected in the development of multiprocessor systems. There are two essential elements of any file system design intended for a highly parallel environment: parallel I/O and effective caching schemes. This paper concentrates on the second aspect of file system design and, specifically, on the question of whether prefetching blocks of the file into the block cache can effectively reduce the overall execution time of a parallel computation, even under favorable assumptions.

Experiments have been conducted with an interleaved file system tested on the Butterfly Plus multiprocessor. Results of these experiments suggest that 1) the hit ratio, the accepted measure in traditional caching studies, may not be a adequate measure of performance when the workload consists of parallel computations and parallel file access patterns, 2) caching with prefetching can significantly improve the hit ratio and the average time to perform an I/O operation, and 3) an improvement in overall execution time has been observed in most cases. In spite of these gains, prefetching sometimes results in increased execution times (a negative result, given the optimistic nature of the study).

We explore why it is not trivial to translate savings on individual I/O requests into consistently better overall performance and identify the key problems that need to be addressed in order to improve the potential of prefetching techniques in this environment.

Index Terms—Concurrent operating systems, disk caching, file systems, MIMD multiprocessors, parallel I/O, prefetching, read-ahead.

I. INTRODUCTION

THE PROBLEM of providing file I/O to parallel programs has been largely neglected in the development of multiprocessor systems. This is a serious deficiency. Many problems that require the computational speed provided by parallel processing have massive data requirements as well. Examples include simulation of large VLSI circuits, simulations or analysis of physical processes based on sensor data (such as seismic data), and manipulation of large databases [20]. It is difficult to provide sufficient I/O bandwidth to keep highly parallel machines operating at full speed for very large problems. There are two essential elements of any file system design intended for a highly parallel environment: parallel I/O and effective caching schemes.

This paper concentrates on the second aspect, intelligent management of a file buffer or block cache, and, specifically, on the issue of prefetching in an MIMD environment which provides some form of parallel disk I/O. In traditional uniprocessor systems, caching and prefetching have been effective in eliminating some of the delays of I/O transfers and, thus, in speeding up the servicing of certain file system requests [44]. In a parallel environment, both the opportunities for prefetching and the management problems associated with it may be richer because it is the collective behavior of the processes in a parallel computation that often determines the performance rather than the speed of an individual thread.

The goal of this paper is to investigate whether prefetching can effectively reduce the overall execution time of a parallel computation, even under favorable assumptions, and to begin to provide guidelines for the development of prefetching policies for more realistic situations. Our approach is experimental. We have developed a file system tested called RAPID Transit (Read Ahead for Parallel Independent Disks) that runs on the Butterfly Plus multiprocessor [7] and allows implementations of various buffering and prefetching techniques to be evaluated.

We describe some related work in the next section, and then go on to discuss some of the issues that distinguish buffering and prefetching strategies in MIMD parallel systems from those in traditional systems. In Section IV, we outline the experiments performed with the testbed including the expected patterns of parallel file access embedded in the workload model. Section V describes some of the results obtained. These results show that 1) the hit ratio, the accepted measure in traditional caching studies, is not by itself an adequate measure of performance when the workload consists of parallel computations and parallel file access patterns, 2) caching with prefetching does significantly improve the hit ratio and the average time to perform an I/O operation, and 3) an improvement in overall execution time has been observed in most cases. In spite of these gains, prefetching sometimes results in increased execution times (a negative result, given the optimistic nature of the study). Finally, we conclude by identifying key problems that need to be addressed in order to improve the potential of prefetching techniques in the MIMD parallel environment.

II. RELATED WORK

A study by Boral and Dewitt [8] argues that I/O bandwidth is a crucial bottleneck for database processing and that a significant increase in bandwidth is necessary if parallel database machines are to be effective. They propose two solutions: interleaving disks to provide hardware parallelism using primarily conventional technology and/or providing a very large file cache to retain blocks that will be reused and to hold blocks that are prefetched based on knowledge of future
access patterns. Other researchers [44], [33], [30] have emphasized the increasing gap in memory and disk access times and have predicted large bottlenecks at the I/O system for high-performance computers. They proposed caching and prefetching [44] and parallel disk hardware [33] as near-term solutions.

A. Hardware Parallelism

By hardware parallelism we mean multiple disk drives and possibly multiple disk controllers and channels. Traditional systems may use multiple disks for reasons of size, speed, or reliability, but they rarely spread a single file over multiple disks to parallelize access to that file. Most of the previous work in I/O hardware parallelism has involved interleaving, or disk striping. In this technique, a file is horizontally partitioned ("interleaved" or "striped") across numerous disks and accessed in parallel to simultaneously obtain many blocks of the file with the positioning overhead of one block. The issues of I/O parallelism and disk striping have been extensively studied [39], [22], [25], [16], [50], [15], [34], [27], [26], [38], [32], [37], [33].

One possible parallel disk architecture is based on the notion of parallel, independent disks, using multiple conventional disk devices addressed independently and attached to separate processors. This arrangement is different than that in a distributed system since a file may be spread over several disks, and thus the disks are more logically related. It is also different from most striping proposals since the disks are connected to separate processors through separate channels, avoiding the bottleneck associated with many striping or disk array designs. Intel has developed such a system for their iPSC/2 multiprocessor [18], [19], [34], [1]. Several researchers have proposed architectures for other hypercube-connected multiprocessors [15], [50], [38].

