4-2-1991

Prefetching and Caching Techniques in File Systems for Mimd Multiprocessors

David F. Kotz
Dartmouth College

CS–1991–16

Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors

David F. Kotz

Department of Computer Science

Duke University

Durham, North Carolina 27706

April 2, 1991

NOTE: this report is meant to be printed 2-sided. For best effect, make a copy this report using a copier that can produce 2-sided output.

You might want to substitute a blank page for this page.

Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors

David F. Kotz

April 2, 1991

Supervised by Carla S. Ellis

Dissertation submitted in partial fulfillment
of the requirements for the degree
of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

This document is a reformatted version of the disseration, and equivalent in content.

Abstract

The increasing speed of the most powerful computers, especially multiprocessors, makes it difficult to provide sufficient I/O bandwidth to keep them running at full speed for the largest problems. Trends show that the difference in the speed of disk hardware and the speed of processors is increasing, with I/O severely limiting the performance of otherwise fast machines. This widening access-time gap is known as the "I/O bottleneck crisis." One solution to the crisis, suggested by many researchers, is to use many disks in parallel to increase the overall bandwidth.

This dissertation studies some of the file system issues needed to get high performance from parallel disk systems, since parallel hardware alone cannot guarantee good performance. The target systems are large MIMD multiprocessors used for scientific applications, with large files spread over multiple disks attached in parallel. The focus is on automatic caching and prefetching techniques. We show that caching and prefetching can transparently provide the power of parallel disk hardware to both sequential and parallel applications using a conventional file system interface. We also propose a new file system interface (compatible with the conventional interface) that could make it easier to use parallel disks effectively.

Our methodology is a mixture of implementation and simulation, using a software testbed that we built to run on a BBN GP1000 multiprocessor. The testbed simulates the disks and fully implements the caching and prefetching policies. Using a synthetic workload as input, we use the testbed in an extensive set of experiments. The results show that prefetching and caching improved the performance of parallel file systems, often dramatically.

Acknowledgements

Needless to say, a Ph.D. cannot be obtained without the support of many other people. The significance of my parents' love, encouragement, and support cannot be understated, having helped me through twenty years of schooling.

My advisor Carla Ellis has been a wonderful mentor, colleague, and yes, "mom", over the last four years. She has guided me into the world of experimental research, and through many difficult new experiences. She has eased me through my failures and encouraged my successes. Best of all, she has allowed me flexibility while still cracking the whip when I needed it.

Pamela Jenkins has been a solid support for me throughout. She has helped me keep my sanity, boosted my motivation when it failed, and shared all the highs and lows of a graduate student's life. In particular, her caring and incredible strength got me through the roughest period of my life, recovering from a broken neck.

Thanks to everyone in the department for their encouragement while I walked around with a "halo" neck brace, looking like an alien in a bad sci-fi movie. Thanks especially to all of my friends in the department, including Owen, Vick, Rick, and many others. What would life be without wasting time in Owen's office, on frisbee golf expeditions, or at lunch-time bridge games?

Thanks to the BUG group and Rick Floyd for helping with Chapter 10 and for listening to many practice talks. Thanks to my committee for their helpful suggestions. Finally, thanks to the organizations that paid the tuition and put food on my table: this research was supported by an MCNC graduate fellowship, two NSF research assistantships (under grants CCR8721781 and CCR8821809), and two DARPA/UMIACS/NASA Parallel Processing Research Assistantships.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *I/O, to disks, to networks, and to user-oriented devices, is expected to become the central problem in future systems* [Cab90].

1.1 The I/O Crisis

As computers grow more powerful, it becomes increasingly difficult to provide sufficient I/O bandwidth to keep them running at full speed for the largest problems. The increasing speed and memory capacity of the most powerful computers allow them to solve ever larger problems, requiring immense amounts of data at high speeds. Disk I/O has always been slower than processing speed, and recent trends have shown that improvements in the speed of disk hardware will not keep up with the increasing speed of processors. This problem, the widening access-time gap, is known as the I/O crisis [PGK88].

Smith [Smi85b] argues that the widening access-time gap affects uniprocessors, multiprocessors, and distributed systems. In short, his argument is that CPU speeds (and therefore I/O needs) are doubling every three to six years, whereas disk access times are decreasing much more slowly. Thus, the gap is widening. Patterson, Gibson, and Katz [PGK88] agree, claiming that microprocessor speeds double every year, whereas disk seek speed has only doubled once in the last ten years, and disk rotation speed has not changed. These trends are expected to continue.

A classic postulate of Gene Amdahl is that computers require 1 byte of memory and one bit per second (bps) of I/O for every instruction per second (IPS) of CPU power [SBN82]. Our 64-node BBN GP1000 multiprocessor [BBN87] has a 2.5-MIPS Motorola 68020 processor at each node, for a total of $64 \times 2.5 = 160$ MIPS. There are $64 \times 4 = 256$ megabytes of memory, which is more than enough according to Amdahl. There should be 160 Mbps of raw I/O bandwidth for this machine, but in the standard configuration the single disk controller peaks at only 19 Mbps.[1] Thus, the standard configuration for this machine is far out of balance with respect to I/O (and would be worse with more processor nodes). Adding eight 19-Mbps disk nodes would balance this system, assuming the full disk bandwidth could be made available and usable.

Several examples of scientific applications requiring high-performance I/O are given in [Int89]. These include fluid-flow modeling, molecular modeling, seismic data processing, and tactical simulation. In general, these applications process tremendous amounts of data in a short time. Often the working data-set size is too large for main memory, requiring large scratch files. When the application must run for a long time, checkpointing is necessary to enable the application to continue after an interruption; such images may require gigabytes of storage. Certain applications,

[1] All GP1000 performance figures from BBN promotional literature.

such as seismic processing, read large amounts of raw data, on the order of 4–16 gigabytes. At the National Center for Atmospheric Research, the mass storage system handles well over 100 gigabytes of data per day, from a combined storage of 15 terabytes [O'L90]. In the future, they expect to have more than a petabyte (10^{15} bytes) of storage, and routinely transfer 5–10 terabytes as part of a single simulation. Such large data requirements require an I/O system that can handle large data transfers quickly, or the machine quickly becomes overwhelmed by I/O.

1.2 I/O Parallelism as a Solution

One promising solution to the I/O crisis is a parallel I/O subsystem. The idea is to connect many disks to the computer in parallel, spreading individual files across all disks. This disk-hardware parallelism has been studied extensively, particularly for application to serial processors, and more recently for parallel computers (see Section 2.1). Parallel disks can provide a significant boost in performance — possibly equal to the amount of parallelism, if there are no significant bottlenecks, such as shared busses or controllers.

1.3 How to Use Parallel I/O

Just as parallel processors are not sufficient to guarantee high computational performance, parallel disk hardware is not the complete answer to the I/O crisis. It is equally important to design effective parallel systems software. A sequential file system, or file access from only one process in an application, forms a bottleneck that reduces the effectiveness of parallel I/O hardware. In contrast, a well-designed parallel file system can help even a naive program to harness the power of parallel disk hardware. This dissertation addresses file system mechanisms to improve the performance of applications on a multiprocessor with parallel disk hardware. The goal is to transparently provide the full disk parallelism, even to applications that do not use parallel file access methods and to those that may not use them effectively.

Smith [Smi85b] acknowledges that multiprogramming is an effective way to mask disk latency, but concedes that even multiprogramming will not suffice for this purpose in the future. Indeed, in many parallel computers the individual processors are not multiprogrammed among several user processes. Smith's solution to the I/O crisis is disk caching. Caching is effective for improving the performance of I/O systems in conventional uniprocessor systems (see Section 2.2). In Section 4.2 we demonstrate with some experiments the importance of caching in parallel file systems.

Cache performance can be improved by reading blocks into the cache before they are requested. This read-ahead is known as *prefetching* [Smi78c]. Prefetching is an important technique in uniprocessor file systems, but the techniques for uniprocessor prefetching may not directly apply to the problem of prefetching in parallel file systems. Certain simple strategies, such as always prefetching the next block, may yield good prediction but may not necessarily work well with respect to improving overall performance. Furthermore, techniques based on detecting sequential file access may be more difficult in parallel due to more complex file access patterns. In fact, the definition of sequential file access must change.

Prefetching is appropriate when *reading* files. Another set of issues are involved in *writing* files, in particular timing the writes of dirty blocks to disk. The solutions to some of these problems are treated in Chapter 9.

In this dissertation we demonstrate three major points:

- Caching and prefetching can improve disk performance in parallel applications. They do this by overlapping I/O with computation and I/O with I/O (using parallel disks).

- Prefetching can make the power of parallel disks available to single-process applications that would otherwise use only one disk at a time.
- While caching and prefetching can deliver significant I/O parallelism without requiring changes to the file system interface, enhancements to the interface (compatible with the traditional interface) might make it easier to use parallel disks, and aid automatic prefetching.

In the next chapter we outline the other work in the field, covering parallel disk hardware, disk caching, file access patterns, and prefetching in CPU caches, disk caches, and virtual memory. In Chapter 3 we define our workload models and architectural assumptions. Our methods are discussed in Chapter 4. Experiments on read-only access patterns are covered by Chapters 5 through 8. Chapter 5 examines the potential for prefetching to improve parallel I/O performance. Chapters 6 and 7 outline heuristic prefetching techniques and give experimental results. In Chapter 8 the effects of several architectural parameters are studied. Buffering of disk writes is considered in Chapter 9. We propose extensions to the traditional file system interface in Chapter 10. In Chapter 11 we conclude and present some ideas for further work. A glossary, beginning on page 175, collects the definitions of terms and acronyms used in this document.

Chapter 2

Literature Survey

Many researchers have emphasized the importance of increasing the performance of I/O to keep pace with ever-increasing processor performance. This is especially a problem with the advent of powerful parallel processors. Boral and Dewitt [BD83] argue that I/O bandwidth is a crucial bottleneck for database processing and that a significant increase in bandwidth is necessary if parallel database machines are to be effective. They propose two solutions: hardware parallelism using multiple conventional disks, or a large file cache to retain blocks that are re-used and to hold blocks that are prefetched. Some researchers [Smi85b, PGK88] have emphasized the increasing gap in memory and disk access times and predict large bottlenecks at the I/O system for high-performance computers. They propose caching and prefetching [Smi85b] and parallel disk hardware [PGK88, FH88, WCM88] as near-term solutions. The file system for the NEC supercomputer demonstrates several techniques for boosting I/O performance, including contiguous allocation, prefetching, and optimizing for large files [NNI89].

2.1 Parallel I/O Hardware

Hardware parallelism implies multiple disk drives[1] and possibly multiple disk controllers and channels. Traditional systems may use multiple disks for reasons of size, speed, or reliability, but they rarely spread a single file over multiple disks to parallelize access to that file.

There are several ways of organizing multiple disks in a parallel fashion. The same terms are often used by the literature in different ways. To alleviate some of the confusion, we define several of these terms here, and use these definitions when discussing the other work below.

traditional: the traditional file system may use multiple disks, but places each file entirely on one disk. The disks may be controlled by a single controller or multiple controllers.

declustered: the blocks of a file are scattered among the disks. The disks are accessed independently, though they may be connected to the same disk controller. This does not imply interleaving (see [Pie89] for a non-interleaved declustered approach).

interleaved: this refers to the way the blocks of the file are partitioned among the disks. The blocks are allocated to the disks in a round-robin fashion, the first block on the first disk, the

[1] We do not consider multiple-arm (see [Smi78a]) or multiple-head disk drives in this discussion, as they still depend on a single controller, something we would like to avoid. The importance of avoiding such a bottleneck is emphasized by the results in [NLS88].

next block on the second disk, and so on.[2] This is a special case of declustering, and does not imply striping.

striped: the blocks of the file are interleaved among the disks, and the disks are controlled by a single controller, which reads a block from all disks simultaneously. Each disk may contribute as little as one bit at a time. There are two varieties, depending on whether the disks are rotationally synchronous. This is a special case of interleaving.

An independent notion that may be combined with all of the above except striping is

parallel, independent disks: the disks are completely independent, having separate controllers and paths to memory, and presumably connected to separate processors.

A wide range of different arrangements, including striping, declustering, and combinations, was studied by Reddy and Banerjee [RB89a, RB89b]. They found that interleaved systems are more scalable, and more appropriate for scientific applications, whereas synchronized striping is better for general file system workloads.

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 [SGM86, Kim86a, Ng89]. The performance improvement of disk striping (in uniprocessor systems with current technology) is limited to a small number of disks (less than five), according to simulations reported in [GMS88]. This is due to the increasing overhead involved in the serial management of the I/O parallelism. As the access-time gap between memory and disks increases, they predict that the performance improvement may be extended to larger number of disks.

Mirrored or shadowed disks are another form of I/O parallelism, which are intended mostly to improve reliability but which can also boost performance [BG88, Bit89].

The RAID group at Berkeley [PGK88, PGK88, Sch88, GHK+89] studied large, tightly coupled arrays of small, inexpensive disk drives. They carefully designed their system with redundancy in mind, since a highly parallel array of disks is too unreliable without some built-in redundancy. Their design is experimentally evaluated in [CGKP90], concluding that a striped-disk system with a rotated parity scheme (RAID level 5) is better than mirrored disks (RAID level 1) for workloads with large (1.5 MByte) transfer sizes (e.g., scientific workloads).

Another possible parallel disk architecture is based on the notion of parallel, independent disks, using multiple conventional disk devices addressed independently and attached to separate processors. This arrangement is different than that in most distributed systems since a file may be spread over several disks, and thus the disks are more logically related. 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. Systems of this sort were proposed by Flynn [FH88] and Reddy [RBA88] for hypercube-connected multiprocessors. Both researchers propose special I/O processors connected by a separate network, and concentrate on the processor layout to minimize communication delays. This plan was implemented in the Intel iPSC/2 multiprocessor (see page 7).

The Gamma project [DGS+90] built a database machine that incorporated parallel, independent disks with different types of declustering. The first version used a network of 20 VAX minicomputers and the second version used a 32-node Intel iPSC/2 hypercube multiprocessor. Another database machine that proposes the use of parallel independent disks is Bubba [BAC+90]. This machine will

[2] Other granularities are possible, e.g., at the record, byte, or even bit level. Thinking Machines' Data Vault interleaves by bit [TMC87].

have hundreds of independent "intelligent repositories", each with processing and storage capability. This project is paying special attention to the placement of data in the system to minimize communication and to balance the load between the processors, and to best take advantage of the disk buffers.

Bridge [Dib90, DSE88] is a file system developed for the BBN Butterfly parallel computer that interleaves individual files over several independent (simulated) disks associated with different processor nodes. The file system allows naive programs to access the files as they might in a traditional file system, but makes no special efforts to fully drive the available disk parallelism. An alternative interface allows computations to take advantage of the hardware parallelism and locality by having the file system dole out blocks from each disk to the process running on the corresponding processor, so all processes always operate on the "nearest" disk blocks and do not contend for the same disk drive. Dibble [Dib90] also discusses resiliency, maintenance, and portability.

2.1.1 Commercial Products

Several companies have added some form of parallel I/O hardware to their system. An overview of the commercial disk-array situation is presented in [Man89]. Digital Equipment has a striping driver available for its VMS operating system [DEC89]. Sun Microsystems provides a "MetaDisk" service that can mirror or stripe across the partitions of a set of disks [Tab90]. Cray Research offers a disk subsystem that uses hardware to combine four disk spindles into one logical disk drive with a sustained transfer rate of 9.6 MBytes/sec [Res90]. Micropolis Systems makes a striped system with 4 disks and one parity disk that is byte-interleaved, with 1.5 GByte capacity, sustained transfer rate of 4 MByte/sec, and MTTF of 140,000 hours [Mok87]. Encore [KBK89] and Sequent [Hai89] have parallelized a traditional Unix[3] file system, but their disks are limited (as are their processors) to a single shared path to memory. Symult Systems [Sym89] has a traditional Unix file system on several disks accessible in parallel by attaching disks directly to processor nodes. Computer System Architects builds a 64-transputer mesh-connected multiprocessor with embedded I/O nodes [M.88], though it is not clear what file system model will be used. Teradata builds database machines with disks accessed through their specialized Y-net, which selects the appropriate disks and merges results from all the disks to satisfy queries [Ter88]. The IBM 3390 disk cache [MH88] is a combined controller and cache that supports up to 64 drives and up to 16 channels from the processor. Redundancy in the controller provides for reliability and increased performance. This system is intended for uniprocessors. Maximum Strategy markets a disk subsystem that stripes 4–8 disks, has up to 4 channels, and up to 4 hot spares [Mea89]. The Connection Machine DataVault [TMC87, TR88] uses a bit-interleaved, striped set of 32 disks and 7 parity disks. Up to eight DataVaults may be attached to a Connection Machine-2, each with its own controller, and each maintaining 40 megabytes per second throughput to 10 gigabytes capacity.

Intel developed an I/O architecture for their iPSC/2 multiprocessor based on dedicated I/O nodes, each with a separate controller and Small Computer Systems Interface (SCSI) bus [Int88a, Int88b]. The maximum configuration is 127 I/O nodes and 889 disk drives with 128 compute nodes. Their Concurrent File System [Pie89, AS89] provides a transparent interface to the multiple disks, treating them as a single logical disk. The performance of this file system is examined in [FPD91]. It automatically declusters the file among the disks (allocating blocks to the file from any disk) and routes block requests to the appropriate disk. The interface is the standard Unix interface, with a few extensions to allow asynchronous I/O and to control concurrent file access. Other examples of I/O architectures that are based on dedicated I/O nodes are the NCUBE (compared with the iPSC/2 in [PFDJ89]), the proposed BBN Monarch [RCCT90], and the IBM Victor [WSB+89].

[3] Unix is a registered trademark of AT&T.

None of these systems have made any significant effort (such as the use of sophisticated caching and prefetching) to help parallel programs use parallel disks efficiently. Intel uses simple caching and prefetching in CFS, but it is not clear from the literature what range of workloads or performance could be supported by their file system [FPD91].

2.2 Caching

Early operating systems relied on user-controlled *buffering* for high-performance disk I/O. A good discussion of buffering techniques, including careful overlap of computation and I/O, is found in [FP77] (pp. 164–173). This method depends on the user (or the run-time system) having total control of the disks, channel, and processor for careful scheduling. Recently, it is common to use automated *caching*, where blocks from the disk are kept in memory as long as they are useful, and flushed from the cache when room is needed for other blocks.

Alan Smith has extensively studied disk and memory caching in uniprocessors. In [Smi85b], simulations of disk caching show that disk caching is an effective way to boost the performance (as measured by the cache miss ratio) of the I/O subsystem (e.g., an 8 MByte cache can service 80–90% of I/O requests). Smith found that an in-memory cache (as opposed to caches at the disk devices or controllers) had the lowest overall miss ratio. Since it is also closest to the users of the data, main memory is the most effective position for the cache. Due to the variety of disk file access patterns, he recommends that the cache dynamically monitor its own miss ratio and use caching and prefetching only when the miss ratio remains low.

Cache management overhead is significantly reduced by using memory-management hardware to manage the disk cache without changing the interface [BRW89].

Stonebraker [Sto81] discusses disk-caching support for database systems. In particular he examined replacement algorithms, recognizing that least-recently-used (LRU) is not necessarily the best policy since much access to databases is sequential. In addition, program-controlled prefetching is an important capability for the support of database applications, as the database manager often knows in advance what block it will access next, even if the access is not sequentially related. It is clear that cache management policies for random access patterns and sequential access patterns are likely to be different, and thus Stonebraker suggests that the application have some of control over the policies. Tokunaga *et al.* [THY80] implemented a disk cache with this in mind, allowing the application to specify the type of access pattern it would use.

Modern operating systems commonly use file caches. The Unix operating system has included a file cache from the start [RT78]. A non-Unix example is the IBM VM/XA disk cache [Boz89]. File caching is also important in distributed systems, such as Sun's Network File System [SGK+85], Sprite [NWO88], and Amoeba [TvRvS+90].

2.3 Prefetching

The central idea behind prefetching is to overlap some of the I/O time with computation by issuing I/O requests before they are requested. Trivedi [Tri77a] points out that the best reduction one can hope for in execution time is 50%, if all of the I/O and CPU time perfectly overlap. Thus programs that have a good balance of CPU and I/O time have a better potential for improvement from prefetching than those that are more CPU-bound or I/O bound. With parallel disk hardware, however, we expect prefetching to also overlap I/O with I/O, obtaining even larger benefits.

Smith [Smi81b] mentions that prefetching is especially valuable for supercomputers since the level of multiprogramming often cannot be increased to cover the idle time caused by I/O delays as in

traditional multiprogrammed uniprocessor systems. This is evident in the processor-scheduling concepts generally used with the operating systems for the BBN Butterfly Parallel Computer [BBN87], in which each processor usually has only one user process running on it.

2.3.1 Prefetching in Disk Caches

Smith has studied two simple prefetching strategies in a uniprocessor file system. Both depend on an assumption of sequentiality but neither studies the reference string in detail to dynamically adjust its behavior. The first, *One-Block Lookahead* (OBL), is described in [Smi85b, Smi82, Smi78b, Jos70, BS76], and involves prefetching block $i + 1$ when block i is accessed, if it is not already in the cache (these block numbers are physical disk block numbers, but the concept is the same for logical block numbers as well).[4] In either case it is marked as most-recently-used for the use of the LRU replacement algorithm. This technique has been successful in reducing the miss ratio in disk caches for batch files and temporary files in uniprocessor simulations by up to 80% [Smi85b].

Smith's second technique, described in [Smi78c], assumes a strong degree of sequentiality. Using knowledge about the various costs of I/O system activities, the length of the current run (a *run* is a string of accesses to consecutive blocks in a file), and the static distribution of run lengths, the strategy prefetches a number of blocks ahead. As the currently observed run length grows, blocks are prefetched increasingly far into the future. He found this to be more effective than prefetching a fixed amount ahead, and reduces the cache miss ratio by about 40% in a simulation on a database workload. He has also found that prefetched blocks should be treated the same as demand-fetched blocks with respect to LRU replacement, as variations in the replacement priority do not seem to affect the miss ratio. His results are based on the assumption that fetching multiple contiguous blocks does not take much more time than fetching a single block, which is true for files that are stored contiguously on the disk.

Smith's simulations in [Smi85b] show that prefetching is more beneficial for some types of files than for others, corresponding to the strength of the assumption of sequentiality for each type of file. The best improvements are for batch files, batch users, and temporary files, since they tend to be sequentially accessed. No improvements are possible on paging data sets, since they have little locality. Thus, Smith suggests that caching and prefetching be used selectively, depending on file type or, ideally, on hints from the user.

Earlier work by Ragaz and Rodriguez-Rosell [RRR76] examined reference traces of segments in a database system. They found strong sequentiality in the references and, in turn, that prefetching improved the average segment access time. Their strategy prefetched several segments ahead of each cache miss. Indeed, blocking several segments into a disk block, and then demand-fetching disk blocks, was nearly as effective as prefetching at the segment level. A second strategy adaptively discovered more complex sequential patterns, such as reading every other segment, but provided little extra improvement for the effort. Another adaptive prefetching strategy is described by [FB76].

Prefetching is included in the design of the MPE XL Data Management System for the HP Precision [Kon88], a memory-mapped file system. The prefetching algorithm uses a heuristic to determine the amount of data to prefetch that depends on the access pattern, the file consumption rate, the fault rate, any access hints, and memory availability. A similar heuristic is used for the write-back algorithm.

Powell [Pow77] proposed a file system for DEMOS, an operating system for Cray-1 supercomputers. The file system is oriented around a small disk cache (10–20 blocks of 4 KBytes each), and uses two basic strategies: when reading sequentially it prefetches blocks to keep the buffers as full as possible and when writing it allocates several contiguous blocks on the disk well ahead of the

[4] OBL on logical file blocks was used in the original Unix file system [RT78].

program to reduce allocation overhead and external fragmentation on the disk. The latter strategy also tends to avoid seeks when accessing the file sequentially.

Bennett and May [BM81] define a fairly sophisticated implementation of prefetching, similar to that in [RRR76], in a disk controller cache. The controller they propose maintains a table of recent disk accesses, separating the accesses into separate sequential streams, and prefetching entire tracks when the reference stream appears to be sequential. The significance of this approach is its hardware implementation and its ability to automatically monitor several separate reference streams.

Prefetching in a multiprocessor file system (Intel's CFS) was studied briefly in [FPD91]. Each I/O node prefetches several blocks ahead when access appears sequential. This strategy gave a 217% improvement in throughput when a single process read an 800 KByte file sequentially from a single disk, compared to random access to the same file. Their study does not directly evaluate prefetching in a multi-process, multi-disk situation. Multiprocessor prefetching was also examined by Towsley [Tow78, TCB78]. This study argues that multiprocessors are likely to benefit from overlapping I/O with CPU time more than uniprocessors, and that prefetching looks promising for large numbers of processors. In particular, overlapping I/O and CPU time is especially useful when the degree of multiprogramming is low. Without this constraint, adjusting the multiprogramming level may often be used to provide the necessary overlap to keep the CPU and I/O devices fully utilized. The emphasis of this study seems to be on multiprocessors operating on independent tasks, whereas we study programs that use many cooperating processes to accomplish one task.

2.3.2 Prepaging in Virtual Memory

Trivedi studied prepaging in a uniprocessor virtual-memory environment in [Tri76, Tri77b]. He describes algorithms for prepaging that are optimal with respect to the number of page faults incurred, with a correspondingly significant reduction in execution time for programs using the algorithms. These algorithms depend on the ability of the programmer, operating system, or compiler to accurately predict the pages that are no longer needed and those that will be needed in the near future. Thus, these are essentially replacement algorithms that decide which pages to keep, which to discard, and when to prefetch, based on the predictions supplied by another mechanism (e.g., the compiler). These techniques are successful for prepaging data pages in array algorithms. In [Tri79] he exploits this technique to attempt to reduce the number of page faults during transitions between phases of program execution.

More generally, Smith examined prefetching with OBL in memory hierarchies [Smi78b]. With no hints from the compiler, as with Trivedi's technique, prefetching is not helpful except for small page sizes (about 32 bytes). This small size is due to the relatively small scope of sequentiality in program instruction and data access patterns. Smith concluded that prefetching is most effective for CPU cache memories, not at the level of virtual memory paging.

Baer and Sager [BS76] attempt to improve the predictive capabilities of OBL by dynamically maintaining an association between page numbers. That is, if page j is the first page to fault after a fault to page i, then j will be prefetched whenever a fault occurs for page i in the future. (Pure OBL associates $j = i + 1$ with i). Their simulations showed that this technique predicts the next page more accurately than OBL and reduces the number of page faults. They then go further by reordering pages in a two-level memory hierarchy, grouping a few small pages of primary memory into large blocks in the secondary memory. The grouping is based on the LRU ordering of pages at the time of page replacement. This permits an elegant preloading of several related pages on each fault while placing seldom used pages together in separate blocks.

2.3.3 Prefetching in Memory Caches

Prefetching is effective for high-speed memory caches in uniprocessors [Smi78b, Smi82]. In this instance only simple prefetching strategies like OBL are feasible, due to the hardware implementation and the high speeds. The small line size and a relatively larger scope of sequentiality, particularly in instruction streams, was shown by Smith [Smi78b] to be important for effective prefetching. In addition, the simplest strategy appears to be superior: initiate a prefetch on every access if the successor is not already in the cache, as opposed to initiating only on cache misses or more complex options. Smith also compares 49 different traces from 6 different machines in [Smi85a], and finds that prefetching is always successful (though to varying degrees), reducing the number of cache misses by about 50%.

Caching and prefetching were also studied for shared-memory multiprocessors [LYL87, Lee87, Mar88]. One of the key points of their research is that the miss ratio is not an adequate measure of cache performance in multiprocessors with pipelined interconnection networks, due to the overlap of miss service times. They used one measure based on the overall execution time [Lee87], and another multiplying the miss ratio by the standard deviation of inter-miss times [Mar88]. They found that caching effectively counters low-bandwidth problems, and prefetching effectively masks the high latency memory. The combination essentially eliminates the problem of high-latency shared memory. The primary flaw in these works (especially [Mar88]) is that they did not consider contention at the memory modules or in the interconnection network.

2.3.4 Prefetching Summary

In short, previous work in prefetching has centered in two different areas: 1) prefetching instructions and data in memory hierarchies, either between a CPU cache and main memory or between main memory and disk (prepaging virtual memory) and 2) prefetching blocks of a file in a disk cache. Most techniques depend on sequential reference patterns. Sequentiality is the simplest access pattern, and choosing blocks whose size is on the same order as the sequentiality of the access (a few tens of words for instructions and data, thousands of words for disk files) seems to be important. Most results have used simulation or modeling to show a reduction in the cache miss ratios of roughly 10–60%, which occasionally extend to increases in system performance of about 10–20%.

2.4 File System Workload

In order to design caching and prefetching techniques for parallel file systems, we must have an idea of the workload expected in such a file system. This is necessary to tailor the design to the workload and to generate synthetic workloads for simulation. In addition, to do prefetching, we must be able to predict future accesses, which requires a good understanding of access patterns. Smith has studied the effect of the choice of workload when evaluating memory caches and prefetching [Smi85a], and found a wide variation in the miss ratios and prefetching effectiveness depending on the workload and architecture generating the trace for the simulation.

A common way to view file system workloads is to break them down into three classes [OD89]:

scientific: This workload is characterized by sequential access to large files. High transfer rates for relatively few large transfers is important here.

transaction processing: Including database systems, this workload typically involves a large number of short requests to independent portions of the database. Individual transactions are limited by disk latency, so the *overall* I/O throughput (in accesses per second) is an important metric.

engineering/office: This workload (which might also be called *general purpose*) is comprised of programs that make a large number of small requests, but with less concurrency than in transaction systems. Here, the speed of the individual application is important, emphasizing the latency of individual disk operations.

We expect the first two to be the most likely candidates for parallel file systems, and these are the workloads emphasized by other researchers [PGK88, PGK88, RB89a]. We concentrate on scientific workloads.

File access patterns have been studied extensively for uniprocessors [Flo86, OCH+85, FE89]. Floyd [Flo86] 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 ([OCH+85]) found that 90% of all files are processed sequentially, either through the whole file (70% of all accesses) or after only one seek. A survey of 47 large IBM mainframe installations found that 67% of the files in the file system were sequential files, although these accounted for only 39% of the disk space ([BSTY78], summarized in [Smi81a]). A study done by Powell [Pow77] on a Cray-1 file system found a similar pattern, with most files small, but the large files occupying most of the disk space. Powell's data and the IBM survey reflect a static file system and do not include the frequency of use or any other dynamic measure of file usage.

Parallel file access is discussed more generally by Crockett [Cro89]. No actual workload was studied. Instead, file access patterns are related to possible storage techniques. He defines a few basic file access patterns, which are either sequential or random in nature. There are four sequential patterns: pure *sequential* (no parallel access), *partitioned*, in which each process reads a separate portion of the file, *interleaved*, in which the processors read the file in a globally sequential fashion but individually read single blocks with regular stride, and *self-scheduled*, similar to the interleaved pattern but with an unpredictable stride (i.e., the processes each read the next unread block). The random access patterns are either truly random or *partitioned*, with each process working in a separate area of the file.

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. The nature of parallel file access patterns is different than that of uniprocess access patterns, and thus we must sometimes look for sequential access in different ways. We further describe our expectations for parallel file access patterns in Section 3.1.

2.5 File System Interface

Several researchers have discussed parallel I/O interfaces for MIMD multiprocessors. The Bridge file system [Dib90] has three interfaces that range from a sequential compatibility interface to a highly-parallel low-level interface that emphasizes locality. Intel's file system for their iPSC/2 multiprocessor supports both the standard sequential interface and two types of simple parallel access patterns [AS89]. Crockett [Cro88] has a standard sequential interface, support for self-scheduled patterns, and independent parallel access by all processes. The CUBIX file system (for hypercubes) connects a sequential file server to a parallel application program [FJL+88].

Some systems provide hints to the file cache based on file type, which is determined from the file name [Kor90], access mode [THY80], or system administrator [Gro85]. Some systems allow the user to provide hints. An example is file preallocation in Intel's CFS [AS89].

For more details on related work in file system interfaces, see Section 10.4.

Chapter 3

Models and Assumptions

In this chapter we present some basic assumptions that underlie our work, along with the models we have chosen for the workload, processor and I/O architectures, and file system control. The bulk of the chapter defines the type of workload we expect in a parallel file system, since its understanding is critical for caching and prefetching policies. Chapter 4 describes our testbed and the details of its policies, implementation, and experimental parameters.

3.1 Workload

Our efforts are geared toward high-performance multiprocessor computers running scientific applications. In other words, we do not consider general-purpose or transaction-processing workloads. We use the model of a single parallel program running on an MIMD parallel processor and consisting of a set of separate processes, each with a dedicated processor node. There is no multiprogramming within a particular processor node (except perhaps with operating system processes). The processes cooperate to access huge data sets, one file at a time.

3.1.1 Parallel File Access Patterns

We must be able to predict the future file access patterns in order to successfully prefetch disk blocks. To predict file access patterns we must first understand the patterns we may encounter. In our study we choose to examine the access pattern within each file accessed by a particular parallel program, rather than the access pattern to a particular disk or other system-wide alternatives. We feel that the sequentiality of access is most evident at the level of the individual file, particularly if the files are not stored contiguously on the disk. Thus our techniques examine sequentiality in the accesses to the (logical) blocks of a particular file, rather than to (physical) blocks on a particular disk. This is in sharp contrast to most previous work on prefetching and disk caching, which expects to find sequentiality at the physical block level. This policy affects the structure of our disk cache. Most disk caches are oriented to caching physical disk blocks and have little or no understanding of the underlying file system or process activity. Our system will employ a separate cache for each open file, which is shared by several cooperating processes. The caching (and prefetching) decisions depend on the activity of that set of processes alone.

We assume that the programmer uses a conventional *open, close, read, write,* and *seek* interface to interact with the file system. The front-line interface converts application read and write requests into requests for individual blocks of the file. Note that we distinguish between file *blocks*, which correspond to the storage of the file in fixed-size units usually related to the disk's sector size, and *records*, which are the logical units used by the application. We assume that the file-system

internals see only the *block* access pattern from an application, and thus all caching, prefetching, and I/O is done in *blocks.*

The file access patterns we see in a multiprocessor file system are likely to be different from uniprocessor file access patterns. Although file access patterns in uniprocessor systems are well-understood [Flo86, OCH+85, KD89, FE89], 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 is to provide a long-term repository for data; growing main memory capacities will reduce the need for most temporary files to be placed on disk. The nature of parallel scientific applications tends to be different than uniprocess, general-purpose workloads. Thus, the files stored on a multiprocessor 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). Parallel file systems and the applications that use them are not sufficiently mature for us to know what patterns might be typical. Parallel applications may use patterns that are more complex than those used by uniprocess versions of the same application. However, a few basic patterns may be defined and we shall see that many likely parallel algorithms using disk I/O fit into these patterns.

Types of Access Patterns

In our research we consider only read-only and write-only access patterns. We do not investigate read/write file access patterns, because we believe that most files are opened for either reading or writing, with few files updated [Flo86, OCH+85]. We expect this to be especially true for the large files used in scientific applications ([Pow77, Ber90]): one or more large input files are read, and one or more large output files are written. There may also be intermediate files that are written and then later read. When a file is only written, it is conjectured that it is usually either completely written or appended [Flo86]. In either case, the writes are to a "new" part of the file, not overwriting old information. This is important when the application process writes only part of a block. If the block already exists on disk, the block must be read into the cache before the buffer can be written. If it is a "new" block, not on disk, then the buffer can be initialized with zeroes, avoiding the disk read. These disk reads can also be avoided when the application makes many small writes that eventually overwrite an entire block. This is easy on uniprocessors, but parallel access patterns make this more complicated. In all of our experiments, we assume that the file being written is truncated to zero length and re-created when opened, so no disk reads should be needed to write the file.

Read-only access patterns: We assign all read-only file access patterns to one of two categories, *random* and *sequential* (see Figure 3.1). Random patterns are defined to be those that do not fit into the sequential models.

All sequential patterns consist of a sequence of accesses to sequential *portions* (*runs* in [Smi78c]). A portion is some number of contiguous blocks in the file.[1] 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. The sequential portions of the individual processes may or may not overlap each other; any overlap has implications for the cache replacement algorithm, as the blocks in common are used more than once each.

[1] Actually, a portion is originally some number of contiguous *records*, as read by the application. The file system, however, works with the resulting block access pattern, rather than the original record access pattern. We thus define portions here from the perspective of the file system, in terms of blocks rather than records.

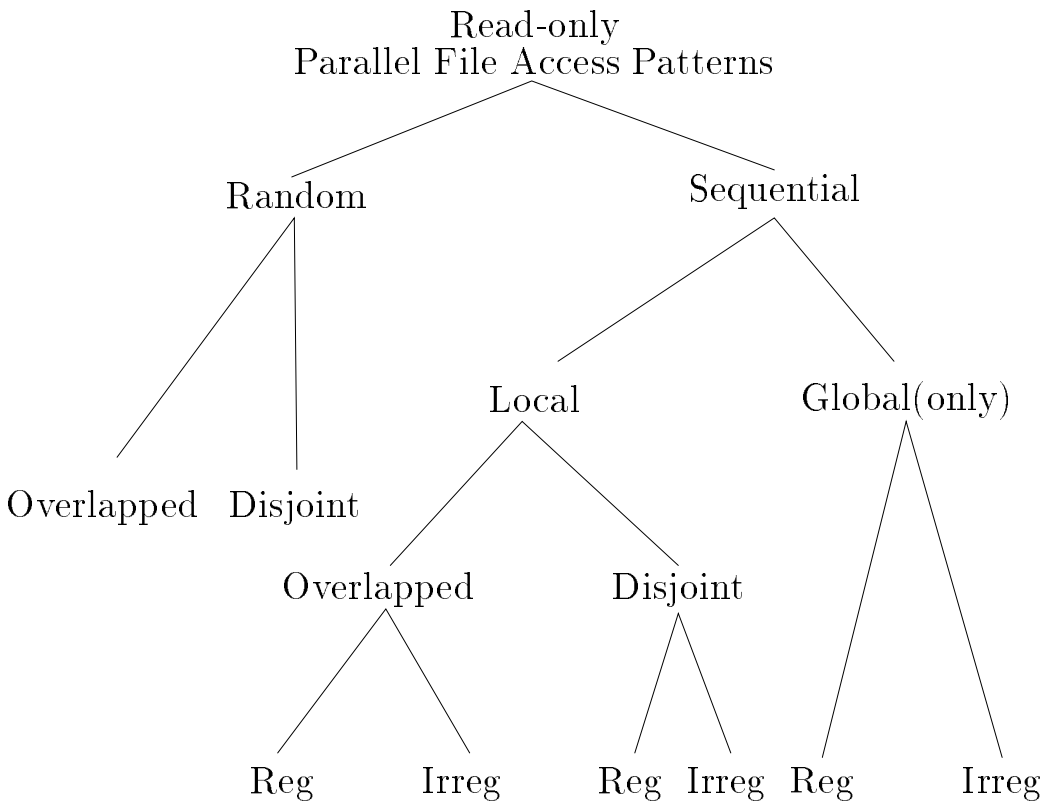


Figure 3.1: Categories of File Access Patterns

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. If the nature of the sequentiality of a local or global pattern can be detected, more sophisticated prefetching than just blindly prefetching the next block may be possible.

In addition, the length of portions (in blocks) may be regular, so the file system may be able to 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 stride), allowing the system to prefetch the first block of the next portion. In the chart we refer to these as regular sequential portions ("Reg") and others as irregular ("Irreg").

The record size or buffering at levels above the file system can transform one pattern into another. For example, a globally-sequential whole-file access pattern with a fixed record size may appear to be an *interleaved* pattern, if the accesses are regular. An *interleaved* pattern (not to be confused with interleaved disks) is a special case of a local pattern with regular portions. A globally-sequential whole-file pattern may also appear to be a locally-sequential whole-file pattern, if the record size is much smaller than the file system's block size. To the file system, it appears

that each process is accessing every block. Although one pattern is similar to another, the patterns are not the same. First, self-scheduled access patterns may sometimes seem interleaved, but due to variations in record size, computation time, contention, *etc.*, strict interleaving may not last for long. Second, a higher-level understanding of the pattern allows for more successful prefetching. Thus, it is important for the file system to understand both local and global patterns.

Note that each of Crockett's proposed access patterns ([Cro89]) may be described by our model. He gives two random access patterns that are exactly represented by our random categories; the *partitioned* random pattern, as a refinement of the random pattern, has little implication for prefetching. It may, as he points out, determine the layout of the file on the disks. His pure *sequential* and *partitioned sequential* are special cases of sequential portions, as a sequential portion may be any part of a file, including the entire file. The *interleaved* pattern is a case of regular sequential portions that happen to be non-overlapping and cooperating to form a larger global sequential portion. Finally, the *self-scheduled* access pattern is essentially our globally-sequential whole-file pattern. Our categorizations go a step further by generalizing to the concept of sequential portions and to the concept of local and global views of the access pattern.

Write-only access patterns: We expect the class of parallel write-only access patterns to be more limited than parallel read-only access patterns. For example, we expect that every byte of the file is written exactly once, whereas in a read-only pattern some bytes may never be read and some may be read many times. For the sequential patterns we expect to be most common, it is unlikely that some parts of the file are not written (i.e., written zero times) or that some parts are rewritten (i.e., written more than once). As with read-only patterns, we expect to see both local and global sequentiality.

Examples of Parallel File Access Patterns

We claim that our basic definitions of access patterns are sufficient to describe all read-only and write-only file access patterns. Later we show the implications of these patterns on prefetching decisions. First we examine some reasonable scenarios and some examples from the literature that support our claim that these categories are interesting ones that may actually be seen in real programs.

It is not hard to imagine programs that read an entire file sequentially; in fact we expect this to be the common case. In parallel programs, however, there are many variations on this activity, in addition to one processor reading the whole file by itself. Each processor may read the file entirely on its own, perhaps performing different computations on the data in the file. Both this pattern and the single-process pattern are special cases of overlapping local sequential portions. Alternatively, all processors may cooperate to read different portions of the file in some manner. This might be done when the file consists of a set of records that may be processed in parallel, perhaps selecting some subset of records that match some criteria. If the processors sequentially read separate partitions of the file, it is an example of disjoint local sequential portions; if they intermingle in the whole file, it is a global sequential portion.

The processes may read only a part of the file, perhaps a single row or set of rows of a large matrix mapped into the file. In this case, they are accessing some sequential portions of the file, either randomly or at regular intervals depending on the algorithm. If the matrix is stored as a dense matrix, the portions are equally long; if stored as a sparse matrix, the portions tend to be of different lengths. Both of these patterns would be used by a local researcher [Pan89], given access to parallel I/O.

Three papers [BBW86, BKZS85, DO86] regarding parallel external sorting and merging of extremely large files show that these types of operations use sequential access to files, usually in non-overlapping local sequential portions.

In her thesis, Kim [Kim86b] describes a parallel method for computing large (larger than main memory) Fast Fourier Transforms (FFT) using a synchronously striped disk system. The data is stored and accessed in sequential portions that are optimized for use with her striped disk system.

One supercomputer program for calculating the electronic structure of molecules requires several gigabytes of scratch files [Ber90]. Several different access patterns appear in the use of these files. All writing is sequential. Some of the files are read sequentially from start to finish, sometimes by a single process and sometimes by several processes in cooperation. Some files involve locally-sequential portions, with all processes reading random portions of the file. Finally, some of the patterns are purely random. The record size varies from 12 bytes to more than 24 KBytes. All of these access patterns are represented in our categorization.

Local developers of VLSI tools note that VLSI applications often use large data files. The files tend to be read sequentially and completely into data structures in virtual memory, where the data is manipulated and then sequentially written back to disk [ABS].

A program for pattern matching in a gene database matches a given gene against every gene in the database [Die89]. The database is about 50 MBytes compressed. The order of the search is irrelevant, but the record length varies. This is a global pattern with variable record size, or a locally-sequential pattern with variable-length portions.

In light of these examples it appears that parallel file access fits into a few basic categories, several of which may facilitate prefetching.

3.2 Processor and I/O Architecture

The architecture on which we base our research efforts is a multiple instruction stream, multiple data stream (MIMD) multiprocessor. In addition it is a non-uniform memory access (NUMA) architecture, in which each processor has its own memory that is also accessible by all other processors (see Figure 3.2). The shared memory assumption is not a key component of our architectural model, but is critical for our testbed implementation. We believe that this class of architecture effectively scales to a large number of processors and disks.

3.3 I/O Architecture

We represent the disk subsystem with parallel, independent disks (Figure 3.2). There are multiple conventional disks, each connected via a separate controller and channel (or bus) to a separate processor, allowing each to be independently addressed. Indeed, with the NUMA architecture each disk has an independent path to memory. This allows for completely parallel access to all disks. Thus the time for a disk access is strictly dependent on the state and activity of that disk and memory alone. This is in contrast to a conventional uniprocessor where many disks may be connected to one processor, sharing some channels, disk controllers, string controllers, and memory for buffers. It also contrasts with striping systems where parallel disks are joined by a single controller.

We define the access time of a single disk request to be a combination of the physical access parameters of the disk and any contention for the disk. For simplicity, we use a constant physical access time for each disk, allowing us to ignore the details of the layout on each disk. In other words, we do not assume that the blocks of the file are contiguous, which typically reduces seek

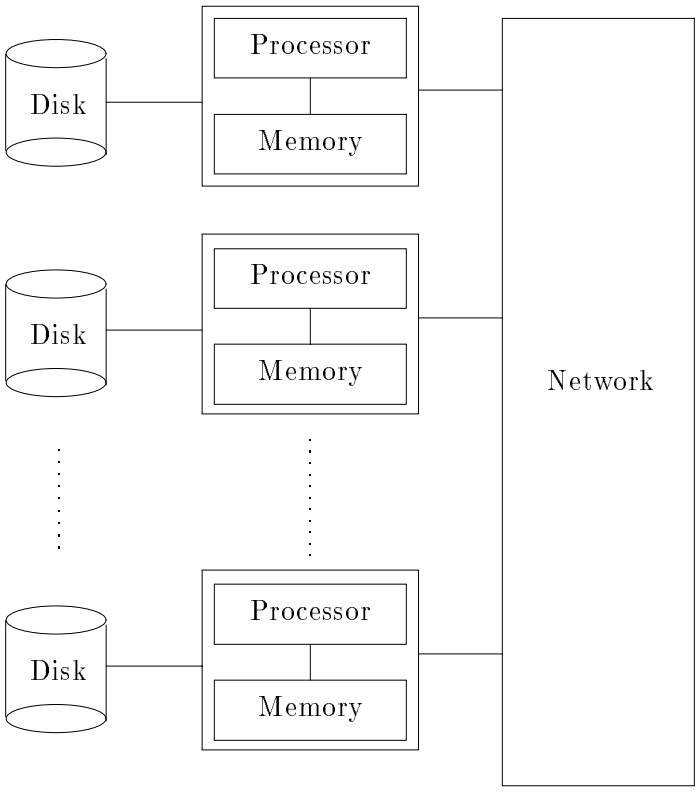


Figure 3.2: Parallel Independent Disks on a NUMA/MIMD Architecture

time for sequential access. A given disk access only contends with other accesses to the same disk, reflecting the physical independence of the disks. The assumption that every processor has a disk is used for the bulk of our experiments, but we vary this in Sections 8.4 and 8.5.

There are several possible ways of mapping files to multiple disks, including placing each file entirely on one disk, partitioning each file among disks (by placing a contiguous segment of the file on each disk), or declustering each file among disks (by scattering blocks or bytes of the file across the disks in some manner). We assume an interleaved structure, with blocks of the file allocated round-robin to all disks in the system. This is a straightforward declustering technique that is easy to calculate, distributes the blocks of the file roughly evenly, and allows easier extension of the file than a partitioned layout.

3.4 File System Control

The file system manager is the entity, perhaps part of the operating system or perhaps a separate process (or processes), that manages the disks and all requests for I/O. We often refer to the file system manager as simply the "file system", considering the control software as a part of the file subsystem. This may be multiprogrammed on the same processor with the application program, stealing cycles and thus affecting the execution pattern of the process, or it may run separately on a dedicated I/O processor, independent of the original process. The latter model is used by several proposed systems [DHS88, FH88, WCM88, Int88b, Pie89, RBA88]. Our model is based on the

multiprogrammed approach, with a file system manager on each processor, handling all the I/O requests for that 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. In either case the application is only detained (blocked) by demand fetches, when it requires the data in order to continue. Prefetching I/O requests are serviced while the application continues to run. The next chapter gives more details about the file system manager.

Chapter 4

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 shared-memory MIMD multiprocessor (BBN GP1000). Since the multiprocessor does not have parallel disks, they are simulated. Without access to a real workload (parallel applications using parallel I/O) we chose to use a synthetic workload. The synthetic workload captures such nuances of real workloads as sequentiality, regularity, and inter-process interactions. These characteristics, all important to caching and prefetching, are easy to incorporate in a synthetic workload but would be difficult to include in an analytical model. Thus, we have simulated disks and a synthetic workload, and a real implementation of the file system on a real multiprocessor. We run our synthetic workload through the testbed, 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. Because of the wide variation in multiprocessor architectures, we cannot claim that this architecture (or its overhead) is typical in any way. It is, however, a direct measure of real overhead, which is better than simulated overhead, since it responds dynamically to different conditions and does not mistakenly avoid unexpected sources of overhead. This is important when beginning to study an area that is not well understood, such as parallel file systems, since there are many of these unexpected sources of overhead. An implementation helps to explore the new area without missing these unexpected effects. In this case, it also evaluates whether *practical* prefetching policies can be implemented efficiently.

In some cases, we compare the measurements with trivial analytic performance models to gain an insight into the behavior of the system, and the overhead imposed by the system. These models work well for this purpose.

4.1 The RAPID-Transit Testbed

The RAPID-Transit testbed is a parallel program implemented on a BBN Butterfly GP1000 parallel processor [BBN87], originally derived from the BBN *RAMFile* system [BBN86]. The Butterfly GP1000 is a NUMA MIMD machine in which all memory resides with the processor nodes (see Figure 3.2), but is accessible from all other processors through a form of a log-depth Omega ("butterfly") network.

The testbed is a set of software routines designed to provide a simple file system interface to a set of (simulated) disks, allowing us to experiment with different techniques for caching and

prefetching. The testbed is heavily parameterized and fully parallelized. It incorporates both the synthetic workload (the application) and the file system (interface and manager). The file system allocates and manages a buffer cache to hold disk blocks, attempting to prefetch blocks while the application is blocked. Finally, the testbed gathers statistics on many aspects of the performance of the file system.

Figure 4.1 outlines the basic structure of the testbed as a collection of software modules. The application reads or writes records of the file by making requests through the file system interface. The interface breaks record requests down into requests for disk blocks. It requests each block, in order, from the cache manager. The cache manager either finds the block in the cache (a *cache hit*), or must read the block from the disk into a free buffer (a *cache miss*). If necessary, a block may be removed from a buffer in the cache, first being written back to disk if it is dirty. Writes into the cache need not immediately trigger a disk write. Section 4.1.1 gives more details about cache management.

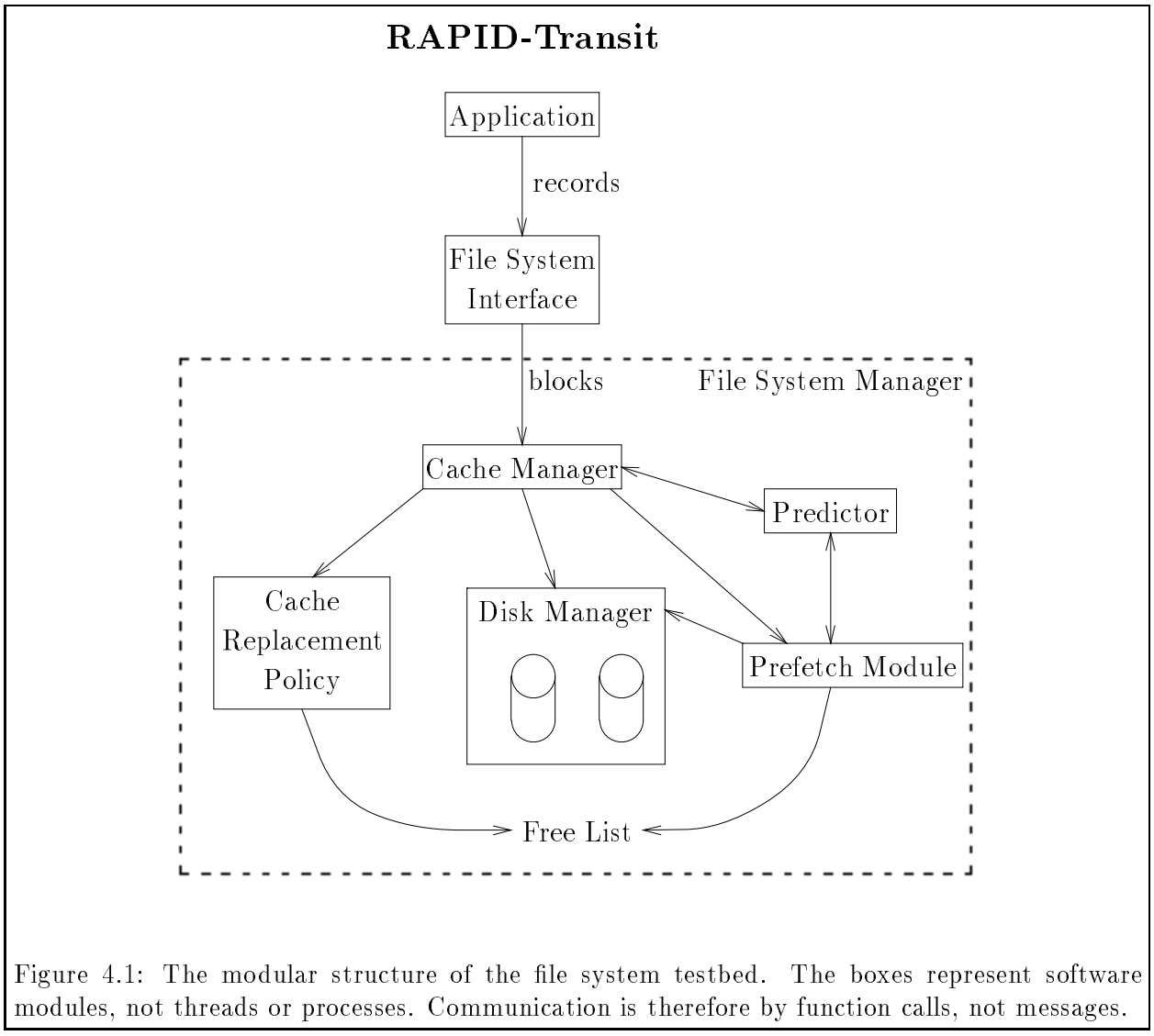


Figure 4.1: The modular structure of the file system testbed. The boxes represent software modules, not threads or processes. Communication is therefore by function calls, not messages.

Prefetching is attempted by a processor node whenever the local application process is idle. The prefetch module running on the same processor repeatedly considers prefetching, releasing control

after each action. Each time, it calls a *predictor*, which encapsulates a particular pattern-prediction heuristic. The predictor makes its predictions based on the reference history of the application.

The block number of each access is provided to to the predictor as it happens. We call this *notification*, since the cache manager notifies the predictor. The time required for the predictor to process the notification is the *notification time.* These notifications are the mechanism for supplying the reference history to the predictor. When asked for a prediction by the prefetching mechanism, the predictor provides either a one-block prediction based on the reference history, or chooses to make no prediction (sometimes the best action is no action). If the predicted block is not already in the cache, the prefetching mechanism obtains a free buffer and prefetches the block (issues a disk request). The prefetch action is *successful* if and only if it issues a disk request. A prefetch action is unsuccessful when there is a surplus of prefetched but unused blocks already in the cache, when there is a lack of blocks to prefetch (if the predictor refuses to predict any further), or when the block chosen for prefetch is already in the cache. Section 4.1.2 gives more details about prefetching.

Although the RAPID-Transit testbed was somewhat tuned during its development, it was not a fully tuned system. Our strategy was always to develop mechanisms and policies, tune through preliminary testing and experiments, and finally run a full set of experiments for interpretation. An iterative development would use the full results to re-tune and re-run the experiments, perhaps several times. We compare our policies and mechanisms as they performed on our workload, rather than tuning them to their best performance on this particular workload. We are looking for general trends and conclusions about the ability of prefetching, not the precise optimized performance of these policies on these workloads on this architecture.

4.1.1 Cache Management

The idea of a disk cache is to retain frequently- or recently-used blocks from the disk, reducing the number of actual disk accesses. We attach the cache to a particular open file, caching the logical blocks of the file rather than the physical blocks of the disk. This distinction becomes important when the blocks of a file may not be stored contiguously on the disk, since any sequentiality in accesses to the file may not be evident in the pattern of physical accesses to the disk. It is also important when files are accessed by multiprocess applications. The buffers are managed in a way that depends on all accesses to the file, not just those made by a single processor, so parallel data structures (shared by all cooperating processes) and algorithms are used instead of standard uniprocessor cache management algorithms.

The effectiveness of the cache depends on the nature of the file-access patterns in the application and on the management policies of the cache. This includes an implied dependence on one or more forms of locality. *Spatial* locality implies that the various bytes of a block tend to be used together. *Temporal* locality implies that blocks tend to be re-referenced shortly after use. In a parallel environment, there is another form of locality, *inter-process* locality. This implies that a block referenced by one process may tend to be referenced by another process. It is wise for cache policies to consider this form of locality as well.

Common file reference patterns involve sequential access to the contents of the file. Sequential access leads to strong temporal and spatial locality, since all bytes in a block are used (access to a sequence of records within a block appears as block re-use), followed by the bytes of the next block in the sequential order, and reasonably soon (at least, before any other bytes of the file). Note, however, that once access has moved on from one block to the next, the first block is not re-referenced by the same process. Thus the spatial and temporal locality applies to the bytes *ahead* of the current access. It is thus helpful to *prefetch* blocks of the file into the cache *before* they are requested, so that the data is in the cache when requested. Without prefetching, the block is

only read into the cache upon a cache miss, forcing the process to experience a delay equal to the physical disk access time, plus any queuing delays.

The replacement strategy used to manage the cache is completely dependent on the idea of locality for its effectiveness. Smith [Smi81b] found that LRU is sufficient for disk caching operations. LRU is not necessarily a good policy, however, for sequential access. Since the blocks are not used after the application moves on to the next block, a "toss-immediately" strategy [Sto81] makes more sense.

There are a variety of possible caching and replacement strategies, responding to the different issues of access patterns, physical disk model, multiprocessor architecture, and so on. In this study we concentrate on a single mechanism. We use an algorithm that is based on least-recently-used (LRU) algorithms and is specifically designed for a shared-memory multiprocessor system. In addition, it is flexible enough to handle sequential access, when LRU alone is not the best policy. A separate list is maintained for each process recording the last few blocks accessed, ordered by the most recent access to each block. This list is called the local recently-used-set, or local *RU-set*, and is fixed in size. The *global* RU-set is the union of all local RU-sets, but is unordered. This is easily maintained: a counter for each buffer containing a block of the file indicates the number of local RU-sets containing that block. A zero count implies that the block is not in the global RU-set, and is subject to replacement. No block in the global set may be replaced; this is satisfactory as long as there are enough buffers to hold the maximal global RU-set (its size being the sum of the local set sizes). We therefore require the number of buffers to be at least that large. When a process accesses a block already in its local RU-set, the RU-set is reordered so that block is the most recent. If the block being accessed is not in its local set, the least-recently used block from its set is removed from the set, and the new block becomes the most recent member. In addition, the count of the removed block is reduced by one and the count of the added block is increased by one. This scheme has the advantage that the more complex data structure (the ordered list) is maintained completely locally and without concurrency, whereas the shared data structure (an array of counters) is simple and is accessed less often. The overall data structure promises low contention and high concurrency.

As blocks leave the global RU-set they become available for replacement; that is, the buffer containing that block may be allocated to another block. These buffers are kept in a global free list. Any buffer removed from this list containing a block that has re-entered the global RU-set is not considered for replacement, and another buffer is chosen from the list. In some situations a buffer in the free list may be involved in a current disk activity. It may be a disk read into the buffer, started by a prefetch decision that was later deemed to be a mistake (page 28), or it may be a disk write used to flush a dirty block back to the disk (Section 9.2). To avoid forcing processes to wait for I/O when they need free buffers, we split the free list into two queues, one for *ready* buffers, and one for *unready* buffers. A buffer is put on the appropriate queue when it is freed. Processes remove buffers from the ready queue as they are needed. If the buffer removed is dirty (contains data not yet written to disk), the disk write is initiated, the (now clean but unready) buffer is placed on the unready free queue, and another buffer is chosen from the ready queue. If the ready queue is empty, and the unready queue contains some ready buffers, the unready queue is scanned, moving ready buffers to the ready queue, and the process tries again. Unnecessary scans are avoided by recording the earliest time when a buffer on the unready queue is expected to be ready. If there are none ready, a process in demand fetch waits (completely idle), and a process attempting a prefetch gives up and tries again later. Thus, a prefetching action never waits for I/O, although it may scan the unready queue (which is to the benefit of all processes).

The choice of the local RU-set size depends on the access pattern, if known. For sequential access patterns, a size of one is all that is needed, implementing "toss-immediately." In fact, the RAPID-Transit testbed treats an RU-set size of one as a special case, using a toss-immediate

mechanism directly. In this case the global RU-set is the set of buffers currently in use by all processes. For some random access patterns, where a block may be used again within a reasonable amount of time, the local set size may be larger (implementing LRU within each process), to take advantage of any temporal locality.

Thus, with p processes and an RU-set size of one, the minimum cache size is p buffers. This ensures that all processes may do I/O simultaneously without competition for buffers. A larger cache is needed when prefetching. However, we limit prefetching in two ways: first, a block that is prefetched into the cache cannot be removed from the cache until it is used (or recognized as a mistake); second, the number of these as-yet-unused prefetched blocks is restricted so that there are always some replaceable blocks in the cache. The former ensures that a prefetched block remains in the cache long enough to be used. The latter restriction requires that there are p replaceable blocks, again ensuring a buffer for every process (to be used for demand fetches). The limit on the number of as-yet-unused prefetched blocks in the cache is called the *prefetch limit*. The cache size is thus defined to be

$$p \times (\text{RU-set size}) + \text{prefetch limit}.$$

4.1.2 Prefetching Issues

Note that by reading blocks of data into fixed-size buffers in memory we are already implicitly prefetching adjacent bytes. If the application sequentially reads single bytes from a file, it is standard practice for the file system to read a block at a time and parcel out bytes to the application, paying the cost of the disk access only once. This dissertation considers prefetching on a higher level, reading blocks into memory before any byte of the block is needed by a process. To do this we must be able to predict future accesses to the file. Before we describe how this prediction is done, we must determine whether the file system has enough spare time to prefetch.

When to prefetch. There are two general situations when a process may have to wait before continuing. First, most multiprocessor algorithms involve some sort of synchronization between processes. An example is a barrier where each process waits until all processes have arrived. This time may be long. Second, a process may need to wait for a demand-fetched block (cache miss). In a uniprocessor, multiprogramming is often used to take advantage of these idle times; in many multiprocessors, however, the individual processors may not be multiprogrammed among user computations. Instead, while the process is idle, the file system manager may perform the necessary computation to prefetch blocks it expects the process to need in the future.

