12-1991

# Practical Prefetching Techniques for Parallel File Systems

David Kotz
*Dartmouth College*

Carla Schlatter Ellis
*Duke University*



Recommended Citation
David Kotz and Carla Schlatter Ellis. Practical Prefetching Techniques for Parallel File Systems. In Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS), December 1991. 10.1109/PDIS.1991.183101

# Practical Prefetching Techniques for Parallel File Systems*

David Kotz

Dept. of Math and Computer Science
Dartmouth College
Hanover, NH 03755-3551
David.Kotz@Dartmouth.edu

Carla Schlatter Ellis

Dept. of Computer Science
Duke University
Durham, NC 27706
carla@cs.duke.edu

## Abstract

*Improvements in the processing speed of multiprocessors are outpacing improvements in the speed of disk hardware. Parallel disk I/O subsystems have been proposed as one way to close the gap between processor and disk speeds. In a previous paper we showed that prefetching and caching have the* potential *to deliver the performance benefits of parallel file systems to parallel applications. In this paper we describe experiments with* practical *prefetching policies, and show that prefetching can be implemented efficiently even for the more complex parallel file access patterns. We also test the ability of these policies across a range of architectural parameters.*

## 1 Introduction

As computers grow more powerful, it becomes increasingly difficult to provide sufficient I/O bandwidth to keep them running at full speed for large problems, which may consume immense amounts of data. Disk I/O has always been slower than processing speed, and recent trends have shown that improvements in the speed of disk hardware are not keeping up with the increasing raw speed of processors. This widening access-time gap is known as the I/O crisis [14, 20]. The problem is compounded in typical parallel architectures that multiply the processing and memory capacity without balancing the I/O capabilities.

The most promising solution to the I/O crisis is to extend parallelism into the I/O subsystem. One such approach is to connect many disks to the computer in parallel, spreading individual files across all disks. Parallel disks could provide a significant boost in performance — possibly equal to the degree of parallelism, if there are no significant bottlenecks in the I/O subsystem, and if the I/O requests generated by applications can be mapped into lower-level operations that drive the available parallelism. Thus, the first challenge to the designers of a multiprocessor file system is to configure parallel disk hardware to avoid bottlenecks (e.g., shared busses), and to avoid further bottlenecks in the system software. An effective file system for a multiprocessor must itself be fully parallel to scale with additional processors or disks. The second challenge is to make this extensive disk hardware bandwidth easily available to application programs. To meet these challenges we propose a highly parallel file system implementation that incorporates caching and prefetching as a means of delivering the benefits of a parallel I/O architecture through to the user programs.

We expect a file cache to be useful in multiprocessor file systems for the same reason as in uniprocessor file systems: locality in file reference behavior. Indeed, we expect multiprocessor file access patterns to have increased opportunities for locality. *Interprocess locality* can arise when all processes in a multi-process program read the same file in some coordinated fashion (e.g., each reading different small records from the same block).

If the file access pattern is sequential, the file system can read blocks into the cache before they are requested, making them quickly available when they are requested. This extension to caching is known as *prefetching*. Prefetching does not work for all access patterns, of course, but it should be beneficial for common sequential patterns. In [9], we showed that prefetching has significant potential to improve read performance in multiprocessor file systems. We measured the potential using an idealistic prefetching policy that was provided with the complete file access pattern in advance. In practice, of course, the prefetching policy does not have access to the file access pattern in advance, and instead must base its prefetching decisions on a real-time view of the access pattern. This leads to several questions:

- Given that we know prefetching has potential, is it possible to design and implement *practical* prefetching policies? A practical policy must be both *effective*, choosing the correct blocks to prefetch, and *efficient*, having low overhead. This question is the primary focus of this paper.
- Can our practical policies achieve their full potential, as determined in [9] by our unrealizable "full-knowledge" policy?
- Can we design general policies that are practical for many different types of access patterns?

*This research was supported in part by NSF grants CCR-8721781 and CCR-8821809 and DARPA/NASA subcontract of NCC2-560.

- Do the prefetching policies and implementation scale well, given more processors, more disks, or a wider gap between processor speed and disk access speed?

To answer these questions, we used the testbed developed for [9]. The testbed implemented many prefetching and caching policies on a real multiprocessor, and simulated the parallel disk I/O. We evaluated many prefetching policies on a wide variety of workloads and architectural parameters.