We have gained considerable experience with these techniques through the implementation and use of a prototype file system called Bridge [12] that runs on a BBN Butterfly parallel processor [4]. Bridge is based on interleaved files in which consecutive logical blocks are assigned to devices on different processor nodes in round-robin fashion and can be accessed in parallel to provide increased I/O bandwidth. Gamma [10], [11] is a database machine architecture that allows the tuples of relations to be interleaved across disks in a similar manner. Several commercial multiprocessors are beginning to include some form of parallel disk architecture [27].

B. Caching and Prefetching

Disk caching is an accepted way of improving disk performance in uniprocessor systems [44], [43], [31], [30], [28]. Increasing main memory sizes allow for large disk caches that can significantly reduce traffic to the disks by satisfying many of the requests from the cache. We can improve the hit rate of the cache by reading blocks into the cache before they are requested, which is called prefetching.

Previous research in prefetching has centered in three areas, and all for uniprocessors: prefetching in disk caches, prefetching in main memory caches [40], [43], [24], and prepagig in virtual memory systems [49], [48]. For disk caches, it is also usually assumed that prefetching is done on a low level, reading physically contiguous blocks from the device. This depends on files being allocated contiguously on disk since any sequentiality in the access pattern is fundamentally among the logical blocks of the file.

The most common technique for prefetching in both main-memory and disk caches is one-block lookahead (OBL) [44], [43], [40], which prefetches block i + 1 when block i is accessed. This technique has been successful in reducing the miss ratio in disk caches by up to 80 percent [44]. Another technique [41] combines the observed and expected sequentiality to decide how much to prefetch. More complex methods are described in [5], [36], [2]. Towsley [49], [46] has studied disk cache prefetching for multiprocessors, although his study is oriented toward using multiprocessors to support large numbers of independent processes, not cooperating processes.

C. File Access Patterns

In order to effectively prefetch blocks of a file, it is necessary to predict the set of blocks that will be used in the near future, and this requires an understanding of the patterns of access to files in parallel computers. File access patterns have been studied extensively for uniprocessors [13], [29], [14], [17]. Floyd [13] studied file access patterns in a Unix system, and found that 68 percent of files opened for reading are completely read, strongly suggesting sequential access. Surveys of IBM systems (ibm, summarized in [42] and [36] and Cray systems [35] have concluded that most files are used sequentially. Crockett [9] also expects sequential file access patterns to be important for parallel computers.

Supercomputer I/O is characterized by sequential access and a need for minimum latency and low throughput [21]. The sequential access characteristic makes it a good canidate for prefetching. Prefetching is especially valuable for supercomputers since the level of multiprogramming often cannot be increased to cover the idle time caused by I/O delays as in traditional multiprogrammed uniprocessor systems. This is evident in the traditional operating systems used with many multiprocessors, including the Chrysalis™ operating system for the BBN Butterfly Parallel Computer [4], in which each processor usually has only one user process running on it.

III. PREFETCHING ISSUES

A prefetch strategy must consider when to prefetch, what blocks to prefetch, which processors should make the prefetching decisions, and where the data should be stored. However, it must first be determined whether it is worthwhile to develop prefetching strategies for the parallel environment.

In a uniprocessor system with a workload consisting of single thread computations, every disk operation that can be replaced by a hit in the block cache translates directly into a time savings (assuming that any extra overhead introduced by buffer management is insignificant). Thus, an accepted and appropriate metric for evaluating caching and prefetching strategies is the hit ratio (i.e., the fraction of access requests satisfied out of a buffer).

The nature of parallel multiprocess computations changes the situation significantly. As the following example (Fig. 1)
that are never needed by the program) represent a major source of overhead. The file system may decide what to prefetch based on information from a variety of sources. The process may give it explicit advice regarding the expected future access pattern. The programmer may provide a static hint, describing the type of pattern to expect. Finally, it may be given no explicit information and be forced to use the observed access pattern of the process to predict future access patterns. In the current set of experiments, designed to answer these questions of whether and under what conditions prefetching can be effective (described in more detail in Section IV), we defer consideration of on-the-fly prediction algorithms for choosing which blocks to be prefetched. Instead, we provide, in advance, accurate information on the reference pattern which can be used to establish an upper bound on the performance benefits of prefetching for our system and workload models.

In typical uniprocessor file systems, access pattern predictions have not been difficult. Prefetching has been optimized for straightforward sequential access patterns; when servicing a read request for block \( i \), the system can perform read-ahead for the following block \( (i + 1) \) so that it may be in the cache for the next expected request. When there is only a single process issuing requests, the time until the actual request for block \( i + 1 \) is likely to be adequate for the prefetched data to arrive. However, it is unclear whether a naive extension of this strategy to the parallel environment can be successful. Consider a parallel file access pattern in which many processes collectively read the file in a sequential manner, but the pattern is not obvious unless viewed from a global perspective. If the file system uses the policy of prefetching the next block in the projected global reference string that has not yet been requested, it is likely that some process will almost immediately issue a request for that block. Thus, regardless of whether it is the read-ahead request or the on-demand request that actually accomplishes the disk access, the requesting process waits for the I/O to complete anyway and derives little benefit from read-ahead. Prefetching may provide only a small head-start on the I/O request. This explains why we described our assumption above as a generous one; a buffer hit (i.e., finding a buffer reserved for the desired block regardless of whether the data are there yet) may not necessarily translate into real savings. These observations suggest that in a parallel computation in which several blocks are being processed simultaneously there may be other, more useful definitions of the "next" blocks to consider. Blocks needed further in the future may seem to be better candidates for prefetching, leaving the blocks very near the active region of the reference string to be brought in by demand fetch. This intuitive discussion motivates our investigation of these implications of parallel I/O requests and the kinds of tradeoffs involved.