Another possibility is to issue prefetch requests while the process is running. With a multiprogrammed file system control, this requires interrupting the process and may perturb the execution sequence of the application process, a possible detriment to carefully tuned parallel programs.

At a synchronization point, it may be that the time spent making a prefetching decision and issuing a read request may delay this process from leaving the synchronization point, if all other processes arrive during this time (see Figure 4.2). This is called prefetch *overrun.* Similarly, if prefetching while a demand fetch is processed, it may not return immediately after the required block becomes available. If the prefetching decision was correct, however, the prefetched block will be requested by some process in the near future. Any amount of the access time of that block (both overhead and I/O) that may be finished before it is requested by any process is a possible reduction in the overall completion time of the program. Notice that it makes no difference which process uses the prefetched block, as long as the block is used by some process.

Simply reducing the time of individual read operations may not be sufficient to shorten the overall execution. If one process of a parallel computation gains from some optimization such

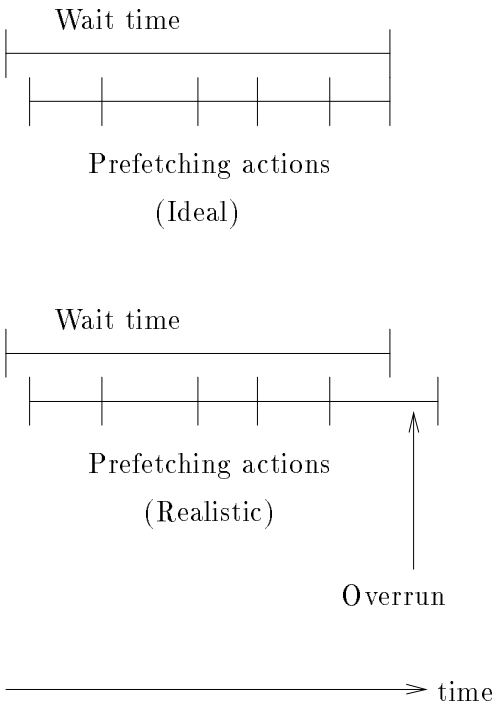
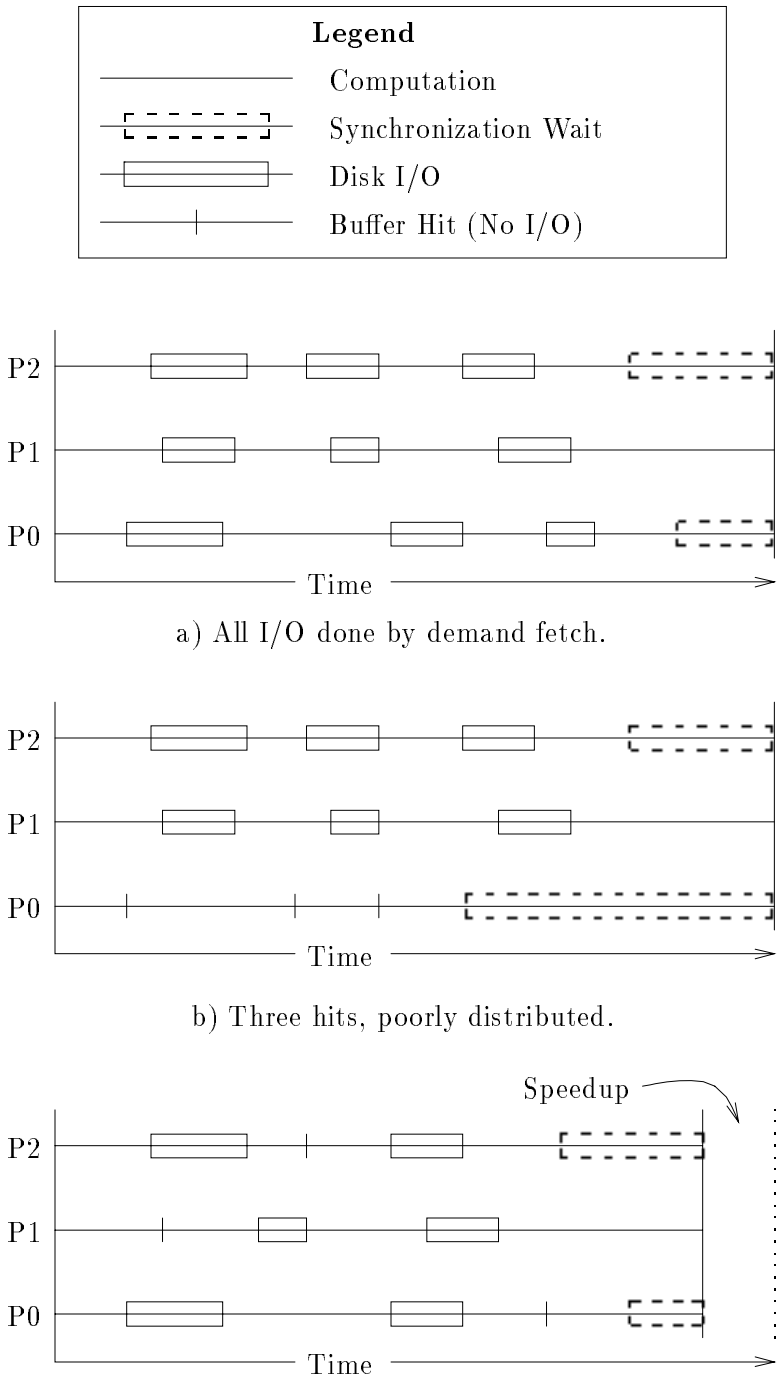


Figure 4.2: Prefetching actions may run past the end of the idle time available due to disk or synchronization waits.

as caching while another waits on disk I/O for every request, the overall benefit might be much smaller than if the advantages of caching could be spread over all the processes. Actual reductions in the overall completion time depend on our ability to reduce the disk access time so that some synchronization point (including the end of the program) occurs earlier than it would have without prefetching. For this to occur, the process that would have been last to arrive at the synchronization point must arrive earlier. This is demonstrated by the example in Figure 4.3.

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 cache 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 both the miss ratio and the average time to service a read request have 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. The interval between the two synchronization points becomes shorter as a result.



a) All I/O done by demand fetch.

b) Three hits, poorly distributed.

c) Three hits, better distributed.

Figure 4.3: Executions of a Parallel Computation

The file system must be careful when it chooses to do prefetching and how many requests it issues. It is possible, in fact, that prefetching may actually increase the execution time of a program. A processor that does prefetching may arrive at (or leave from) a synchronization point later than it would without doing the prefetch. In addition, the I/O for the prefetched block may conflict with I/O for data needed immediately, thus slowing down some process that is later waited for by all other processes. In either case the execution of the program as a whole may be slowed as well. Finally, in an imperfect prefetching strategy some blocks may be read in that are never used, occupying valuable cache space for a period of time (Smith terms this "memory pollution" [Smi78c]), and of course using valuable disk and channel time. We call these *prefetch mistakes.*

Prefetch Mistakes. Recall that the predictor module in the file system is responsible for tracking the access patterns and providing a prediction to the prefetch module when requested. Many predictors are likely to make *mistakes*, and predict the incorrect block reference sequence. If a mistaken prediction is supplied to the prefetcher, the block may be prefetched, wasting disk bandwidth and cache space. If the prediction is indeed incorrect, the block is never used. Our replacement policy removes blocks from buffers only after they are used by at least one process. Since a block may be prefetched into a buffer and never used, we supply a mechanism for predictors to explicitly free a buffer containing a mistakenly-prefetched block. They may recognize a mistake, for example, when the next block access occurs and proves the last prediction incorrect. A freed buffer may not be immediately available for re-use, since the disk read from the prefetch operation may not yet be complete. The buffer can not be used for another purpose until this I/O is complete, so it is put on the unready queue of the free list (page 24). Forcing blocks to remain in their buffer until all read I/O is complete is a conservative simulation choice. An alternative is to consider all I/O to be completely abortable, allowing the unfinished I/O to be ignored. Another is to consider I/O that is queued, but not yet started, to be abortable. Both of these alternatives require some optimistic assumptions about the way the I/O subsystem works. For generality, we use the conservative approach.

Prefetching can fail to get a free buffer when no free buffers are ready. Thus, it can fail after getting a block number from the predictor, without the block being fetched somehow. Each predictor accepts a block number back in order to undo the recommendation. This block number can then be predicted again.

4.1.3 Workload

Due to a lack of actual applications using parallel I/O, we use synthetic applications to drive our testbed. Each synthetic application is described by the combination of an access pattern and the values of various parameters. This allows us to generate a wide variety of conditions with well-known controls, and to repeat the same experiment with different prefetching policies and algorithms.

An application is represented as a partially ordered list of the records to be accessed, along with an amount of computation for each record. This list of records is directly related to the block access pattern, with each record reference expanded into the blocks it references. The list is either a set of local reference strings, for which the local order is exact but the global interleaving is determined at run time, or a single global access pattern that is followed in a "self-scheduled" manner by the processes. Due to natural, run-time fluctuations in the relative timing of parallel processes, the order of the references cannot be known in advance, but roughly follow the order given.

The Access Patterns

Read-only access patterns. We use access patterns describing eight basic application types, one for each of eight representative parallel file access patterns. Four of these are local patterns, three are global patterns, and one is random.

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

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

lrp Local Random Portions: this local pattern uses portions of irregular (random) length and spacing. Portions may overlap by coincidence.

seg Segmented: in this local pattern, the file is divided into a set of non-overlapping contiguous segments, one per process. Each process thus has one sequential portion. This is a special case of the non-overlapped local sequential pattern.

gw Global Whole file: this global pattern reads the entire file 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, but locally each processor only reads some small subset of the file with no discernible portions.

gfp Global Fixed-length 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.

rnd Random: this pattern accesses records at random.

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.

Write-only access patterns The following are the write-only access patterns that seem intuitively likely and that we use for our experiments:

lw1: A single process writes the entire file from start to finish.

seg: This pattern divides the file into segments, one per process, and each process writes its segment from start to finish.

gw: Like its read-only counterpart, this pattern writes records of the file in an arbitrary order, roughly sequentially from start to finish, with all processes cooperating to write the file. Some (external) synchronization method determines which processes write which records.

Except below in Section 4.2, we only use these access patterns in Chapter 9.

Synchronization Points

One of the opportunities for prefetching occurs at points in the program when the processors synchronize. A processor may take advantage of the time it spends waiting for the other processors by doing prefetching. We use four different types of synchronization points. The first type is no synchronization at all, and we call it *none*. There are two types of barrier synchronization: the processors synchronize after reading x blocks each (*each(x)*), or after reading x blocks total (*total(x)*). The final type is pairwise synchronization, in which each process must synchronize with each of two "neighboring" processors before continuing. We expect this style to be common in many scientific applications, when processors are each assigned a contiguous set of rows in a matrix. In this case, each processor synchronizes with its neighbor after reading x blocks, and we call it *neighbor(x)*.

4.1.4 Experimental Parameters

We fix most parameters for our initial tests in Chapters 5, 6, and 7. Chapter 8 explores parameter variations. These experiments are all for read-only patterns; Chapter 9 considers write-only patterns. The parameters described here are the base from which we make other variations.

All tests were run with 20 processes on 20 processors, each running the same application with the same set of parameters. Each experiment was repeated five times and the average of each measure was used for the data point. The patterns all contained exactly 4000 record accesses, where the record size was one block. 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.

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 except where noted. All other tests had no delay after each read, simulating an I/O-intensive process as an example of one extreme in the computation spectrum. Although this represents one extreme, we believe that it is common for many applications to cluster much of their I/O into small, I/O-intensive periods, such as at initialization.

The file was interleaved over 20 disks, at the granularity of a single block. The disk I/O performed in all of the tests was simulated using artificial delays to approximate disk access times. The delay, as with all measures of time in the testbed, was in real time. Each disk had 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 processes synchronized after reading 10 records on each processor (*each(10)*), after 200 total (*total(200)*, which is about 10 each), pairwise after 10 each (*neighbor(10)*), or not at all (*none*). The type of synchronization in any given test was the same across all processors. Prefetching was done both at synchronization points and while waiting for disk fetches (whether demanded by this process, demanded by another process, or prefetched by the file system).

The *RU*-set size of each processor was one block, totaling 20 blocks, implementing the "toss-immediately" version of our replacement algorithm. The prefetch limit was 60, making the total cache size 80 blocks. Because of the sequential nature of the access patterns, the additional 60 buffers were not generally useful without prefetching, but were included for fair comparison with the prefetching experiments. We chose this 60-block prefetch limit based on preliminary results that showed it to be a good compromise. We evaluate the actual effect of the prefetch limit (cache size) in Section 8.2.

4.1.5 Measures

The RAPID-Transit testbed records many statistics intended to measure and interpret the performance of prefetching. The primary metric for measuring the performance of an application is the overall completion time. This, and all time measures, is real wall-clock time, including *all* forms of overhead. We also record the average time to read a block, the average effective disk access time (the time from enqueuing a disk request to completion of the request), the total synchronization time, the cache hit ratio, prefetch overrun time (Figure 4.2), prefetch notification time, the number of mistakes, and many others.

A note on the data: In almost all cases, 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). Except for a few wildly variable data points (which are all noted), the cv was usually much less than 0.010, meaning that the standard deviation over five trials was less than 10% of the mean. In most places we just give the maximum cv for a given data set.

4.1.6 The Ideal Execution Time

To help in interpreting the results, we 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 T is

$$T = \max(\text{I/O time}, \text{comp time}).$$

With this workload, architectural model, and parameter values, the ideal I/O time is 6 seconds:[1]

$$\frac{4000\,\text{blocks}}{20\,\text{disks}} \times \frac{30\,\text{msec}}{\text{block}} \times \frac{1\,\text{second}}{1000\,\text{msec}} = 6\,\text{seconds}$$

Essentially this represents the minimum physical disk time that is necessary to complete the synthetic program. This assumes that the workload is evenly divided among the disks and that the disks are perfectly utilized. The ideal computation time is also 6 seconds, since there are 4000 records with an average computation time of 30 msec each, spread over 20 processors. Thus, the ideal total execution time, assuming I/O and computation perfectly overlap and there is no overhead, is also 6 seconds. Keep this ideal in mind while examining the data. No real execution of the program can be faster than the ideal execution time. In Chapter 8 we return to the ideal execution time when we vary the number of disks, number of processors, and disk access time.

4.2 The Benefits of Caching Alone

Before we examine the benefits of prefetching in detail, we first reinforce the importance of caching. Certainly, prefetching is only possible with some form of cache or buffer to hold the prefetched information. But caching has significant performance benefits of its own. A cache is useful as long as the access pattern has either temporal locality, in which data in the cache tend to be referenced again, or spatial locality, in which future data references are close to past data references (e.g., in the same block). As a demonstration, we ran all of our access patterns with and without caching.

[1] The ideal time for **lw** is shorter, only 0.3 seconds, since it only needs 200 disk reads.

The cache, when used, contained 80 one-block buffers. Other parameters were as follows: no computation, no synchronization, 20 disks, 20 processes, and 1 KByte blocks.

Table 4.1 shows the results of experiments on our full set of read-only access patterns. We also include a preview of the prefetching results. With one-block records (Table 4.1a), there was actually a slight performance degradation due to caching overhead. There was no improvement because most of these patterns did not rereference data in the cache. Some patterns (**lrp**, **grp** and **rnd**) made some rereferences, but so rarely that they were insignificant. The **lw** pattern had many (inter-process) rereferences, but execution time did not improve with caching because all processes read the same block simultaneously, and thus did not use the available I/O parallelism. The third column previews the prefetching results. Prefetching was able to use the sequential locality to improve the execution time for all access patterns except **rnd**. The dramatic improvement in **lw** was due to the use of all 20 disks in prefetching.

Table 4.1: Total execution time, in seconds, with and without caching, and with caching and prefetching. Read-only patterns. ($cv < 0.038$)

a) One-block records, Read-only patterns

Pattern	No caching	Caching	Prefetching
lfp	6.3	6.7	6.2
lrp	8.3	8.4	6.7
lw	6.9	7.1	0.7
seg	6.9	7.1	6.6
gfp	6.3	6.5	6.0
grp	6.4	6.7	6.3
gw	6.3	6.5	6.0
rnd	10.6	10.7	

b) Quarter-block records, Read-only patterns

Pattern	No caching	Caching	Prefetching
lfp	24.6	7.1	6.2
lrp	41.0	8.6	6.7
lw	36.5	7.3	1.2
seg	36.5	7.5	6.7
gfp	60.4	25.5	6.1
grp	60.4	25.4	6.8
gw	60.4	25.4	6.1
rnd	41.1	40.8	

The situation changed significantly when the record size was less than one block (Table 4.1b). Except in the **rnd** pattern, each block was referenced four times, once for each quarter-block record in the block. Without a cache, the block was read four times from the disk. The cache avoided this wasted disk bandwidth, but could not use the full disk parallelism in the global access patterns (four processes waited for each block to be read from the disk, and thus only one-fourth of all disks were in use at any time). With prefetching and caching, the disks were well utilized. Note that the benefits would be larger for smaller record sizes, and significant for all non-integral record sizes.

Table 4.2 shows the results of experiments on our write-only access patterns. Here we compared a simple write-back caching policy (see Chapter 9) with not caching. Caching was faster in **gw**, since the delayed write allowed some overlap between overhead and I/O. The **lw1** pattern was most improved because, with caching, this one-processor pattern was able to use more than one disk. Experiments with quarter-block records, shown in Table 4.2b, demonstrate the real power of caching: without a cache, all writes to a disk block after the first write had to read the block from the disk, update the block, and write the block back to disk. With n records per block, a cache reduces the $2n-1$ disk accesses per block to one per block. The cache also allows concurrent access to blocks; without a cache, all access was serialized. This is evident by comparing the non-cached results for **gw** (where four processes used each block) with those for **seg** (where only one process used each block).

Table 4.2: Total execution time, in seconds, with and without caching. Write-only patterns. ($cv < 0.015$)

a) One-block records, Write-only patterns

Pattern	No caching	Write-back caching
lw1	127.3	16.4
seg	6.9	7.2
gw	6.3	6.1

b) Quarter-block records, Write-only patterns

Pattern	No caching	Write-back caching
lw1	853.1	55.7
seg	63.3	7.7
gw	103.0	8.7

Chapter 5

The Potential of Prefetching

Our strategy is to first determine the *potential* for prefetching to improve file system performance on our read-only patterns. To do this, we make the unrealistic assumption that the entire access pattern is known in advance. This is the basis for the experiments in this chapter. In the next two chapters, we define and evaluate heuristic prefetching methods that make real-time predictions based on the accesses as they occur.

Here we use the EXACT predictor, which is provided with the entire access pattern in advance. This makes it a perfect predictor, since it makes no mistakes and requires little overhead, but not realistic, since a real predictor does not know the entire access pattern in advance. Nonetheless, 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 is reasonable to expect prefetching success within but not between sequential portions. Thus, EXACT 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 chooses not to use information it has in its supplied reference string that would be impossible to predict). In addition, no prefetching is possible in the **rnd** pattern, so we do not consider it further in this chapter.

We measure the potential for prefetching in terms of its ability to improve some performance measure over the value obtained without prefetching. The combination of several factors contributes to the general success of prefetching as determined by one of the following measures:

- reduced average block read time
- increased cache hit ratio
- reduced overall execution time

Although the final goal may be the ultimate measure of prefetching success, the others are important to consider. Change in the overall execution time may depend more strongly on the characteristics of our workload than the other two measures may, whereas the first two goals may be more significant in other workloads. In addition, 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 caching techniques. As we noted on page 26, however, an improvement in the block read time or hit ratio does not necessarily translate directly into an improvement in the overall execution time. We examine this relationship further in Section 5.2.8.

Our test set for each access pattern used all four synchronization styles, both without computation and with 30 msec of computation. Each of these eight combinations represents one test case. Each test case was run for seven patterns (all except **rnd**), producing 56 combinations. Each combination was run with and without prefetching to produce one data point.

A note on the graphs: In many of the graphs that follow, the distribution of a set of values is shown as a *cumulative distribution function* (CDF), with the fraction of the points having less than or equal to a particular value plotted against that value. The actual data values are plotted and connected by lines. This allows easy location of the median (at 0.50), upper (0.75) and lower (0.25) quartiles, and a general view of the distribution. In other graphs, pairs of values (e.g., one with prefetching and one with no prefetching) are compared in a *scatter plot*. Other graphs plot the *percent reduction due to prefetching* of some measure, which is simply the change caused by prefetching as a percentage of the value measured without prefetching.

5.1 Prefetching Support for One Processor

Most existing programs, both for uniprocessors and multiprocessors, do not make use of any form of parallel I/O. The usual paradigm for parallel programs that need to read or write files is to have one controlling process open and read the input, then use parallelism for processing, and then use a single process to write the output file. In our execution model, there are several idle processes when this single process is reading or writing. A single reader process may speed up using parallel disks, but may speed up more if the otherwise idle processors are used to prefetch for the reader process. Additional speedup is possible if all processes are used to read the file in parallel, as in the experiments described in the next section.

The following experiment demonstrates the potential for prefetching to improve I/O performance in parallel computations. The experiment involved a 20-process computation on 20 processors with 20 disks. The file was 4000 blocks long and block-interleaved over all disks. First, a single process was used to read the file. Then one process was used to read, and all 20 processes (both the reader and the 19 otherwise idle processes) did prefetching. Finally, all 20 processes read the file as in the **gw** pattern. The results are shown in Table 5.1. The speedup values are relative to the first case.

Table 5.1: Speedup attained by prefetching and reader parallelism.

Experiment	Execution time (sec)	Speedup
One reader, no prefetch	126.0	1.0
One reader, 20 prefetch	8.7	14.5
20 readers, no prefetch	6.4	19.7
20 readers, 20 prefetch	6.0	21.0

Execution was greatly speeded by the use of the other 19 processes, for reading, prefetching, or both. The first case simply did not use the available disk bandwidth, reading from one disk at a time. Since this example involved an I/O-bound computation, its execution was speeded significantly when the full parallel bandwidth was used. A perfect speedup of 20 is possible by parallelizing the I/O requests over 20 disks and the overhead over 20 processors. The third case closely approached this perfect speedup of 20. In the last case, the additional speedup came from overlapping the I/O and the read operations through prefetching. In this case, the execution time matched the ideal 6 seconds. There are two lessons from this experiment:

- Parallel disk I/O, if used, can greatly improve disk bandwidth.

- Prefetching alone (as in the second case above) can be provided transparently by the file system to attain significant speedup (14.5) on an otherwise sequential part of the computation.

5.2 Prefetching in Multi-process Patterns

In this section, we experiment with prefetching on a wide variety of workloads (see Section 4.1.4), intended to explore a broad range of possibilities. The results are presented in graphs with each workload represented by a single data point. Thus the graphs represent our particular workload mix. Remember that the performance of a real system depends on its particular workload mix.

5.2.1 Average Block Read Time and Hit 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 is an attempt 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 5.1, the average block read time was significantly reduced through the efforts of prefetching in all cases we studied. In this figure, the read time with 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 was reduced. The improvement in the average read time exceeded 50% for 54% of the experiments, had a median of 56%, and reached as high as 93%.

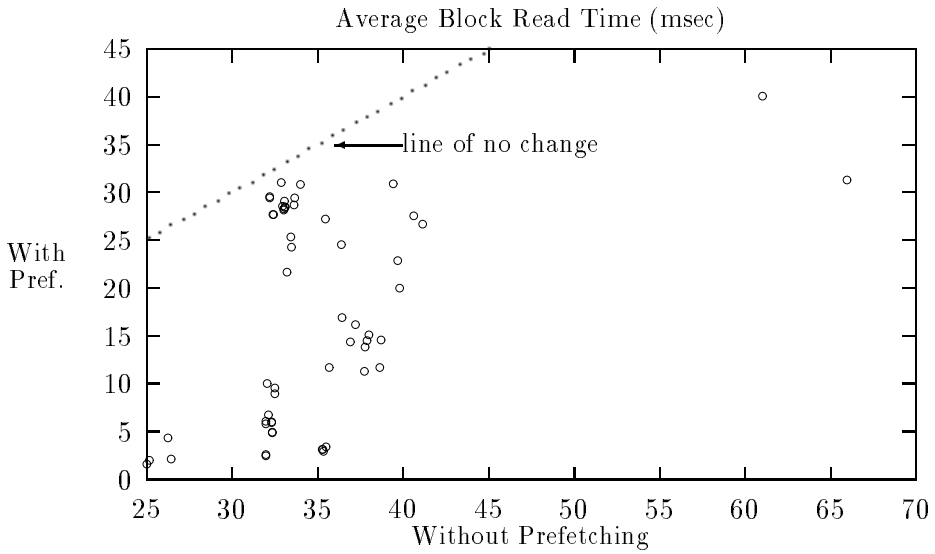


Figure 5.1: Prefetching consistently lowered the read time.

A possible reason for the improvement in the block read times was the high cache hit ratio, shown in Figure 5.2. The hit ratio for all prefetching experiments was over 0.72, with more than a third of them over 0.98. Due to the sequential nature of the accesses, most of the corresponding

experiments without prefetching had a hit ratio of 0.00 (those that did not had some measure of inter-process locality, as in the **lw** pattern).

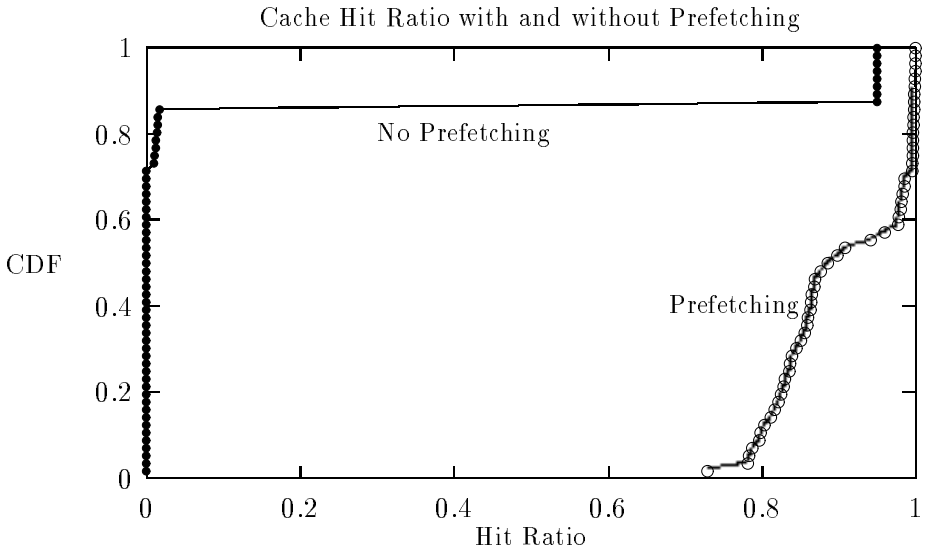


Figure 5.2: The cache hit ratio with prefetching was consistently over 0.72, with more than a third of the points over 0.98.

A strong improvement in the hit ratio was not enough to lower the average read time, however, since even a block that was found to be in the cache (perhaps due to a prefetch action or the demand request of another process) may still have had a large proportion of its I/O time outstanding. The process had to wait for the I/O to complete, so we call this the *hit-wait* time. The hits with a non-zero hit-wait time were *unready* cache hits (ready cache hits had a zero hit-wait time) and often represented a significant portion of all cache hits. In Figure 5.3 the hit ratio (as in Figure 5.2) is compared with the ratio of unready cache hits. These unready cache hits may be construed as misses and so we examine the hit ratio that includes both types of misses in Figure 5.4. Prefetching still has a clear advantage over not prefetching.

Although the unready cache hits may be construed as misses, they did not have the same impact as true misses (see Figure 5.5). In general (79% of all cases), the average hit-wait time was less than the disk access time of 30 msec. Of course, the effect of the hit-wait time on the average block read time also depended on the hit ratio and the unready hit ratio. A small hit ratio would virtually eliminate the effect of hit-wait time, whereas a high hit ratio could be accompanied by either a large or small average hit-wait time depending primarily on characteristics of the benchmark program (e.g., access pattern and computational intensity). Our results show that the hit-wait time was usually smaller than the time required for a miss but was large enough (when combined with a high number of unready hits) to be a contributing factor in the time required to read a block. While there seems to be a fuzzy relationship between the average block read time and the hit-wait time, no obvious relationship has been found between the read time and the hit ratio measure over the full range of experiments.

One factor contributing to the average block read time (when prefetching) was the cost of the prefetching overhead. Prefetching incurs overhead both directly, due to overruns (page 25), and

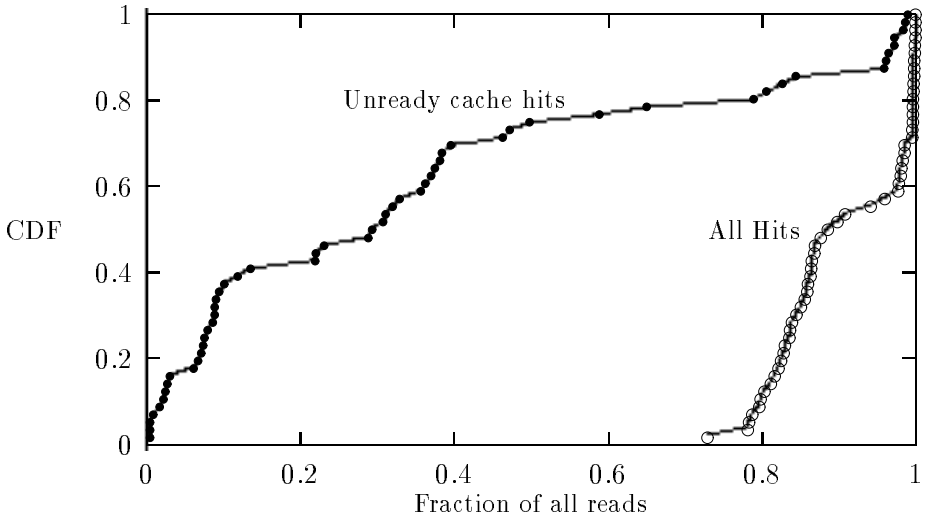


Figure 5.3: The unready cache hits represented a significant fraction of all reads.

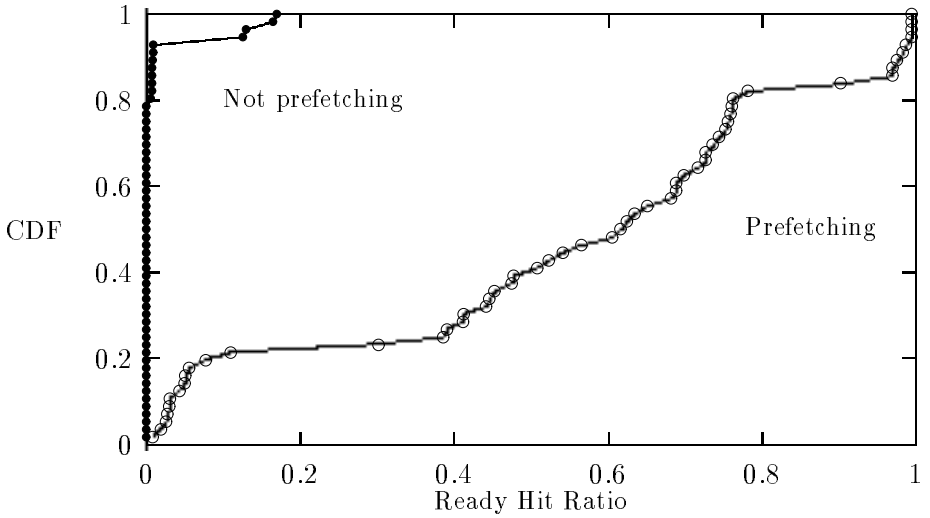


Figure 5.4: If the unready cache hits are considered misses, the hit ratio was lower, though still much improved with prefetching.

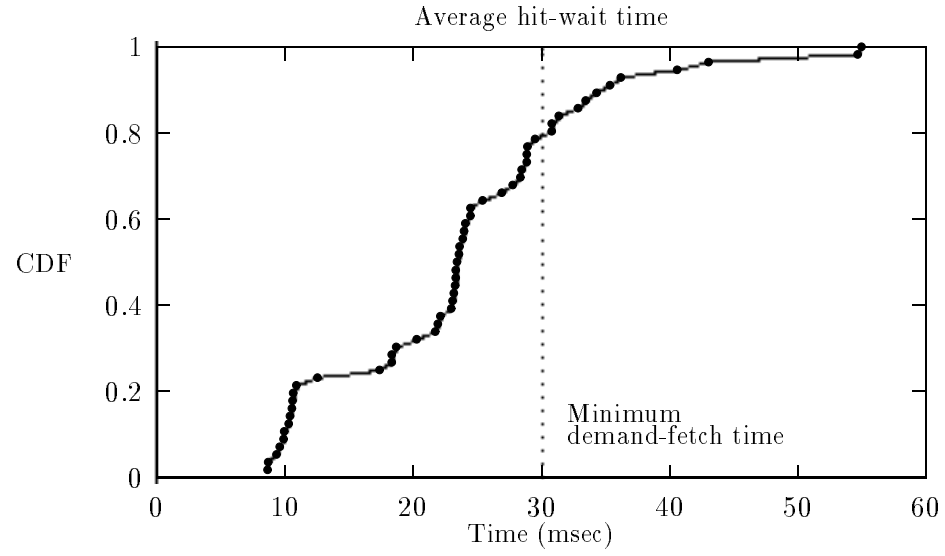


Figure 5.5: The average time spent waiting for unfinished disk I/O can be large. Nonetheless, the wait for a cache hit was usually less than the minimum disk response time of 30 msec.

indirectly, due to increased contention for the disks and internal data structures. Long prefetching times contribute to long overrun times and, of course, reduce the number of prefetches that are initiated during a given period of idle time. A prefetching overrun causes a direct slowdown to one process, as it delays the process from continuing its computation when it is otherwise ready. Clearly, for prefetching to be a successful technique, these costs must be kept to a minimum. The prefetching time in our experiments was usually low, keeping the average overrun time near 1–2 msec in most cases (although certainly some overruns were much longer than the average).

Contention for the disks was 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 was sometimes larger than the physical disk access time when contention for the disks forced disk requests to be queued for service. Since the physical disk access time was fixed at 30 msec for our experiments, this was really a measure of the disk queuing delay. The disk response time slowed as contention for the disks increased. Prefetching increased the contention for the disks as it filled the queues with read requests, shown in Figure 5.6.

Note that the disks serviced no more requests under prefetching than they did without prefetching, since the EXACT predictor fetched no unnecessary blocks. The extra disk load arose from the same number of requests issued in a smaller amount of time. This is clear when the total execution time was reduced. Even when the total time was not reduced, the disk reads tended to be issued non-uniformly in time, creating periods of high disk contention. In general, an experiment that had high disk utilization even without prefetching experienced sharp increases in the disk response time as prefetching filled the disk queues. This effect can be seen for several points in Figure 5.6. Because of prefetching, however, the longer disk operations were overlapped with other I/O, with computation, and with file system overhead, resulting in faster average block read times.

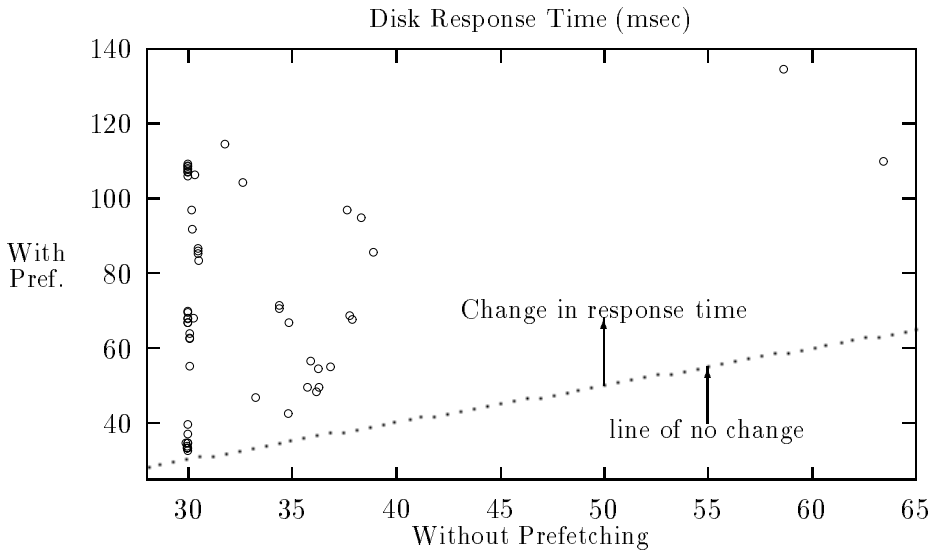


Figure 5.6: Increased contention for the disks lengthened the disk response time.

5.2.2 Effect on the Total Execution Time

Our primary measure of prefetching effectiveness is the total execution time of the program. We have found that prefetching reduced the total execution time, often significantly, for most of the cases we studied (see Figure 5.7). The biggest improvements, up to 90%, were in the **lw** pattern where all 20 processes could benefit from each prefetched block.

Occasionally, prefetching increased the execution time. Two of the **lfp** pattern experiments slowed down as much as 10%, despite solid improvements in the hit ratio and the average block read time. This was due to an uneven distribution of the benefits of prefetching, as outlined in Figure 4.3b. In local patterns, including **lfp**, the processes prefetched only for themselves. Thus, any prefetching completed by a process benefited only that process, in the form of hits and shorter read times. It appears that, due to subtle timing issues, some processes grab several buffers and prefetch for themselves, leaving few buffers for other processes. Their time was improved, but they had to wait at the next synchronization point for the less fortunate processes. Those other processes, in their own attempts to do prefetching (often unsuccessful due to the lack of free buffers), wasted some time in prefetching overruns, *lengthening* the interval and hence the total execution time. This issue, and a solution, is discussed further in Section 5.2.8. Another solution was an increased cache size (Section 8.2), where **lfp** had a 10% speedup due to prefetching.

The **lrp** pattern, which had reasonable improvements in the total execution time, did not exhibit this effect as strongly. Recall that with random portions we restricted the prefetching of any one process to its current portion, which was usually short. Thus, it was difficult for one process to use many prefetch buffers.

Synchronization delays, the time between arrival of a process at a synchronization point and the moment all processes achieve synchrony, were affected in other patterns as well. This was another factor that affects the total execution time. The synchronization delays often increased as some of the savings on I/O operations were converted into longer synchronization waits. Prefetching increased average synchronization time in about half of our test cases. In half of those cases the

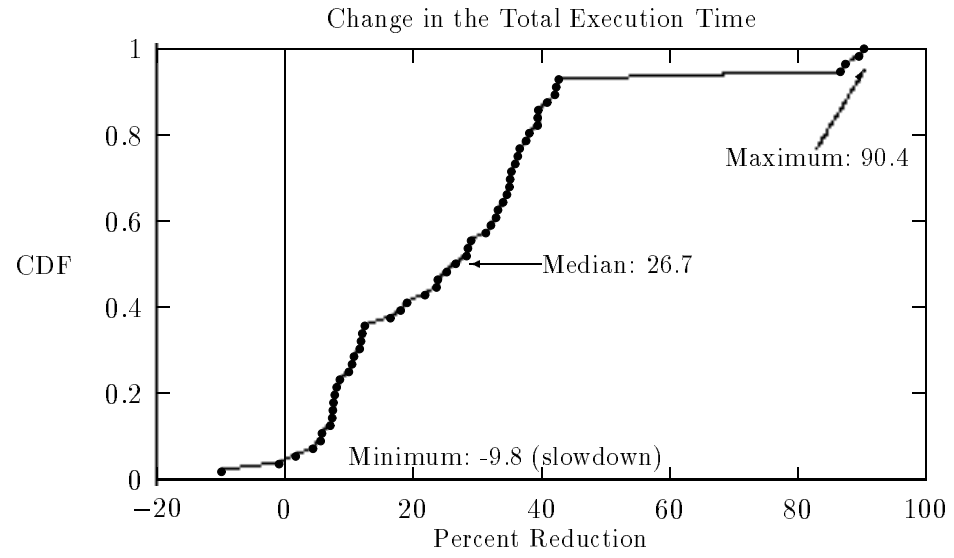


Figure 5.7: Prefetching improved the total execution time in most cases.

increase was less than 100%, but in some cases the synchronization delay increased by almost 4000%.

Without some way to distribute the primary benefit of prefetching (a lower block read time) among the processes, any reduction in the *average* block read time did not necessarily translate into a reduction in the total execution time. Figure 5.8 plots the reductions measured in our experiments, demonstrating at best only a fuzzy relationship. Figure 5.9 plots the reduction in total execution time against hit ratio.

For these experiments, neither the hit ratio nor the average block read time were strong predictors of overall success. Nonetheless, some significant improvements in both measures (read time and hit ratio) were obtained with prefetching.

5.2.3 The Balance between Computation and I/O

The preceding results have looked at the data for all data points, from all of the parameter combinations we used in our experiments. Many of these runs simply read one block after another with no time spent processing each block and represent one endpoint of the workload spectrum. When the processors devote all of their time to I/O, there was an increased likelihood that they contended for access to the disks and internal data structures. To simulate programs with some computation, we associated computation time with each block fetched in many of the runs, as described in Section 4.1.4. These runs are included in the preceding figures, but it is valuable to examine the effect of this variable separately.

We chose one access pattern and one synchronization style and varied the average computation time per block over a wide range of values. The experiments in this section all use the **gw** pattern and *each(10)* synchronization. The computation time is exponentially distributed about a given mean. The idea was to study the effects of prefetching on various measures as the program changed from I/O-bound to compute-bound.

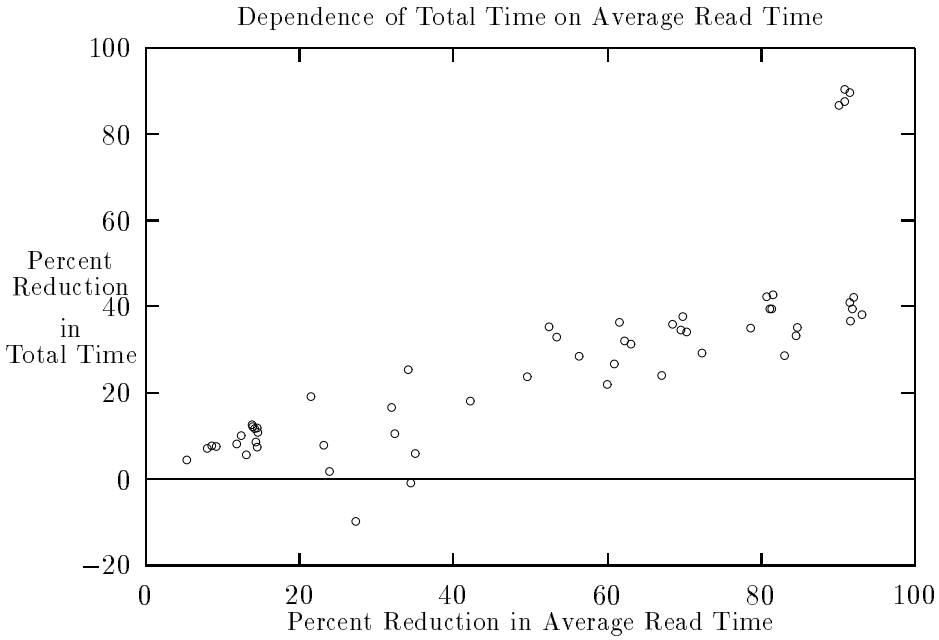


Figure 5.8: Reducing the average block read time does not necessarily imply a reduction in the total execution time.

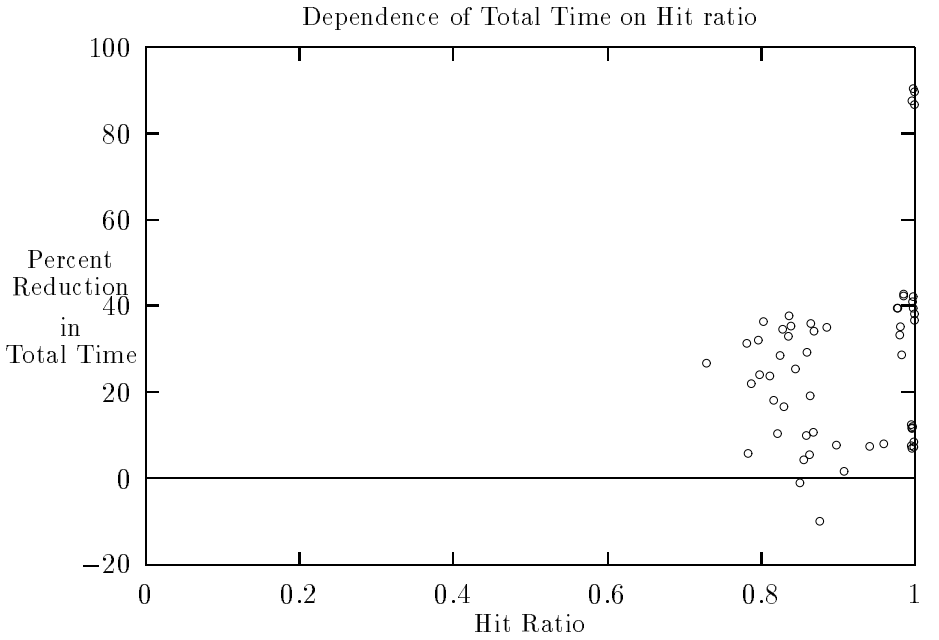


Figure 5.9: Increasing the hit rate does not necessarily imply a reduction in the total execution time.

The results obtained were as might be expected. As the character of the program switched from I/O-bound to compute-bound, prefetching benefited from overlapping the I/O and the computation. Indeed, Figure 5.10 shows that the total execution time improved more when the program spent some time in computation, but this tailed off as the bulk of the program's time was spent in computation (which did not improve with prefetching) and the effect of the I/O time improvement became less significant. The improvement was due to an increasingly large *reduction* in the average block read time, which dropped to 9% of its value without prefetching.

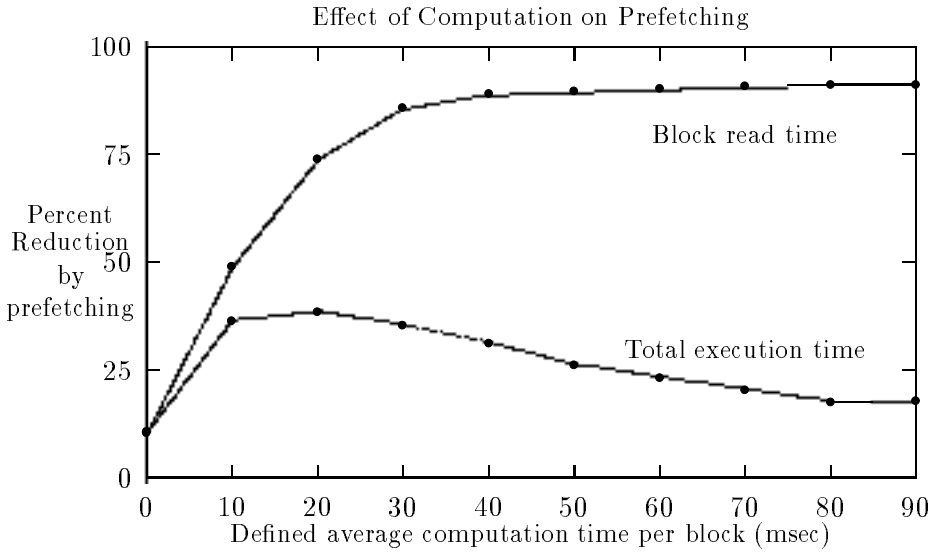


Figure 5.10: Larger improvements in the block-read time and some improvements in the total execution time are obtained when the processes are less I/O-bound.

The primary reason for the reduced average block read time was the overlap between computation and I/O. When prefetching, most or all of the I/O delay was finished during periods of computation, leaving shorter hit-wait times. Another reason the average block read time improved was the reduced response time of the disk. When the processes were I/O-bound, there was a great deal of contention for the disks and internal data structures. This contention decreased steadily as more time was spent processing each block, although the disk response time when prefetching was still higher than the response time when not prefetching.

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

5.2.4 Attempts to Improve Prefetching

Given what we learned about prefetching and the effect of various measures on the average read time, we explored a change to the prefetch strategy that could improve performance. Although a

high cache hit ratio was important to lower the block read time, we found that a low hit-wait time was also an important component. It seems that the blocks that were prefetched were needed soon after the prefetch had been initiated. A possible improvement to the strategy is to avoid prefetching blocks that would be used soon, and instead to prefetch well ahead of the current activity in the file. In other words, to have the prefetch activity "lead" the demand-fetch activity by some distance. We tried this strategy with a varying amount of minimum prefetch lead for several patterns. For space reasons we do not include any plots of the data.

As hoped, the hit-wait time was reduced considerably by increasing the minimum lead, except for the **lw** pattern, whose hit-wait time decreased only slightly. In sharp contrast, however, the cache hit ratio dropped significantly. Unfortunately, the significant decrease in the hit ratio diminished the effect of the improvements in the hit-wait time on the the average block read time, and the average block read time increased for most patterns (it improved slightly for **gfp**).

It was therefore no surprise that most patterns slowed down overall. The total time for **lfp** was slightly improved, and then only for large values of the minimum prefetch lead. It is unlikely, however, that prefetching leads of more than 10 or 20 would be workable in an automated prefetching system, due to the inaccuracy of many prefetching decisions. In any case, the result was that no satisfying improvements were obtained by using a minimum prefetch lead.

5.2.5 The Importance of Synchronization Points

A portion of our experiments used no synchronization points at all, synchronizing only at the end of the program. These experiments give us an indication of the importance of synchronization points and of the prefetching that occurs during synchronization points. Interestingly, not synchronizing at all produced the largest read times for many patterns, though the improvements in both block read time and total execution time were similar to other synchronization styles. One reason for the higher read times was the disk contention: synchronization was often a respite for the file system, as processes stopped reading blocks. Synchronization points also provided a chance for prefetched I/O initiated before or during the synchronization point to overlap with the idle time of the synchronization period, and to be completed (or more nearly completed) before the block was requested by some process after synchronization. Without synchronization points, processes spent more time waiting for I/O.

5.2.6 Differences Between the Patterns

Most of the preceding discussions make no distinction between the data points based on the access pattern. In fact, the access pattern often accounts for many interesting differences between experiments. For example, the **lw** (local whole-file) pattern gained the most benefit from prefetching. This pattern is represented in most of the preceding graphs as the most-improved points, with improvements in the total execution time of about 30–90% whereas the other patterns had improvements of 0–44% with some slowing down by up to 10%. The pattern that slowed down was **lfp**, as described in Section 5.2.2. This slowdown is discussed further in Section 5.2.8.

Even without prefetching, **lw** had a hit ratio of 0.95 since all 20 processors read only 200 blocks of the file (chosen so the total number of accesses was 4000, for consistency with the other patterns). Therefore, as long as the processes maintained inter-process locality, there were 19 cache hits for each block fetched. With prefetching, the number of misses was often reduced to a single block: the first one. This meant a hit ratio of 0.99975, or nearly 1. Because 20 processes benefited from each prefetched block, the benefits of prefetching were enormous compared to the other patterns, which had little (if any) inter-process locality. It was this aspect of **lw** that made it unique among

the other patterns and that makes its data stand out.

The parameters were chosen to make all of the experiments similar, so that the actual total execution time could be compared. The **lw** pattern had a much lower total execution time than all of the others, due to the reduced number of disk fetches. Most of the others were similar to each other. Otherwise, *total(200)* synchronization rather than *each(10)* added a measure of load-balancing that generally reduced the execution time. Furthermore, not synchronizing at all (*none*) usually gave the lowest execution time of all.

5.2.7 The High Cost of Prefetching Overhead

A prefetch action was sometimes complex, requiring several milliseconds to complete. To prefetch a block, a number of parameters and indicators were checked, such as the number of buffers filled with unused prefetched blocks. Then a block number to prefetch was obtained from the predictor (which in EXACT meant checking the provided reference string), possibly contending with other processors for access to the predictor's data structures. Following this, a buffer was found for the prefetched block, and any block resident in the buffer was removed (although no disk operation was necessary since these patterns are read-only). Then the I/O request was placed on the correct disk queue, and the status of the prefetched block updated.

Much of the activity involved in prefetching required several accesses to data structures in shared memory, which (on the Butterfly) were significantly slower than local memory accesses, particularly with memory contention. Programs that spent more time processing and less time doing I/O had lower prefetch times due entirely to reduced memory contention. We expect that although extremely compute-intensive programs might have smaller hit-wait times, leaving less time to prefetch, they would be able to prefetch more blocks due to a substantially reduced prefetch action time.

In our experiments we found the prefetching overhead to be high at first. The most significant indication of this was extremely high (60 msec) prefetch times and overruns. It was necessary to optimize the paths through the I/O subsystem, both for prefetch actions and demand fetches. 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. Expensive operating system calls to map buffers in and out of the file system's virtual memory space were avoided, keeping buffers mapped in as much as possible. Statistics were gathered in data structures kept completely in local memory. We used atomic memory operations (such as fetch-and-add) in the place of critical sections maintained by locks. The locks that were used on the data structures were held for short periods only and each lock was designed to affect only a small piece of the data structure. In short, our experience with the implementation of a simulated system implies that any practical implementation of prefetching must pay a great deal of attention to optimization.

5.2.8 Balancing the Benefits of Prefetching

In Section 5.2.2 we note an unexpected slowdown in the **lfp** pattern caused by prefetching. In local patterns, each process chose blocks for prefetching exclusively from its own list, ignoring the access patterns of other processes. All processes shared the same set of prefetching buffers. If one process moved more quickly than the others, it could fill many buffers with prefetch requests from its own reference list. This restricted the ability of other processes to prefetch for themselves, since the number of buffers was limited. Thus the more aggressive process benefited more from prefetching than the others, and finished its work more quickly. The other processes did not improve as much,

and the quick process waited for them at the next synchronization. Thus, there were improvements in the block read time, but they were lost to synchronization delays. The *slowdown* arose from the overhead expended by the prefetch manager attempting (and failing) to prefetch for the slower processes. We call this the *greedy-process problem.*

The ultimate goal of any solution to the greedy-process problem is to more effectively balance the benefits of prefetching in order to lower the overall execution time of the application. We examine two solutions to the problem here. Other mechanisms that solved the problem include a restriction on prefetching (Section 6.6) or a larger cache size (Section 8.2).

Private Prefetch Limits. A direct solution to the greedy-process problem is to limit in some way the amount of prefetching done by any one process. This may be done by placing a *per-process*, rather than global, limit on the the number of buffers used for prefetched blocks. In our experiments, this meant limiting each of the 20 processes to three unused prefetched blocks, rather than limiting the group of 20 processes to a total of 60 unused prefetched blocks. The number of buffers allocated for prefetching was the same in each case, but in this solution no process could use buffers intended for another process. Inter-process cache hits (due to opportune portion overlaps in the **lrp** access pattern) were still permitted. This solution is called Private Prefetch Limits (PPL).

Prefetching For Others. In our prefetching model, the blocks prefetched by the file system prefetch manager during the idle time of any given process were chosen from the reference list of that process. In the greedy-process problem, more blocks were prefetched for an ambitious (greedy) process than for the others. The Prefetching For Others solution, called PFO, forced every prefetch manager to choose prefetch candidates from *all* reference lists. This solution used a self-scheduled selection of prefetching candidates from the various lists by all prefetch managers.

Table 5.2: Percent improvement of prefetching over not prefetching, for different prefetch techniques.

Pattern	Normal	PFO	PPL
	-0.8	10.9	10.9
	-9.8	12.5	12.6
	1.8	12.4	11.0
lfp	8.1	9.7	9.7
	26.7	46.1	33.8
	24.0	40.6	29.9
	34.6	44.9	32.1
	37.6	48.4	34.0
	18.1	21.0	-2.1
	5.9	10.0	-2.4
	16.6	16.9	0.7
lrp	19.1	19.3	-10.9
	31.3	36.1	9.2
	21.9	23.1	6.1
	32.1	35.0	17.3
	36.4	38.0	10.5

We compare our prefetching techniques in Table 5.2. The measure used for comparison is the percent improvement due to prefetching, as compared to not prefetching. This result is given for each of the three prefetch techniques: our normal prefetch algorithm, prefetching for others, and private prefetch limits. A negative improvement represents a slowdown caused by prefetching. There are eight variations for each pattern, representing different combinations of the synchronization style and computation parameters. These results show that PFO was always faster than (or as fast as) the normal prefetching method. The PPL technique was not as successful, especially on the **lrp** pattern. It solved the greedy-process problem as well as PFO did, but was over-restrictive in cases where the greedy-process problem was not an issue.

The primary reason for the improvements in the **lfp** pattern was an improvement in the load balance at each synchronization point (data not shown). Thus, these techniques had the desired effect, to balance the benefits of prefetching. These techniques, however, are not general-purpose, since they work well only on patterns that have inherently balanced amounts of computation and of I/O. More sophisticated techniques might be necessary if a load imbalance already exists within the application. On the other hand, increasing the cache size (and thus reducing the competition for buffers) is a simple solution that was also effective (Section 8.2).

5.2.9 Summary of Multi-process prefetching

The experiments in this chapter are intended to measure the effectiveness of our prefetching and caching techniques in a file system incorporating parallel I/O, given perfect pattern prediction. We found that prefetching did indeed help to significantly reduce the average block read time and to increase the cache hit ratio, generally contributing to a decrease in the overall execution time of the parallel program. The cache hit ratio was an optimistic measure while the read time took into account prefetching overhead and the hit-wait time as well as the cache hit ratio. Some important contributors to the read time were the hit-wait time and the disk response time. The average block read time and the hit ratio were only part of the story. The total execution time was the only good indicator of overall performance.

Although these techniques were intended to reduce the I/O time seen by a parallel program, they were most effective to a program that did some computation as well as I/O, since much of the I/O time could be overlapped with the computation time, further reducing the I/O time actually seen by the program. The speedup to be gained, however, was largest for programs that were roughly balanced between computation and I/O, just overlapping their I/O with computation. Indeed, parallel programs have opportunities for multiplying this overlap factor when more than one process requires a particular disk block at the same time: without prefetching, all processes must wait for the I/O to complete; with prefetching, the I/O may be overlapped with the computation of all processes.

The **lw** pattern benefited the most from prefetching. There are two reasons, both due to **lw**'s overlapped access pattern. First, any prefetched block benefited all processes. Second, without prefetching **lw** used only one disk at a time (every process waited for one block, then went on to the next). Prefetching was able to use all of the disks and thus run much faster.

There was no reason to avoid prefetching blocks needed too soon in the future, since our experiments suggest that any advance work done reading a block was helpful in reducing the effective access time for that block.

It may seem that a process that does little I/O would leave little time for prefetching to take place, suggesting that the application should be interrupted for the purpose of prefetching. It appears, however, that idle times are sufficient for prefetching. Indeed, interrupting the application may disturb carefully tuned sections of parallel algorithms. Therefore we advocate prefetching only during application idle periods.

Chapter 6

Automatic Prediction in Local Patterns

In the previous chapter we established the potential for prefetching to improve the performance of file I/O. The results depend on the EXACT predictor, which had full knowledge of the access pattern in advance. In any practical system an off-line predictor like EXACT is not possible, so on-line predictors are needed. In this chapter we examine several on-line predictors for local access patterns (global patterns are treated in Chapter 7), to determine whether on-line predictors can reach the full potential identified in the previous chapter. This chapter compares several predictors on a fixed set of architectural parameters; Chapter 8 examines the effects of these parameters on one local predictor.

6.1 Introduction

A predictor recommends blocks to prefetch by predicting the near future of the access pattern. *On-line predictors* are designed to form their prediction at any point from only the access pattern seen up to that point. As file read requests occur, the predictor revises its prediction for the next block (or blocks) to be accessed. These predictions are used for prefetching when time is available. The RAPID-Transit testbed supports several predictors in addition to the original EXACT algorithm. The experiments in this chapter determine what predictor to use for each type of local access pattern, and whether there is any candidate for a single general-purpose local predictor. First we describe the idea behind each predictor. Then we outline a set of experiments for evaluating the predictors' performance, and present and discuss the results.

6.2 The Predictors

Four basic and three hybrid predictors are defined here. These are all on-line predictors, making predictions in real time based on the data at hand. Remember that predictors do not actually prefetch anything; they are used by the prefetch module to make predictions, which are then used for prefetching. There are two baseline predictors: the original "off-line" EXACT algorithm, and the degenerate case of no prefetching, which we call the NONE predictor. Each predictor is suited to a different type of access pattern, although some may be more flexible than others. They also vary in implementation complexity.

Depending on the mix of patterns in a given workload and the importance of optimizing for particular patterns, one may be able to choose a specific predictor for that workload. Each of the

first six predictors is designed for a specific kind of pattern. However, one that is designed for the **lw** pattern is definitely not the right choice for the **rnd** pattern. By mixing the characteristics of several basic predictors, we derive three hybrid predictors that may more easily accommodate multiple patterns.

6.2.1 OBL — One-Block Look-ahead

This algorithm always predicts block $i + 1$ after block i is referenced. This is the only block it recommends for prefetching. It has no prediction at the start.

Note that our implementation of OBL is not the same as most implementations of OBL found in the literature. Usually, OBL implies prefetching the next block after *every* read. RAPID-Transit prefetches only during process idle times, so blocks may not be prefetched after every read.

6.2.2 IBL — Infinite-Block Look-ahead

This algorithm is like OBL, in that it predicts block $i+1$ after block i is referenced. It also predicts that $i+2, i+3, \ldots$ will follow, and recommends that they all be prefetched in that order. Whether they are actually prefetched, of course, depends on the currently available resources. IBL is a logical extension of OBL, and is designed for the **lw** and **seg** patterns.

6.2.3 PORT — Portion Recognition

This algorithm attempts to recognize sequential portions. Essentially, PORT tries to handle the **lfp** access-pattern family. It expects a regular portion length and regular portion skip (the distance from the end of one portion to the beginning of the next). 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. This number of blocks is the *prefetch distance*, because it represents a distance ahead in the access pattern. With a short prefetch distance, PORT may predict a few blocks ahead in the pattern, but reach past hundreds of blocks in the file by jumping over portion skips. The prefetch distance depends on the following parameters:

MinLen is the number of blocks requested in a row necessary to call the observed sequence a portion. We use MinLen=1.

LenRep is the number of consecutive, identical portion lengths needed to consider them regular. We use LenRep=2.

SkipRep is the number of consecutive, identical portion skips needed to consider them regular. We use SkipRep=2.

MaxDist is the upper limit on the prefetch distance. We use MaxDist=5.

PORT has no prediction at the start, and predicts nothing until it sees a few (MinLen) blocks in a row. At this point it predicts a few blocks in advance, depending on a distance function (below). If it sees a new portion begin, it records the old portion length and the skip. If, after a few (LenRep) portions, the length remains constant, it limits its predictions to portions of that length. If the portion length is regular, and if the skip also remains constant (for SkipRep skips), it predicts right over the skip into the next portion (possibly over multiple skips, if MaxDist is greater than the portion length), up to a distance of MaxDist.

