CS-1988-23

Prefetching in File Systems for MIMD Multiprocessors

Carla Schlatter Ellis
David Kotz

Department of Computer Science
Duke University
Durham, NC 27706

Prefetching in File Systems for MIMD Multiprocessors

Carla Schlatter Ellis
David Kotz

Duke University TR CS-1988-23, November 1988

References

[1] Carla Schlatter Ellis and David Kotz. Prefetching in file systems for mimd multiprocessors. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume 1, pages 306–314, St. Charles, IL, August 1989. Pennsylvania State University Press. Online at https://www.cs.dartmouth.edu/~kotz/research/ellis-prefetch/index.html.

[2] David Kotz. *Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors*. PhD thesis, Duke University, April 1991. Available as technical report CS-1991-016, Online at https://www.cs.dartmouth.edu/~kotz/research/kotz-thesis/index.html.

[3] David Kotz. Multiprocessor file system interfaces. Technical Report PCS-TR92-179, Dept. of Math and Computer Science, Dartmouth College, March 1992. Online at https://www.cs.dartmouth.edu/~kotz/research/kotz-fsint/index.html.

[4] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 60–67. IEEE, December 1991. DOI 10.1109/SPDP.1991.218296.

[5] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for parallel file systems. In *Proceedings of the International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 182–189. IEEE, December 1991. DOI 10.1109/PDIS.1991.183101.

[6] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. *Journal of Parallel and Distributed Computing*, 17(1–2) pages 140–145, January 1993. DOI 10.1006/jpdc.1993.1012.

[7] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1) pages 33–51, January 1993. DOI 10.1007/BF01277519.

[8] David F. Kotz and Carla Schlatter Ellis. Prefetching in file systems for mimd multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2) pages 218–230, April 1990. DOI 10.1109/71.80133.

Prefetching in File Systems for MIMD Multiprocessors*

Carla Schlatter Ellis
David Kotz

Department of Computer Science
Duke University
Durham, NC 27706

July 1988

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 overall execution time of a parallel computation. MIMD multiprocessor architectures have a profound impact on the nature of the workloads they support. In particular, it is the collective behavior of the processes in a parallel computation that often determines the performance. The assumptions about file access patterns that underlie much of the work in uniprocessor file management are called into question. Results from experiments performed on the Butterfly Plus multiprocessor are presented showing the benefits that can be derived from prefetching (e.g. significant improvements in the cache miss ratio and the average time to perform an I/O operation). We explore why is it not trivial to translate these gains into much better overall performance.

*This research was supported in part by NSF grant CCR-8721781.

1 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 of physical processes based on sensor data, and manipulation of large databases. Providing sufficient I/O bandwidth to keep highly parallel machines operating at full speed for very large problems is considerably difficult. There are two essential elements of any file system design intended for a highly parallel environment: parallel I/O and effective caching schemes.

The issues of I/O parallelism and disk striping have been extensively studied [15,11,13,9,10]. A prototype file system called Bridge [6] that runs on a BBN Butterfly parallel processor [1] is based upon the notion of an *interleaved file* in which consecutive logical blocks are assigned to devices on different processor nodes and can be accessed in parallel to provide increased I/O bandwidth. Gamma [5] is a database machine architecture that allows the tuples of relations to be interleaved across disks in a similar manner.

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. Traditionally, 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 [16]. 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 overall execution time of a parallel computation, even under favorable assumptions, and to provide guidelines for the development of prefetching policies for more realistic situations. In a subsequent paper, we consider heuristic procedures for deciding what blocks to prefetch based on the kind of information that can be easily gathered during execution. Our approach is experimental. We have developed a file system testbed called **RAPID Transit** (Read Ahead for Parallel Independent Disks) that runs on the Butterfly Plus multiprocessor [3] and allows implementations of various buffering and prefetching techniques to be evaluated.

In the next section, the expected patterns of parallel file access are identified to provide a better understanding of the demands on a parallel file system. Our models of the disk hardware and the file system interface and organization are defined in Section 3. Section 4 discusses some of the issues that distinguish buffering and prefetching strategies in parallel systems from those in traditional systems. In Sections 5 and 6, we describe the design of experiments performed with the testbed and results obtained that support our hypothesis that *carefully designed* prefetching can help to increase the performance of multiprocessor applications that have substantial I/O requirements.

2 Parallel File Access Patterns

Knowledge of the expected file access patterns is important for considering the layout of the file on the disk(s), choosing the block caching strategies, and designing prefetching techniques. The ability to recognize a pattern is a prerequisite for accurately predicting which data are no longer needed in the block cache and which data will be needed in the near future. In addition, these policies, once established, influence future access patterns.

Although file access patterns in uniprocessor and distributed systems are well-understood [7,14,8], it is not clear that similar patterns will be found in parallel environments, for a variety of reasons. We expect that the basic function of the file system will be to provide a long-term repository for data; large main memory capacity will reduce the need for temporary files to be placed on disk. The nature of parallel applications tends to be different and there is often a lack of program preparation activity within the multiprocessor itself. Thus, the files stored there should tend to be larger than on general purpose interactive systems where most of the referenced files have been found to be quite small (e.g. text files and program sources). Within parallel machines, it is also more common to have little or no multiprogramming of user processes from different computations. Thus, related cooperating processes must be responsible for generating the file reference locality that can be exploited by buffering and prefetching policies. This justifies examining access patterns on a per-computation per-file basis rather than system-wide. Finally, parallel algorithms and I/O may dictate the use of more complex patterns for the same application.