The nature of the parallel environment also makes the prefetch timing an interesting issue. In a uniprocessor, multiprogramming is often used to take advantage of the time that the processor would otherwise spend idle while a given process is blocked, perhaps waiting for I/O. Thus, multiprogramming achieves an overlap of computation and I/O. In many medium to large scale multiprocessors, multiprogramming of
user processes on the individual processors is either discouraged or not supported. The computations required for the file system to do prefetching (e.g., determining the block to be prefetched, allocating buffers) can be scheduled whenever the user process experiences idle time such as when it is waiting for other processes to arrive at a synchronization point or waiting for an I/O request that requires a disk operation. Unless the file system runs on a dedicated processor node, prefetching work done outside of these natural idle times is system overhead competing for processor cycles with user processes involved in the parallel computation. This may perturb the execution sequences of the user computation, including delaying the exit from the next synchronization point for all threads. Thus, one goal may be to overlap prefetching decision-making time with user process I/O or synchronization time as much as possible to minimize the impact on the overall execution sequence of the application. It is not clear whether restricting prefetching opportunities to just these points can achieve significant enough benefits to cross a performance threshold. It is also possible that this approach tends to cluster prefetch requests (in time) so that they experience more contention for the disks with other prefetches or with concurrent demand fetches. We intend to examine these questions.

IV. Experiments

The experiments described in this paper are designed to determine whether prefetching can be effective in improving the overall execution time of parallel programs. We are also interested in determining the effect of prefetching on individual I/O requests (e.g., whether the expected response time for a read operation can be reduced). The design of buffering and prefetching techniques depends on assumptions made about the underlying multiprocessor model, the physical disk subsystem, and the interface to the file system. Our testbed reflects only one set of design decisions chosen from a variety of possible alternatives. Our results must be interpreted within this context. A model of the expected file access patterns is also important in designing and evaluating prefetching strategies. The workload presented to the testbed covers a range of potential access patterns based on speculation about how parallel programs will evolve to use files. Thus, another objective of the study is to investigate the impact of different patterns of access (possibly identifying some styles that should be avoided or encouraged). On-going experiments are attempting to focus more deeply on substantiating cause-and-effect relationships for some of the performance trends suggested in these data and on understanding particular interesting interprocess interactions.

A. Testbed

We have built a fully parallel file system testbed, the RAPID Transit system, that was derived from the BBN RAMFile system [3] and stresses caching on a BBN Butterfly Plus multiprocessor.

The architectural model for our testbed can be described as a medium to large scale MIMD shared memory multiproces-

1 The concrete example used in our experiments, the BBN Butterfly Plus, is classified as a nonuniform memory access time (NUMA) architecture. In a NUMA machine, the placement of data such as file system data structures and buffers relative to the origin of requests can have a significant impact on performance.
RU set. This offers strong locality for the more complex list manipulations while enforing a global policy.

Another question is whether file system software is multi-programmed on the same processors with the application program, stealing cycles and affecting the execution sequence of the computation, or whether it runs on dedicated I/O processors. In our current multiprogrammed model, there are components of the file system on each processor handling all the I/O requests initiated locally and managing disk operations for any disks connected to that processor node. Prefetching operations involve some extra computation to determine the blocks to be fetched, which may compete with the computation for cycles and ideally should occur during application idle time, and the actual I/O, which can overlap application computation time. We attempt to exploit three types of idle times: while the user process is waiting for other processes at a synchronization point, while it is waiting for self-initiated disk I/O, and while waiting for disk I/O initiated elsewhere (i.e., an apparent buffer hit whose data have not yet arrived). As long as the user process remains in the idle state, the file system repeatedly considers prefetching, releasing control only at the completion of an action.²

B. Workload

We assume there exists a single parallel computation executing within its own set of dedicated processor nodes with one user process per node. For the kinds of very large-scale programs in our target workload, this has been the preferred utilization of hardware resources and is supported by the cluster constructs in the Butterfly operating system (both Chrysalis and Mach 1000). In such parallel programming systems, a file can be accessed, among other computations, any file reference locality that can be exploited by buffering and prefetching policies must be generated by related cooperating processes from the same computation. This justifies examining access patterns on a per-computation per-file basis rather than system-wide. This study also restricts its attention to read-only files, for the present time.

Although file access patterns in uniprocessor and distributed systems are well-understood [13, 29, 14], it is not clear that similar patterns will be found in parallel environments. The ideal approach involves performing similar studies in multiprocessor environments to collect file usage data from parallel applications. Unfortunately, it is premature to do this because the inadequate attention given to multiprocessor file systems until very recently has discouraged typical parallel access patterns from developing. Based on experience, we have identified several possible forms of parallel file access. File access patterns naturally break down into two categories: random and sequential. Sequential file access has always been important for file prefetching. In a parallel environment, sequential file access may be considered in several new ways (see Fig. 2). Random patterns are defined to be those that do not fit into any of the sequential forms.