In the next section we provide more background information. In Section 3 we describe the testbed, the workload, and the experimental methods. Section 4 defines our practical prefetching policies. In Section 5 we present the experiments, performance measures, and results. Section 6 concludes.

## 2 Background

Much of the previous work in I/O hardware parallelism has involved disk striping. In this technique, a file is interleaved across numerous disks and accessed in parallel to simultaneously obtain many blocks of the file with the positioning overhead of one block [16, 7, 14]. All of these schemes rely on a single controller to manage all of the disks.

For multiprocessors, one form of parallel disk architecture is based on the notion of *parallel, independent disks*, using multiple conventional disk devices addressed independently and attached to separate processors. The files may be interleaved over the disks, but the multiple controllers and independent access to the disks make this technique different from disk striping. Examples of this I/O architecture include the Concurrent File System [15, 6] for the Intel iPSC/2 multiprocessor, and the Bridge file system [4, 3] for the BBN Butterfly multiprocessor.

Caching commonly-used disk blocks can significantly improve file system performance [20], and indeed is a technique used in most modern file systems. Prefetching is also successful in uniprocessor file systems [20, 18, 19, 17]. The central idea behind prefetching is to overlap some of the I/O time with computation by issuing disk operations before they are requested. With parallel disk hardware, however, we expect prefetching to also overlap I/O with I/O, obtaining even larger benefits.

File access patterns have never been studied for parallel computers, but have been studied extensively for uniprocessors [5, 12]. Floyd [5] studied file access patterns in a Unix system, and found that 68% of files opened for reading are completely read, usually sequentially. Over 90% of all files opened are opened read-only or write-only. A classic Unix file system study [12] found that 90% of all files are processed sequentially, either through the whole file (70% of all accesses) or after only one seek. Parallel file access is discussed by Crockett [2]. Although he did not study an actual workload, he related file access patterns to possible storage techniques. Many of his basic file access patterns are reflected in our workload model.

We concentrate on scientific workloads, characterized by sequential access to large files [13, 11]. Despite the lack of any parallel file access study, we expect there to be enough sequential access in the parallel file access patterns of scientific applications for prefetching policies that assume sequential access to be successful.

## 3 Models and Methods

Our methodology is experimental, using a mix of implementation and simulation. We implemented a file system testbed called RAPID-Transit ("Read-Ahead for Parallel Independent Disks") on an actual multiprocessor. Since the multiprocessor does not have parallel disks, they are simulated. Unfortunately, few parallel programs use parallel I/O and so we did not have access to a real workload. Thus, we were forced to use a synthetic workload. The synthetic workload captures such nuances of real workloads as sequentiality, regularity, and inter-process interactions. It consists of real parallel programs that generate file requests and may incur synchronization delays. The testbed executes the synthetic application, measuring the elapsed real time and other significant statistics. This implementation of the policies on a real parallel processor, combined with real-time execution and measurement, allows us to directly include the effects of memory contention, synchronization overhead, inter-process dependencies, and other overhead, as they are caused by our workload under various management policies. This method allows us to evaluate whether *practical* prefetching policies can be implemented.

### 3.1 Models and Assumptions

**Architecture:** The architecture on which we base our research efforts is a multiple instruction stream, multiple data stream (MIMD) shared-memory multiprocessor. A subset of the problems and many of our proposed solutions (although not our implementation) may apply to message-passing architectures as well.

We represent the disk subsystem with parallel, independent disks. We assume an interleaved mapping of files to disks, with blocks of the file allocated round-robin to all disks in the system. The file system handles the mapping transparently, managing the disks and all requests for I/O. There is a file system manager running on each processor. This spreads the I/O overhead over all processors and allows the use of all processors for computation, rather than reserving a set of processors exclusively for I/O.

**Workload:** Parallel file systems and the applications that use them are not sufficiently mature for us to know what access patterns might be typical. Parallel applications may use patterns that are more complex than those used by uniprocess versions of the same application.

We work with file access patterns, rather than disk access patterns. That is, we examine the pattern of access to *logical* blocks of the file rather than *physical* blocks on the disk. The file access pattern is the best place to look for sequentiality, since disk access patterns are complicated by the layout of logical blocks on the disk and by the activities of multiple files. Thus

we make no assumptions of disk layout. Note also that the application is accessing *records* in the file, which are translated into accesses to logical file *blocks* by the interface to the file system. The file system internals, which are responsible for caching and prefetching, see only the block access pattern.