We have identified several possible forms of parallel file access. File access patterns 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 Figure 1). 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 [17]). 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,

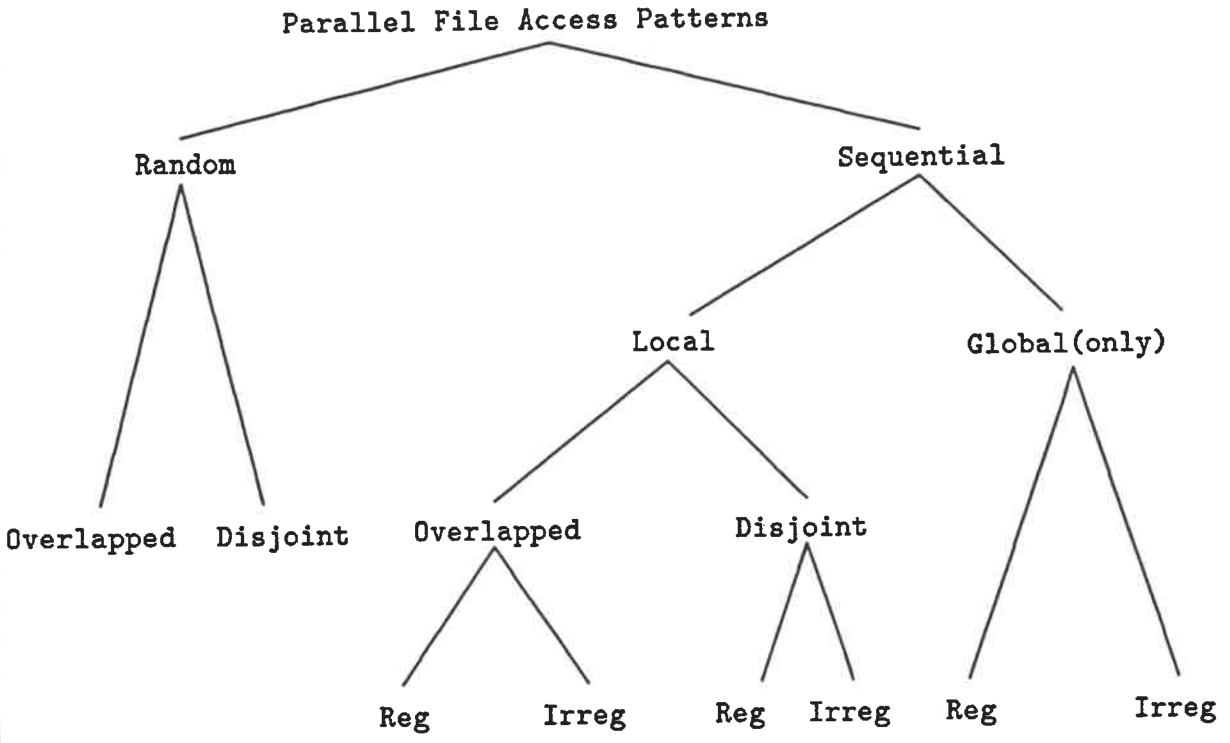


Figure 1: Categories of File Access Patterns

possibly reading disjoint blocks of the portion. In this view each process may be accessing blocks within the portion in some random or regular but increasing order. 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 makes global patterns more difficult to detect.

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*.

Crockett [4] proposes 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.

3 Architectural and File System Models

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. In this study, we have chosen only one of a variety of possibilities and leave consideration of alternative for future work.

The architectural model can be described as a medium to large scale MIMD shared memory multiprocessor with non-uniform memory access (NUMA). Most of the results also apply to MIMD distributed memory message-based multiprocessors. We believe that the key aspects of this architectural model offer significant advantages and will be relevant as larger and larger multiprocessors are designed and built. Distributed memory or NUMA architectures and those based upon multistage switching networks are exactly the MIMD designs that can scale.

In either case, secondary memory devices are attached directly to individual processor nodes or possibly to the switching network. We assume the availability of multiple conventional disks, each with a separate controller and channel. This allows for parallel access to all disks, each independently addressed. There are several possible ways of mapping a file to these disks including placing a file entirely on one disk, partitioning the file among disks, or interleaving the file over the disks in some manner. We use an interleaved structure with blocks of the file allocated round-robin to the disks as is done in Bridge [6].

To model the disk from a performance standpoint, we need to model the time required for a given disk request. The access time of a single request is defined by the physical access parameters of the device and by any contention encountered. For simplicity, we use a constant physical access time for each disk, allowing us to ignore the details of the layout on each disk. Contention can be handled by determining which disk holds the requested data and applying a simple event scheduling operation.

Basic assumptions about the structure of the file system are also necessary. One key question is whether file system software is multiprogrammed 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. Our current model is based on the multiprogrammed approach. 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. Another issue is the effect of an I/O request on the application. Logically, the application can be blocked for reads serviced by demand fetch since the data is required in order for it to continue. Prefetching operations involve some extra computation to determine the blocks to be fetched, which competes with the computation for cycles and ideally should occur during application idle time, and the actual I/O, which can overlap application computation time. The granularity of a fetch operation is one whole block, regardless of what the semantics may be for the read operation at the file system interface.

Buffering is an obvious prerequisite for prefetching. The area of management policies for the block cache is rich with design choices, some of which are topics of investigation in our broader study. One issue is the replacement strategy used to manage the buffer pool. Least-recently-used (LRU) replacement is an obvious candidate. For sequential access in database systems, however, Stonebraker [18] advocates a "toss-immediately" strategy. We use an algorithm that is an approximation of LRU and is specifically tuned for our architectural model. It is based upon a set of local, fixed size, recently-used (RU) lists, each ordered by time of last access originating locally, and a very simple, unordered, shared data structure encoding the global RU-set. This offers strong locality for the more complex list manipulations while enforcing a global policy.

Another issue, given our NUMA (or distributed memory) assumption, is the location of buffers allocated for a request. Alternative allocation strategies include:

1. Scattering the buffers, but treating them as a global buffer pool (i.e. any available buffer may be used). The scattering reduces the danger of memory contention in accessing the buffers, but raises the possibility of a bottleneck in allocation or mapping functions.

2. Using a buffer located at the site of the first request for the block, if possible. This strengthens the locality of access for the first request. If there are any subsequent requests from other processors that result in a hit on this buffer, they must pay for remote access. This choice offers more latitude for distributing management functions.

3. Using a buffer located at the processor to which the disk holding the requested block is attached. If only a subset of processor nodes support disks, this concentrates buffer operations. It allows distributed management, but even most first requests experience remote access.

It appears that this is a crucial issue. The overhead of maintaining buffers so that subsequent requests from any process have a chance of being satisfied without incurring a disk transfer can be significant. We have chosen to base the prefetching experiments described in this paper on alternative 1, for simplicity. The evaluation of these alternatives (and certain hybrids) is beyond the scope of this paper.