All sequential patterns consist of a sequence of accesses to sequential portions (runs in [41]). A portion is some number of consecutive blocks in the file. Note that the whole file may be considered one large portion. The accesses to this portion may be sequential when viewed from a local perspective, in which a single process accesses successive blocks of the portion. A degenerate case of a local sequential pattern, which is actually common in current practice, is for one processor to handle all of the data with no participation from other processors. This is the traditional notion of sequential access used in uniprocessor file systems. Alternatively, the pattern of accesses may only look sequential from a global perspective, in which many processes share access to the portion, possibly reading disjoint blocks of the portion. If the reference string of all the processes is merged with respect to time, the accesses follow a (roughly) sequential pattern. The pattern may not be strictly sequential due to the variations in the global ordering of the accesses; it is this variation that will make global patterns more difficult to detect when we begin to consider on-the-fly techniques.

The sets of data accessed by the individual processors may or may not intersect with each other; any overlap has implications for the buffer replacement algorithm, as the blocks in common will each be used more than once. This leads to another breakdown into overlapped and disjoint patterns. In addition, the length of sequential portions may be regular, so that it is possible to predict the end of a portion and not prefetch beyond it. The difference between the last block of one portion and the first block of the next portion may also be regular (a stride), allowing the system to prefetch into the next portion. In the chart, we refer to these patterns as regular sequential portions and others as irregular.

Our test applications are simple synthetic programs each designed to drive a particular access pattern on a single file. Each class of file access pattern has a corresponding prefetching algorithm provided for it in the testbed. Prefetching strategies based on knowledge of the forthcoming block references are employed for this study. The information supplied is not a perfect reference string (i.e., an exact ordered list of the blocks, ordered by time of request) since the parallel execution sequences of the application program determine the ultimate ordering of requests. These strategies are optimistic in the sense that they always choose a block that will be needed.

² A prefetching action is defined as the attempt (either successful or unsuccessful) to prefetch one block. This involves selecting a block and locating a buffer. It may initiate I/O, but does not wait for it to complete.
in the near future and never make mistakes. This is tempered by the fact that, even though the reference information is available, prefetching is not done in situations where the information could not feasibly be derived. In addition, no explicit attempts are currently being made to balance the caching benefits over processors. However, in some cases processors do prefetch blocks that will be used exclusively by other processors.

Thus, there are six representative parallel file access patterns embedded in our synthetic applications. They correspond to five of the leaves of the taxonomy tree in Fig. 2 with the sixth pattern being a potentially important special case. The random patterns are not amenable to prefetching and, thus, have not been included in the workload. The local sequential pattern with disjoint and irregular portions has also been excluded because it offers similar, but less interesting information than the corresponding overlapped pattern. Specifically, the patterns used and the associated prefetching policies are described as follows:

**Ifp**—local fixed length portions: in this local sequential pattern, the sequential portions are of regular length and spacing (although at different places in the file for each process). Since an on-the-fly local procedure could easily predict future access following this pattern after a short amount of observation, our prefetching algorithm allows prefetching into succeeding portions. The encoding of the reference string for this and all local patterns is a set of strings, one per processor.

**lrp**—local random portions: this local sequential pattern uses portions of irregular (random) length and spacing. Portions may overlap by coincidence. It is reasonable to expect prefetching success within but not between sequential portions. Thus, our prefetching algorithm does not prefetch past the end of the portion until a demand fetch has established the location of the next sequential portion (i.e., it chooses not to use information it has in its supplied reference string that would be difficult to predict).

**lw**—local whole file: in this local sequential pattern, every process reads the entire file from beginning to end. It is a special case of the overlapped local sequential pattern with a single, fully overlapped portion.

**gf**—global fixed portions: (analogous to Ifp) in this pattern, processors cooperate to read what appears globally to be sequential portions of fixed length and spacing. In this and all of the global patterns, a single global reference string is used to represent the access pattern. The prefetch algorithm may prefetch into subsequent portions.

**gr**—global random portions: (analogous to lrp) processors cooperate to globally read sequential portions with random length and spacing. The prefetch algorithm does not proceed ahead to successive portions.

**gw**—global whole file: this global pattern reads the entire file from beginning to end, the processors reading distinct blocks from the file, so that globally the entire file is read exactly once, but locally each processor only reads some small subset of the file with no discernible portions.

Crockett [9] has proposed another categorization of potential parallel file access patterns as they relate to possible storage techniques. Our definitions capture the characteristics that pertain to buffer management and prefetching strategies. Certainly, there are variations or combinations of the pure access patterns described above. For example, it is possible that some subset of processors is generating one access pattern while another subset is using a different pattern. We do not expect these hybrid patterns to be very important. This is based on an informal survey of existing application programs which have been designed for our Butterfly Plus and on discussions with the authors about the parallel file access patterns they project may be useful in those applications.

In addition, there are a number of different synchronization styles that may arise in a parallel computation and the test applications simulate a few of them. The approaches we have studied create synchronization intervals that are related to the processing of a specified quantity of data. The following four types of synchronization points are considered: the processors may all synchronize after reading a fixed number of blocks per processor, after reading a fixed number of blocks total, after each sequential portion (whether local or global), or none at all.