The prefetch distance for PORT, when the portion length is irregular, is a function of the current portion length c:

$$\text{distance}(c) = \begin{cases} 0 & \text{if } c < \text{MinLen} \\ c - \text{MinLen} + 1 & \text{if MinLen} \le c < \text{MinLen+MaxDist} \\ \text{MaxDist} & \text{if } c \ge \text{MaxDist+MinLen} \end{cases}$$

The MaxDist cutoff serves to reduce the number of mistakes. In Section 6.4.2 we show that it can also have another significant effect on performance.

Choosing PORT Parameters

The value of the PORT parameters (MinLen, LenRep, SkipRep, and MaxDist) depend directly on the expected workload. If the workload is always (or primarily) of one type of pattern, the parameters may be chosen to optimize for that pattern. With a more general workload, the ability to vary these parameters may make PORT more flexible. We discuss each parameter in turn. Note that all parameters must be at least 1.

MinLen should be small (1 or 2) unless the patterns in the workload suggest avoiding prefetching in small portions. One example of such a workload is one consisting of two types of portions, some tiny (say 3 blocks) and some huge (say 1000 blocks). Here, MinLen=4 would defer prefetching until the portion was confirmed to be long. Preliminary experiments indicate that a MinLen of 1 was a good choice to allow prefetching to begin immediately. MinLen=2 was a slightly more conservative choice that was rarely much better than MinLen=1. Note that a random pattern (single blocks at random locations) does have regular portion lengths, and PORT will not prefetch past the end of a regular portion length. Thus, MinLen=1 does not cause excessive prefetching in random patterns, a major fault of some predictors.

LenRep should also be small, probably 1 or 2. If the portion lengths in the workload are always (or nearly always) of regular length, then choose LenRep=1 to quickly take advantage of this regularity. Otherwise, choose LenRep=2.

SkipRep should also be about 1 or 2. Choose SkipRep=1 if the workload exhibits fixed skip lengths. If there are random-access patterns in the workload, however, this will be a poor choice. Thus, for any workload with some irregular skip lengths, choose SkipRep≥2. SkipRep=2 should suffice.

MaxDist is a much more flexible parameter, and may be much larger. PORT never prefetches more than MaxDist blocks ahead of the current position. If MaxDist is too small, processor and disk idle time may be wasted for lack of prefetching work. If MaxDist is too large, the number of mistakes caused by an incorrect prediction may be high. Thus, the expected accuracy of the predictions is an important factor in selecting MaxDist.

It is difficult to determine the best MaxDist for a particular workload. For highly regular patterns, the prediction accuracy is likely to be high, so MaxDist may be large. For more variable patterns, MaxDist should be lower. See Section 6.6 for a discussion of the sensitivity of PORT to MaxDist.

6.2.4 ADAPT — Adaptive

This predictor also recognizes sequential portions, and is an attempt at handling **lrp**. It easily recognizes the regular portion lengths of **lfp**. It makes no use of any regular skips, however.

ADAPT is based on probability theory. At any given time ADAPT has some idea of the portion-length distribution. From this distribution and the current portion length, ADAPT computes the expected final length of the current portion given that the current portion is already a certain length. This is the conditional expectation of the portion-length distribution.[1]

Let L be the discrete random variable representing the portion length, and c be the length (so far) of the current portion. Then the expected portion length of the current portion, given that it is at least c, is given by

$$E[L|c] = \sum_x x\ P(L = x|L \geq c).$$

The formula for conditional expectation is from [Tri82]. This, by the definition of conditional probability, becomes

$$E[L|c] = \sum_x x \frac{P(L = x \text{ and } L \geq c)}{P(L \geq c)}.$$

We can then divide the range of x into two parts, based on c,

$$E[L|c] = \sum_{x<c} x \frac{P(L = x \text{ and } L \geq c)}{P(L \geq c)} + \sum_{x \geq c} x \frac{P(L = x \text{ and } L \geq c)}{P(L \geq c)}.$$

The first term drops out, since the numerator goes to probability zero. The second term can be simplified since $(L = x \text{ and } x \geq c)$ implies $(L \geq c)$. We now have simply

$$E[L|c] = \sum_{x \geq c} x \frac{P(L = x)}{P(L \geq c)}.$$

We now define the pmf and CDF of L:

$$\begin{aligned} p_L(x) &= P(L = x) \\ F_L(x) &= P(L \leq x) \\ &= \sum_{y \leq x} p_L(y). \end{aligned}$$

Then we note that $P(L \geq c) = 1 - F_L(c) + p_L(c)$ and that this is independent of x. Then the conditional expectation becomes

$$E[L|c] = \frac{1}{1 - F_L(c) + p_L(c)} \sum_{x \geq c} x\ p_L(x).$$

We wish to use this formula without advance knowledge of the distribution of L. We thus define the distribution of L by the portions seen before the current portion. ADAPT records enough information about the distribution to be able to compute $p_L(x)$ (for any x) and $F_L(c)$ easily, and computes the expected portion length every time c changes.

There is a case where the above formula does not work. When the current portion is longer than any prior portion (which is always true for the first portion), the function becomes infinite. In this case we use a simple distance predictor (as with PORT), shown below. Thus this algorithm is also affected by the MaxDist parameter.

$$\text{distance}(c) = \begin{cases} c & \text{if } c < \text{MaxDist} \\ \text{MaxDist} & \text{if } c \geq \text{MaxDist} \end{cases}$$

[1] The concept for ADAPT is loosely based on Smith's method [Smi78c].

This is equivalent to the formula

$$E[L|c] = \begin{cases} 2c & \text{if } c < \text{MaxDist} \\ c + \text{MaxDist} & \text{if } c \geq \text{MaxDist} \end{cases}$$

6.2.5 IOBL — IBL/OBL

IOBL is a hybrid predictor, combining IBL and OBL. It is IBL when started, and becomes OBL whenever an incorrect decision is made. This happens at the first portion break, if any. We pay a little overhead over each of the two algorithms for more generality. The success of this predictor depends on the workload consisting of one long sequential portion, or many short portions. IBL alone is probably better for a pattern with multiple long portions.

6.2.6 IPORT — IBL/PORT

IPORT is a mixture of IBL and PORT. PORT uses a distance function that is limited to MaxDist. IPORT uses the same function, but removes the limit when in the first portion. Thus, during the first portion, we nearly have IBL, although retaining some conservatism to avoid too many mistakes in multi-portion patterns. Once a portion break is noticed, IPORT becomes exactly PORT.

6.2.7 IOPORT — IBL/OBL/PORT

This hybrid is slightly more conservative than IPORT, in that it uses OBL instead of the linear function when in irregular portions. The prefetch distance function is the same as for PORT, with MaxDist=1 for irregular portions, and MaxDist=∞ for the first portion.

6.3 Experiments and Methods

In this set of experiments we evaluate all of our local predictors in terms of their ability to improve performance on our synthetic workload. We note the strengths and weaknesses of each predictor, and identify a predictor (or set of predictors) that work best for each pattern. We also try to determine whether any one predictor is generally useful for all workloads.

All predictors (including NONE and EXACT) were used with each local access pattern, computation ratio, and synchronization style. Each combination of these four parameters represented one test case. For each test case, we averaged the total execution time over five trials, and used this as our comparison measure. We recorded many other measures (Section 4.1.5) although we do not directly report them here.

For the ADAPT and the PORT family predictors, which required the MaxDist parameter, we chose MaxDist=5. This choice was fairly arbitrary. Section 6.6 examines the effect of MaxDist on the PORT-family predictors.

We used our standard workload set, but restricted it to local access patterns. This set included the **lfp**, **lrp**, **lw**, and **seg** patterns. We added the **rnd** pattern here since it is important for predictors to properly handle random-access patterns.[2] The synchronization styles were, as before, *each(10)*, *neighbor(10)*, *total(200)*, and *none*. Each test was repeated with computation simulated between block reads, averaging 30 msec per block.

[2]They need not handle global patterns, which are handled by global predictors (Chapter 7). The choice of a local or global predictor can also be automated (Section 7.7).

The other experimental parameters were the same as in our previous experiments. There were 20 processes and 20 disks. Each process read 200 records (blocks) for a total of 4000. There were 80 buffers in the cache, with up to 60 allowed for prefetched blocks. The record and block sizes were both 1 KByte. We used the standard prefetching mechanism with none of our other variations (i.e., no prefetching-for-others, no private prefetch buffers, and no minimum prefetch lead).

To use the results of these experiments to choose a predictor for a given workload, one must know the relative predominance of each of our pattern types in the given workload. Practical workloads likely do not have the same composition as our synthetic workload. Since the predictors respond differently to the patterns in our workload, we describe the results for each pattern separately. For unknown workloads, we define the requirements for a general-purpose predictor, and compare our predictors to find a good general-purpose predictor.

6.4 Results and Discussion for each Pattern

Our comparison measure is the total execution time. For each test case, we averaged the total execution time over five identical trials. The *cv* was always less than 6% (0.0–0.7 seconds except for one case). Thus, small differences in total execution time should be ignored. We present a separate graph for each pattern, including the maximum *cv* for that graph in its caption. Each graph plots the total execution times for all test cases in a horizontal bar chart. The graphs group the test cases by synchronization style, to allow easy comparison of the different predictors. The predictors are ordered so as to group the IBL family (IBL, IOBL, IPORT, IOPORT) and the PORT family (PORT, IPORT, IOPORT). The NONE (no prefetching) case is drawn as dotted lines, both vertical and horizontal, to facilitate comparison. Occasionally, a predictor was faster than EXACT. These anomalies are discussed in Section 6.4.2.

The results for **lw** are shown in Figures 6.1–6.2. The key to success with both **lw** patterns was IBL (one anomaly in **lw** with computation is described in Section 6.4.2). The performance was almost identical to EXACT, so here on-line prediction was successful. The hybrid predictors all attempted to mimic IBL on **lw** patterns and thus came close in performance. The worst performer was OBL, whose conservatism sharply limited the number of disks used simultaneously. This is an example of where a simple predictor useful for uniprocessor, single-disk systems was not sufficient for multiprocessor, parallel-disk systems.

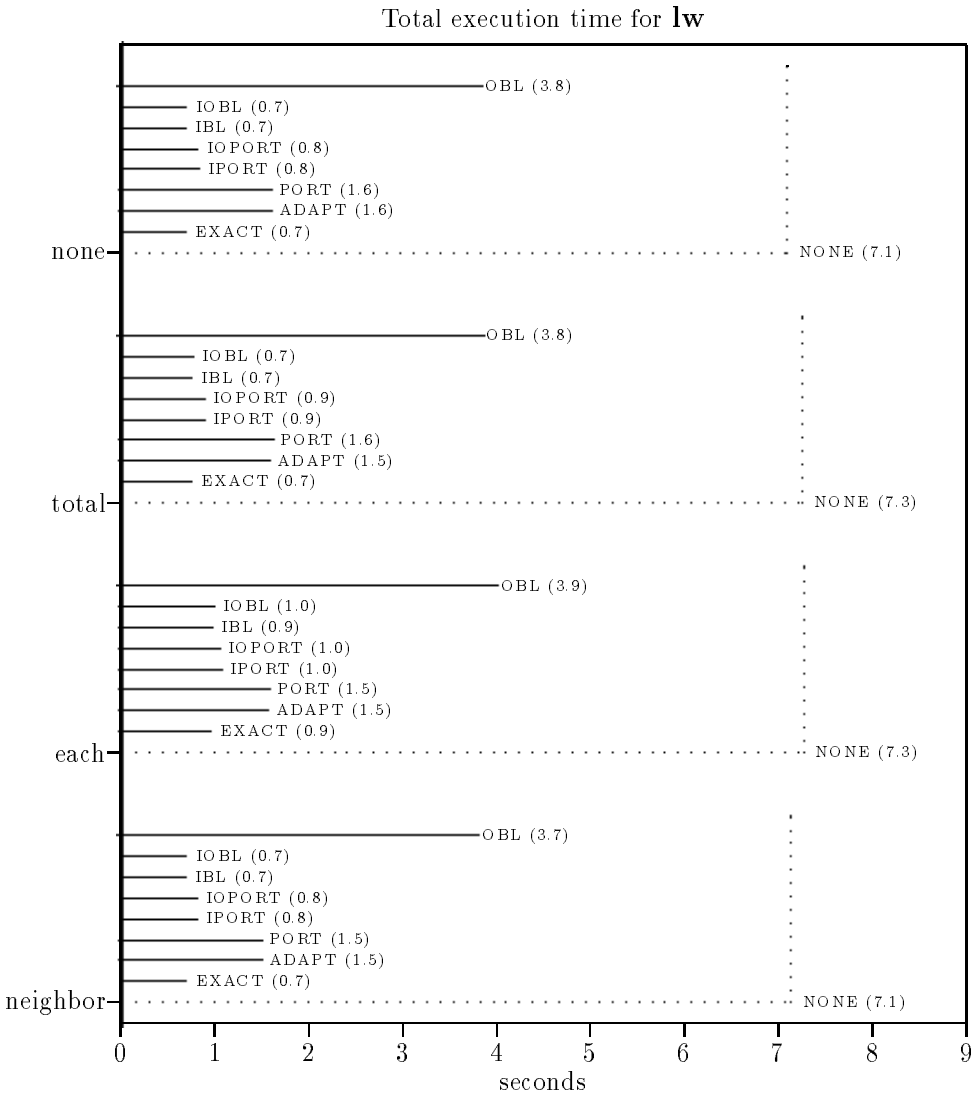


Figure 6.1: Total execution times for **lw**. ($cv < 0.047$)

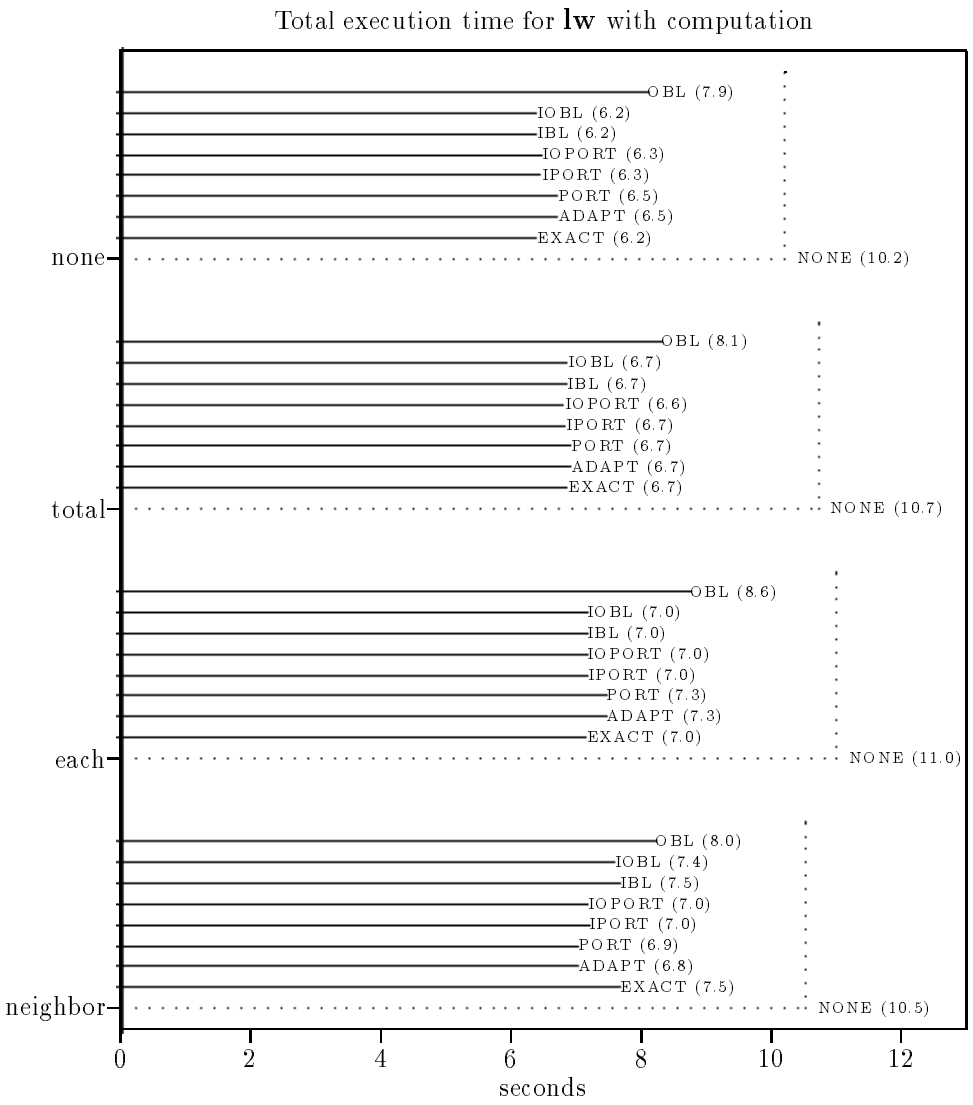


Figure 6.2: Total execution times for **lw** with computation. ($cv < 0.040$)

The results for **lfp** are shown in Figures 6.3–6.4. The most striking feature in Figure 6.3 is that IBL was much slower than the others. IBL made many mistakes at each portion skip. The IBL hybrids avoided most of the mistakes by switching to OBL or PORT at the end of the first portion. The PORT family and ADAPT are designed to recognize **lfp** patterns, and so they were more successful. Still, only small benefits (i.e., improvement over NONE) were obtained for **lfp**. With computation, however, the benefits were more significant. Here, the PORT family and ADAPT were essentially tied. All have the same ability to recognize the fixed-length portions, and avoid mistakes after the start of the third portion. ADAPT could not prefetch into future portions, like PORT, but that does not seem to have been a deficiency here.

There were several anomalies due to the greedy-process problem (Section 5.2.8), which are discussed further in Section 6.4.2. ADAPT was sometimes faster than the PORT family because its limitation to prefetching within a portion helped to solve the greedy-process problem. PORT was highly sensitive to the MaxDist parameter in this pattern (see Section 6.6). Indeed, with different (lower) MaxDist values, PORT avoided the problem and did as well or better than ADAPT. The greedy-process problem was also solved, and better performance obtained, with a large 200-block cache (see Section 8.2).

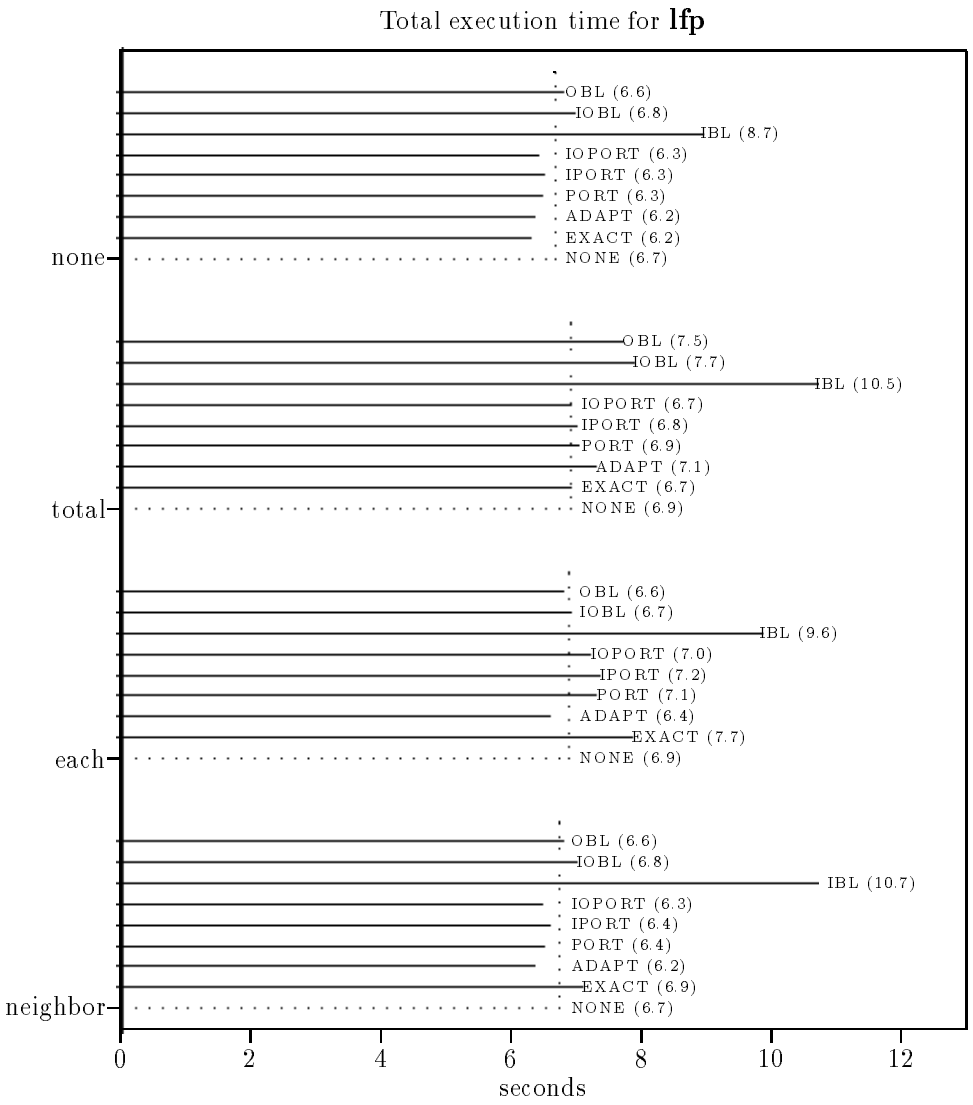


Figure 6.3: Total execution times for **lfp**. ($cv < 0.044$)

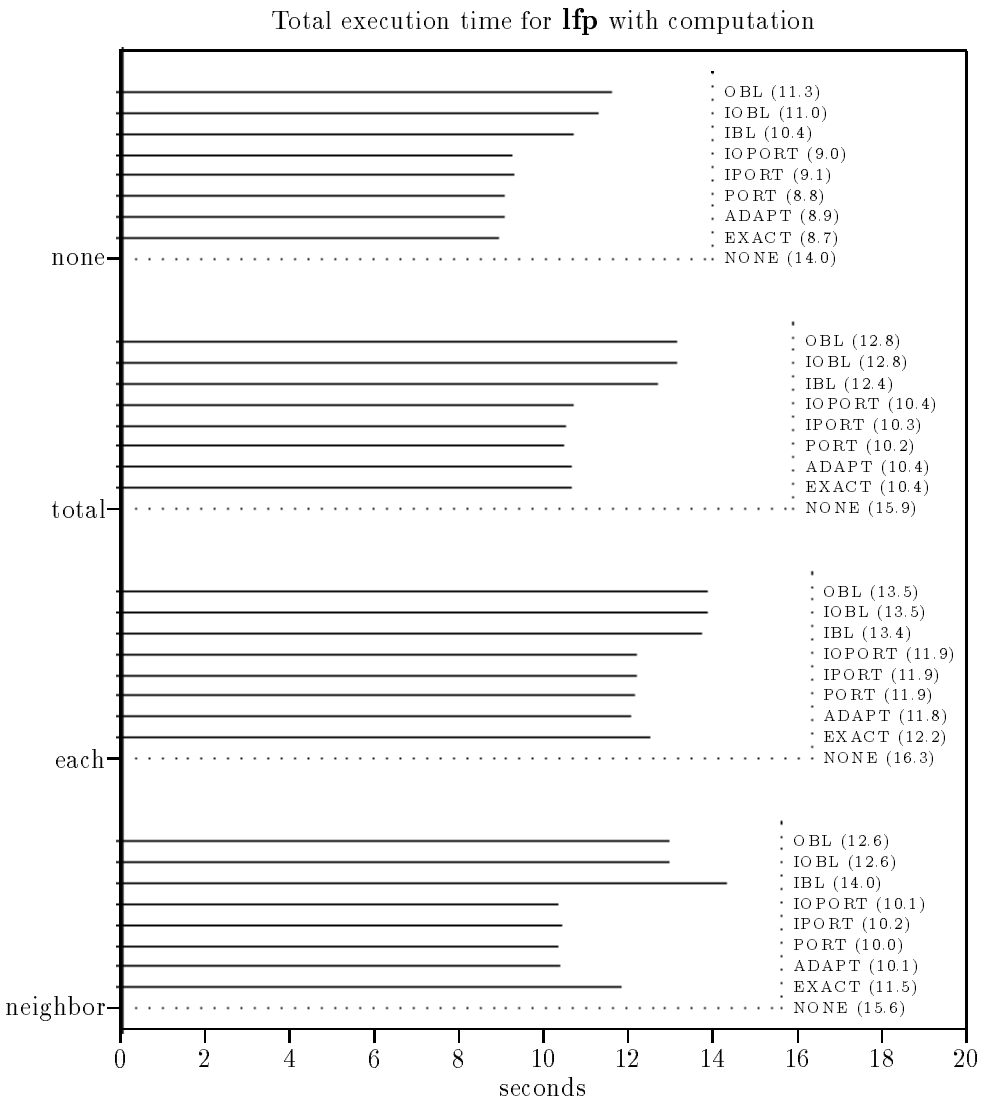


Figure 6.4: Total execution times for **lfp** with computation. ($cv < 0.038$)

In the **lrp** pattern (Figure 6.5), the OBL predictor performed better than less-conservative predictors (those that prefetched further ahead of the current position). Only the OBL family was significantly faster than NONE. More ambitious predictors (such as IBL) made too many mistakes in this hard-to-predict pattern. Thus, conservatism in the predictor was the key to success, validating OBL hybrids like IOBL and IOPORT. ADAPT seems to have been too slow to learn the nature of the portion-length distribution to make effective predictions.

In **lrp** with computation (Figure 6.6), the results were more mixed, and either PORT or OBL was the best choice. This pattern allowed more opportunities to overlap computation and I/O, and thus a more ambitious predictor (like PORT) had some success. In general, though, OBL was a safe choice for **lrp**. The hybrids IOBL and IOPORT, intended to mimic OBL in **lrp**-like patterns, came fairly close to OBL in most cases, and were thus reasonable substitutes. No predictor could match EXACT, which was immune to mistakes.

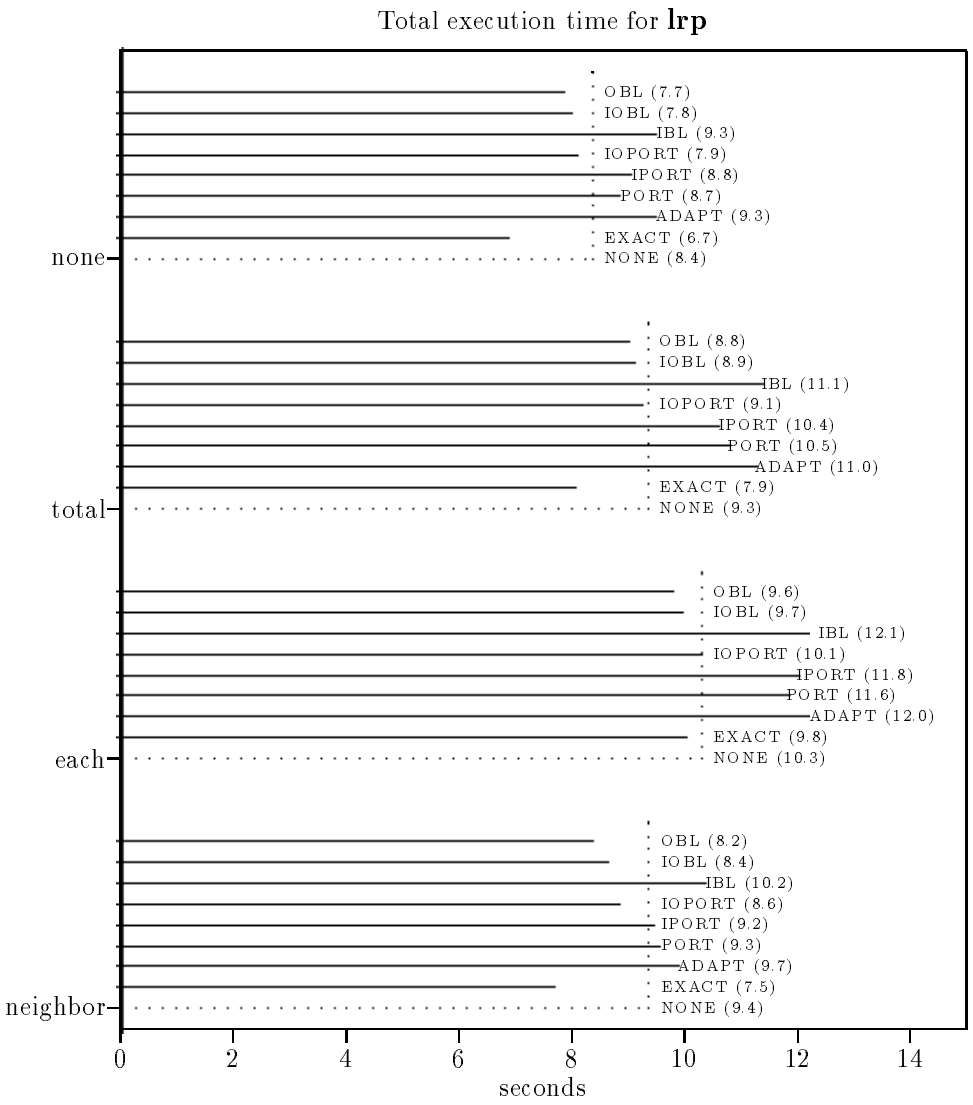


Figure 6.5: Total execution times for **lrp**. ($cv < 0.039$)

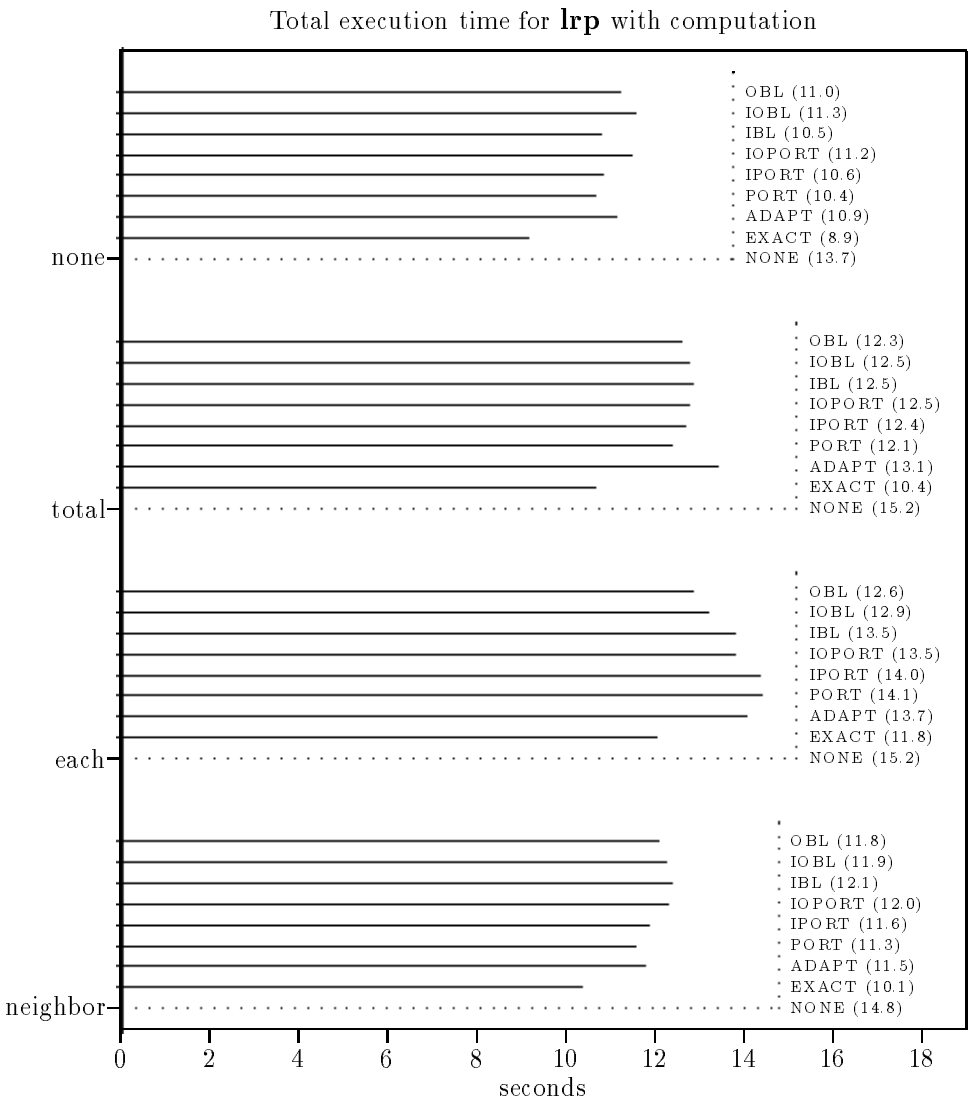


Figure 6.6: Total execution times for **lrp** with computation. ($cv < 0.030$)

The EXACT predictor is not used with the **rnd** pattern (Figure 6.7), since no prefetching is reasonably possible; here, EXACT is conceptually equal to NONE. The **rnd** pattern required a predictor more intelligent than OBL or IBL, which both performed poorly due to mistakes. ADAPT or any of the PORT family sufficed. In any random pattern, any prefetching is usually wasted, and thus only slows down the computation. A predictor must shut off prefetching to be successful with **rnd**. The results for **rnd** with computation are similar (not shown).

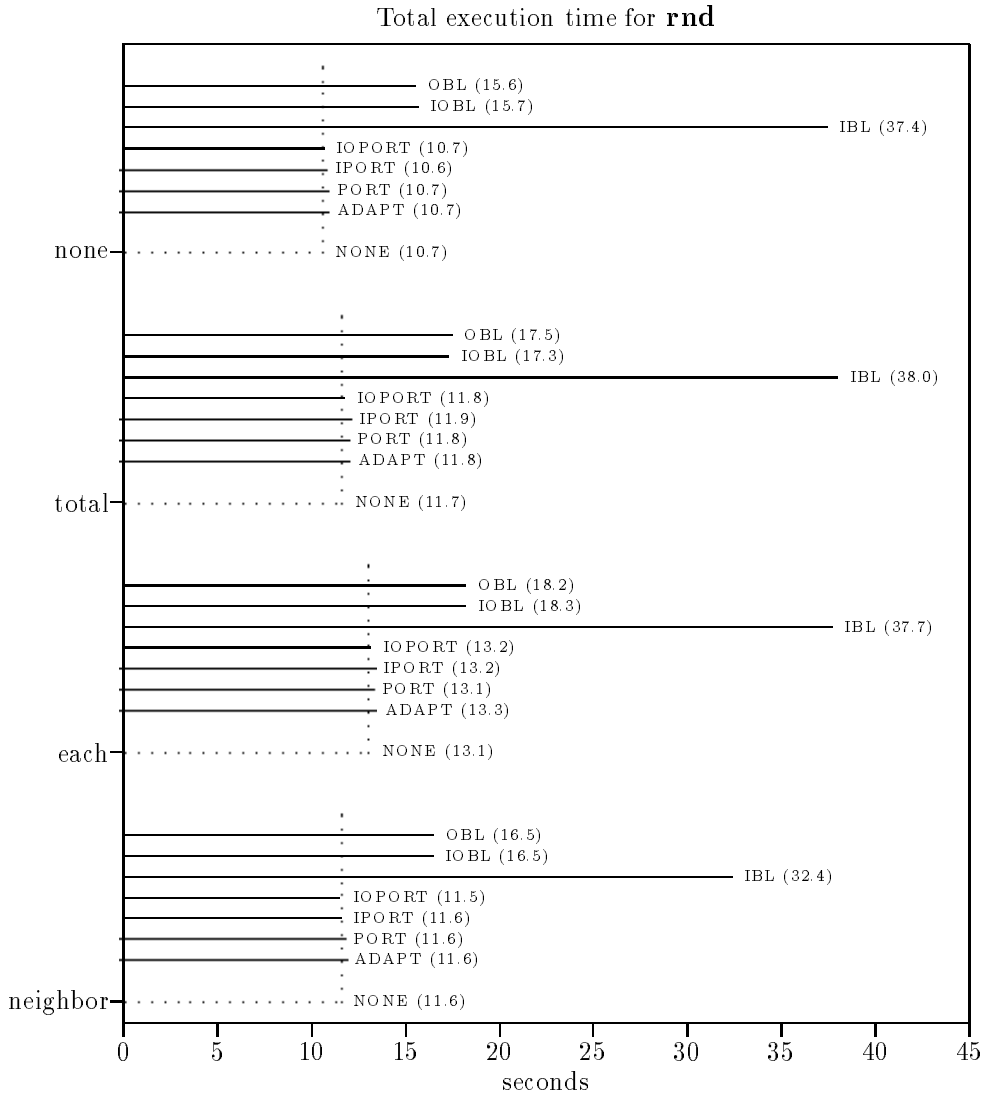


Figure 6.7: Total execution times for **rnd**. ($cv < 0.040$)

The **seg** pattern was difficult to handle, since the disk access pattern is critical to performance (Figures 6.8–6.9). The disk access pattern was more important than the prediction accuracy or other factors. In the **seg** pattern, all processes began their round-robin disk access pattern on the *same* disk. This caused severe initial contention, followed by a neat pipelining of processors through the disks. Synchronization and prefetching both interfered with this pipeline. It turns out that a larger cache helped to handle disk contention by allowing more prefetching; much better performance was possible (Section 8.2).

When the disk access pipeline was reset at each synchronization point, as in *each(10)* synchronization, the large amount of prefetching allowed by IBL predictors helped to spread out the disk accesses after the synchronization point, reducing disk contention. Thus, the IBL family was best for *each(10)*. The conservative OBL was best for the *total-* and non-synchronized cases, maintaining a neat pipelined access pattern. MaxDist-limited predictors PORT and ADAPT were similar to OBL for *neighbor(10)* and *none*. Adding computation (Figure 6.9) allowed more potential for overlap between computation and I/O, and the conservatism of OBL was no longer necessary. PORT and ADAPT were generally best, except for *each(10)*, where the IBL family was still best.

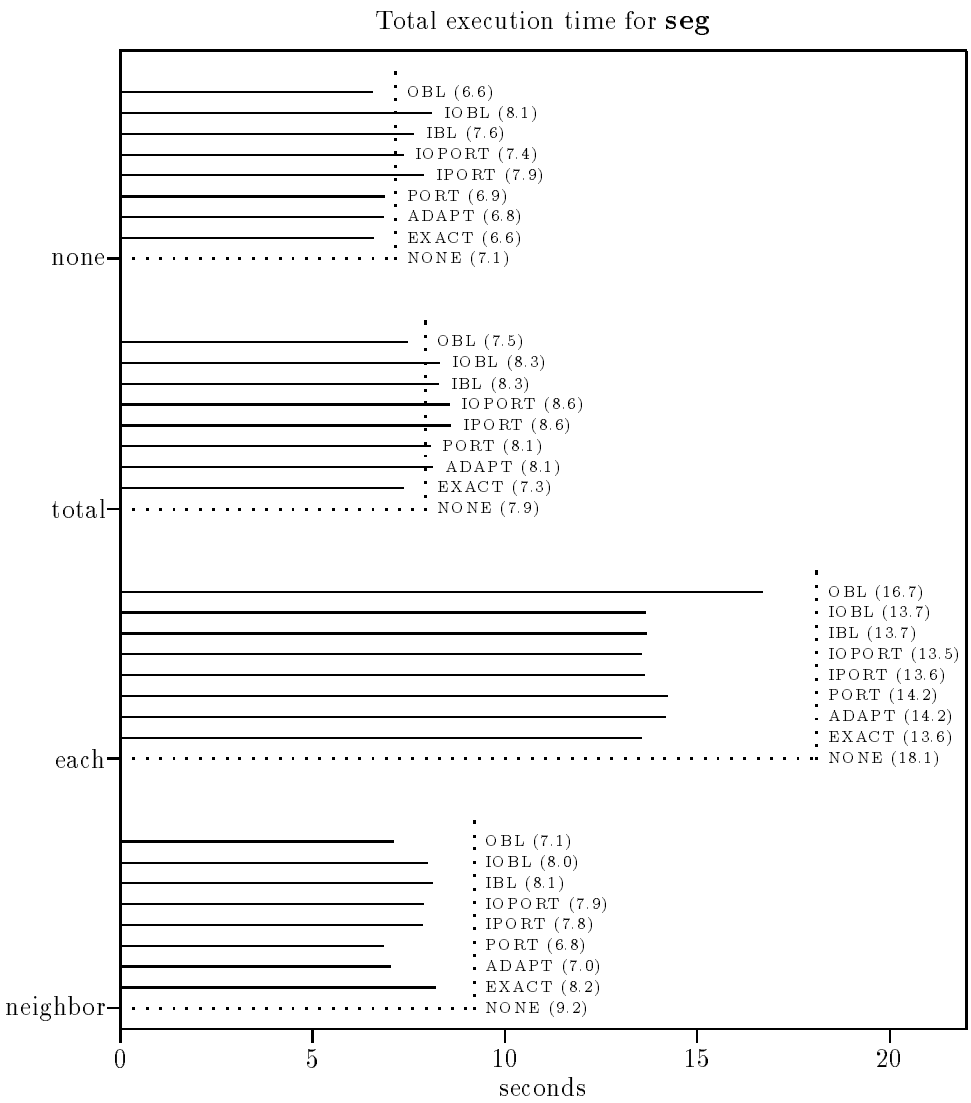


Figure 6.8: Total execution times for **seg**. ($cv < 0.051$)

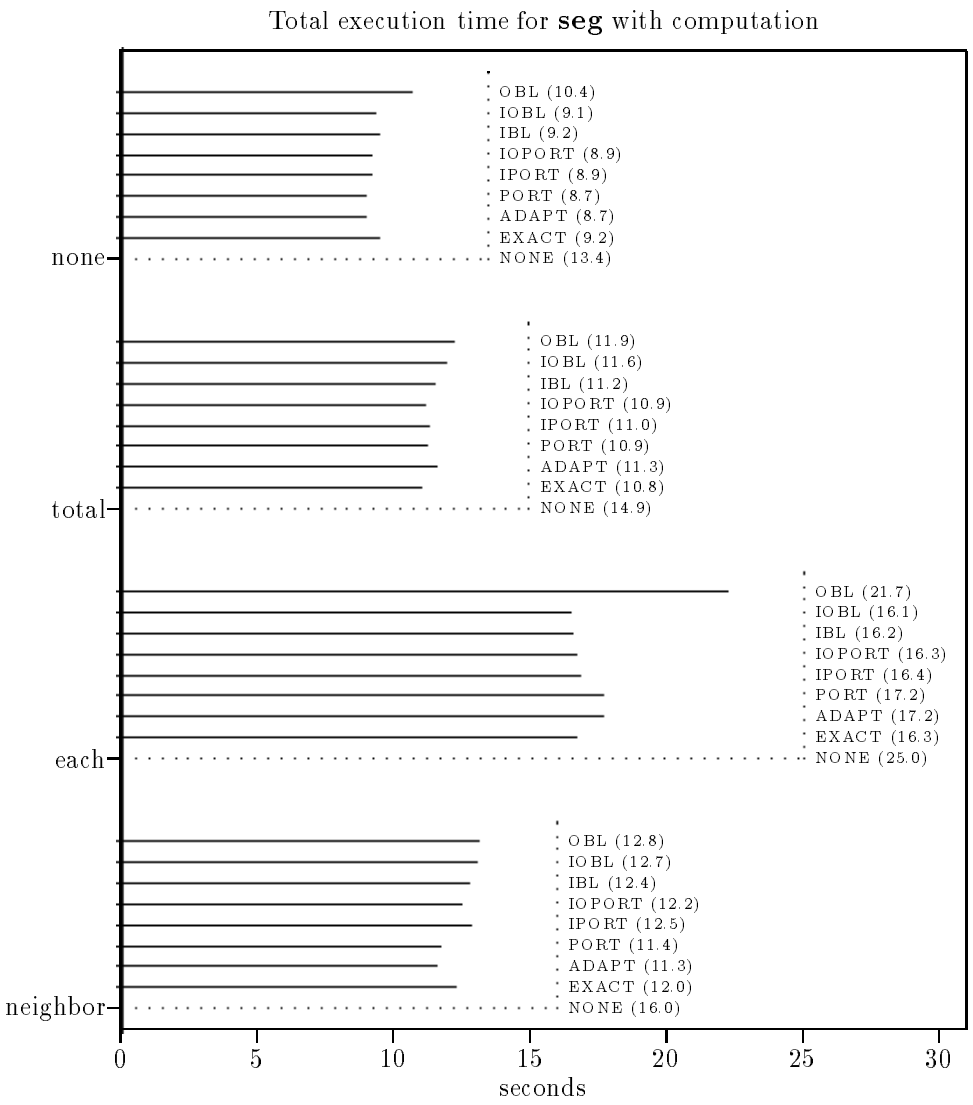


Figure 6.9: Total execution times for **seg** with computation. ($cv < 0.058$)

A slightly different **seg** access pattern, reading 201 blocks rather than 200 blocks per process, starts each process on a different disk. Thus, the pipeline is explicitly built into the pattern. The results for this pattern, **seglong**, are shown in Figure 6.10. OBL was consistently the fastest predictor. OBL did not disturb the pipeline, especially under the tight *each(10)* synchronization, while still prestaging accesses to keep the disks fully utilized. In comparing with Figure 6.8, it is interesting to note that this pattern, though longer, runs faster, because of the explicit pipeline. As before, the disk access pattern is more important than the ability of the predictor.

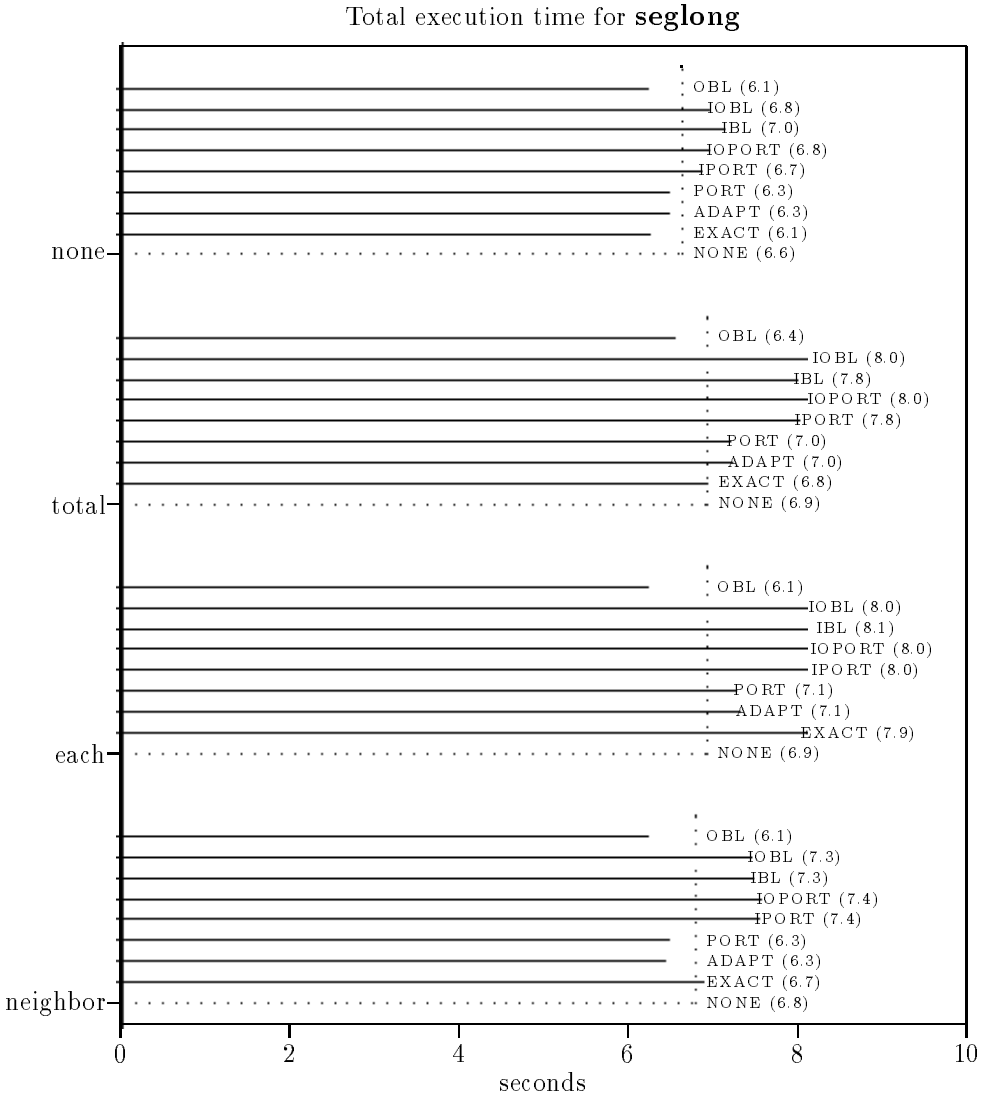


Figure 6.10: Total execution times for **seglong**, which is a slightly longer pattern. ($cv < 0.058$)

When a variable amount of computation accompanied each block read in the **seglong** pattern, the pipeline was disrupted and the advantage of OBL was lost. In each test case, PORT was at least equivalent (within measurement error) to the best predictor for this pattern, shown in Figure 6.11. More than one-block lookahead was clearly necessary here. Indeed, for all but the *neighbor*-synchronized cases all on-line predictors except OBL were similar. The anomaly in the *neighbor*-synchronized tests is similar to that in the **seg** pattern with computation, which is discussed in Section 6.4.2.

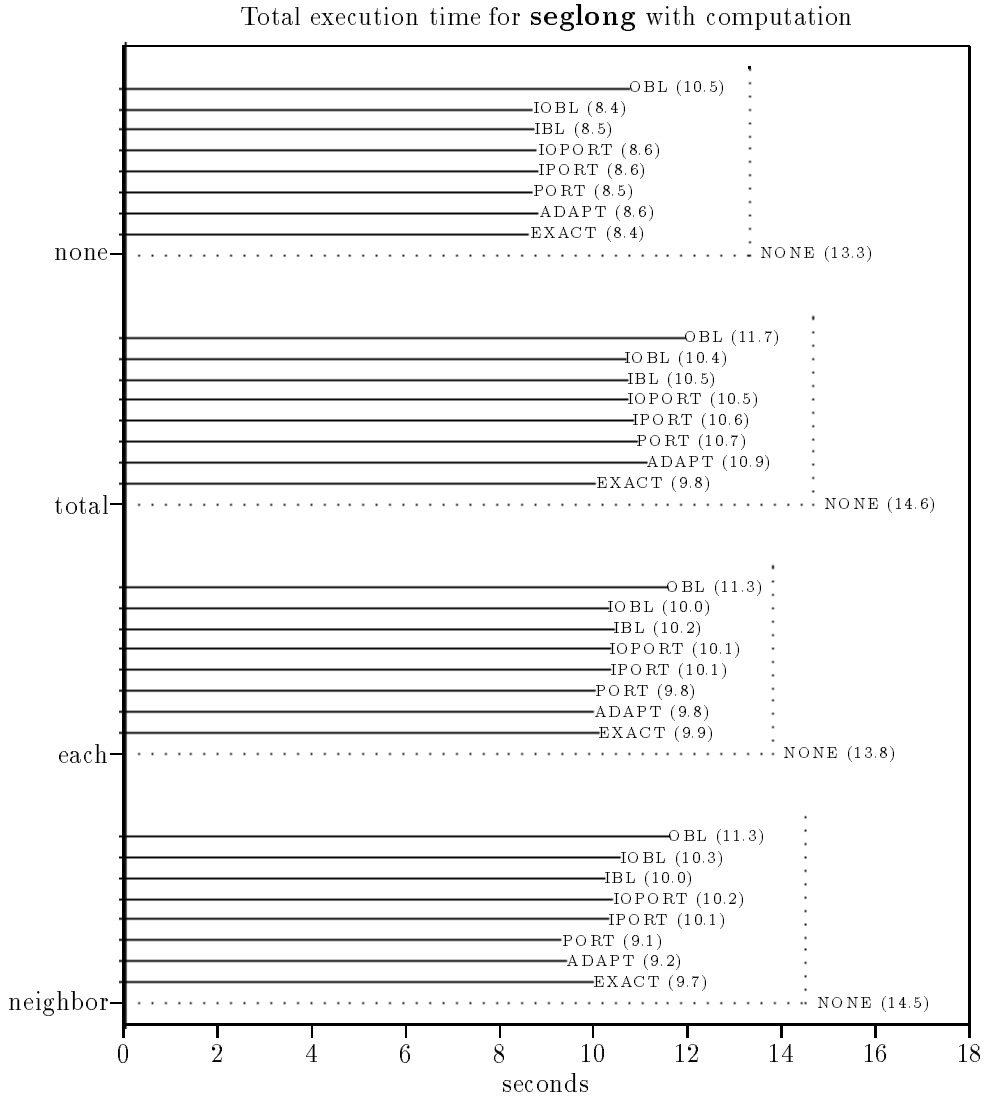


Figure 6.11: Total execution times for **seglong** with computation. ($cv < 0.042$)

6.4.1 Choosing a General-purpose Predictor

The determination of the "best" predictor clearly depends on the relative importance of different access patterns in a particular workload. Given knowledge of the workload, the preceding discussion helps to choose the best predictor. With a highly-mixed workload, or no knowledge of the workload, a general-purpose predictor is necessary. A general-purpose predictor should work reasonably well on all workloads, and provide high performance to most of the access patterns encountered. We designed the hybrid predictors in the search for a general-purpose predictor. Although our synthetic workload is not necessarily typical, it is broad enough to encompass the many kinds of patterns that may be found in practical workloads. We thus use our synthetic workload to evaluate our predictors: a general-purpose predictor will handle all test cases reasonably, and most test cases well.

Our first comparison measure was the percent deviation of each predictor's time from that of the best on-line predictor. Thus, if the total execution time on a given test with on-line predictor i is t_i, and the best time t_b is defined to be the minimum t_i, the percent deviation for predictor i is

$$d_i = \frac{t_i - t_b}{t_b} \times 100\%.$$

If the deviation for a predictor were zero in every test case, then the predictor would be the best choice for every test case. This is unlikely, but we can expect a general-purpose predictor to have many deviations near zero without any that are high. Since the run time for each predictor was averaged over five trials, and the run time often varied by about 5%, deviations of this magnitude were indistinguishable from noise in the data. Thus a deviation of less than 5% should be considered to be essentially no deviation.

The collection of deviations for each predictor forms a distribution. We present each distribution in a common graphical form called the *box plot*, which allows inspection of five key points: the minimum, maximum, median, and lower and upper fourths. These five points divide the distribution into four parts, each representing one quarter of the data points in the distribution. The position and size of these parts summarize the shape of the distribution.

The box plot of the deviation distribution for each predictor is shown in Figure 6.12. Each distribution is given on a separate line, with a circle representing the median, and a line on either side representing the upper and lower fourths of the distribution. In this particular plot, the lower fourth is barely visible, if at all. The OBL and IBL distributions are cut off on the right, since their maximums are large. The predictors are grouped somewhat to keep the PORT family (PORT, IPORT, IOPORT) and the IBL family (IBL, IOBL, IPORT, IOPORT) together for easy comparison.

From Figure 6.12, ADAPT and the PORT family had the lowest median deviations (less than 3%). IPORT and IOPORT had the additional advantage of a low maximum (29%), meaning they were never more than a third slower than the best predictor for any test case. There is little information here to distinguish IOPORT from IPORT. Note that the simple OBL and IBL predictors were poor general-purpose selections.

A similar comparison measure computes the percent deviation from NONE. This is the negative of the percent improvement due to prefetching. In Figure 6.13, negative percent deviation represents improvement due to prefetching, and positive deviation represents a slowdown due to prefetching. IBL is again cut off on the right. Although all on-line predictors had similar medians (12 to 24% improvement), IOPORT had the lowest maximum (8.9% slowdown), and the IBL family had the lowest minimum (88–90% improvement). Here the difference between IOPORT and IPORT is more clear, since IPORT had a higher maximum deviation. IOPORT was thus a good choice for a general-purpose predictor.

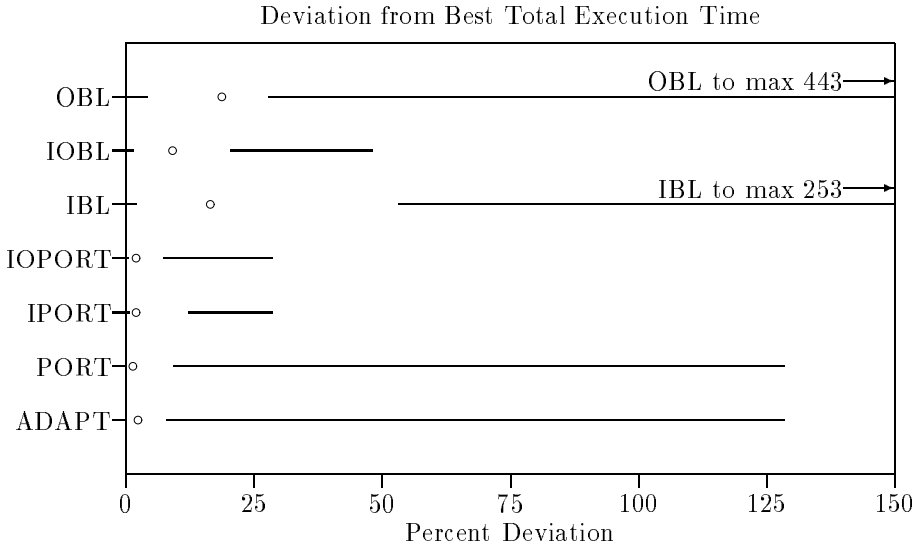


Figure 6.12: A box plot of the percent deviation from the best for each test case. The lowest medians belong to ADAPT and the PORT family, but the lowest maximums to the hybrid PORT patterns.

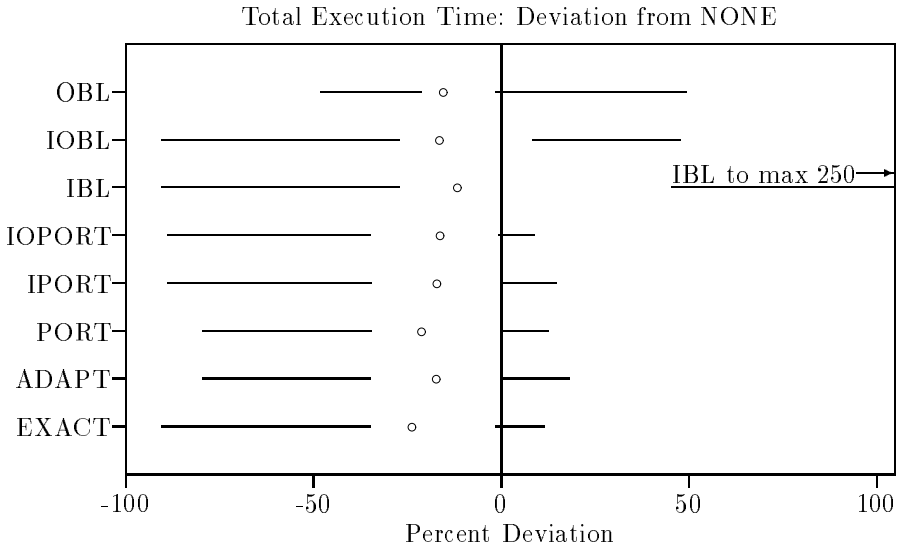


Figure 6.13: A box plot of the percent deviation from NONE.

A box plot of the predictors compared to EXACT (not shown) was of little use due to several anomalies, where on-the-fly predictors beat EXACT. EXACT was a perfect predictor in terms of accuracy, but was unfortunately not a perfect "best case" for comparing results.

6.4.2 Anomalous Cases

In some cases an on-line predictor actually beat the EXACT algorithm. The reason varied with the case, but each depended on an effect separate from prediction issues. We discuss each effect separately here.

The Greedy-Process Problem

This problem affecting the **lfp** pattern involved an imbalance in the benefits of prefetching. One or more processes prefetched an inordinate amount for themselves, using all of the buffers and slowing down the other processes (page 47). With explicit inter-process synchronization the computation as a whole was slower. This effect was responsible for the anomaly in the **lfp** pattern with *each(10)* and *neighbor(10)* synchronization, with or without computation. With some mechanism for balancing the benefits of prefetching, the performance of EXACT would improve. For example, the MaxDist cutoff in PORT restricted the prefetching of individual processes, allowing all processes a chance to prefetch. By similarly restricting the EXACT predictor, its performance improved. This points out that perfect prediction does not guarantee the best performance.

As an example, consider **lfp** with computation and *neighbor(10)* synchronization. With EXACT it completed in 11.5 seconds whereas PORT and ADAPT took 10.0 and 10.1 seconds respectively. Varying the MaxDist parameter demonstrates the dependence:

	MaxDist					
Algorithm	1	5	10	20	40	60
PORT	12.5	10.1	10.7	11.3	11.8	11.6
ADAPT	10.0	10.1	10.0			

Once we removed the low MaxDist restriction from PORT, its run time climbed to match that of EXACT. ADAPT, by its nature, was limited to prefetch within the portion. A similarly restricted EXACT had a total time of 10.2 seconds. The other **lfp** anomalies can be explained in the same way. The effect of MaxDist on PORT is discussed further in Section 6.6.

The lw Phase Problem

In the **lw** pattern, all processes read the same set of blocks in the same order. As long as they all read them at about the same time, the cache ensured that each block was only read from the disk once. If the processes became spread out over the file, reading a wide range of blocks, blocks may have been read from disk, used, and flushed from the cache before they were needed by some process. This process must then reread the block from the disk.

In some cases synchronization, or a balanced load, ensured that the processes stayed together. With variable computation per block, and with the loose *neighbor(10)* synchronization, the processes spread out. In this case, EXACT ran in 7.5 seconds (275 disk reads), while PORT and ADAPT took 6.8 or 6.9 seconds (200 disk reads). In fact, only OBL, PORT, and ADAPT managed to stay together. Again, it was the MaxDist restriction that solved the problem.

One possible solution to this problem is to predict the number of uses of each block from the number of uses of previous blocks in the pattern. Then only flush blocks from the cache when

they have been used the predicted number of times. This would require some flexibility, since non-integral record sizes would cause inequalities in the number of uses of different blocks. In addition, if the prediction is incorrect, another mechanism is needed to expire the block.

In a more exact solution to this problem, processes record their "interest" in a block whenever they predict the block. We add a counter for each block to count the processes interested in the block. The counter is incremented when a process selects the block, and decremented when a process uses the block. The first interested process also prefetches the block. The block remains in the buffer until the interest count drops to zero. This has benefits whenever many processes try to prefetch a block, either correctly or incorrectly. This mechanism is not guaranteed to solve the problem with **lw**, since it is possible for a process to not mention its interest in a block until after the block is prefetched, used, and flushed by other processes.

When the tests for **lw** with computation and *neighbor(10)* synchronization were repeated with the latter solution in place, the problem disappeared. Only 200 blocks were read in all cases. Although NONE and OBL were not affected, all of the other predictors managed to lower their time to 6.8 seconds. Although this mechanism seems like a desirable addition, it is not the default in the rest of our experiments.