In our research we do not investigate read/write file access patterns, because most files are opened for either reading or writing, with few files updated [5, 12]. We expect this to be especially true for the large files used in scientific applications. This paper covers read-only patterns, whereas write-only patterns are covered in [10, 8].

All sequential patterns consist of a sequence of accesses to sequential *portions*. A portion is some number of contiguous 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. We call these *locally sequential access patterns,* or just local access patterns. 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, reading disjoint blocks of the portion. We call these *globally sequential access patterns,* or just global access patterns. In this view each process may be accessing blocks within the portion in some random or regular, but increasing order. If the reference strings of all the processes are merged with respect to time, the accesses follow a (roughly) sequential pattern. The pattern may not be strictly sequential due to the slight variations in the global ordering of the accesses; it is this variation that makes global patterns more difficult to detect.

In addition, the length of portions (in blocks) may be regular, so the file system could predict the end of a portion and not prefetch past it. The difference between the last block of one portion and the first of the next may also be regular (a regular skip), allowing the system to prefetch the first blocks of the next portion.

We use eight representative parallel file access patterns. Four of these are local patterns, three are global patterns, and one is random.

**lw** Local Whole file: every process reads the entire file from beginning to end. It is a special case of a local sequential pattern with a single portion.

**lfp** Local Fixed-length Portions: each process reads many sequential portions. The sequential portions have regular length and skip, although at different places in the file for each process.

**lrp** Local Random Portions: like **lfp**, but using portions of irregular (random) length and skip. Portions may overlap by coincidence.

**seg** Segmented: the file is divided into a set of non-overlapping contiguous segments, one per process. Each process thus has one sequential portion.

**gw** Global Whole file: the entire file is read from beginning to end. The processors read distinct records from the file in a self-scheduled order, so that globally the entire file is read exactly once.

**gfp** Global Fixed-length Portions: (analogous to **lfp**) processors cooperate to read what appears globally to be sequential portions of fixed length and skip.

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

**rnd** Random: records are accessed at random. This represents all patterns that are too complex to be represented as sequential in any way.

Note that these patterns are not necessarily representative of the *distribution* of the access patterns actually used by applications. We feel that this set covers the *range* of patterns likely to be used by scientific applications.

### 3.2 Methods

The RAPID-Transit testbed is a parallel program implemented on a BBN GP1000 Butterfly parallel processor [1]. The testbed is heavily parameterized, and incorporates the synthetic workload, the file system, and a set of simulated disks. The file system allocates and manages a buffer cache to hold disk blocks. See [8] for details.

Prefetching is attempted whenever the processor is idle. Assuming a commonly used processor-allocation strategy of one processor for each user process [21], the processor becomes idle whenever its assigned process is idle, usually waiting for disk activity or synchronization to complete. To decide on a block to prefetch, the prefetching module calls a *predictor*, which encapsulates a particular policy, a pattern-prediction heuristic. The predictor makes its predictions based on the observed reference history of the application.

The base for all of our evaluations of prefetching policies is the simple NONE policy, which is equivalent to not prefetching. We also use an *off-line* predictor called EXACT, which is provided with the entire access pattern in advance. (This is the approach used in [9].) The advance knowledge makes it a perfect predictor, since it makes no mistakes and requires little overhead. However it is not realistic, since a real predictor does not know the entire access pattern in advance. In this sense, EXACT gives us a rough upper bound on the potential of prefetching. (EXACT does have some limitations, however: in the **lrp** and **grp** patterns, it does not prefetch past the end of a portion until a demand fetch has established the location of the next sequential portion, and in the **rnd** pattern, EXACT does no prefetching, since none is reasonably possible.) We use these two simple predictors to evaluate our *on-line* predictors, described below.

## 4 Practical Predictors

Our strategy is to begin with a coarse comparison of many predictors on all the patterns, for a relatively

limited set of parameters. Then we evaluate the most generally practical predictors on a wide range of parameters, examining the scalability of the predictors to other architectural situations. We begin with predictors for local patterns, then consider global patterns.