The suite of test runs consists of a uniform mix of the six file access patterns, the four synchronization styles, and two levels of I/O intensity. This implies that no attempt has been made to select the programs which generate “typical” program behavior and weight them more heavily. The results we report obviously depend on how well this mix captures the spectrum of real applications.

**C. Measures**

The performance of the system is measured both with and without prefetching and the exact access pattern is recorded for off-line analysis of prefetching strategies.

The primary metric for measuring the performance of the application is the overall completion time since this is what prefetching is attempting to reduce in our workload model. The average time to read a block is also recorded, as this should improve as well. The average effective disk access time (e.g., the time from enqueuing a disk request to completion of the request) serves as a measure of disk contention. Also recorded are the number of blocks prefetched and the number demand fetched, providing the cache hit ratio (indicating the level of prefetching achieved).

For each of the three types of idle time mentioned in Section III, the length of the logically necessary idle period is recorded, as well as the time to actual resumption of user computation. The length of each individual prefetching action is also taken. The overrun of prefetching activity, extending past the point at which the user process is again able to continue, is calculated as one measure of the cost to the user program of doing prefetching.

**D. Experimental Parameters**

All tests were run with 20 processors, each running the same application with the same set of parameters. They used a file containing 2000 1024-byte blocks, requesting precisely
one block at a time. The file was interleaved over 20 disks. The total number of blocks read was always 2000. The processes synchronized after reading 10 blocks per processor, after 200 total (i.e., about 10 each), after 2000 total (i.e., no synchronization), or after each portion. The type of synchronization was the same across all processors. After each block was read, delay was added to simulate computation. This delay was exponentially distributed with a mean of 30 ms, except in the lw pattern, which used a mean of 10 ms. These numbers were used to attain a nearly even balance between the amount of time spent in I/O and in computation. The high degree of interprocess locality seen in lw contributes to a high hit ratio and, thus, a lower average block read time in the nonprefetching base case. With less time spent in I/O, less computation time is needed to balance the I/O time. Data from another set of experiments, with no additional computation simulated for each block, are included in the results below.

The local RU-set size of each processor was one block, emulating a variation of a “toss-immediately” strategy [45] within the context of our replacement algorithm. In other words, the cache size was 20 blocks. The cache size of 20 was adequate to accommodate any interprocess locality present within these sequential access patterns (e.g., from overlapped portions) that would require reuse of a block. When prefetching was active, three additional buffers per processor node were allocated to be used only for prefetching, bringing the cache size to 80 blocks. The global number of prefetched but not yet used blocks was, however, limited to a total corresponding to three per processor (i.e., 60 blocks, in this case).

V. RESULTS

The behavior of prefetching can be examined through a variety of performance criteria including reduced average block read time, improved cache hit ratio, and reduced overall execution time. Although the final goal may be the ultimate measure of prefetching success, the others are important to consider. In fact, the average block read time and the cache miss ratio are the measures most commonly used in the literature to evaluate the performance of prefetching and buffering techniques. We begin by examining the effects of prefetching on the average read time and cache miss ratio.

A. Average Block Read Time and Cache Miss Ratio

The average block read time is the average time necessary to read a block from the file. If the block is in the cache, the read time is much lower than when a disk operation is necessary. Prefetching hopes to fill the cache with the right blocks, so that more read requests may be satisfied quickly. A lower average block read time is one measure of the success of this attempt. As shown in Fig. 3, the average block read time is reduced through the efforts of prefetching in all cases we have studied. In this figure, the read time under prefetching for a given set of parameters is plotted against the read time without prefetching for the same set of parameters. The line \( y = x \) is plotted as a reference. This scatter plot presentation is interesting because it illustrates the change in the read time while showing the actual times (which can be compared with the 30 ms physical disk access time). Since all of the points lie under this line, the average read time for each instance has been reduced. The improvement in the average read time exceeded 35 percent for 60 percent of the experiments, had a median of 48 percent, and reached as high as 88 percent.

The hit ratio (shown as cumulative distributions in Fig. 4, with points plotted as "P," for prefetching cases, and "N," for those with no prefetching) was greatly improved in all prefetching experiments. In all cases, it was over 0.69 and, in more than half of them, it was over 0.86. Due to the sequential nature of the accesses, most of the corresponding experiments without prefetching had a hit ratio of nearly zero. Those cases with a significant hit ratio in the absence of prefetching had some degree of interprocess locality, as in the lw pattern, making caching alone ineffective.

A strong improvement in the hit ratio is not necessarily sufficient to significantly lower the average read time. One reason is that a block that is found to be in the cache (perhaps due to a prefetch action or the demand request of another process) may still have a large proportion of its I/O time outstanding. The requesting process must still wait for the I/O to complete. We call this the hit-wait time. The hits with a positive hit-wait time are unready buffer hits whereas ready buffer hits have a zero hit-wait time. Fig. 5 shows, for the

---

3 Portion synchronization was not used with the lw pattern since it could not be fairly compared with the other patterns. The variable length of the portions meant that each process had a different number of portions for the same number of reads, or a different number of reads for the same number of portions.
prefetching cases, the fraction of accesses serviced by unready hits (plotted as “U”) and ready hits (plotted as “R”). Unready hits make up a significant portion of the hits in our prefetching experiments. These unready buffer hits may be construed as misses, but they do not have the same impact as true misses. With our model of full interleaving over all nodes, the average hit-wait times are relatively low, with 70 percent of the values less than 6 ms and all points less than 17 ms. Although a small hit ratio does virtually eliminate the effect of hit-wait time, a high hit ratio can be accompanied by either a large or small average hit-wait time depending primarily on characteristics of the benchmark program (e.g., access pattern and computational intensity). Our results show that the hit-wait time is smaller than the time required for a miss but is large enough (when combined with a high number of unready hits) to be a contributing factor in the time required to read a block. While there seems to be a fuzzy relationship between the average block read time and the hit-wait time (shown in Fig. 6), no obvious relationship has been found between the read time and the hit ratio measure over the full range of experiments.