Disk Access Pattern Details

The **seg** pattern produced a most difficult disk access pattern: all processes read from the disks in a round-robin fashion, and all began on the same disk. This could cause a lot of disk contention, and the effect of the predictor (or synchronization) on this pattern could mask the efficiency of the predictor.

Without prefetching, the processes fell into a pipeline. With prefetching, this pipeline often never formed and the accesses became more spread out. With a lot of prefetching, the access times became highly variable, the synchronization times increased, and the computation slowed down.

For example, in **seg** with *neighbor(10)* synchronization, EXACT required 8.2 seconds, while PORT and ADAPT needed only 6.8 and 7.0 seconds. We devised some special variants of the prediction algorithms: the first was EXACT with a maximum distance restriction on each process; the second was PORT with a distance function fixed at MaxDist. Note that changing the PORT limit to a fixed number, instead of the varying function, only affected prefetching at the beginning of the pattern, since there was only one portion in this pattern. The effect of the limit parameter is shown for several algorithms below.

	Limit					
Algorithm	1	5	10	20	40	60
Restricted EXACT	7.0	7.2	7.7	7.8	7.8	7.8
Fixed PORT	7.3	6.8	7.6	7.8	8.0	8.1
PORT	7.1	6.9	7.8	7.8	7.7	7.8
ADAPT	7.3	7.0	7.5	7.7	7.8	7.9

EXACT was improved by sharply restricting its prefetching (to OBL, essentially). PORT and ADAPT were best with a moderate limit of 5. Thus, a local limit on the prefetching distance led to the best performance. Indeed, the pattern with no limit on the prefetch distance, IBL (and its hybrid variants), had the poorest performance for this synchronization style.

Similar results were found for **seg** with computation and *neighbor(10)* synchronization. The same explanation does not work, however, for the unsynchronized case, which also appeared anomalous. The above explanation for the anomaly blames the slowdown on increased synchronization delays. With no synchronization this cannot happen. Using the same special set of experiments

we found that the Fixed PORT predictor slowed down to match EXACT, and Restricted EXACT was no faster. Thus, we believe that it was the relatively conservative startup used by PORT and ADAPT that allowed the pipeline to be primed correctly, giving better performance.

6.5 Overhead

One measure of the overhead of the predictor was the notification time (page 23), the amount of time used to notify the predictor of each block that was referenced. Since this was time out from processing a read, not overlapping some otherwise idle time, it was important to keep the notification mechanism efficient. Another part of the predictor overhead came during prefetching, when the predictor was asked for a prediction.

The predictors did most of their work during notification and quickly provided predictions when they were requested. If there were any blocks prefetched by mistake, the notification procedure arranged for their removal from the cache. This could take a long time, and mask the pure notification time of the predictor.

We recorded the notification time for each test case, averaged over the 4000 references in the test. In Figure 6.14, the distribution of these average notification times is shown for each predictor. Some cases had large notification times, from processing many prediction mistakes. This was particularly apparent when the IBL predictor was used with the **rnd** pattern. For all predictors, however, the median notification time over all test cases was less than 0.2 msec, and the lower fourth was under 0.02 msec. These represent the test cases where few mistakes were made, and indicate that the notification overhead was usually negligible. The cost of mistakes, however, was occasionally quite high.

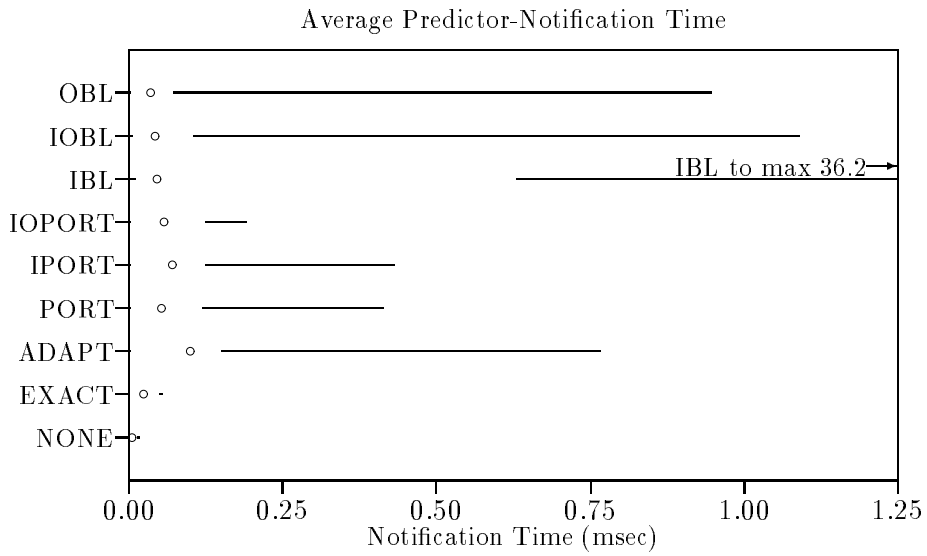


Figure 6.14: The average time needed to notify the predictor for each block referenced.

6.6 The Sensitivity of PORT Predictors to the MaxDist Parameter

MaxDist is an important parameter of all the PORT-family predictors, since it controls the amount of prefetching by limiting predictions to the near future. An understanding of MaxDist's effect allows us to better understand PORT, and to tune PORT for particular workloads. To determine the sensitivity of PORT to MaxDist, we experimented with a range of MaxDist values (1 to 20) for every test case and all three PORT predictors (PORT, IPORT, IOPORT). Although the data are too numerous to show here, we summarize the results below, using total execution time as the performance measure. The general behavior was often difficult to characterize, implying no simple relationship between MaxDist and PORT performance, but we present the key observations.

Note that MaxDist was inherently ignored in many cases: in the IPORT and IOPORT predictors for **lw** and **seg** patterns, since they were in IBL mode, and in the IOPORT predictor for **lrp** patterns, since it was in IBL and then OBL mode. MaxDist was irrelevant to all three predictors in the **rnd** pattern.

lw: MaxDist is only relevant to PORT. The total execution time decreased steadily with increasing MaxDist, slowly approaching the constant performance of IPORT and IOPORT. Since for **lw** PORT with a large MaxDist is essentially the same as IPORT, which in turn is the same as IBL, this behavior is no surprise. The **lw** pattern needs a lot of prefetching, and IBL (or an imitation) is the best way to do it. The same results hold with and without computation.

lfp: MaxDist is relevant to all three predictors. The **lfp** pattern is easily predicted, once the regular portions are recognized. Thus, early conservatism is warranted until the regularity is detected, and then aggressive prefetching can be used for the bulk of the pattern. However, the greedy-process problem (Section 5.2.8) is always an issue with **lfp**. Here, a low MaxDist helped to limit prefetching and avoid the greedy-process problem. In most cases, execution was fastest with MaxDist=1, and then slowed down steadily for increasing MaxDist, as the greedy-process problem became more significant. It did level off around MaxDist=10, when other factors (such as the prefetch limit or prefetch overhead) limited prefetching. The exception was *total(200)* synchronization, which was slow for MaxDist=1 and for high MaxDist, but had a sharp minimum at MaxDist=3. Note that 3 is exactly a single process's share of the prefetch limit; thus, MaxDist=3 is essentially an implementation of private prefetch limits (PPL; page 47), one of our earlier solutions to the greedy-process problem. Although there are better solutions to the problem (e.g., PFO or a larger cache), PPL may be implemented with MaxDist equal to the prefetch limit divided by the number of processes.

With computation, however, the conservatism of MaxDist=1 (OBL) was not as successful. Adding computation to the pattern provided an opportunity for prefetching to overlap computation and I/O. Thus, a successful prefetch was even more beneficial, and a mistake was less costly. All three predictors were worst at MaxDist=1, speeding up significantly with increasing MaxDist until about MaxDist=5 (MaxDist=3 for *each(10)*). Thus, the benefits of the added prefetching were important. Beyond MaxDist=5, *total(200)* and *none* leveled off, but *each(10)* and *neighbor(10)* slowed down slightly, showing a return of the greedy-process problem.

lrp: MaxDist is not relevant to IOPORT, which is in IBL and then OBL mode. The PORT and IPORT predictors were equivalent to IOPORT at MaxDist=1, where they all have the conservatism of OBL, which was successful in this hard-to-predict pattern. From there, however, PORT slowed

down steadily with increasing MaxDist, as more predictions were mistakes. It leveled off around MaxDist=10. Thus for **lrp** the best MaxDist was 1.

As before, conservatism was a poor policy when there was computation to overlap with I/O. PORT was similar to IPORT and IOPORT at MaxDist=1, sped up for moderate MaxDist values (around 3), and then slowed down for larger MaxDists. Thus for this pattern a moderate MaxDist of 3 was best, representing the moderate predictability of **lrp**.

seg: MaxDist is only relevant to PORT, since the others are in IBL mode. The results for PORT were complicated, and highly dependent on the synchronization style. The root of all explanations is **seg**'s difficult disk access pattern. Each process began its round-robin disk access pattern on the same disk, so there was a lot of disk contention. Under the right conditions, the processes begin a pipeline through the disks, which can be successful. Strong synchronization (such as *each(10)*) or aggressive prefetching can disrupt the pipeline.

With no synchronization, MaxDist=1 limited prefetching and kept the pipeline intact. PORT was fastest at MaxDist=1, and slowed steadily as MaxDist increased, eventually matching IPORT and IOPORT (which are in IBL mode, similar to having a high MaxDist). For *each(10)* synchronization the opposite was true: PORT was slowest at MaxDist=1, and slowly sped up to match IPORT at higher MaxDist values. Here, the pipeline is emptied at each synchronization point, and the aggressive prefetching of a large MaxDist helped to fill it quickly after each synchronization. The *total(200)* and *neighbor(10)* synchronization styles, compromises in terms of synchronization, needed a compromise in MaxDist. PORT was faster than IPORT at MaxDist=1, but sped up around MaxDist=3 and then slowed to match IPORT for high MaxDist values. This kept the pipeline full without overly disrupting it.

As always, with computation more prefetching was necessary for best performance. Here, MaxDist=1 was *slowest,* in contrast to the above result. This corresponds to the poor performance of OBL on **seg** with computation. Except in *neighbor(10)* synchronization, PORT sped up with increasing MaxDist to match IPORT and IOPORT for large MaxDist. The *neighbor(10)* synchronization still needed a compromise, with PORT slow for low and high MaxDist, and a deep minimum at MaxDist=4.

Summary. There are several issues involved in MaxDist's effect on the PORT family of predictors. First, in some cases it is irrelevant. Second, since it controls the aggressiveness of prefetching, its effect corresponds to the predictability of the pattern, with highly predictable patterns needing a high MaxDist, and poorly predictable patterns preferring a low MaxDist. Third, predictability is sometimes not as important as other factors, such as the greedy-process problem or disk contention. Fourth, a pattern with computation needs to be less conservative than the same pattern with no computation, since prefetch I/O may be overlapped with computation. This overlap increases the benefit of success and decreases the cost of failure. Fifth, IOPORT had two advantages: it was often the fastest of the PORT family, and MaxDist was more often irrelevant. In some **seg** and **lrp** cases, it is possible to identify a MaxDist that gave PORT better performance than IOPORT, but this would be difficult to do dynamically. IOPORT was more generally successful and required less tuning.

6.7 Conclusions

It was no surprise that conservative predictors were important for random patterns, and ambitious predictors were better for regular patterns. OBL was best for **lrp**, and NONE (or a predictor smart

enough to do no prefetching) for **rnd**. PORT with a solution to the greedy-process problem worked for **lfp**, and IBL for **lw**. The results for **seg** were more dependent on the disk pattern than the predictor. We show in Section 8.2 that a larger cache significantly improves the performance of IOPORT on **lfp** and **seg**, so more prefetching success is possible.

Patterns with some computation allow for overlap between computation and I/O. The benefit of a successful prefetch can be much larger due to this overlap, and the relative cost of a mistake much less. Thus, less conservatism is necessary in predicting for patterns with more computation than in the patterns with no computation. In other words, when I/O-bound, a predictor should concentrate more on the I/O that must be done, and speculate less on the I/O that may be done.

The on-line predictors matched the performance of the EXACT predictor in many cases. All on-line predictors were less than 20% slower than EXACT half of the time, and were within a few percent of EXACT a quarter of the time. In some cases, due to effects like the greedy-process problem, the on-line predictors actually beat EXACT.

IOPORT appears to be the best general-purpose local-pattern predictor, in that it provided high performance to a wide variety of patterns without causing poor performance to any pattern. IOPORT was always within a third of the execution time of the best on-line predictor, and in half of our test cases was within 3% of the best predictor.

Chapter 7

Automatic Prediction in Global Patterns

7.1 Introduction

We have shown that locally-sequential access patterns can be predicted with sufficient accuracy and efficiency to allow prefetching to improve the run time of programs that use them. This chapter concentrates on the problem of recognizing and predicting globally-sequential patterns at runtime. To do so, we define several *global predictors*. Although the problem is more difficult, and the overhead is larger, our experimental results show that the benefits are still significant. In this chapter we compare several predictors on a fixed set of architectural parameters, to investigate the tradeoff between accuracy and efficiency, the impact of some workload and predictor parameters, and the overhead involved in prefetching. In Chapter 8 we examine the effects of different architectural parameters.

There are three primary challenges for a global predictor: first, to recognize sequentiality; second, to prefetch intelligently and recognize mistakes; and third, to have an efficient and concurrent implementation. The emphasis in local patterns is on intelligent prefetching, since recognition is relatively easy. In global patterns, the blocks in the pattern may be referenced in only a *roughly* sequential order, so that each block number may not be simply the previous block number plus one. Thus it is more difficult just to recognize sequential access. Our buffer-replacement policy, which requires mistakenly-prefetched blocks to be explicitly flushed from the cache, complicates prefetching with the need to recognize mistakes. Efficient, concurrent implementations are difficult due to the need for global decision making.

In any concurrent algorithm there is a tradeoff between accurate information and high concurrency. The algorithms we describe here involve extensive computation and cooperation between processes, and their implementations require significantly more overhead than any of the local predictors. 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.

In the next section we discuss some assumptions and theory behind our techniques. We outline our primary predictor in Section 7.3 and its implementation in Section 7.4. Some alternative predictors are presented in Section 7.5. The results of experiments using the global predictors are described in Section 7.6. A predictor for both local and global reference patterns is discussed in Section 7.7. We conclude in Section 7.8.

7.2 Theory

Globally-sequential access to a portion of a file (or perhaps the whole file) involves processes cooperating to read the portion so that all blocks of the portion are read, and the blocks are referenced in a more-or-less increasing order, with variations that are small compared to the overall pattern. With this in mind, we present some assumptions, definitions, and theorems we use as the basis for global predictors.

7.2.1 Assumptions

We make a number of assumptions about globally-sequential access patterns:

- First, we assume that the processes are all cooperating to read portions of the file. This is different from locally-sequential patterns, where the processes work independently. For simplicity, we do not consider the possibility of several groups of processes cooperating in groups to read independent portions of the file. We also assume that all processes continue to participate; no process drops out or runs arbitrarily slowly.

- We assume that the pattern is the result of either a *self-scheduled* assignment of work, or more strictly, a round-robin assignment of work. In the self-scheduled method, processes choose records to read from the file based on a shared variable (or similar method) that atomically and incrementally assigns work to the processes. The round-robin assignment needs no shared variable, since the record choice is dependent on a regular, well-known pattern, where p processes read every pth record. In any case, the specific assignment method is not known to the file system.

- We assume that the file system is only aware of references to file system blocks. Although access to the actual byte ranges accessed by the processes may provide other information (such as record size), this is often obscured by user-level buffering anyway (for example, the *stdio* package).

- The above assumptions imply a series of block numbers that is nondecreasing. Although the file system may not see a list of block references that is smoothly increasing with time, we can expect the reference string from each process to be nondecreasing. A process experiences a *jump-back* when it references a block number less than its previous reference. Jump-backs are useful to global predictors, since they indicate the end of a sequential portion.

- Another assumption is part of the definition of global portions. The global access pattern within a sequential portion references all blocks of that portion. By definition, the missing blocks would divide the reference stream into smaller sequential portions. This gives rise to *completeness:* all blocks in a portion are eventually accessed.

- The processes are reading records, which may or may not be of constant size. Although our experiments use constant-size records, our theory, algorithm, and implementation allow for variable-size records. Since highly variable record sizes make sequentiality hard to recognize, the record sizes must be constant or nearly constant for the method to work effectively. In other words, we are not optimizing for wildly variable record sizes in this work.

- We also assume that the record size is at most a few file system blocks. Larger record sizes appear to the file system to be locally sequential portions, since a process reads several consecutive blocks. We thus leave large record sizes to the local predictors.

We believe that these assumptions are reasonable, and encompass many kinds of global access patterns.

7.2.2 Zones of Activity

Combining these assumptions and observations allows an understanding of some important features of globally-sequential access patterns. Imagine several processes cooperating to read a single sequential portion. The reference string from each process is *ordered*, that is, it has no jump-backs. At any time the next block number for a given process will be greater than or equal to its last-referenced block number (called *last*). Overall, the next block from any process must be greater than or equal to the minimum of the set of *last* blocks from all processes. We call this minimum block number *minlast*. There is also a corresponding *maxlast*, which is the maximum *last* of any process.

These two values define three zones of activity (see the example in Figure 7.1). The *old zone* is a range of blocks that will not be referenced in the future. From our assumption of completeness, these blocks have all been referenced. The *active zone* contains the set of blocks between *minlast* and *maxlast*, inclusive. Some of these blocks may have already been referenced. The rest are likely to be referenced soon. The *future zone* contains blocks that will be referenced in the near- and far-future. None of these have been referenced yet. Note that the zones change as blocks are referenced and the processes' *last* values change.

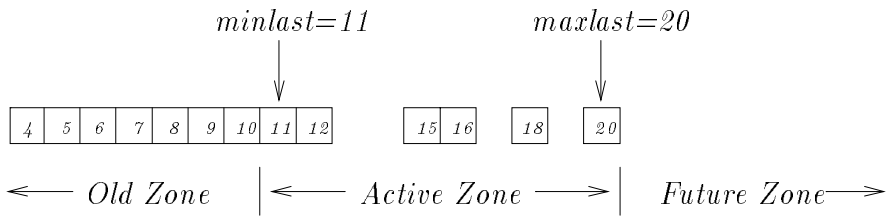


Figure 7.1: The three zones of activity. The numbered squares in this example represent blocks that have been referenced. Four blocks in the active zone have not yet been referenced (13,14,17,19).

7.2.3 Bounding the Future Zone

There are several properties about the active zone, and changes to the active zone, that are useful for detecting and predicting sequentiality. At any time, the active zone is defined by the current values of *minlast* and *maxlast*. We derive a bound on the extension of the active zone when a process references a block beyond *maxlast*. This bound aids in detecting the end of a sequential portion, by specifying the blocks that are likely to be part of the current portion.

Theorem 1 *For a pattern with p processes and a fixed record size of r blocks (r not necessarily integral), the next block to be referenced (assuming it is in the same portion) is between* minlast *and* maxlast+$\lceil pr \rceil$, *inclusive.*

Proof: From our observation that the references of each individual process are ordered, the next reference for each process is greater than or equal to the *last* for that process. Since *minlast* is

the minimum *last* value, the next reference from any process is certainly greater than or equal to *minlast*.

We assume that records are assigned to processes in increasing order, in either a self-scheduled or round-robin pattern. Since no process is assigned more work until it has finished its current assignment, there are at most p records assigned that have not yet been requested from the file system, involving at most $\lceil pr \rceil + 1$ blocks. The $+1$ only arises in certain situations when the first record begins in the middle of a block. If all of these outstanding blocks were greater than *maxlast*, then the highest outstanding block number would be *maxlast*$+\lceil pr \rceil$. We would calculate it as *maxlast*$+\lceil pr \rceil + 1$, but the $+1$ does not belong here, since if the first outstanding record does begin in the middle of a block, it must begin in the middle of *maxlast*. Since the next reference must be one of these outstanding block numbers, the next reference is also limited to *maxlast*$+\lceil pr \rceil$. The theorem follows. □

This theorem is easily generalized to variable record sizes: the theorem still holds if r is the maximum record size. The upper bound in the theorem can also be tightened: since each process reads the blocks of each record in order, only the first block of the highest outstanding record is a possible next reference. Thus, the maximum next reference is *maxlast*$+\lceil pr \rceil - \lceil r \rceil + 1$.

We now have a well-defined range of blocks for the next reference. The bound on the extension of the active zone is useful, but unfortunately there is no bound on the size of the active zone itself. This has strong implications for any implementation that tries to track the active zone. The size of the active zone is unbounded because it is possible for one process, whose *last*=*minlast*, to not reference any blocks for a long time while the other processes move ahead. Our implementation optimizes for small active zones, due to our assumption that no process will drop out or run relatively slowly.

7.3 The GAPS Predictor

Armed with this understanding of globally-sequential access patterns, we have developed an algorithm for recognizing, tracking, and using global sequentiality. We call this predictor GAPS, for Global Access Pattern Sequentiality. The method is much more complex than any of the local predictors, and also involves more guesswork. On the other hand, this prediction algorithm bases its decisions on more references than do the local predictors, so its decisions may sometimes be more accurate. We describe the technique first, and delay implementation details to Section 7.4.

7.3.1 The Overall Plan

Like other predictors, the GAPS predictor is a self-contained module within the file system, notified on each block reference and queried when prefetching work is desired (page 23). The predictor is then a "black box" that takes block reference streams as input, responds to queries for prefetching predictions, and issues flush commands for mistaken prefetches. As with the rest of the file system, the GAPS predictor is concurrent: every user process may be active simultaneously. GAPS maintains separate state information for each process, as well as global state information.

The structure of the GAPS predictor is a state machine with three states (Figure 7.2). The only events that trigger state transitions are the notifications from the file system. Although it is possible for processes to be in different states, they usually move between the states in unison (these are distinguished in the figure). This is accomplished with a mixture of private and shared state variables. Initially, all processes are in *Watch* mode, watching the access pattern for signs of sequentiality or extreme randomness. If the pattern appears extremely random, all processes shift to the much simpler *Random* mode, in which they do just enough processing to shift back

to Watch mode if the pattern becomes less random. If sequentiality is detected, all processes shift to *Continuation* mode, where prefetching is finally possible. Continuation mode tracks the sequentiality carefully; if sequentiality appears to end, the processes drop out, one by one, back to Watch mode. The separate transitions here allow each process to complete the portion before going to Watch mode. All processes must be back in Watch mode before any new check for sequentiality, and thus before any process re-enters Continuation mode. In the next few subsections, we describe each state separately.

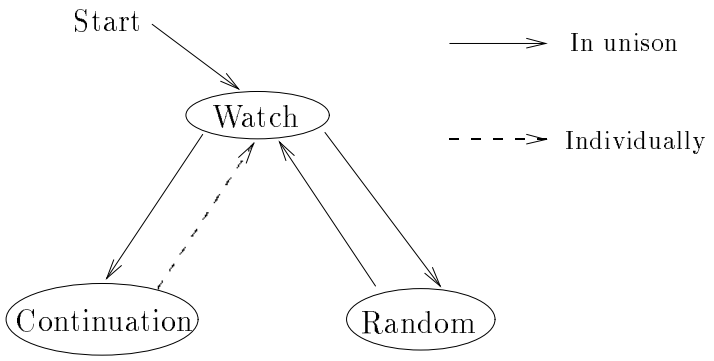


Figure 7.2: The three states in GAPS. Most state transitions occur simultaneously across all processes.

7.3.2 Watch Mode

Watch mode has two primary purposes: to detect sequentiality, and to record enough information about the sequential portion to start Continuation mode and enable prefetching. No predictions are made while in Watch mode.

The access pattern is presented to GAPS via a notification procedure called on each reference. All that is provided about each reference is the block number. To detect sequentiality GAPS records the access pattern in a shared *access list*. The blocks in the list are tagged with process numbers. The *last* block number is recorded by each process, for comparison with each new reference. If the new block number is the same as or greater than the last block number, then the ordered property has been maintained, and the new reference is appended to the list. If the new block number is less than *last*, a jump-back has occurred. Under our assumptions, that process must have moved to a new sequential portion. Thus, all of its previous references are irrelevant to the new portion, and are flushed from the list (using the process tag on each list element). If the jump-back is indicative of a new portion, then all processes eventually jump back and flush their entries from the list.[1] The list then contains only blocks from this new portion, in their global reference order.

To be considered sequential, the access pattern must pass three successive tests, described below. If sequential, several values are computed before switching from Watch mode to Continuation mode: *minlast*, *maxlast*, *start* (the starting block number of the portion), and *maxjump* (the furthest we expect to extend *maxlast* in any one reference). Continuation mode also requires a list of blocks that have already been referenced in the active zone (copied from the access list), and *last* for each

[1] Note that if we were to flush the whole list on every jump-back, the earliest blocks in the portion would be erased from the list by the last process to join the portion.

process.

The first test, that of enough information, ensures that all processes are in Watch mode and that each has contributed to the access list. The second and third tests are described below.

Completeness

The Completeness check examines the access list as a set of blocks without regard to their time ordering. First, we compute *minlast* from all the processes' *last* values. Then we examine the blocks in the list to determine the completed part of the old zone. In other words, what is the value of *start* such that all blocks between *start* and *minlast*, inclusive, have been referenced? Note that blocks greater than *minlast* or less than *start* are ignored. Because there are no jump-backs in the access list, blocks less than *start* are from an earlier portion, no longer relevant.

Slope

If the pattern passes the Completeness check, we test it for linearity. In this check, the order of the blocks in the access list is important. We first restrict the access list to blocks greater than *start*. We then treat the blocks in the list as a function of their position in the list, and fit a line to this function, obtaining a slope. If the slope is (for some reason) negative, the check fails immediately. A positive slope, however, does not tell us whether the function was even close to linear. Thus, we compute the coefficient of determination (c_d) for the function [Tri82]. The coefficient is a measure of the deviation of the function from the linear fit, and is between zero and one. If c_d is close to one, the fit is good, our slope check succeeds, and we consider the access pattern to be sequential. A piece of an actual access pattern is shown in Figure 7.3, along with the fitted line, the slope, and c_d. Here the fit is good.

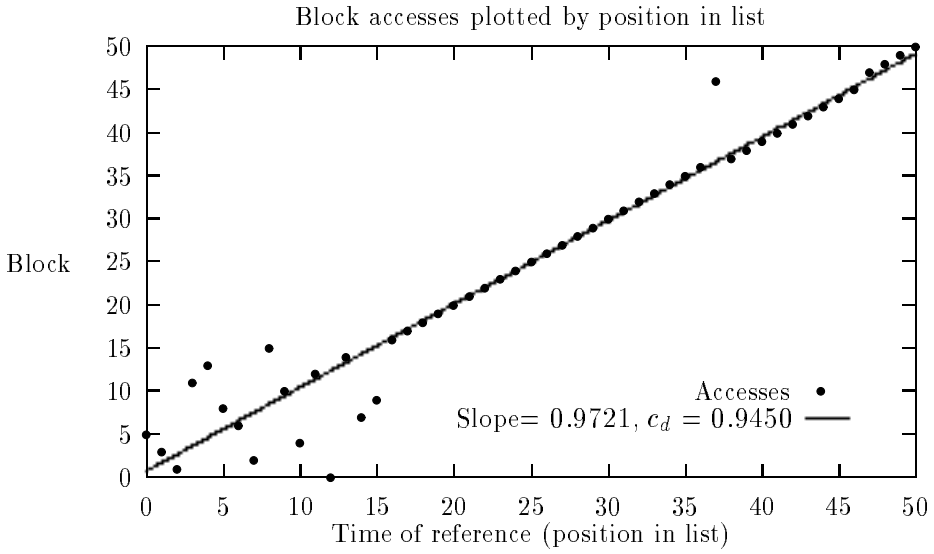


Figure 7.3: An actual access list plotted with respect to our measure of time, the position in the list. The line resulting from a linear regression of the data is also plotted.

Occasionally there are two or more sequential portions represented in the access list, with the later one involving higher block numbers. Thus, there were no jump-backs to flush the old portion from the list. If *minlast* and *start* are still in the lower, older portion, the slope calculation encompasses both portions, obtaining a large slope. Usually c_d is low, and the check fails. As an optimization, the check fails due to a large slope before computing c_d. This is safe, since the slope is rarely much more than 1.0 for the sequential reference patterns that fit our initial assumptions.

An Optimization: Random Mode

Watch mode is sufficient to distinguish sequential access patterns from random access patterns. However, it also entails a lot of overhead. For random access patterns, this overhead is costly, because there are no prefetching benefits to balance the overhead. For this reason, we add *Random* mode.

In a truly random access pattern, the ordered property of references on a particular process is rarely maintained for many references. The probability of three or more references in nondecreasing order is roughly[2] $\frac{1}{6}$. For p processes to simultaneously be ordered, the probability is $(\frac{1}{6})^p$, which is small even for moderate p. In a sequential reference pattern, however, references are usually ordered. Thus, we shift to Random mode when several processes are not maintaining reference order, and shift back to Watch mode when all processes again have ordered references. In Random mode, there is only a quick check for changes in ordered status. This reduces overhead and increases concurrency.

7.3.3 Continuation Mode

Continuation mode has several purposes. Primarily, it tracks the sequential portion recognized by Watch mode. It detects breaks in sequentiality, and decides how to handle them, perhaps by going back to Watch mode. It tracks the portion length and portion skip, in an effort to detect regular sequential portions. Finally, of course, it records enough information to make predictions and to catch mistakes.

As each reference arrives, Continuation mode updates *last*, *minlast*, and *maxlast*. It checks for failures in three critical areas: orderedness (jump-backs), completeness (in the old zone, as it grows), and *maxlast* extension. Any of these failures signal the end of a sequential portion, and can force the process to leave Continuation mode. Despite these extensive efforts, Continuation mode is more efficient than Watch mode, and also allows prefetching. Thus, Continuation mode tries to handle sequentiality failures and stay in Continuation mode.

A jump-back clearly indicates that the process is in a new portion. Since the new references could overlap the current portion, and the two portions become confused, the process must leave Continuation mode. The other processes follow when they have also finished the current portion.

A completeness failure is detected only when *minlast* is updated. When *minlast* changes, the old zone is extended. We expect the old zone to be complete, that is, for all blocks in the old zone to have been referenced. Any gaps signal the end of one portion and the start of the next. A gap may or may not have been detected previously by the *maxlast*-extension check.

The third check involves references that exceed *maxlast*. From Theorem 1 we know that a reference in the current portion cannot be past *maxlast*+$\lceil pr \rceil$, for p processes and record size r. (Our implementation does not use the tighter bound from Section 7.2.3.) The quantity $\lceil pr \rceil$ is called *maxjump*. Thus, any reference that is past *maxlast*+*maxjump* is likely to be in another

[2]The probability of being ordered can be easily found with a combinatoric calculation. For an N-block file, the exact probability is $\frac{1}{6} + \frac{1}{2N} + \frac{1}{3N^2}$.

portion. If this happens, one option is to leave Continuation mode and watch for sequentiality to begin again. There are, however, two optimizations: if the portion length and portion skip are regular, and the current reference is consistent with those values, then GAPS assumes the pattern is continuing. If the portions are irregular, but the completed portion is long, GAPS treats the new reference as a new portion and remains in Continuation mode. This makes the assumption that the pattern consists of irregular sequential portions. These optimizations allow the GAPS predictor to spend most of its time in Continuation mode, and to do more prefetching.

7.3.4 Prefetching

The bulk of the GAPS predictor is involved with watching for and tracking sequentiality. All of the information is used to predict future accesses and to recommend blocks for prefetching. Essentially, GAPS recommends blocks for prefetching that have not yet been accessed, are greater than *minlast*, and meet other special-case constraints. It prefetches in both the active and future zones.

The prediction code is similar to that in the IPORT local predictor. That is, it detects regular sequential portions, and prefetches accordingly. Unlike IPORT, it limits the prefetch distance only when the portion length is not regular. In the first portion, or with regular portions, it may prefetch far past *maxlast*. When the portion length is not regular, it limits its prefetching at all times to *maxlast*+MaxDist, where MaxDist is a parameter as before (see Section 7.4.3). It also limits its prefetching to the active zone whenever any process leaves Continuation mode, which implies the current portion is ending.

7.4 Implementation of the GAPS Predictor

In order to make a scalable predictor, concurrency must be high and the amount of serialization kept to a minimum. This is one problem with GAPS: it uses a great deal of serialization to make global decisions, especially in Watch mode. Thus, the scalability of this predictor may be limited (see the scalability experiments in Section 8.5). Thus, it represents one endpoint of the tradeoff spectrum between accuracy and efficiency.

Without going into detail, we briefly describe the implementation of each of the primary modes, along with the transitions between them.

7.4.1 Watch Mode

Watch mode is completely serial, operating as a (rather long) critical section. More concurrency may be available in Watch mode, but to gain concurrency we would have to sacrifice some accuracy. In GAPS we concentrate on accuracy. The entrance to Watch mode is controlled by a FIFO queue, so the processes enter the critical section in the same order that they arrived. This is crucial to maintaining the original block ordering, which is important to the slope check. The access list is represented by a shared array.

7.4.2 Random Mode

The Random mode has a simple implementation. The access list is not used. Each process in Watch mode counts the total number of references it has made since its last jump-back. If this is higher than the threshold (two references) then this process is considered *ordered*. There is a global count of the number of processes that are ordered. When this count falls below a threshold (70% of processes), GAPS enters Random mode. When it returns to the number of processes (all processes are ordered), GAPS returns from Random mode to Watch mode.

7.4.3 Determining *maxjump* and MaxDist

Before GAPS may enter Continuation mode it must determine two values: *maxjump*, used for tracking sequential portions, and MaxDist, used to limit prefetching.

The ideal value of *maxjump* is $\lceil pr \rceil$, where p is the number of processes and r is the (maximum) record size in blocks. The file system does not have access to r, so it must be estimated. The record size is estimated with heuristics based on observations of the average number of references to each block, and the average length of locally-consecutive runs (a series of consecutively numbered references from a single process).

The ideal value of MaxDist is more difficult to determine. A large MaxDist leads to more mistakes, and a small MaxDist may not allow enough prefetching. Section 7.6.5 examines this issue further. Preliminary experiments led us to determine MaxDist from the record size as follows: for $r < 1$, use MaxDist= p. For $r = 1$, use MaxDist= $3p/4$. For $r = 2$, use MaxDist=$p/4$. For $r > 2$, use MaxDist= 0.

7.4.4 Continuation Mode

The primary data structure for Continuation mode is an array with one entry for each block in the file. Our implementation uses the file system's existing, memory-resident block map for the file instead of a separate array.[3] Each entry in the array contains a *used* bit, indicating that GAPS knows that the block has been referenced. These bits are initialized from the access list, and updated by later block notifications. Once Continuation mode is started, block notification can proceed concurrently with other notifications and with prefetching. On each notification, the *used* bits are adjusted, and *minlast* and *maxlast* are updated, possibly catching mistakes or deciding that the portion has ended. When a process leaves Continuation mode, it limits prefetching to the current active zone, and drops out into Watch mode. The last process to leave must clean up the data structures.

7.4.5 Prefetching

The GAPS prediction code is executed on request from the file system, and can run concurrently with other processes in Continuation notification code. There is a short lock used to choose a block for prefetch. The block is chosen by updating a single counter, then adjusting for any known portion skips and the prefetch limit. Also, the block must not have already been used or prefetched. The block is marked for later detection of prefetch mistakes, and the block number is given to the file system.

7.5 Other Global Predictors

For comparison with the accurate but inefficient GAPS predictor, we consider two predictors that are less accurate but more efficient. These are two more points in the tradeoff spectrum between accuracy and efficiency.

The GAPS predictor has three primary states: Watch mode, Random mode, and Continuation mode. Watch mode is expensive, due to the serialization and the amount of computation. Watch mode determines whether the access pattern is random or sequential, and enters Random or Continuation mode, respectively. If all patterns are either random or sequential, then why not eliminate Watch mode entirely? This is the basis of the RGAPS predictor, which begins in Random

[3] Other data structures are possible, of course.

mode. In RGAPS, when Random mode decides that the pattern is not random, it shifts directly to Continuation mode, constructing the data structures as well as possible. Of course, the information usually found in the access list is not available, so the start of the portion and the initial list of used blocks are inaccurate, and estimates of *maxjump* and MaxDist are crude. Continuation mode exits directly to Random mode when necessary. Some wrong decisions may be made, but at least they are made quickly. Of course, not all patterns are either random or globally sequential, so this predictor may be prone to failure on pathological patterns.[4] In any case, it is an interesting alternative.

Another possible alternative is to build a predictor specific to a particular pattern. GAPS is intended to handle the **gw**, **grp**, **gfp**, and **rnd** (random) patterns. It is likely, however, that some workloads may contain only, or predominately, **gw** patterns (at least among the global patterns). Thus, we have implemented a GW predictor, designed specifically for **gw**, that prefetches from the start to the end of the file and ignores mistakes. It adjusts its prefetching in only two ways: to accept block numbers back when they could not be prefetched, and then recommend them again; and to keep the prefetching ahead of *maxlast*. This predictor is especially valuable if the file system allows hints from the user to direct its prefetching (Section 10.3).

7.6 Experiments and Results

We ran a broad suite of experiments using the global predictors. Each of the patterns **grp**, **gfp**, **gw**, and **rnd** was used with an appropriate set of predictors. NONE, GAPS, and RGAPS were used with all patterns. EXACT was used with all patterns except **rnd**, since the best predictor for **rnd** is NONE. Finally, the GW predictor was used with the **gw** pattern. Each of the four synchronization styles *each(10)*, *total(200)*, *none*, and *neighbor(10)* was used with all patterns. Finally, each of these tests was run both with and without computation on each block. Each combination of these parameters represented one test case.

We ran all predictors on the **lw** pattern, since **lw** is a type of **gw** pattern where every process reads every block. We also included the IBL predictor (which had had the best results for **lw**) for comparison. Due to the short execution times of the normal **lw** pattern, we used an extended **lw** here, in which every process read the same 1000 blocks (five times as many as before). We used only the *each(10)* synchronization style, to avoid the **lw**-phase problem. Each case was run with and without computation on each block.

For each test case, we averaged the total execution time over five trials, and used this as our comparison measure. The standard deviation over five trials was always less than 11% of the mean (at most 1 second), and was less than 2% in 90% of the cases. Small differences (in most cases, about 0.2 seconds or less) between the average times for two predictors should therefore be considered insignificant relative to measurement error. We present the results in the same style as we did the local pattern results. The time in seconds and predictor name is given next to each line in the graph, and the maximum *cv* for all lines in a graph is given in the graph's caption.

The other parameters were the same as in our previous experiments. There were 20 processes and 20 disks. Each pattern (except **lw**) involved reading 4000 blocks. There were 80 buffers in the cache, with up to 60 allowed for prefetched blocks. The block size and the record size were both 1 KByte. The computation simulated for each block, when used, averaged 30 msec.

[4] This includes local patterns. Section 7.7 describes a special predictor for distinguishing local and global patterns, which chooses either a local or a global predictor as appropriate.

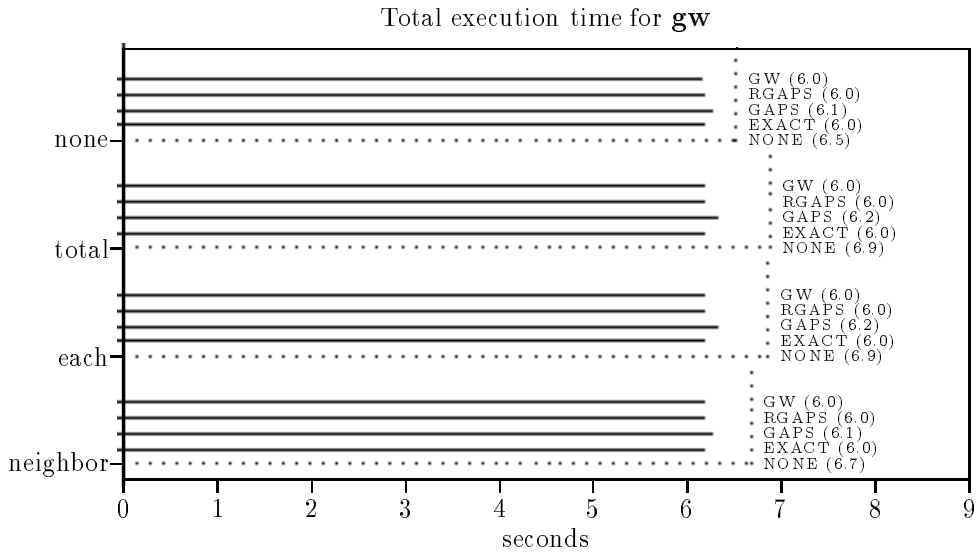


Figure 7.4: Execution times for **gw**. ($cv < 0.012$)

The results for **gw** (Figure 7.4) were encouraging and fairly straightforward. It was no surprise that the GW predictor was best among the on-line predictors, always matching EXACT. GAPS and RGAPS nearly matched those two predictors, and were also essentially equivalent to each other. This is because the expensive Watch mode (the difference between GAPS and RGAPS) was used little here. The results for **gw** with computation (not shown) were similar, except that the improvements were more significant (about 40% faster with prefetching than without) and that RGAPS was more similar to GAPS. With computation, the overhead of GAPS was reduced due to decreased contention in Watch mode, since the added computation meant there were fewer processes active in the file system code at any one time.

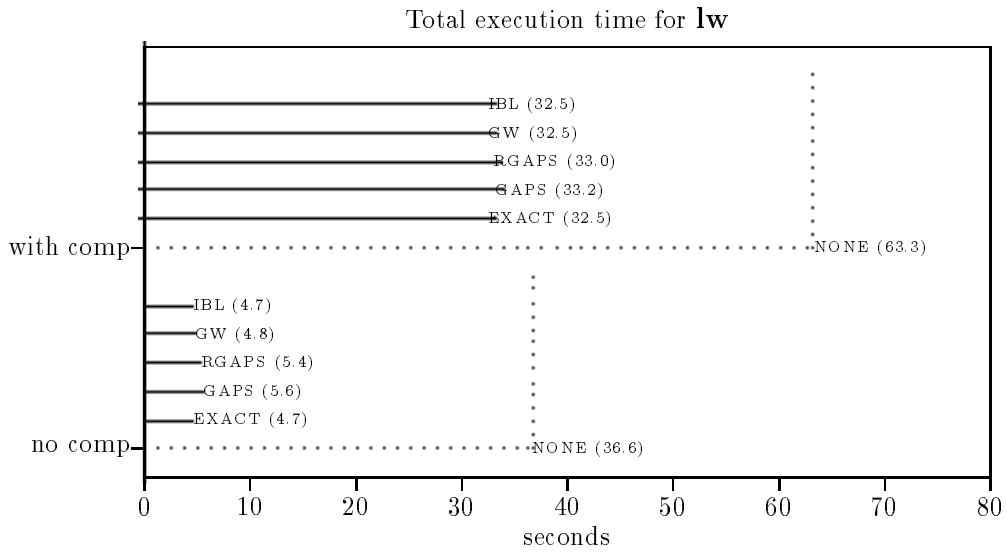


Figure 7.5: Execution times for **lw**, both with and without computation. Both cases use the *each(10)* synchronization style. Note that this is an extended **lw**, reading 1000 blocks. ($cv < 0.015$)

In the experiments with the **lw** pattern, shown in Figure 7.5, we used only the *each(10)* synchronization, so there is room for both the with-computation and no-computation cases on the same graph. The results here were similar to those for the **gw** pattern, with GW matching EXACT, and GAPS and RGAPS coming fairly close. GAPS and RGAPS had more overhead to slow them down than did GW or EXACT. With longer patterns, the difference between GAPS and EXACT (which was also due to start-up effects) should be less significant. For comparison, we included the IBL predictor here. The EXACT, GW, and IBL predictors had equivalent performance.

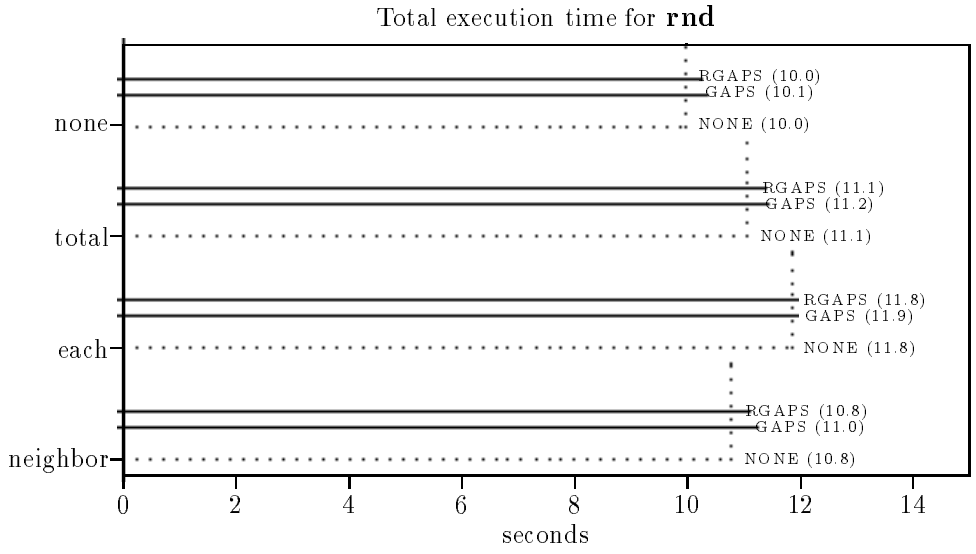


Figure 7.6: Execution times for **rnd**. ($cv < 0.0074$)

Results for the **rnd** pattern are shown in Figure 7.6. No prefetching was possible because of the random access pattern, so NONE represented the best possible time. That GAPS and RGAPS were as fast as NONE shows that they recognized and handled **rnd** patterns with low overhead. This success was due entirely to the "Random" mode of GAPS and RGAPS (an early version of GAPS without Random mode was almost four times slower than NONE!). The conclusions for **rnd** with computation are identical (not shown).

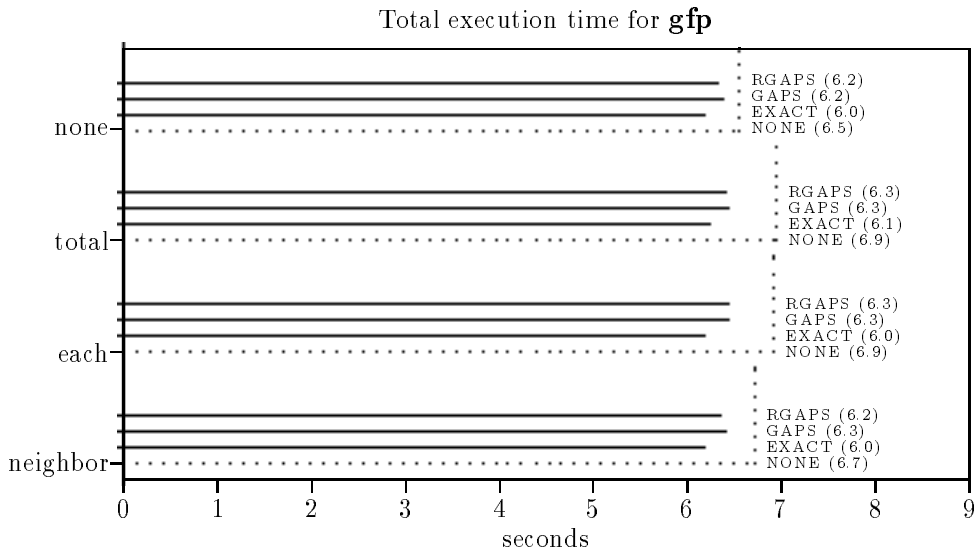


Figure 7.7: Execution times for **gfp**. ($cv < 0.011$)

The **gfp** pattern (Figure 7.7) is more difficult than the preceding patterns, due to the portion changes. The regularity of the portions, however, was recognized by both GAPS and RGAPS, and used to prefetch over the portion skips into future portions. The RGAPS predictor required one more portion to recognize the regularity, since it did not correctly notice the start of the first portion (Random mode is less accurate than Watch mode). Nonetheless, RGAPS and GAPS were essentially equivalent on this pattern. The results for **gfp** with computation (Figure 7.8) show that GAPS and RGAPS are closer to EXACT than to NONE, something that is not evident in Figure 7.7.

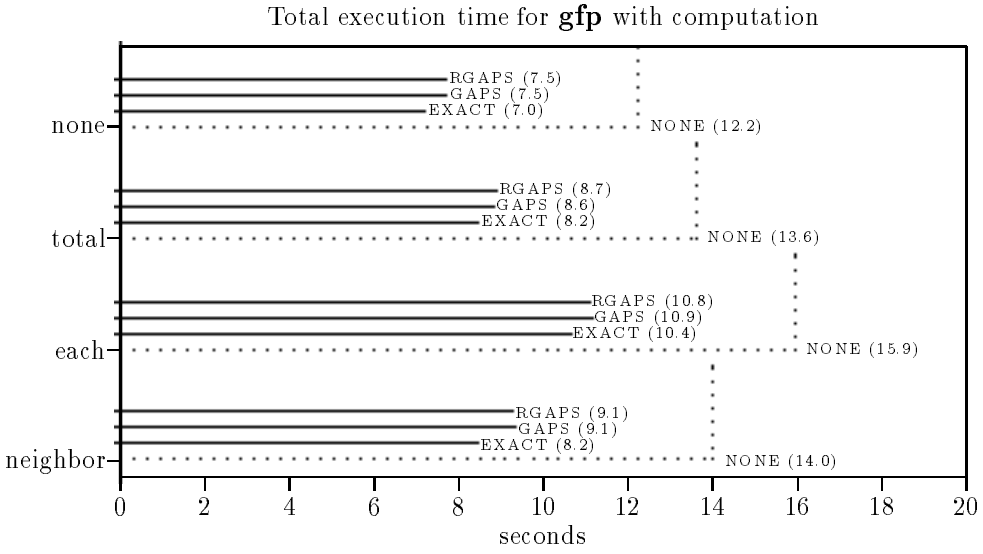


Figure 7.8: Execution times for **gfp** with computation. ($cv < 0.072$)

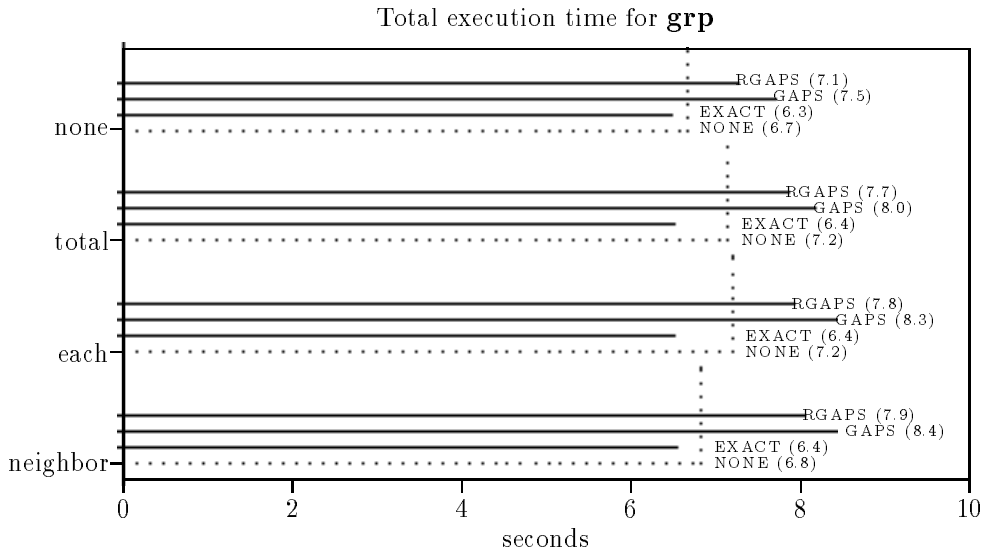


Figure 7.9: Execution times for **grp**. ($cv < 0.024$)

The **grp** pattern (Figure 7.9) is the most difficult pattern for the GAPS predictor. Because of the unpredictable portion lengths and jump-backs, GAPS spent a lot of time in Watch mode, and made more prefetching mistakes than in other patterns. In our experiments, GAPS was never able to break even and match NONE in the no-computation test cases (but see Section 7.6.6). Note that RGAPS was slightly faster than GAPS in all cases (we examine this further in Section 7.6.2). We also ran this experiment with a different **grp** pattern constructed with the same parameters. The results (not shown) from this experiment were similar to those for the original **grp** pattern, except that RGAPS was more clearly faster than GAPS. Thus, although GAPS was more accurate, the efficiency of RGAPS was more important in **grp**.

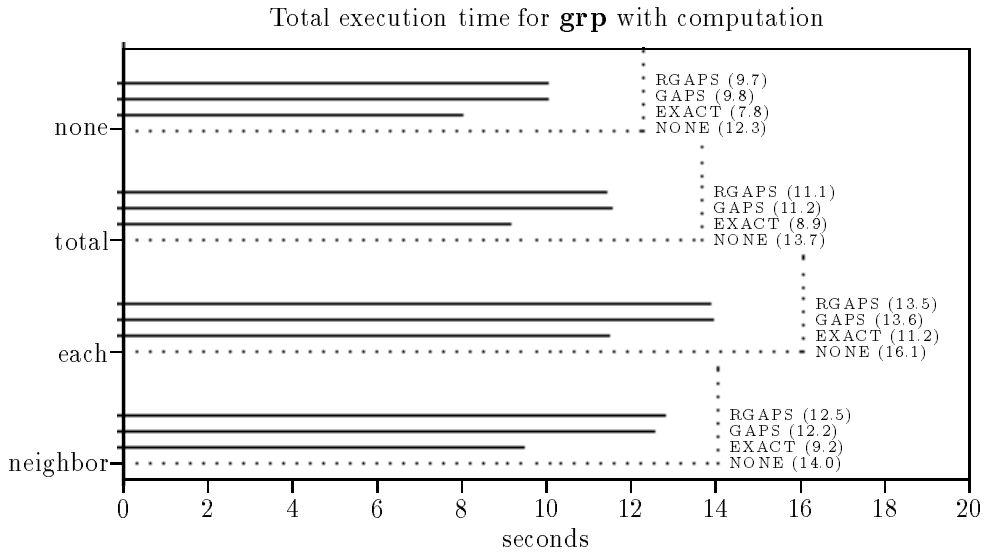


Figure 7.10: Execution times for **grp** with computation. ($cv < 0.054$)

When there was some computation on each block, the opportunities for overlapping I/O and computation were increased, and the effects of I/O overhead were less significant. In the **grp** pattern with computation (Figure 7.10), as in all test cases with computation, the benefits of prefetching were more significant. Indeed, GAPS and RGAPS were able to obtain significant benefits by prefetching, about half those obtained by EXACT. There was also less of a difference between GAPS and RGAPS (same reason as for **gw**, page 87). As before, we also tried a different **grp** pattern generated with the same parameters. Again, there was little qualitative difference between the results for that pattern (not shown) and those for the first **grp** pattern, except that RGAPS was a little faster than GAPS.

7.6.1 Performance of the Global Predictors

With the usual caveat that our workload does not necessarily contain the distribution of patterns in any real workload, we formed a summary presentation of the performance of GAPS and RGAPS. Since EXACT is a good baseline for evaluating global predictors, we evaluate GAPS and RGAPS in terms of their relative performance to EXACT. Our measure was 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 was the execution time for EXACT, t_n was the time for NONE, and t was the time for some other predictor, the normalized performance of this other predictor was

$$\frac{t - t_n}{t_e - t_n}$$

Note that the normalized performance was 1 when the predictor in question did as well as EXACT. It was zero when it did only as well as NONE, and negative when slower than NONE. We computed the normalized performance for all test cases except those of the **rnd** pattern, since there $t_e = t_n$. We plot the CDFs of the distributions of these values in Figure 7.11. The low-performance (negative) cases were all from the **grp** pattern, where GAPS and RGAPS were often 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.

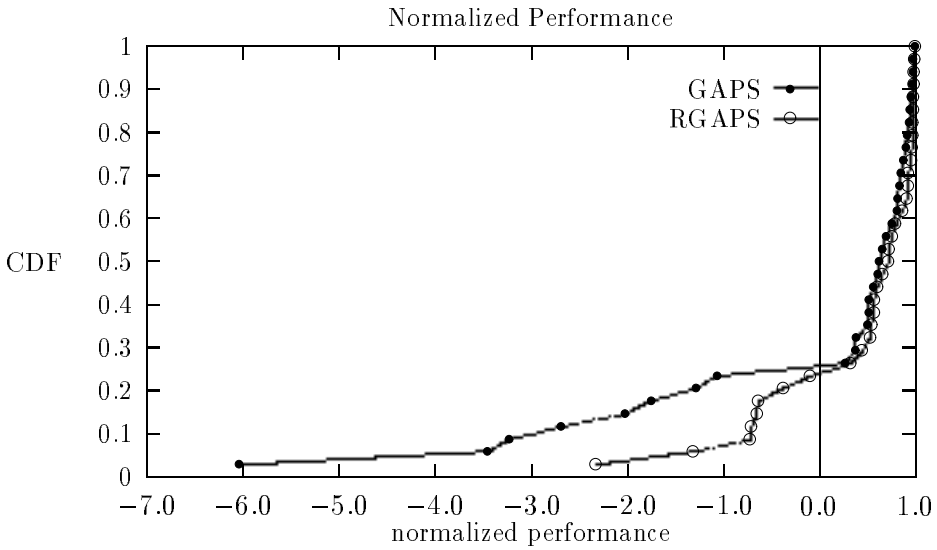


Figure 7.11: The normalized performance for GAPS and RGAPS on all patterns except **rnd**. A normalized performance of 1.0 indicates that the predictor matched EXACT, and a negative number indicates that it was slower than NONE.

7.6.2 GAPS *vs.* RGAPS

From the preceding discussion of the results, it appears that there was no performance difference between the GAPS and RGAPS predictors for most patterns. The exception was **grp**, where

RGAPS was faster in most cases. This makes sense, since Watch mode (the difference between GAPS and RGAPS) is not important to the other patterns. In Figure 7.11, the predictors had similar distributions in the high-performance areas, but differed in the low-performance areas (i.e., the **grp** pattern). Thus GAPS and RGAPS were essentially identical for most test cases here, with RGAPS better for **grp** patterns. RGAPS was much more robust (and usually had better performance) in the experiments described in Chapter 8 (in particular see Section 8.1). Section 7.6.6 points out a situation where GAPS was more robust than RGAPS.

7.6.3 Accuracy

Judging by the performance of GAPS and RGAPS on most patterns, the effort expended to prefetch was worthwhile. One reason for this was the accuracy of the predictions. Figure 7.12 shows the distribution of the fraction of prefetched blocks that were wasted (read from disk but never used). The **rnd** pattern is excluded here, since there were no prefetches. There were many zero-waste cases from the **gw** and **lw** patterns. Only the **grp** and **gfp** patterns allow for mistakes, and even in these cases it is clear that the waste rate was extremely low, always less than 13%. Note also that RGAPS tended to have a higher waste rate, as expected.

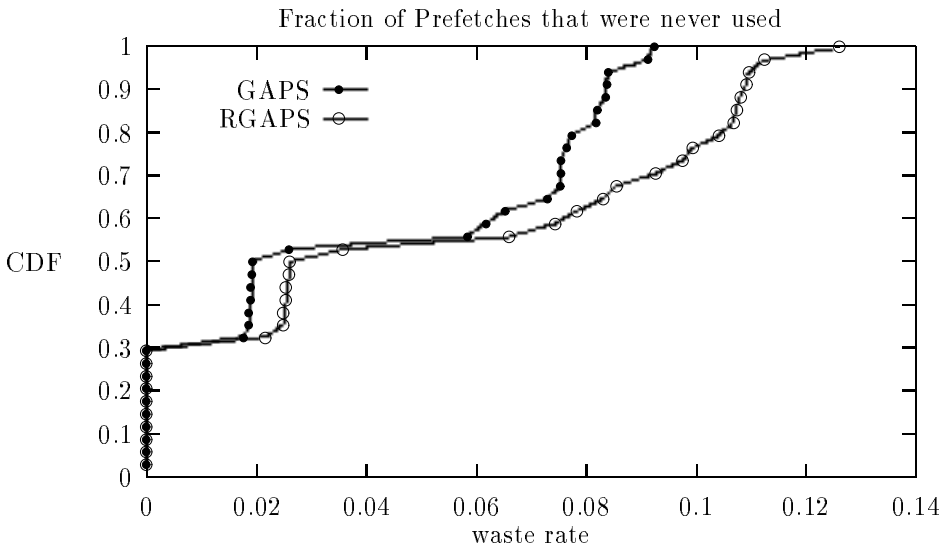


Figure 7.12: The waste rate for GAPS and RGAPS. The waste rate is the ratio of the number of wasted blocks to the number of prefetched blocks.

7.6.4 Overhead

The overhead of the GAPS predictor can be high. One measure of the overhead is the notification time, the time spent in the GAPS predictor for every block read (page 23). From an examination of the distribution of notification times, most notifications were short (less than 5 msec). Some notifications were extremely long (100–500 msec), with the longest notification times typically in the **grp** pattern, an evidence of its difficulty.

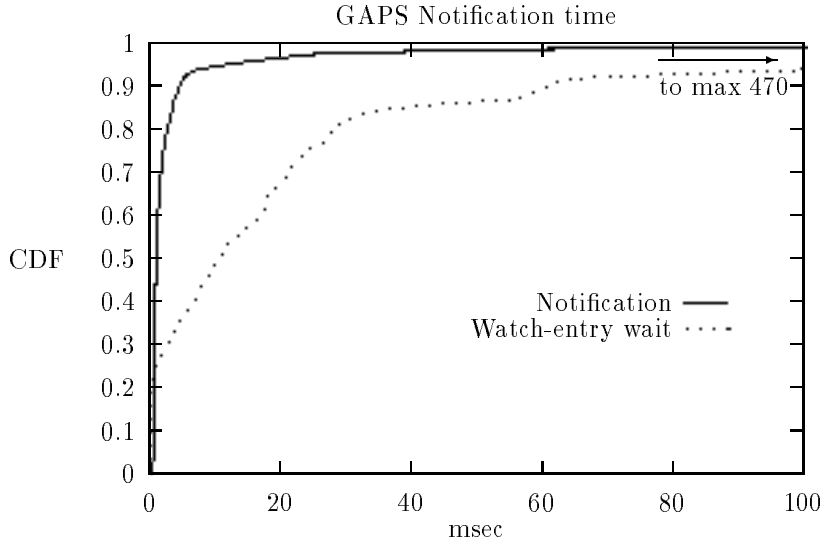


Figure 7.13: Distribution the GAPS notification time for a typical **grp** test case.

In Figure 7.13, we display the distribution of the notification time for a typical **grp** case, along with the distribution of the time waiting to enter Watch mode's critical section. Here we see the severe skew on the distributions: 90% of the notification times were less than 5 msec. There was often a long wait to enter Watch mode, due to the critical section. Thus many of the long notifications were when GAPS was in Watch mode, and many processes were forced to wait at the entry point. This kind of serialization is likely to limit the scalability of such an algorithm. The longest notification times, however, occurred in Continuation mode, and reached up to 470 msec. These were the result of a major update to *minlast*, which often had to scan large portions of the (memory-resident) file map when moving from one portion to another. Fortunately, these long updates were rare, and were concurrent with other Continuation-mode notifications.

