Informed Prefetching and Caching

Russel Hugo Patterson III

December 1997
CMU-CS-97-204

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree Doctor of Philosphy
in Electrical and Computer Engineering*

Thesis Committee:
Garth A. Gibson, Chair
Mahadev Satyanrayanan
Daniel P. Siewiorek
F. Roy Carlson, Quantum Corporation

This research was supported in part by Advanced Research Projects Agency contracts DABT63-93-C-0054 and N00174-96-0002, in part by the Data Storage Systems Center under National Science Foundation grant number ECD-8907068, in part by an IBM Graduate Fellowship, and in part by generous contributions from the member companies of the Parallel Data Consortium: Hewlett-Packard Laboratories, Symbios Logic, Data General, Compaq, IBM Corporation, Seagate Technology, EMC Corporation, Storage Technology Corporation, and Digital Equipment Corporation. The views and conclusions contained in this document are my own and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

Keywords: prefetching, caching, file systems, resource management, cache management, TIP, I/O, cost-benefit analysis, economic markets, disk arrays, RAID

For Lee Ann

Abstract

Disk arrays provide the raw storage throughput needed to balance rapidly increasing processor performance. Unfortunately, many important, I/O-intensive applications have serial I/O workloads that do not benefit from array parallelism. The performance of a single disk remains a bottleneck on overall performance for these applications. In this dissertation, I present aggressive, proactive mechanisms that tailor file-system resource management to the needs of I/O-intensive applications. In particular, I will show how to use application-disclosed access patterns (hints) to expose and exploit I/O parallelism, and to dynamically allocate file buffers among three competing demands: prefetching hinted blocks, caching hinted blocks for reuse, and caching recently used data for unhinted accesses. My approach estimates the impact of alternative buffer allocations on application elapsed time and applies run-time cost-benefit analysis to allocate buffers where they will have the greatest impact. I implemented TIP, an informed prefetching and caching manager, in the Digital UNIX operating system and measured its performance on a 175 MHz Digital Alpha workstation equipped with up to 10 disks running a range of applications. Informed prefetching on a ten-disk array reduces the wall-clock elapsed time of computational physics, text search, scientific visualization, relational database queries, speech recognition, and object linking by 10-84% with an average of 63%. On a single disk, where storage parallelism is unavailable and avoiding disk accesses is most beneficial, informed caching reduces the elapsed time of these same applications by up to 36% with an average of 13% compared to informed prefetching alone. Moreover, applied to multiprogrammed, I/O-intensive workloads, TIP increases overall throughput.

Acknowledgments

First, I would like to thank my advisor, Garth Gibson, for all the help, guidance, and advice he has given me. He has spent many long hours teaching me how to do systems research. Throughout, he has been a tireless advocate of excellence, and this document is much better for it. He has also provided a huge amount of support. I am particularly grateful for the way he anticipated my needs as I started work on cost-benefit resource management and ensured that all the other components, including benchmark applications, programming assistance, and a hardware platform, were in place when I needed them. He let me focus on what I could do best and, by providing what I could not, guaranteed the success of the project. Beyond that specific support, in founding the Parallel Data Lab, Garth has created a wonderful environment for systems research, and pulled together an impressive group of staff and students. I feel lucky to have been a member of the Lab. For all of this, and for being such a good sport when I couldn't resist teasing him, I thank him.

I would like to thank Satya for his help and advice throughout. Recently, I had the opportunity to review some notes of our early meetings. I now realize that at the time I only understood a fraction of our discussions. Still, some if it sank in. I thank him for helping me understand the key problem and develop a solution to it.

Thanks are also due to the other members of my thesis committee. I thank Rick Carlson for his patience as I finished up. I look forward to continuing to work with him at Quantum as I make the transition to the real world. I am very grateful to Dan Siewiorek who gallantly joined the committee at the last minute and read the dissertation on short notice.

I'd also like to thank the many faculty that provided early guidance and support including Dave Lambeth, Stan Charap, Onat Menzilcioglu, H.T. Kung, and Sandro Forin. They all patiently endured many long meetings with me and helped me find my way when I was a newcomer to the field of I/O systems research.

I am particularly indebted to Jim Zelenka for his tireless efforts debugging and improving the raw TIP code I threw over the wall to him during the crush to produce the SOSP paper. He has an uncanny ability to track down kernel bugs which I gave him many opportunities to exercise. I am fortunate to have both benefited and learned from this skill. I am also deeply grateful to Danner Stodolsky who helped optimize both the first and second TIP prototypes, annotated Sphinx and Gnuld to give hints, and spent many a sleepless night running experiments and helping in other ways to get the SOSP paper out the door. Both Danner and Jim helped with many constructive conversations about the TIP design and implementation. Many thanks to them both.

A number of other students also helped me in my research along the way and I would like to thank each of them. Eka Ginting annotated Postgres to give hints and helped with other aspects of the SOSP paper. Mark Holland set me up with RAIDSim which was the foundation of the simulation environment for TIPTOE. And, Jiawen Su investigated asynchronous name resolution in TIP and ported the first TIP prototype to Digital UNIX.

I would like particularly to thank Andrew Tomkins for taking up the problem of deep prefetching and becoming such an energetic collaborator on the TIP project. His work developing TIPTOE extended TIP in a significant and exciting way. Without him, the OSDI and Sigmetrics papers would never have existed, and the TIP project would not have been nearly as complete.

Writing a dissertation requires more than just research, and I am grateful to the many folks who helped me keep my senses of perspective and humor over the years. Peter Stout, besides answering my endless computer questions, provided a fine example of how it should be done. He also taught me how to drink coffee in large quantities. In the later years, David Steere and Brian Noble were always ready with a humorous diversion even when the slogging seemed slow and endless. Thanks too to Patty Mackiewicz who helped with many practical problems and always looked out for me. But, more importantly, she found a way to make me smile even in the face of adversity.

None of this would have been possible without my parents. I thank them for the many years of love and support that first prepared me to undertake this work and then so eased my way during the long travail.

Finally, I would like to thank Lee Ann who first encouraged me to go back to school and pursue an advanced degree. That pursuit turned into a task larger than either of us ever imagined and demanded sacrifices neither of us anticipated. Lee Ann generously made those sacrifices and supported me throughout. When times were good, she cheered and shared in my joy. When times were tough, she cheered me up and gave me strength. Lee Ann has been there for me in these and many other ways I'm sure I cannot even imagine. I am deeply and forever grateful. As a small sign of my appreciation, I dedicate this dissertation to you.

Hugo Patterson
Pittsburgh, Pennsylvania
December, 1997

Table of Contents

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction

"It is a poor craftsman who blames his tools."
— unknown master

Magnetic disk drives are marvels of modern engineering. Into one small, inexpensive package they pull together technologies from magnetics, aerodynamics, mechanical engineering, coding theory, material science, chemistry, manufacturing, integrated circuits, signal processing, control theory, electrical engineering, thermodynamics, and many more to store over a billion bits in a square inch of disk surface, any of which may be accessed in about 10 milliseconds. Disk drives are such effective data-storage devices that, in this data-hungry world, the aggregate disk-storage capacity shipped is growing at an average rate of 99% per year [IDC96]. And yet, disks are often the target of complaints that they are too slow and that too much time is wasted waiting for them.

Although disks are fast by a human time scale, they are mechanical devices and purely electronic processors can perform millions of operations in the time it takes them to complete a single access. Nevertheless, when organized into arrays [Salem86, Patterson88, Gibson92a], disks can overwhelm any processor with data. So, the problem is not that disks are inadequate to the task of supplying processors with data or storing the data they produce. The problem is that all too many processors are running software that insists on issuing a request and then waiting idly while a disk services it. This is a poor use of both processor and disks; besides idling the processor, it idles all but one disk in an array and so fails to take advantage of disk-array parallelism.

The solution is not to replace disks, it is to be smarter in using them. Workloads need to exploit array parallelism for storage throughput and mask access latency by initiating reads in advance and completing writes in the background. Master programmers could largely achieve these goals through careful use of existing batch and asynchronous-I/O interfaces. But, not all authors of important applications are also masters of systems programming. In this dissertation, I show how operating systems can take full advantage of secondary storage technology and free programmers from the burden of carefully orchestrating their disk accesses to achieve good performance.

Researchers have already shown that write buffering, which requires no programmer intervention, can mask write latency and accumulate multiple writes for parallelism. In a similar manner, aggressive parallel prefetching could mask latency and exploit array parallelism for reads if only the operating system knew what to prefetch. Unfortunately, no known technique can predict, without assistance from the application, enough accesses far enough in advance to guide such aggressive prefetching. Further, attempts to prefetch aggressively without reliable information can waste large amounts of disk and cache resources and they risk hurting, not helping, performance. However, no matter how random and unpredictable accesses may appear to the operating system, they are often quite predictable within the application.

I propose that applications take advantage of this predictability to disclose knowledge of future file reads in hints to the operating system which then uses them to guide aggressive parallel prefetching. In the short term, and for the purposes of this dissertation, programmers must annotate applications to give hints by hand. However, I will argue that such annotation is preferable to explicit programmer I/O management for both master and naive programmers. First, it is easier than adding either explicit parallelism or asynchronous accesses. Second, it only requires programmers to disclose information about how their applications behave and does not require programmers to posses intimate configuration and performance details of the system running the application to obtain good performance. Third, because these system-specific details vary from one system to the next whereas application behavior does not, hints are more portable than explicit I/O management. And fourth, explicit I/O ties down memory and disk resources whereas hints enhance global resource management. In the long term, I believe that compilers and other automatic tools will be able to generate hints without programmer intervention. Indeed, researchers have already demonstrated that this is possible for some applications [Mowry96].

With enough hints and a large enough array, parallel prefetching can virtually eliminate application stalls for access completion. But, hints enable other operating system I/O optimizations that can reduce the size and therefore cost of the array needed to achieve a given level of performance, or can improve performance when an array is not available. Specifically, informed by hints, an operating system can deliver four primary benefits:

1. informed caching to hold on to useful blocks and outperform LRU caching independent of prefetching;
2. informed clustering of multiple accesses into one larger access;
3. informed disk management that better schedules accesses to increase access efficiency; and, of course,
4. informed prefetching to parallelize the read request stream and mask access latency.

At the same time, not all accesses may be hinted. Some applications may not give hints about all of their reads, and some may not give any hints at all. These accesses depend on traditional LRU caching for performance. In fact, all of the above optimizations require cache buffers either to initiate disk accesses or to hold on to cached data. A system that takes advantage of hints for all of these optimizations while preserving an LRU cache for unhinted accesses needs a mechanism for allocating buffers among these competing demands for all of the processes running on the system.

It is my thesis that many important, I/O-bound applications can provide accurate hints about their future accesses, that operating system prefetching and caching according to these hints can substantially reduce application wall-clock elapsed time, and that run-time cost-benefit analysis can be the basis of effective resource management that balances the use of cache buffers for all of these competing demands.

To support these claims, I first survey in Chapter 2 the many approaches to improving I/O performance. I find that asynchrony to mask latency coupled with parallelism to exploit disk array throughput can provide the scalable I/O performance needed to balance rapidly increasing processor performance. I go on to argue that, for software engineering, application portability, and global resource management reasons, annotating applications to give hints is a good way to add asynchrony and parallelism to serial, I/O-intensive applications.

In Chapter 3, I argue for a specific kind of hints that disclose knowledge of future requests and contrast such hints with ones that give advice about what the operating system should do. After defining a disclosure hint interface, I develop three techniques for annotating applications to give hints. I go on to show how to apply these techniques to annotate a broad suite of six important I/O-intensive applications that includes: Davidson computational physics, XDataSlice 3D scientific visualization, Gnuld object code linker, Sphinx speech recognition, Agrep text search, and two queries to the Postgres relational database. This chapter proves the first claim of my thesis.

In Chapter 4, I develop a framework for resource management based on cost-benefit analysis that includes three key components: locally-computable estimates of the cost of ejecting a block from the cache and the benefit of using a buffer to initiate an I/O; a common currency for the expression of these estimates that ensures they are comparable at a global level; and an allocation algorithm that uses the estimates for prefetching and cache management. In a nutshell, the algorithm ejects the lowest-cost block and reallocates its buffer to fetch a block from disk when the estimated benefit of the fetch exceeds the estimated cost of the ejection.

In Chapter 5, I describe my implementation, called TIP, of cost-benefit resource management. In Chapter 6, I evaluate TIP performance and find that:

1. informed prefetching from an array can virtually eliminate stalls for hinted accesses and reduce application elapsed time by as much as 84%;

2. informed caching, clustering, and disk management are most beneficial when bandwidth is limited, for example on a single disk, and can reduce elapsed time by as much as 36% compared to prefetching alone; and,

3. the combination increases throughput for both single and multiprogrammed applications across array sizes.

The results show that TIP can use run-time cost-benefit analysis to manage prefetching and caching according to hints and substantially reduce elapsed time. Thus, the results prove the second and third claims of my thesis.

In Chapter 7, I explore TIP performance in greater depth and argue that the cost-benefit resource management framework is robust to changes in system parameters. I also point to opportunities to extend the framework and other areas for future work.

Together, these chapters present the following five primary contributions of this dissertation:

1. the identification of disclosure hints as a mechanism for communicating application knowledge about future requests to a lower level of software, and especially for communicating knowledge of future file accesses to the operating system;
2. three techniques for annotating applications with disclosure hints about their future, and a demonstration of their use to annotate six important, I/O-intensive application;
3. a framework for resource management based on run-time application of cost-benefit analysis;
4. estimates expressed in the common currency required by the cost-benefit framework for the benefits of prefetching and clustering, and the costs of ejecting a block that a hint indicates will be reused or that resides in the LRU queue; and,
5. an implementation, TIP, of the cost-benefit framework in a file buffer cache manager and a demonstration that this system reduces elapsed time for all six of the annotated applications.

Chapter 2

Asynchrony + Throughput = Low Latency

Because economic forces mandate a position for disk drives in the memory hierarchy of modern computer systems, the long mechanical access latencies for data stored on disks are a drag on the performance of I/O-intensive applications. Even high-performance disks have access latencies of 10 milliseconds or more. In that time, a modern processor can execute millions of instructions. And, as shown in Figure 2.1, the trend is for this performance disparity to grow, not shrink. The key implication, as Amdahl's Law tells us [Amdahl67], is that reductions in elapsed time due to increasing processor performance will ultimately be limited by the ever-larger portion of an application's elapsed time that will be spent waiting for disk accesses to complete. How can secondary storage performance be increased so that it doesn't become the bottleneck on overall performance?

Unfortunately, replacing disk drives in the storage hierarchy with something faster is not economically viable. Although disks are substantially slower than DRAM, they are also much cheaper. And, they are much faster than optical disk jukeboxes or tape libraries, although more expensive. Because the cost and performance differences between disk-drive storage and its neighbors above and below in the hierarchy are measured in orders of magnitude, the disk's position in the hierarchy is secure.

The growing performance disparity between processors and disks requires a solution to secondary storage performance that can scale with time. Redundant Arrays of Inexpensive Disks (RAID) were proposed to be just such a scalable solution [Patterson88, Gibson92a]. Because arrays of any size could be built, disk arrays do provide a scalable

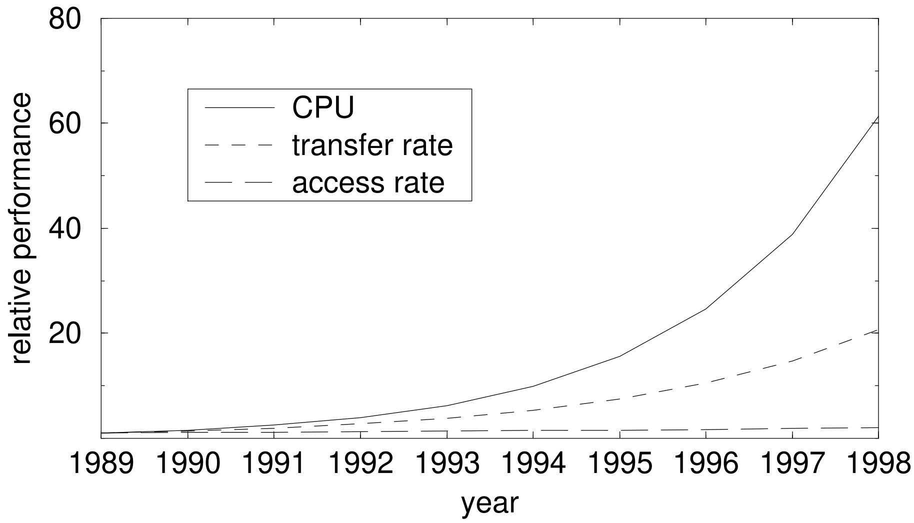


Figure 2.1. Processor and disk performance trends. Although disk data transfer rates are increasing at a dramatic 40% per year, mechanical constraints have limited the growth in the number of random accesses per second a disk can perform to about 8% per year [Grochowski96]. Neither is enough to keep up with CPU performance which is increasing at about 58% per year [Hennessy96]. Setting the relative performance of each to 1.0 in 1989, this graph shows how the difference in performance growth rates leads to a growing disparity between processor and disk performance. There is also a growing performance gap between disk transfer rates and disk access rates which makes it increasingly advantageous to transfer ever-larger chunks of data and to minimize seeks.

amount of raw secondary-storage throughput. Unfortunately, as shown in Figure 2.2, many applications do not take advantage of that potential throughput. Just as single-threaded programs cannot exploit the computing power of a parallel processor, so serial I/O workloads cannot in general exploit the I/O throughput of a disk array. Consequently, disk arrays will not be a complete solution until mechanisms are found for parallelizing serial I/O workloads to exploit array parallelism.