One factor contributing to the average block read time and hit-wait times is contention for the disks. The average disk response time (defined as the time from the entry of the request on the queue of the appropriate disk to the completion of the I/O) is an indicator of disk contention. The disk response time for a block may be larger than the physical disk latency when contention for the disks forces disk requests to be queued for service. The disk response time slows as contention for the disks increases. Prefetching increases the contention for the disks as it fills the queues with read requests, and thus the disk response time worsens, as shown in Fig. 7.

Note that the disks serve no more requests under prefetching than they did without prefetching, since the prefetching strategy in our experiments fetches no unnecessary blocks. The extra disk load arises from the same number of requests issued in a smaller amount of time. This is clear when the total execution time is reduced. Even when the total time is not reduced, the disk reads tend to be issued nonuniformly in time, creating periods of high disk contention. In general, an experiment that has high disk utilization even without prefetching will experience sharp increases in the disk response time as prefetching fills the disk queues. This effect can be seen for several points in Fig. 7. The accompanying improvement in the hit ratio, however, still allows for a reduction in the average block read time.

B. Effect on the Total Execution Time

Our primary measure of prefetching effectiveness is the total execution time of the program. We have found that prefetching reduces the total execution time, often significantly, for most of the cases we studied (see Fig. 8). The biggest improvements, up to 69 percent, appear in the bw pattern where all 20 processes can benefit from each prefetched block.

Occasionally, prefetching can increase the execution time. Three of the Ifp pattern experiments slow down as much as 15 percent, despite solid improvements in the hit ratio and the average block read time. This is due to an uneven distribution of the benefits of prefetching, as outlined in Fig. 1(b). In local patterns, including Ifp, the processes prefetch only for themselves. Thus, any prefetching a process completes will later benefit that process in the form of hits and shorter read times. It appears that, due to subtle timing issues, some processes grab several buffers and prefetch for themselves, leaving few buffers for other processes. Their time is improved, but they end up waiting at the next synchronization point for the less fortunate processes. Those other processes, in their own attempts to do prefetching (often unsuccessful due to the lack of free buffers), waste some time in prefetching.
overruns, *lengthening* the interval and hence the total execution time. \(^4\)

The *l rp* pattern, which had small improvements in the total execution time, does not exhibit this effect as strongly. Recall that with random portions we restrict the prefetching of any one process to its current portion, which is usually short. Thus, it is difficult for one process to use very many prefetch buffers.

One factor affecting the total execution time is the impact of prefetching on average synchronization times (the time between arrival of a process at a synchronization point and the moment all processes achieve synchrony) which may increase as some of the savings on I/O operations are converted into longer synchronization waits. Fig. 9 demonstrates that prefetching usually increases average synchronization time, in a few cases quite dramatically.

Without some way to distribute the primary benefit of prefetching (a lower block read time) among the processes, any reduction in the *average* block read time does not necessarily translate into a reduction in the total execution time. Fig. 10 plots the reductions measured in our experiments, demonstrating at best only a fuzzy relationship. Fig. 11 plots the reduction in total execution time against hit ratio.

For these experiments, neither the hit ratio nor the average block read time appear to be very strong predictors of overall success. Nonetheless, significant improvements in all three measures (read time, hit ratio, and execution time) can be obtained with prefetching.

**C. The Balance between Computation and I/O**

The preceding results have looked at the data for all data points, from all of the parameter settings we used in our experiments. It is clear that many of the programs are I/O-bound, as evidenced by the high disk response times.\(^5\) These particular runs simply read one block after another with no time spent processing each block and represent one endpoint of the workload spectrum. When the processors devote all of their time to I/O, there is an increased likelihood that they will contend for access to the disks and internal data structures. To simulate programs with some computation, we added artificial computation time associated with each block fetched in many of the runs, as described in Section IV-D. These runs are included in the preceding figures, but it is valuable to examine the effect of this variable separately.

To more closely examine the effects of computation on prefetching, we chose one access pattern and one synchronization style and varied the parameter controlling the computation time performed for each block over a wide range of values.

---

\(^4\) Recall that overrun is the continuation of prefetching activity after the user process is again ready to run.

\(^5\) Remember there are as many disks as there are processes and that the physical access time is fixed at 30 ms.
The data discussed in this section are from runs all using the global whole-file access pattern (gw) synchronizing every time each processor finishes reading ten blocks. The computation time is exponentially distributed about a given mean. The idea is to study the effects of prefetching on various measures as the program changes from I/O bound to compute-bound. The results obtained were as might be expected.