### 4.1 Local Pattern Predictors

We present four predictors that are designed for predicting local access patterns. The fourth is a hybrid of the first three simpler predictors. These predictors monitor the individual process reference patterns, looking for sequential access. Since the process reference patterns are independent, these predictors are totally concurrent.

**OBL — One-Block Look-ahead:** This algorithm (as in [20]) always predicts block $i+1$ after block $i$ is referenced, and no more.

**IBL — Infinite-Block Look-ahead:** IBL predicts that $i+2, i+3, \ldots$ will follow a reference to $i$, and recommends that they all be prefetched in that order. Whether they are actually prefetched depends on the currently available resources. IBL is a logical extension of OBL, and is designed for the **lw** and **seg** patterns.

**PORT — Portion Recognition:** This algorithm attempts to recognize sequential portions. Essentially, PORT tries to handle the **lfp** access-pattern family. It watches for a regular portion length and regular portion skip. Like IBL, it tries to predict the pattern further ahead than the next reference, in order to prefetch more blocks. Unlike IBL, however, it limits the number of blocks that it predicts into the future (to limit mistakes), and it may also jump portion skips (if the portions are regular). In random patterns (short portions with irregular skip) PORT predicts nothing.

**IOPORT — IBL/OBL/PORT:** This predictor is a hybrid of the other three, attempting to combine the best of each. It begins as IBL, to treat **lw** and **seg** patterns efficiently, but switches to OBL on the first non-sequential reference. The conservative OBL is more appropriate when the pattern has unexpected non-sequential accesses. If regular portions are detected, then PORT is used.

### 4.2 Global Pattern Predictors

To recognize and predict globally sequential patterns at runtime is more difficult. The predictor must collect and examine the global reference history by merging local reference histories. Even then it is difficult to recognize sequential access, since the blocks in the pattern may be referenced in only a *roughly* sequential order due to variations in process speed. In addition, efficient, concurrent implementations are difficult due to the need for global decision making.

To determine the importance of the tradeoff between accuracy and efficiency, we compare a highly accurate (but inefficient) predictor with a less accurate (but efficient) predictor. Both predictors are concurrent, in that several processors may be active simultaneously, with internal synchronization controlling access to shared state information. The first, called GAPS, works hard to detect sequentiality in the global access pattern before doing any prefetching. The second, called RGAPS, assumes that the pattern is sequential unless it appears random. Detecting random access is much simpler and more concurrent, although less accurate, than detecting sequential access. Once they decide to prefetch, both predictors track all accesses and prefetches, and suggest blocks for prefetching that have not yet been fetched. In this mode they are capable of recognizing sequential portions, much like PORT, with unexpected non-sequential accesses requiring re-evaluation of the pattern. See [8] for details on these predictors.

## 5 Experiments

We begin with some details of our experiments and measures, then give results from experiments that compare the practical predictors against EXACT and NONE. Finally, we evaluate the scalability of the most general predictors.

### 5.1 Experimental Parameters

In all of our experiments, we fix most of the parameters and then vary one or two parameters at a time. The parameters described here are the base from which we make other variations. Each combination of parameters represents one test case.

There were 20 processes running on 20 processors. We generated a set of access patterns to be used by all predictors, including EXACT and NONE. The patterns all contained exactly 4000 record accesses, where the record size was one block. The block size was 1 KByte. In local patterns this was divided up as 200 references per process. Note that in most patterns this translates to 4000 blocks read from the disk, but in **lw** only 200 distinct blocks are read since all processes read the same set of 200 blocks. The cache contained 80 one-block buffers.

After each record was read, delay was added in some tests to simulate computation; this delay was exponentially distributed with a mean of 30 msec. All other tests had no delay after each read, simulating an I/O-intensive process.

The file was interleaved over 20 disks, at the granularity of a single block. Disk requests were queued in the appropriate disk queue. The disk service time was simulated using a *constant* artificial delay 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.

### 5.2 Measures

The RAPID-Transit testbed records many statistics intended to measure and interpret the performance of prefetching. The primary performance metric for measuring the performance of an application is the total execution time. This, and all time measures in the testbed, is real time, including *all* forms of overhead. We also record the average time to read a block, the total synchronization time, the cache hit ratio, prefetch overhead, and many others. In [9] we found that measures such as cache hit rate and average block

read time are improved with prefetching, but are not good indicators of overall performance. Total execution time incorporates those measures as well as other effects, such as synchronization delays, and thus it is the best measure of overall performance.