Parallelism is just one workload characteristic that affects the performance of disk-based secondary storage. For the purposes of this dissertation, I observe that disk arrays provide adequate, scalable performance, and ask the question: how can I/O workloads be improved to take full advantage of the hardware that already exists? In this dissertation, I argue that applications can disclose their future accesses in hints that the file system can use for informed prefetching, caching, clustering, and disk management. In this chapter, I explore the many possible alternative solutions and argue that my solution, if feasible, is preferred to all others. In the remaining chapters, I will show that my solution is in fact both feasible and effective.

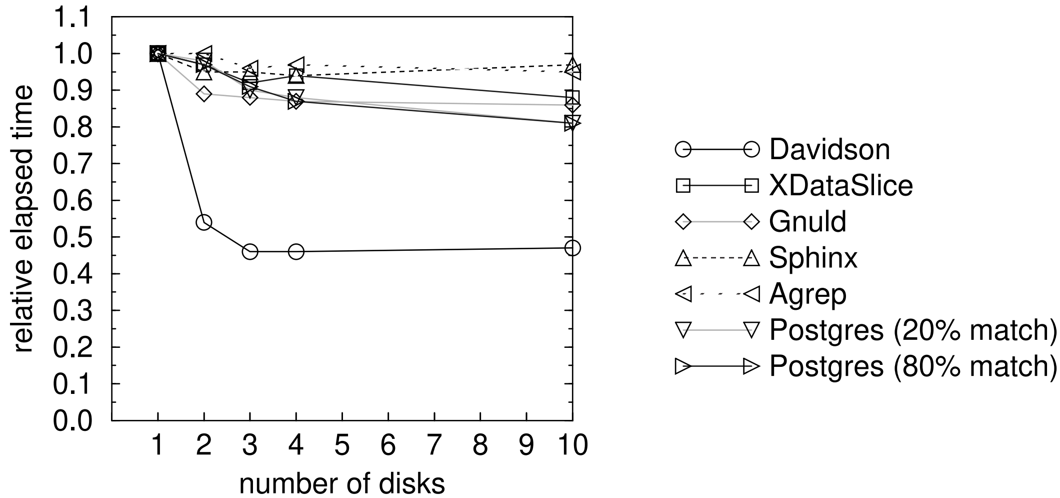


Figure 2.2. Elapsed time vs. array size. This graph shows elapsed time on multi-disk arrays as a fraction of elapsed time on a single disk for a suite of I/O-intensive applications that includes: Davidson computational physics, XDataSlice 3-D scientific visualization, Gnuld object code linker, Sphinx speech recognition, Agrep text search, and two queries to the Postgres relational database. These benchmarks are described in detail in Chapter 3. Davidson reads a 16-MByte dataset sequentially so sequential readahead can take advantage of the array for parallel transfer. But, none of the other applications obtains more than a 20% reduction in elapsed time from even a ten-disk array because their workloads are not parallel and so cannot exploit array parallelism.

I start this chapter with a review of disk hardware and the very particular performance characteristics of disks and disk arrays. I identify the ASAP virtues of disk workloads that increase performance: avoidance, sequentiality, asynchrony, and parallelism. I then survey existing techniques for enhancing each of the ASAP virtues and evaluate the potential for further enhancements that could lead to a long-term scalable improvement in secondary storage performance. I find that although increases in all four virtues are possible, asynchronous accesses coupled with parallelism in the form of aggressive prefetching offers the greatest opportunity to increase performance. Unfortunately, aggressive prefetching requires knowledge of what to prefetch. This fact leads me to propose application hints as a solution. I conclude this chapter with a discussion of the many advantages of this approach and a review of related work.

2.1 Disk drive performance characteristics

Disk drives consist of a stack of flat disks or platters coated with magnetic material. Each side of each disk has a dedicated read/write head mounted on an arm that swings back and forth over the surface to position the head on any one of the thousands of concentric tracks of data. Each track is divided into a number, perhaps 50 to 100, of sectors which each typically store 512 bytes of data. The sectors define the smallest unit of data that may be read or written. The file system running in the computer hosting the disk often organizes these sectors into larger units called blocks which typically contain 8 or 16 sectors each.

In most drives, there is one data channel that can read or write data via only one head at a time. The *n*th track on each surface are together called a cylinder. The arms for all surfaces move as a unit under the control of a single actuator so that all heads are positioned in the same cylinder. The configuration is very much like an audio turntable and an LP record except that there are several platters in a disk drive and disk tracks are concentric rings instead of a continuous spiral.

The time to access data is the sum of the time to move the head to the desired track (the seek time), the time for the requested data to rotate under the head (rotational delay), and the time to read the requested data off the disk which is the same as the time it takes for all the data to pass under the head (the transfer time). High-performance 3.5” disks today have an average seek time of 8.0 msec, an average rotational delay of 4.1 msec (half of a full rotation at 7200 rpm), and a transfer time for an 8 KByte block of 0.1 to 0.06 msec depending on whether the data is stored on the inner track, the outer track, or one in between.

The most important consequence of disk geometry is a large performance bias in favor of sequential over random accesses. Sequential accesses stream data from consecutive sectors on the disk surface and maximize utilization of the disk’s data channel. Random or non-sequential accesses suffer seek and rotational delays during which the channel transfers no useful data. Sequential access of 8 KByte blocks is about an order of magnitude faster than random access. The difference is greater for smaller blocks. Thus, repositioning delays, which reduce channel utilization, should be kept as short and as infrequent as possible.

The bias in favor of sequential access is increasing. Channel data rates, which are a function of the rotational speed of the disk and the linear bit density within a track, are limited only by the speed of the channel electronics which is increasing about 40% per year [Grochowski96]. In contrast, accesses per second, which are largely determined by mechanical constraints, are increasing at a rate of only about 8% per year

[Grochowski96]. Figure 2.1 shows the impact of these different growth rates on the relative performance of sequential vs. random accesses. Even sequential access is not increasing as fast as processor performance, but for actual disk performance to keep pace with even the increase in channel rates, workloads will have to become ever more sequential.

The advent of redundant disk arrays (RAID) [Patterson88, Gibson92a, RAB96], has added a new dimension to I/O subsystem performance, namely parallelism. In disk arrays, data is striped across the disks in the array by first grouping disk blocks into stripe units, which are typically about 64 KBytes in size, and then assigning these stripe units to physical disks in a round-robin fashion. The stripes assigned in one round over the disks are together called a stripe. Redundancy is most often achieved by reserving one stripe unit in each stripe to store the bit-by-bit XOR or parity of the other stripe units in the same stripe. When a stripe unit is lost, its data is the XOR of the remaining stripe units. There are many subtle variations on this theme that have different reliability and performance characteristics. In general, however, taking maximal advantage of the multiple disks in an array requires that all disks be utilized simultaneously. To first approximation, if only a quarter of the disks are utilized at a time then the subsystem is delivering only a quarter of its potential bandwidth.

One special optimization results from the way redundant information is maintained in most arrays. To update the parity when new data is written to a single stripe unit, the old data must be read, XOR'd with the new data, the result XOR'd with the old parity, and the resulting new parity written. Thus, a single write turns into two reads and two writes, albeit on two different disks. However, if an entire stripe is written at once, then the old data and parity need not be read; it is sufficient to compute the all new parity across the new data. Instead of four operations per stripe unit, this only requires one write of each data stripe unit plus a write of the parity stripe unit. Most often, blocks across a stripe are sequentially numbered, so writes that span a full stripe take advantage of this large-write optimization. Thus, the advent of RAID increases the desirability of large sequential writes which can span an entire stripe.

Array utilization is as workload-dependent as channel utilization. Workloads that consist of large requests for a substantial fraction of a full stripe across the array successfully utilize most of the available bandwidth with parallel transfer of data from multiple disks in the array. Scientific applications processing large datasets often have such a workload. On the other hand, workloads that consist of multiple, independent, concurrent requests can also achieve high array utilization. Because data is striped across the array, independent accesses are likely to be randomly distributed over the disks in the array, and so a large number of outstanding requests is likely to utilize a large number of disks in an array. Transaction processing systems that are concurrently processing many independent transactions often have such a concurrent workload. The problem, as shown in Figure 2.2, is that many I/O-intensive applications are written as single-threaded programs that never have more than one modest-sized request outstanding at the I/O subsystem. Such workloads rarely utilize more than one disk at a time.

As an aside, it is interesting to note that this same bias in favor of sequential access is beginning to appear at the next higher level of the memory hierarchy as well. Dynamic RAM access latencies have been decreasing only slowly. Meanwhile, new access techniques such as burst transfers and RAMbus have substantially increased sequential performance. Furthermore, memory interleaving emphasizes parallel transfer from main memory devices just as disk arrays do from secondary storage devices. It will be interesting to see how many of the techniques for improving disk performance will be applied to primary storage.

2.2 ASAP: the four virtues for I/O workloads

Many system components from the application down through software libraries, file systems, device interfaces, and the firmware in array controllers and storage devices themselves affect the disk operations performed and therefore the performance obtained. Nevertheless, the performance characteristics of disk drives dictate that there are only four fundamental ways to change workloads to improve secondary storage performance: avoid disk accesses, increase access sequentiality, perform accesses asynchronously, and access more disks in parallel. The more a workload applies these strategies the more it possesses the four ASAP virtues for I/O workloads.

1. **Avoidance**. Data needs satisfied without a disk accesses are unaffected by disk performance. As a secondary effect, reducing the number of accesses required to transfer a given amount of data reduces both host and drive CPU overheads for request-

processing which may itself reduce application elapsed time. File buffer caches are a common mechanism for avoiding accesses.

2. **Sequentiality**. Sequential accesses maximize disk data channel utilization. Next best is to minimize seek and rotational delays and therefore the time that data channel is not utilized. For example, the Berkeley Fast File System (FFS) stores files from the same directory in the same group of neighboring cylinders to reduce seek distances [McKusick84].

3. **Asynchrony**. Asynchronous accesses mask disk latency by allowing computation to continue while disk operations complete. Buffered writes and readahead are common examples of such asynchronous accesses.

4. **Parallelism**. Taking advantage of multiple disks through parallel transfer or multiple concurrent requests is a relatively recent innovation. Beyond disk-array architecture itself, relatively little effort has been devoted to increasing the parallelism of disk workloads. An exception is the Log-Structured File System (LFS) [Rosenblum92] which organizes multiple small writes into large writes that can take advantage of parallel transfer. One goal of this dissertation is to explore informed prefetching as a technique for taking advantage of array parallelism for reads.

Although these are the only virtues, there are many ways to add them to workloads. In the next four subsections, I survey some of the mechanisms that have been proposed with an eye towards identifying the best opportunities for further improvement.

2.2.1 Avoiding accesses avoids latency

Access avoidance can start with the applications themselves. Restructuring programs and reorganizing file data so that data accessed together are stored together can substantially reduce the number of accesses. Techniques such as blocking and tiling regular data structures have been developed to store data for more efficient access [McKellar69, Wolfe96].

When applications do need to read data, the system can have an impact on how many accesses it takes to satisfy those requests. Organizing the data on the disk so that multiple small accesses may be replaced with fewer, larger accesses both avoids some accesses and increases sequentiality (which will be discussed more below). Even if the same number of bytes is transferred in the two cases, reducing the number of accesses reduces CPU access overhead and request-processing delays at the drive. A simple technique for accomplishing this is to group file data into larger blocks [McKusick84] which reduces the number of accesses when application data requests have high spacial locality, that is, when applications are likely to access data logically near data recently accessed in the same file. Similarly, reading multiple blocks in a single cluster or extent [Peacock88, McVoy91] can reduce the number of accesses.

Another approach is to focus on avoiding metadata accesses which can be a substantial portion of the total workload, especially when there are accesses to many small files. For example, the Log-Structured File System (LFS) avoids some synchronous metadata writes by appending new data followed by the updated metadata [Rosenblum92]. Because the appends are sequential, LFS also enables more clustered writes and even the RAID 5 large-write optimization. Greg Ganger has shown how to embed inodes in directories to avoid some metadata accesses and how to co-locate multiple small files in the same directory with their metadata so that a single access can read all the metadata and user data for multiple files [Ganger97]. Again, these techniques also increase workload sequentiality.

On the write side, buffering written data and delaying the write to disk can avoid some write accesses because data may be over-written or deleted before the disk writes occur [Baker91, Kistler93]. This is particularly useful when an application is appending data to a file with small writes. Because file systems usually write entire blocks as a unit even when only a few bytes are dirty, coalescing multiple, small, application writes into a single full-block disk write can decrease the number of bytes written as well as the number of accesses. Buffering writes is also useful for avoiding writes altogether for short-lived, temporary files that are soon deleted. But, the primary benefit of buffering writes, as will be discussed shortly, is that it adds asynchrony to write accesses.

The most common and direct mechanism for avoiding disk accesses is caching which holds data in memory for fast access if needed in the future. When a cache manager chooses to continue holding some block in preference to another that it ejects, it is effectively predicting that the held block will be accessed before the ejected one. Thus, cache management is a game of predicting the future. Most current systems use a simple, history-based mechanism for choosing what to cache called the Least-Recently-Used (LRU) algorithm [Mattson70]. When a buffer is needed, LRU ejects the block that has not been accessed for the longest time. This heuristic has proven quite effective for general workloads which often have high temporal locality, that is, for which recently accessed blocks tend to be reaccessed again soon.

There are access patterns, such as repeated sequential access to a file larger than the cache, for which LRU fails to cache any blocks. In contrast, the Most-Recently-Used (MRU) algorithm, which ejects the block just accessed, takes full advantage of the cache for repeated accesses. Other workloads only access data once and would not benefit from caching no matter which algorithm is used. Some researchers have explored ways to identify different sets of blocks that will be accessed with different patterns and then to apply the appropriate algorithm to each set. For example, Kim Korner proposed analyzing traces of file activity to correlate programs with their access patterns to files in particular directories or to files whose names end with a particular extension [Korner90]. For example, an assembler typically reads a file in the '/tmp' directory and then deletes it, so there is no point in caching any blocks that it reads from that directory. On the other hand, the C language preprocessor reads files ending in '.h' sequentially, and these files tend to be reaccessed by the C preprocessor when the next file is compiled, so these files should be cached with the MRU algorithm. Korner's approach was to analyze traces off-line to generate access pattern rules which are then given as hints to a remote file server, although it is not hard to imagine using such hints in the local cache as well.

More recently, Pei Cao *et al.* proposed that applications generate their own hints or advice about which caching policies to apply to which blocks [Cao94, Cao94a]. They showed how to apply the LRU algorithm at a global level to allocate buffers among processes while allowing each process to specify caching polices for the blocks in its partition. In subsequent work, she and collaborators proposed using application hints for prefetching as well [Cao95, Cao96]. This work is closely related to the work described in this dissertation and is discussed in more detail in Section 2.4.