As the character of the program switches from I/O to computation, prefetching benefits from overlapping the I/O and the computation. Indeed, in Fig. 12 we can see that the total execution time improves more when the program spends some time in computation, but this tails off as the bulk of the program’s time is spent in computation (which does not improve with prefetching) and the effect of the I/O time improvement becomes less significant. This is due to a increasingly large reduction in the average block read time, which reaches 20 percent of its value without prefetching.

One reason the average block read time improves is the reduced response time of the disk. When the processes are I/O bound, there is a great deal of contention for the disks and internal data structures. This contention decreases steadily as more time is spent processing each block, although the disk response time when prefetching remains higher than the response time when not prefetching.

Keeping processors busy with computation (and thus away from activity in the I/O subsystem) reduces contention for the internal data structures as well. Thus, prefetching actions may do their work more quickly, reducing the average prefetch time from 22 to 5 ms, with a corresponding reduction in the overrun time. These reductions directly benefit the average block read time by reducing the overhead of prefetching activity.

The generality of these results, which are based on a single access pattern, is supported by looking at the difference between the I/O bound (no computation time with each block read) and balanced (some computation time on each block) runs in our full set of experiments. Indeed, the average block read time is always increased significantly when the process is I/O bound rather than balanced. Correspondingly, the total execution time generally improves less with prefetching in the I/O bound processes than in their balanced counterparts.

D. Prefetching Overhead

The file system component that initiates prefetch actions whenever an application process enters an idle state does not release control until it has completed an operation. A prefetch action can be quite complex, requiring several milliseconds (averaging between 3 and 31 ms) to complete without including actual I/O. The overrun time can also be high (averaging between 1 and 25 ms). It is crucial to carefully streamline the implementation of the prefetching daemon, making every effort to reduce the amount of time necessary to initiate a prefetch.

In our initial implementation, we found the prefetching overhead to be very high. It was necessary to optimize the paths through the I/O subsystem, both for prefetch actions and demand fetches. Much of the activity involved in prefetching required several accesses to data structures in (slower) remote shared memory. Data structures were replicated where possible to reduce the number of remote memory references and the amount of memory contention, and local pointers to remote data structures were maintained for fast access.

In an attempt to control overrun time, we modified the prefetching strategy so that the file system would not start a new prefetch action unless it had a certain amount of the estimated idle time remaining (called the minimum prefetch time). Experiments varying the minimum prefetch time parameter resulted in lowering the prefetch overrun time, but only negligibly improved the total execution time and the block read time, due to a steady degradation in the hit ratio. Thus, this proved to be an unproductive idea.

E. An Attempt to Improve Hit-Wait Time

Although a low cache miss ratio may be important to lower the block read time, we found that a low hit-wait time was also an important component since many of the prefetched blocks hit in the cache have not yet completed their I/O, forcing the process to wait until the block is ready. It seems that the blocks that are prefetched are used too soon after the prefetch has been initiated. A possible improvement to the strategy is to avoid prefetching blocks that are to be used soon, and to instead prefetch well ahead of the current activity in the file, that is, to have the prefetch activity “lead” the demand-fetch activity by some distance. We tried this strategy with a varying amount of minimum prefetch lead for several patterns. The prefetch algorithm would not select blocks that were less far ahead (in the reference string) of the current demand-fetch activity than the minimum prefetch lead (except near the end of the file, when the restriction was relaxed). This approach can have little positive effect in the access patterns with random portion lengths since those patterns did not permit prefetching beyond the current portion, so we studied the effect only on the local and global fixed-portion (lfp and gfp) and the local and global whole-file (lw and gw) access patterns.

In the rest of our experiments with local patterns, each process reads 100 blocks for a total of 2000, for comparison with the global patterns that cooperate to read 2000 blocks. In this experiment, however, we use minimum prefetching leads of between zero and 90 blocks, which is too large when the entire pattern is only 100 blocks, as in the local patterns. Thus,
for these experiments the local patterns each read 2000 blocks for a total of 40,000. The total execution time is reduced by a factor of 20 for direct comparison with the global patterns, but the other measures we use may be examined unchanged.

Although we examine minimum prefetch leads of up to 90 blocks, the asymptotic effects are not necessarily more important than the effects for small leads. A lead of about 20 blocks (one per processor) may be the optimum size. The following graphs should be examined with this in mind.

As hoped, the hit-wait time was reduced considerably by increasing the minimum lead, except for the lw pattern, whose hit-wait time actually increased (see Fig. 13). The effect on lw is severe because each block is read by all processes and is therefore hit by nearly all processes. Normally one process reads the first block while the other 19 processors prefetch the next 19 or more blocks. After processing the first block the process is repeated. All blocks but the first are prefetched, and prefetching stays far enough ahead to keep the hit-wait time low except in the first few blocks. With a prefetch lead, the processes cannot prefetch some of the earlier blocks, requiring all of them to wait nearly the entire demand-fetch time for those blocks. Thus, the effects of even one lost prefetch opportunity are magnified 20 times.

In sharp contrast to any improvements in the hit-wait time, the cache miss ratio has climbed drastically for the global patterns to nearly 0.80 (Fig. 14). The miss ratio for lfp rises slowly to nearly the same level. Although lw appears to be little affected, the change in its miss ratio (from 1 miss to 1556 misses out of 2000 possible misses) is actually quite significant because every block is used by every process (as discussed above). We will see that its average block read time and total execution time more than double.