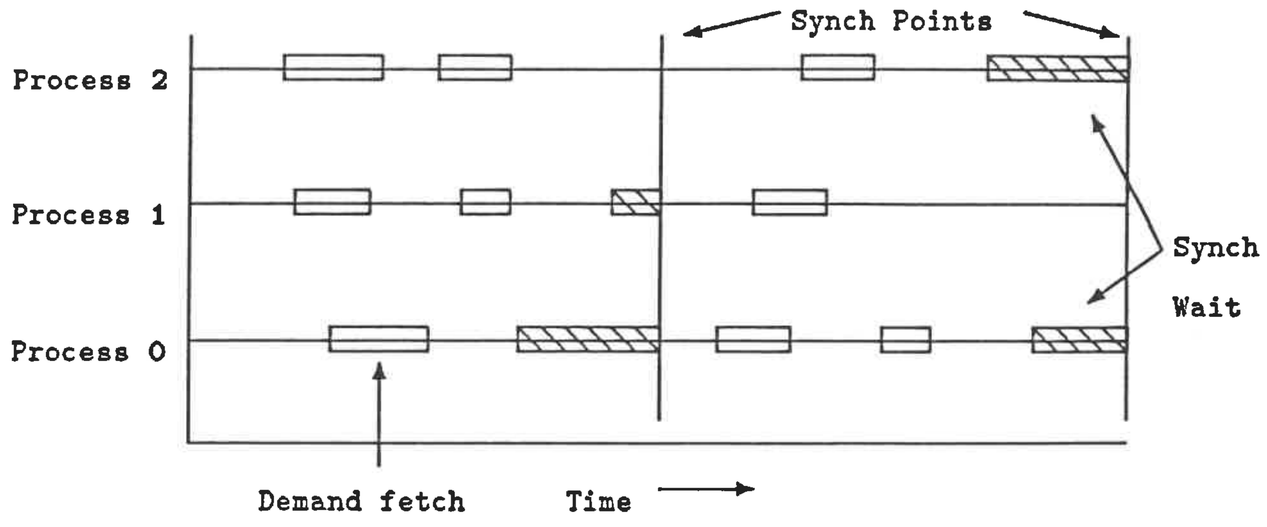
4 Prefetching Issues

A prefetch strategy must consider when to prefetch, what blocks to prefetch, and on which processor the prefetching decisions should be made and the data should be stored. However, the first question that needs to be addressed is whether it is worthwhile to develop prefetching strategies for the parallel environment.

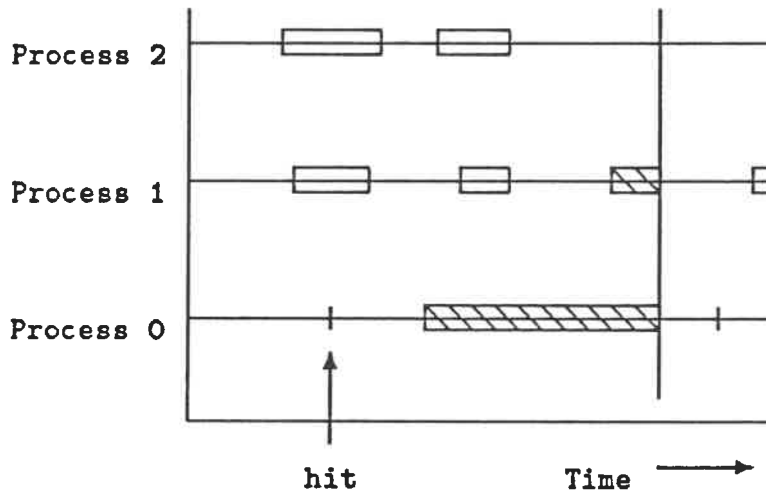
In a uniprocessor system with a workload consisting of single thread computations, every disk operation that can be replaced by a hit in the block cache translates directly into a time savings (assuming that any extra overhead introduced by buffer management is insignificant). Thus, an accepted and appropriate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

The nature of parallel multiprocess computations changes the situation significantly. As the following example (Figure 2) shows, the simple goal of increasing the hit ratio does not necessarily improve overall performance for the computation. This example shows three different executions of a three process parallel computation. Each process makes several read requests and there are periodic synchronization points where all three processes must meet. In case *a*, every read issued by the program results in a demand fetch from disk, delaying the process making the request relative to the other processes. In case *b*, three of the read requests are hits on the block cache (which has been filled by some unspecified mechanism with the needed blocks). For now, we allow the (generous) assumption that these buffer hits result in major time savings for the affected read requests. Unfortunately, the benefits are experienced by only one of the threads of the computation and it waits longer at the synchronization points for everyone else to catch up. There is no improvement in completion time of the computation from case *a* to case *b*, although the average time to service a read request has been reduced. The savings due to avoiding disk I/O are absorbed into increased synchronization time. Case *c* has exactly the same number of hits as case *b*, but in this case they are distributed more evenly over the processes. One of the intervals between synchronization points becomes shorter as a result. An optimal distribution of hits is one in which each additional hit affects the critical process that is determining the length of some synchronization interval. Without explicit control over the distribution of prefetching benefits, the completion time is likely to change in steps as the number of hits increases. For example, the completion time in case *c* is possible with 2, 3, 4, or 5 hits. Another hit crosses a threshold to shorten another iteration and further improve the completion time. This example illustrates the problem, introduces some of the new factors involved, and motivates the question of whether it will be possible to design effective policies.

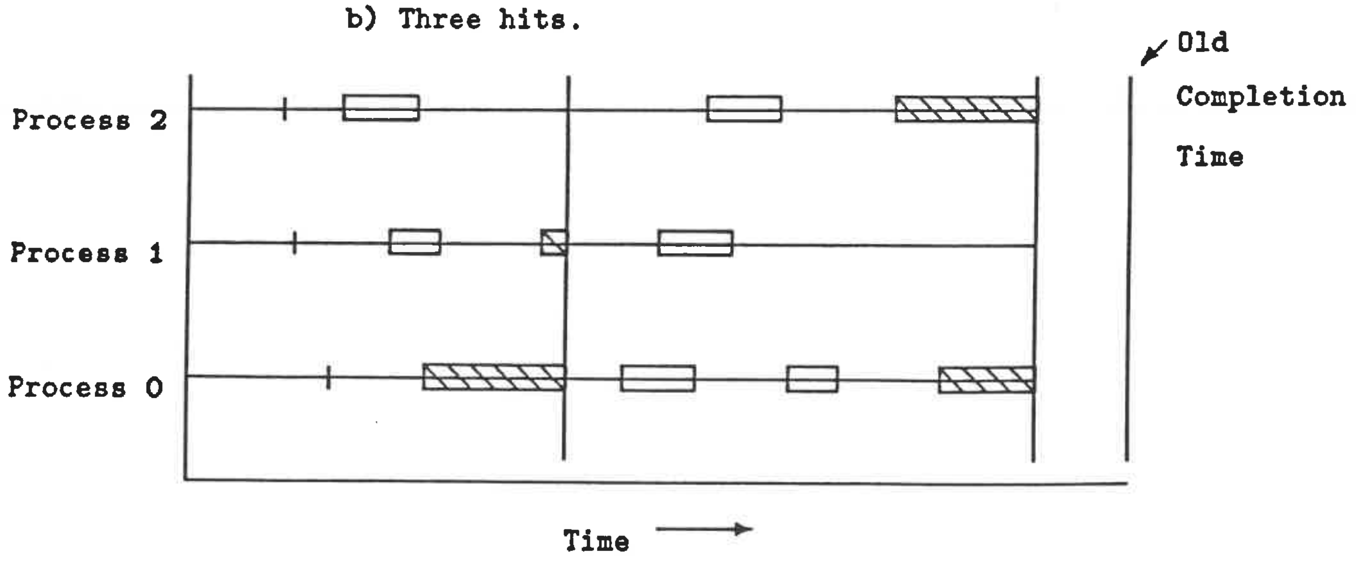
The issue of choosing blocks that make good candidates for prefetching is important. Prefetching mistakes (i.e. bringing in blocks that are never needed by the program) represent a major source of overhead. The file system may make decisions about what to prefetch based on information from a variety 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 typical uniprocessor file systems, such 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