The RGAPS predictor uses Random mode instead of Watch mode. This mode is simple and concurrent, compared to the complex, serialized Watch mode in GAPS. The notification times for RGAPS were generally lower than for GAPS (not shown), except in the **grp** pattern, where they were similar. Figure 7.14 shows the distribution of the notification times and Random-mode times for the same pattern as in Figure 7.13, but with the RGAPS predictor. Note the time scale is shorter than Figure 7.13. The Random-mode time was short, never more than 3 msec and less than 0.75 msec in 97% of the cases. As with GAPS, RGAPS had some long Continuation-mode times, up to 430 msec. This was reflected in the notification-time distribution. Nonetheless, 50% of the notification times were less than 1 msec.

7.6.5 The Effect of MaxDist

In any predictor, the prefetch distance represents a commitment to prefetching. The prefetch distance is the number of blocks that the predictor is willing to prefetch past the highest-known block in the portion. There is a tradeoff in many situations, since a small distance overly restricts prefetching, and a large distance increases the number of mistakes when a portion ends unexpectedly.

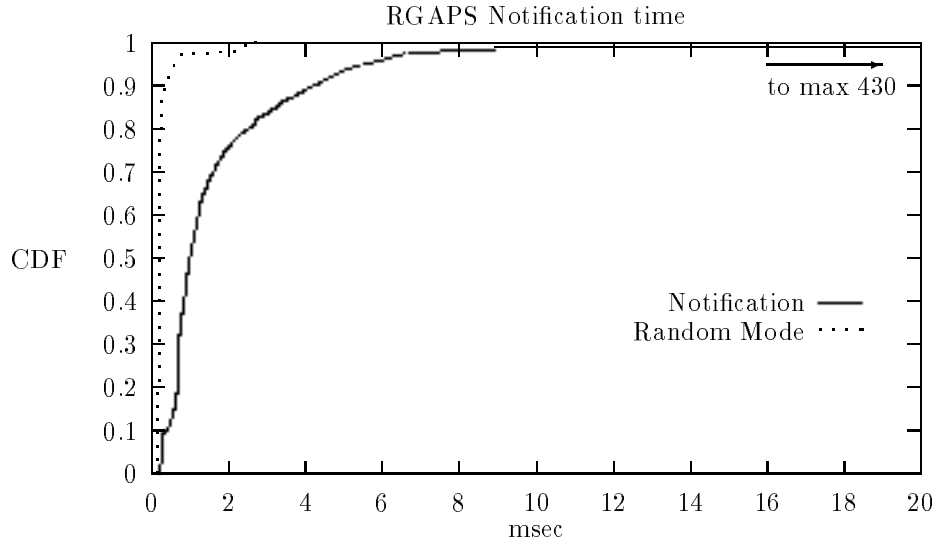


Figure 7.14: Distribution the RGAPS notification time for a typical **grp** test case. Note the time scale is shorter than Figure 7.13.

In the preceding experiments the prefetch distance (called MaxDist) was determined as described in Section 7.4.3 (which were derived from preliminary experiments). For these tests this means MaxDist was 15. In the first portion, and when the portions were regular, MaxDist was infinite, allowing plenty of prefetching for **gw** and **gfp** patterns.

To examine the effect of MaxDist on the GAPS predictor, we ran a set of experiments varying MaxDist from zero to 60 blocks. It was still infinite for the first portion and for regular portions, so it did not affect **gw** or **lw**, and scarcely affected **gfp** in preliminary tests. It did not affect **rnd**, of course, so we only studied the **grp** pattern. All other test parameters were the same. Figure 7.15 shows the results for tests with no computation, and Figure 7.16 shows the results for tests with computation. Each point represents the average of five trials.[5]

Whereas it is difficult to determine the single "best" value of MaxDist from these results, some general conclusions are possible. First, it is important that MaxDist not be too small, since the performance was clearly worse for small MaxDist (small being less than 10 blocks for no-computation tests, 20 for computation tests). In some cases, the performance curve shows a definite minimum; in others the performance tends to level off. Overall, the best performance was found with MaxDist between 10 and 35, with 20 as a reasonable compromise. Within this range the performance differs little. Our choice of 15 allowed reasonable, though not optimum, results. Overall, a moderate MaxDist was needed, corresponding to the moderate predictability of **grp**.

[5]In a few of *neighbor(10)* tests there were some trials that were discarded from the average due to extreme behavior. These cases are the result of the extreme synchronization stress on the pattern, which idles some processors in synchronization points while the others are still working. The execution sequence and the execution times are thus highly dependent on random run-time variations. The affected data points are marked.

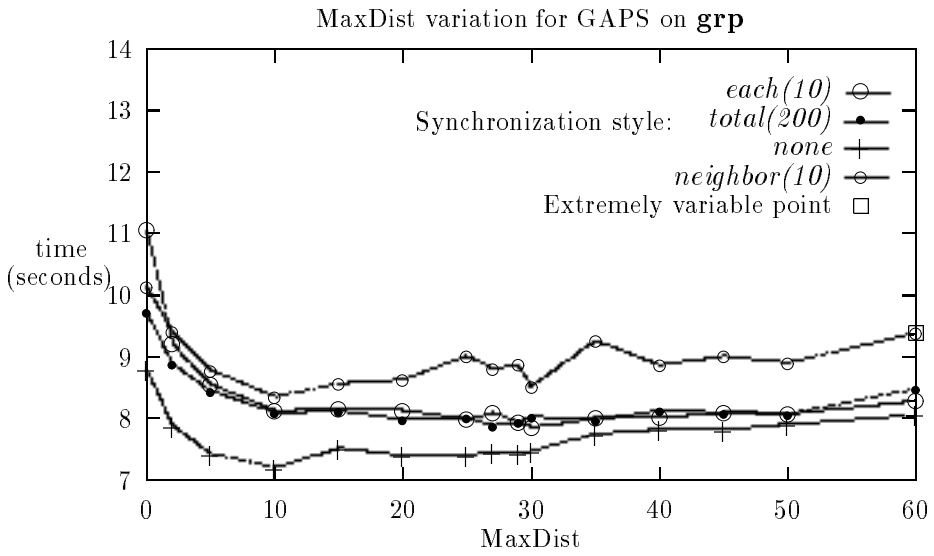


Figure 7.15: The effect of varying MaxDist on **grp** patterns with no computation.

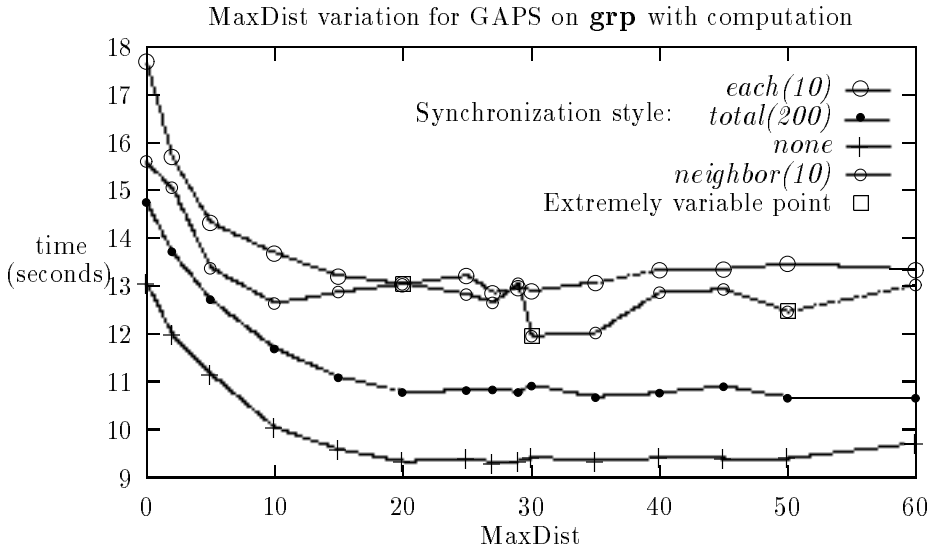


Figure 7.16: The effect of varying MaxDist on **grp** patterns with computation.

7.6.6 The Effect of Portion Length

The performance benefits from prefetching in sequential portions using the GAPS and RGAPS predictors sometimes depend on the length of the portion. Short portions are more difficult to recognize, particularly when the length is not constant. With longer portions more time is spent in Continuation mode, and less time is spent in handling portion skips, so overhead is reduced. To examine the effect of portion length on the global predictors, we ran a set of experiments using the GAPS and RGAPS predictors on several **grp** and **gfp** patterns, each with a different portion length. In the **gfp** patterns the portion length was fixed, of course; in the **grp** patterns, the portion length represents an average portion length. Our original experiments (and those in the previous section) used a 200-block portion length; here we vary the length from 100 to 1000 blocks. We ran each experiment with no computation on each block, and again with computation (keeping the set of computation times identical across all patterns). For simplicity, we restricted ourselves to the *none* synchronization style. For comparison, we also used the EXACT and NONE predictors. All other experimental parameters remained the same. Of course, when the portion length increases and the number of blocks in the pattern remains the same, the number of portions necessarily decreases.

The **gfp** pattern was generally insensitive to the portion length, due to prefetching over portion skips. Neither the NONE nor the EXACT predictors were affected by the portion length. The execution time for the GAPS predictor did not change appreciably with different portion lengths, at least for the no-computation test case. Since there were essentially no changes, the normalized performance (defined on page 94) for GAPS remained steady (Figure 7.17). The RGAPS predictor was only affected by the shortest (100-block) portions. Due to the incomplete information available in the RGAPS Continuation mode, RGAPS never recognized the pattern as **gfp** and could not take advantage of the regularity. This was true with and without computation, and is the reason for the odd RGAPS points in Figure 7.17. This is one (rare) case where GAPS was more robust than RGAPS.

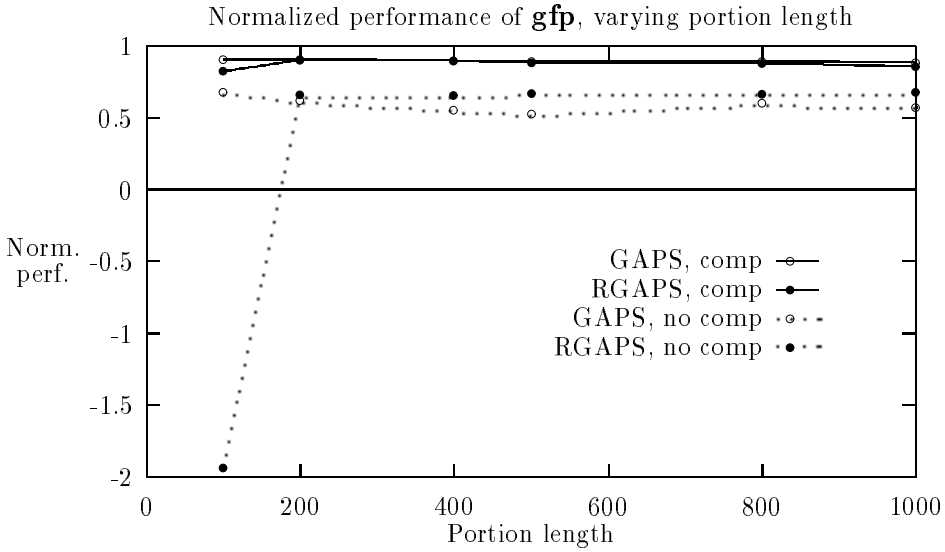


Figure 7.17: The normalized performance for the **gfp** pattern, while varying the portion length.

In the **grp** pattern, a decrease in the number of portions is as significant as an increase in the portion length. Thus, we expected GAPS, RGAPS, and EXACT all to improve their execution time on **grp**, and our experiments confirmed this. To gauge the significance of the faster execution times, we plot the normalized performance for GAPS and RGAPS in Figure 7.18. The normalized performance increased roughly with portion length. Thus, not only did the execution time improve, but the predictors were closer to EXACT's performance. The difference between GAPS and RGAPS also decreased, due to the smaller significance of portion skips.

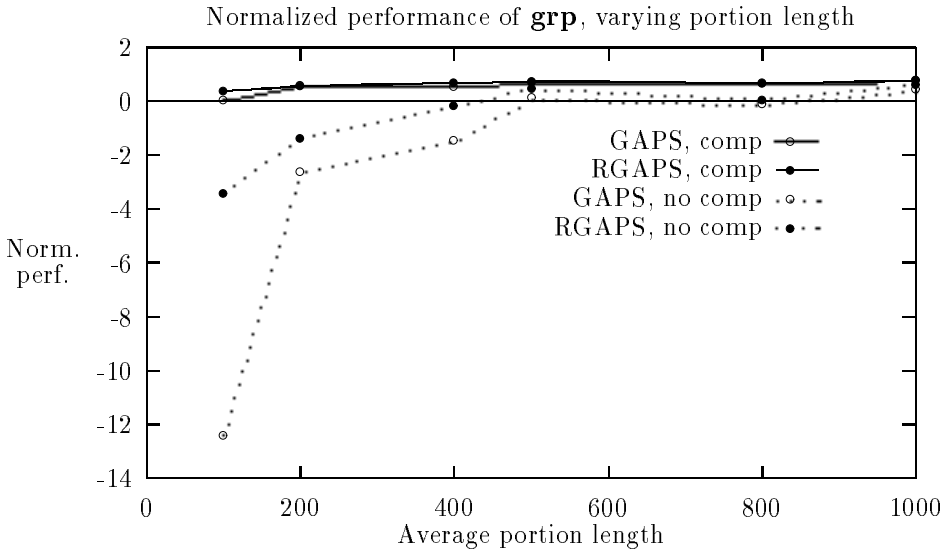


Figure 7.18: The normalized performance for the **grp** pattern, while varying the average portion length.

MaxDist and Portion Length

To examine any interactions between the MaxDist parameter and the portion length, we repeated the above portion-length variation experiment with several different MaxDist values from 0 to 60. We only summarize the data here:

- For **grp**, the optimum MaxDist value depended slightly on portion length, but otherwise the effects of MaxDist and portion length were independent.
- For **gfp**, the effects of MaxDist and portion length were only coincidentally correlated. Large MaxDist values were the solution.
- MaxDist should be higher for patterns with computation, to keep the disks busy during periods of computation.

7.7 Using Both Global and Local Predictors

We have a range of predictors for both local and global patterns. We have a predictor that is sufficient for most general-purpose local-pattern workloads (IOPORT). We also have predictors

sufficient for general-purpose global-pattern workloads (GAPS or RGAPS). However, GAPS is definitely not suitable for local patterns, and IOPORT is not suitable for global patterns. Is it possible to implement both in a file system, and use the appropriate predictor for each pattern? There are several possibilities:

1. Include two predictors (one local and one global); notify both on each access, and somehow arbitrate between their decisions by choosing one or the other. This requires too much overhead.

2. Include two predictors (one local and one global), plus a fast recognition mechanism to choose one or the other during the initial accesses of the pattern. This seems most promising, provided the recognition is fast and accurate.

3. Include two predictors (one local and one global); start working with one, but switch to the other if it does not work well. The problem is determining when it does not work well. This could use a mechanism similar to that in option 2 (above), or some other mechanism.

4. Merge a local and a global predictor into a single, more general predictor. Such a general predictor would unnecessarily complicate the simpler local pattern prediction.

We chose the second option and implemented a predictor called SWITCH. Once this predictor recognizes a pattern as either local or global, it switches control to a local or global predictor, respectively. SWITCH is completely independent of the particular local and global predictors. The recognition is based on consecutive references: whereas the individual process reference streams in both locally- and globally-sequential access patterns are *ordered*, only in local patterns are they likely to be a *consecutive* set of block numbers.[6] The SWITCH predictor concurrently monitors the local reference pattern for each process. If any process makes a non-consecutive reference, the pattern is assumed to be global. If all processes make three or more consecutive references, the pattern is assumed to be local. Once the decision is made, each process switches to the appropriate (local or global) predictor. This involves passing the recorded reference pattern (represented by the first and last block numbers of a consecutive run, plus one possibly non-consecutive reference) to the predictor.

Local predictors do not lose any information by this technique, but global predictors lose the inter-process interleaving order of the blocks. For global predictors, however, the number of blocks involved in the switch is usually small, because the switch is made early in the pattern. This limits the difference between the actual interleaving and the interleaving recorded during the switch.

We ran a full set of experiments using the SWITCH predictor with IOPORT as the local predictor and GAPS as the global predictor. The record size was one block. The results for each experiment were compared with former results for the identical experiment with either the IOPORT or GAPS predictor. In every test case except those involving the **lrp** pattern, SWITCH chose the correct predictor. The **lrp** pattern we used occasionally had short portions (two or three blocks), so the pattern did not meet the consecutive-references criterion and the global predictor (GAPS) was used. An **lrp** pattern with longer portions was correctly switched to a local predictor (IOPORT). A fully-general solution would add some of solution 3 (above) to GAPS; if a local pattern is suspected, switch back out of GAPS and into the local predictor.

We measured the percent difference in the total execution time between SWITCH and either GAPS or IOPORT, whichever was appropriate for the test. This represented the percent overhead

[6]Note that a global pattern with large (many blocks) records can also be viewed as a local pattern, and would be by SWITCH.

required by the SWITCH predictor. The **lrp** pattern, due to the poor choice of the global predictor, was much slower than with IOPORT (2400%). The differences for most other cases are so small (-8.4% to 8.8%) that they are essentially insignificant compared to measurement error. (One case, **grp** with *neighbor(10)* synchronization, was 12.1% slower using SWITCH.) Thus, the overhead was minimal. Note that, except in **lrp**, all of the overhead is at the start of the pattern, and essentially represents a delay in the start of prefetching by the predictors. For longer patterns, the SWITCH overhead percentage would be smaller.

It is therefore possible to construct a reasonably efficient general-purpose predictor for the types of sequential access patterns we consider. SWITCH chose the correct predictor in all cases except in the **lrp** pattern with short portions. Another case where it might have difficulty is a local pattern where some processes reference blocks much more slowly than other processes. Some modifications may be necessary to handle these cases more intelligently. In addition, the SWITCH overhead was usually quite small. Although SWITCH with GAPS and IOPORT may be a fair general-purpose predictor, it may be possible to use extra information to choose the predictor that gives best performance for a particular type of pattern (see Section 10.3).

7.8 Conclusion

We present one primary global-pattern predictor (GAPS), a variant (RGAPS), and a specialized predictor (GW). In our experiments, these three predictors each managed to successfully reduce the execution time of global access patterns in most cases, approaching the best time, represented by the EXACT predictor, closely in many cases. The GAPS predictor had a high overhead, which may limit its scalability (see Section 8.5). The overhead of the RGAPS variant was lower, and its concurrency was high, so it should be more scalable. In our experiments, the RGAPS and GAPS predictors were essentially equivalent in most cases, except in the **grp** pattern, where RGAPS was faster than GAPS. From these experiments, it is not clear whether RGAPS was strictly better than GAPS. Although GAPS was more robust for short-portioned **gfp** patterns, the next chapter shows that RGAPS was usually superior to GAPS. In particular, Section 8.1 shows that GAPS could not handle large record sizes. The GW predictor did well on **gw** and **lw** patterns, matching EXACT, although GAPS and RGAPS came close to this performance.

We studied the effect of the MaxDist parameter and found that MaxDist should not be too small (for these tests, less than about 15 or 20 blocks), but a MaxDist that is too large is rarely a significant problem. There is thus a lot of latitude for this parameter. We also studied the significance of the portion length of the pattern. As expected, longer portions allow for greater efficiency and better performance from all predictors.

Any real implementation of GAPS or RGAPS might obtain better performance through further tuning, new cache replacement algorithms, the use of programmer-supplied hints, or optimizing for a particular workload.

Chapter 8

Effect of Architectural and Workload Parameters

In all of the experiments described so far, most of the parameters to the RAPID-Transit file system testbed were fixed while we explored the potential for prefetching (in Chapter 5) and the capabilities of various on-line predictors for local and global access patterns (in Chapters 6 and 7). In this chapter we investigate the effect of some of these other parameters on prefetching across our full range of access patterns, using a few selected predictors. We begin in Section 8.1 by changing the record size, which is an important factor in predictors like GAPS as well as in the replacement algorithm. Then, we vary the cache size in Section 8.2, and examine how prefetching might use more or fewer buffers. In Section 8.3 we examine the effect of fast and slow disks. Section 8.4 discusses varying the number of disks, and Section 8.5 discusses varying the number of processors. Finally, in Section 8.6, we pull all of this together with some overall conclusions.

8.1 Varying the Record Size

So far all of our experiments have used access patterns consisting of records that were all exactly one block long (1024 bytes). The record size is the size of each request to the file-system interface, which converts the request into a set of individual block requests for the file system. Note that sequential access with records that are not an integral number of blocks results in multiple references to some blocks. Records larger than a block can radically change the block-request sequence of globally-sequential access patterns. These are two ways the record size can affect the performance of the file system. In this section, we explore the effects of different record sizes on the potential for prefetching and on the predictors.

8.1.1 Experiments

We ran experiments varying the record size from $\frac{1}{4}$ block (256 bytes) to 10 blocks (10240 bytes). Note that it was the relationship between the record size and the block size that was important, not the actual sizes in bytes. We used the NONE and EXACT predictors as well as on-line predictors (GAPS and RGAPS, or IOPORT, as appropriate). To keep things simple, we used only the *none* synchronization style. This keeps synchronization effects separate from record-size effects. The experiments also did not include any computation with each record. This I/O-bound workload stresses the file system to its maximum. All other parameters were the same as usual: 20 processors, 20 disks, 1 KByte block size, 80-block cache, and five trials per test case.

We experimented with the **lfp**, **lw**, **seg**, **rnd**, **grp**, **gfp**, and **gw** patterns. For any given pattern, several variants were constructed with various record sizes. The *block* access pattern was identical across all variants of a particular pattern (except for **rnd**). It was not possible to build similar **lrp** variants due to the short, random portion lengths. Thus, the **lrp** pattern was not included in these experiments.

8.1.2 Results and Discussion

The results are presented in Figures 8.1–8.7. The total execution time, averaged over five trials, is plotted as a function of record size. The maximum coefficient of variation (cv) is noted for each figure. With some (noted) exceptions, the curves are tight.

To help in interpreting the results, we compare the experimental execution time to a simple model of the ideal execution time (page 31). There is no computation in these experiments, only 4000 block reads each requiring 30 msec of I/O. Ideally these are spread evenly over 20 disks, so the ideal I/O time is 6 seconds. One exception is the **lw** pattern, which ideally has only 200 block reads spread over 20 disks, so the I/O time is 0.3 seconds. The ideal execution time is plotted as a dotted line in all of the figures in this section.

There is one effect that was common to all predictors in all patterns. Whenever blocks may contain multiple records (either whole small records or parts of larger records), the file system processes some blocks many times. We have kept the number of blocks in the pattern constant to maintain a constant I/O time (except in the **rnd** pattern; see page 108). Thus the number of file system requests (and hence the file system overhead) is related to the number of records in the pattern. This effect was most evident as a speedup in many cases as the record size increased from $\frac{1}{4}$ to one block.

Local Patterns

The results for the local patterns are shown in Figures 8.1–8.3. The NONE predictor was unaffected by the record size except as mentioned above. (The smaller variations are noise in the data.) The same was true for EXACT and IOPORT on the **lw** pattern (Figure 8.2). EXACT and IOPORT were mostly unaffected by record size in the **lfp** pattern. We expect **lrp** to behave similarly to **lfp**.

IOPORT was erratic in the **seg** pattern, shown in Figure 8.3. The standard deviation of the five trials was as large as the deviations observed for different record sizes. Random runtime fluctuations determined the order that processes were served by the disks. Because of the heavy disk contention, shuffling the disk service ordering could significantly change the execution times. For the same reason, the cost of mistakes by the IOPORT predictor (not an issue in the EXACT and NONE predictors) varied wildly. It is not possible to make any significant conclusions for **seg** except that it was roughly independent of the record size.

Global Patterns

The results for the global patterns are shown in Figures 8.4–8.6. All predictors were slower for non-integral record sizes due to several effects. First, there were more file system references since each block was requested several times, so there was more overhead. Second, different processes were sometimes simultaneously reading several records within a block, which resulted in competition for the data structures for that block. Third, until the I/O was complete several processes were waiting for the same block. Without prefetching, no other I/O was started, and several disks were idle at all times. The poor disk utilization is reflected as slower execution times for NONE on the

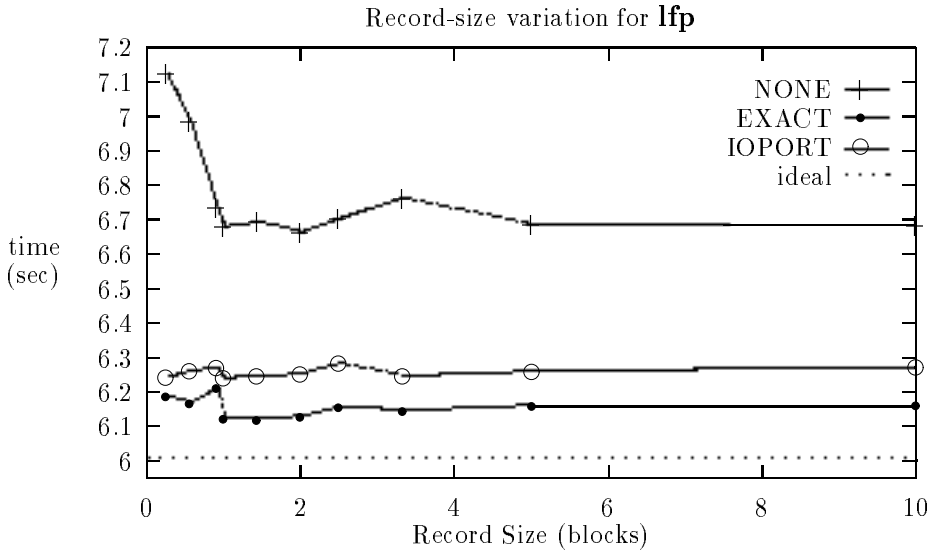


Figure 8.1: Total execution time for **lfp** as a function of record size. Note y axis does not begin at zero. ($cv < 0.038$)

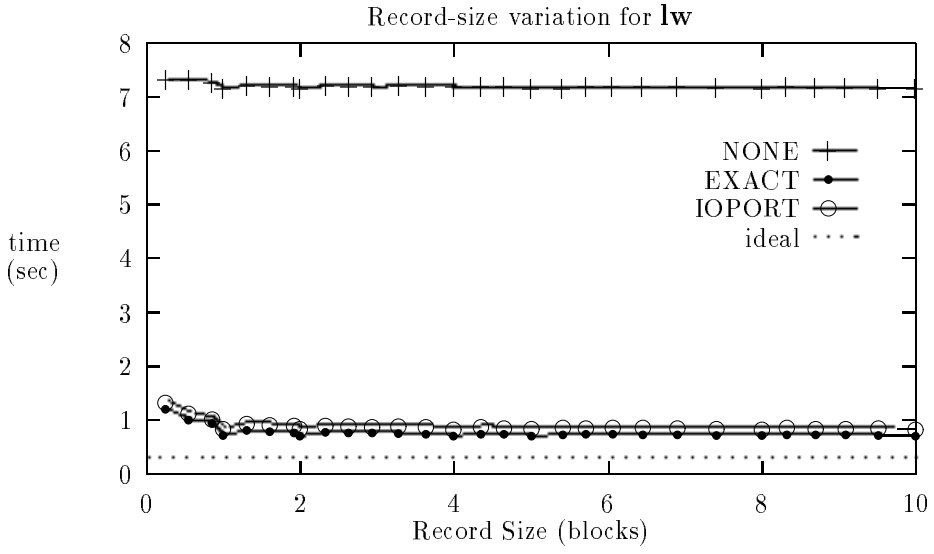


Figure 8.2: Total execution time for **lw** as a function of record size. ($cv < 0.098$)

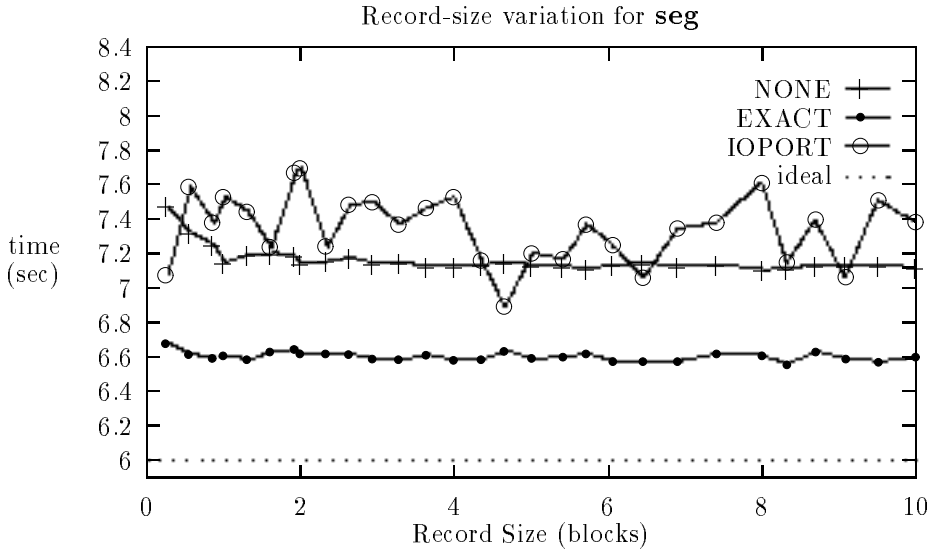


Figure 8.3: Total execution time for **seg** as a function of record size. Note y axis does not begin at zero. Here IOPORT had $0.022 < cv < 0.101$. ($cv < 0.010$ otherwise)

non-integral sizes, with the most acute effects in record sizes less than one block. This effect was also evident, although to a lesser extent, when prefetching for non-integral record sizes.

Finally, the replacement algorithm did not handle multiple references per block well, and some blocks were flushed from the cache before they had been fully used. This caused extraneous disk I/O. This points out the need for a better replacement mechanism to avoid flushing blocks that may be used by different processes. A simple solution is to avoid flushing blocks until they have been referenced a given number of times. This solution is inadequate, however, since for many record sizes (e.g., $\frac{3}{2}$ blocks) the number of references per block is not a single constant. It would also add complexity and interdependence between the prediction and replacement algorithms. Another possibility is a method similar to that used to solve the **lw** phase problem (see Section 6.4.2), where blocks remained in the cache until all processes "interested" in the block had used it.

Note that for integral record sizes (1, 2, 4, 5, 8, and 10 blocks) the execution time for EXACT, NONE, and RGAPS was nearly constant (roughly 6–7 seconds), and close to the ideal execution time.[1] Thus, there were no other record-size dependent effects on these predictors.

The GAPS predictor slowed down significantly for records larger than two blocks. This was due to the increased effort needed to recognize sequentiality in the block access pattern, since a mismatch between the record size and the block size made the already convoluted global reference string even more convoluted. In each case, records larger than four blocks caused GAPS to fail completely, prefetching no blocks and running 10 times slower (the GAPS curves go off the scale at this point, although it worked a little for **gw** with 5-block records). This was planned into the design of GAPS, with the understanding that larger record sizes can be treated as local reference patterns (and they are more efficiently treated as local patterns).

From these figures it is now clear that RGAPS is superior to GAPS (at least for these access patterns). RGAPS parallels the performance of EXACT throughout, only slightly slower. GAPS only

[1] We use these record sizes because they divide the 4000 blocks into an integral number of fixed-size records.

comes close to RGAPS for some small (0–2 blocks) record sizes. This reinforces the interpretation of the GAPS slowdown as a problem with recognizing sequentiality.

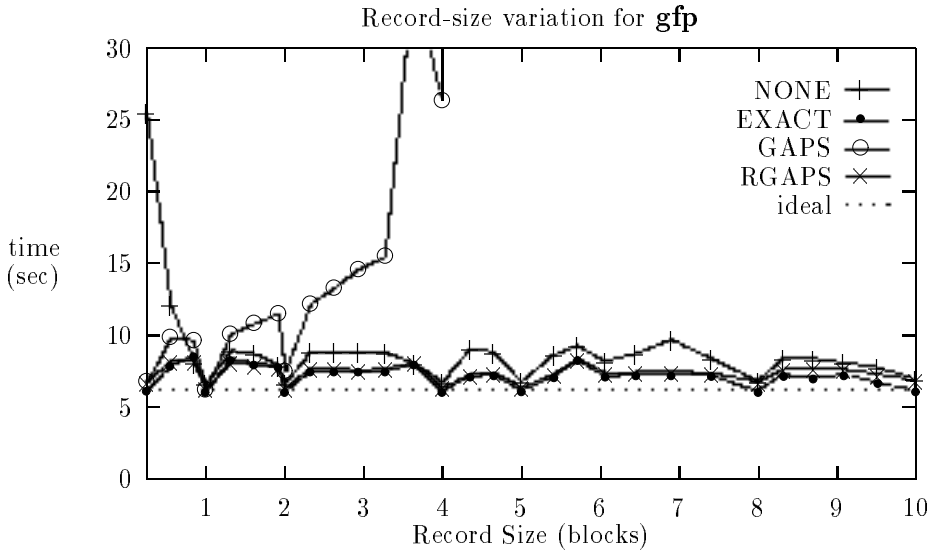


Figure 8.4: Total execution time for **gfp** as a function of record size. ($cv < 0.080$)

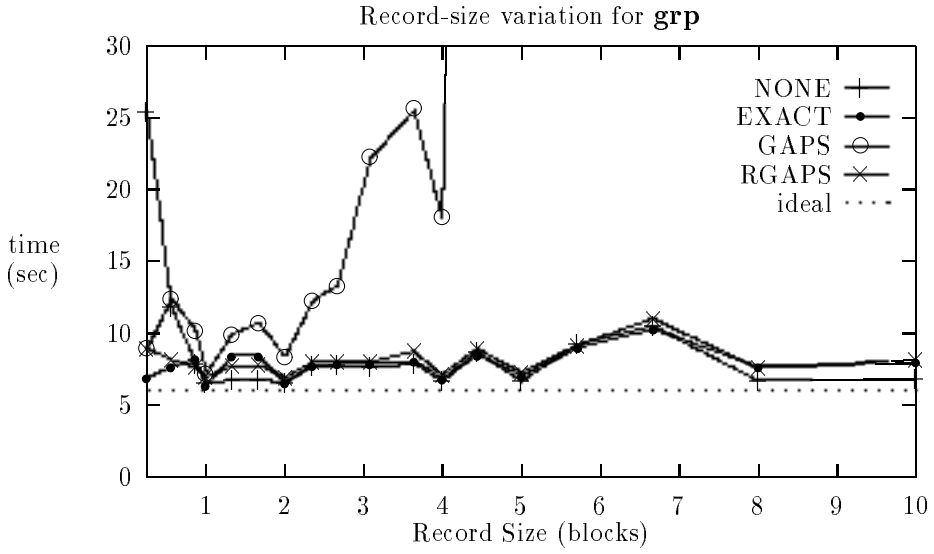


Figure 8.5: Total execution time for **grp** as a function of record size. The GAPS experiments with records larger than 3 blocks were highly variable ($cv < 0.431$), but the significant slowdown is still clearly evident. ($cv < 0.081$ otherwise.)

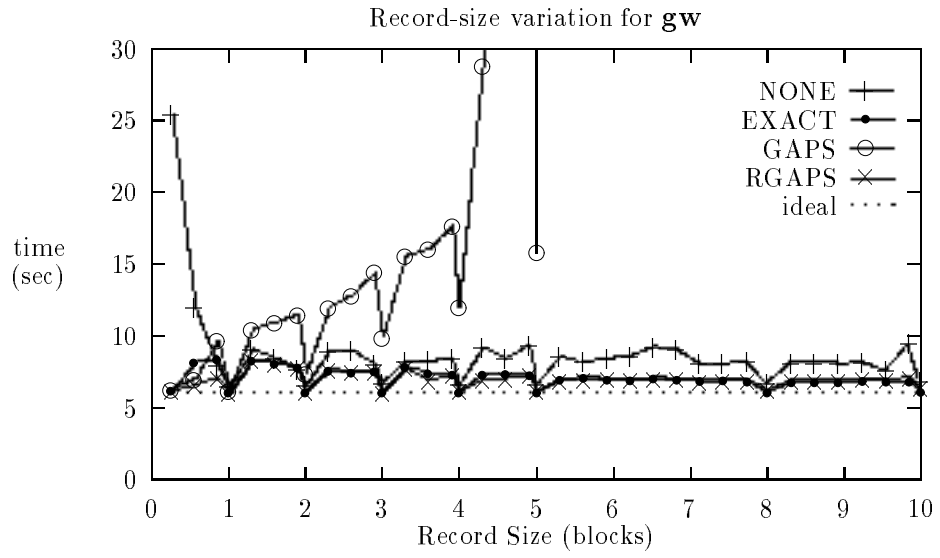


Figure 8.6: Total execution time for **gw** as a function of record size. ($cv < 0.20$)

Random Pattern

In our previous experiments we used the **rnd** pattern to show that some predictors handled random patterns intelligently. Those tests used 1-block records. With other record sizes, there are other issues: parts of some blocks are unused, and some sequentiality is present. We now show how our predictors were affected by these issues. We experimented with the **rnd** pattern by generating random patterns with different record sizes. In each pattern all references were to fixed-length records aligned on record boundaries, much like random access to a file of fixed-length records. We used the NONE, IOPORT, GAPS, and RGAPS predictors.

There were several interesting effects, shown in Figure 8.7. First, the execution time improved with longer records. This was because of a reduction in wasted I/O bandwidth. For each record of r blocks, up to $\lceil r \rceil + 1$ blocks were read. Since the patterns were random, adjacent records were rarely needed, wasting $\lceil r \rceil + 1 - r$ blocks. The total wasted bandwidth decreased, on the whole, with increasing r. The waste was especially large for $r < 1$. For integral record sizes, r blocks were read and r were used, so there was no waste. Thus, the integral record sizes were fastest.

Also note that IOPORT was faster than NONE for all but the smallest records. Once records used more than one block (even records smaller than one block may involve two adjacent blocks), there was some sequentiality and prefetching was possible. In other words, the block access pattern looked to IOPORT like **lrp**, instead of pure **rnd**. On the other hand, the execution time for RGAPS was usually a little slower than NONE. There was no sequentiality on a global scale, so RGAPS was able to do little prefetching. GAPS was so much slower than the others that most of the curve is off the scale. The **rnd** pattern with records larger than one block had just enough sequential access to keep GAPS in Watch mode, but not enough to do any prefetching. This overhead caused the GAPS execution time to exceed 200 seconds in some cases.

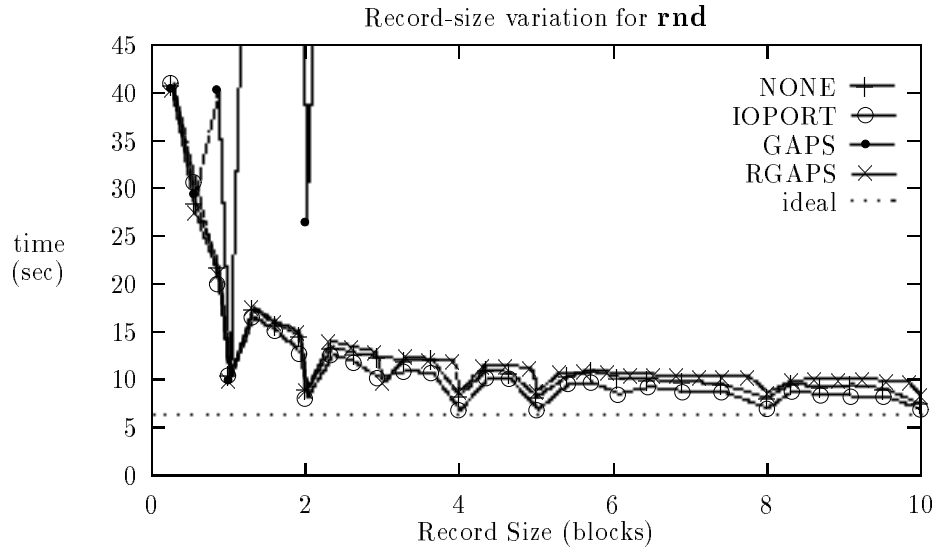


Figure 8.7: Total execution time for **rnd** as a function of record size. ($cv < 0.025$, except for GAPS: $cv < 0.33$)

8.1.3 Conclusions

There are several significant conclusions:

- GAPS had its best performance for one-block records. Since all our previous experiments with GAPS used this record size, those results are optimistic.
- GAPS failed for large record sizes (more than 4 blocks), and for random-access patterns with records larger than one block.
- RGAPS was clearly more general than GAPS when varying the record size, at least on these access patterns. Other than being a little slower for non-integral record sizes, its performance was little different from that in Chapter 7.
- Prefetching was possible in the **rnd** pattern for some record sizes, and IOPORT gained some improvement over NONE.
- There is a need for tuning the replacement algorithm, to avoid flushing blocks before all references (especially from separate processes) are completed.

8.2 Varying the Cache Size

All of our experiments so far used 20 processes. Thus the minimum cache size was 20 blocks (see page 25). We used an 80-block cache, implying a prefetch limit of 60 blocks when prefetching. We chose a 60 block prefetch limit based on preliminary experiments that showed it to be a reasonable compromise. In this section we examine the effect of this parameter (effectively, the cache size) on prefetching. We determine what cache size is appropriate for each workload, and why.

8.2.1 Experiments

Using all of our access patterns except **rnd**, we varied the prefetch limit from 20 blocks to at least 120 blocks, and in some cases as high as 440 blocks. Remember that the prefetch limit is in addition to the base cache size of 20 blocks. The NONE and EXACT predictors were used in every case, with the IOPORT predictor included for all local patterns and the GAPS and RGAPS predictors included for all global patterns. All four synchronization styles were used, and only the I/O-bound (no-computation) experiments were included. All other experimental parameters were the same as usual: 20 processors, 20 disks, 1-KByte block size, 1-block record size, and five trials per test case.

Although the prefetch limit does not directly affect experiments with the NONE predictor, to be fair we also varied the cache size in those test cases. Although our access patterns are primarily sequential, some patterns (**grp**, **lrp**, and **rnd**) have a few repeated block references and thus are affected by different cache sizes.

8.2.2 Results and Discussion

We only present the data for the *each(10)* synchronization style, since we found the behavior depended little on the synchronization style. The total execution time was measured for all test cases (with coefficient of variation never larger than 5%). This was compared with the no-prefetching execution time in terms of the percent improvement due to prefetching. With this measure, large positive percentages are desirable. Figures 8.8–8.13 plot the percent improvement as a function of the prefetch limit.

Local Patterns

The data for local patterns are shown in Figures 8.8–8.10. Here we examined the IOPORT and EXACT predictors. The performance was heavily dependent on the prefetch limit except in the **lw** pattern. To examine the effects of large prefetch limits, we extended the experiments for **lfp**, **lrp**, and **seg** out to a prefetch limit of 440 blocks. This was more than enough buffers for every process to have one prefetch outstanding on every disk simultaneously.

Both EXACT and IOPORT were able to use many prefetch buffers in the **lfp** pattern (Figure 8.8). Since IOPORT was limited by MaxDist = 5, its performance curve has a sharp knee at prefetch limit 100 (20×5). EXACT leveled off at around 280 blocks. Note that EXACT was slower than IOPORT, and both were slower than NONE (i.e., negative improvement), for small caches (less than about 100 blocks); this was due to the greedy-process problem. For prefetch limits of 100 or more, both IOPORT and EXACT were finally faster than NONE. For limits over 200, EXACT finally beat IOPORT. Thus, one solution to the greedy-process problem is to increase the cache size. The primary conclusion here is that a reasonably large prefetch limit gave the best performance, though there were diminishing returns.

In the **lrp** pattern (Figure 8.9) the performance of EXACT improved significantly with increasing prefetch limit. Unburdened by mistakes and most prefetch limitations, EXACT was able to use more prefetch buffers to reduce both the number of misses and the wait time associated with buffer hits. No additional prefetching benefits were possible for IOPORT, since it was in OBL mode and could not use much more than 20 buffers for prefetching. All predictors (including NONE) benefited from an increased hit ratio from random re-references in larger caches. This effect is cancelled out in our presentation of percent improvement.

The **lw** pattern (not shown) was essentially not affected by the prefetch limit. EXACT was constant at 87% improvement, and IOPORT at 86% improvement, independent of the cache size. In other words, 20 prefetch buffers were sufficient. This is no surprise, since every block was used

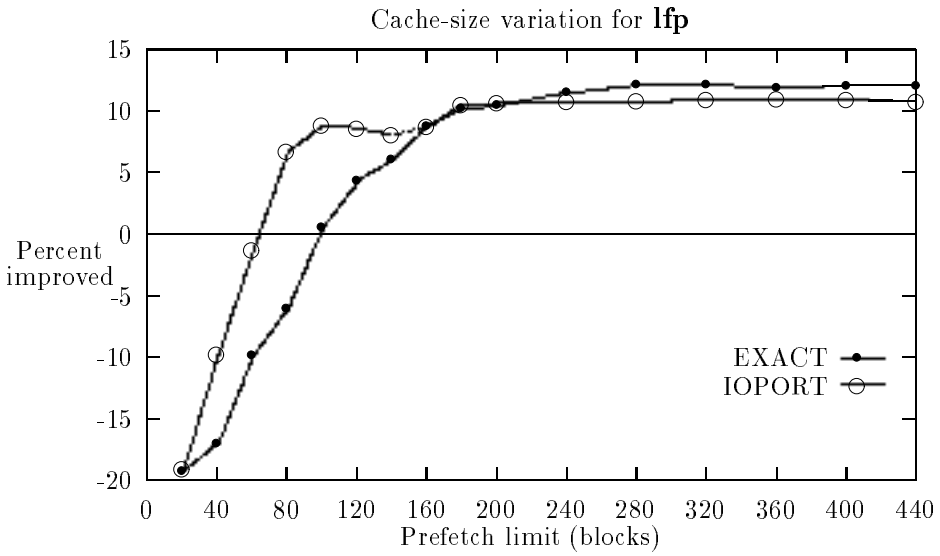


Figure 8.8: The percent improvement of prefetching over not prefetching, for the **lfp** pattern. ($cv < 0.027$)

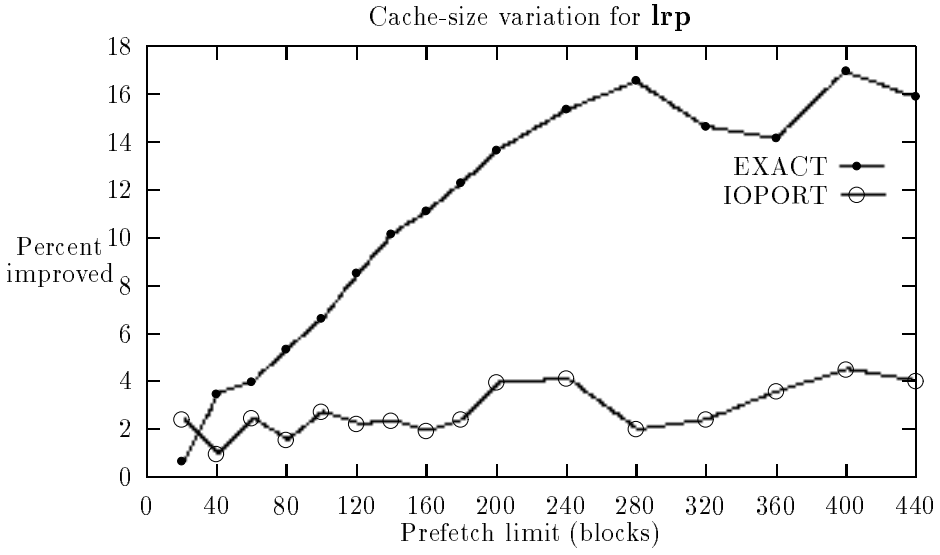


Figure 8.9: The percent improvement of prefetching over not prefetching, for the **lrp** pattern. ($cv < 0.023$)

by every process, significantly reducing overall buffer space requirements. A 40-block cache was large enough to keep all the disk queues full, so the disks were always busy.

In contrast, it was clearly important to use many prefetch buffers in the **seg** pattern (Figure 8.10). This was due to the high-contention disk access pattern in **seg**. More prefetch buffers meant more prefetching, and more prefetching allowed for much better disk utilization. This was true both for EXACT and for IOPORT.

The curves in Figure 8.10 leveled off at a prefetch limit of 200 blocks. This point corresponds to the total size of one synchronization interval (20 processors doing 10 blocks each), and reflects **seg**'s use of only 10 disks during any one synchronization interval.

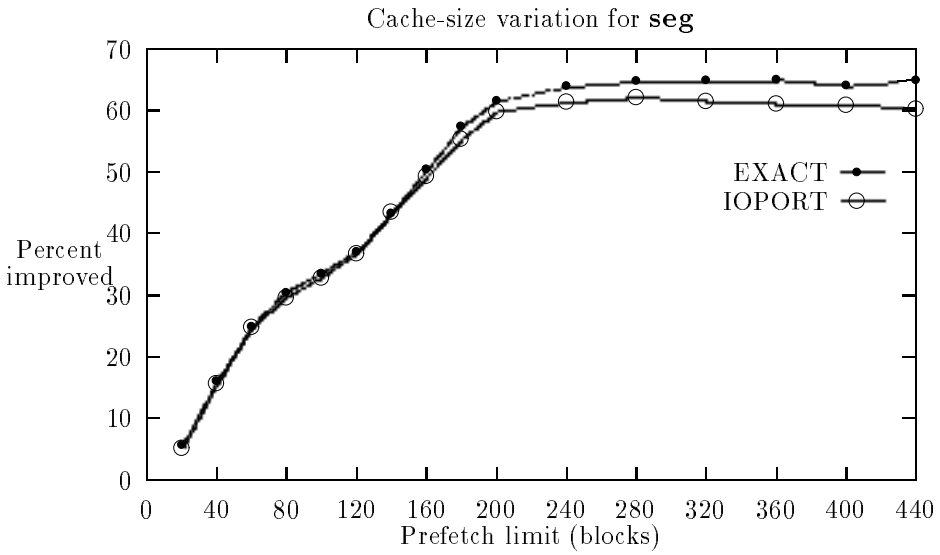


Figure 8.10: The percent improvement of prefetching over not prefetching, for the **seg** pattern. ($cv < 0.048$)

These experiments provide valuable information about the effect of cache size on local pattern prefetching. Larger caches helped to solve the greedy-process problem in **lfp** and to handle disk contention in **seg**, but made little difference to **lrp** and **lw**. These results show that our previous results about prefetching in local patterns are pessimistic. The previous results for **lfp** show that our on-line predictors were slightly faster or slightly slower than not prefetching at all, whereas a 200-block cache gave a solid 10% improvement. The previous results for the **seg** pattern show 25% improvement, but 60% improvement was possible with a 220-block cache. Results for **lrp** and **lw** changed little.

Global Patterns

For the global patterns we studied the EXACT, GAPS, and RGAPS predictors. The data are shown in Figures 8.11–8.13. The EXACT predictor generally leveled off in all three patterns after a 40-block prefetch limit. Apparently 40 prefetch buffers were sufficient to keep all the disks busy and to minimize the number of cache misses. Indeed, with all predictors the disk utilization was consistently over 80% for **grp**, 90% for **gfp** and **gw**.

The performance of the GAPS and RGAPS predictors steadily declined for **grp** (Figure 8.12) and most of **gfp** (Figure 8.11). This was due to an increasing number of mistakes in the first portion, where the only limit on mistakes was the prefetch limit. In **gfp**, there was an improvement from a 20-block to 40-block prefetch limit, where there was a reduction in the number of cache misses to offset the increased number of mistakes. Note that adjustments in the prefetch limit did not allow either GAPS or RGAPS to attain positive improvement in the **grp** pattern.

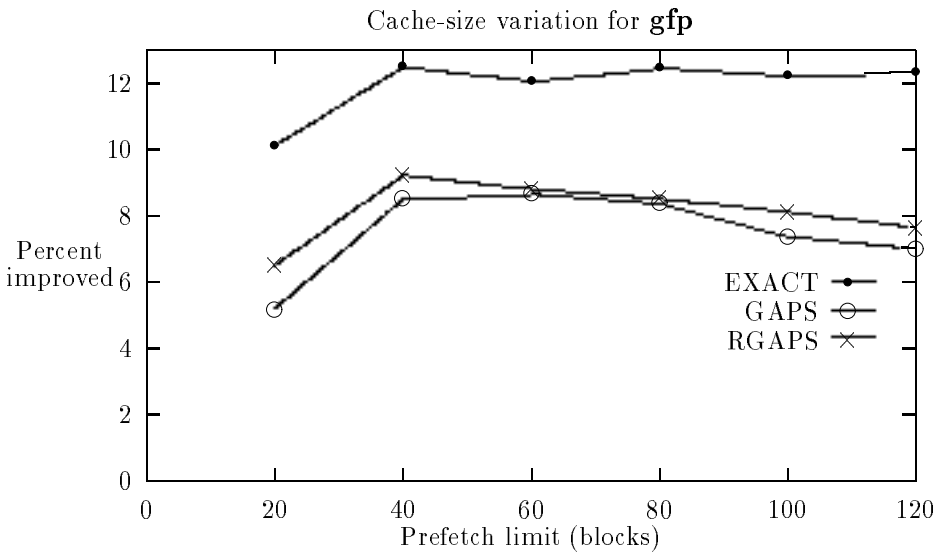


Figure 8.11: The percent improvement of prefetching over not prefetching, for the **gfp** pattern. ($cv < 0.025$)

Finally, in the **gw** pattern (Figure 8.13), GAPS and RGAPS were unaffected by varying the prefetch limit (the fluctuations were smaller than the measurement error). In short, it seems that 40 blocks (two per process) were sufficient for prefetching in global patterns. There was not much difference between prefetch limit 40 and 60, so our other experiments using prefetch limit 60 were close to the best performance we can expect (at least in terms of prefetch limit).

8.2.3 Conclusions

In the I/O-bound patterns that we examined, disk utilization was the key to good performance. Prefetching (usually) improved disk utilization by filling the disk queues with appropriate requests, avoiding costly disk idle time. Unless the disk utilization was already nearly perfect, the availability of more prefetch buffers allowed more prefetching and hence higher disk utilization and higher performance. This effect is clear in the results from the local access patterns. The global access

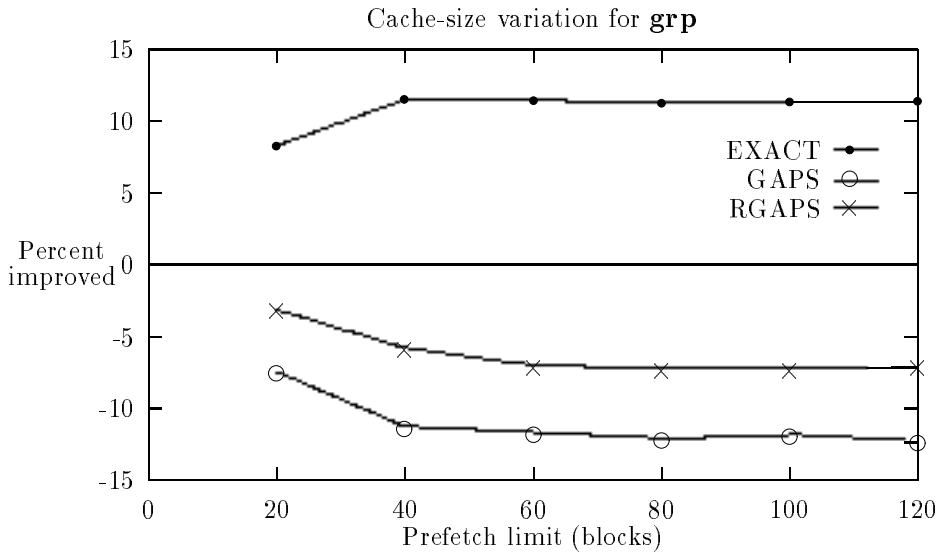


Figure 8.12: The percent improvement of prefetching over not prefetching, for the **grp** pattern. ($cv < 0.026$)

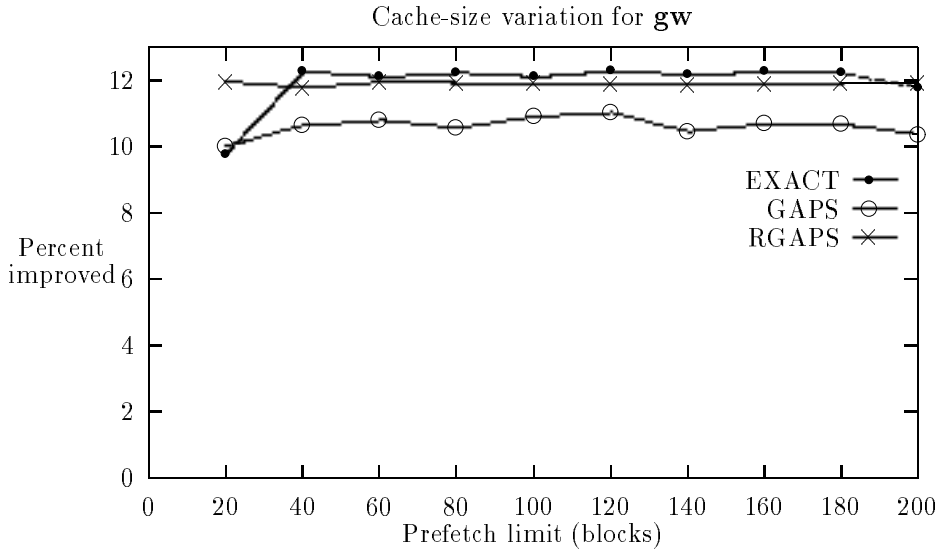


Figure 8.13: The percent improvement of prefetching over not prefetching, for the **gw** pattern. ($cv < 0.026$)

patterns, however, had high disk utilization (over 80 or 90%) even for small cache sizes and thus were not much improved by adding buffers. This reflects the contrast between the cooperative nature of global patterns and the private nature of local patterns: in global patterns, prefetching a block into the cache benefits the entire computation, whereas in local patterns it benefits only one process. The self-serving nature of prefetching in local patterns, and their less-uniform disk access patterns, may be the reason why they require more prefetching buffers. More prefetch buffers would probably also be helpful if the application mixed computation and I/O, when more aggressive prefetching is useful.

Our original choice of an 80-block cache was indeed a compromise between some local patterns, which preferred large (220-block) caches, and all global patterns and the other local patterns, which were less affected by cache size and which were content with a 60-block cache. Given a larger cache, the benefits of IOPORT to the **lfp** and **seg** patterns were better than we report in Chapters 5 and 6.

Because the best prefetch limit seems to depend on the access pattern, and is fairly easy to adjust in a running system, it is possible to use the predictors' information to adjust the prefetch limit. A simple policy is to use a large prefetch limit (e.g., 200) for local patterns, and a small prefetch limit (e.g., 40) for global patterns. This could be built into the SWITCH predictor.

8.3 Varying the Disk-Access Time

In RAPID-Transit the physical disk is modeled by a constant disk access time. In all of our other experiments, the disk access time (per block) was 30 msec. This is roughly the average access time for the kind of small cheap drives that might be replicated in large quantities as part of a parallel disk system [PGK88, Sch88]. This also ignores disk layout and any benefit that might come from physically contiguous disk access.

Although one motivation for parallel disk systems is the slow rate of improvement in disk access time, disks are getting faster. We study the effect of faster and slower disks in this section. Although we are not changing the speed of our processors (which will also happen as technology changes) our disk-access time variation is essentially a study of the *relative* speed of processors and disks. Thus, the study of slower disks is useful, since slower disks are analogous to faster processors, or to a situation where both processors and disks are faster, but the speed differential is wider than it is now. The latter is the expected future. The study of faster disks is also useful, in case some architectures have a narrower gap between processor and disk speeds.

We also use these experiments to determine how well prefetching overlaps I/O and computation.

8.3.1 Experiments

We varied the physical disk access time from 10 msec to 50 msec for our usual set of experiments. Thus, we used the **lrp**, **lfp**, **lw**, and **seg** patterns with NONE, EXACT, and IOPORT predictors, and the **grp**, **gfp**, and **gw** patterns with NONE, EXACT, GAPS, and RGAPS predictors. We used all four synchronization styles with each of the above combinations, and used both the no computation (I/O-bound) and computation variants. The computation, when present, averaged roughly 30 msec per block, and was the same across all experiments with a given pattern. All of the other experimental parameters were the same as usual: 20 processors, 20 disks, 1-KByte blocks, 1-block records, 80-block cache, and five trials per test case.

8.3.2 Results and Discussion

We present only the results for the *each(10)* synchronization style. The other styles usually had similar results, and the few exceptions are pointed out. The coefficient of variation of the total execution time is provided for each figure.

To help in interpreting the results, we compare the experimental execution time to a simple model of the ideal execution time. Recall that the ideal time T is

$$T = \max(\text{I/O time}, \text{comp time}).$$

There are 4000 block references in our patterns, each requiring t seconds of I/O, with t varying from 10 to 50 msec. Ideally these are spread evenly over 20 disks, so the ideal I/O time is $200t$. One exception is the **lw** pattern, which ideally has only 200 disk reads spread over 20 disks, so the I/O time is $10t$. In the I/O-bound experiments there was no computation, so the ideal execution time is the ideal I/O time, either $200t$ or $10t$.

In the experiments with computation, the total amount of computation (C) varied slightly since it was randomly generated, but it was roughly 100–120 seconds for all patterns. There were 20 processors in all these tests, so the parallel computation time was (ideally) $C/20$ and hence

$$T = \max(200t, \frac{C}{20}) \text{ seconds.}$$

Similarly, for **lw**, the ideal is

$$T = \max(10t, \frac{C}{20}) \text{ seconds.}$$

In all plots of these experiments, we show the "ideal" execution time as a dotted line, using the actual value for C. The ideal curve may be somewhat obscured by the experimental-result curves.

I/O-bound Experiments

In these experiments there was only I/O, and no computation. This corresponds to the extreme case of processors that are so fast that I/O is totally dominant. The ideal disk I/O time assumes perfectly balanced disks and no extra overhead. If the line for a given experiment has a slope greater than the ideal line's slope, the experiment did not have a balanced disk load. If they have the same slope, but different intercept, then the difference is some other source of overhead (e.g., synchronization time) that is independent of the disk access time.