**A note on the data:** Every data point in each experiment represents the average of five trials. The *coefficient of variation* (*cv*) is the standard deviation divided by the mean (average). For all experiments in this paper, the *cv* was less than 0.09 (usually much less), meaning that the standard deviation over five trials was less than 9% of the mean. In many places we give the maximum *cv* for a given data set.

**Normalized Performance:** Due to limited data space we cannot present all of the experimental data (but see [8]). Instead, we use a summarizing measure. Since EXACT represents the potential for prefetching performance, we evaluate our on-line predictors in terms of their relative performance to EXACT. Our measure is the *normalized performance*, the ability of the on-line predictor to improve on NONE compared to EXACT's ability to improve on NONE. Thus, if $t_e$ is the execution time for EXACT, $t_n$ is the time for NONE, and $t$ is the time for some other predictor, the normalized performance of this other predictor is

$$\text{np} = \begin{cases} \frac{t-t_n}{t_e-t_n} & \text{if } t \geq t_e \\ 1 & \text{otherwise} \end{cases}$$

In the normal case $t \geq t_e$, so the normalized performance is 1 when the predictor in question does as well as EXACT, zero when it does only as well as NONE, and negative when slower than NONE. If both EXACT and the on-line predictor are slower than NONE, the normalized performance may also be greater than 1. Thus, it is best to have a normalized performance near 1. The case $t < t_e$ is considered an anomaly, since an on-line predictor should not run faster than EXACT (although it did sometimes happen for subtle reasons [8]). We assign these cases a normalized performance of 1, since they have certainly reached the full potential of EXACT. The normalized performance is undefined for the **rnd** pattern, in which $t_e \equiv t_n$.

**The Ideal Execution Time:** We also compare the experimental execution time to a simple model of the ideal execution time. The total execution time is a combination of the computation time, the I/O time, and overhead. In the ideal situation, there is no overhead, and either all of the I/O is overlapped by computation or all of the computation is overlapped by I/O. Thus, the ideal execution time is simply the maximum of the I/O time and the computation time. This assumes that the workload is evenly divided among the disks and processors and that the disks are perfectly utilized. No real execution of the program can be faster than the ideal execution time. With the base parameter values, both the I/O and computation times are 6 seconds, and thus the ideal execution time is also 6 seconds. The ideal I/O time for **lw** is shorter, only 0.3 seconds, since it only reads 200 blocks from disk.

### 5.3 Results for Local Pattern Predictors

We measured the performance of the local pattern predictors on the synthetic workload, using the experimental parameters defined in Section 5.1, and varying the pattern, predictor, synchronization style, and computation (either some computation or no computation), each variation forming a different test case. The primary measure was total execution time, summarized with the normalized-performance metric. Figure 1 plots the distribution of normalized performance that each predictor achieved over the set of test cases, in the form of a cumulative distribution function (CDF). Recall that the desired normalized performance is 1.0, indicating that the on-line predictor performed as well as EXACT. IBL's extreme negative and positive values indicate that it was much slower than EXACT in some cases. OBL had relatively few values near one. IOPORT had the best minimum value, with only two negative points, and was within 5% of EXACT's performance in over half of all test cases.

In the **rnd** pattern, which is not included in Figure 1, PORT and IOPORT were within 2% of the execution time for EXACT (NONE) in all test cases. They recognized the random pattern as an irregular set of one-block portions, and did no prefetching. OBL and IBL, however, prefetched blindly, running up to 3.5 times slower than NONE. Thus, IOPORT is a good general-purpose local predictor: excellent performance most of the time, mediocre performance some of the time, and never any terrible performance.

All of the above experiments used a one-block record size. With non-integral record sizes (i.e., not a multiple of the block size), some blocks are rereferenced. All of the above predictors handle such rereferences by ignoring them, and thus the performance did not vary much with the record size (we experimented with IOPORT for record sizes varying from one-quarter block to 10 blocks). For small records (less than one block) the overhead of the rereferences was enough to slow down execution by a few percent in some cases (NONE was the most affected, slowing down by 8% in one case).