a) All references served by demand fetch



b) Three hits.



c) Three hits, nicely distributed.

Figure 2: Executions of a Parallel Computation

the following block (i+1) so that it may be in its 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 the globally sequential parallel access pattern. 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.

In our experiments, designed to answer these questions of whether and under what conditions prefetching can be effective (described in more detail in Section 5), we defer full consideration of on-the-fly prediction algorithms for choosing which blocks should be prefetched. Instead, we provide, in advance, accurate information on the reference pattern. This near-perfect[1] prefetching advice can be used to provide an upper bound on the performance benefits of prefetching.

The nature of the parallel environment 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 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. Other than during these natural idle times, prefetching work 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 delay-

[1]The exact set of blocks to be accessed is provided, but the precise ordering may be impossible to determine in advance.

ing 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. One question arising from this approach is whether restricting prefetching opportunities to just these points can achieve significant enough benefits to cross a performance threshold. Another question is whether this approach tends to cluster prefetch requests (in time) so that they experience more severe contention for the disks with other prefetches or with concurrent demand fetches.

5 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.

5.1 Testbed

We have built a fully parallel file system derived from the BBN *RAMFile* system [2] that stresses caching on a BBN Butterfly Plus multiprocessor. In order to study the evolution of access patterns in parallel applications, we have made one version of this system that does I/O to the host file system available to our user community. However, in these prefetching studies, we use another version as a testbed, the **RAPID Transit** system, that simulates parallel disk I/O. Each processor is assumed to support a disk with files interleaved over all disks at the granularity of a single 1024-byte block. The disk I/O is simulated with an in-memory "disk" using artificial delays to approximate disk access times. Each disk has a *constant* access time of 30 msec, a reasonable approximation of the average access time in current technology for small, inexpensive disk drives of the kind that might be replicated in large numbers on a multiprocessor system. The size of the file system cache is a parameter of the experiments.

Each class of file access pattern has a corresponding prefetching strategy defined. Prefetching strategies based upon knowledge of the forthcoming block references are employed for this study. The supplied information 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 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 for random references or between irregular sequential portions where the information could not easily be derived. In addition, no explicit attempts are currently being made to balance the caching benefits over processors. However, processors do prefetch blocks that will be used exclusively by other processors.

Prefetching is performed during 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 (not waiting for I/O to complete).

5.2 Workload

We assume there exists a single parallel computation with one user process per processor node. This is the preferred organization for the kinds of very large programs in our target workload. The test applications are simple programs designed to drive a particular access pattern on a single file. This study restricts its attention to read-only files. 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 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.

There are six representative parallel file access patterns embedded in our synthetic applications:

lfp 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). Since an on-the-fly local procedure could easily predict future accesses following this pattern after a short amount of observation, our prefetching algorithm allows prefetching into succeeding portions.

lrp Local Random Portions: this local sequential pattern uses portions of 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 a portion until a demand fetch has established the location of the next sequential portion (i.e. it choses 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. This represents the ultimate single overlapped local sequential portion.

gfp Global Fixed Portions: (analogous to lfp) in this pattern, processors cooperate to read what appears globally to be sequential portions of fixed length and spacing.

grp Global Random Portions: (analogous to lrp) processors cooperate to globally read sequential portions with random length and spacing.

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. In this pattern, the blocks may be requested slightly out of order from the global viewpoint.

Certainly there are other possible access patterns, but these are variations or mixtures of the 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. Given the access patterns defined above, the following three types of synchronization points are considered: the processors may synchronize after reading a fixed number of blocks *per processor*, after reading a fixed number of blocks *total*, or after each sequential portion (local or global).

5.3 Measures

The primary metric for measuring the performance of the application is the overall completion time, since this is what prefetching is attempting to reduce. 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 a measure of the cache miss ratio.

For each of the three types of idle time mentioned above, 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. The complete history of events is also recorded.

5.4 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. After each block was read, delay was added in some tests to simulate computation; this delay was exponentially distributed with a mean of 30 msec, except where noted. 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.

The local RU-set size of each processor was one block, emulating a variation of a "toss-immediately" strategy within the context of our replacement algorithm. A number of buffers were allocated to be used only for prefetching, one block per processor (i.e. 20 blocks, in this case). This limited the global number of prefetched but not yet used blocks to a total corresponding to one per processor.

6 Results

The behavior of prefetching can be examined through a variety of performance criteria including reduced average block read time, reduced cache miss 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 have found that prefetching generally succeeds on all three counts, and we begin by examining the effects of prefetching on the average read time and the cache miss ratio.

6.1 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 Figure 3, the average block read time is significantly 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. Since all of the points lie under this line, the average read time for each instance has been reduced. It turns out that the *distribution* of read times is roughly the same when prefetching, except that there tend to be more very short reads (due to hits) and a few very long reads (due to prefetching overhead).

The miss ratio (shown in Figure 4) for all prefetching experiments was under 0.60, with nearly half of them under 0.08 (i.e. 92% of requests were found in the cache). Due to the sequential nature of the accesses, most of the corresponding experiments without prefetching had a miss ratio of 1.0 (those that did not had some measure of interprocess locality, as in the **lw** pattern).