Global Patterns. The results for I/O-bound global patterns are shown in Figures 8.14–8.15. In Figure 8.14 the total execution time of the **gfp** pattern is plotted for NONE, EXACT, GAPS, and RGAPS as a function of the disk access time. All predictors matched at least the *slope* of the ideal curve, indicating only a constant overhead, for all but the fastest disks. Our results thus scale well to the situation when processor speed has increased relative to disk speed, represented here by the slowest disks. An interesting effect, however, occurred with fast disks: GAPS and RGAPS degraded for disks with access times less than 20 msec. EXACT also degraded, but only for disks faster than 15 msec. (The precise point also varied a little depending on the synchronization style.) There is a simple explanation for this effect. Prefetching in global patterns added a substantial amount of overhead. For slow disks, there was plenty of idle processor time available to be used for prefetching, and the overhead was absorbed by the idle time. In addition, the benefits of prefetching could be quite large, by avoiding lengthy disk waits. With fast disks, however, the overhead time remained the same, but the idle processor time and the potential benefits were both reduced. In

this case the overhead necessary for accurate prefetching was not worth the small benefits obtained, and only small speedups were obtained by using disks faster than 20 msec. Indeed, for the fastest disks GAPS and RGAPS were slower than NONE.

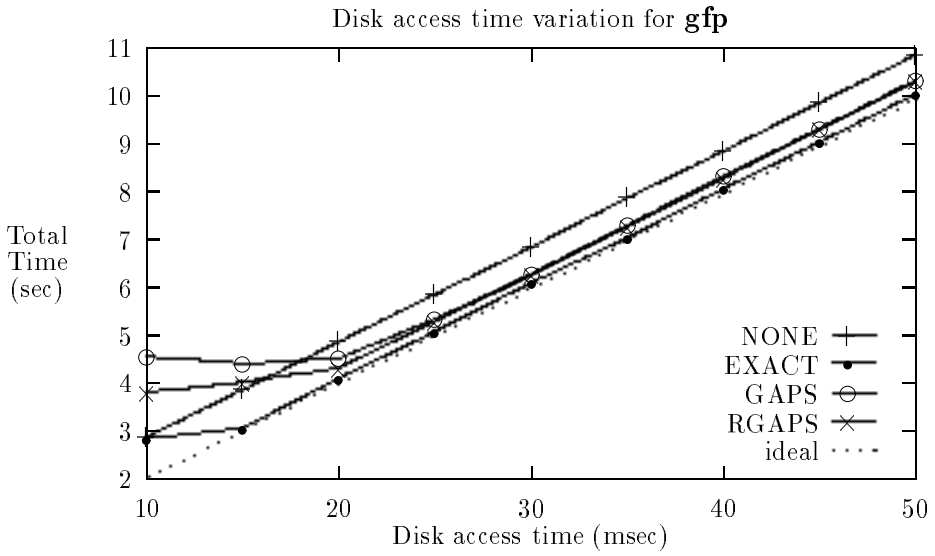


Figure 8.14: The result of varying disk access time for the I/O-bound **gfp** pattern. ($cv < 0.087$)

When considering the poor performance of prefetching with fast disks, remember that the processor speed is fixed in this experiment. Also remember that, as technology improves, processor speeds are increasing faster than disk speeds. Thus, as the fast disks become available, even faster processors are available. Prefetching overhead will be reduced more quickly than the disk-access time, so we expect prefetching performance to remain in the linear part of the curve, where prefetching was successful. Indeed, it may be in the rightmost end of the curve, where disks have slowed relative to processor speed.

The results for **gw** (not shown) were essentially the same as those for **gfp**.

In Figure 8.15, for the **grp** pattern, the slope for NONE was larger than ideal, indicating that the disk load was not quite balanced in this less-regular access pattern. EXACT behaved much like it did on **gfp**, with a slight degradation below 15 msec. Otherwise it seems to have a balanced disk load. GAPS and RGAPS did not have the same sharp degradation point, but changed more smoothly. Note also that they were slower than NONE, as usual for **grp**. In the *none* synchronization style (not shown), the degradation point was sharper. In the *neighbor(10)* synchronization style (not shown), GAPS had so much overhead that it degraded for disks faster than 30 msec, though these results were highly variable (typical for GAPS on **grp** with *neighbor(10)* synchronization).

In summary, prefetching in global patterns should scale well with disk and processor technology, given the expected increasing gap between processor speed and disk speed.

Local Patterns. The results for local patterns are shown in Figures 8.16–8.19. The disk loads were less balanced, as indicated by the slopes that were higher than ideal. In the **lfp** pattern (Figure 8.16), EXACT and IOPORT both exhibited this effect, each with a different slope. This

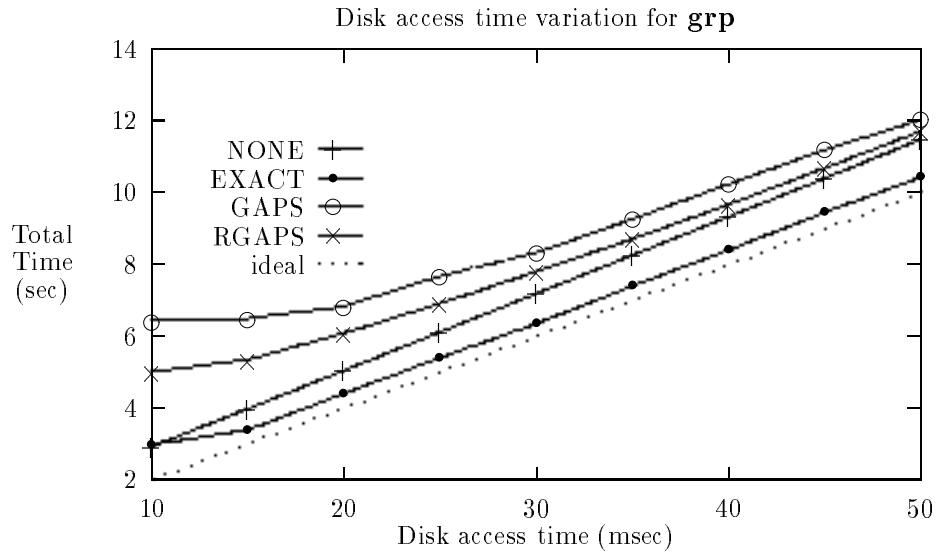


Figure 8.15: The result of varying disk access time for the I/O-bound **grp** pattern. ($cv < 0.085$)

was due in part to the greedy-process problem (note that they were often slower than NONE) and to a disturbance of the uniformity in the access pattern. Because NONE did not disturb the uniform disk access pattern, it had only a constant overhead. These conclusions were supported by the non-synchronized version of these experiments (not shown).

In the **lrp** pattern, shown in Figure 8.17, all three predictors (NONE, IOPORT, and EXACT) had an unbalanced disk load, as indicated by the slopes that were higher than ideal. In this case, the slopes were all about the same, since it was the pattern, not the predictor, that was unbalancing the load.

In the **lw** pattern (Figure 8.18) the NONE predictor has a high slope because the disks were completely unbalanced (only one disk was used at a time). The run times of EXACT and IOPORT, however, were independent of the disk access time, since they were able to use all of the disks and to completely overlap all I/O.

Figure 8.19 shows the results for the **seg** pattern. NONE had poorly-balanced disk accesses, exhibited here by the high slope. EXACT and IOPORT improved the slope, though still higher than the ideal. In the non-synchronized experiments (not shown), the disks were balanced naturally by the pipeline effect. In that case, both NONE and EXACT were better able to balance the disks and had lower overhead, but IOPORT tended to request blocks out of the pipeline order and was slower than NONE for all but the fastest disks.

In summary, the I/O-bound experiments represent the most stressful test of the file system. Our results about local- and global-pattern prefetching that are valid with 30 msec disks seem to scale well to the likely future hardware balance, with faster disks tied to much faster processors.

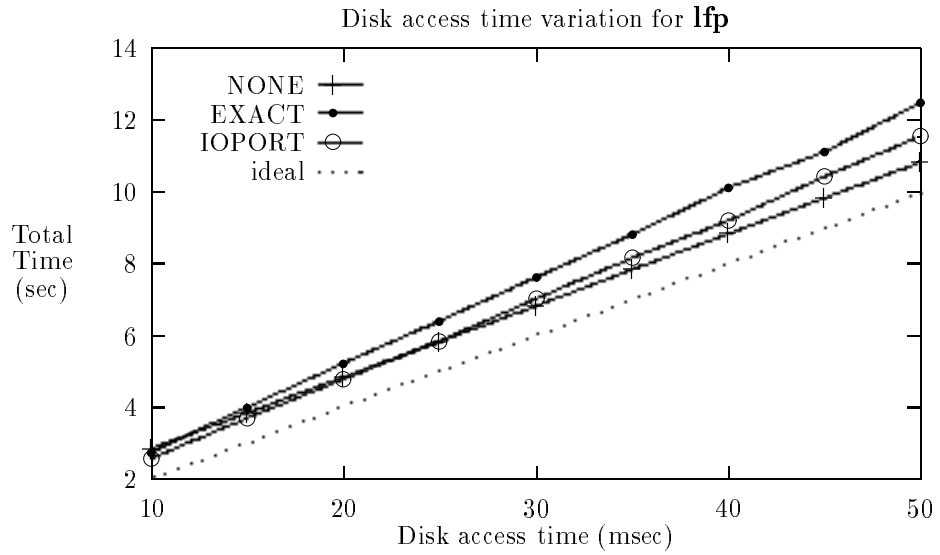


Figure 8.16: The result of varying disk access time for the I/O-bound **lfp** pattern. ($cv < 0.032$)

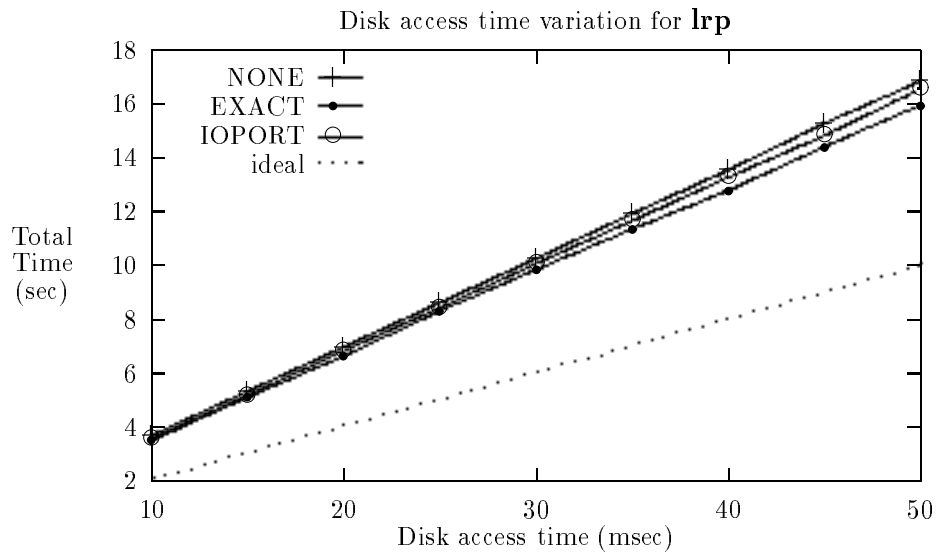


Figure 8.17: The result of varying disk access time for the I/O-bound **lrp** pattern. ($cv < 0.031$)

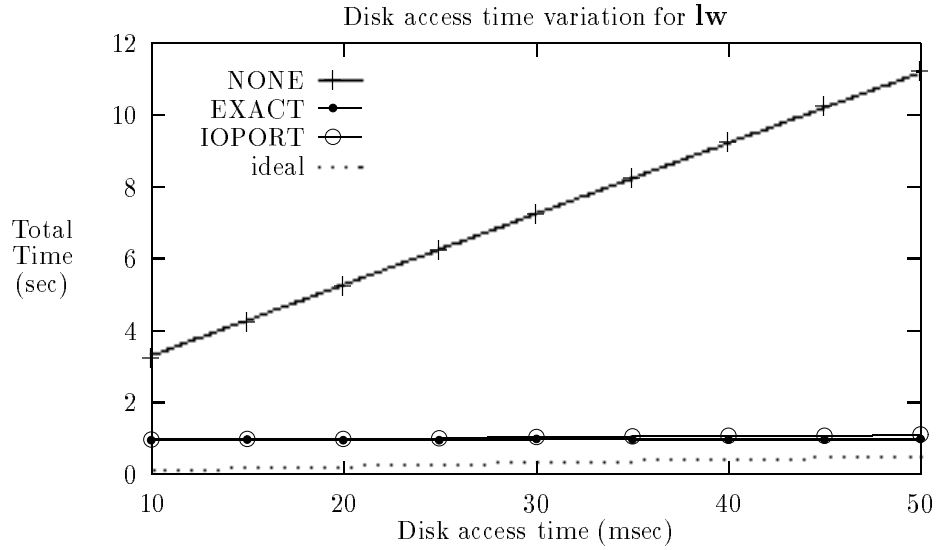


Figure 8.18: The result of varying disk access time for the I/O-bound **lw** pattern. ($cv < 0.029$)

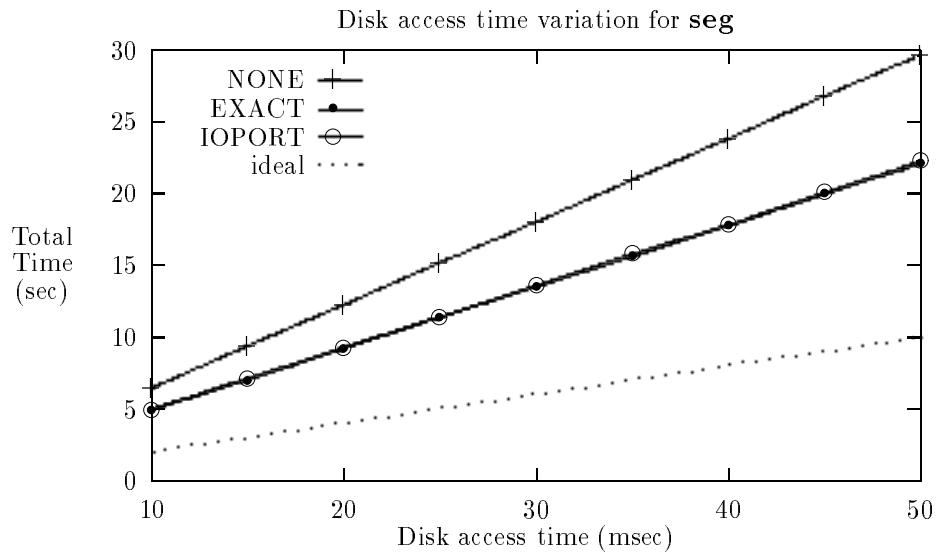


Figure 8.19: The result of varying disk access time for the I/O-bound **seg** pattern. ($cv < 0.013$)

Experiments with Computation

When the application performs some computation on each block read from the file, the load on the file system is reduced and there is another opportunity for prefetching to improve performance: by overlapping I/O and computation. As long as there is more computation than I/O, it is possible for all of the I/O to be masked by computation. This is reflected in the ideal execution time plotted in each figure.

Global Patterns. The total execution time of NONE on the **gfp** pattern with computation increased with the disk access time (Figure 8.20). Prefetching with GAPS or RGAPS was always faster. For fast disks, the I/O time was completely overlapped with computation, so the time was independent of disk access time. With slower disks, the I/O was the bottleneck and thus the total execution time was dependent on the disk access time. This effect is more clear in Figure 8.21, the non-synchronized version of Figure 8.20. Notice that the total time was near to the ideal disk time, indicating that the computation was mostly overlapped by I/O, or *vice-versa*.

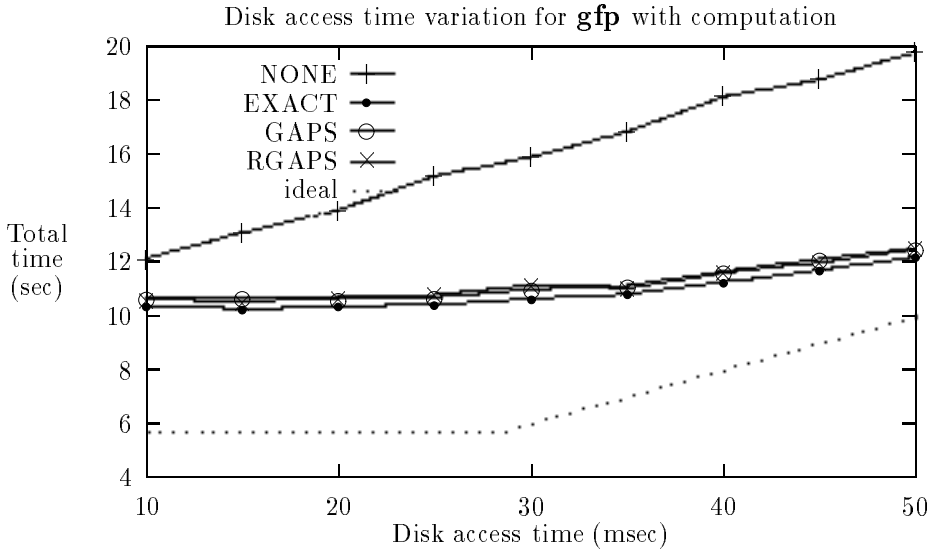


Figure 8.20: The result of varying disk access time for the **gfp** pattern with computation. ($cv < 0.030$)

The results for **grp** and **gw** (not shown) were similar to those for **gfp**. They changed gradually from I/O-bound to compute-bound, unlike in Figure 8.21. As before, the tradeoffs in the non-synchronized versions were more clearly defined. Also, in **grp** GAPS and RGAPS were slower than EXACT.

Local Patterns. The results for **lfp** are shown in Figure 8.22. EXACT and IOPORT were able to overlap part of the I/O with computation, since the slopes of their lines were smaller than the ideal, but NONE was not able to overlap any I/O with computation. The greedy-process problem (page 47) was evident for slow disks.

The total execution time for **lrp** with computation was essentially linear for all three predictors

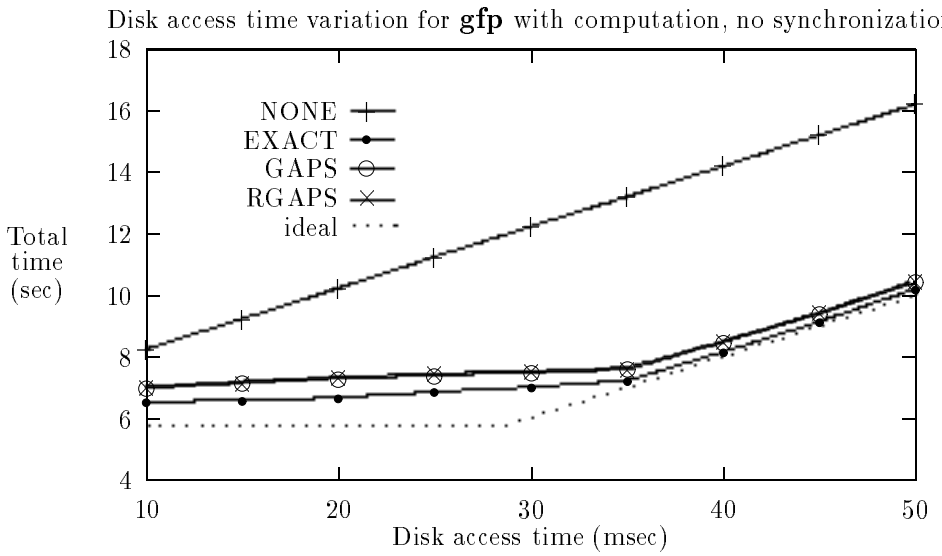


Figure 8.21: The result of varying disk access time for the **gfp** pattern with computation, no synchronization. ($cv < 0.013$)

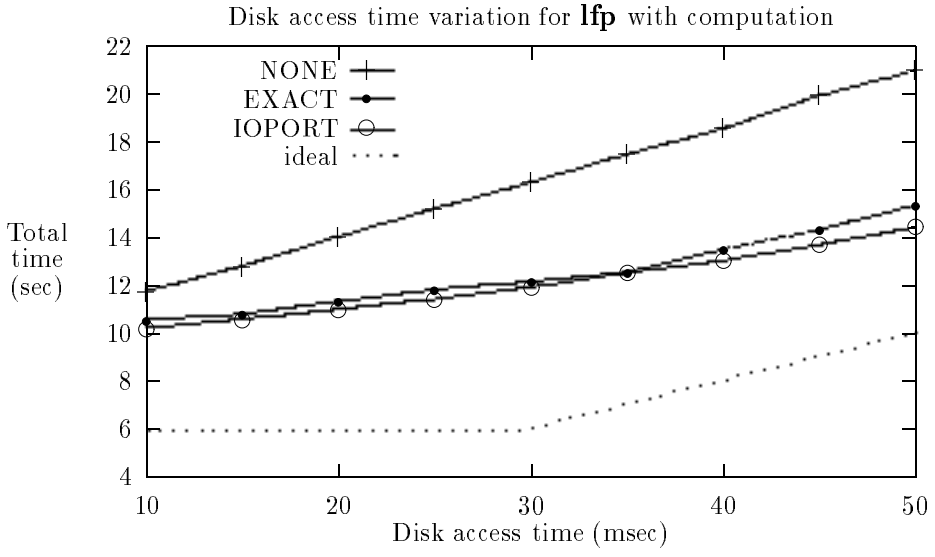


Figure 8.22: The result of varying disk access time for the **lfp** pattern with computation. ($cv < 0.030$)

(Figure 8.23). This indicates that the I/O never completely overlapped the computation time, as it did for the global patterns. For other synchronization styles (not shown) the curves were bent slightly, so there may have been a small amount of overlap.

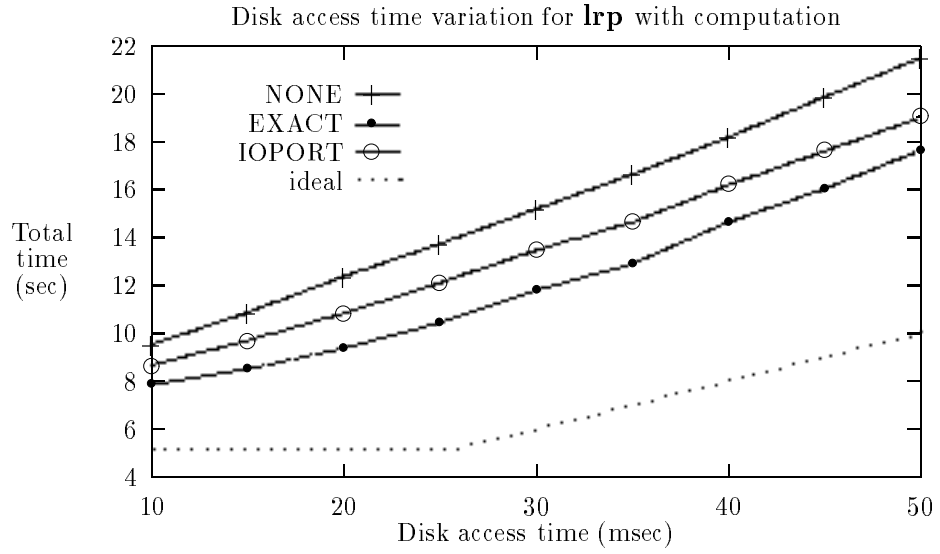


Figure 8.23: The result of varying disk access time for the **lrp** pattern with computation. ($cv < 0.034$)

EXACT and IOPORT were independent of the disk access time in the **lw** pattern (Figure 8.18). Nearly all I/O was overlapped with computation or other I/O.

The **seg** pattern results (not shown) were similar to **lfp** except for the greedy-process problem.

8.3.3 Conclusions

These are the primary conclusions:

- Slower disk-access times represent the likely architectural change to faster disks and much faster processors.
- The disk speed parameter had the obvious linear effect on the total execution time. Based on the slope of the line, it was clear whether the disk loads were balanced. Also, many I/O-bound global patterns had a total execution time that was equal to the ideal disk I/O time.
- In many cases the benefits of prefetching (in terms of an improvement over not prefetching) increased with the disk access time. Slower disks meant more idle time that could overlap with overhead or computation, which is one area where prefetching finds benefits. It also meant that the cost of a cache miss was increased, increasing the value of a prefetch relative to its cost.
- In addition to the obvious linear effect, in certain predictors the overhead was large enough to make prefetching unproductive for fast disks on I/O-bound patterns. Increased processor speeds should keep prefetching overhead ahead of the fast disks.

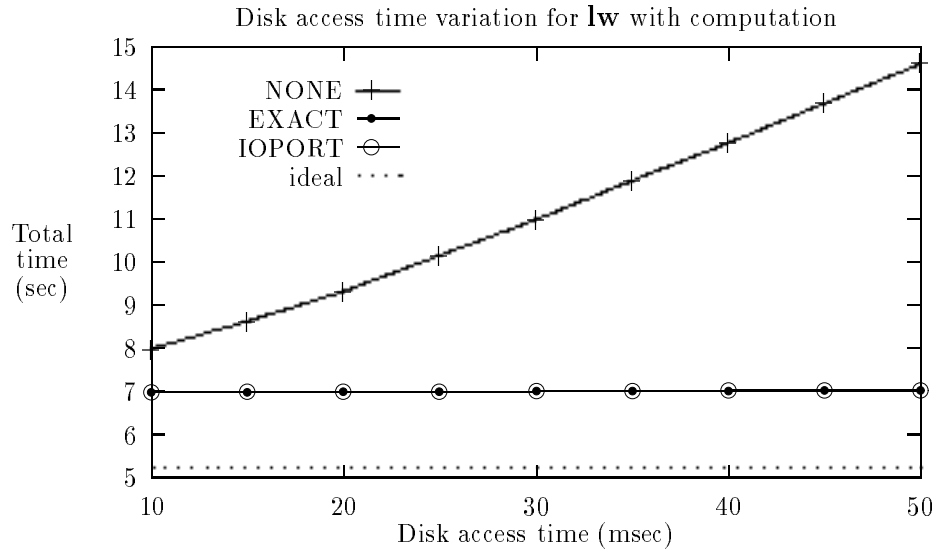


Figure 8.24: The result of varying disk access time for the **lw** pattern with computation. ($cv < 0.0031$)

- The **lw** pattern with prefetching was never affected by the disk speed since all I/O was overlapped with other I/O.
- For some compute-intensive global patterns and fast disks, most or all of the I/O was overlapped by computation. These patterns were insensitive to changes in disk speed within that range.

8.4 Varying the Number of Disks

In most of our other experiments we used 20 processors and 20 disks. In many real multiprocessors it is likely that there are fewer disks than processors. Also, many applications may be run with more or fewer processors than there are disks. To understand the importance of the number of disks, and how well prefetching can use all of the disks, we experimented with various numbers of disks.

8.4.1 Experiments and Results

We ran several experiments using 1 to 35 disks, using our usual set of patterns and predictors. We used only *none* synchronization, which simplifies analysis by removing synchronization effects. The number of processors was fixed at 20. As usual, there was one application process on each processor. In this section, we refer to *processors* instead of *processes*, to reinforce the fact that there is full physical parallelism available for the 20 processes. All of the other experimental parameters were the same as usual: 20 processors, 1-KByte blocks, 1-block records, 80-block cache, and five trials per test case.

The ideal execution time is derived as before:

$$T = \max(\text{I/O time}, \text{comp time})$$

There are 4000 block references in our patterns, each requiring 30 msec of I/O, for a total of 120 seconds (**lw** has 200 disk accesses, for a total of 6 seconds). The 120 seconds of I/O is spread over a variable number of disks, d. In the I/O-bound experiments there is no computation, so

$$T = \frac{120}{d} \text{ seconds}$$

In the experiments with computation, the total amount of computation (C) varied slightly since it was randomly generated, but it was roughly 100–120 seconds for all patterns. There were 20 processors in all of these tests, so the parallel computation time was (ideally) $C/20$ and hence

$$T = \max(\frac{120}{d}, \frac{C}{20}) \text{ seconds}$$

In all plots of these experiments, we show the "ideal" execution time as a dotted line, using the actual value for C (it may be somewhat obscured by the experimental-result curves). We first discuss the I/O-bound cases, then move on to the cases with computation.

I/O-bound Experiments

The results for the I/O-bound **gfp** pattern are shown in Figure 8.25. The overall impression is that all predictors, including NONE, roughly followed the ideal curve, running faster with increasing numbers of disks. Most of the speedup occurs for fewer than 10 disks, with diminishing returns after that.

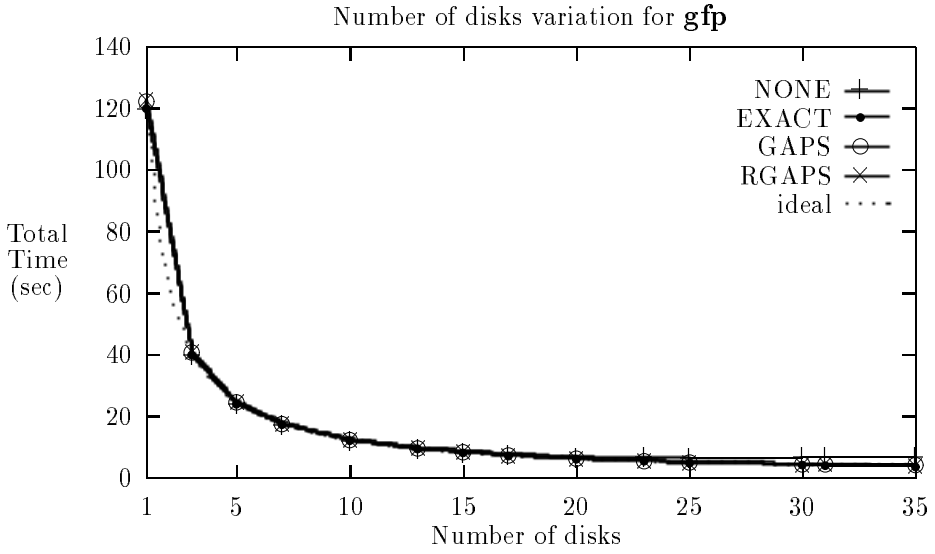


Figure 8.25: The result of varying the number of disks for the I/O-bound **gfp** pattern. ($cv < 0.036$)

There are a few interesting details, however. Note that the NONE predictor leveled off after 20 disks. With only 20 processors, and no prefetching, there was no way for NONE to use more

than 20 disks. NONE was faster than the others, however, for fewer than 5 disks. At this point, the overhead of prefetching was larger than the benefits; the extra parallelism provided by four or more times as many processors as disks was enough to keep all of the disks busy. This is a common feature of all the results in this section.

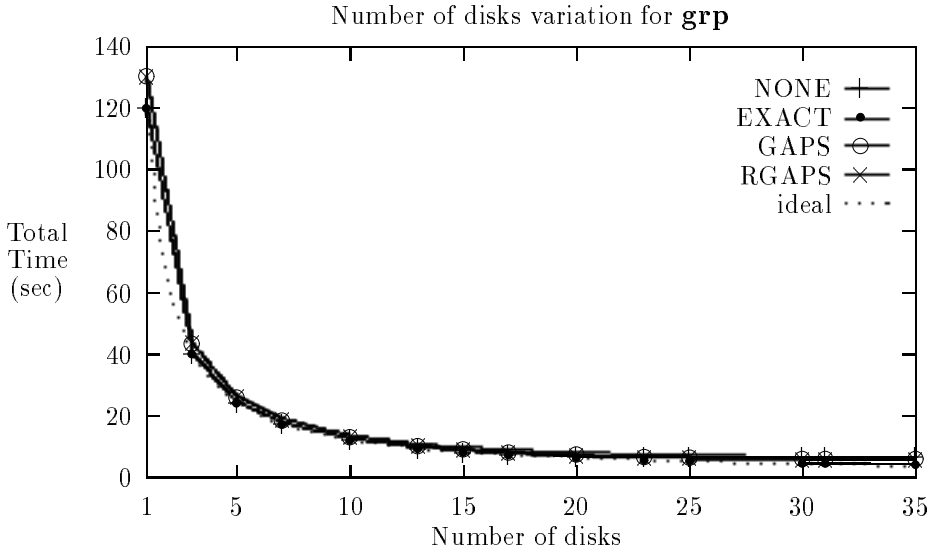


Figure 8.26: The result of varying the number of disks for the I/O-bound **grp** pattern. ($cv < 0.017$)

The overall shape of the curves for **grp**, shown in Figure 8.26, is the same as for **gfp**, but the details are a little different. Here NONE was faster than both GAPS and RGAPS for fewer than 20 disks. For more than 20 disks, however, GAPS and RGAPS were faster than NONE, which leveled off. GAPS and RGAPS were further from the ideal than they were in **gfp**, due to the increased mistakes and prediction overhead.

The results for **gw** are not shown. NONE leveled off above 20 disks, and all other predictors matched the ideal curve throughout.

The results for **lfp** (Figure 8.27) were remarkably similar to those for **gfp**. With fewer disks than processors, there were enough processors to keep the disks busy without prefetching, and NONE ran at ideal speed. The added overhead of prefetching, coupled with the high cost of mistakes, slowed down IOPORT. Then again, the shortage of disks limited the application so severely that the difference between NONE and IOPORT was insignificant. For greater than 20 disks, however, 20 processors could no longer keep the disks busy without prefetching, so NONE leveled off while IOPORT was essentially identical to both EXACT and the ideal curve. The **seg** pattern results (not shown) were essentially the same as in the **lfp** pattern.

In the **lrp** pattern (Figure 8.28), we again note that NONE was faster than IOPORT for small numbers of disks (here fewer than 20 disks), and IOPORT was faster than NONE for more than 20 disks. Unlike **lfp**, however, neither IOPORT nor EXACT were as fast as the ideal. This was due to the added mistakes, the inability to predict over portion skips, and IOPORT's conservative treatment of random portions.

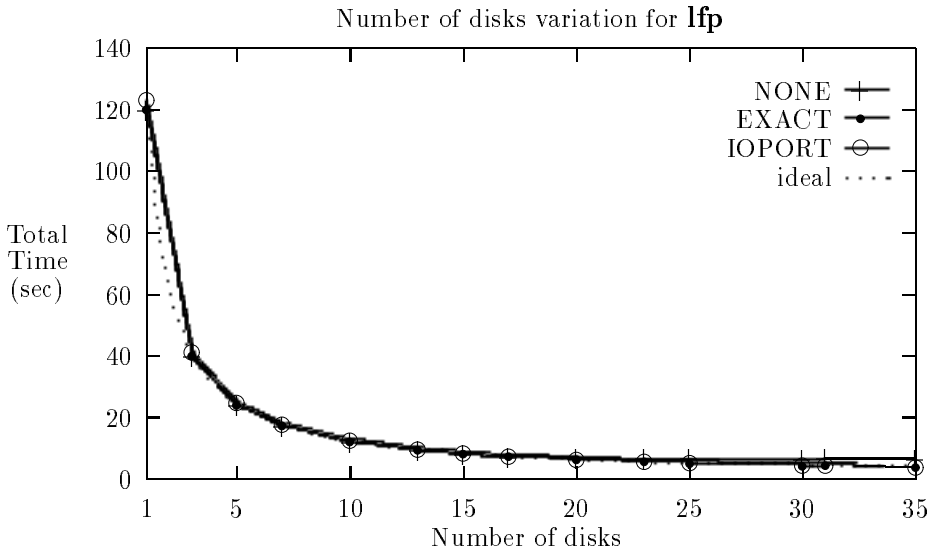


Figure 8.27: The result of varying the number of disks for the I/O-bound **lfp** pattern. ($cv < 0.027$)

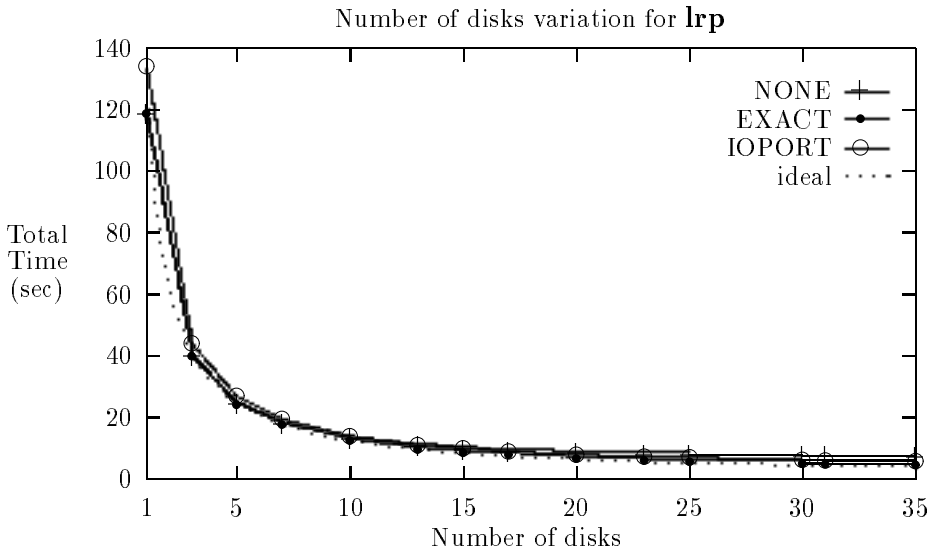


Figure 8.28: The result of varying the number of disks for the I/O-bound **lrp** pattern. ($cv < 0.021$)

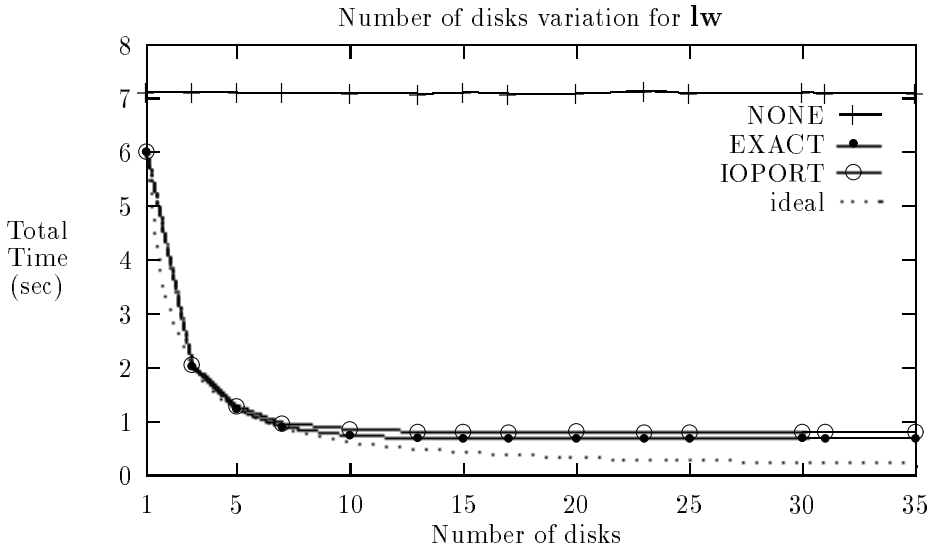


Figure 8.29: The result of varying the number of disks for the I/O-bound **lw** pattern. The ideal time is based on 200 disk accesses instead of 4000. ($cv < 0.024$)

The **lw** pattern results (Figure 8.29) were remarkably different than the others. Without prefetching, all processors in the **lw** pattern worked on only one block at a time, and hence only one disk at a time. Thus, the time for NONE was roughly constant for all numbers of disks. The other predictors leveled off for more than 7 disks. This phenomenon is related to our choice to do prefetching only during processor idle times. In **lw**, with prefetching and enough disks, the disks supplied blocks faster than they could be processed, reducing the hit-wait time to zero. There was then no idle time and hence no prefetching. Eventually the buffers were emptied and a demand fetch was necessary, which provided idle time for more prefetching. These events repeated in a cycle, causing several (time-consuming) demand fetches, and limiting the speed of **lw**. This was one case where it might have been productive to force a prefetch when there was no processor idle time available.

The **rnd** pattern is neither local nor global (Figure 8.30). This graph is a little different because it includes all of the predictors except EXACT (NONE, IOPORT, GAPS, and RGAPS). For few disks (i.e., 1–3 disks) the 20 processors were able to fully use all the disks and run close to the ideal speed. With more disks, the random access pattern caused short-term imbalances in the disk load that left some disks idle and thus the ideal was not reached. The disk contention was reduced when the pattern was spread over more disks, so performance gains were possible when there were more disks than processors.

Experiments with Computation

In our experiments with computation, the execution time had a lower bound of about 6 seconds, which was the total computation time spread evenly over 20 processors. This lower bound was achieved only if all of the I/O was overlapped with computation. This was possible when there were enough disks to make the I/O time shorter than the computation time, and if some mechanism was used to overlap the two. Since the I/O and computation times were roughly balanced for 20

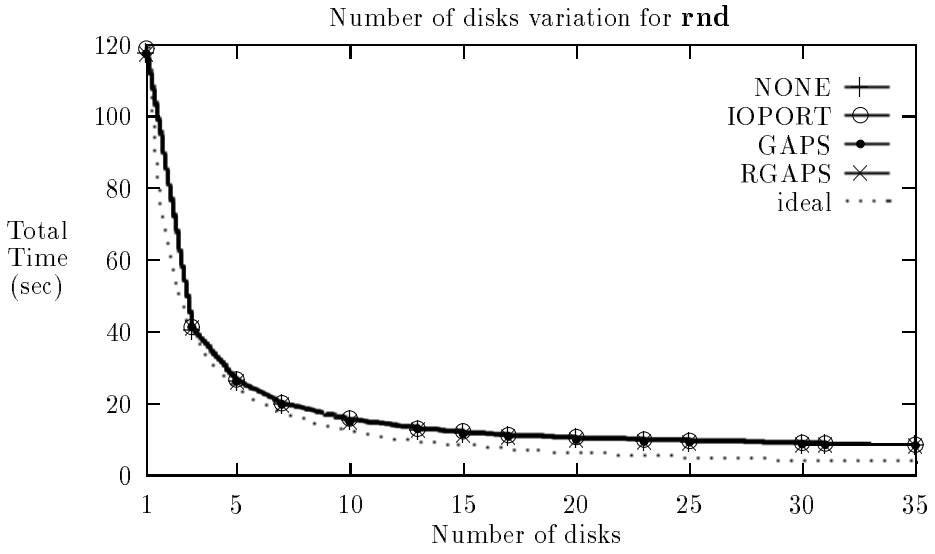


Figure 8.30: The result of varying the number of disks for the I/O-bound **rnd** pattern. Note that IOPORT, GAPS, and RGAPS are all included. ($cv < 0.020$)

disks and 20 processors, the ideal execution time leveled off for more than 20 disks. If there were more computation time, the ideal curve would level off with fewer disks. Thus, a program with more computation needs fewer disks, while a program that has less computation needs more disks, to attain its best performance. This comes directly from the equations on page 125. Indeed, applications with much more computation than I/O could not benefit from large numbers of disks (but could still benefit from prefetching). For example, if our experiments had roughly 120 seconds of computation per process (instead of 6), the ideal would be 120 seconds regardless of the number of disks.

Figure 8.31 shows the results for the **gfp** pattern with computation. Note that NONE leveled off above 10 disks. With 10 or more disks, on average half of the processors were using the disk and half were computing at any one time (since the average computation time is equal to the disk access time, per block). Hence, at most 10 disks (half of 20) could be used by NONE.[2] For 10 or fewer disks NONE was close to ideal speed, implying that it was able to overlap I/O and computation. This was because there were enough processors to keep all the disks busy, although some processors were computing.

The "ideal" curve leveled off at 20 disks, and the other predictors stayed close to ideal, leveling off around 17 or 20 disks. The difference between these predictors' times and the ideal time is the amount of I/O and overhead that could not be overlapped with computation, plus the extraneous I/O caused by prefetching mistakes.

The results for **grp** with computation (Figure 8.32) were similar to those for **gfp** with computation, except that the GAPS and RGAPS predictors were a little slower. This was typical for **grp**, of course, because of the greater number of mistakes and the difficult pattern recognition. This added overhead made them slower than NONE for fewer than 10 disks.

[2]If there had been more computation involved, NONE would have leveled off sooner, able to use fewer disks.

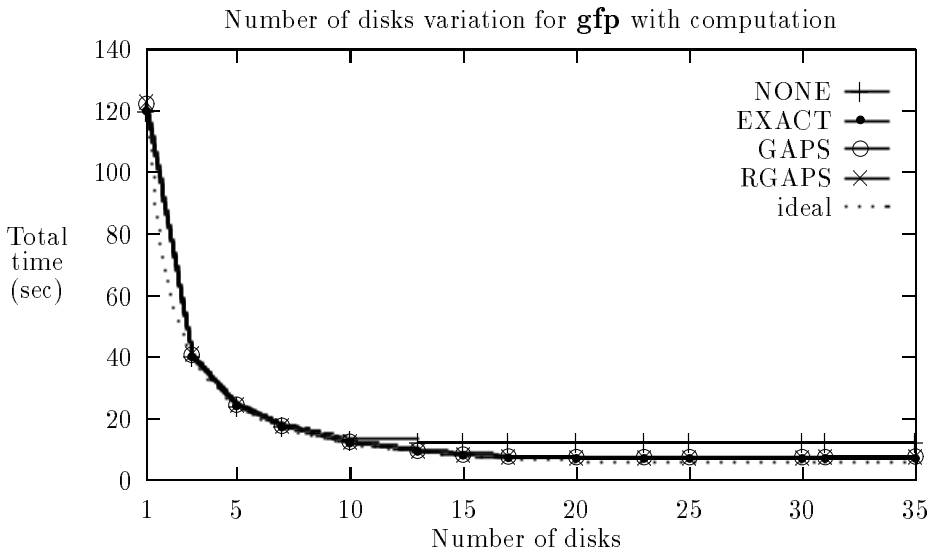


Figure 8.31: The result of varying the number of disks for the **gfp** pattern with computation. ($cv < 0.025$)

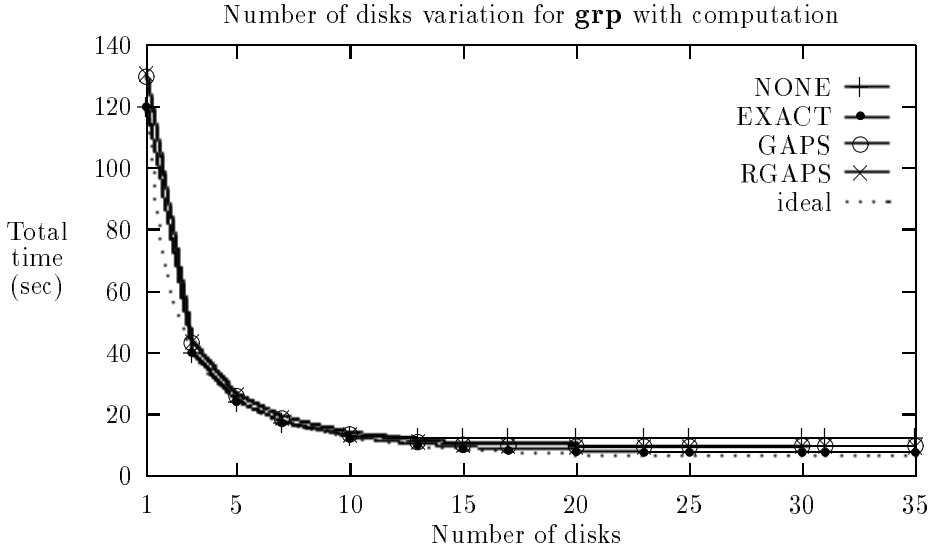


Figure 8.32: The result of varying the number of disks for the **grp** pattern with computation. ($cv < 0.024$)

The results for **gw** with computation (not shown) were similar to those of **gfp** with computation. GAPS, RGAPS, and EXACT were essentially identical in all cases, and all close to the ideal execution time.

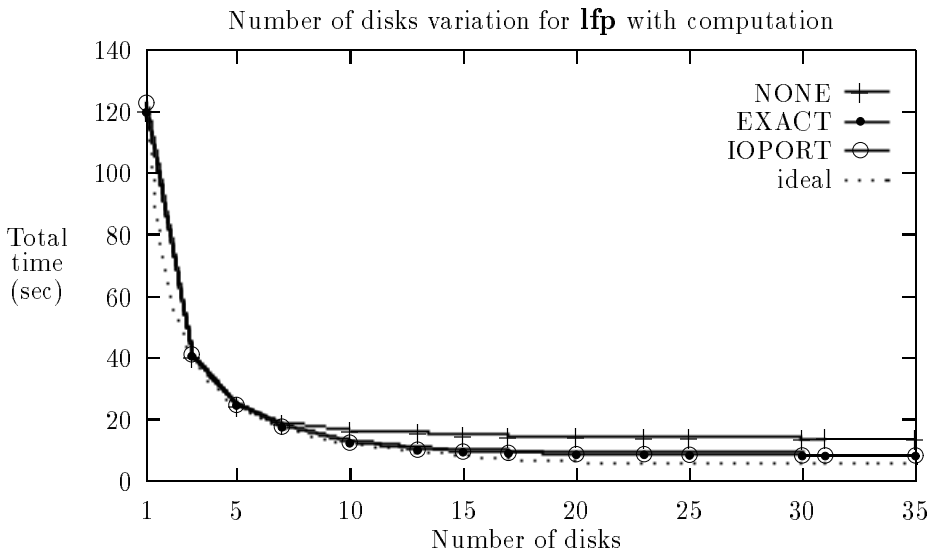


Figure 8.33: The result of varying the number of disks for the **lfp** pattern with computation. ($cv < 0.021$)

The **lfp** pattern with computation (Figure 8.33) was similar to the **gfp** pattern with computation. EXACT and IOPORT were essentially identical, though not quite ideal for more than a few disks. The results for the **seg** pattern with computation (not shown) were similar to those of the **lfp** pattern with computation.

The **lrp** pattern with computation, in Figure 8.34, performed much like the **grp** pattern with computation. For fewer than 10 disks, NONE was faster than IOPORT, and for 10 or more disks, IOPORT was faster than NONE. Due to the mistakes and IOPORT's reluctance to prefetch in **lrp**, IOPORT was slower than EXACT.

In the **lw** pattern with computation, as in the I/O-bound **lw** pattern, the NONE predictor had a roughly constant execution time (Figure 8.35). However, so did the EXACT and IOPORT predictors, which both came within a fixed overhead of the ideal execution time for more than one disk. Here the ideal curve was mostly constant (equal to the computation time), since the total I/O time was always less than or equal to the computation time (there were only 200 total disk accesses). In this pattern, therefore, there was no need for a large number of disks. The computation time dominated.

For the **rnd** pattern with computation (Figure 8.36) we again used all of the global and local predictors. There was no possibility of prefetching, so the predictors did not come close to the ideal time for more than a few disks. For more than 20 disks, the total execution time was the sum of the I/O and computation times, plus a little overhead. For fewer than 20 disks, the added processor parallelism allowed some overlap between computation and I/O. The rest was similar to the I/O-bound **rnd** pattern in Figure 8.30.

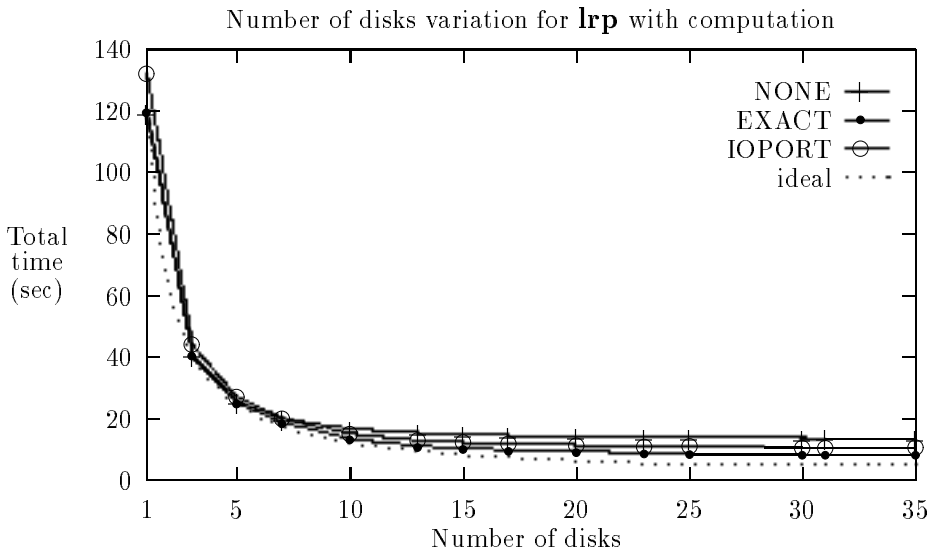


Figure 8.34: The result of varying the number of disks for the **lrp** pattern with computation. ($cv < 0.025$)

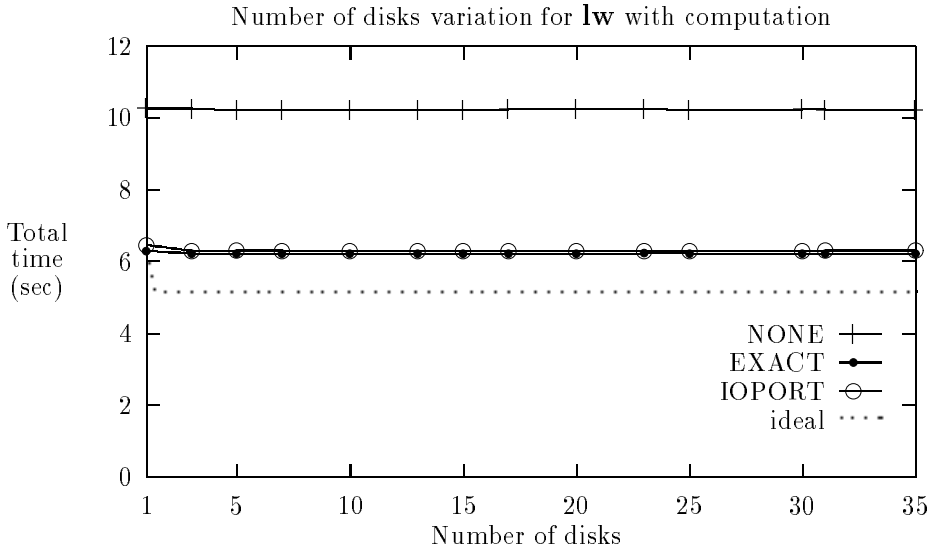


Figure 8.35: The result of varying the number of disks for the **lw** pattern with computation. The ideal time is based on 200 disk accesses instead of 4000. ($cv < 0.0094$)

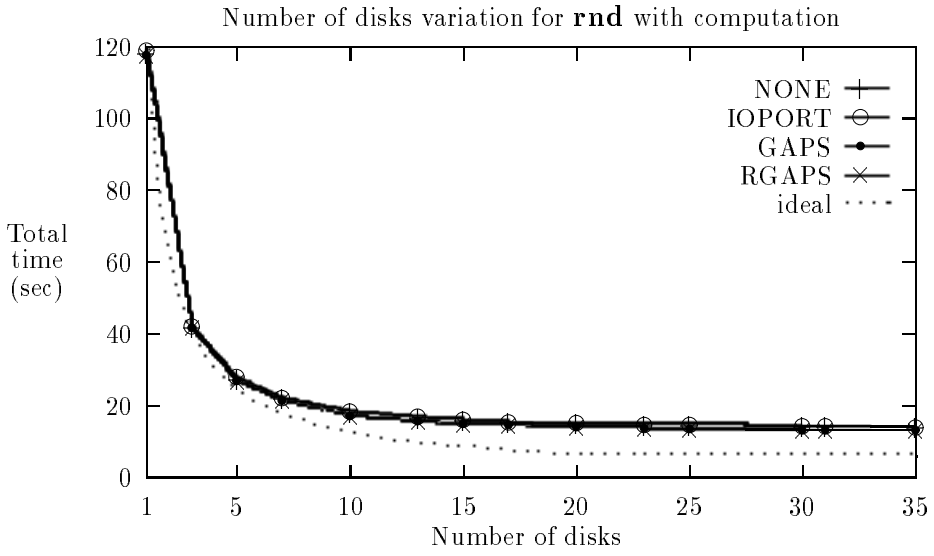


Figure 8.36: The result of varying the number of disks for the **rnd** pattern with computation. Note IOPORT, GAPS, and RGAPS are all included. ($cv < 0.029$)

8.4.2 Conclusions

These are the primary conclusions:

- Remember that achieving the ideal execution time, while a good measure of prefetching's effectiveness, is not necessarily the overall goal. It is little help to be close to the ideal when the ideal is slow, as is the case, for example, when there are only a few disks.
- The predictors all roughly followed the shape of the ideal-time curve, except for NONE, which could not use more than 20 disks with 20 processors (10 disks in the computation experiments).
- Only with prefetching did performance continue to improve when there were more disks than processors (except in the **rnd** pattern).
- NONE was often slightly faster than prefetching when there were many fewer disks than processors (except in **lw**). In this case, the parallelism in file system requests was able to keep the disks busy with less overhead than by prefetching. This seemingly negative result is tempered by the fact that, in these experiments, NONE was only slightly faster than prefetching, whereas in many other situations (fewer processors than disks, small record sizes, high disk contention, *etc.*) NONE was much slower than prefetching. Overall, prefetching was worthwhile.

8.5 Varying the Number of Processors

In all of our other experiments our test applications had 20 processes running on 20 processors. In most of those experiments, the number of disks was also 20. When we varied the number of disks,

we found several interesting effects when the number of processors did not equal the number of disks. We revisit these effects in this section, where we vary the number of processors, and we also examine the scalability of both the file system and the predictors.

As always, we run exactly one process on each processor, and so the number of processes equals the number of processors. In this section we often use the word "processor" to emphasize the variation in available *physical* parallelism.

8.5.1 Experiments

We used our standard set of patterns, with and without computation. All patterns (except **lw**) read 4000 blocks total (as always). Thus the local patterns had to be regenerated for each number of processors. We varied the number of processors from 1 to over 30.[3] This is enough processors to get a rough idea of scalability, though more processors are required to seriously test scalability of the implementation. We used the NONE and EXACT predictors, plus the appropriate on-line predictors (IOPORT or GAPS and RGAPS). We used only the *none* synchronization, which simplifies the analysis by avoiding synchronization effects. All other parameters were the same as usual (20 disks, 1-KByte blocks, 1-KByte records, 80-block cache, and five trials per test case). Wherever possible we used the same pattern, or pattern parameters, across all tests with that pattern. Due to the random nature of the **rnd** and **lrp** patterns, the pattern we used was necessarily different for each number of processors, but we believe that the difference had only a minor effect on these experiments.

In order to isolate the effect of the number of processors, we held most of the parameters constant. In particular, the cache size, and hence prefetch limit, was fixed. For local patterns, this meant that the prefetch limit per process varied with the number of processes. In a production system, it would be more logical to fix the cache size per process, and vary the total cache size with the number of processes.

8.5.2 Results and Discussion

The ideal execution time for these experiments is derived much like it is for the number-of-disks variation. Here, however, the number of disks is fixed at 20, and the number of processors (p) varies:

$$T = \max(\text{I/O time}, \text{comp time}) = \max(6, \frac{C}{p}) \text{ seconds.}$$

Here C is the total amount of computation in the pattern. In the I/O-bound experiments $C = 0$, and $T = 6$ seconds. In the global patterns the computation was randomly generated, and so C varied a little from pattern to pattern. It was usually around 100 seconds. In the local patterns we used a fixed 30 msec computation time on each block, to avoid difficulties in generating patterns that could be fairly compared as the number of processors varied. Here $C = 120$, except for **lw**, where $C = 6p$. The ideal curve is plotted in all of the figures, using the actual values for C.

I/O-bound Experiments

The results for the **gfp** pattern are shown in Figure 8.37. Adding processors reduced the execution time for the NONE predictor by helping to keep the disks busy. More than 20 processors (i.e., more processors than disks) made little difference. With prefetching, the ideal execution time was

[3] We stopped at 34 because that was the maximum useful size of our machine when the tests were run. In most local patterns we stopped at 32 because it was the largest number that evenly divided the work among the processors.

achieved with fewer processors. For EXACT, three processors were sufficient to keep 20 disks busy with prefetching. Thus, the disks were fully utilized either by prefetching or by using a sufficient number of processors.

There were two interesting effects when there were more processors than disks. First, NONE was faster than all other predictors (though not much faster). 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. This result was confirmed on similar experiments with 10 disks. Second, note that GAPS (and to some extent RGAPS) *slowed down* for more than 30 processors. The inter-process contention was increasing, a hint that these predictors may not scale well.

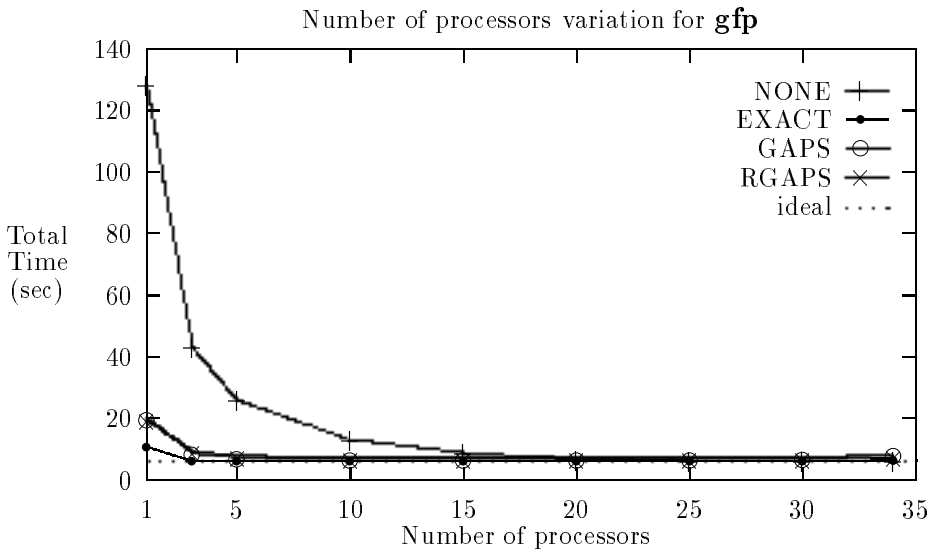


Figure 8.37: Total execution time for **gfp** as a function of number of processors. ($cv < 0.050$)

Less prefetching was possible in the **grp** pattern, shown in Figure 8.38. The NONE predictor behaved much as it did on **gfp**, but the three prefetching predictors needed more processors to match the ideal execution time than they did for **gfp**. NONE was fastest for more than 20 processors. This result was confirmed by a corresponding effect in experiments with 10 disks. GAPS began to slow down (due to predictor contention) for more than 25 processors.

The results for the **gw** pattern (not shown) were similar to those for **gfp**. The prefetchers had nearly ideal execution time except for few processors (fewer than three).

The results for the **lfp** pattern, shown in Figure 8.39, were similar to the **gfp** results. EXACT was always close to the ideal execution time, especially with more than 5 processors. IOPORT was also close to the ideal for more than 5 processors, although it began to slow down some when the number of processors exceeded the number of disks. In fact, for 32 processors IOPORT was slower than NONE. This was due to two factors: increased contention for prefetch buffers and other prefetching data structures, and an increasing number of prefetching mistakes. The prefetching mistakes occurred primarily at the end of the first portion on each processor: when the number of processors increased, the number of first portions increased, so the number of mistakes increased.