### 5.4 Results for Global Pattern Predictors

Using a set of tests similar to those for local predictors, except using global patterns, we measured the performance of GAPS and RGAPS on the synthetic workload. We plot the CDFs of the distributions of the normalized performance in Figure 2. The low-performance (negative) cases were all from the **grp** pattern, where GAPS and RGAPS were slower than NONE. In general, however, half of the GAPS cases reached at least 0.62 normalized performance (i.e., 62% of the performance improvement of EXACT), and half of the RGAPS cases reached at least 0.71 normalized performance. In the **rnd** pattern, which is not included in Figure 2, GAPS and RGAPS were both within 2% of the the EXACT (NONE) time, which is essentially no difference. Thus, they both handled random patterns efficiently.

All of the above experiments used a one-block record size. With longer records (multiple blocks),

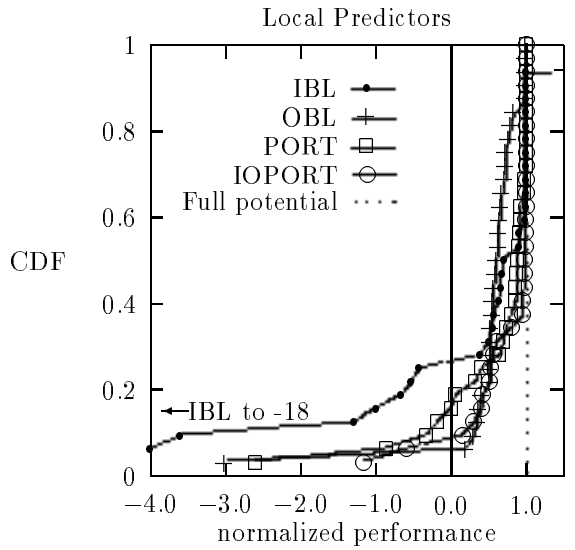


Figure 1: The normalized performance for the local predictors on all patterns except **rnd**. A normalized performance of 1.0 indicates that the predictor matched (or exceeded) EXACT's performance, and a negative (or large positive) number indicates that it was slower than NONE. IBL's range was -18 to 20. (Total execution time $cv < 0.058$.)

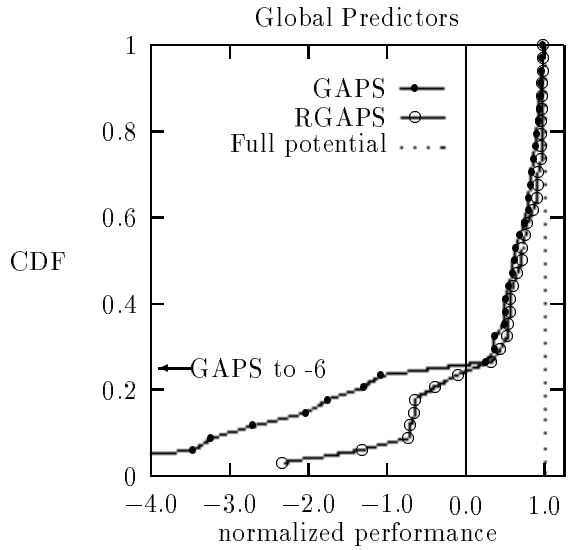


Figure 2: The normalized performance for GAPS and RGAPS on all patterns except **rnd**. (Total execution time $cv < 0.072$.)

it became more difficult to detect sequentiality in the block access pattern. GAPS, in fact, failed for records larger than four blocks, and ran up to 10 times slower than without prefetching, because of its failed efforts to recognize the sequentiality. RGAPS had little difficulty with varying record size, closely following EXACT's performance. Thus RGAPS was a more generally successful predictor than GAPS.

### 5.5 Scalability

Once we knew that IOPORT and RGAPS were reasonably general and successful predictors for the various access patterns in our workload, we evaluated their practicality across a wide range of architectural variations. In particular, we varied the number of processors, the number of disks, and the ratio of processor speed to disk speed. We give a sample of the results here, along with the key conclusions; see [8] for a full presentation.

**Number of processors:** We varied the number of processors to test the scalability of the file system software, including the predictors. By holding the number of disks constant at 20, this also allowed us to study the effects of having more or fewer processors than disks, since the preceding experiments always had 20 processors and 20 disks. (Essentially the same conclusions were found when holding the number of processors at 20 and varying the number of disks from 1 to 35.) The total amount of work (blocks read, computation time) was also held constant. The ideal execution time was then $\max(6, \frac{C}{p})$ seconds, where $C$ was the total computation time in seconds, and $p$ was the number of processors. We used either $C = 0$ or $C = 120$ seconds, as before.