Unfortunately, the significant increase in the miss ratio diminishes the effect of the improvements in the hit-wait time on the average block read time. Indeed, we see increases in the average block read time for lw and gw in Fig. 15, and slight improvements for gfp and lfp for small values of the minimum prefetch lead. It is therefore no surprise that gw and lw slow down overall, as shown in Fig. 16. The gfp pattern also slows down, due to its severely increased miss ratio, while the total time for lfp is improved, as its miss ratio was less affected. Indeed, the total execution time for lfp with leads of 30 or more is less than the nonprefetching time. The net result, however, is that no satisfying improvements are obtained for all patterns by using even small minimum prefetch leads.

**F. Other Findings**

Many other experiments have been and continue to be performed, attempting to improve prefetching performance and exploring factors that may contribute to the results presented. We summarize a number of these below; details can be found in [23].

**The Number of Buffers:** Experiments that used only one buffer per process for prefetching show that smaller improvements are obtained for all patterns. It appears that one, two, or three buffers per process will suffice. While the determination
of the number of buffers required for the best performance is an area of continuing research, preliminary data suggest the choice (in the range 2–5 buffers per process) will have a minor impact on total execution time improvement.

Differences Among the Patterns: For brevity, many of the preceding discussions have made no distinction between the data points based on the access pattern. A breakdown of results by pattern, while beyond the scope of this paper, can provide programmers with guidance in making design decisions pertaining to file I/O. The different patterns capture very different notions of interprocess locality. The lW pattern exhibits strong interprocess temporal locality (as well as intraprocess spatial locality). We have already mentioned that the best data points belong to this pattern. The global patterns exhibit interprocess spatial locality. The other two local patterns exploit primarily intraprocess locality and, as a consequence, have lower hit ratios. As mentioned previously, processes using these patterns prefetch blocks only for themselves, reducing many of the potential benefits of prefetching. Subsequently, the lrp and lfp patterns show the least improvements with prefetching.

VI. CONCLUSION

MIMD multiprocessor architectures with parallel I/O capabilities have a profound impact on the nature of computations they support. The assumptions about file access patterns that underlie much of the work in uniprocessor file management are called into question. The traditional measures and methods used in studying the performance of file management strategies are not necessarily appropriate given the dynamics of parallel executions.

In this paper, we have begun to explore some of the issues involved in the prefetching of file data in multiprocessor environments. We have developed a useful tool for experimenting with prefetching strategies. Experiments based upon our RAPID Transit file system tested, while simulating disk I/O and being driven by synthetic programs, realistically capture many of the interactions within a parallel machine: the parallelism (both of the test programs and the file system implementation), the impact of the memory architecture, and the contention are real. While it is important to reduce the impact of file I/O on parallel computations, we are discovering that it is not always trivial to translate savings on individual I/O operations into gains in overall performance. However, there are many positive results and the negative results provide valuable insights.

Our results show that prefetching does indeed help to significantly reduce the average block read time and to improve the cache hit ratio. Both of these measures reflect the beneficial effect of caching and prefetching on individual I/O requests.

One of the major contributions of this work has been to demonstrate that the hit ratio metric is only a rough indicator of overall performance, defined as computation completion time. The cache hit ratio tends to be an optimistic measure which ignores prefetching overhead, the hit-wait time, contention factors, and the distribution of prefetching benefits among the processes.

Prefetching generally contributes to a decrease in the overall execution time of the parallel program. In most of the cases, the improvement exceeded 15 percent with improvement extending up to almost 70 percent. However, if the benefits are poorly distributed between processes, the total execution time can increase under prefetching, despite a reduced average block read time.

This study represents a substantial beginning in defining the problems that need to be addressed in order to make prefetching an effective approach, but more work is underway. Before attempting to replace the optimistic assumptions of this study with more realistic ones, it is desirable to strengthen the performance gains achievable with prefetching. Our results suggest that one of the critical issues is the conversion of I/O time into synchronization wait time rather than into actual time savings. Clever techniques for managing the distribution of prefetching benefits over the threads of a parallel computation may be valuable. Future work includes investigating mechanisms to gain information about the access patterns that may then be used in prefetching decisions, determining the scalability of these schemes, and examining other variations on file system organization.

REFERENCES


David F. Kott received the B.A. degree in computer science and physics from Dartmouth College, Hanover, NH, in 1986.

He is currently completing the Ph.D. degree in computer science at Duke University, Durham, NC. His research interests include secondary memory management and file system design for MIMD multiprocessors and algorithms for concurrent data structures.

Mr. Kott is a member of the Association for Computing Machinery.

Carla Schlatter Ellis received the B.S. degree from the University of Toledo, Toledo, OH, in 1972 and the M.S. and Ph.D. degrees from the University of Washington, Seattle, WA, in 1977 and 1979.

She is currently an Associate Professor in the Department of Computer Science, Duke University, Durham, NC. Previously, she was a member of the computer science faculty at the University of Oregon, Eugene, from 1978 to 1980, and at the University of Rochester, Rochester, NY, from 1980 to 1986. Her research interests include operating systems for multiprocessors, distributed file systems, and algorithms for concurrent data structures.

Dr. Ellis is a member of the Association of Computing Machinery and the IEEE Computer Society.