Caches are crucial to the performance of modern systems. Hints about future requests, whether inferred or explicit, can make them even more effective. Could caches be made so effective that they could permanently relieve the I/O bottleneck? At one time, it appeared

	1985 BSD study						1991 study
cache size	390 KB	1 MB	2 MB	4 MB	8 MB	16 MB	7 MB
miss ratio	49.2%	36.6%	31.2%	28.0%	26.2%	25.0%	41.4%

Table 2.1. Comparison of caching performance in 1985 and 1991. The numbers in this table are drawn from [Ousterhout85] and [Baker91]. The 1985 tracing study of the UNIX 4.2 BSD file system predicted cache performance for a range of cache sizes assuming a 30 second flush back policy for writes. The 1991 study measured cache performance on a number of workstations running Sprite. The cache size varied dynamically, but averaged 7 MBytes. The diminishing returns from increasing cache size are evident in the 1985 results. Also striking is the difference between the predicted and measured performance of a large cache.

that by increasing cache size, caches could virtually eliminate slow synchronous data reads [Ousterhout89]. But, for caches to compensate for the growing disparity between processor and disk performance, their miss ratios will have to drop proportionately so that an ever-smaller proportion of data accesses actually suffer the full latency of a disk read. Is such improvement likely?

Table 2.1 compares the performance predicted for a variety of operating system file cache sizes in 1985 [Ousterhout85] with that observed in 1991 [Baker91] by a group at Berkeley. The first observation, based on the 1985 data, is that increasing the size of an already large cache does not reduce the miss ratio much. Some data sets just don't cache well, either because they are too large or because they are accessed infrequently, or only once. In fact, the situation gets worse through time as the 1991 data shows. Because data sets are growing, it is necessary to increase the size of the cache just to maintain miss ratios. This is common experience for anyone who has upgraded to a new version of some software package; there is an ever-increasing amount of data to be processed. Furthermore, so much data is accessed infrequently, or even only once, that even a large, optimally-managed cache would not completely compensate for high disk-access latencies. Essentially, caches would have to become comparable in size to secondary storage for them to eliminate I/O performance as a bottleneck.

2.2.2 Increasing sequentiality increases channel utilization

A fully sequential workload optimizes disk performance because it has 100% channel utilization (the channel is transferring data for 100% of the disk service time). The closer workloads can come to this ideal, the less time they will waste on positioning delays, and the greater the disk performance they will achieve.

There are only two basic techniques for increasing workload sequentiality. First, file blocks may be allocated to physical disk blocks in such a way that file accesses end up being sequential on the disk. Alternatively, taking block allocation as a given, requests may be re-ordered to maximize sequentiality. In practice, there are four ways to apply these basic techniques: storing data so that both the write and subsequent reads are largely sequential; profiling data usage patterns and then moving the data so that subsequent accesses have higher sequentiality; taking allocation as a given, but trying to schedule accesses to maximize sequentiality; and, two-stage techniques in which, data is written with high sequentiality now, and later asynchronously moved to a permanent location. I will discuss these approaches in turn.

Because many applications access files sequentially, a common policy is to write logically contiguous blocks sequentially on disk. Achieving such sequentiality can be difficult in practice because storage is continually allocated and de-allocated in varying size chunks which eventually fragments available storage into short sequential runs which make allocating new, long sequential runs impossible. To put a lower bound on the length of an individual sequential run, many systems allocate space in fixed sized blocks. If less than a whole block is needed, the unused portion of the block is left empty. Larger blocks ensure greater sequentiality (less external fragmentation) at the cost of more wasted space (internal fragmentation). Smaller blocks make the opposite trade-off. The Berkeley FFS tried to have the best of both worlds by subdividing whole file blocks into fragments when needed to recapture lost space [McKusick84]. An alternative is the dynamic solution adopted by Microsoft for its MS-DOS file system: periodically run a defragmentation utility that moves blocks around to form new, long sequential runs of free space [Microsoft93].

Larger blocks increase sequentiality for reasons beyond a reduction in fragmentation. At first glance, it may appear that sequential accesses to a large number of small blocks and to a small number of large blocks would have the same sequentiality. But, for writes, the smallest delay between requests can cause the disk to miss the beginning of the next sector which adds the latency of a full rotation of the disk, perhaps 10 msec, to the service time for the request. Reads do not suffer this problem on modern disks which perform sequential readahead into buffers internal to the drive. However, when other accesses are interleaved with either reads or writes, larger blocks and clusters of blocks cause the unit of interleaving to be larger, which reduces the number of seeks among the interleaved access streams, and thereby increases the sequentiality of the workload experienced by the disk. Consequently, mechanisms such as LFS and Ganger's embedded inodes and grouped files which facilitate larger and clustered accesses also serve to increase the sequentiality of disk accesses.

Files are often accessed sequentiality, but multiple files in the same directory are also often accessed together. When space is available, FFS stores files in the same directory in the same disk neighborhood known as a cylinder group. This does not guarantee sequentiality, but it does decrease seek distances when multiple files are accessed. Grouping files goes further by tying to achieve true sequentiality for multiple small files.

The file system does not have a monopoly on allocating storage space. The SCSI disk interface provides a layer of abstraction that the disk can exploit to reassign logical blocks to any physical block it chooses. One approach is to use a greedy algorithm that assigns the free block with lowest access latency given the current head position to blocks as they are written [Ruemmler91]. Unfortunately, this approach can leave the data in non-sequential locations which can reduce physical sequentiality for reads.

The Logical Disk interface extends SCSI's linear block address space to a two-dimensional space of a meta-list of lists of blocks [de Jonge93]. When the file system assigns blocks to the same list, it effectively is giving the disk a hint that the blocks are likely to be accessed sequentially. When it puts lists near each other in the meta-list, it is indicating that accesses to the two lists may be correlated in time. Effectively, this two-dimension structure abstracts the two-dimensional access performance of disks: sequential blocks are accessed most quickly; after that, blocks located near each other will have lower latency than blocks further apart.

Recent work on Network-Attached Secure Disks (NASD) [Gibson97a] further raises the interface so that disks export objects, not just a space of blocks. This provides the disk with even more information than the two-dimensional block lists: the disk manages its own metadata, and knows what blocks are unused and may be reallocated. This new interface should enable new allocation optimizations at the drive level without requiring operating system changes.

Block allocation is not necessarily static. I already mentioned defragmentation utilities that move blocks around to increase sequentiality. But such defraggers are almost an afterthought. Some researchers have explicitly regarded block allocation as a two-stage process: allocate short-term storage for high sequentiality and low latency now, and later move the data to free space and/or minimize latency for anticipated future access. LFS could be viewed as falling in this category with the log providing low-latency, sequential writes in the short term, and the segment cleaner performing the second-stage reallocation to a long-term home. There are many other examples, although most are implemented below the file system in the storage subsystem and are aimed at avoiding the high latency parity-update reads and writes. Examples include HP Autoraid which initially writes to mirrored storage and later migrates data in large chunks to a RAID 5 arrays to reduce the space overhead of redundancy information and provide higher, parallel bandwidth for subsequent reads [Wilkes96]. Parity-logging is a technique for initially logging parity updates with sequential writes and later truncating the log, applying the updates to a RAID 5 array, but taking advantage of multiple writes to the same region of the array to avoid multiple overwrites, to coalesce neighboring writes to achieve the large-write optimization, or just to increase the locality and therefore decrease the latency of more isolated updates [Stodolsky93]. Other researchers have looked at dynamically building new stripes to avoid parity updates [Mogi94].

Most of the foregoing techniques use static policies to govern storage reallocation. Autoraid goes a step further and allocates mirrored or RAID 5 storage depending on the rate of updates to the blocks. But, there are a number of techniques for reallocating blocks, not according to static policies, but according to dynamic usage patterns. For example, systems can take advantage of the fact that the distribution of accesses tends to be highly skewed to a small portion of the data stored on a disk. By profiling data accesses, disk subsystems [Vongsathorn90, Akyürek93, Akyürek93a] or file systems [Staelin90] can migrate or replicate [Akyürek92] the most active blocks to the center of the disk to reduce seek distances.

The second approach to increasing sequentiality is to reorder accesses to reduce positioning delays. Scheduling requests to minimize average access time is itself an old and well-developed field of study [Denning67, Geist87, Seltzer90, Jacobson91, Worthington94]. But such scheduling techniques are only applicable if there are multiple requests to schedule. If there is only one request outstanding at the drive, there is little that can be done to reduce the service time for that request. Effectively, the more work outstanding at a disk, the greater the opportunity for efficient scheduling and the higher the effective sequentiality or locality of the resulting workload for the disk arm.

How, then, can the system generate more outstanding requests and so improve the scheduling opportunities? On the write side, multiple requests may be buffered for asynchronous writes which may be scheduled in any order. FFS does this when it buffers data for up to 30 seconds as described above. Some have advocated buffering thousands of writes [Seltzer90]. The problem comes on the read side. Because many applications only issue one read request at a time, there is often no opportunity to schedule reads. One solution is issuing prefetches of additional blocks along with the read of the requested block. In this way, asynchrony may be used to increase workload sequentiality. The question becomes, how to generate these asynchronous requests. I will shortly address this question when I take up the general discussion of asynchrony as a mechanism for increasing I/O performance.

Increasing workload sequentiality increases disk channel utilization and therefore disk performance. It can even increase array utilization if accesses are sequential enough to take advantage of parallel transfer from the many disks in an array. But, can increasing sequentiality provide the parallelism needed to relieve the I/O bottleneck? The Log-Structured File System could, with an appropriately sized write buffer, make nearly all writes sequential. But, reads and writes don't necessarily occur in the same order which implies that data would have to be reallocated between the write and the subsequent read. And, when the same data is read in a different order on different occasions, the data would have to be reallocated between reads. Sometimes, as in the case of application launches mentioned above, it is possible to do this reorganization. But, in general, there may not be enough time to reallocate the data even if it were known in what order to lay the data out on disk. It is not possible to make all accesses, both reads and writes, sequential. A mechanism is needed to convert serial, non-sequential accesses into parallel accesses that can take advantage of array parallelism.

2.2.3 Asynchrony masks latency

Asynchrony can mask long disk latencies. If an application doesn't have to wait for disk accesses to complete then it doesn't matter that the accesses have high latency.

Buffered write-behind is an effective means for adding asynchrony to the write workload. Written data are temporarily stored in main memory, the application continues processing, and data are flushed to disk in the background. A pitfall of this approach is that the data are not in persistent storage when the application continues and so may be lost in the event of a failure. But, battery-backed RAM or uninterruptable power supplies can protect from data loss due to power failure, and write-protecting the data cache can protect against software failures such as operating-system scribble bugs and crashes as shown by the Rio file system [Chen96]. For maximal security, the write buffer could be constructed with a solid-state disk made of flash memory.

Prefetching data into the cache is the read-equivalent of write buffering. In its simplest form, file-system prefetching is based on the prevalence of sequential file access. Large file blocks implicitly prefetch unrequested data in the latter portions of the block. More explicitly, the file system can "readahead" sequential blocks of a file. But, because not all accesses are sequential, and because it can hurt performance to prefetch unused data, it is advantageous to scale the depth of prefetching according to the length of a run of sequential accesses [Smith78, Smith85]. In practice, SunOS prefetches one block ahead when the last two blocks referenced were sequential, or for clustered I/Os, it prefetches the next cluster when the last cluster was read sequentially [McVoy91]. Digital UNIX takes a more aggressive approach and prefetches ahead roughly the same number of blocks that have been read sequentially up to a maximum of 8 clusters of 8 blocks.

David Kotz has looked at detecting and prefetching for strided access patterns within a file [Kotz90, Kotz91, Kotz93]. This work primarily focussed on the parallel computing domain, and there is more about it in the next section.

Many researchers have explored ways to discover access patterns among files. For example Griffieon and Appleton observe the sequence of files opened and build a probability graph that records how often file B is referenced soon after file A [Griffioen93, Griffioen94, Griffioen95, Griffioen96]. Then, when A is referenced, if the likelihood that B will be referenced is above a threshold, the system prefetches file B. The system also prefetches any other files that are accessed soon after A with a frequency above the threshold. Using this technique, they were able to initiate prefetches for many files in advance of their use.

Duchamp and collaborators observe the sequence of files, including other programs, that a program accesses during the course of a single run [Tait91, Lei97]. They store the pattern in an access tree. Over time, the system may build up multiple pattern trees for each program. When the program is later run again, its sequence of accesses is compared to the ones stored in the access trees for the program. If a matching tree is found, then the system prefetches the first block of later files in the tree. The system relies on sequential readahead to prefetch the rest of the blocks in the file. The multiple access trees allow the systems to distinguish different patterns of use for the same files and prefetch for the currently occurring pattern.

Kroeger and Long have explored using data compression techniques to discover frequently occurring sequences of file references [Kroeger96]. When the current sequence matches one or more prior sequences, the system prefetches the next file in each sequence whose frequency of occurrence is above a threshold.

The idea of using compression techniques for prefetching was first proposed by Vitter and Krishnan [Vitter91]. They and Curewitz applied the approach to page references in an object-oriented database [Curewitz93]. Palmer and Zdonik have also explored pattern matching for database references [Palmer90, Palmer91]. But, instead of using compression algorithms, they use an associative memory to find close matches to the current sequence of accesses.

All of these approaches to prefetching have the attractive advantage of being transparent to the user. The system simply observes accesses and, predicated on the idea that past access patterns are being repeated, prefetches for the current access pattern. The drawback of such transparent approaches is that because the predictions are not completely accurate, prefetches based on them cannot be too aggressive or else the performance penalty when the predictions are incorrect will be too high. Consequently, most tend to be conservative and therefore can't be scaled up to compensate for the growing processor-disk performance disparity. Furthermore, many applications have access patterns that appear random or that touch data only once, and for which such heuristic techniques are ineffective.

Instead of relying on the file system to guess what to prefetch, programmers can explicitly prefetch data with asynchronous I/O calls. In doing so, programmers take on the responsibility of determining how far in advance to issue requests and how many to issue at a time. And, they must manage the requests and the buffers they use. Thus, using asynchronous I/O can require substantial programmer effort.

Intermediate between these two, applications can give hints about blocks they will access in the future [Gibson92, Patterson94, Patterson95, Cao96]. In some cases, compilers can generate such hints automatically [Mowry96]. This approach is the focus of this dissertation and will be discussed in some depth shortly.

Can asynchrony continue to mask disk latency even as processor performance continues to increase? For it to do so, three issues must be addressed. First, asynchrony is only effective if backed by sufficient throughput. Buffering writes only frees an application to continue as long as there are free buffers. An application that fills buffers faster than the storage subsystem can empty them will eventually run out of empty buffers and stall while dirty data are flushed to disk. Similarly, asynchronous reads and prefetching allow an application to read data without stalling only if disk reads complete before the application requests the data. If the application consumes data more quickly than the storage subsystem can deliver it, the application will eventually stall while data are fetched from disk. Thus, asynchrony decouples application elapsed time from disk latency, but only if it is paired with sufficient throughput to satisfy application demands. Or, conversely, asynchrony leverages throughput to mask latency.

The second issue for scaling asynchrony is the number of staging buffers. For writes, there must be enough to hold the written data while they are being flushed to disk. As processor performance increases, processors will be able to write more data in the time it takes a disk write to complete, and therefore more buffers will be needed. If disk arrays are used to provide the necessary storage throughput, then, conceptually, the multiple disks in the array can be used to empty the multiple buffers concurrently. Roughly speaking, there need to be enough buffers to keep enough disks utilized to provide enough storage throughput to balance application throughput. A similar argument applies to read buffers: there must be enough of them to satisfy data demands while a fetch completes. Thus, the number of buffers scales with the size of array needed and therefore with the size of the performance gap between processors and disks. A system that can afford the disks should be able to afford the much less expensive buffers, so buffer cost is not a barrier to scaling asynchrony.

The third issue in scaling asynchrony is determining what data should be put in these staging buffers. The writes themselves dictate what to put in the write buffers, but, there is no equivalent oracle for reads. As I/O latencies grow in terms of processor cycles, asynchronous fetches and prefetches must begin ever farther in advance if they are to complete in time. When the processing time between requests is less than the latency of a disk access, reads must be initiated several requests in advance to completely mask their latency. If all accesses are initiated multiple accesses in advance, then there are necessarily multiple outstanding fetches. Scaling asynchrony for writes is a simple matter of scaling the write buffer size. Scaling asynchrony for reads implies scaling the number and timing of the asynchronous fetches.