The results for **lrp** are shown in Figure 8.40. The general form of these results is similar to the **lfp** results, except that EXACT and IOPORT were able to prefetch less and thus needed more

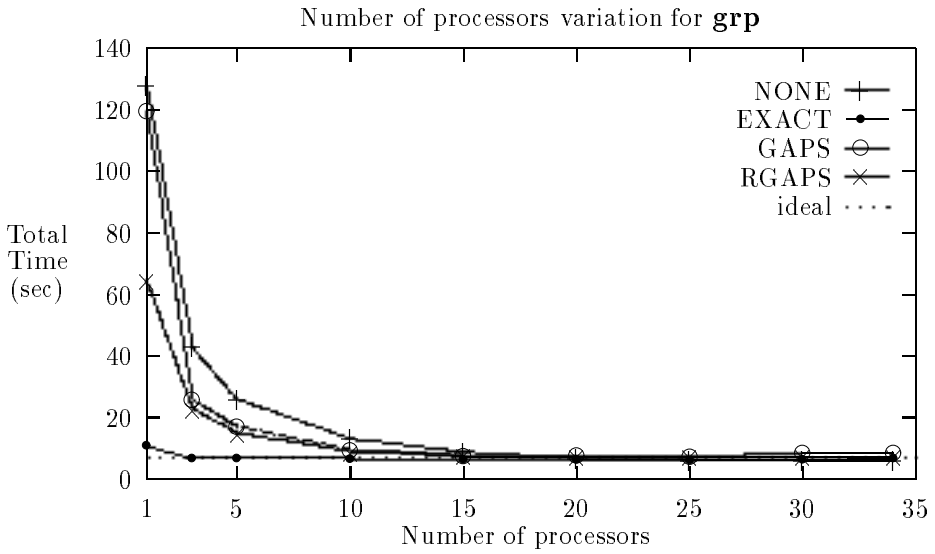


Figure 8.38: Total execution time for **grp** as a function of number of processors. ($cv < 0.070$)

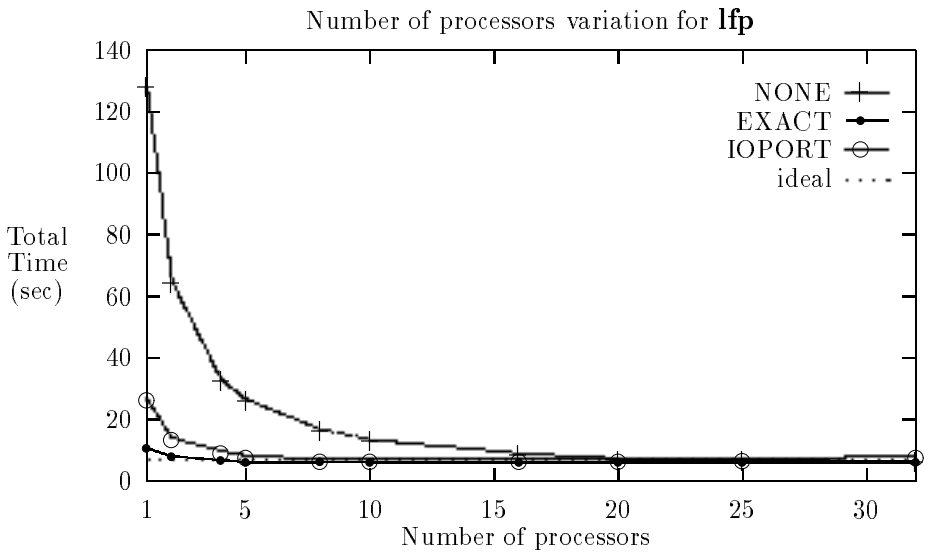


Figure 8.39: Total execution time for **lfp** as a function of number of processors. ($cv < 0.059$)

processors to approach the ideal execution time. Their approach was further from the ideal than in **lfp**, since the prefetching potential in **lrp** was more limited. NONE was faster than IOPORT for more processors than disks, a result confirmed in similar experiments with 10 disks. Here NONE was never faster than EXACT, but with more than three times as many processors as disks (32 processors, 10 disks) it was essentially the same as EXACT (not shown). None of the predictors showed any sign of slowing down for large numbers of processors.

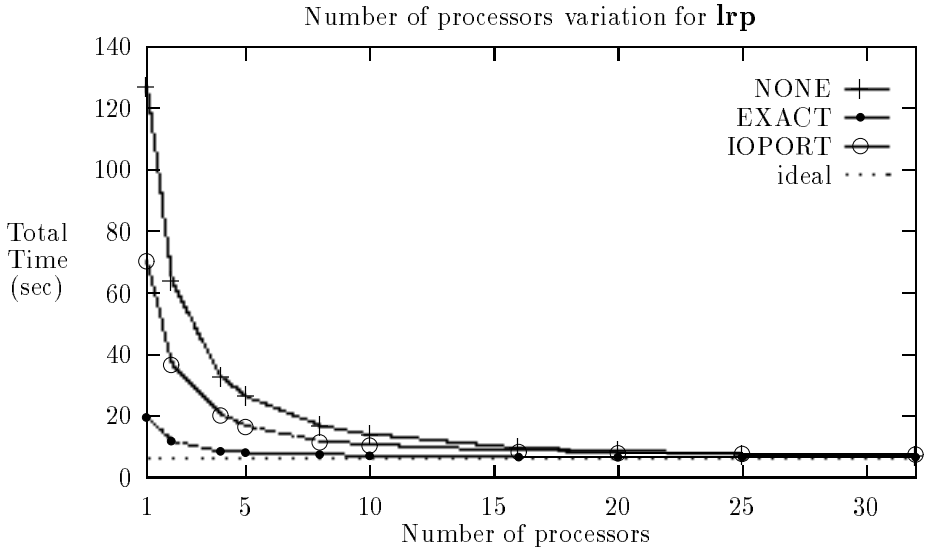


Figure 8.40: Total execution time for **lrp** as a function of number of processors. ($cv < 0.036$)

The **lw** pattern was different from the other patterns, as shown in Figure 8.41. In this pattern, every process read the whole file, rather than some share of the file. Thus, the number of file system requests, per process, was independent of the number of processors, and the total number of file system requests grew linearly with the number of processors. The amount of I/O was constant, as in all the other experiments. As Figure 8.41 shows, there was no improvement in the execution time by adding more processors. NONE could not use more than one disk at a time, regardless of the number of processors, whereas EXACT and IOPORT were able to use all of the disks with only one processor. Because of the increasing file system overhead, however, the execution time actually *increased* with more processors. The conclusions from similar experiments with 10 disks were the same.

Prefetching was successful in the **seg** access pattern, shown in Figure 8.42. Except for the one-processor case, and for more processors than disks, both EXACT and IOPORT ran at essentially ideal speed. IOPORT slowed down some for 32 processors, and NONE was then faster. Prefetching reordered the disk accesses and increased the disk contention. This had two effects: to reduce the number of blocks prefetched, and to decrease the load balance between the processors by delaying some processors more than others. With more processors, the load was less balanced, and the execution time increased.

For the **rnd** pattern, we necessarily used a different random pattern for each number of processors. Given the uniform distribution of block references throughout the file (and hence over the disks), we believe the differences had only small effects on the timing. We used the NONE, IO-

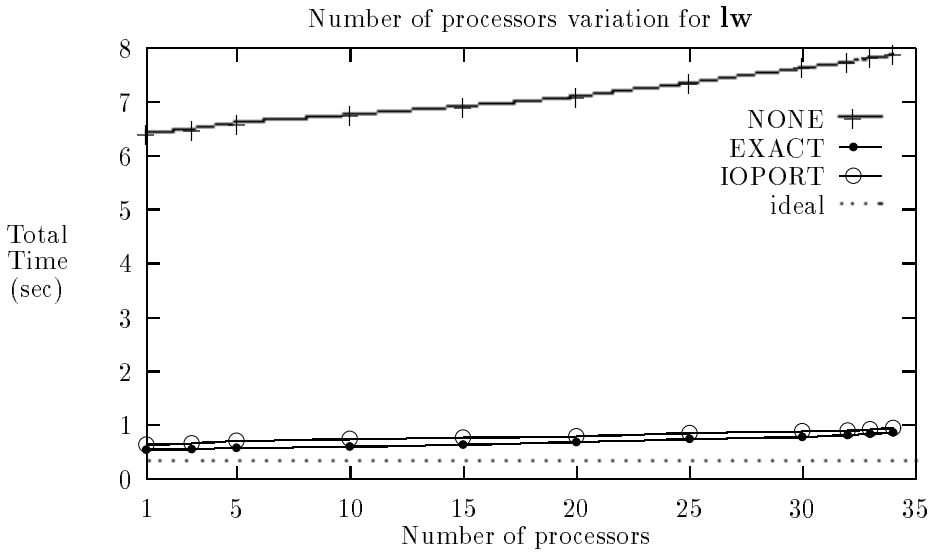


Figure 8.41: Total execution time for **lw** as a function of number of processors. ($cv < 0.025$)

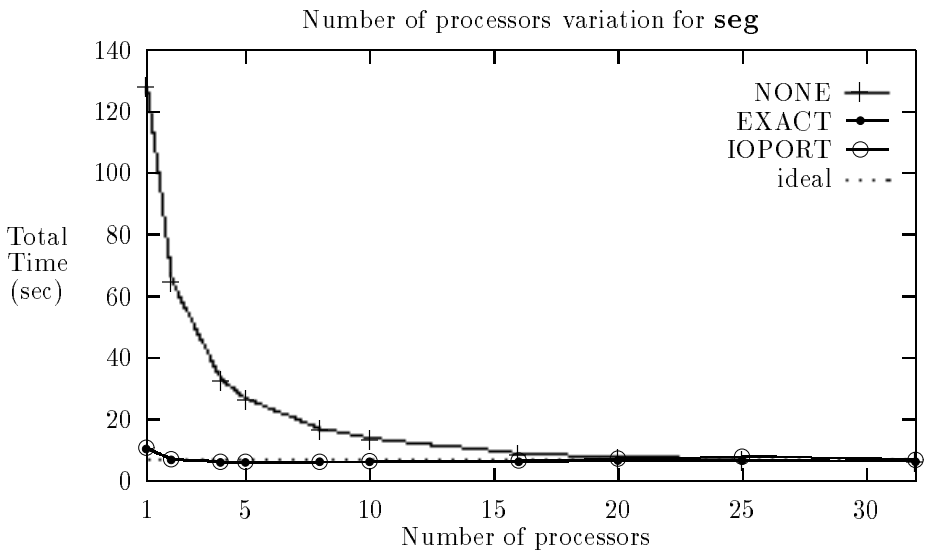


Figure 8.42: Total execution time for **seg** as a function of number of processors. ($cv < 0.076$)

PORT, GAPS, and RGAPS predictors. The results are shown in Figure 8.43. All of the predictors were essentially the same as NONE. None of them ever quite reached the ideal execution time, although they all continued to speed up (albeit slightly) past 20 processors. These results were confirmed in a similar experiment using 10 disks.

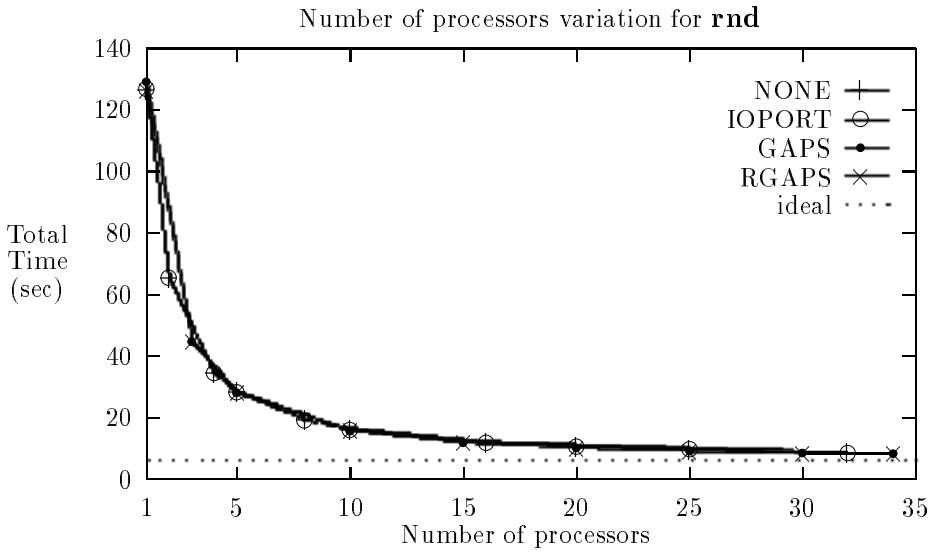


Figure 8.43: Total execution time for **rnd** as a function of number of processors. Note that not all predictors were run at all test points. ($cv < 0.021$)

Experiments with Computation

The ideal execution time for experiments with computation is attained by overlapping all computation with I/O, or all I/O with computation. Note that NONE could not use more disks than it had processors. Once there were more processors than disks, however, NONE queued multiple requests on some disks. It was possible, therefore, for all the disks to be busy while some processors were computing. Thus, the execution time for NONE could be less than the sum of the I/O time and the computation time, and approach the ideal execution time, when there were more processors than disks. This also held for all predictors in the **rnd** pattern, where none of them did any prefetching.

In the **gfp** pattern with computation (Figure 8.44), the prefetchers all came close to the ideal execution time, especially once they were I/O-bound (more than 20 processors). Thus, the overlap between computation and I/O was nearly perfect. As the number of processors increased, NONE was closer to the ideal execution time, though it was always slower than the others. In the 10-disk experiments (not shown), NONE was faster than GAPS and RGAPS for 25 or more processors, so we believe that NONE would also be faster than GAPS and RGAPS with 20 disks, given enough processors.

The results for **gw** with computation (not shown) were similar to those for **gfp** with computation.

The results for **grp** with computation are shown in Figure 8.45. All of the predictors' curves follow the same rough shape, with EXACT running at close to ideal speed throughout. GAPS

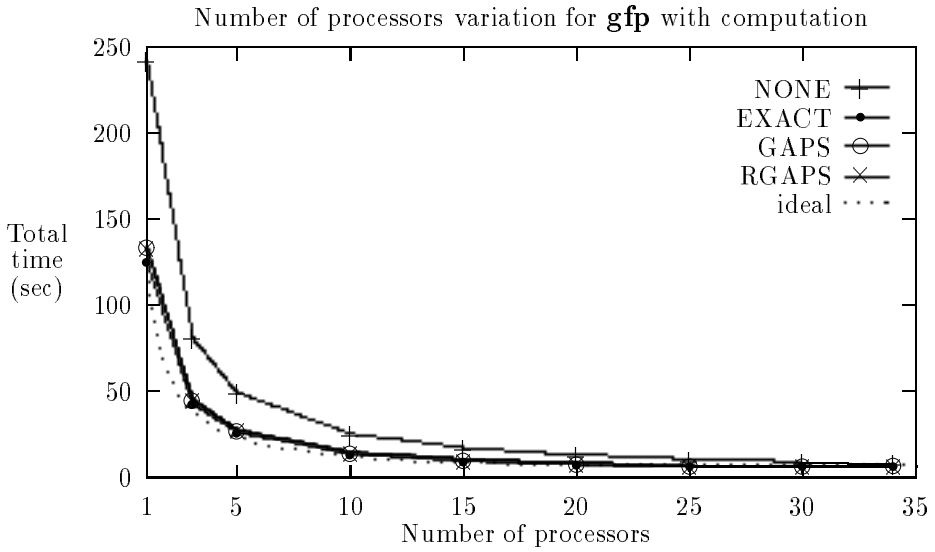


Figure 8.44: Total execution time for **gfp** with computation as a function of number of processors. ($cv < 0.053$)

and RGAPS were slower than EXACT, but still faster than NONE in most cases. For more than 30 processors, NONE was slightly faster than either GAPS or RGAPS, which began to level off about 1–2 seconds slower than ideal (due to the inevitable mistakes). Once again, a surplus of processors was able to provide the same benefits as prefetching, without the added overhead or costly mistakes. GAPS and RGAPS were roughly the same, except for one processor and more than 25 processors, where RGAPS was faster. Overhead slowed GAPS down.

The results for **lfp** with computation, in Figure 8.46, were similar to those for **gfp** with computation. EXACT was fairly close to the ideal execution time, and IOPORT was slightly slower for fewer than 20 processors. Both were always an improvement over NONE.

In the **lrp** pattern with computation, shown in Figure 8.47, IOPORT was never quite as fast as EXACT, which in turn was never quite as fast as the ideal. Both were always an improvement over NONE.

The results for the **lw** pattern with computation are shown in Figure 8.48, and seem to be independent of the number of processors. Without prefetching, only one disk was used at a time, so the I/O time was 6 seconds instead of 0.3 seconds. Computation (which for **lw** was independent of p) was not overlapped with I/O, so the total execution time was about 12 seconds. With prefetching, all computation was overlapped by I/O, so the total was closer to the ideal 6 seconds.

In the **seg** pattern with computation, IOPORT was essentially the same as EXACT, and both were close to the ideal. They were both always faster than NONE.

The **rnd** pattern with computation (Figure 8.50) was much like the I/O-bound **rnd** pattern, in that all of the predictors were about the same, and none ever matched the ideal execution time. With more processors, however, they were increasingly close to the ideal.

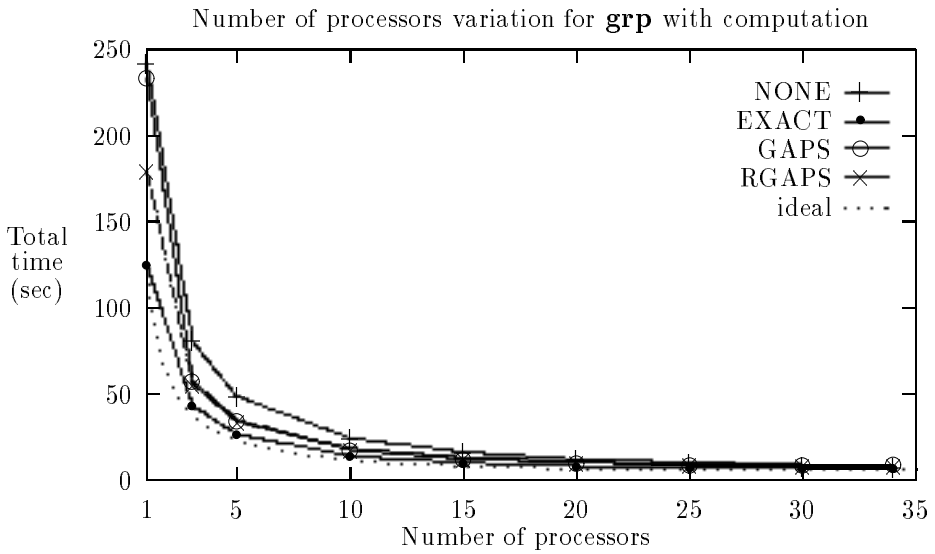


Figure 8.45: Total execution time for **grp** with computation as a function of number of processors. ($cv < 0.054$)

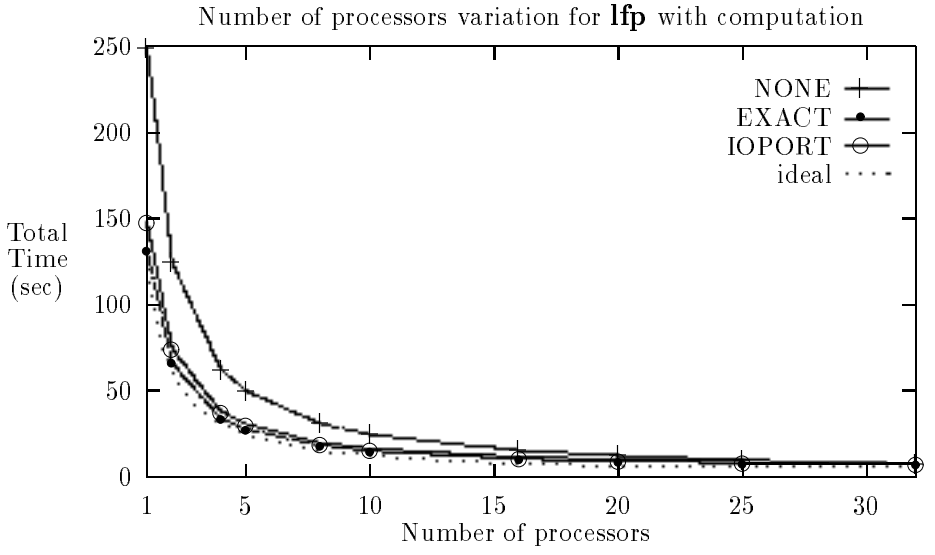


Figure 8.46: Total execution time for **lfp** with computation as a function of number of processors. ($cv < 0.016$)

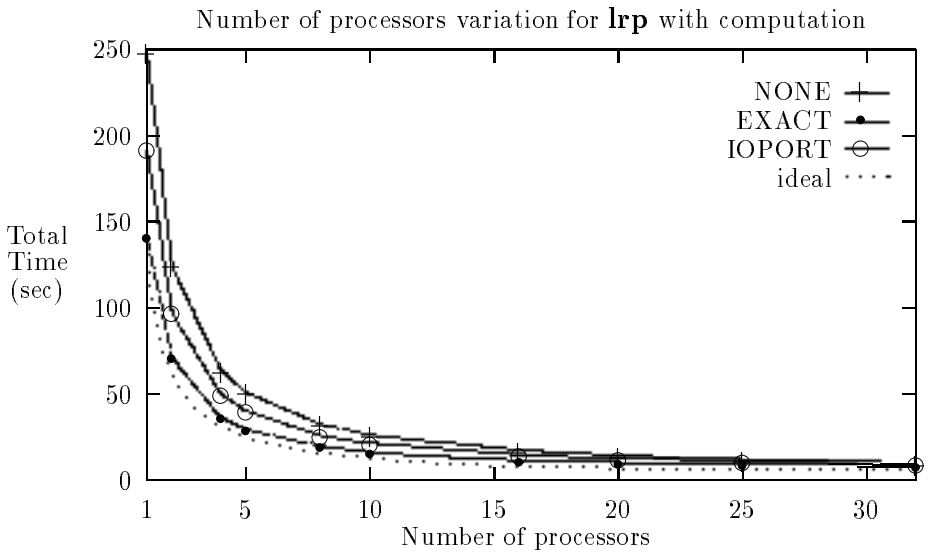


Figure 8.47: Total execution time for **lrp** with computation as a function of number of processors. ($cv < 0.037$)

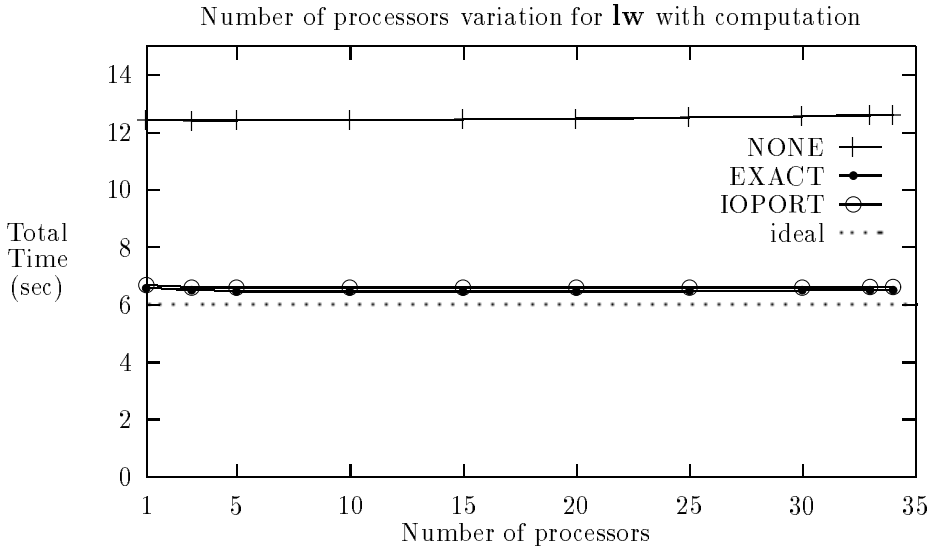


Figure 8.48: Total execution time for **lw** with computation as a function of number of processors. ($cv < 0.0023$)

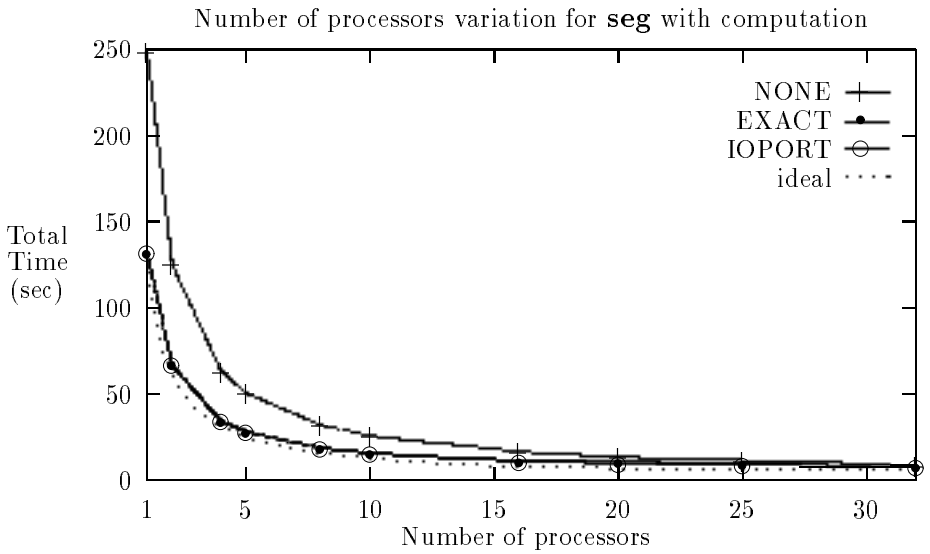


Figure 8.49: Total execution time for **seg** with computation as a function of number of processors. ($cv < 0.037$)

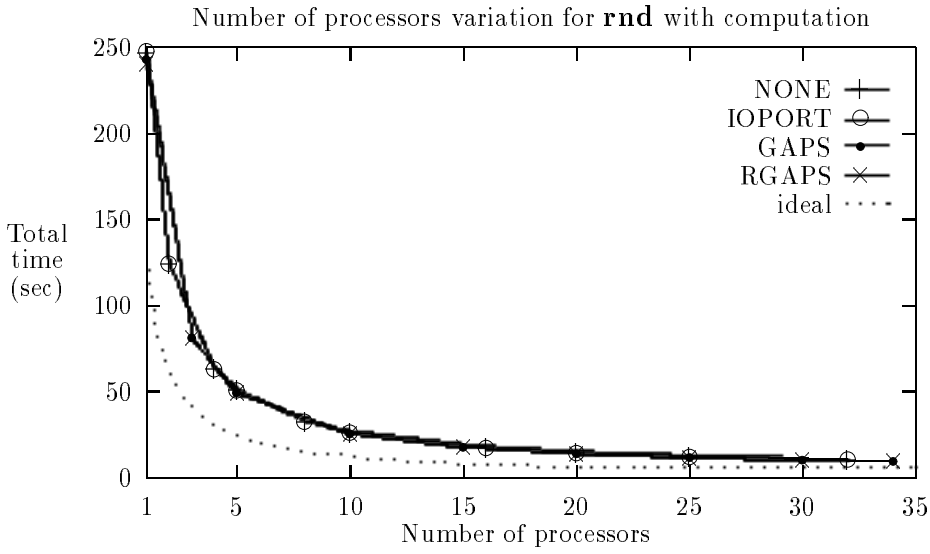


Figure 8.50: Total execution time for **rnd** with computation as a function of number of processors. ($cv < 0.0097$)

8.5.3 Scaling both Disks and Processors

In another set of experiments, we varied both the number of disks and processors so that there was always one disk per processor. The results from these experiments were similar to the experiments described above, except that the theoretical and experimental total execution time never leveled off within our test range (1 to 34 processors), as was the case when we varied only the number of disks or the number of processors. The ideal execution time for p processors and $d = p$ disks is

$$T = \max(\text{I/O time}, \text{comp time}) = \max(\frac{IO}{d}, \frac{C}{p}) = \frac{\max(IO, C)}{p}.$$

Our experimental data followed the ideal curve in much the same way as described above, where we varied the number of disks or processors independently.

8.5.4 Conclusions

These are the significant conclusions:

- The ideal execution time was often achieved without prefetching, given a surplus of processors (except in **lw**). In I/O-bound applications, it sufficed to have more processors than disks. With some computation, it was necessary to have many more processors than disks.
- Using prefetching, fewer processors were required to reach the ideal execution time.
- It was possible to overlap computation and I/O almost perfectly.
- In some cases, the predictors ran more slowly when given more processors; in particular, it appears that the GAPS predictor had trouble scaling to many more processors than disks. We expect that GAPS would not scale well if it had more processors available.

The first conclusion is perhaps the most important. Since we expect that most multiprocessors will (and do) have more processors than disks, this is a common situation. Remember, however, that in other situations (e.g., small record sizes or unbalanced disk loads) prefetching was much better than not prefetching. In general, it is a valuable addition with significant benefits most of the time, and occasionally a small slowdown.

8.6 Overall Conclusions

The performance of our GAPS, RGAPS, and IOPORT predictors varied a lot in some of the experiments discussed in this chapter. Their performance relative to each other, and to EXACT, NONE, and the ideal execution time, also varied. In an effort to combine all of these results, and contrast them with the results of other chapters, we make the following conclusions:

- Disk utilization was obviously the key to performance, particularly in the I/O-bound experiments. Prefetching usually increased disk utilization and hence performance.
- All three predictors, GAPS, RGAPS, and IOPORT, were fairly robust across most of the experiments in this chapter. Within limits, the number of disks, number of processors, and disk access time could vary and prefetching came close to the ideal execution time, and was usually faster than not prefetching. There were exceptions, usually at the extremes of the variation (e.g., few disks, many processors, or fast disks).
- RGAPS is clearly better than GAPS as a general-purpose choice. RGAPS was able to handle unusual record sizes, where GAPS was sometimes 10 times slower than NONE. Because of its lower overhead and higher concurrency, we expect RGAPS to scale better than GAPS when given more processors, judging by the effects we saw beginning around 30 processors.
- When there were fewer processors than disks, prefetching was better able to keep the disks busy than was NONE. In this case, the execution time with prefetching was often close to the ideal execution time.
- When there were more processors than disks, the NONE predictor was often faster than all of the others. The parallelism alone was able to keep the disks busy, with less overhead and no mistakes. When there was computation involved, more processors were required for NONE to be faster than the other predictors, since it could not overlap as much I/O with computation.
- Prefetching helped to overlap computation and I/O, and in many cases the execution time was close to the ideal. The execution time was, of course, bounded below by the computation time, since prefetching could only improve the I/O time.
- Limited prefetching was possible in the **rnd** pattern when the record size was not one block. Thus, some execution time improvements are possible even for "random" access patterns.
- With fast disks, prefetching was sometimes slower than not prefetching. In this case, the overhead of prefetching was not worth the small benefits. Increasing processor speeds will lower this overhead faster than disk speeds will improve, so this effect may not be significant.
- Prefetching in many local patterns was much improved with larger caches (on the order of 200 blocks instead of 80 blocks).

Although there were some situations where prefetching was not helpful, it was not much worse than not prefetching (with the exception of GAPS on unusual record sizes). Prefetching was most useful in these situations:

- When there were fewer processors than disks.
- When there was some computation to be performed.
- When the disk-access speed was slow relative to processor speed.

- In the **lw** pattern.

In addition, successful prefetching required more buffers than were needed without prefetching. In general, prefetching gave huge performance increases and sometimes small performance decreases. Thus, on the whole, prefetching was effective.

The results in previous chapters were based on only one assignment of the parameters we explore in this chapter. We cannot say whether those results were consistently optimistic or pessimistic, either in terms of the potential for prefetching or the performance of any given predictor. From the results in this chapter:

- With larger caches, IOPORT and EXACT performed much better on some local patterns.
- With fewer processors, or more disks, the benefits of prefetching were more significant.
- With slower disks the benefits of prefetching were more significant.
- The all-round performance of GAPS was much worse than in Chapter 7.

Thus our previous results were neither pessimistic nor optimistic, except in the case of the GAPS predictor, where they were overly optimistic.

Our experiments were limited by the size of the machine available to us. We believe, however, that prefetching will scale to larger machines. 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, depending on the particular access patterns and computational loads that are expected. To scale to much larger machines (hundreds of processors) the quality of the implementation becomes even more important. Our experiments only began to test the scalability of the file system and predictors, all of which showed some stress even with 30 processors. IOPORT looks like it will scale well to larger numbers of processors and disks, and RGAPS may scale some more, but will be more limited. GAPS, on the other hand, is not likely to scale much further at all. The file system itself has moderate scalability; a more careful implementation is needed for significant scalability.

Chapter 9

Buffering for Write Access

9.1 Introduction

In the preceding chapters we concentrate on read-only access patterns, investigating the potential for prefetching and caching techniques to improve parallel I/O performance. In this chapter we consider another important class: write-only access patterns. Recall from page 14 that these patterns are writing newly created files, not overwriting existing files. The issue, then, is not prefetching, since there is no data to fetch from disk, but buffering data written to the cache and deciding when to write it back to disk. The timing of disk writes can have significant performance effects. As before, the goal is high disk utilization and minimization of mistakes. We explore several methods in this chapter, and evaluate their performance on our write-only access patterns (page 29).

We assume that the space for the file is preallocated, so no file-extension or other overhead is required while the file is written. This assumption is necessary since we do not model the disk layout or simulate the associated file system overhead involved with opening files and allocating disk space. Preallocation is already found in some supercomputer file systems [Pow77, NNI89].

9.2 Methods

All I/O is done to buffers in the disk cache. The application writes data into a cache buffer (which is then "dirty"), and the data are written to disk later. Cache consistency is not an issue because there is only one, *shared* cache. We implemented several distinct methods for triggering the physical disk writes:

WriteThru, the simplest scheme, forces a disk write on every file write request from the application.

WriteBack delays the disk write until the buffer is needed for another block.

WriteFree issues a disk write when the buffer enters the free list. Thus, it issues a write before the buffer is needed for re-use, but after it is no longer in use by some processor. This is a compromise between WriteThru and WriteBack.

WriteFull issues the disk write when the buffer is "full," defined to be when the number of bytes written to the buffer is exactly equal to the size of the buffer in bytes. This assumes that each byte of the file is written exactly once (page 16).

Each of these methods is easy to implement, and has its own advantages and disadvantages. WriteThru, for example, is ideal for blocks that are only accessed once, because the disk I/O is started immediately. It is poor, however, for patterns where the block may be accessed many times in a short interval (e.g., when the record size is smaller than the block size; there are several records per block and thus several accesses per block). WriteFull was designed for this case. WriteBack and WriteFree provide interesting alternatives.

There are two types of mistakes possible in any write-only access pattern with non-integral record sizes:

rewrite: A disk write is issued prematurely. The application writes to a buffer, the buffer is written to disk, and then the application writes to the buffer again, requiring another disk write. Several rewrite mistakes are possible before a buffer finally leaves the cache. Each rewrite mistake represents one extraneous disk write.

reread: A block is removed from the buffer cache prematurely. In this case, the application writes a buffer and then the buffer is written to disk and used for another block. If the application writes to part of the first block again, the block must be read back into the cache before it can be updated. Each reread mistake represents two extraneous disk operations (one premature write, one reread).

9.3 Experiments

We designed a set of experiments to evaluate the effectiveness of our write-buffering policies across variations in workload and cache size. These experiments answer the following questions:

- What is the effect of cache size? Is a large cache useful?
- How do the policies react to the record size?
- Which (if any) policy is the most generally successful?
- Can a cache using a smart write-buffering policy help an application to better use the available parallel I/O bandwidth?

We experimented with all three write-only access patterns (page 29), both with computation (averaging 30 msec per block) and without (i.e., I/O bound). For each case, we tried all of the write methods described above, first varying the cache size with one-block records, then varying the record size with an 80-block cache. In all tests there were 20 processes and 20 disks, the RU-set size was one block, the block size was 1 KByte, and there was no synchronization (i.e., *none* synchronization). The ideal execution time was 6 seconds for all cases except **lw1** with computation, which was limited by its 120 seconds of computation.

In one final experiment, we varied the RU-set size using the WriteFree method (Section 9.4.3).

9.4 Results

All results represent the average over five trials for each test point. In all cases $cv < 0.065$, that is, the standard deviation was never more than 6.5% of the mean. Most cases had much smaller cv.

9.4.1 Cache-size Variation

In these experiments, the cache size varied from 20 one-block buffers to 200 one-block buffers (1 to 10 blocks per process). The workload patterns all issued write requests of exactly one block (i.e., the record size was one block). Thus, each block was accessed only once. The most successful methods issued disk writes soon after the buffer was filled. Note that WriteFull and WriteThru are inherently equivalent in these access patterns, because the buffer is full when it is first written.

In the **gw** pattern, shown in Figure 9.1, WriteThru, WriteFull, and WriteFree were clearly faster than WriteBack, which delayed the disk write too long. In **gw** with computation, shown in Figure 9.2, WriteFree is also slower than WriteThru or WriteFull. This is because WriteFree delays the disk write for a full but most-recently-used block until the next file system access, which is after the process's compute cycle. This delay was too long, slowing down overall execution. Note that 60 or more buffers for **gw**, and 40 or more buffers for **gw** with computation, were sufficient to run these patterns close to the ideal execution time of 6 seconds. Forty buffers corresponds to two buffers per process, which allowed one to be filled while the other is written to disk. More buffers (60) were required in the I/O-bound **gw**, because it was issuing write requests at a faster rate.

The slight rise in the time for **gw** with computation (Figure 9.2) for caches larger than 60 buffers was due to increasing overhead: unready-queue scans took longer when there were more buffers. With computation, this pattern was barely keeping the 20 disks busy, so a slight increase in overhead slowed it down and caused a lower disk utilization. The lower utilization led to a longer overall execution time. The **gw** pattern with no computation had no trouble utilizing the disks even with increased overhead, and so was not affected (Figure 9.1).

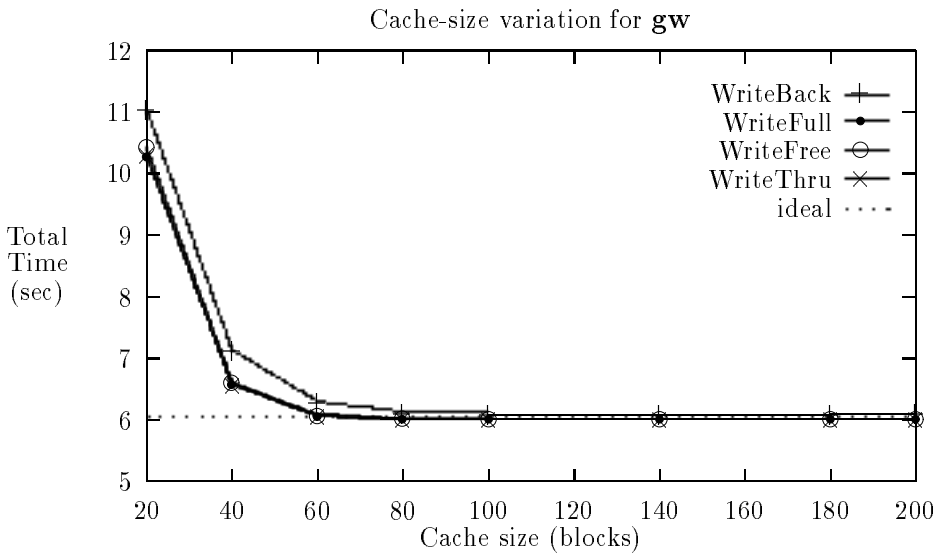


Figure 9.1: Cache-size variation for write pattern **gw**.

The **lw1** patterns ran more slowly than the **gw** patterns, because one process could not drive all 20 disks at full efficiency (Figures 9.3–9.4). WriteBack was much worse than the other methods, and WriteFree again was slow for **lw1** with computation. Larger caches benefited the **lw1** pattern by allowing more disk parallelism to be used, but this effect was not a factor when computation clearly dominated, as in Figure 9.4. Note that **lw1** with computation is compute-bound, ideally having

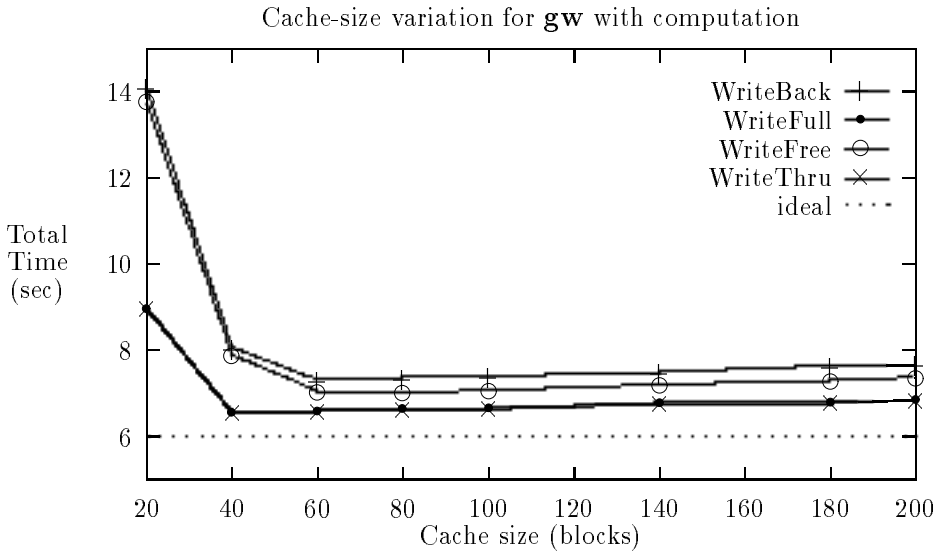


Figure 9.2: Cache-size variation for write pattern **gw** with computation.

120 seconds of computation and 6 seconds of disk time. WriteThru was the fastest in both cases, barely faster than WriteFull, presumably by having slightly less overhead in the implementation.

The write-only **seg** patterns, like their read-only counterparts, had a difficult disk access pattern (all processes began on the same disk). A large cache helped to alleviate this problem, as seen in Figure 9.5 and Figure 9.6, since the larger cache allowed processes to continue writing even when some disks were overloaded. In effect, large caches allowed a longer pipeline to form, using more disks concurrently than with a short pipeline. This is especially important as processor speeds increase relative to disk speeds. A 200-block cache allows 10 outstanding disk writes for each of 20 processes.

Summary. From these results, both WriteThru and WriteFull (essentially equivalent here) appear to be good write-buffering methods, in that they had the best overall performance. Note that WriteThru with a 20-block cache and a one-block record size is conceptually similar to not caching at all, and had poor performance. Our results show that large caches *can* use more disk parallelism and improve overall performance. The *necessary* cache size, however, depends on the workload: a larger cache is needed to absorb disk contention problems (as in **seg**) or a high write request rate (as in **gw** without computation). For the experiments in the next section we chose an 80-block cache because that was a reasonable compromise for all workloads, based on the results in this section.

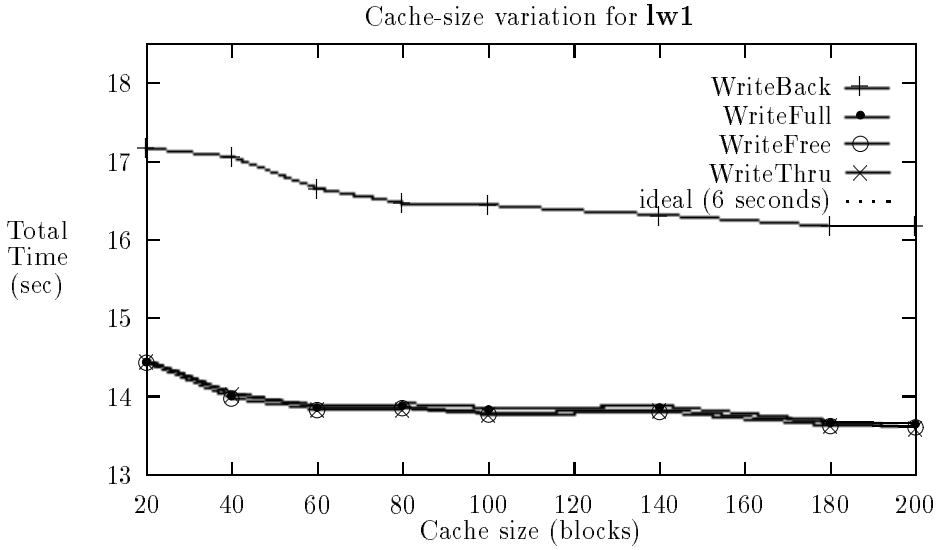


Figure 9.3: Cache-size variation for write pattern **lw1**.

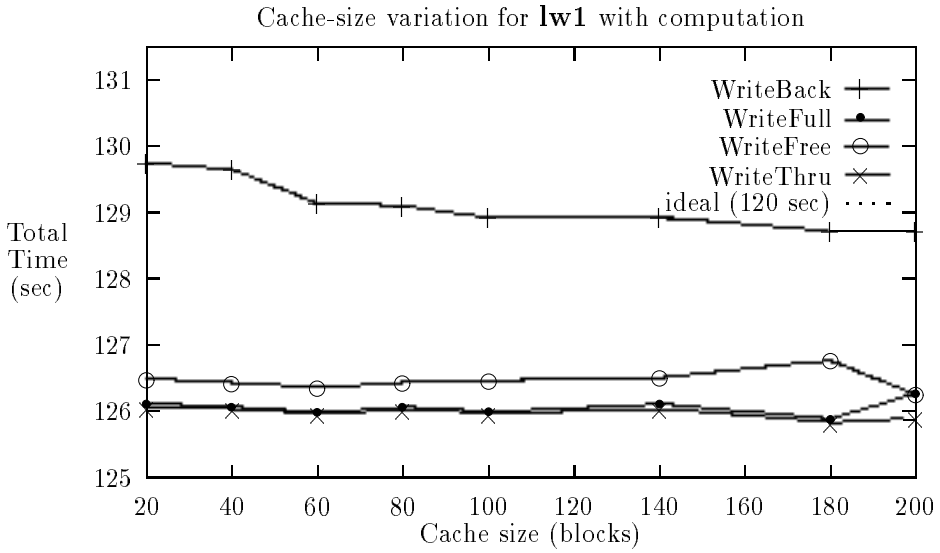


Figure 9.4: Cache-size variation for write pattern **lw1** with computation.

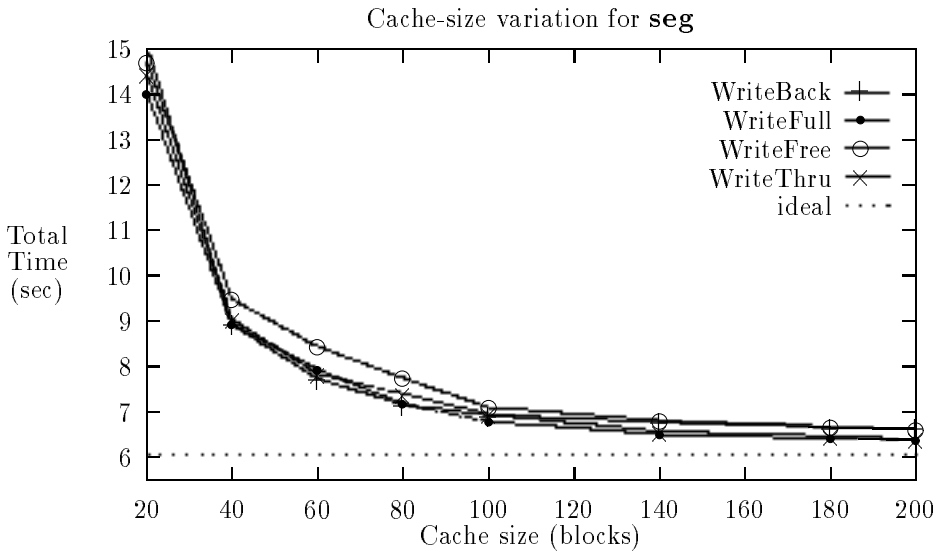


Figure 9.5: Cache-size variation for write pattern **seg**.

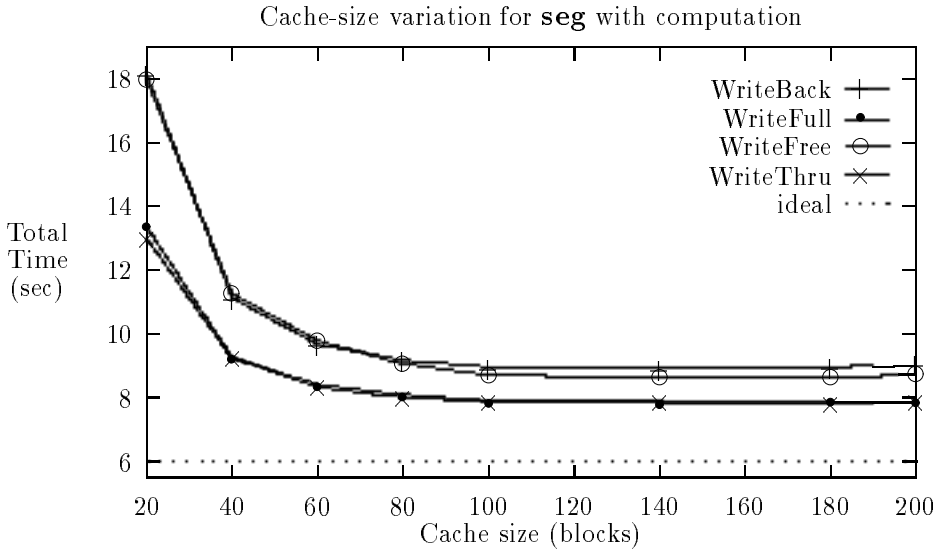


Figure 9.6: Cache-size variation for write pattern **seg** with computation.

9.4.2 Record-size Variation

In the experiments of this section, we varied the record size of the access pattern with a fixed cache size of 80 one-block buffers. The variation includes both integral and non-integral record sizes. The latter are important because they cause multiple accesses to many blocks.

Figure 9.7 shows the record-size variation for the write-only **gw** access pattern. WriteThru is clearly a poor choice for small record sizes, due to a huge number of rewrite mistakes. WriteFree was smarter, waiting until the buffer was mostly unused before issuing a disk write, but it was still not perfect due to some mistakes and to not immediately writing the blocks to disk when they finally were ready to be written. (Some mistakes could be avoided by increasing the RU-set size; see Section 9.4.3). WriteBack was sometimes faster than WriteFree because it had fewer rewrite mistakes. Finally, the WriteFull method had a nearly perfect 6-second execution time over all record sizes, because it issued the write precisely when the block was ready to go to disk, and made no mistakes.

Note the dips in the curves for all but WriteFull. These occur at integral record sizes (1, 2, 4, 5, 8, and 10 blocks),[1] where there was only one access per block. This avoided any opportunity for mistakes, which were common in the non-integral record sizes. Note that as the record size increased, an increasing number of blocks were accessed only once, even with a non-integral record size. This accounts for the convergence of these three methods as the record size increases.

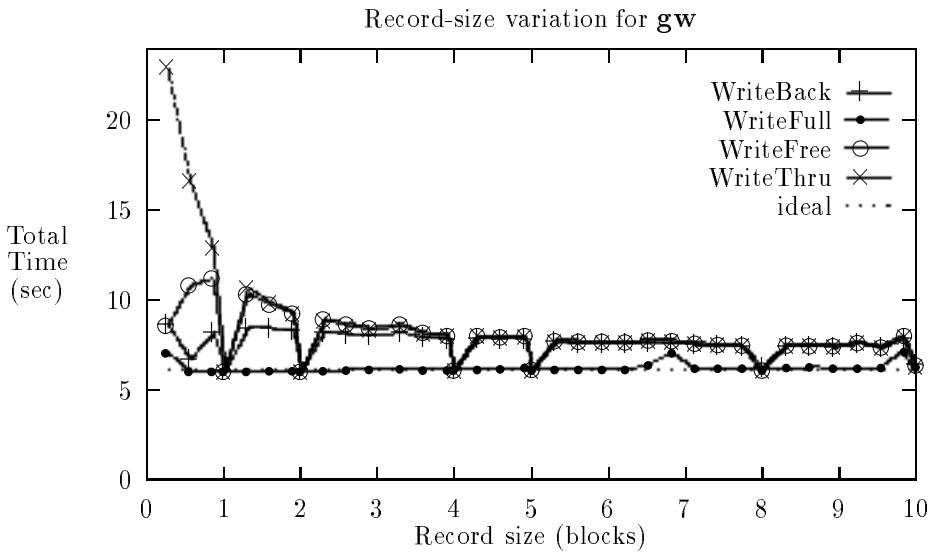


Figure 9.7: Record-size variation for write pattern **gw**.

[1] We use these record sizes because they divide the 4000 blocks into an integral number of fixed-size records.

The results for **lw1** are shown in Figure 9.8. The high execution times were due to reduced I/O parallelism, because one process could not keep 20 disks busy, even with an 80-block cache. In fact, one process could only keep 5–10 disks busy at any one time, a limitation due to the overhead of filling buffers and queuing them for disk I/O. With non-integral record sizes this time was increased due to repeated accesses to some blocks. Thus, the time varies widely for non-integral record sizes. WriteBack was usually slowest, because it delayed the write too long. WriteThru was also slower for small non-integral record sizes, especially for the smallest record size, due to the rewrite mistakes. No other method could have rewrite mistakes. No method had reread mistakes (since only one process wrote the file, and sequentially, no block could leave the cache early). WriteFree tied WriteFull, although in a pattern with computation it would be slower for some record sizes (see page 149).

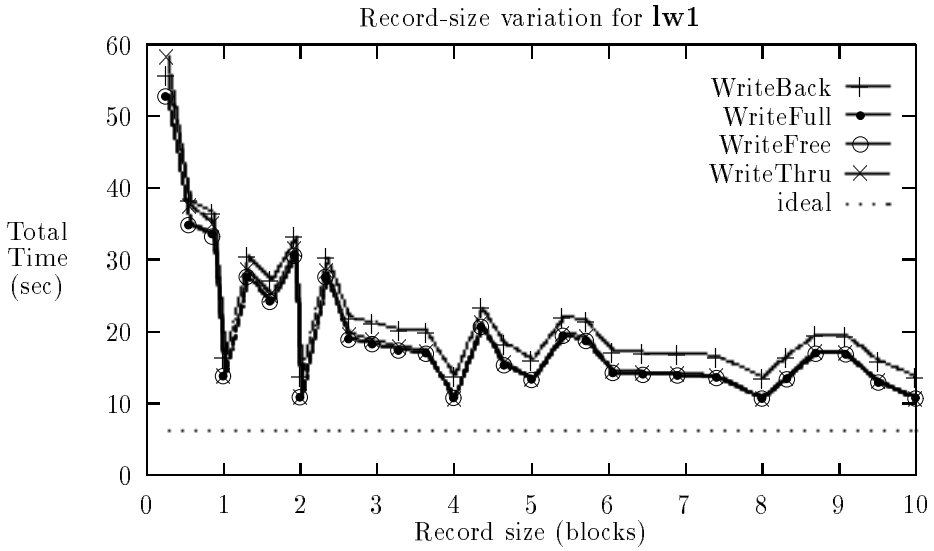


Figure 9.8: Record-size variation for write pattern **lw1**.

The record-size variation for the **seg** pattern (Figure 9.9) shows that WriteThru was much slower than the others. This was due to WriteThru's extreme number of rewrite mistakes. Due to the sequential access pattern on each processor, none of the others had rewrite mistakes, and none had reread mistakes. WriteFree delayed the writes a little more than WriteFull or WriteBack, and was thus a little slower.

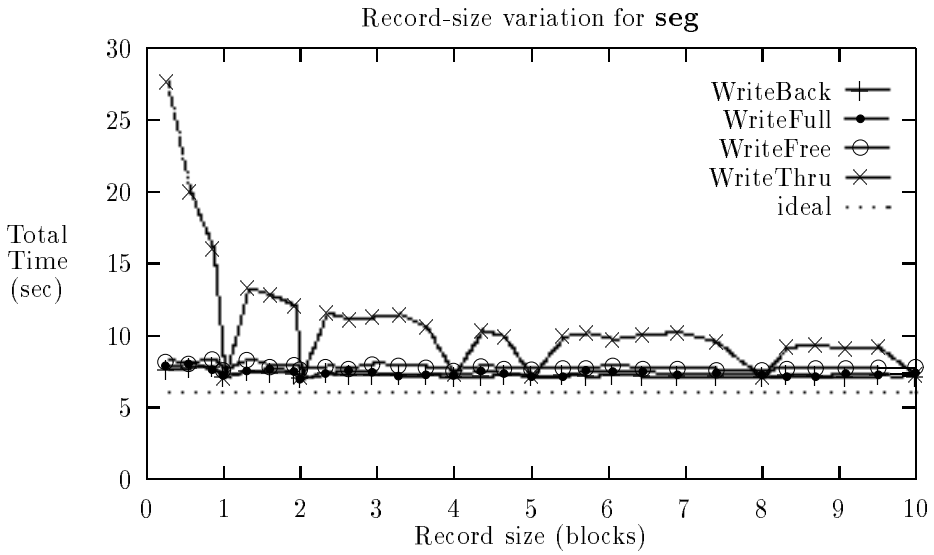


Figure 9.9: Record-size variation for write pattern **seg**.

Summary. The record size was an important factor in the performance of our write methods, because partial-block writes could lead to rewrite and reread mistakes. This was especially bad for records smaller than one block, where WriteThru's deficiency stands out. It was also significant for other non-integral record sizes larger than one block. For integral record sizes, all methods were essentially independent of record size. WriteFull was the most successful, never making mistakes regardless of record size.

9.4.3 The WriteFree Method

The performance of the WriteFree method was tied to the choice of the local RU-set size, since this size determined when a block left the local (and hence global) RU-set (page 24). As a demonstration, consider Figure 9.10. With one-block records (essentially, any integral record size), it was important not to delay in issuing the write, so a small RU-set size was better. For quarter-block records (a non-integral record size), a larger RU-set size helped to delay the block from leaving the set until it was fully written, thus reducing the number of reread and rewrite mistakes. Thus, the best choice would depend on the record size.

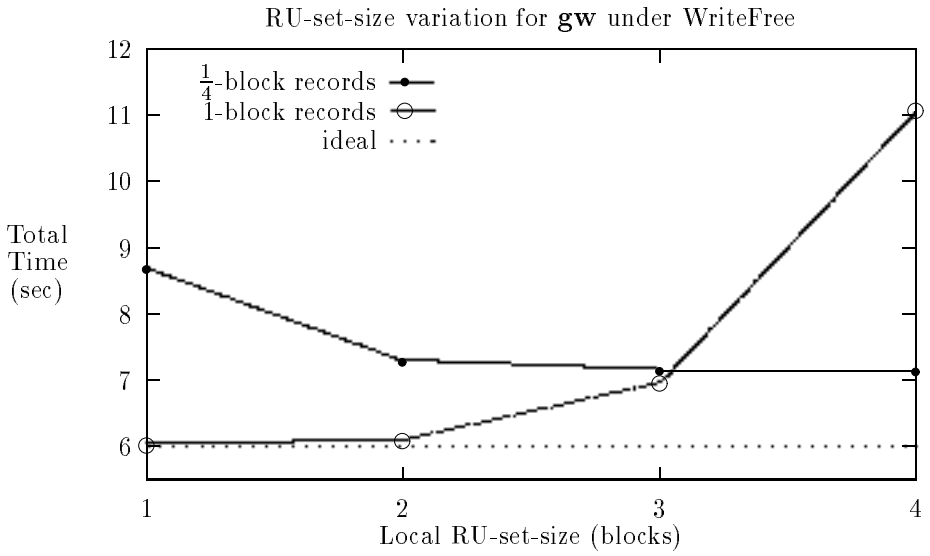


Figure 9.10: Demonstration of the effect of RU-set size on the WriteFree style. Pattern is **gw**, cache size is 80 blocks.

9.5 Conclusion

Given the types of write-only access patterns we expect to be common, our exploration of four methods shows that WriteFull, the most sophisticated of the methods, was consistently at or near the best performance in all situations. In the **lw1** pattern with one-block records, WriteThru and WriteFree did run slightly faster, but these methods were not nearly as successful in other cases.

A fairly small cache (40–80 blocks, i.e., 2–4 blocks per process) was sufficient to obtain the best performance, except in the **seg** pattern, where larger caches helped mask the disk contention.[2] The **gw** pattern had the fastest total execution time (nearly ideal), and the (sequential) **lw1** pattern was significantly slower. High-performance parallel file writing is definitely possible with these caching techniques.

[2] With faster processors, larger caches would be needed to cover **seg**'s disk contention.

Chapter 10

The File System Interface

Most of this dissertation concentrates on the ability of caching and prefetching to deliver the performance of parallel disk I/O hardware to the application. We assume that the application uses a conventional file system interface, through operations like *open, close, read, write,* and *seek*, to access a file that is interleaved across multiple disks. This interface hides the underlying parallel nature of the file and the file system, so that the application programmer need not be concerned with the details of these issues. Our results show that automated caching and prefetching in the file system can bridge the gap between the application, which has no knowledge of the underlying parallelism, and the parallel disks, leading to increased performance. Indeed, they could successfully use parallel disks even when the access pattern was not parallel (i.e., from a single process). More success, however, was possible when the application used a parallel access pattern (Section 5.1). Based on these results, we can now consider interface issues: (1) how convenient it is for the programmer to specify parallel file access patterns, (2) whether and how the file system interface should be changed to make such specifications more convenient and more amenable to caching and prefetching, and (3) whether entirely different paradigms (such as memory-mapped I/O) are needed to realize performance. The success of prefetching and caching suggests that we consider the first two issues, since it does not seem that entirely different I/O paradigms are necessary for performance reasons. Our discussion of (1), in the next section, indicates that *some* interface changes are needed for programmer convenience, if not for performance.

For concreteness, we use the Unix file system interface [RT78] as an example of a conventional interface. The Unix file system interface is in widespread use, even in multiprocessors (e.g., those made by Sequent, Encore, BBN, and Intel). Note that some of these implement the Unix file system interface without the Unix file system or the rest of the Unix operating system. The advantages to using the Unix (or similar) interface for a multiprocessor include application portability, programmer familiarity, and simplicity. This interface does not, however, directly support parallel disk I/O, and thus sometimes impedes the use of parallel disks. Thus, we believe that a change is needed for parallel systems. We propose an extension to the conventional interface that allows the use of the conventional interface without modification to existing software. Thus, all files can be accessed by parallel applications using the new extensions as well as by unmodified traditional applications.

In the next section we outline many problems with using the Unix interface for programming parallel file access patterns. Note that our complaints are not with Unix specifically, but with the Unix file system model (which was never intended for a multiprocessor environment). Then we describe our proposed extension to the conventional file system interface, which simplifies the specification of parallel access patterns and which can provide valuable information to the file system to be used in caching and prefetching decisions. We also discuss optional semantic information that

the user can give to the file system to help it do better caching and prefetching.