Figure 3 shows the results for the **lfp** pattern with computation, for various numbers of processors. The ideal execution time decreased with more processors until, limited by I/O, it leveled off to 6 seconds at 20 processors. EXACT followed this curve closely, and IOPORT nearly matched EXACT (normalized performance 0.86–0.96 throughout). NONE was much slower, particularly for few processors. NONE could not use more disks than it had processors, so it was unable to use the full parallel disk bandwidth or to overlap computation and I/O. This graph shows that prefetching successfully overlapped computation and I/O, and scaled well (at least up to 32 processors). The results for other patterns with computation were similar (using RGAPS instead of IOPORT in global patterns).

Figure 4 shows the results for the I/O-bound **gfp** pattern. The ideal execution time is a constant 6 seconds. NONE could not use more disks than it had processors, and thus could not use the full parallel disk bandwidth. However, prefetching was able to use all of the disk bandwidth with only a few processors. The results for **gw**, **lfp**, and **seg** were similar. Prefetching had more difficulty in the **grp** and **lrp** patterns, though still faster than not prefetching for less than 20 processors. In the **lw** pattern, NONE was limited to one disk at a time, regardless of the number of processors, while prefetching used all of the disks.

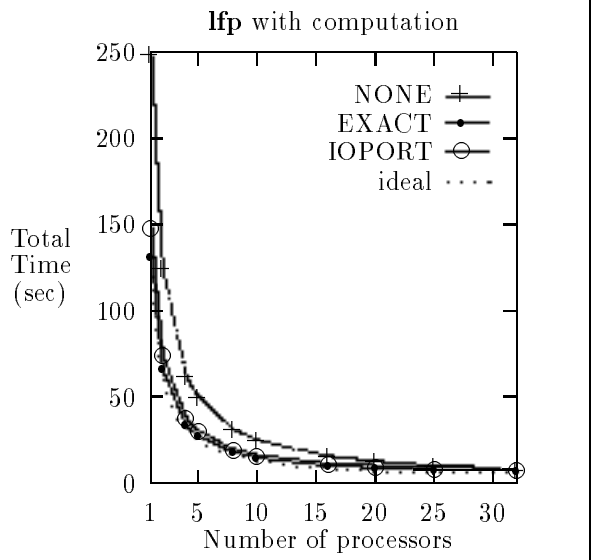


Figure 3: Processors variation. ($cv < 0.016$)

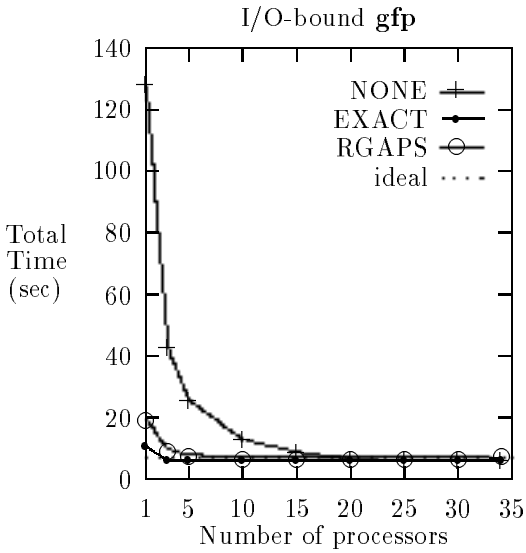


Figure 4: Processors variation. ($cv < 0.050$)

When there were more processors than disks, NONE was sometimes slightly faster than all other predictors. At this point the parallelism alone was enough to keep the disks occupied, whereas prefetching required more overhead for the same task, and also made mistakes. Since we expect that most multiprocessors will (and do) have more processors than disks, this is somewhat of a negative result. However, the small slowdown caused by prefetching when there were more processors than disks is a small price to pay for the many other cases where prefetching had significant benefits (e.g., small record sizes, fewer processors than disks, the **lw** pattern, or unbalanced disk loads).

In all, the IOPORT and RGAPS predictors were practical across the variation of the number of processors (there is not enough evidence to extrapolate RGAPS's scalability past 34 processors). They had particularly good performance when there were fewer processors than disks, and only slightly negative performance in some cases when there were more processors than disks. In any application, the bottleneck will limit performance, so for higher performance both the number of processors and the number of disks must be increased, with the exact ratio depending on the expected access patterns and computational loads.