Scaling the number of asynchronous reads is not trivial. For asynchronous I/O, it could mean additional programmer effort to retune the application for each new faster processor. For heuristic prefetching, the farther in advance predictions are made, the more likely they are to be inaccurate. As ever-greater resources are devoted to prefetching, the risk of hurting, not helping performance increases. For prefetching to be the long-term solution sought, prefetching will have to become much more aggressive and more accurate. Prefetching has the needed scalable potential, but it requires much more accurate predictions of future accesses.

2.2.4 Parallelizing I/O workloads increases array utilization

Organizing disks into arrays is a fundamentally scalable approach to increasing secondary-storage throughput. By scaling up array size, arrays can, in principle, provide whatever throughput is required. The problem is that this throughput is only available if the workload itself has sufficient parallelism to utilize the multiple disks.

As mentioned above, scientific workloads that request large chunks of data, and therefore have highly sequential workloads, can take advantage of parallel transfer from the array. At the other extreme, highly concurrent workloads such as transaction processing can take advantage of arrays for concurrent servicing of multiple small requests. But, many I/O-intensive applications neither make large requests, nor are highly concurrent and therefore cannot fully utilize disk-array parallelism. Instead, their workloads consist of a serial stream of moderately-sized requests that only utilize a single disk in an array at a time. The array's potential throughput remains untapped and the access latency for these individual disk accesses dominates I/O service time. For such workloads, it is as if the I/O subsystem had only a single disk.

The techniques for scaling asynchrony are also effective for generating workload parallelism. Larger numbers of write-behind buffers increase the parallelism of asynchronous writes. Both aggressive prefetching and larger numbers of concurrent asynchronous I/O requests increase workload parallelism. But, as larger arrays are needed to supply the data needs of a single processor, these techniques must be ever more aggressively applied to maintain array utilization.

Batch or vector requests provide explicit parallelism with a single system call. For example, Cray's UNICOS operating system supports a `listio` system call that initiates a list of distinct I/O requests [Cray93]. But, many applications are not written to support batch or vector processing. For these applications, taking advantage of these calls would require many of the same code modifications needed to support asynchronous I/O.

There is a substantial push in the parallel computing community to support parallel file systems and I/O [Dibble88, Cao93, del Rosario94, Kotz94, Krieger94, Corbett95, Corbett96, Haskin96]. There has also been some effort in this direction in the distributed domain [Cabrera91, Hartman93, Lee96, Gibson97, Thekkath97]. But, parallel and distributed computing is not the focus of this dissertation. Certainly, using parallel threads is one approach to generating parallel I/O. But, the issue here is generating parallel I/O for every processor because a single processor, even if in a parallel computer, is capable of processing data faster than a single disk can deliver it. Some applications are easily parallelized and can be split into more threads than there are processors. Multiprogramming multiple such threads on a single processor can generate needed I/O parallelism, although at the cost of overhead to switch among the threads. However, there are many applications that are not easily parallelized. Such applications would like to exploit all available threads on different processors to maximize processor performance. For these applications the issue again becomes one of how can I/O parallelism be generated for a single thread running alone on a processor. Thus, although parallel and distributed computing is not the focus of this dissertation, the arguments and techniques developed here also have application in that domain.

Prefetching has been studied specifically in the parallel domain by David Kotz who was perhaps the first to emphasize its importance for increasing I/O parallelism [Kotz90, Kotz91, Kotz93]. He explored techniques for detecting sequential and strided access patterns and prefetching in parallel for them with the goal of increasing array utilization. He was able to demonstrate significant reductions in elapsed time for the parallel computations he studied. These parallel computations already had some intrinsic I/O parallelism because each disk had an independent processor associated with it, but the prefetching helped overlap I/O with computation at each node and more significantly, it allowed I/O to continue even when the processor was stalled at a synchronization point for the parallel computation. However, as was the case for the single-processor prefetching studies, the lack of certain knowledge about what data will be accessed limited the aggressiveness of the prefetching and therefore the performance gains possible.

2.2.5 ASAP summary

Summarizing this survey of techniques for generating workloads that improve the performance of disk-based secondary storage, we have that:

1. avoidance, although capable of delivering substantial gains, is not a scalable solution to the I/O bottleneck;

2. sequentiality maximizes the utilization of individual disks, and through buffered writes and LFS, it scales to multiple disks for writes, but because there is no general mechanism for converting random reads into sequential ones, it does not scale for reads;

3. asynchrony for writes is scalable through write buffering, whereas scaling for reads depends on scaling either the number of outstanding asynchronous I/Os or prefetching aggressiveness, but for both reads and writes, asynchrony must be backed by scalable throughput and buffer sizes; and,

4. parallelism that can exploit disk arrays is possible for some applications with explicitly parallel I/O requests, but for serial workloads, scalable parallelism can only be achieved by scaling the number of asynchronous requests.

Techniques already exist for eliminating the I/O bottleneck for writes. Writes cannot be completely eliminated, but buffering can make them both asynchronous and parallel. Asynchrony eliminates write latency, and parallelism provides throughput. Scalability is achieved by scaling the size of the write buffer and the disk array.

On the other hand, no existing techniques scalably relieve the I/O bottleneck for reads. This survey made it clear that avoidance, sequentiality, and asynchrony cannot provide the necessary throughput. Parallelism, which can, must be a part of any scalable, long-term solution. Disk arrays already provide hardware parallelism. The challenge, then, in relieving the I/O bottleneck for reads is adding parallelism to the read workload that can exploit array parallelism.

Parallelism, whether achieved explicitly or implicitly through prefetching, is the most important factor in improving I/O performance. But, because avoidance reduces the number of requests that secondary storage must service and because sequentiality increases the throughput of individual disks, both can reduce the degree of parallelism and therefore the cost of secondary storage needed to balance a given processor. Equivalently, by increasing the utilization of individual disk read/write channels, fewer channels, and therefore fewer disks, are needed to provide a given level of throughput. Thus, there is benefit in maximizing all ASAP workload virtues.

Unfortunately, it is not always possible to maximize all four at once. Caching, write-behind, prefetching, asynchronous I/O, batch requests, multiprogramming, and, more generally, virtual memory all require memory resources. A comprehensive approach to maximizing I/O performance should balance these competing demands to make the best use not just of the disk resource, but the memory resource as well.

2.3 Disclosure hints for aggressive prefetching and I/O parallelism

Aggressive prefetching could provide the necessary parallelism for reads just as write buffering does for writes. Parallel prefetches could fill prefetch buffers and allow applications to continue computing without stall. Prefetching is usually thought of as a technique

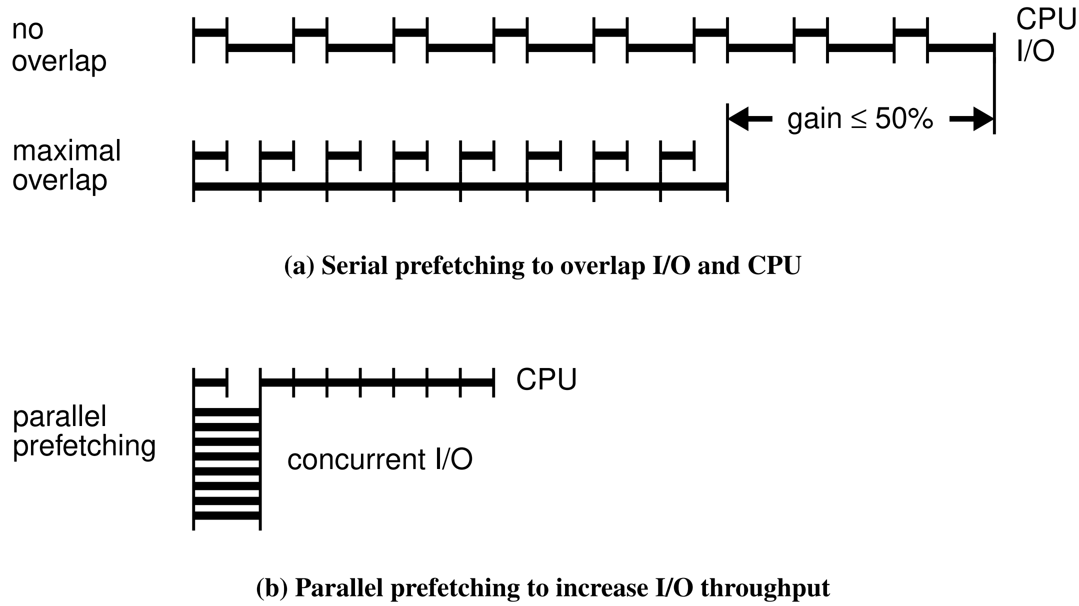


Figure 2.3. Gains from prefetching. The traditional goal of file prefetching is to overlap the latency of a disk access with computation. Figure (a) shows that overlapping disk accesses with computation can reduce elapsed time by at most 50%. Much larger gains are possible when multiple prefetches are performed concurrently on a disk array. Such parallel prefetching can in principle eliminate all but the latency of the first access.

for overlapping I/O and computation. Prefetching can certainly achieve such overlapping, but, as Figure 2.3 shows, prefetching in parallel can provide much greater, scalable performance gains than simple overlapping alone. Prefetching is better thought of as a way to add parallelism to a serial read workload.

The main barrier to using prefetching for read parallelism is that the file system does not know what to prefetch. The problem is that, from the file system's perspective, application accesses can seem random and unpredictable so guessing what to prefetch is difficult or impossible. And yet, programmers wrote applications to perform meaningful, purposeful tasks. Reads are predictable, just not to the file system.

I propose using this predictability to inform the file system of future demands on it. Specifically, I propose that applications disclose their future accesses in hints. In this dissertation, I show, first, that applications can give such hints and, second, how the file system can use these hints to:

1. improve cache performance to avoid accesses;
2. cluster multiple accesses into one and better schedule other accesses to increase sequentiality;

3. aggressively prefetch needed data to asynchronize file reads from disk accesses; and
4. to parallelize the read workload to exploit storage parallelism.

But, perhaps this approach is misguided and knowledge of future accesses could be better exploited at user level to achieve these optimizations. Programmers could modify their code to issue multi-block sequential reads, multi-block batch or vector reads, multiple concurrent asynchronous I/O requests, or to spawn multiple concurrent read-issuing threads. Indeed, some applications already do this. However, there are at least three reasons why the hint approach is superior.

First, modifying programs for explicit parallel I/O is challenging. Breaking a program into multiple threads which can generate independent I/O requests is equivalent to parallelizing the application which is known to be hard except in a few select, intrinsically parallel cases. Asynchronous I/O requires code nimble enough to handle the out-of-order completion of any of the multiple outstanding requests which is certainly more complex than a programming model which only allows synchronous reads of a byte range. Furthermore, both asynchronous I/O and batch accesses require user-level buffer management to reserve space for the read data and recycle the space when done with the data. Finally, unless accesses are already sequential, using knowledge of future accesses to generate sequential ones, if it is even possible, at least requires resorting accesses into some order other than the natural, logical one that is used by the existing code.

In fairness, application hints too require some modifications to program. But, there is no need to parallelize the program, handle out-of-order requests, manage buffers, or resort requests into a possibly unintuitive order dictated by the location of data on disk. Programmers can continue to use the same serial, synchronous programming model that they are used to. In Chapter 3, I show that three straight-forward techniques are sufficient for annotating a broad range of applications to give hints. It is already possible for a compiler to generate some hints without programmer intervention [Mowry96] and there is a reasonable expectation that compiler and other techniques will eventually be able to generate hints for a broader range of applications automatically.

The second, more significant, problem with explicit parallelism is that the programmer must scale the parallelism when the application is ported to a new system. Faster processors require more I/O parallelism to balance performance and I/Os need to start further in advance. The programmer is faced with three choices: not adapt and suffer from insufficient parallelism or wasteful use of cache buffers; manually retune for every system configuration at potentially huge programming cost; or, write code that automatically adapts to each system. But, implementing such automatic adaptation can be difficult. How much further in advance should I/Os be initiated, how many more outstanding requests are needed? Few systems provide meaningful information to user-level programs about processor performance, disk performance, array size, memory size, network speed, etc., that could help determine answers to these questions. And, even if some systems do, they certainly don't provide it in a consistent manner which itself adds to the difficulty of porting such an application.

In contrast, once an application is annotated to give as many hints as it can as early as possible, there is no need to further modify or tune the application. It is up to the operating system and especially the file system to initiate I/Os and manage the cache as appropriate for that system. The operating system may have to be tuned for the particular system configuration, but it makes sense to localize such machine-dependent function there. In Chapters 4 and 5 of this dissertation, I will show how to use a handful of key system parameters to tune file-system I/O and cache management to the particular system. Once the application discloses its knowledge, the system can take advantage of them to scale I/O parallelism to take full advantage of that system's I/O and buffer resources.

Finally, and most significantly, even if an application could be ported to new systems, explicit parallel I/O usurps operating system control of global resources by dictating how many and when buffers should be used for I/O and thereby cripples resource management. Highly parallel I/O can consume large amounts of memory and storage bandwidth resources. Taking buffers for I/O shrinks the pool available for virtual memory and file caching, so applications risk losing more to increases in paging and cache misses than they gain from I/O parallelism. And, they may unfairly hurt the performance of other applications sharing the machine.

Resource allocations should be dynamic, varying in response to changing application needs and system conditions. When applications are sharing the processor, their throughput drops and I/O parallelism and prefetching can be scaled back. When access to a disk is highly contested, caching for that disk becomes more important both to avoid queuing delays and to reduce the load on the disk. When data is read once and not reused, cache buffers can be freed for other uses. Global resource allocation is one of the central tasks of an operating system and explicit parallelism denies the operating system the flexibility it needs to balance competing resource demands and optimize global resource usage.

Disclosure hints empower instead of cripple global resource allocation. Because the operating system is free to ignore hints, they impose no demands on resources. Instead, by providing the system with information about the future, hints enable proactive resource management that anticipates demands. Blocks that will be reused can be held in the cache. Missing blocks can be prefetched.

In Chapter 4, I show how to combine the new information provided by hints with other information about historical resource usage to manage resources and improve I/O performance. I develop a framework for resource management based on cost-benefit analysis that includes three key components. First, it uses locally-computable estimates of the cost (increase in I/O service time) of ejecting a block from the cache and the benefit (decrease in I/O service time) of using a buffer to initiate an I/O that are derived from a model of system performance. Second, to ensure that these local estimates are comparable at a global level, the framework requires that all estimates be expressed in terms of a common currency that relates the change in I/O service time to the amount of buffer resource used. Finally, the framework's allocation algorithm uses the estimates expressed in the common currency to balance the use of buffers for prefetching, clustering, caching according to hints, and caching in the LRU queue for unhinted accesses. In a nutshell, the algorithm ejects the block that will cost least to use its buffer to fetch a block from disk if the estimated benefit of the fetch exceeds the estimated cost of the ejection.

2.4 Related work

Hints are a well established, broadly applicable technique for improving system performance. Lampson reports their use in operating systems (Alto, Pilot), networking (Arpanet, Ethernet), and language implementation (Smalltalk) [Lampson83]. Terry proposes their use for distributed systems [Terry87]. Broadly, these examples consult a possibly out-of-date cache as a hint to short-circuit some expensive computation or blocking event.

An alternate class of hints are those that express one system component's advance knowledge of its impact on another. Perhaps the most familiar of these occurs in the form of policy advice from an application to the virtual-memory or file-cache modules. In these hints, the application recommends a resource management policy that has been statically or dynamically determined to improve performance for this application [Trivedi79, Sun88, Cao94, Cao94a].

In some cases this policy advice can be generated automatically from observations of file system activity. I already mentioned Korner's work on automatically generating caching hints at the client for a remote file server [Korner90]. An example from parallel computing is Madhyastha's work on automatically classifying access patterns to set parallel file system policies [Madhyastha97].

In large integrated applications, detailed knowledge may be available. The database community has long taken advantage of this for buffer management. The buffer manager can use the access plan for a query to help determine the number of buffers to allocate [Sacco82, Chou85, Cornell89, Ng91, Chen93]. Ng, Faloutsos and Sellis's work on marginal gains considered the question of how much benefit a query would derive from an additional buffer. Their work stimulated the development of my approach to cache management. It also stimulated Chen and Roussopoulos in their work to supplement knowledge of the access plan with the history of past access patterns when the plan does not contain sufficient detail.