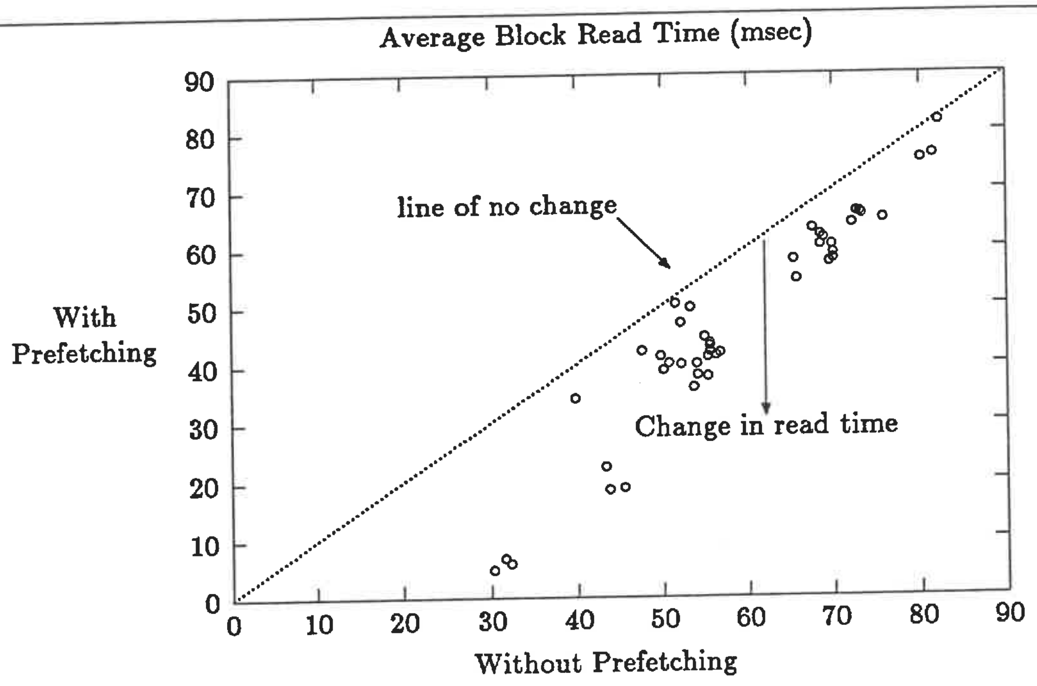


Figure 3: Prefetching consistently lowered the read time, with the largest improvements in the shorter read times.

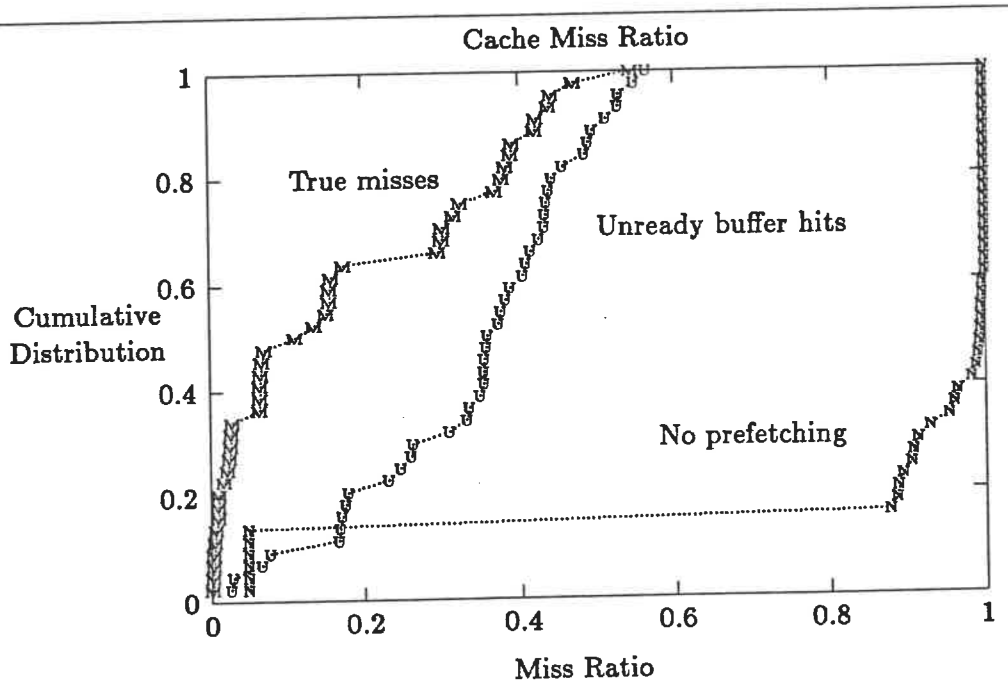


Figure 4: The cache miss ratio with prefetching is consistently under 0.6, with half of the points under 0.08. Unready buffer hits represent a significant fraction of all reads.

A strong improvement in the miss ratio is not enough to lower the average read time, however, since even 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 process must wait for the I/O to complete, so we call this the ***hit-wait*** time. The hits with a positive hit-wait time are ***unready*** buffer hits (ready buffer hits have a zero hit-wait time), and comprise about 50–60% of all buffer hits. These unready buffer hits may be construed as misses, but they do not have the same impact as true misses. The hit-wait time is generally much smaller than for a miss. However, hit-wait time can still be quite significant (although half of the points are under the minimum disk response time of 30 msec) and is a deciding factor in the time required to read a block. Figure 5 demonstrates the dependence, plotting the read time against the average hit-wait time. There is a very rough linear relation between the two. The roughness is to be expected due to other factors contributing to the read time. In contrast, no obvious relationship was found between the miss ratio and the read time.