**Disk access time:** It is expected that both processor speed and disk speed will increase with time, but that the increase in processor speed will outstrip any increases in disk speed, making disks appear slower to processors than they are today. We were not able to change the processor speed, since we were using a single type of machine, but (since the disks were simulated) we could easily change the disk access time. Thus we could test the behavior of prefetching as the access-time gap changed.

As an example, Figure 5 plots the total execution time for **gfp** as a function of the disk access time. The ideal execution time is linear in the disk access time, since this pattern contains no computation. EXACT followed the ideal curve, and the others at least matched its slope except for the fastest disks, indicating only a constant overhead. With faster disks relative to the processor speed (an unlikely occurrence given architectural trends), RGAPS broke down and became slower than NONE. This is because the benefits of prefetching were reduced with the decreased disk access time, but the costs of prefetching (a function of processor speed) were unchanged. For slower disks, the success of prefetching scaled directly with the disk access time. Thus, as the access-time gap widens, prefetching should continue to be beneficial. Similar conclusions were reached for other patterns.

## 6 Conclusion

We present a practical predictor for general-purpose local-pattern workloads (IOPORT), and a practical predictor for general-purpose global-pattern workloads (RGAPS). The two predictors were able to improve on the non-prefetching time in many cases. In the few cases where their prefetching was not beneficial, the resulting performance loss was minor. They were remarkably successful at reaching the potential for prefetching, as determined with the EXACT predictor and originally reported in [9]. In addition, we found that these predictors were robust across variations in architectural parameters, such as the number of disks, number of processors, and disk access time. These are important considerations, because we expect to see an increasing gap between processor speed and disk access time, and we expect to see machines with more processors and more disks.

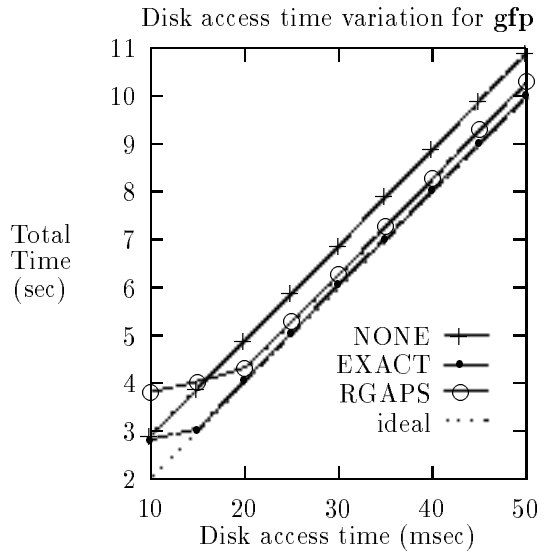


Figure 5: Disk-access time variation. ($cv < 0.087$)

## References


[1] BBN Advanced Computers. *Butterfly Products Overview*, 1987.

[2] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[3] Peter Dibble, Michael Scott, and Carla Ellis. Bridge: A high-performance file system for parallel processors. In *Proceedings of the Eighth International Conference on Distributed Computer Systems*, pages 154–161, June 1988.

[4] Peter C. Dibble. *A Parallel Interleaved File System*. PhD thesis, University of Rochester, March 1990.

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

[6] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 178–187, 1991.

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

[8] 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.

[9] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990.

[10] David Kotz and Carla Schlatter Ellis. Caching and writeback policies in parallel file systems. In *1991 IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[11] Ethan Miller. Input/Output behavior of supercomputing applications. Technical Report UCB/CSD 91/616, University of California, Berkeley, 1991. Submitted to Supercomputing '91.

[12] John Ousterhout, Hervé Da Costa, David Harrison, John Kunze, Mike Kupfer, and James Thompson. A trace driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[13] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM Operating Systems Review*, 23(1):11–28, January 1989.

[14] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[15] Paul Pierce. A concurrent file system for a highly parallel mass storage system. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 155–160, 1989.

[16] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[17] Alan Jay Smith. Sequential program prefetching in memory heirarchies. *IEEE Computer*, pages 7–21, December 1978.

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

[19] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

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

[21] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 159–166, December 1989.