The use of estimates of the cost of an operation have long been used to develop allocation policies. For example, in his study of sequential prefetching [Smith78], Smith developed estimates of the cost of prefetching a block, the cost of a demand miss, and the cost of the loss of cache effectiveness due to the early ejection of a block to reuse a buffer for prefetching. Then, based on the distribution of sequential run lengths measured in traces of system activity, he determined the number of blocks ahead to prefetch given that a sequential run already has a certain length. Although his estimates of both the distribution of run lengths and the cost of dedicating cache buffers for prefetching were static, he observed that there was some variation in the actual values over the course of the trace he studied. In this dissertation, I do not study heuristic readahead, but I do show how to use dynamic estimates of costs and benefits to guide prefetching and caching decisions. It would be interesting to incorporate estimates of the benefit of heuristic, sequential prefetching into the TIP system described here.

Researchers have considered a variety of rich languages for expressing and exploiting disclosure. One example is collective I/O [Kotz94] in which collections of processes running on a parallel machine describe their related accesses so that the underlying I/O subsystem can optimize across the accesses. This is particularly useful when, for example, each processor is performing strided access to a matrix, but collectively they are accessing the matrix in its entirety.

Another example in the parallel domain is the use of templates to specify parallel I/O access patterns [Parsons97]. In this approach, users specify how the system should coordinate the file accesses of multiple parallel processes. For example, the processes might share a common file pointer so that accesses are serialized, or the file might be broken into distinct segments for each process. Many different patterns are possible, but the use of the template to specify them means that the system can be aware of what access patterns it will be asked to support.

Another example is Dynamic Sets [Steere97] in which users specify a set of files over which they will iterate performing some operation. The specification of a set of files discloses likely access to all files in the set. A call to iterate on the next file in the set may return any file in the set. This gives the underlying system the flexibility to re-order accesses for maximum performance. For example, already cached files can be accessed first while prefetching proceeds for other files. Or, in the context of a distributed system such as the world-wide-web, the system can initiate fetches for multiple objects and deliver objects to the user in the order in which they are received.

Another example interface is an object-oriented file system implemented as library on top of the UNIX file system called ELFS [Grimshaw91]. ELFS has knowledge of file structure (e.g. 2-D matrix) and high-level file operations (e.g. FFT) that it uses for its own prefetching and caching operations. When users request these high-level operations, they in effect disclose to ELFS a large quantity of work. Although ELFS emphasizes user-level control over file activity, it could use its knowledge to give hints to the file system. Applications could further help ELFS performance by disclosing their knowledge in hints to ELFS which could translate them into hints for the underlying file system. ELFS encapsulates a lot of knowledge of file activity in a library, and as such it could be a good complement to an informed prefetching and caching system.

Relatively little work has been done on the combination of caching and prefetching. In one notable exception, however, Cao, Felton, Karlin and Li derive an aggressive prefetching policy with excellent competitive performance characteristics in the context of complete knowledge of future accesses on a single disk [Cao95]. These same authors go on to show how to integrate prefetching according to hints into their system that exploits application-supplied cache management advice [Cao96] which I mentioned earlier. The hints they use for prefetching are very similar to the disclosure hints I advocate, although they supplement them with caching advice hints whereas I rely on the one type of hint for both caching and prefetching. But, a greater distinction between the two efforts is that Cao *et al.* studied prefetching and caching on a single disk, whereas my emphasis is on the use of prefetching to add parallelism to an otherwise serial workload. Recent joint work compared the two approaches and found that an adaptive approach that incorporated features of each worked best across array sizes [Kimbrel96]. A further distinction is the buffer allocation algorithms. Cao *et al.* propose a two-level approach that uses the LRU algorithm to allocate buffers among processes and uses a local, application-controlled manager for each process' buffers. My proposal is for a single unified manager that uses locally-generated cost and benefit estimates to find the best global allocation. A recent study compared and contrasted these two approaches [Tomkins97]. I will discuss both the prefetching and buffer allocation comparison studies in more depth in Chapter 7.

2.5 Conclusions

The starting point for this chapter was the observation that because processor performance is increasing so much more rapidly than disk performance, secondary storage is a worsening bottleneck on overall system performance. Disk arrays, which provide scalable throughput and can balance processor performance, are only a partial solution: without parallel workloads that exploit array parallelism, the latency of individual single-disk accesses dominates storage subsystem performance. Some I/O workloads are already parallel, but many fall between the extremes of large sequential accesses and highly-concurrent small ones. These workloads typically consist of a serial stream of modest-sized requests that run little faster on an array than they do on a single disk.

A sufficiently parallel workload running on a large enough disk array is enough to balance any processor. But, even though disk prices have been dropping rapidly, arrays of disks can still be expensive. Maximizing single-disk utilization minimizes array size and cost. Thus, the broad question is not just how to exploit disk arrays, but how best to maximize the performance of disk-based secondary storage.

After reviewing the performance characteristics of disk-based storage, I identified the ASAP workload virtues for high performance: avoid accesses when possible; increase sequentiality to minimize seeks and maximize read/write channel utilization; mask latency with asynchrony; and maximize throughput with parallelism. Of these, only parallelism can scale with processor performance. However, avoidance and sequentiality can both reduce the size of the array needed by respectively reducing the number of requests or increasing the throughput of individual disks.

Servicing a request asynchronously can mask latency for that request. But, servicing multiple requests asynchronously provides parallelism and therefore throughput and ultimately low latency for lots of requests. Asynchrony coupled with throughput can provide the I/O throughput needed to balance increasing processor performance.

Generating multiple asynchronous writes is easy: accumulate the data in multiple buffers. Asynchronous reads are not as easy to generate. Unless programs are rewritten to issue explicit asynchronous I/Os, the only source of asynchronous reads is prefetching. The challenge in initiating multiple prefetches is determining what to prefetch. Heuristic techniques cannot predict what data will be needed reliably enough to fill the many empty prefetch buffers.

My thesis is that many I/O-intensive applications can predict their own accesses, disclose this knowledge in hints to the file system, and that the file system can take advantage of these hints to improve I/O performance through all four ASAP optimizations. Hints expose concurrency that aggressive, asynchronous prefetching can exploit with a disk array. Further, hints reveal data reuse that can guide cache replacement decisions. Finally, hints provide opportunities to cluster multiple requests into fewer larger disk accesses and to better schedule the disk arm to minimize seeks.

Unfortunately, these many potential optimizations are potential competitors. Prefetching, caching, clustering, and queuing requests early to improve scheduling all require cache buffers. The challenge in implementing a system that exploits application hints is determining how to allocate the limited supply of cache buffers to these alternative uses to maximize performance. Hints enable proactive resource management that anticipates future demands instead of simply responding to current demands. In this dissertation, I will show how to balance buffer usage for prefetching versus caching and integrate this proactive management with traditional LRU (least-recently-used) cache management for unhinted accesses.

Although I will not demonstrate it in this thesis, I believe this approach has applicability in the broader context of I/O that includes network transmission and tertiary storage. New technologies can generally increase throughput, but often fail to reduce latency especially for small requests whose service time is dominated by factors other than simple data transmission. Avoiding accesses to the next lower level in the memory hierarchy always improves performance. Clustering smaller requests into larger ones reduces per-request overhead in distributed and local systems alike. Accessing remote data or a tape drive only increases latency and makes latency-masking asynchrony more important. Finally, parallelism increases performance whenever issuing individual requests leaves some resource idle. In a distributed system, multiple servers may provide an opportunity for parallelism. But, even parallelism in the form request pipelining can increase network and server utilization and therefore performance.

Chapter 3

Disclosing I/O Requests in Hints

This chapter presents *disclosure hints*, the key to my approach for reducing read latency and relieving the I/O bottleneck. The aggressive, proactive management strategy described in this dissertation depends on a reliable picture of future demands. I ask that applications be modified to disclose their future accesses in hints and provide the system with the knowledge it needs. But, not all hints are created equal, especially from a software engineering perspective. This chapter describes the disclosure hints that I advocate and explains why disclosures are preferable to advice hints.

Hints may be a wonderful concept in theory, but to be useful in practice, applications must be able to generate disclosure hints. In the long term, I hope that compilers and profilers may generate reliable hints automatically. Indeed, recent work has already produced promising results. For example, Todd Mowry led a group that showed how compilers can apply memory prefetching techniques in the I/O domain [Mowry96]. But, in the short term, manual techniques for annotating applications to give hints remain the most powerful. Section 3.3 identifies three techniques for manual hint annotation. Then, Section 3.4 shows how to apply these techniques to annotate a broad suite of applications that includes text search, 3D scientific visualization, relational database queries, speech recognition, object code linkers, and computational physics. Using these techniques, the relational data base discloses half of the bytes it reads, speech recognition discloses 90% of the bytes read, and the other four applications disclose over 99% of bytes read. Furthermore, in most cases, the hints disclose hundreds to thousands of accesses at once thereby exposing

I/O concurrency and, as we will see in later chapters, enabling the system to add much needed read parallelism to these application's workloads.

3.1 Hints that disclose

I advocate a form of hints based on advance knowledge called *disclosure* [Patterson93]. An application *discloses* its future resource requirements when its hints describe its future requests in terms of the existing request interface. For example, a disclosing hint might indicate that a particular file is going to be read sequentially four times in succession. Or, at a more detailed level, it might disclose a list of specific segment offsets and lengths that will be read with an implied seek from the end of one segment to the beginning offset of the next segment.

Disclosure hints stand in contrast to hints which give advice. For example, an advising hint might specify that the named file should be prefetched and cached with a caching policy whose name is "MRU." Advice exploits a programmer's knowledge of application and system implementations to recommend how resources should be managed. Disclosure is simply a programmer revealing knowledge of the application's behavior, revealing how the application will use the interface that the system already exports.

Disclosure has three advantages over advice. Together, they show that disclosure hints are a mechanism for passing portable optimization information across module boundaries without violating modularity. First, because it expresses information independent of the system implementation, disclosure remains correct when the application's execution environment, system implementation or hardware platform changes. Hints are not prefetch commands so they can and should be given as early as possible; it is up to the system to take full advantage of them. Even though the appropriate prefetching and caching policy might depend on the number of disks in a disk array, the amount of buffer cache available, or the existence of a large amount of non-volatile RAM, the hints do not need to change. In contrast, to give the best possible prefetching and caching advice, a programmer would have to be sensitive to such variations in system configurations. But, even if a programmer could anticipate all possible configurations today, it would be impossible to anticipate all future configurations. Suppose, for example, that micro-mechanical or holographic storage devices with new and unknown characteristics come into existence. How can a programmer today give advice on their use? Disclosure hints reveal what an application will do and therefore don't depend on system configuration or implementation[1]. As such, disclosure is a mechanism for portable I/O optimizations, portable across platforms today and portable through time to tomorrow's platforms.

Second, because disclosure provides the evidence for a policy decision, rather than the policy decision itself, it is more robust. Specifically, if the system cannot easily honor a particular piece of advice, there is more information in disclosure that can be used to choose a partial measure. For example, if there is too little free memory to cache a given file, advice to cache the file does not help the system decide whether it is more useful to cache the beginning, the end, or recently used portions of the file. Disclosures reveal which portions of the file will be accessed next and therefore which blocks of the file to cache.

Such robustness is particularly important for global optimizations. If two processes advise the system to cache their file, but they don't both fit, which should the system cache? Should it cache all of one and none of the other? Parts of each? Which parts? Simple advice provides inadequate information to answer these questions, whereas disclosure arms the system with the facts that, as we will see, allow it to make an appropriate allocation.

The third advantage is that disclosure conforms to software engineering principles of modularity because it is expressed in terms of the interface that the application later uses to issue its accesses; disclosure hints are expressed in terms of file names, file descriptors, and byte ranges, rather than inodes, cache buffers, or file blocks. Disclosure does not require knowledge of another module's implementation, it simply requires that one module disclose what invocations of the other module it will make.

Disclosure's respect of modularity is in a sense a general statement of the first advantage: that disclosure is portable from one system implementation to another. But, it has more profound ramifications as well. In particular, it means that disclosure hints may be passed through multiple layers of software. Suppose, for example, that an application uses

[1] Of course, if the application's behavior is dependent on system configuration, then the hints must also be dependent on configuration. However, such dependencies must be known to the application writer to be included in the code and so in principle this knowledge may be used to give appropriate hints.

a math library for complex out-of-core matrix manipulations. For the application to give advice to the system about how to support these operations, it would need to understand how the library was implemented. An alternative is for the application to disclose to the library what library calls the application will make and for the library to translate these calls into file access hints which it then discloses to the system. In this way, disclosure hints may be passed through layers of software, transmitting optimization information down through all layers, without violating the modularity of these layers. Admittedly, this is easier for some layers to implement than others. Caches, in particular, have difficulty predicting far in advance which accesses will hit and which miss. Further work is needed both on predicting cache misses and on support in the hint interface for modules that cannot predict requests with total accuracy. Nevertheless, disclosure hints are already a useful mechanism for passing optimization information through multiple layers of software and in Section 3.4.5, I will give an example of an application that does just that.

Modularity often stands in opposition to performance because it hides implementation details which are important for performance. In response to this problem, researchers have been exploring ways for a user to influence the underlying implementation. A notable example is Gregor Kiczales and his work on metaobject protocols and open implementations [Kiczales92] which allow users to influence implementation choices and so tune an implementation to support particular applications.

In the operating systems research community, efforts have been focussed on moving functionality out of the kernel and into user space where the user can customize behavior. Examples of this include external pagers [Harty92], scheduler activations [Anderson92], and, in the extreme case, micro-kernels themselves [Accetta86, Rozier88, Engler95]. Certainly, this approach can lead to dramatic performance gains for applications that are willing to rewrite significant chunks of the operating system. But, many application programmers don't wish to become systems programmers. They would prefer to focus their efforts on their own algorithms. Because disclosure hints only require knowledge of what the application itself does, they do not require new, specialized knowledge on the part of the programmer and so are an attractive alternative to reimplementing system code. A deeper problem is that when multiple applications reimplement the same system function, global knowledge is lost along with the opportunity for global optimizations.

Disclosure hints offer an alternative to these efforts. I acknowledge that application information is helpful, even necessary, for good performance, but also appreciate the advantages of keeping resource allocation and system-specific tuning in the operating system. Ideally, applications and the system would cooperate in managing resources and optimizing performance. Disclosure hints are portable, enable global resource management, and do not violate modularity.

The remainder of this chapter details the disclosure hint interface, describes three techniques for annotating applications to give hints, and shows how to apply these techniques to annotate a broad range of six applications that includes: Davidson computational physics, XDataSlice 3D scientific visualization, Gnuld object code linker, Sphinx speech recognition, Agrep text search, and two queries to the Postgres relational database. A major thrust of the remainder of this dissertation will be showing that a system can successfully optimize its behavior based on disclosure information. It is not necessary to move the burden of writing operating systems on application writers to obtain significant performance gains.

3.2 The hint interface

My dual goals in designing the hint interface for TIP, my implementation of an informed prefetching and caching system, were simplicity and support for the test applications. Only three pieces of information are needed to describe UNIX reads: the file, the byte offset to start the read, and the number of contiguous bytes read before a seek elsewhere. Disclosure hints reveal this same information in advance of the actual read. The order hints are given indicates the order accesses are anticipated. Conceptually, new hints are added to the end of a list. I could have explored richer interfaces that supported, for example, insertion of new hints at arbitrary locations, compact representations of strided accesses, concurrent hint streams with unknown interleaving, probabilistic hints that indicate data might be accessed, etc. But, the suite of applications studied did not need such embellishments, so I left them out.

The programming model for giving hints includes only three rules. First, applications should issue hints as early as possible so that the system has as much time as possible to take advantage of them. Second, applications should issue as many hints as possible to

hint	target	parameter	description
TIPIO_SEG	/dev/tip	tipio_segbuf_t *	batch of <offset, length> segments for a named file
TIPIO_FD_SEG	open file descriptor	tipio_fd_segbuf_t *	batch of <offset, length> segments for an open file
TIPIO_MFD_SEG	/dev/tip	tipio_mfd_segbuf_t *	batch of <fd, offset, length> segments for multiple open files
TIPIO_CANCEL	/dev/tip or open file descriptor	null	cancels segment at head of hint list; used when a hint turns out to be erroneous

Table 3.1. Ioctl calls in the disclosure hint interface. Disclosure hints describe future requests in the same terms as the existing file interface. Thus, they must specify the file, the starting offset of the access, and the length of the sequential access before a *seek* to a new offset. This information is relayed to the file system via *ioctl* system calls using one of the hints specified in this table. Hints specifying a file by name are given in *ioctl* calls to the /dev/tip pseudo-device, whereas *ioctls* giving hints about open files can target those files directly. The structures which are the parameters for some of the hints are defined in Figure 3.1.

expose as much I/O concurrency as possible.[2] Not all of the concurrency may be needed today, but maximizing the exposure of concurrency ensures the effectiveness of the hints in the future. And third, to guarantee that the system does not discount their hints as too inaccurate, applications should perform all accesses for which they have issued hints. If a hint turns out to have been incorrect, the application should issue a `TIPIO_CANCEL` hint as described below.