10.1 The Conventional Interface

In the Unix file system a file is modeled as an addressable sequence of bytes (sometimes referred to as a "seekable stream"). The interface is defined by the kernel file system calls [RT78], not by the *stdio* library package. The operations provided are *open*, *create* (called *creat* in Unix), *close*, *read*, *write*, and *seek* (called *lseek* in Unix). The *open* and *close* operations mark the start and end of activity on a given file. *Create* creates a file if necessary. *Open* is provided a file name and an intention (read, write, append, or read-write), and returns a *file descriptor* that is used in all of the other operations. Associated with the file descriptor is an implicit *file pointer* that maintains the current file position. The file pointer is used and updated by *read* and *write*, and reset by *seek*. *Read* and *write* take a file descriptor, a user buffer, and a length in bytes, and return the actual number of bytes read or written (zero at end of file). The data are written at the file position indicated by the file pointer, and the file pointer is updated to point just after the last byte read or written. *Seek* requires a file descriptor, a byte offset, and a mode indicating that the offset is relative to the beginning of the file, to the end of the file, or to the current file position. *Seek* returns the new file position. Extra features, such as support for logical records and indexed files, are not part of the basic Unix file system.

Depending on the particular multiprocessor implementation of the Unix interface, there are many difficulties in using the interface to program a parallel file access pattern. In some cases, the Unix file system operations are not atomic, leaving synchronization and atomicity to the user. Thus, many of the difficulties involve synchronization among the cooperating processes. Sometimes other features of Unix (or some versions of Unix) can be used to satisfy the needs of parallel I/O programming, but only in an awkward way. We discuss several problems here, sometimes by considering how one would specify our parallel file access patterns using the Unix interface.

In our model of parallel applications, all processes that are part of a single parallel program access a common file. Unless a single *open* operation opens the file once for all processes in the application, each process must open the file independently. This requires all processes to have access to the file name and read/write intention. It also generates many *open* requests that must be processed by the file system. Thus, it is both inconvenient and inefficient to depend on a single-process *open* operation.

Note that with Unix *process* semantics, not necessarily included in a system supporting Unix-like *file* semantics, a file open at the time of a *fork* is also open in the new process created by the *fork* ([LMKQ89], page 175). They also share the same file pointer. For systems supporting this or some other form of open-file inheritance, the multitude of single-process *open* operations can be avoided. It is, however, limited to files open before the *fork*, and thus to closely related process groups. It is not a general-purpose mechanism for opening files in arbitrary process groups. In Unix 4.3BSD, an open file can be shared with an arbitrary process by passing it through a Unix-domain socket ([LMKQ89], page 175), although this mechanism is complicated.

Our global access patterns arise when the processes read or write the file in a self-scheduled order. The ideal mechanism for this is a file pointer that is shared by all processes, and atomically updated by the *read* and *write* operations. The Unix file system interface does not directly provide shared file pointers. With Unix process semantics, however, a file pointer can be shared when an open file is inherited after a *fork* operation or passed through a Unix-domain socket. Unfortunately, there is not enough concurrency control in implementations of this mechanism to make accesses to

the shared file pointer atomic.[1] Unix 4.3BSD supports an atomic-append mode ([LMKQ89], page 174), which handles one common case, but not the general case.

As an example of the difficulties programmers have with global access patterns, consider the case of a local programmer who wanted to create a new file and have several processes append records (lines of text) to the file. The order of the records was not important. Using the Unix interface, all processes independently opened the file for "append," and proceeded to write to the file. Since each process had an independent file pointer, however, a record appended by one process did not affect the file pointers in other processes. Thus, each appended record was later overwritten by other processes, and the file ended up containing only garbage. This programmer needed an "atomic-append" mode supported by the file system interface. As we mentioned, some versions of Unix do support atomic append.

A general self-scheduled access order can be implemented using only the Unix file system semantics (which do not include the shared file pointers provided by Unix process semantics). A shared counter is used to indicate the next byte of the file to be read or written. The counter is atomically incremented by the length of the record a process wishes to read (write), using a fetch-and-add operator.[2] The original value of the counter, obtained from the fetch-and-add, is used in a *seek* operation, which is followed by the *read* or *write*. There are two problems with this implementation. First, it requires care by the programmer to properly maintain the atomicity of the overall operation. Second, the record length must be known in advance, which is difficult when reading variable-length records. This case requires either a separate record index or more serialization. Note that a strictly interleaved pattern, which is a special case of the self-scheduled pattern, avoids the fetch-and-add and some of the atomicity problems, but still forces the user to compute file positions for *seek*. It also has the problem with variable-length records. Finally, if the global pattern has sequential portions (i.e., is not **gw**), additional synchronization is needed to detect the end of a portion, to choose the next portion, and to reset the shared counter used above.

We assume that each file is interleaved, or at least declustered (page 5), across many disks in the system. If the file system does not maintain the declustering information for each file, forcing the programmer to specify the set of disks, disk files, or disk blocks, then transparency is lost and the interface is much harder to use. An example of this situation is in [Cro88]. Another example is the NCUBE file system, which does not distribute a single file across disks [PFDJ89]. Instead, the user must explicitly manage single-file parallelism. We believe that it is important to have a single name (e.g., Unix pathname) that defines the parallel file, and to leave the rest to the file system.

Now consider programming the read-only **seg** access pattern. In this pattern, the file is divided into disjoint segments, one per process. Each process must open the file, then locate and read its segment. The process (or some master process) must find the length of the file, use the length to compute the length of the segments, determine the segment it is to read, seek to the beginning of its segment, and read bytes of the file until the end of its segment is reached. If the division into segments is a simple matter of dividing the file length by the number of processes, then little work is needed. If, however, the file contains logical records, care must be used to divide the file at record boundaries. Another problem is assigning segments to processes, which may be facilitated by a shared counter or by predetermined process identifiers. In all, this is not too difficult, but is not convenient.

Now consider programming the write-only **seg** access pattern. Here, each process writes a separate segment of the file. The assignment of segments to processes is similar to the read-only

[1] One would expect the individual *read* and *write* operations to be atomic, but we found that this was not always true. File locking is supported by some Unix versions, and could be used to enforce atomic access.

[2] Fetch-and-add is described in [GLR83]. Note that it can, if necessary, be implemented on top of an existing lock primitive.

case, but this time it is much more difficult to determine the starting position and length of each segment. Unless the eventual length of each segment is known in advance, the starting positions of the segments are impossible to compute. The alternative is to create a separate file for each process, but this fills the file system with many more files than is really necessary, and makes later manipulation of the data more difficult. It would be easier if the file system supported the idea of segmented files.

Finally, note that user-level buffering, such as that in the Unix *stdio* interface, can lead to incorrect results. If the user-level buffers are allocated on a per-process, per-file basis, then buffer consistency problems arise. For example, one process writes some data to a file, but the data remains in the user-level buffer. Another process then tries to read that part of the file, and receives outdated data since it (and the file system) has no knowledge of the new data in the first process's buffer. This same effect could occur with a poor implementation of kernel-level buffers. Thus, any user-level buffering must be carefully integrated with the file system caching mechanism.

Overall, the Unix file system interface and semantics either cannot support our expected parallel I/O access patterns, or can only support them with great difficulty. For example, with Unix 4.3BSD [LMKQ89] a file can be opened by one process, passed to a group of processes through a Unix-domain socket, and then accessed atomically using file locking, but the interface and mechanisms are complex, not usually known to the average programmer, and not portable. A higher-level interface is needed for programmers to more easily take advantage of parallel I/O. Certainly, an implementation of the high-level interface could use these Unix facilities where available.

10.2 Our Proposed Interface

Our experiments show that high performance file I/O is possible with the conventional interface, when assisted by caching and prefetching. The previous section demonstrates, however, that the conventional interface is difficult to use for programming parallel I/O access. It is thus for programmer convenience, rather than performance or functionality, that some new interface constructs are needed. Extensions to the conventional interface retain the performance benefits of caching and prefetching while adding convenience for the programmer. There are several goals for the new interface:

- The conventional interface should still work. We want to support programs ported from other systems, and programmers who do not require the expressive power of the extended interface.
- The parallel extensions should be easy to use. One reason for extending the interface is programmer convenience.
- The common parallel access patterns should be supported.
- Details of the underlying parallel disk structure should be hidden from most applications, to enhance portability.
- The interface should be consistent with caching and prefetching. Since we can depend on caching and prefetching for high performance, our interface concentrates on convenient mechanisms for parallel file access patterns. In some places the interface actually helps prefetching efforts by exposing the programmer's access pattern intentions to the file system. Thus, in some cases the new interface should further improve performance.

We describe the basic concepts, along with a few implementation notes. Each concept directly addresses one or more of the problems outlined in the previous section. The syntax of the interface,

operation names, and parameter types depend on the language and operating system, and so ours are only a rough sketch.

10.2.1 Concepts

Directory Structure. There should be a single file-naming directory structure for the entire parallel file system. This hides the disk layout from the user and programmer. In some parallel file systems, the user must specify the list of disks involved [Cro88] or the list of local disk files [PFDJ89] when opening a file, since each disk has a separate directory structure. This is too burdensome for the programmer, and also makes the program less adaptable to changes in the disk resources, disk load, and so on. The name structure should be the same for parallel applications as for sequential applications (such as file-maintenance and directory-listing tools).

Note that a single directory structure can be physically distributed across multiple file servers without a central bottleneck. In the Sprite file system, for example, the tree-structured name space is partitioned among the disk servers, and all clients maintain a *prefix table* that maps file pathname prefixes to disk server locations [OCD+88]. Intel, whose Concurrent File System has a single directory structure, chose to use a centralized directory manager for simplicity, since optimizing *opens* and *closes* was not a priority [Pie89].

Multiopen. For a file to be accessed by all processes in an application, it must somehow be opened for all processes in that application. Every process could open the file independently. This, as we pointed out, is inconvenient and inefficient. Alternatively, if the process-creation mechanism includes open-file inheritance (as does Unix's *fork*), the file could be opened before all the processes are created, and the open file inherited by all processes. This is insufficient for our purposes, since it is limited to files that are open before the processes are created, to process groups that are created from one master process, and to systems that have open-file inheritance. We would like a mechanism that is not dependent on such process semantics.

We propose adding a *multiopen* operation, which opens the file for the entire parallel application when run from any process in the application. This assumes a way to group the processes into an "application", presumably more general than the set of children of one parent process. Most significantly, the *multiopen* is executed *after* the process group exists, so the group is not limited to pre-opened files. In most applications the *multiopen* would be executed in the "master" process. *Multiopen* opens the file only once, avoiding repeated directory searches and other overhead, and gives each process in the application its own file descriptor (through some implementation-dependent mechanism, e.g., shared memory or Unix-domain sockets). *Multiopen* can optionally create a file if it does not exist. In addition to the parameters required by *open*, *multiopen* requires a pointer to the file descriptor variable, the file pointer type, discussed next, the access mode (and possibly associated parameters), discussed below under *Type Coercion*, and finally, a list of optional hints.

File pointer. When a file is opened with *multiopen*, the programmer specifies whether the file pointer should be *local* (providing each process with an independent, local file pointer), or *global* (providing a single shared file pointer for all processes). These two choices correspond directly to our local and global access patterns. A global file pointer provides the synchronization needed to implement global file access patterns: a *read* or *write* operation on a global file pointer combines the transfer and file pointer update into a single atomic action, facilitating self-scheduled access

patterns.[3] The file system can ensure this atomicity without sacrificing concurrency. Either type of file pointer can be changed with the *seek* operation.

The global file pointer provides for atomic access to a shared file pointer. Because this implements self-scheduled access to the file, the process has no control over exactly which record is read or written when it uses *read* or *write* on a global file pointer. Since it may need to know the position of the transfer (to know, for example, the record number of the data just read), the original value of the file pointer should be returned after the transfer is complete, along with the number of bytes transferred. For compatibility, we do not change the interface of *read* and *write.* We define the *readp* and *writep* operations, which are the same as *read* and *write*, respectively, except that they also return the original file pointer position.

Portion support. The global file pointer supports simple self-scheduled access, such as that in **gw** read or write patterns. For global patterns that need more than sequential access (i.e., that have multiple portions), more synchronization support is necessary to handle the transition between portions. It is not clear how common these pattern types will be. If it is determined that they are commonly used, we have a mechanism to support them. Otherwise, this new mechanism can be omitted with no effect on the rest of the interface. Note that its use is optional to the programmer, and that although it is intended for global file pointers and read-only patterns, it also works for local file pointers or write patterns.

The problem is to control portion skips when using self-scheduled access within each portion. A process examining the global file pointer may satisfy itself that the file pointer is still within the current portion, but a subsequent *read* may occur outside of the portion due to concurrent *read* operations incrementing the file pointer. Unless the processes have some external mechanism to limit their access to the current portion, they will read past the end of the portion. With *readp* they can detect the condition, but only after reading data that they did not need. What is needed is a way to tell the file system where the portion ends.

The idea is to provide the file system with the position of the beginning and end of the current portion. The global file pointer is moved to the beginning of the portion, and subsequent *reads* or *writes* atomically increment the file pointer through the portion. At the end of the portion, *reads* or *writes* block until the next portion has been specified. If the next portion is fully specified before the end of the current portion, there is no delay or loss of concurrency between portions.

The primary method for specifying the next portion is through a user-specified *upcall* function [Cla85]. When necessary, the file system calls the function to request the position of the next portion. Normally, the function returns this information. Alternatively, it may specify that there is no next portion (which is treated as an end of file), that the current portion should be extended, or that the portion mechanism should be disabled. All of these release blocked read and write operations.

As an optimization, the *nextportion* operation specifies the next portion's bounds before the upcall is needed. Another optimization allows the file system to use the upcall prematurely, although the application need not make a decision until it is necessary. The purpose of these optimizations is to avoid delay at portion transitions.

Logical Records. Dibble [Dib90] argues for direct support for logical records in the file system. The Unix file system does not have any built-in support for logical records, in contrast to some traditional systems (typified by commercial mainframes). Such support increases the complexity

[3] Note that an alternative is to add a file-position argument to the *read* and *write* operations. This does not help, though, to make a self-scheduled pattern.

of the file system, but there are good reasons for logical record support in a parallel file system, even when not supported in a similar uniprocessor file system:

- The record support can be combined with global file pointer synchronization to provide atomic operations for reading and writing records. This is particularly useful if the records have variable length.
- By understanding logical records, the file system can avoid splitting a record over two blocks. In some parallel access patterns, this increases concurrency (justified by the results in Section 8.1). It can also increase performance in random access patterns (at the cost of wasted space).
- The file system can provide record locks as a convenience to the programmer.

In our interface, then, we divide the files into *byte* files and *record* files. The file type is an attribute of the file. All references to "position" in a record file are record numbers instead of byte offsets. This affects the *read*, *readp*, *write*, *writep*, *seek*, and portion operations. Fixed-size logical records are trivial to support, since the location of any record is easily calculated from the record number. Variable-sized records are more difficult, since an implementation must be able to atomically read the next record and update the file pointer, with high concurrency.

Multifiles. In most parallel programs, a data set is divided among the processes in the program. In the conventional file system, however, a single data set is usually represented as a single file. For a parallel program to use a conventional file system, the individual process subsets of the data set must either be combined into one file or stored in separate files, one per process. Neither option is convenient, as we showed in our examples using the **seg** patterns. We provide a new type of file called a *multifile* for these situations. To the file system a multifile is a single file, with one directory entry, but it is different from a *plain* (conventional) file in that it is not a single sequence of bytes. Instead, it is a collection of subfiles, each of which is a separate sequence of bytes. A multifile is created by a parallel program with a certain number of subfiles, usually equal to the number of processes in the program. Once created, the number of subfiles is fixed. Each process writes its own subfile. Later, when the multifile is opened for reading, each process reads its own subfile. (Note that a multifile implies local file pointers.) Each process has the illusion of reading an independent small file, since each subfile is independently addressed with its own first byte and end-of-file marker. Each subfile can be extended or truncated without affecting the addressing in any of the others. Thus, a multifile combines the advantages of a single file (single name for a single data set) with those of multiple files (independently addressable and extendible, easily located beginning and end).

When opening an existing multifile, an optional mapping may be specified that indicates the assignment of subfiles to processes. With the default mapping, the number of subfiles must match the number of processes, and an arbitrary one-to-one mapping is used. With a user-specified mapping, there is no requirement on the number of processes. In fact, the mapping may specify that some subfiles are not used, or that some processes have no subfile. For applications that want to manipulate many subfiles with few processes, we provide a *newsubfile* operation that switches the mapping for that process to a given subfile. Although a multifile is defined to be an unordered set of subfiles, the subfiles should be given some fixed, if not deterministic, order. This allows the subfiles to be numbered, which is needed for specifying mappings and for the *newsubfile* operation.

Multifiles are most useful between parallel programs, so data can be written as separate subsets and later read as separate subsets. They are also useful for output intended for sequential programs. An example is a single file that contains debugging output, with a separate subfile for each process.

There are two primary possibilities for the storage of multifiles in a parallel disk system: interleaved, where all subfiles are interleaved over all disks, and localized, where the subfiles are distributed over all disks, but with each subfile stored entirely on a single disk. The localized approach reduces contention and communication delays, but suffers if the file is read as a plain file by a single process (see *coercion* below), since there is no way to use the disks concurrently. The interleaved approach spreads the load over all disks, at the risk of increased contention and communication delays. Our experiments with the **seg** access pattern indicate the danger of starting all subpatterns on the same disk (page 64), so the interleaved multifile format begins each subfile's interleaving pattern on a different disk. This helps to start the pipeline. The storage mode for a multifile defaults to interleaved, unless indicated by the user in a storage hint.

Type Coercion. Our file system interface supports four file types:

	byte	record
plain	byte plain file	record plain file
multifile	byte multifile	record multifile

Note that the "byte plain file" is the same as conventional files. Every file in the file system is stored as one of these four types. These file types also represent four access modes that can be specified at the time the file is opened. For compatibility, all files in the file system can be read as a byte plain file. In fact, for convenience we allow any file to be *read* in any mode, with the file system coercing the stored file into that mode. Note that coercion is just a mapping operation; the stored file does not change. We do not allow a file to be opened for writing if coercion is necessary, since it is not always clear how to map some write operations.

Although most coercions are done transparently, some applications may want to adjust themselves to the stored file type. The *type* operation can be used to request information about file type (plain or multifile, byte or record). This operation may be merged with existing mechanisms that query other file attributes (*stat* in Unix).

To coerce a record file into a byte file, we ignore record boundaries, fragmentation overhead (empty space in blocks), and any other overhead, such as length fields or indexes. To coerce a byte file into a record file, the user provides either a fixed record size or a record delimiter character (e.g., newline). The details depend on the particular implementation of records.

To coerce a multifile into a plain file, the subfiles are ordered in some way (for sanity the same order is used every time), and concatenated together to form the illusion of one long file. A plain file can also be coerced into a multifile. This is a useful way to divide a file's data into contiguous chunks for a variable number of processes. The user specifies the desired number of subfiles (usually the number of processes), and the file is divided roughly evenly among the subfiles, with each subfile assigned a contiguous portion of the original file. If the file is a byte file, the division is by bytes; if the file is a record file, or coerced into a record file, the division is made at record boundaries. In any case, the end of a coerced subfile appears as an end-of-file to the process assigned to the subfile.

10.2.2 Implications

Note that this interface has few implications for the underlying parallel file system. The parallel file system must support files interleaved across all disks, optionally restrict a file to a single disk, and provide a simple block interface to the files. The interface, including the directory hierarchy, multiopen, global and local file pointers, portion support, multifiles, and logical records, can all be built on top of this primitive parallel file system. For good performance, it is best for the caching

and prefetching mechanisms to be integrated with the interface, or at least to allow the interface implementation to interact closely with the caching and prefetching mechanisms, so that knowledge of record size, portion length, *etc.*, can be used for caching and prefetching policy decisions.

Within the interface, there are many synchronization issues. In particular, the support of global file access patterns requires atomic access to a shared file pointer. This is particularly complicated if the file-pointer update involves checking for the end of a portion, or finding the length of the next logical record. The latter may require reading data from disk, unless there is a separate record index.

Note that most of the functionality we propose in our interface is already supported by some Unix systems, or could be added in a relatively transparent way through Unix operations such as *ioctl*. This would not, however, accomplish the primary goal of our interface: to encourage parallel I/O programming by making it easier to use.

10.2.3 Examples: Our Access Patterns

Every one of our parallel file access patterns can be easily supported by the new interface. The local or global file pointer choice is clearly intended to support local and global access patterns. Thus, all of the local access patterns (including the **rnd** access pattern) involve a *multiopen* with a local file pointer, and use *read*, *write*, and *seek* to access the portions of the file. One special case is the **seg** access pattern, which clearly motivates multifiles. The **seg** pattern (read or write) is directly supported by the multifile construct, whether coerced or not. Another special case is the **lw1** pattern, for which the original *open* operation suffices.

The global file access patterns (the **gw**, **grp**, and **gfp** read patterns, and the **gw** write pattern) are supported by the global file pointer construct. The global file pointer and the semantics of the new read and write operations directly support self-scheduled access within portions, and the portion mechanism supports portion skips.

10.3 Additional Semantic Information

Often the programmer has knowledge about the access pattern that would help the file system's caching and prefetching efforts. If the file system interface provides a way for the programmer to communicate this information to the file system, then the performance may increase. It is optional for the programmer to provide this information, and it is optional for the file system to use it, but its use may help improve performance. When the information is not guaranteed to be correct, it is called a *hint*. The hint may be explicitly provided by the programmer, or implicit in other file system calls or parameters. We first describe the types of information that are useful, then some mechanisms for providing it to the file system.

10.3.1 Types of Information

There is much optional information that a programmer can provide. The inspiration can be from the file system documentation, which describes the factors that affect performance, the characteristics of access patterns, and the kind of information that can be helpful to the file system. Inspiration may also come from the file system itself, in the form of statistics and other feedback. The information that can be supplied by the programmer includes:

- Local or global access pattern. This is implicit in the choice of a local or global file pointer.
- Sequential or random access pattern. Prefetching can be avoided in random patterns.

- Whole file access pattern. If it is known in advance that the whole file will be read, a bold, simple prefetching method (such as IBL or GW) can be used.
- Specific access pattern style. Knowing the access pattern style (e.g., **lfp**) allows the file system to choose an appropriate predictor.
- Access pattern details. The location and length of portions are invaluable prefetching information, and are implicit in the portion support operations.
- Specific predictor. Here the user explicitly states which predictor or write method to use.
- Record size. The logical record size is useful in global pattern prefetching. This is implicit when record-mode access is used, but is useful even when the application manages its own records.
- Amount of data sharing. Whether any data will be used by more than one process.
- Rate of data use. An estimate of the data throughput needs of the application.
- Phase change is coming. This hint warns that the access pattern is changing. An example is a jump to a new portion, or a change from one pattern to another. A seek is an implicit hint of a phase change (although a little late to help prefetching much).
- Preallocate. Knowing the eventual size of a new file can cut the overhead used for repeated automatic extensions.
- Do not cache. There are times when caching is not appropriate.
- Do not prefetch. There are times when prefetching is not appropriate.
- Prefetch this record. A specific request useful in non-sequential patterns. This is more general than the common asynchronous read operation, which is limited to one outstanding asynchronous read. The prefetch hint could be arbitrarily mixed with normal read operations.
- Done with this record. Encourages removing this record from the cache, possibly causing a write to disk.
- Storage. Disk storage recommendations, such as which disks to use, interleaving unit size, and so on.

This information can help choose the right predictor, provide key information for prefetching, avoid mistakes, or make storage recommendations. Thus they can increase performance.

10.3.2 Mechanisms

The mechanism for providing semantic information varies with the rest of the implementation details. We propose the *hint* operation, with parameters *code* and *value*, to supply one hint with its associated argument. The *hintv* operation accepts a *hintlist*, which is a vector of *(code, value)* pairs. In addition, the *multiopen* operation accepts a (possibly empty) hintlist as a parameter.

Note that a *hintlist* is stored by the file system with the other file system attributes, and is loaded when the file is opened. The stored *hintlist* is constructed by the file system from information that the file system accumulates while managing a file: the predictor used, the type of access pattern and parameters, and the general success of prefetching. This is essentially a long-term

prediction mechanism. These hints should be considered less trustworthy than any provided by the application, since they may be based on information from entirely different uses of the file. The file system should be able to provide this accumulated information (and more, including statistics) to the user on request, perhaps in a human-readable report format, to suggest hints or changes to the application.

10.4 Related Work

10.4.1 Interface

Several researchers have discussed parallel I/O interfaces for MIMD multiprocessors. Dibble, in his design of the Bridge file system [Dib90], defines three interfaces: standard, which is essentially our conventional interface; parallel open, in which a control process issues all the read and write requests, automatically transferring one record in or out of every process; and tools. Tools have transparent access to the local file systems of each disk, allowing the data on each disk to be handled by the attached processor, minimizing data flow in the processor interconnection network. The standard interface is there for compatibility, the tools for performance, and the parallel-open interface is a compromise. Our proposed interface hides the underlying disk layout (unlike Dibble's tools), and has powerful constructs for expressing parallel access patterns, with the file system handling much of the optimization.

Intel's file system for their iPSC/2 multiprocessor, CFS [Pie89], also provides three interfaces [AS89]: standard (conventional); random-sequential access, which uses a self-scheduled global file pointer; and coordinated, which is for interleaved access with either a fixed or variable record size. The last is interesting, since it uses some serialization when the record size is variable, essentially implementing atomic append.

Another parallel file system is based on ways to lay out a file on parallel disks [Cro89, Cro88]. One interface provides self-scheduled access with a global file pointer. Another provides local file pointers. A "unified" access mode provides the standard interface for compatibility. One deficiency in this interface is that the user must supply a list of disks to the *open* operation.

The file system for the NCUBE hypercube multiprocessor ([PFDJ89]) is primitive, in the sense that each disk has a local file system independent of the others, and no global file system is provided. Parallel I/O must be managed explicitly by the user, using separate files on each disk.

The CUBIX file system for the CrOS system on hypercubes [FJL+88] connects a sequential file server to a parallel application program. It has two interfaces: singular, in which all processes simultaneously write the *same* data, and multiple, in which variable-length records are interleaved by process. Variable-length records are buffered until complete, then atomically written to the file.

Our interface grew out of an understanding of parallel I/O as determined by the results of our experiments with caching and prefetching. It is designed to conveniently support what we think will be common parallel access patterns, by combining many ideas from these other researchers along with several new ideas, supporting both sequential and parallel I/O, local and global pointers, logical records, and multifiles.

10.4.2 Hints

The idea of hints is not a new one. One distributed file system proposal uses the file type (determined by directory and file name extension) to make caching decisions (whether to cache, what replacement algorithm to use), based on past knowledge of access patterns for each file type [Kor90]. A similar system is used in [THY80], where the file type is either random or sequential, and either

temporary or permanent. File caching is controlled by the system administrator in [Gro85], deciding what files to cache, and in what way. Intel's CFS [AS89] allows the user to choose the disks to use and to preallocate files. The Casper distributed file system [FE90, Flo89] associates a property list with each file, which could contain file usage information and hints.

10.5 Summary

Our new interface allows for parallel *open* (with *multiopen*), synchronization for global file access (including portion support), support for logical records, and a new file organization (multifiles). All of the new features are compatible with the conventional interface, so that a file can be used by both a sophisticated, high-performance parallel application and a general-purpose sequential file-maintenance tool. Our interface also allows the user to provide hints that may improve performance, and the file system to provide feedback in the form of suggestions and statistics. We believe that this interface would make the task of programming parallel disk applications much easier, and would also increase performance.

Chapter 11

Conclusions and Future Work

We built a file system testbed called RAPID-Transit, which runs on a BBN GP1000 multiprocessor, and used it to evaluate prefetching and caching techniques for parallel file-access patterns in a scientific workload. In this chapter we outline the key results, and list some possible areas for future work.

11.1 Summary of Results

The benefits of caching and prefetching depended on the workload and other parameters. Fortunately, the best performance was often in (what we expect to be) the most common access patterns: **lw**, **lw1**, and **gw**.

11.1.1 Single-Process Access Patterns

In applications that are directly ported from a uniprocessor, or in which the programmer has not bothered to explicitly use parallel I/O, files may be read or written by a master process while other processes idle waiting for data. In a file system where files are spread over many disks, the full multi-disk bandwidth is not used by these simple access patterns. Prefetching, however, can be provided transparently by the file system and attain significant speedup (14.5 in a 20-process, 20-disk experiment) on an otherwise sequential part of the computation, by exercising the parallelism in the disk system. In a similar experiment (Chapter 9) where a single process wrote a file to 20 disks, intelligent buffer replacement improved performance by 15%. Despite our short study of these **lw1** patterns, we believe they will be commonly used in a parallel file system, so these results are encouraging.

11.1.2 Read-only Parallel Access Patterns

In parallel applications where all processes read the file, caching and prefetching can sometimes significantly improve disk performance. We first established the potential for improvements due to prefetching using the EXACT predictor. Then we defined and evaluated several local and global on-line predictors. We varied the access pattern, synchronization style, computation load, record size, cache size, disk-access time, number of disks, and number of processors. These are the main conclusions:

- Caching alone can have a tremendous affect on performance, particularly when the record size is less than the block size (Section 4.2). It does this by reducing the number of disk accesses

when there is strong locality (caused, for example, by sequential access to small records, or by inter-process data re-use, as in **lw**).

- The best prefetching improvements were for the **lw**, **seg**, and **gw** patterns. Since these will probably be the most common parallel access patterns, this is encouraging. Prefetching was less successful for the other patterns (**lfp**, **lrp**, **gfp**, and particularly **grp**), but then these will probably be less common. Thus, the common case has the best improvements.

- IOPORT appeared to be the best general-purpose local predictor, in that it provided high performance to a wide variety of patterns without causing poor performance for any pattern. Its performance was often better when supplied with a larger cache. In half of our test cases IOPORT was within 3% of the best predictor.

- The GAPS, RGAPS, and GW predictors each managed to successfully reduce the execution time of global access patterns in most cases. They also approached the best time, represented by the EXACT predictor, closely in many cases. GW was effective only for the **gw** and **lw** patterns, but may be useful because we expect these two patterns to be commonly used.

- The RGAPS and IOPORT predictors were fairly robust across all parameter variations. Although our initial experiments found that the RGAPS and GAPS predictors were essentially equivalent, the parameter-variation experiments show that RGAPS was usually superior to GAPS.

- An automatic switch mechanism was devised that was able to quickly determine whether a pattern was local or global, and switch to either IOPORT or GAPS, respectively. It added little overhead.

- Prefetching helped to overlap I/O with computation, file system overhead, and other I/O. Nearly ideal execution times were observed in some cases.

- In the less-predictable **grp** and **lrp** patterns, conservative predictors were more successful. When computation was mixed with I/O, it was better to be a little less conservative, to take advantage of the potential for overlapping I/O and computation.

- All of our experiments accessed 4000 blocks (200 for **lw**) in the file, which corresponds to 4 MBytes of data transferred. Certainly many scientific applications use larger data files. Our results should scale to larger files, with the benefits of prefetching probably increasing as the start-up overhead (e.g., recognizing the pattern, early mistakes) is amortized over the longer pattern. In no case should the benefits of prefetching decrease.

- When there were fewer processors than disks, the disks were better utilized with prefetching than without. In this case, the execution time with prefetching was often close to the ideal execution time. When there were more processors than disks, it was often faster to not prefetch at all. The parallelism alone was able to keep the disks busy, with less overhead and no mistakes.

- With fast enough disks (relative to processor speed), the overhead of prefetching was not worth the small benefits. That is, prefetching was sometimes slower than not prefetching. We expect, however, that increasing processor speeds will avoid this effect by lowering the cost of overhead.

- Our study of slower disks simulates the increased gap in speed between future processors and future disk drives. In this case prefetching was an important benefit in managing the disk bottleneck.

In short, prefetching was most useful when there were fewer processors than disks, when there was some computation to be performed in addition to the I/O, or when the **lw** pattern was used.

Our experiments were limited by the size of the machine available to us. We believe, however, that prefetching will scale to larger machines. For high performance the number of processors and the number of disks must be increased simultaneously. The precise ratio between the number of disks and the number of processors depends on their relative speeds and on the I/O and computation requirements of the workload.

11.1.3 Write-only Access Patterns

Given the types of write-only access patterns we expect to be common, our exploration of four methods shows that WriteFull, the most sophisticated of the methods, was consistently at or near the best performance in all situations. It timed the disk writes correctly, without making any mistakes that caused extraneous disk I/O.

11.1.4 Interface

Our proposed file system interface should make it easier to use parallel disks, and would aid automatic prefetching. It has many new features, including *multiopen*, synchronization for global file access, logical records, and a new file organization called *multifiles*. All of the new features are compatible with the traditional interface. Our interface also allows the user to provide hints that may improve performance, and the file system to provide feedback in the form of suggestions and statistics.

11.2 Future Work

Although prefetching is well explored by this dissertation, there are many possible extensions that examine architecture and workload alternatives, new prefetching and caching techniques, and issues such as fault tolerance.

11.2.1 Techniques

In this dissertation we concentrate on one replacement algorithm and one prefetching technique (with several predictors). There are inevitably more algorithms and techniques that we have not yet discovered. One general idea is to loosen the replacement algorithm, so that mistakes may be removed without an explicit request from the predictor. This may decrease the complexity and increase the concurrency of some predictors, at the expense of accuracy. Currently, for example, a prefetched block is not removed from a buffer until it is either used or specifically marked as a mistake by the predictor. A heuristic technique could use aging to flush unused prefetched blocks, relieving the predictor from catching its own mistakes.

11.2.2 Workload

The workload we use with our testbed is entirely synthetic, consisting of a set of likely access patterns. Its composition is based on our knowledge of uniprocessor access patterns and on discussions with parallel applications programmers. A study of parallel file access patterns is necessary to

improve our understanding of the workload. Unfortunately, parallel access patterns are probably influenced by the interface and file system architecture, so general characterization may be difficult.

This study leaves out files that are open for both reading and writing because we believe that they are less commonly used. In addition, it is not clear what read-write patterns would be used, and therefore what prefetching or write-back techniques might be useful. A related subject is overwriting files. This study assumes that all files open for writing are new files and are preallocated to their full size. We should consider disk-block allocation overhead and file-overwriting issues.

Finally, this study concentrates on scientific applications. General-purpose systems and transaction-processing systems have different workload characteristics, possibly requiring different caching and prefetching techniques. For example, program-development systems tend to have single-process applications, small files, and hence small transfers. Parallelism comes from running many sequential applications rather than a few parallel applications. Multiprogramming is more important. Caching commonly-used small *files* is more important than caching blocks within a large file. In another example, transaction-processing systems have large databases with unusual access patterns. In this case, caching and prefetching may be best left to the programmer. A low-level interface to the disks and simple caching support might be helpful to the programmer.

11.2.3 Architecture Changes

Our techniques are intended for shared-memory MIMD multiprocessors, with disks attached independently to several processors. There are many architectural issues, some changing those assumptions:

- How can these techniques be used on a non-shared-memory machine? Would entirely new techniques be necessary, or simply a new implementation? Our implementation of global predictors depends on centralized, shared data structures; these predictors may need to be relaxed for a non-shared-memory machine. Indeed, the self-scheduled global access patterns may be more difficult to implement; interleaved patterns may be more common. Buffers may need to be localized to particular processors. Blocks from a disk would be read into a buffer on the processor attached to the disk, then copied to a buffer in the requesting processor. If several processors need the block, replication or migration of the block among their caches might be useful (although consistency issues arise).

- What if some processors are dedicated to I/O, and some to processing? In this case, the I/O nodes would probably handle the prefetching and caching decisions. The pattern might then be viewed from the disk (I/O node) instead of from the process, which may be beneficial in scheduling disk accesses.

- What is the effect of Non-Uniform Memory Access (NUMA), as compared to UMA? Here, NUMA refers to either memory access times or disk access times.

- How much of a bottleneck is the disk controller? Do we really need one controller per processor/disk pair? Or could we put several disks on one controller on one processor? To some extent, this is a hardware balance issue.

- Our experiments use a fixed disk access time. What is the effect of a more realistic variable disk access time? To answer this question a disk layout must be determined.

- What disk layout is important? Should files be stored contiguously? Contiguous files are beneficial for sequential access, particularly for large files, since the sequential disk access

time is much lower than random access times. How would this affect our prefetching results? The prefetching strategy should account for this kind of layout with new techniques. For example, a common technique for prefetching in contiguously-stored files is to piggyback prefetches on each disk access, saving disk overhead.

- We assume that a parallel program consists of a set of processes, each assigned to its own processor. There is no multiprogramming. If multiprogramming were used, the idle time we currently use for prefetching could be filled by switching to another process. What are the tradeoffs here? When do we switch contexts and when do we prefetch?

11.2.4 Multiple Files

In this study we concentrate on a single application reading or writing a single file, with no outside contention for the disks. In many applications, of course, several files are in use simultaneously. And, of course, the disks may be busy with other traffic. There are many difficult issues involving buffer allocation, prefetching, and disk scheduling:

- How does one allocate buffers between the files? Should buffer allocation be local (one pool per file) or global (one pool for all files)? If local, should one file have more buffers than another file? Is a static allocation sufficient, or should a complex dynamic assignment be used?
- How should prefetching efforts be divided between files? How do you determine what file to prefetch first?
- When a disk is used for several files at once, can you control the disk schedule to increase performance?

11.2.5 Reliability

One of the major problems with parallel disk systems is the reduced hardware reliability. Roughly speaking, if n disks are involved in one file system, the file system breaks n times sooner than a single-disk file system. The Berkeley RAID project [PGK88] solves this problem by using parity blocks to recover information lost when a disk fails. Since any real parallel I/O system must use some error-recovery technique, what is the best way to incorporate both prefetching and fault-tolerance techniques in a single system?

11.2.6 Implementation

Certainly the current implementation can be improved and tuned. My testbed is a research prototype, and could be streamlined with careful tuning. It is not directly useful as a file system component, since it is intended only for research purposes. An interesting future project, however, would add my techniques (perhaps using some of the same code) to a working file system, and evaluate the system experimentally using real workloads and real disks. In a real implementation, concurrency is critical to performance. Ideally, all aspects of the file system would be highly concurrent: file opens and closes, directory management, cache initialization, and so on. Reliability would be a key concern. Some of the architectural issues would need to be resolved (e.g., disk layout).

11.3 Conclusion

We believe that high-performance parallel file systems can (and should) be built for MIMD multiprocessors, and that efficient caching and prefetching methods can boost file system performance. Most of these file system efforts are transparent to the user. We propose an interface that helps programmers to make the most of parallel I/O, with several parallel constructs as well as a sequential compatibility interface, and a mechanism to give hints to the file system. All of these techniques can alleviate the I/O crisis by scaling file system performance with multiprocessor performance.

Glossary

For some definitions, the page number with more information or first definition is given in parentheses.

access pattern (13) The list of logical records of the file, in the order they are accessed by the program. The predictors see the *block* access pattern, which is a list of blocks in the order that they are accessed.

ADAPT (51) Local predictor. ADAPT uses statistical methods to predict the eventual length of the current portion given its current length.

average block read time The average amount of time required to read one block from the file system. This averages the cost of cache misses and cache hits together.

block (13) A block is a contiguous set of bytes of the file. The size is determined by the disk. The logical blocks of the file are mapped to physical disk blocks, although we do not specify the mapping. Contrast this with a *buffer*, which is a space in memory that can hold one block. Also contrast with a *record.*

buffer (23) A space in memory that is exactly the size of one block. The *cache* is a collection of buffers.

cache (23) A collection of buffers in main memory to hold blocks of the file.

cache hit (22) A block is hit in the cache if the block's data is currently resident in some buffer in the cache. A cache hit is this event. See *cache miss.*

cache miss (22) A block misses in the cache if the block's data is not currently resident in some buffer in the cache. A cache miss is this event. In this case, the block will be demand-fetched from the disk. See *cache hit* and *demand fetch.*

coefficient of variation (31) The standard deviation divided by the mean, abbreviated as cv. We use this as a simple measure of measurement error. For example, $cv = 0.04$ means that the standard deviation was 4% of the mean. Note $cv \geq 0$.

cv See *coefficient of variation.*

declustered (5) The blocks of a file are scattered among the disks. The disks are accessed independently, though they may be connected to the same disk controller. This does not imply *interleaving.*

demand fetch When a block is read from the disk into the cache after a *cache miss.*

each(x) (30) Synchronization style. Barrier synchronization after reading x blocks each. Contrast with *total(x).*

EXACT (35) The predictor that is given full knowledge of the access pattern in advance. For local and global patterns. Contrast with *on-line predictors.*

free list (24) In our testbed, the buffers available for replacement are kept on a free list, which is implemented as two queues: the *ready queue* and the *unready queue.*

GAPS (80) Global predictor. GAPS tries to detect sequentiality in a global access pattern, and then to use it to do prefetching much like IPORT.

gfp (29) Global read-only access pattern (Global Fixed-length Portions). In this pattern, processors cooperate to read what appears globally to be sequential portions of fixed length and spacing.

global predictor A *predictor* for *global patterns.*

global access pattern (15) A global access pattern is a globally-sequential access pattern. That is, all processes in an application cooperate together to read the file (or sequential portion) in a roughly sequential order. Contrast with *local access pattern.*

greedy-process problem (47) In local patterns, where one fast process uses most of the buffers, and the other processes slow down due to lack of buffers. Then all wait at the next barrier for the slowest process.

grp (29) Global read-only access pattern (Global Random Portions). Processors cooperate to globally read sequential portions with random length and spacing.

gw (29) Global read-only access pattern (Global Whole file). This global pattern reads the entire file from beginning to end, the processors reading distinct blocks from the file, so that globally the entire file is read exactly once, but locally each processor only reads some small subset of the file with no discernible portions.

gw (29) Write-only access pattern. This pattern writes records of the file in a self-scheduled order, roughly sequentially from start to finish, with all processes cooperating to write the file.

GW (86) Global predictor. GW predicts that the whole file will be accessed, much like IBL, but for global patterns.

hit See *cache hit.*

hit-wait (38) The amount of time spent waiting for a cache hit, due to uncompleted I/O.

IBL (50) Local predictor: infinite-block lookahead. IBL predicts that every block following the most recent access will be used in the future.

ideal execution time (31) The absolute lower bound on the execution time, assuming no overhead and perfect load balance of the processors and disks.

interleaved (5) This refers to the way the blocks of the file are partitioned among the disks. The blocks are allocated to the disks in a round-robin fashion, the first block on the first disk, the next block on the second disk, and so on. This is a special case of *declustering*, and does not imply *striping.*

IOBL (53) Local predictor: hybrid of IBL and OBL. IOBL begins as IBL but then switches to OBL if a non-sequential access is detected.

IPORT (53) Local predictor: hybrid of IBL and PORT. IPORT begins as IBL but switches to PORT when a non-sequential access is detected.

IOPORT (53) Local predictor: hybrid of IBL, OBL, and PORT. IOPORT begins as IBL but switches to OBL when a non-sequential access is detected. If regular portions are detected, PORT is used.

jump-back (78) In a sequential pattern, the block numbers should be nondecreasing. A jump-back is the point where a process's pattern is decreasing, i.e., where a block number is less than the previous block number.

lfp (29) Local read-only access pattern (Local Fixed-length Portions). In this local pattern, the sequential portions have regular length and spacing (although at different places in the file for each process).

local access pattern (14) A local access pattern is a locally-sequential access pattern. That is, each process is independently reading the file in a sequential manner. It is represented as a set of per-process access patterns. Contrast with *global access pattern.*

local predictor A *predictor* for *local patterns.*

lrp (29) Local read-only access pattern (Local Random Portions). This local pattern uses portions of irregular (random) length and spacing. Portions may overlap by coincidence.

lw (29) Local read-only access pattern (Local Whole file). In this local sequential pattern, every process reads the entire file from beginning to end.

lw1 (29) Write-only access pattern. A single process writes the entire file from start to finish.

MaxDist A parameter to the PORT, IPORT, IOPORT, ADAPT, GAPS, and RGAPS predictors, that controls the maximum distance they will predict into the future.

MIMD (17) Multiple instruction stream, multiple data stream. A class of parallel architecture.

mirror (6) To store identical data on two independent disk drives. All writes are sent to both drives, and reads can be serviced by either drive. A mirrored disk is also called a *shadow* disk.

miss See *cache miss.*

mistake See *prefetch mistake.*

neighbor(x) (30) Synchronization style. Pairwise synchronization, where each processor synchronizes with its neighbor after reading x blocks.

none (30) Synchronization style. No explicit inter-process synchronization.

NONE (49) The name of the predictor that predicts nothing. Thus, it is the same as not prefetching.

NUMA (17) Non-Uniform Memory Access. A class of shared-memory architecture.

OBL (50) Local predictor: one-block lookahead. OBL predicts that block $i + 1$ will follow a reference to block i.

on-line predictor (49) A predictor required to predict the future in real time, based on the access pattern seen so far.

overrun (25) The amount of delay added to a process idle period by a prefetching action that took longer than the time available.

parallel, independent disks (6) Multiple disks attached to multiple processors so that the disks are completely independent, having separate controllers and paths to memory.

pattern See *access pattern.*

PFO (47) Prefetch For Others, a possible solution to the greedy-process problem.

PID See *parallel, independent disks.*

PPL (47) Private Prefetch Limits, a possible solution to the greedy-process problem.

PORT (50) Local predictor. PORT is able to track sequential portions, and use any regularity to predict the end of the current portion and possibly blocks in future portions.

portion (14) A contiguous set of blocks in the file. Although a single block is technically a portion, for prefetching purposes we do not usually consider it to be a portion.

predictor (23) A policy algorithm that analyzes the access pattern and predicts the future access pattern. It accepts requests from the prefetching code for a block number to be prefetched.

prefetch (2) To read a block from the disk into the cache before any part of the block is requested by the application.

prefetch limit (25) In our testbed, we limit the number of buffers in the cache holding blocks that have been prefetched but not yet used. This limit is the prefetch limit.

prefetch mistake (28) An incorrect prediction, or a block prefetched into the cache based on an incorrect prediction.

RAPID-Transit (21) Name of the testbed. RAPID stands for "Read-Ahead for Parallel Independent Disks".

read ahead See *prefetch.*

ready queue (24) A queue of free buffers that have no outstanding I/O activity, and are thus available for immediate replacement. Part of the *free list.* See *unready queue.*

record (13) The unit that is requested by the application from the file system. The record size is not necessarily the same as the block size.

replacement strategy (24) The algorithm used to select blocks for removal from the cache when a free buffer is needed for another block. Ours moves buffers to the *free list* when they leave the global *RU-set.*

RGAPS (85) Global predictor. This is similar to GAPS, except that it assumes the pattern is sequential unless it appears random.

rnd (29) Local and global access pattern. Accesses random blocks of the file.

RU-set (24) The recently-used set of blocks for either one process or all processes in the application.

seg (29) Local read-only access pattern (Segmented). In this local pattern, the file is divided into a non-overlapping set of contiguous segments, one per process. Each process thus has one sequential portion.

seg (29) Write-only access pattern. This pattern divides the file into segments, one per process, and each process writes its segment from start to finish.

self-scheduled access (12) When asynchronous processes access the file by atomically accessing the globally "next" record. The global interleaving of processes in the order is determined by dynamic run-time fluctuations.

sequential access pattern (14) An access pattern consisting of sequential portions.

sequential portion (14) A *portion* that is read or written sequentially. Note that a global access pattern using self-scheduled access only follows a loose sequential ordering.

shadow (6) See *mirror.*

striped (6) The blocks of the file are *interleaved* among the disks, and the disks are controlled by a single controller, which reads a block from all disks simultaneously. Each disk may contribute as little as one bit at a time. There are two varieties, depending on whether the disks are rotationally synchronous.

SWITCH (101) Local and global predictor. SWITCH watches the early stages of the access pattern, switching to either a local or global predictor as appropriate.

synchronization style (30) The mode of inter-process synchronization. One of *none, each(x), total(x),* and *neighbor(x).*

total(x) (30) Synchronization style. Barrier synchronization after reading x blocks total. Contrast with *each(x).*

unready queue (24) A queue of free buffers that have some outstanding I/O activity, either due to a disk write flushing dirty data to disk, or to a disk read from a prefetch mistake. Part of the *free list.* See *ready queue.* We also track the soonest time that a buffer in the queue is to become ready, so that we know when not to search the queue for ready buffers.

WriteBack (147) Write buffering policy. Delays the disk write until the buffer is needed for another block.

WriteFree (147) Write buffering policy. Issues a write when the buffer enters the free list. This is a compromise between WriteThru and WriteBack.

WriteFull (147) Write buffering policy. Issues the write when the buffer is "full."

WriteThru (147) Write buffering policy. Forces a disk write on every write request from the application.

Bibliography

[ABS] Jitendre Apte, Jack Briner, and Peter Suaris. Personal communication.

[AS89] Raymond K. Asbury and David S. Scott. FORTRAN I/O on the iPSC/2: Is there read after write? In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 129–132, 1989.

[BAC+90] Haran Boral, William Alexander, Larry Clay, George Copeland, Scott Danforth, Michael Franklin, Brian Hart, Marc Smith, and Patrick Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[BBN86] BBN Advanced Computers. The Butterfly RAMFile system. Technical Report 6351, BBN Advanced Computers, September 1986.

[BBN87] BBN Advanced Computers. *Butterfly Products Overview*, 1987.

[BBW86] Micah Beck, Dina Bitton, and W. Kevin Wilkinson. Design and evaluation of a parallel sort utility. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 934–941, 1986.

[BD83] H. Boral and D. DeWitt. Database machines: an idea whose time has passed? In *Proceedings of the Fourth International Workshop on Database Machines*, pages 166–187. Springer-Verlag, 1983.

[Ber90] David Bernholdt. Personal communication. University of Florida, February 1990.

[BG88] D. Bitton and J. Gray. Disk shadowing. In *14th International Conference on Very Large Data Bases*, pages 331–338, 1988.

[Bit89] Dina Bitton. Arm scheduling in shadowed disks. In *Proceedings of IEEE Compcon*, pages 132–136, Spring 1989.

[BKZS85] J.-L. Baer, S. C. Kwan, G. Zick, and T. Snyder. Parallel tag-distribution sort. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 854–861, 1985.

[BM81] B. T. Bennett and C. May. Improving performance of buffered DASD to which some references are sequential. *IBM Technical Disclosure Bulletin*, 24(3):1559–1562, August 1981.

[Boz89] G. P. Bozman. VM/XA SP2 minidisk cache. *IBM Systems Journal*, 28(1):165–174, 1989.

[BRW89] Andrew Braunstein, Mark Riley, and John Wilkes. Improving the efficiency of UNIX file buffer caches. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 71–82, December 1989.

[BS76] Jean-Loup Baer and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, March 1976.

[BSTY78] S. Berbec, A. Shibamiya, S. Togasaki, and H. Yoshida. Use of direct access storage devices by MVS customers — Guide survey results. In *Proceedings of the Guide 47 Conference*, pages 1121–1138, November 1978.

[Cab90] Luis-Felipe Cabrera. Technical summary of the second IEEE workshop on workstation operating systems. *ACM Operating Systems Review*, 24(3):7–21, July 1990.

[CGKP90] Peter Chen, Garth Gibson, Randy Katz, and David Patterson. An evaluation of redundant arrays of disks using an Amdahl 5890. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1990.

[Cla85] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180, December 1985.

[Cro88] Thomas W. Crockett. Specification of the operating system interface for parallel file organizations. Publication status unknown (ICASE technical report), 1988.

[Cro89] Thomas W. Crockett. File concepts for parallel I/O. In *Proceedings of Supercomputing '89*, pages 574–579, 1989.

[DEC89] Digital Equipment Corporation. *VAX Disk Striping Driver for VMS*, December 1989. Order Number AA-NY13A-TE.

[DGS+90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsaio, and Rick Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[DHS88] Monty M. Dennau, Peter H. Hochschild, and Gideon Schichman. The switching network of the TF-1 parallel supercomputer. *Supercomputing Magazine*, pages 7–10, Winter 1988.

[Dib90] Peter C. Dibble. *A Parallel Interleaved File System.* PhD thesis, University of Rochester, March 1990.

[Die89] Marge Dietz. Personal communication. Duke University, March 1989.

[DO86] Eliezer Dekel and Istvan Ozsvath. Parallel external merging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 921–923, 1986.

[DSE88] 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.

[FB76] P. A. Franaszek and B. T. Bennett. Adaptive variation of the transfer unit in a storage hierarchy. Technical Report RC-6310 (27098), IBM Yorktown, November 1976.

[FE89] Richard Allen Floyd and Carla Schlatter Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, June 1989.

[FE90] Richard A. Floyd and Carla Schlatter Ellis. Pushing the limits of transparency in distributed file systems. Technical Report CS-1991-06, Dept. of Computer Science, Duke University, December 1990.

[FH88] Robert J. Flynn and Haldun Hadimioglu. A distributed Hypercube file system. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1375–1381, 1988.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1, chapter 6 and 15. Prentice Hall, Englewood Cliffs, NJ, 1988.

[Flo86] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report 177, Dept. of Computer Science, Univ. of Rochester, March 1986.

[Flo89] Richard Allen Floyd. *Transparency in Distributed File Systems*. PhD thesis, University of Rochester, 1989.

[FP77] Donald E. Freeman and Olney R. Perry. *I/O Design: Data Management in Operating Systems*. Hayden Book Company, 1977.

[FPD91] James C. French, Terrence W. Pratt, and Mriganka Das. Performance measurement of a parallel input/output system for the Intel iPSC/2 hypercube. Technical Report IPC-TR-91-002, Institute for Parallel Computation, University of Virginia, 1991. Appeared in, Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems.

[GHK+89] Garth A. Gibson, Lisa Hellerstein, Richard M. Karp, Randy H. Katz, and David A. Patterson. Failure correction techniques for large disk arrays. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, April 1989.

[GLR83] Allan Gottlieb, B. D. Lubachevsky, and Larry Rudolph. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Transactions on Programming Languages and Systems*, 5(2):164–189, April 1983.

[GMS88] Hector Garcia-Molina and Kenneth Salem. The impact of disk striping on reliability. *IEEE Database Engineering Bulletin*, 11(1):26–39, March 1988.

[Gro85] C. P. Grossman. Cache-DASD storage for improving system performance. *IBM Systems Journal*, 24(3,4):316–334, 1985.

[Hai89] Chris Haight, 1989. Personal communication with engineer at Sequent Computer, Inc.

[Int88a] Intel beefs up its iPSC/2 supercomputer's I/O and memory capabilities. Electronics, November 1988.

[Int88b] iPSC/2 I/O facilities. Intel Corporation, 1988. Order number 280120-001.

[Int89] Concurrent I/O application examples. Intel Corporation Background Information, 1989.

[Jos70] M. Joseph. An analysis of paging and program behavior. *The Computer Journal*, 13(1):48–54, February 1970.

[KBK89] Bob Knighten, Joseph Boykin, and Terry Kelleher, 1989. Personal communication with engineers at Encore Computer, Inc.

[KD89] John P. Kearns and Samuel DeFazio. Diversity in database reference behavior. *ACM SIGMETRICS Performance Evaluation Review*, 17(1):11–19, May 1989.

[Kim86a] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Transactions on Computers*, C-35(11):978–988, November 1986.

[Kim86b] Michelle Y. Kim. *Synchronously Interleaved Disk Systems with their Application to the Very Large FFT*. PhD thesis, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, 1986. IBM Report number RC12372.

[Kon88] Alan J. Kondoff. The MPE XL data management system exploiting the HP Precision architecture for HP's next generation commercial computer systems. In *Proceedings of IEEE Compcon*, pages 152–155, Spring 1988.

[Kor90] Kim Korner. Intelligent caching for remote file service. In *Proceedings of the Tenth International Conference on Distributed Computer Systems*, pages 220–226, 1990.

[Lee87] Roland Lun Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, University of Illinois, May 1987. CSRD tech report number UILU-ENG-87-8005.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[LYL87] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, 1987.

[M.88] T. J. M. Now: Parallel storage to match parallel CPU power. *Electronics*, 61(12):112, December 1988.

[Man89] Tom Manuel. Breaking the data-rate logjam with arrays of small disk drives. *Electronics*, 62(2):97–100, February 1989.

[Mar88] David Michael Marcovitz. A multiprocessor cache performance metric. Technical Report UILU-ENG-88-8011, University of Illinois, August 1988.

[Mea89] Wes E. Meador. Disk array systems. In *Proceedings of IEEE Compcon*, pages 143–146, Spring 1989.

[MH88] Jai Menon and Mike Hartung. The IBM 3990 disk cache. In *Proceedings of IEEE Compcon*, pages 146–151, Spring 1988.

[Mok87] Nicholas Mokhoff. Parallel disk assembly packs 1.5 GBytes, runs at 4 MBytes/s. *Electronic Design*, pages 45–46, November 1987.

[Ng89] Spencer Ng. Some design issues of disk arrays. In *Proceedings of IEEE Compcon*, pages 137–142, Spring 1989. San Francisco, CA.

[NLS88] S. Ng, D. Lang, and R. Selinger. Trade-offs between devices and paths in achieving disk interleaving. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 196–201, 1988.

[NNI89] H. Nishino, S. Naka, and K Ikumi. High performance file system for supercomputing environment. In *Proceedings of Supercomputing '89*, pages 747–756, 1989.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[OCD+88] John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[OCH+85] 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.

[OD89] 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.

[O'L90] Bernard T. O'Lear. Pitfalls and triumphs of mass storage systems. Colloquium at North Carolina Supercomputing Center, October 1990.

[Pan89] Ricardo D. Pantazis. Personal communication. Duke University, March 1989.

[PFDJ89] Terrence W. Pratt, James C. French, Phillip M. Dickens, and Stanley A. Janet, Jr. A comparison of the architecture and performance of two parallel file systems. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 161–166, 1989.

[PGK88] 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.

[Pie89] 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.

[Pow77] Michael L. Powell. The DEMOS File System. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pages 33–42, November 1977.

[RB89a] A. Reddy and P. Banerjee. Evaluation of multiple-disk I/O systems. *IEEE Transactions on Computers*, 38:1680–1690, December 1989.

[RB89b] A. Reddy and P. Banerjee. An evaluation of multiple-disk I/O systems. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:315–322, 1989.

[RBA88] A. L. Reddy, P. Banerjee, and Santosh G. Abraham. I/O embedding in hypercubes. In *Proceedings of the 1988 International Conference on Parallel Processing*, volume 1, pages 331–338, 1988.

[RCCT90] Randall D. Rettberg, William R. Crowther, Philip P. Carvey, and Raymond S. Tomlinson. The Monarch Parallel Processor hardware design. *IEEE Computer*, 23(4):18–30, April 1990.

[Res90] Cray Research. DS-41 disk subsystem, 1990. Sales literature number MCFS-4-0790.

[RRR76] Niklaus Ragaz and Juan Rodriguez-Rosell. Empirical studies of storage management in a data base system. Technical Report RJ-1834 (26703), IBM San Jose, October 1976.

[RT78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 6(2):1905–1930, July-August 1978.

[SBN82] Daniel P. Siewiorek, C. Gordon Bell, and Allen Newell, editors. *Computer Structures: principles and examples*. McGraw-Hill, 1982.

[Sch88] Martin Schulze. Considerations in the design of a RAID prototype. Technical Report UCB/CSD 88/448, UC Berkeley, August 1988.

[SGK+85] Russell Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the 1985 Usenix Conference*, pages 119–130, 1985.

[SGM86] Kenneth Salem and Hector Garcia-Molina. Disk striping. In *IEEE 1986 Conference on Data Engineering*, pages 336–342, 1986.

[Smi78a] Alan Jay Smith. On the effectiveness of buffered and multiple arm disks. In *Proceedings of the 5th Annual International Symposium on Computer Architecture*, pages 242–248, 1978.

[Smi78b] Alan Jay Smith. Sequential program prefetching in memory heirarchies. *IEEE Computer*, pages 7–21, December 1978.

[Smi78c] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[Smi81a] Alan Jay Smith. Input/Output optimization and disk architectures: A survey. *Performance Evaluation*, 1(2):104–117, 1981.

[Smi81b] Alan Jay Smith. Optimization of I/O systems by cache disks and file migration: A summary. *Performance Evaluation*, 1(3):249–262, 1981.

[Smi82] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[Smi85a] Alan Jay Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 64–72, 1985.

[Smi85b] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[Sym89] Symult Systems, Monrovia, CA. *Programmer's Guide to the Series 2010 System*, first edition, March 24 1989.

[Tab90] David Taber. MetaDisk driver technical description. SunFlash electronic mailing list 22(9), October 1990.

[TCB78] D. Towsley, K. M. Chandy, and J. C. Browne. Models for parallel processing within programs: Application to CPU: I/O and I/O: I/O overlap. *Communications of the ACM*, 21(10):821–831, October 1978.

[Ter88] DBC/1012. Teradata Corporation Booklet, 1988.

[THY80] T. Tokunaga, Y. Hirai, and S. Yamamoto. Integrated disk cache system with file adaptive control. In *Proceedings of IEEE Compcon*, pages 412–416, Fall 1980.

[TMC87] Connection Machine model CM-2 technical summary. Technical Report HA87-4, Thinking Machines, April 1987.

[Tow78] Donald F. Towsley. The effects of CPU: I/O overlap in computer system configurations. In *Proceedings of the 5th Annual International Symposium on Computer Architecture*, pages 238–241, April 1978.

[TR88] Lewis W. Tucker and George G. Robertson. Architecture and applications of the Connection Machine. *IEEE Computer*, 21(8):26–38, August 1988.

[Tri76] K.S. Trivedi. Prepaging and applications to array algorithms. *IEEE Transactions on Computers*, C-25(9):915–921, September 1976.

[Tri77a] Kishor S. Trivedi. Prepaging and applications to the STAR-100 computer. In *Proceedings of the Symposium on High Speed Computer and Algorithm Organization*, pages 435–446, April 1977.

[Tri77b] Kishor S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, C-26(10):938–947, October 1977.

[Tri79] Kishor S. Trivedi. An analysis of prepaging. *Computing*, 22(3):191–210, 1979.

[Tri82] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Prentice-Hall, 1982.

[TvRvS+90] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

[WCM88] Andrew Witkowski, Kumar Chandrakumar, and Greg Macchio. Concurrent I/O system for the Hypercube multiprocessor. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 1398–1407, 1988.

[WSB+89] W. W. Wilcke, D. G. Shea, R. C. Booth, D. H. Brown, M. F. Giampapa, L. Huisman, G. R. Irwin, E. Ma, T. T. Murakami, F. T. Tong, P. R. Varker, and D. J. Zukowski. The IBM Victor multiprocessor project. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 201–207, 1989.

Biography

I was born in Ashton-upon-Mersey, Cheshire, England on July 14, 1964, and spent most of my childhood in Oneonta, New York. My high school diploma, received in 1982, is from Choate Rosemary Hall in Connecticut. I graduated *magna cum laude* with an A.B. in both Computer Science and Physics from Dartmouth College in 1986. While at Duke University, NSF twice awarded me honorable mention in their graduate fellowship competition. I was an MCNC graduate fellow my first year, and was awarded a DARPA/UMIACS Parallel Processing Assistantship for my final two years.

I have always loved outdoor activities, including hiking, backpacking, climbing, and skiing. In high school my backpacking interest expanded with winter expeditions in the Adirondack mountains. Much of my spare time at Dartmouth was spent with the Dartmouth Outing Club, playing in the mountains of Vermont and New Hampshire. I was leading hiking trips, building trails, skiing, repairing equipment, competing on the woodsmen's team, and directing Freshman Trips. I spent the summer of 1986 as a backcountry ranger in Olympic National Park. While at Duke University I have taken time to explore the Southern Appalachians, the islands of Fiji (where I managed to break my neck while bodysurfing), and the mountains of Washington State.