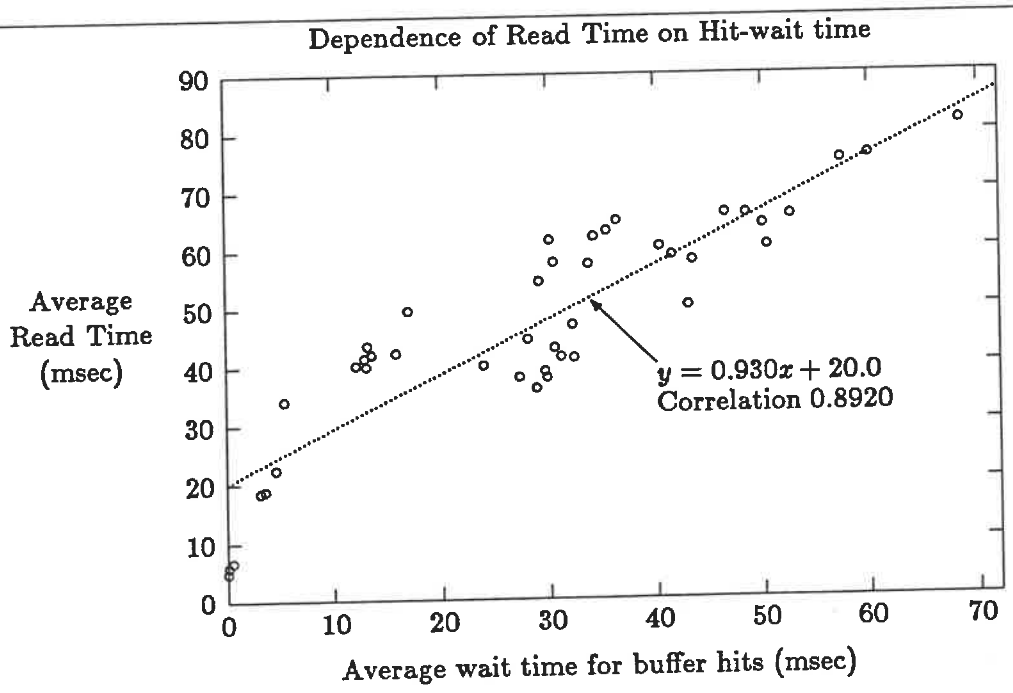


Figure 5: The average block read time is influenced strongly by the average hit-wait time, seen in the rough trend indicated by the line fit to the lower points.

One factor contributing to the average block read time and hit-wait times is disk contention. Contention for the disks is measured by the disk *response time*, the time from the entry of the request on the queue of the appropriate disk to the completion of the I/O. The disk response time, and therefore the hit-wait 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 Figure 6.

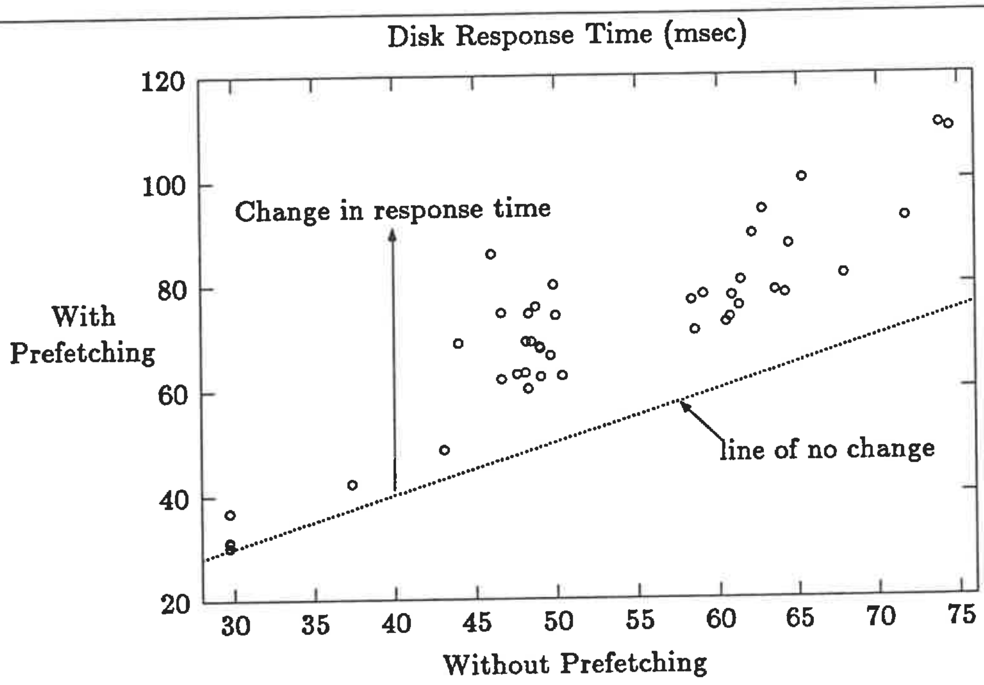


Figure 6: Scatter plot of the disk response time with and without prefetching, showing the increase caused by prefetching.

Note that the disks process no more requests under prefetching than they did without prefetching, since the prefetching strategy 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. However, even when the total time is not reduced, the disk reads tend to be issued non-uniformly, clustered into shorter intervals.

6.2 Effect on the Total Execution Time

Clearly the ultimate measure of the success of an optimization technique is the total execution time of the program. If the program takes less time to complete, then the technique is successful. We have found that prefetching is successful for most of the cases we studied, reducing the total execution time up to 53% in some cases. Figure 7 shows the change in execution time for all cases.

The improvement in the successful cases is often slight, and the slowdown in the other cases is less than 5%. The biggest improvements came to the **lw** pattern, discussed further in Section 6.4.

The total execution time was generally improved when the improvement in the block read time was high. It appears that the average read time must improve at least 16% for the total time to show any improvement, although the exact value is impossible to determine and depends on the particular application. This suggests a threshold effect coming into play. We conjecture that at this level of improvement in the read time, the benefits are beginning to be felt by all processes and, consequently, intervals between synchronization points, including the increases in the time required to synchronize which are caused by prefetching, have begun to shrink.

In contrast, the total execution time does not seem to be directly related to the miss ratio.

6.3 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 (between 6 and 16 msec.) to complete. The overrun time (i.e. time during which prefetching activity continues after the user process is again ready to run) can also be high (between 4 and 14 msec.). 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 remote (slower) 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 (minimum prefetch time). Experiments, varying the minimum prefetch time parameter, showed that the effect on the improvement in the total execution time was negligible. The improvement in average block read time was very slight. Thus, this proved to be an unproductive idea.

6.4 Differences Among the Patterns

Most of the preceding discussions have made no distinction between the data points based on the access pattern. In fact, the access pattern often accounts for many interesting differences between experiments.