In the TIP system, hints are passed to the file system via an I/O-control (*ioctl*) system call. Table 3.1 summarizes the supported calls. There are two ways to specify the file: file descriptor or file name. If the file is already open, then the open file descriptor can be the target of the *ioctl* call as in the `TIPIO_FD_SEG` hint. Alternatively, the file can be specified by name, in which case the target of the *ioctl* call is a pseudo-device named “/dev/tip” as in the `TIPIO_SEG` hint.

Hints specify contiguous file segments with an <offset, length> couple (not all the bytes need be requested in a single read call). To reduce what the application must know before hinting, a length of 0 indicates the file will be read from the offset to the end of the

[2] At the limit, hint storage itself could become a problem if too many hints are given. The system could limit the number of outstanding hints, but a better solution might be to store the hints themselves on disk. None of the benchmark applications ever had more than 16,000 hints outstanding at a time, so this was not a problem.

```
/* contiguous file segment */
typedef struct tipio_seg {
   int   offset;
   int   length;
} tipio_seg_t;

/* batch of segments in a named file */
typedef struct tipio_segbuf {
   char  *path;            /* file name */
   int   nsegs;            /* number of segments */
   tipio_seg_t*seg;        /* array of segments */
} tipio_segbuf_t;

/* batch of segments in an open file */
typedef struct tipio_fd_segbuf {
   int   nsegs;            /* number of segments */
   tipio_seg_t*seg;        /* array of segments */
} tipio_fd_segbuf_t;

/* segment in a specified open file */
typedef struct tipio_mfd_seg {
   int   fd;               /* open file descriptor */
   off_t offset;
   off_t length;
} tipio_mfd_seg_t;

/* batch of segments in multiple open files */
typedef struct tipio_mfd_segbuf {
   int   nsegs;
   tipio_mfd_seg_t *seg;   /* array of segments */
} tipio_mfd_segbuf_t;
```

Figure 3.1. Structure definitions for the disclosure hint interface. These structures specify the starting offset and length of sequential accesses to files. To reduce system call overhead, batches of hints may be given in one call.

file. Thus, a hint with offset=0 and length=0 indicates a sequential read of the whole file.[3] To reduce system call overhead, a single *ioctl* call may deliver an ordered list of many such couples; the ordering in the list indicates the order of accesses. Figure 3.1 defines the structures which are the parameters to the *ioctl* that pass in these couples.[4]

Finally, the `TIPIO_CANCEL` hint lets applications cancel hints. Some applications, even when trying to be precise, occasionally give incorrect hints. Specifically, applications that

---

[3] For historical reasons, the interface includes TIPIO_SEQ and TIPIO_FD_SEQ hints as shorthand for this special case of whole-file sequential read.

[4] The implemented interface differs from this in several minor cosmetic ways. For example, the tipio_fd_segbuf_t structure actually contains an array of segments instead of a pointer to a separate array.

maintain an internal cache sometimes fail to predict an internal cache hit and an anticipated access never occurs. The current TIP implementation tolerates accesses for which no hint was ever given, but expects that all hinted accesses eventually occur in the hinted order. Because TIP matches accesses to hints and may only prefetch to a limited depth in the hint sequence, inaccurate hints risk halting prefetching. To avoid this, applications can cancel their erroneous hints as they discover them.

Ideally, the system would be resilient to minor inaccuracies in the hint stream. Such resilience would require matching an arbitrary subsequence of the actual accesses to the hinted sequence, possibly with reordering. Because the entire access sequence is unknown ahead of time, when an access doesn't match the hint sequence, the system cannot be certain whether the hints were inaccurate or whether the hinted access simply hadn't occurred yet. Such uncertainty complicates the system's task of deciding whether and what to prefetch. A further complication is that resilience could add undesirable ambiguity to the programming model if programmers became uncertain whether the system would tolerate their slightly inaccurate hints or not. Different system implementations might tolerate varying levels of inaccuracy. How can programmers know whether their hints will be accurate enough? The exacting requirements of the current system are at least deterministic. For all these reasons, adding resilience to inaccurate hints is difficult and remains a topic for further research.

3.3 Annotation techniques

In the future, hints may be generated automatically by a compiler, run-time analyzer, or other means, but for this work, applications were annotated by hand to give hints. In this section, I describe the three techniques used to annotate the six benchmark applications.

The hinting techniques range in complexity from the in-line insertion of hints where knowledge of future accesses becomes available to loop splitting in which application code is restructured to get around data dependencies that would otherwise make early hints hard to generate. Loop duplication is intermediate between these two in that it may require noticeable amounts of new code, but leaves the original code largely intact.

After describing the three hint techniques in abstract terms, I will go on show how to apply these techniques to annotate the benchmark applications.

3.3.1 In-line hinting

In some applications, specific access patterns are known to the programmer at certain points in the code. *In-line* hints simply disclose this information as it becomes available. Such hints require little or no special work to generate and consequently they often are added very easily to a program.

In their simplest form, a separate in-line hint may be given in advance of individual read calls. Of course, for such hints to be useful they must be given well before the actual access; hints given immediately before the corresponding read provide no additional information. It is always best to give hints as early as possible.

In-line hints that disclose multiple accesses expose I/O concurrency even when they are not given far enough in advance to overlap much computation with I/O. Such hints are possible, for example, when a program loop reads from a file with a predictable pattern. Before entering the loop, an in-line hint could disclose the access pattern. Such a hint does not give much advance warning for the first access, but it may allow the second and subsequent accesses to be prefetched in parallel. Moreover, if the body of the loop contains significant computation, such a hint does give significant advance notice for the later accesses.

Even when it is not possible to anticipate all of the accesses within a loop, it may be possible for an in-line hint to disclose accesses one iteration in advance. When loops enclose multiple accesses or significant computation, hints in-lined in the body of the loop can still expose I/O concurrency or at least give sufficient advance warning of impending accesses to overlap computation with the I/O.

Agrep and Davidson are good examples of applications that give in-line hints. Postgres also gives in-line hints for some of its accesses.

3.3.2 Loop duplication

Within some program loops, in-line hints may only be possible in the same iteration as the actual access. This often results in hints that give little advance warning and disclose little concurrency. *Loop duplication* is a technique for lifting these hints out of the enclosing loop so that multiples of them may be given far in advance thereby giving much more advance warning and disclosing substantial I/O concurrency.

In loop duplication, the control structures of the enclosing loop are duplicated in a new shadow loop placed before the original one. The shadow loop does not perform any of the file accesses of the original. Instead, the body of the shadow loop includes only those operations necessary to determine what the accesses of the original loop will be. As the accesses are determined, hints can disclose them. Using this technique, all the accesses in some loops may be disclosed before the start of the loop.

Gnuld uses this technique to generate some of its hints.

3.3.3 Loop splitting

Loop duplication as described above is sometimes unsatisfactory for two reasons. First, it may lead to unacceptable overhead if the shadow loop must duplicate a significant amount of computation. Second, it may not be effective when, as described below, there are data-dependent reads within the loop body. The solution is to use *loop splitting* to avoid duplicating work and separate data-dependent accesses.

Like loop duplication, loop splitting duplicates the original loop's control structures to create a second loop. But, if determining what data will be accessed requires a substantial amount of computation, loop splitting temporarily stores partial results generated in the first or top half of the split loop. After issuing hints based on the results of the first loop, the second or bottom half of the split loop takes advantage of the stored partial results to avoid the overhead of recomputing them and computes the final result. The use of temporary storage for partial results and the modification of the original loop distinguish loop splitting from loop duplication.

Sometimes a read depends on other data read within the loop body. This occurs, for example, when an initial read of a header determines the offset of data accessed in a second read. Splitting the loop body between the two accesses separates such data-dependent accesses from their dependencies and solves the problem that a hint for the second read is only possible after the first read completes. The first loop performs all the initial reads, for example of the headers, and stores the offsets for all of the data-dependent reads. The second loop refers to the stored offsets, skips the initial reads and only performs the second, data-dependent reads. But, before the second loop begins, hints disclose the offsets of the data-dependent reads in the second loop. Not only are these hints given much further in advance, but, more importantly, they are given all at once thereby exposing I/O concurrency.

Postgres, Gnuld, and XDataSlice all take advantage of loop splitting to give hints.

3.4 Annotating applications to give hints

In this section, I describe the six applications that have served both as case studies in annotating applications with hints and as benchmarks for evaluating the performance benefits of informed prefetching and caching based on these hints.

3.4.1 Agrep

Agrep, version 2.04, a variant of the standard UNIX *Grep* utility, was written by Wu and Manber at the University of Arizona [Wu92]. It is a fast full-text pattern matching program that allows matching errors. Invoked in its simplest form, it opens the files specified on its command line one at a time, in argument order, and reads each sequentially. In our benchmark, Agrep searches 1349 kernel source files occupying 2922 disk blocks for a simple string that does not occur in any of the files.

Because the arguments to Agrep completely determine the files it will access, Agrep can issue hints for all accesses upon invocation. When searching data collections such as software source files or mail messages, hints from Agrep frequently specify hundreds of files too small to benefit from history-based, sequential readahead. In such cases, informed prefetching has the advantage of being able to prefetch across files and not just within a single file.

Annotating Agrep is easy. Before searching any files, the program loops through the argument list and checks that each argument is a valid file name. As Figure 3.2 shows, an in-line hint inserted within this loop discloses the names of all the files Agrep will read. Adding this call requires about ten lines of code, most of which are ‘#include’ directives or variable declarations.

in-line hint

```
foreach arg_string on command line {
   if arg_string names a file {
      add arg_string to list of files to search;
      hint (arg_string, offset=0, length=0);
   }
}
```

Figure 3.2. In-line hints in Agrep. Before searching for a string in any file, Agrep loops over its command line arguments and adds those that are files to a list of files to search. It is a simple matter to insert a hint that the file will be searched. An offset of 0 indicates that the file will be searched from the beginning and a length of 0 indicates that it will be searched sequentially till the end of the file.

3.4.2 Gnuld

Gnuld version 2.5.2 is the Free Software Foundation's object code linker which supports ECOFF, the default object file format under Digital UNIX. In the benchmark, Gnuld links the 562 object files that make up a Digital UNIX kernel. Gnuld passes over the input object files several times in the course of producing the output linked executable. In the first pass, Gnuld reads each file's primary header, a secondary header, and its symbol and string tables. Hints for the primary header reads are easily given by duplicating the loop that opens input files[5]. Thus, Gnuld gives hints across files just as Agrep does. The read of the secondary header depends on data in the primary header. It would have been possible to apply loop splitting to disclose these reads, but usually neither the primary nor secondary headers are large, so they tend to be co-located in the same block. Thus, even though Gnuld doesn't hint them, the secondary-header reads usually hit in the cache.

The secondary headers provide the location and size of the symbol and string tables for that file. To give hints for the table reads, the loop is split after the read of the secondary header.

After verifying that it has all the data needed to produce a fully linked executable, Gnuld makes a second pass over the object files to read and process debugging symbol information. This involves up to nine small, non-sequential reads from each file. Fortunately, the previously read symbol tables determine the addresses of these accesses. Loop duplication is used to generate hints for this second pass.

During the second pass, Gnuld constructs up to five shuffle lists which specify where in the executable file object-file debugging information should be shuffled or copied.

[5] Thanks to Daniel Stodolsky who annotated Gnuld to give hints.

shadow loop

```
foreach ECOFF segment {
   foreach input_file {
      consult load map;
      hint(input_file, offset, size);
   }
}
```

original loop

```
foreach ECOFF segment {
   foreach input_file {
      consult load map;
      seek to offset in input_file;
      read size bytes from input_file;
      patch addresses;
      seek & write to output_file;
   }
}
```

Figure 3.3. Loop duplication in Gnuld. In the course of building an executable file, Gnuld constructs a map that indicates what data from each input file belongs where in each ECOFF segment of the output file. When the time comes to build the output file, Gnuld loops over the different segments and consults this map to read in the relevant data from each input file which it patches and writes to the appropriate destination in the output file. In the pseudocode above, this code in labeled as the 'original loop.' To give hints, the control structures of this loop are duplicated in a shadow loop which consults the same load map and discloses the many reads that the original loop will perform.

When the second pass completes, Gnuld finalizes the link order of the input files, and thus the organization of non-debugging ECOFF segments in the executable file. As shown in Figure 3.3, loop duplication again serves to exploit this order information and the shuffle lists to give hints for the final passes.

3.4.3 Postgres

Postgres version 4.2 [Stonebraker86, Stonebraker90] is an extensible, object-oriented relational database system from the University of California at Berkeley. In our benchmark, Postgres executes a join of two relations. The outer relation contains 20,000 unindexed tuples (3.2 MByte) while the inner relation has 200,000 tuples (32 MByte) and is indexed (5 MByte). The benchmark suite includes two cases. In the first, 20% of the outer relation tuples find a match in the inner relation. In the second, 80% find a match. One output tuple is written sequentially for every tuple match.

To perform the join, Postgres reads the outer relation sequentially. For each outer tuple, Postgres checks the inner relation's index for a matching inner tuple and, if there is one, reads that tuple from the inner relation. From the perspective of storage, accesses to the inner relation are random, defeating sequential readahead, and have poor locality, original loop

```
foreach tuple in outer relation {
   look for match in index;
   if there's a match {
      seek & read match from inner_relation;
   }
}
```

---

top of split loop

```
foreach tuple in outer_relation {
   look for match in index of inner_relation;
   if there's a match {
      store offset of match in temp_array;
   }
}
hint (inner_relation, offsets in temp_array, block_size);
```

bottom of split loop

```
foreach tuple in outer_relation {
   if there's a match in temp_array {
      seek & read match from inner_relation;
   }
}
```

Figure 3.4. Loop splitting in Postgres. In the original loop, read from the inner relation depends on the index lookup in the top half of the loop. Without loop splitting, hints for these reads could only be given one-at-time after the index lookup. Such hints would neither give much advance warning nor expose I/O concurrency. Splitting the loop after the index lookup separates the lookup from the dependent inner-relation read. The top loop performs all the index lookups in one pass. Then, a hint discloses the offsets of all the inner-relation blocks that will be read in the bottom loop.

defeating caching. Thus, most of these inner-relation accesses incur the full latency of a disk read.

The inner-relation accesses depend on the result of the index lookup. Thus, as shown in Figure 3.4, loop splitting is used to separate the accesses and generate hints for the inner-relation reads[6]. In the top half of the split loop, Postgres reads the outer relation (disclosing its sequential access), looks up each outer-relation tuple in the index (unhinted)[7],

[6] Thanks to Eka Ginting who annotated Postgres to give hints.

[7] It would have been possible to further split the loop to give hints for the index, but this was not done for two reasons. First, the structure of the code makes splitting the index lookup difficult. The index is a B-tree, so to give hints for all accesses, each descent to a deeper level of the tree would have to be split into its own loop. The code treats the index lookup as a single step so this would require significant restructuring. An application written from scratch could accommodate this, but a programmer might resist modifying the existing code. I didn't want the benchmark applications to be overly-tuned, but instead wanted them to be a reasonable representation of what a programmer might actually do. The goal is to discover how well the disclosure approach works without resorting to heroic efforts. The second reason is that the index's B-tree structure and relatively small size compared to the inner relation mean that the index accesses have fairly good locality and cache reasonably well. Thus, hints for these accesses would not have as great an impact on elapsed time as hints for the inner-relation tuples.

and stores the offsets of blocks containing matching inner-relation tuples in an array. Postgres then discloses these offsets to TIP, 4000 in 20%-match case and 16000 in the 80%-case. In the second pass, Postgres rereads the outer relation but skips the index lookup and instead directly reads the inner-relation tuple whose address is stored in the array. Postgres does not give hints for the second read of the outer relation because TIP does not currently support hints for multiple streams of accesses with unknown interleaving. Postgres could observe how the inner-relation access are interleaved with the matching outer-relation tuples, but we did not go to the extra effort that this would have required.

As a complication, Postgres maintains its own internal cache of 100 blocks. This cache significantly speeds index lookups because the index is organized as a B-tree and there is high locality for accesses to the nodes near the root of the tree. Even though the hinted, inner-relation accesses have very poor locality, a few of them hit in this cache and the hinted access never occurs. To confirm to the TIP system that Postgres is consuming the data it hints and is not a rogue application, Postgres issues a `TIPIO_CANCEL` hint for these blocks.

3.4.4 Davidson

The Multi-Configuration Hartree-Fock (MCHF) is a suite of computational-physics programs which were obtained from Vanderbilt University where they are used for atomic-physics calculations. The *Davidson* algorithm [Stathopoulos94] is an element of the suite that computes, by successive refinement, the extreme eigenvalue-eigenvector pairs of a large, sparse, real, symmetric matrix stored on disk. In the benchmark, the size of this matrix is 2089 8-KByte blocks or 16.3 MBytes. In practice, the matrix may be many times this size.

The Davidson algorithm iteratively improves its estimate of the extreme eigenpairs by computing the extreme eigenpairs of a much smaller, derived matrix. Each iteration computes a new derived matrix by a matrix-vector multiplication involving the large, on-disk matrix. Thus, the algorithm repeatedly accesses the same large file sequentially. The algorithm terminates when the errors of its eigenpair estimates are within some threshold. In our benchmark, this requires 60 iterations through the matrix.

Annotating this code with in-line hints was straightforward[8]. One hint placed above the loop hints the first whole-file, sequential read of the matrix. A second hint within the loop discloses, at the start of each iteration, the read anticipated in the next iteration. The first sixty of Davidson's hints are accurate. When the algorithm terminates, one, unconsumed hint is left outstanding.

3.4.5 XDataSlice

XDataSlice (XDS) is a data visualization package developed by the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign [NCSA89]. Among other features, XDS lets scientists select and view a false-color representation of an arbitrary planar slice through their 3-dimensional scientific dataset. The datasets may originate from a broad range of applications such as airflow simulations, pollution modeling, or magnetic resonance imaging, and tend to be very large. In our benchmark, XDS retrieves 25 random slices from a dataset of 512^3 32-bit floating-point numbers which is a total of 512 MBytes in size.

It is often assumed that because disks are so slow, good performance is only possible when data resides in main memory. Thus, many applications, including XDS, read the entire dataset into memory before beginning computation. Because memory is still expensive, the amount available often constrains scientists who would like to work with higher resolution and therefore larger datasets. Informed prefetching invalidates the slow-disk assumption and makes out-of-core computing practical, even for interactive applications. To demonstrate this, I first added an out-of-core capability to XDS. I next added disclosing hints for TIP to exploit.

The selection of XDS as an example application for informed prefetching was fortuitous because XDS has an internally layered structure. In adding hints to XDS, I show how to use layered disclosure to pass optimization information through layers of software without violating the integrity of module interfaces.

I describe first the structure of XDS and then go on to describe how I added dynamic data loading and then disclosing hints to it.

[8] Thanks to Daniel Stodolsky who annotated Davidson with hints.

3.4.5.1 *XDataSlice* organization

XDS reads data from files stored in a self-describing format called the Hierarchical Data Format (HDF). NCSA has developed a library of routines to simplify access to HDF files and to enforce the HDF standard format. XDS binds this HDF library between itself and the file system. The HDF library is itself composed of two layers: low-level storage management in the *H* layer and scientific dataset object management in the *DFSD* layer.

A single HDF file may contain many data objects such as raster images, raw scientific data, or the format specifier for numerical data. But, to the low-level H layer of the HDF library, all are just arrays of bytes with a name. The high-level DFSD layer of the library refers to these elemental data objects by name and may request the H layer to deliver logical byte ranges from within individual objects. It is up to the H layer to allocate file space and keep track of the location and size of the various data objects.

The DFSD layer groups a number of elemental data objects together to form a scientific data set. These objects include one holding the raw scientific data and others holding dataset metadata such as the dimensions of the data, the data type, and units and labels for the axes. Applications built on top of the DFSD layer refer to the scientific data set as if it were one complex data object with many typed data fields.

The original XDataSlice code, operating above the HDF library, uses the DFSD interface first to determine dataset size so it can allocate adequate memory, and then to read the entire dataset into memory. To render a slice of the dataset, XDS loops through all the pixels in the slice mapping each to a data element stored in memory. False color is applied based on a data element's value and the resulting bitmap is displayed in an X window.

I extended this basic package to load data dynamically from large datasets. Standard 3-D HDF data objects are written to disk in row-major order. Several full rows of data from even a large dataset fit in each 8-KByte file block. Consequently, when rendering a slice, all file blocks containing rows that intersect the slice must be read from disk. This has the disadvantage of requiring that the entire data object be read from disk to render a slice that cuts across all rows. To make loading arbitrary slices efficient, and in keeping with state-of-the-art tiling techniques [Wolfe96], I reorganized the object into submatrices as shown in Figure 3.5 and extended the DFSD layer to export a blocked view of the scientific data object. I then modified XDS to first determine which blocks are needed and

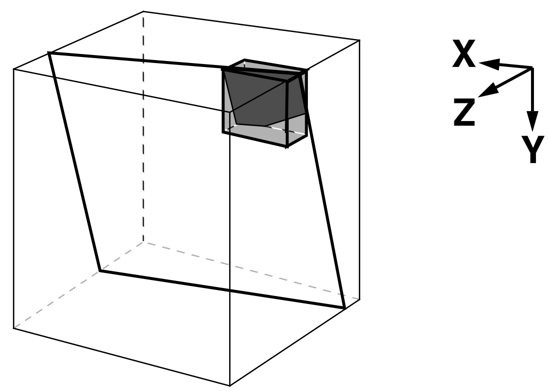


Figure 3.5. Blocked dataset storage layout. To facilitate the retrieval of arbitrary slices of data, the dataset is partitioned into submatrices each stored in its own file system block. The shaded cube above shows one such block and its share of a slice through the dataset. The blocks themselves are stored in row-major order, Z-axis first. Thus, sequential disk access favors slices in the Y-Z plane. To compensate, the blocks are asymmetrical, so that rendering slices in the X-Y plane requires fewer total blocks.

then load only these blocks into memory before rendering the requested slice in the usual way. All of these changes add useful functionality and are independent of TIP.

3.4.5.2 Extending HDF to disclose hints to TIP

For this new version of XDataSlice to take advantage of TIP, it must disclose its expected accesses. Because the primary benefit of TIP is exposing I/O concurrency, the source of hints should be at a level aware of a large volume of work before it is actually requested. There are a number of possibilities, but a simple and natural choice is to issue hints within the DFSD layer of the library because XDS hands this layer a complete list of the needed blocks. This list is an excellent hint for TIP.

Unfortunately, the DFSD layer cannot directly pass the list of blocks on to TIP. Even after the DFSD layer translates block coordinates into logical offsets within the scientific data object, it does not know the offsets within the enclosing file. It relies on the lower H layer for addressing and accessing files. The DFSD layer could “peek beneath the covers” of the H layer to compute the offsets itself, but this would violate the design’s modularity and could break when the H layer is independently modified at some later time.

A much better solution is to incorporate a path for the disclosure of optimization information into the interface to the H layer of the library. I have done this by adding an `Hhint()` routine to the library. It accepts hints from higher layers of the library in the language used by the rest of H layer: offsets and lengths within data objects. `Hhint` maps data object offsets to file offsets and issues a `TIPIO_SEG` hint. Such disclosure is consistent with the module interfaces already in place; the DFSD layer issues hints about data objects and the `Hhint` routine translates these data-object hints into file-access hints which it discloses directly to TIP. The modularity of the HDF library is not a barrier to hints that disclose.

The `Hhint` routine provides the DFSD layer with a modular mechanism for issuing hints, but the DFSD layer still needs to find a way to call `Hhint` to issue hints. We accomplish this through loop splitting.

The DFSD layer receives a list of the coordinates of the submatrices required by the XDataSlice application to render a slice. The original, unhinting code loops over these coordinates translating each to an offset within the data object and then calling `Hseek` followed by `Hread` to retrieve the needed block. Inserting an `Hhint` call within this loop would not provide much advance warning. An alternative is to duplicate the loop and translate the coordinates and issue hints. The unchanged, original loop would re-translate the coordinates and perform the seek and read. To save the cost of re-translating the coordinates, I take advantage of the fact that the translated coordinates are passed in an array to `Hhint`. I use this same array to store the translated coordinates from the first loop and iterate over these in a second loop. Thus, the original loop is split to deliver hints efficiently.

3.4.6 Sphinx

Sphinx [Lee90] is a high-quality, speaker-independent, continuous-voice, speech-recognition system developed at Carnegie Mellon. In the benchmark, Sphinx recognizes an 18-second recording commonly used in Sphinx regression testing.

Sphinx represents acoustics with Hidden Markov Models and uses a Viterbi beam search to prune unpromising word combinations from these models. To achieve higher accuracy, Sphinx uses a language model to effect a second level of pruning. The language model is a table of the conditional probability of word-pairs and word-triples. At the end of each 10 msec acoustical frame, the second-level pruner is presented with the words likely to have ended in that frame. For each of these potential words, the probability of it being recognized is conditioned by the probability of it occurring in a triple with the two most recently recognized words, or occurring in a pair with the most recently recognized

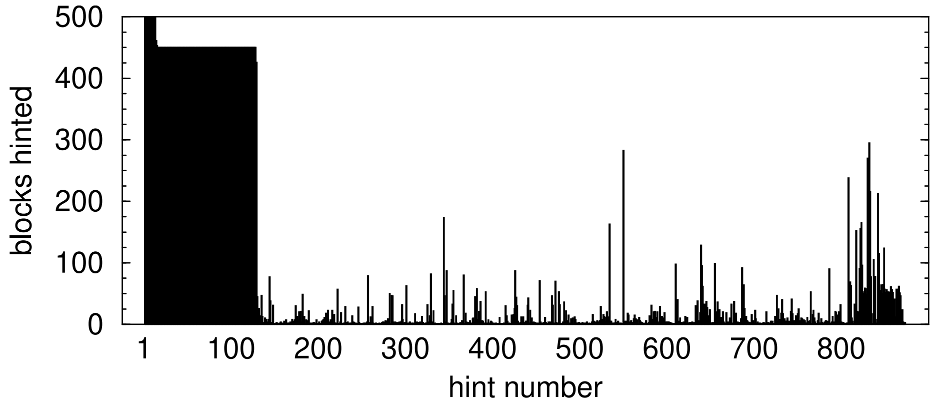


Figure 3.6. Sphinx: blocks hinted in each hint. This graph shows the distribution of the number of blocks hinted by each of Sphinx's 873 hints. The first approximately 120 hints are for initialization and disclose a maximum of 2477 blocks. The rest disclose dynamic loads of language model data as needed during the course of recognizing the speech segment. The average hint during the recognition phase is for about 15 blocks, although many are for a lot more than that.

word when there is no entry in the language model for the current triple. To further improve accuracy, Sphinx makes three similar passes through the search data structure, each time restricting the language model based on the results of the previous pass.

Sphinx, like XDS, was originally an in-core only system. Because it was commonly used with a dictionary containing 60,000 words, the language model was several hundred megabytes in size. With the addition of its internal caches and search data structures, virtual-memory paging occurs even on a machine with 512-MBytes of memory. Sphinx was modified to fetch the language model's word-pairs and word-triples from disk as needed[9]. This enables Sphinx to run on a 128-MByte test machine 90% as fast as on a 512-MByte machine.

Sphinx was also modified to disclose the word-pairs and word-triples that will be needed to evaluate each of the potential words offered at the end of each frame. Figure 3.6 shows the distribution of the number of blocks hinted by each of Sphinx's 873 hints. The hints for the initialization-phase reads disclose a high degree of concurrency. Hints during the recognition phase are highly variable. Because the language model is sparsely populated, at the end of each frame there are about 100 byte ranges that must be consulted, of which all but a few are in Sphinx's internal cache. However, there is a high variance on

[9] Thanks to Daniel Stodolsky who modified Sphinx to operate out-of-core and then annotated it with hints.

the number of pairs and triples consulted and fetched, so, although the hints provide little advance warning, they often expose I/O concurrency.

3.5 Conclusion

The first hurdle in making the case that application hints about future file reads can compensate for the growing disparity between processor and disk performance is demonstrating that important applications can in fact give hints. Without this, there is no need to pursue this line of research any further.

In this chapter, I first argued for hints that disclose future accesses in preference to hints that give advice about filesytem behavior. The distinguishing feature of disclosure hints is that they are expressed using the same terms of file, byte offset, and byte length as the existing file-system interface. I presented a prototype interface for delivering disclosure hints.

I then described three techniques for annotating applications to give disclosure hints: in-line hinting, loop duplication, and loop splitting. In six application case studies, I described how to use the three techniques to annotate important, I/O-intensive applications to give hints. In one, XDataSlice, I showed how disclosure hints may be translated by intermediate software layers to preserve modularity.

How successful were the annotations at disclosing these application's read accesses? Table 3.2 reports the I/O workloads of the benchmarks and the percentage of the read traffic disclosed in advance by hints. For four of the applications, hints disclose more than 99% of the bytes read. Hints disclose 90% of the bytes read by a fifth application, Sphinx. Although hints disclose Postgres' random, data-dependent inner-relation accesses, no annotations disclose the large number of index accesses. However, as will be seen in Chapter 6 which describes the performance of the benchmarks, these index accesses cache well even without hints, and so, although incomplete, the hints yield huge performance wins for the application.

Hints may disclose every access, but if they don't also expose concurrency, then they don't add the parallelism to the read workload needed to relieve the I/O bottleneck. Fortunately, even though the applications are drawn from a broad range of fields, they all tend to give hints in bursts. Here, I define a burst as a sequence of hints given between hinted

benchmark		read calls	read blocks	read bytes	write calls	write blocks	write bytes
Agrep	total	4277	2928	18,091,527	0	0	0
	% hinted	68%	> 99%	> 99%			
	inaccurate	0	0	0			
Gnuld	total	13,037	20,091	60,158,290	2343	3418	8,824,188
	% hinted	78%	86%	> 99%			
	inaccurate	0	0	0			
Postgres (20%)	total	8678	8676	70,657,265	156	156	1,279,590
	% hinted	51%	51%	51%			
	inaccurate	70	70	576716			
Postgres (80%)	total	31,245	31,243	255,526,130	539	539	4,417,126
	% hinted	51%	51%	52%			
	inaccurate	242	242	1,982,464			
Davidson	total	19,000	144,425	1,027,634,130	1474	1487	110,860
	% hinted	99%	> 99%	> 99%			
	inaccurate		2089	17,113,088			
XDataSlice	total	46,356	46,352	370,663,914	2	2	4081
	% hinted	98%	98%	> 99%			
	inaccurate	0	0	0			
Sphinx	total	65,282	77,714	193,350,787	18	20	18,030
	% hinted	96%	96%	90%			
	inaccurate	0	0	0			

Table 3.2. Summary of benchmark workloads and hints. This table shows the number of read and write calls issued by each of the benchmarks as well as the number of blocks and bytes accessed by these calls. If one read requests the first half of a block and the next the rest of the block, it is counted as two one-block reads for a total of two blocks. For the reads, it also shows the percentage of the calls, blocks, and bytes that had been hinted in advance. Three of the benchmarks issue some inaccurate hints as described in the text which are recorded here. There is no number for the inaccurate Davidson read calls because the hinted blocks are simply abandoned at the end of the run; there is no hint cancellation call. In no case are the inaccurate hints a sizable portion of the total.

reads. In most cases, the applications perform all the reads hinted by a burst before issuing another burst of hints.[10] Burst size is important because it is a rough measure of the concurrency exposed by the hints. Table 3.3 reports details about the hints given by each application including burst sizes. As the table shows, five out of six of the applications have average burst sizes in the thousands, and even minimum burst sizes in the hun-

benchmark	hint calls	total segments	total blocks	bursts of hints			
				number	min. size	max. size	avg. size
Agrep	1349	1349	2922	1	2922	2922	2922
Gnuld	8322	8322	15,371	4	562	7402	3843
Postgres (20%)	2	4047	4455	2	409	4046	2228
Postgres (80%)	2	15,917	16,325	2	409	15,916	8163
Davidson	61	61	127,429	59	4178	6267	4213
XDataSlice	25	45,241	45,241	25	46	3448	1810
Sphinx	873	62,586	74,871	873	1	2477	86

Table 3.3. Hints issued by the benchmarks. This table characterizes each benchmark's hints. On the left, it shows the number of hint calls and how many sequential segments of how many blocks these calls disclosed. With only one exception, the applications give hints in large bursts where a burst is defined as a sequence of disclosures unbroken by a read of a hinted block. Thus, bursts are peaks in the number of outstanding hints through time. The table presents burst information on the right. The benchmarks cover a broad range of hint characteristics. Davidson issues one hint for 2089 block matrix each iteration and has two outstanding at a time. Postgres gives two hints in two bursts, one for the sequential read of the 409 block outer relation, and one with thousands of segments of one block each for the thousands of inner relation blocks read. Gnuld and Agrep issue large numbers of hints, but for very few blocks each and in only a small number of bursts. The typically huge average burst sizes and even minimum bursts sizes show that these application's hints expose large amounts of I/O concurrency.

dreds[11]. This is a huge amount of potential concurrency; it will be quite some time before the bandwidths of thousands of disks are needed to balance just one processor. The only exception is Sphinx, but, even there, most hints expose substantial concurrency.