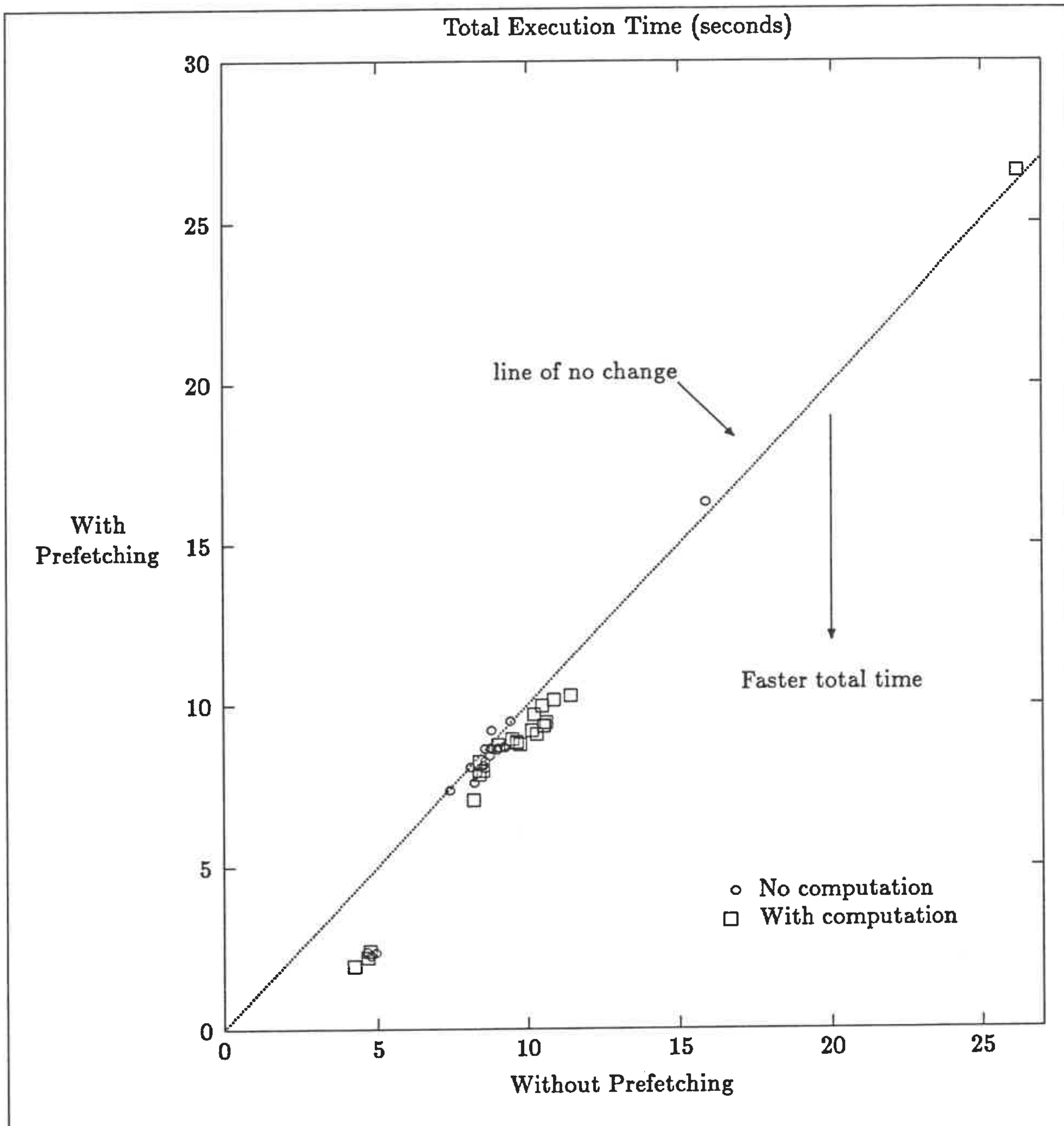


Figure 7: Prefetching improved the total execution time in most cases, with significant improvements in only a few cases.

For example, the **lw** (local whole-file) pattern appears to be gaining the most benefit from prefetching. Even without prefetching, **lw** has a miss ratio of only 0.05 since all 20 processors read only 100 blocks of the file (chosen so the total number of accesses is 2000, for consistency with the other patterns). Therefore, as long as the processes maintain inter-process locality, there will be 19 buffer hits for each block fetched. With prefetching, the number of misses is reduced to a single block, the first one. Because 20 processes benefit from each prefetched block, the benefits of prefetching are enormous compared to the other patterns, which have little (if any) inter-process locality. It is this aspect of **lw** that makes it unique among the other patterns and that makes its data stand out.

The local random portions pattern (**lrp**) stands out when synchronizing on every portion, since the portions are of unequal length and cause long synchronization delays. Otherwise, synchronizing every 200 blocks total rather than after every 10 each adds a measure of load-balancing that reduces the execution time.

The miss ratio shows strong differences among the patterns. It breaks up into four distinct groups, as shown in Figure 8. The local portioned patterns have the highest miss ratio and also have the lowest improvements in miss ratio. It is likely that this is due to the fact that local patterns will only prefetch blocks that are in their future reference string, with **lrp** not prefetching beyond its own current portion.

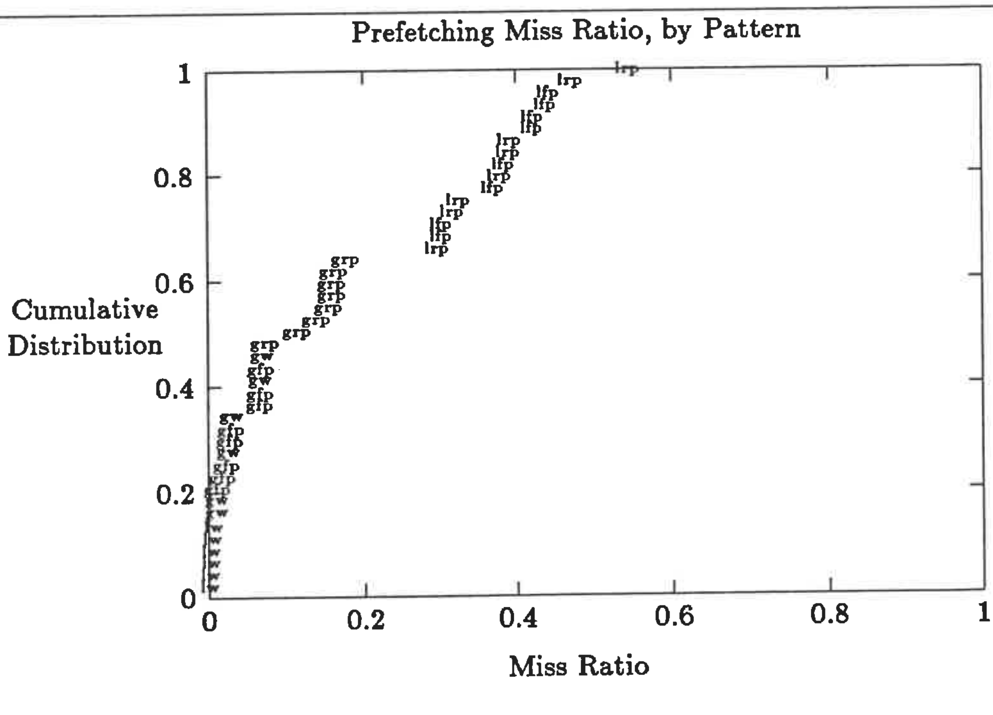


Figure 8: The miss ratio when prefetching tends to be very different for different patterns.

6.5 Other Findings

Many other experiments have been 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 [12].

The number of buffers

Increasing the number of buffers available for prefetching showed little performance improvement.

The balance between computation and I/O

In exploring the spectrum between I/O intensive and computationally intensive workloads (by varying the parameter controlling the delay that models computation time), we obtain expected results. As the character of the program switches from I/O to computation, the room for improvement from prefetching increases. The total execution time improves more with increasing amounts of 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 largely due to a increasingly large reduction in the average block read time, which falls to nearly 75% of its non-prefetching value. The hit-wait time is considerably reduced as more of the I/O time is overlapped with computation. One reason the hit-wait time and block read time improve 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.

An attempt to improve hit-wait time

Because of the importance of a low hit-wait time, we tried a change in the prefetching strategy to avoid prefetching blocks that are to be used soon and to prefetch well ahead of the current activity in the file. Experiments in which we varied this *minimum prefetch lead* showed that the hit-wait time was reduced considerably by increasing the minimum lead, except for the **lw** pattern, whose hit-wait time actually increased. However, the cache miss ratio climbed drastically for the global patterns to nearly 0.80 and for **lfp** to nearly 0.90.

We have emphasized the importance of the hit-wait time in determining the average block read time and found little dependence on the miss ratio. In this case, however, the miss ratio has increased so significantly that the improvements in the hit-wait time and fraction of ready hits have little effect on the the average block read time which actually increases slightly. The total execution time is not significantly affected (if anything, worsened) by increased minimum prefetching lead.

7 Summary

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 dynamics of parallel executions make traditional measures (e.g. hit ratio) much less indicative of overall performance. In this paper, we have begun to explore some of the issues involved in file data prefetching in multiprocessor environments. We are discovering that while it is important to reduce the impact of file I/O on parallel computations, it is not trivial to translate savings on individual I/O operations into gains in overall performance. However, there are some positive results and the negative results provide valuable insights.

We have developed a useful tool for experimenting with prefetching strategies. Experiments based upon our **RAPID Transit** file system testbed, 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), impact of local/remote memory accesses, and contention are real.

Our results show that prefetching does indeed help to significantly reduce the average block read time and the cache miss ratio, generally contributing to a decrease in the overall execution time of the parallel program. The total execution time depends primarily on the average block read time, rather than the miss ratio, as the cache miss ratio tends to be an optimistic measure while the read time takes into account prefetching overhead and the hit-wait time as well as the cache miss ratio. Indeed, the hit-wait time appears to be a crucial ingredient of the average block read time, as many buffer hits have not yet completed their I/O when they are accessed by processes of the parallel computation. Other important contributors to the read time are the disk response time, which tends to increase due to increased contention for the disks, and the prefetching overrun, caused by prefetching actions that do not complete before the end of the idle period they are intended to use.

Occasionally, the total execution time becomes longer under prefetching, despite a reduced average block read time. It appears that the block read time must improve a fair amount to reduce the overall execution time, since the total execution time is more affected by the variations in the lengths of the intervals between synchronization points; unless the slowest process in an interval is reduced, the length of the interval is not reduced. A significant reduction in average block read time may signify that the benefits of prefetching are being more generally seen by all processes in the computation.

Much more work is needed. We are investigating strategies for explicitly attempting to spread benefits more globally over all the threads of the parallel computation. How these schemes scale with increasing numbers of processors is an important consideration. Mechanisms for gaining information about the access patterns that may then be used in prefetching decisions must be studied. Further experiments with variations of file system organization and cache management (e.g. locality of buffers) are also planned.

References

[1] BBN. *Butterfly Parallel Processor Overview.* Technical Report, Bolt Beranek and Newman Adv. Computers Inc., June 1985.

[2] BBN. *The Butterfly RAMFile System.* Technical Report Number 6351, Bolt Beranek and Newman Adv. Computers Inc., September 1986.

[3] BBN. *Inside the Butterfly Plus.* Bolt Beranek and Newman Advanced Computers, Inc, Cambridge, MA, October 1987.

[4] Thomas W. Crockett. *File Concepts for Parallel I/O.* Technical Report, ICASE, NASA Langley, 1988.

[5] David J. Dewitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. *GAMMA: A High Performance Dataflow Database Machine.* Technical Report TR-635, Dept. of Computer Science, Univ. of Wisconsin-Madison, March 1986.

[6] P. Dibble, M. Scott, and C. Ellis. Bridge: a high-performance file system for parallel processors. In *Proceeding of International Conference on Distributed Computing Systems*, 1988.

[7] R. Floyd. *Short-term file reference patterns in a UNIX environment.* Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[8] R. Floyd and C. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Software Engineering*, to appear.

[9] H. Garcia-Molina and K. Salem. The impact of disk striping on reliability. *IEEE Database Engineering Bulletin*, 11(1):26–39, March 1988.

[10] R. Katz, J. Ousterhout, D. Patterson, and M. Stonebraker. A project on high performance I/O systems. *IEEE Database Engineering Bulletin*, 11(1):40–47, March 1988.

[11] M. Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[12] David Kotz. RAPID-Transit: Prefetching and buffering techniques for parallel I/O systems. July 1988. In preparation.

[13] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 59–68, May 1987.

[14] J. Ousterhout, H. DaCosta, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of 10th Symposium on Operating System Principles*, pages 15–24, December 1985.

[15] K. Salem and H. Garcia-Molina. *Disk Striping*. Technical Report 332, EECS Dept. Princeton Univ., December 1984.

[16] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[17] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[18] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.