The key point, as will be explored in depth in the next chapter, is that the concurrency exposed by these hints is orders of magnitude more than is needed today to balance disk and processor performance. Thus, these hints provide substantial potential for greater concurrency in the future. Further, in virtually every case the burst sizes are determined by dataset size. As dataset sizes grow over time, these applications will be able to give even larger bursts of hints and expose more concurrency.

Overall, experience with these applications strongly suggests that, in general, it is possible to annotate I/O-intensive applications with large numbers of useful disclosure hints.

---

[10] There are only two occasions when applications did not consume all outstanding hints before issuing more. Gnuld's second 'burst' is built up gradually while it consumes its first burst and therefore is not technically a burst as defined above. But, the gradual disclosure actually improves the value of the hints because they are all given over 500 accesses in advance in addition to exposing concurrency. Davidson always has a hint for the next iteration outstanding, so for most of its computation, it has hints for between 2089 and 4178 blocks outstanding at any time.

[11] XDataSlice has one slice that just nicks the corner of the dataset and results in a burst of just 46 blocks. But, these blocks represent all of the reads required to service the mouse click in this interactive application.

On the strength of this conclusion, in the following chapters I will explore how a system can take advantage of these disclosure hints to relieve the I/O bottleneck.

Chapter 4

Cost-Benefit Analysis for Informed Resource Management

In Chapter 3, I demonstrated that a broad collection of important, I/O-intensive applications can give hints that disclose most of their file reads in advance. The next step is for the file system to take advantage of these hints to improve I/O performance. Specifically, the file system could use the hints for

1. aggressive prefetching that adds latency-masking asynchrony and throughput-providing parallelism to the read workload,
2. caching blocks for reuse to avoid disk accesses,
3. clustering multiple accesses into fewer larger accesses that increases sequentiality and reduces CPU overheads, and
4. disk scheduling that reduces seek distances and thus increases sequentiality.

If the system had an infinite supply of cache buffers it would be relatively easy to accomplish all of these goals. As soon as hints arrived, blocks that were already cached could be locked down so they wouldn't be ejected, and buffers could be allocated to initiate prefetches for all missing blocks. The system could sort this large collection of prefetches to minimize seeks and cluster contiguous prefetches to create maximally-sized disk accesses.

Unfortunately, cache buffers are a limited resource. Given the large numbers of hints that the benchmark applications issue, only the largest machines could allocate buffers for all hinted blocks. These same applications, when processing larger datasets, could issue enough hints to exhaust any machine's supply of buffers. Furthermore, cache buffers are not idle; unhinted accesses depend on them to cache data for improved performance. The system has to make hard choices about how to allocate its limited pool of cache buffers. Should it prefetch or should it cache? If there is a cache miss, which block should it eject to free a buffer? Should it be a hinted block or one for which there is no hint but which was recently accessed? It would not be too hard to derive some benefit from hints, but fully utilizing them requires a resource manager that balances the use of cache buffers for all of these competing demands.

In this chapter, I develop a framework for resource management that continually applies cost-benefit analysis to find the right balance. I go on to show how to build a resource manager on top of this framework. In a nutshell, the idea is to estimate the cost or increase in I/O service time of ejecting a cached block to free a buffer and the benefit or decrease in I/O service time of using a buffer to initiate a prefetch, and reallocate a buffer from caching to prefetching when the benefit exceeds the cost.

4.1 A framework for I/O management by cost-benefit analysis

In general, cost-benefit analysis quantifies and compares the costs and benefits of pursuing a course of action. If the benefits exceed the costs then it is advantageous to proceed, otherwise it isn't. If there are several candidate courses of action, then cost-benefit analysis can determine which would provide the greatest net benefit and therefore would be most advantageous to pursue.

Cost-benefit analysis could be applied to the buffer allocation problem through a three-step process. First, compute the cost in increased elapsed time of ejecting each block. Second, compute the benefit in reduced elapsed time of allocating a buffer to each needed uncached block. Finally, pair the lowest-cost ejection with the greatest-benefit allocation to arrive at the replacement decision that leads to the greatest net reduction in elapsed time. Effectively, this approach would perform gradient descent on elapsed time as a function of the buffer allocations to the alternative uses.

Before describing, in the next subsections, the three key components of my I/O management framework based on this approach, I need to be a little more precise about the scope of the cache manager's control and the terms used in the framework.

Although the goal of this work is to minimize application wall-clock elapsed time, the cache manager does not control system components such as the process scheduler and virtual memory manager and so it cannot control elapsed time. The manager's control is limited to the manipulation of cache buffers and the initiation of disk accesses. Specifically, it can make replacement decisions which eject cached blocks to reallocate buffers to fetch uncached blocks. These decisions largely determine the time it takes to service an I/O request. Thus, a more precise statement of the problem facing the cache manager is to make the replacement decisions and initiate the disk accesses that minimize *I/O service time*. Thus, in the resource management framework, costs and benefits are changes in I/O service time, not changes in application elapsed time.

4.1.1 Independent estimates

Prefetching, clustering prefetches, and servicing demand accesses are alternative strategies for reducing I/O service time that require free buffers. And, the LRU queue and the caching hinted blocks are alternative strategies for determining which blocks to hold onto to reduce I/O service time. The first component of my framework for cost-benefit I/O management is independent estimates of the impact on I/O service time of applying each of these strategies.

The estimates are based on a model of system performance which I will describe in Section 4.2. In later sections, I will show how to derive the estimates for all of the strategies mentioned above, but here I give a quick summary of some of them. The benefit of allocating a buffer for prefetching is that it may mask disk latency. There is an estimate of how much this latency masking will reduce I/O service time. On the other hand, the cost of ejecting a hinted block is that it will have to be prefetched back. There is an estimate for how much this will increase I/O service time. The cost of taking a buffer from the LRU queue is a reduction in the hit ratio for unhinted accesses. There is an estimate of how much this will increase the average I/O service time of these accesses.

These independent value estimates avoid the need to consider all possibilities all at once. Thus, compared to allocation strategies that consider full replacement schedules or even just pairings of ejections and reallocations, they reduce the complexity of the cost-benefit analysis. Further, they ease the integration of new optimization strategies into the framework. For example, integrating virtual memory management into the cost-benefit framework would only require a new estimate for the cost of ejecting a virtual memory page; it would not require modifying the other estimates. Lastly, this same independence and extensibility enable modular implementation of system built on this framework.

4.1.2 A common currency for comparing estimates

In the cost-benefit framework, the independently determined costs and benefits are compared to determine which replacement, if any, would lead to the greatest net reduction in I/O service time. For these comparisons to be meaningful, the value estimates must all be expressed in the same terms; there must be a *common currency* for costs and benefits. Unfortunately, the estimates as described above are not directly comparable. For example, the cost of ejecting a hinted block is the one-time increase in I/O service time that results from prefetching that one block back, whereas the cost of taking a block from the LRU queue is an average increase in I/O service time for unhinted accesses. How can an average change be compared to a one-time increase? As, another example, the increase in I/O service time of prefetching back ejected blocks 1000 vs. 5000 accesses from now should be about the same. Does this mean there is no reason to eject one over the other? Clearly, I/O service time alone is an insufficient metric of comparison.

The missing factor is buffer usage. In optimizing buffer allocation, the real issue is not the absolute reduction in service time, but the reduction achieved per unit of buffer usage. Ejecting the block to be reaccessed in 5000 accesses frees a buffer for nearly 5000 instead of only 1000 accesses. The increase in I/O service time for the two cases may be about the same, but one frees a buffer that can be used elsewhere to reduce service time for a lot longer than the other does. This freed buffer is a benefit that offsets the cost of ejecting the block and should be taken into account when comparing value estimates. A metric of change in service time per unit of buffer usage accomplishes this.

To arrive at a formal definition of the common currency, first define the unit of buffer usage, or *bufferage*, as the occupation of one buffer for one inter-access period and call it one *buffer-access*. Then, define the *common currency* for the expression of all value estimates as the *magnitude of the change in I/O service time per buffer-access*. With this common currency, the buffer allocator can meaningfully compare the independent value estimates.

Because this common currency relates resource usage to the system goal of reducing service time, it eases extension of the manager. Adding support for remote files to the manager would not require new analysis of the relative merits of caching for remote vs. local files to arrive at some calibration that would allow the manager to choose between ejecting a local vs. a remote block. Instead, all that would be required are cost and benefit estimates of the change in I/O service time per buffer-access. Value estimates expressed in the common currency are already calibrated. If prefetching from the remote file system would reduce service time more per buffer-access than prefetching locally would, then prefetching remotely would be the right course of action.

4.1.3 An allocation algorithm

The final component of the framework is an allocation algorithm that can accept the many cost and benefit estimates, compare them at a global level and, identify the replacement that would produce the greatest net reduction in I/O service time. The algorithm must resolve two issues. First is how to merge multiple estimates of the value of a single buffer. This can happen when, for example, a block is on the LRU queue and there is also a hint that it will be reused. Then, there are both LRU and hinted cache estimates of its value. The second issue is how to find efficiently the need for a buffer with the greatest benefit and the available buffer with lowest cost. Because there may be thousands of hints and buffers, an exhaustive search could add substantial overhead. Section 4.2.6 addresses the first of these issues and Section 4.3.5 addresses the second.

4.1.4 Assembling the components

These three components are assembled to form the resource manager shown in Figure 4.1. Each potential buffer consumer and supplier has an *estimator* that independently computes the value of its use of a buffer. The buffer allocator continually compares these esti-

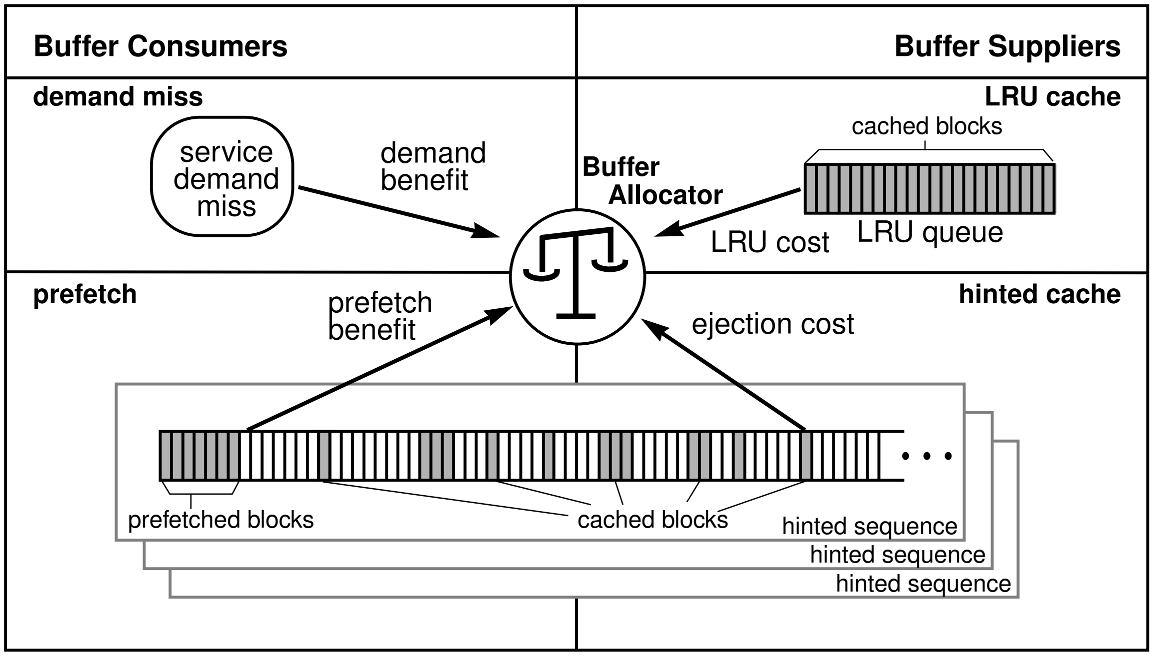


Figure 4.1. Informed cache manager schematic. Independent estimators express different strategies for reducing I/O service time. Demand misses need a buffer immediately to minimize the stall that has already started. Informed prefetching would like a buffer to initiate a read and avoid disk latency. To respond to these buffer requests, the buffer allocator compares their estimated benefit to the cost of freeing the globally least-valuable buffer. To identify this buffer, the allocator consults the two types of buffer suppliers. The LRU queue uses the traditional rule that the least recently used block is least valuable. In contrast, informed caching identifies the block whose next hinted access is furthest in the future as least valuable. The buffer allocator takes the least-valuable buffer to fulfill a buffer demand when the estimated benefit exceeds the estimated cost.

mates and reallocates buffers from the supplier with lowest cost to the consumer with greatest benefit when doing so would reduce I/O service time.

The primary buffer consumers are demand accesses that miss in the cache, and prefetches of hinted blocks. Additionally, once a buffer has been allocated for prefetching, clustered prefetches (not shown) may ask for additional buffers. The buffer suppliers are the traditional LRU cache, and the cache of hinted blocks.

Estimators for each buffer consumer and supplier independently determine the impact on I/O service time they anticipate if they respectively gain or lose a cache buffer and express this impact in terms of the common currency to the buffer allocator. Consumers compute the benefit or reduction in I/O service time per buffer-access they anticipate if allocated a buffer. Suppliers compute the cost or increase in I/O service time per buffer-access they anticipate if asked to give up a buffer.

From one perspective, this approach to resource management is like a market economy. Bidding to acquire cache buffers are the buffer consumers. Holding out are the buffer suppliers. The buffer allocator makes the market and manages trades. In this market there is no inflation or price gouging because all transactions are conducted using the gold-standard common currency that relates actual resource usage to the overall system goal of reducing I/O service time.

The market analogy does not go too far, though. Where markets tend to be chaotic and content to let any trade happen that is mutually beneficial, the buffer allocator in this cost-benefit approach maintains tight control to ensure that the consumer promising the greatest benefit gets the buffer that can be sacrificed at lowest cost. For this reason, the image of this algorithm performing gradient descent on the dynamically changing surface of I/O service time as function of the buffers allocated to the different uses is more revealing.

4.2 Cost-benefit analysis for informed prefetching and caching

The previous section motivated and gave an overview of cost-benefit I/O management, but it left out many details. In this section, I fill in those details from an ideal, theoretical perspective. First, I present the system model from which the various cost and benefit estimates are derived. I then go on to present the derivations for each estimator. Finally, I show how to compare the estimates at a global level to find the globally least valuable buffer and the globally most beneficial consumer. In the next section, I describe the adjustments to this ideal needed for a practical implementation.

4.2.1 System model & assumptions

The I/O manager's goal is to deploy its limited resources to minimize I/O service time. At its disposal are disk arms and file cache buffers. But, because I am primarily concerned with the exploitation of storage parallelism, I assume an adequate supply of disk arms and focus on the allocation of cache buffers. In Chapter 7, I will discuss the effect of limited array size on the value estimates developed here.

For the purposes of the model, I make certain assumptions about the system. In particular, I assume a modern operating system with a file buffer cache running on a uniprocessor with sufficient memory to make available a substantial number of cache buffers. With respect to workload, consistent with my emphasis on read-intensive applications, I assume

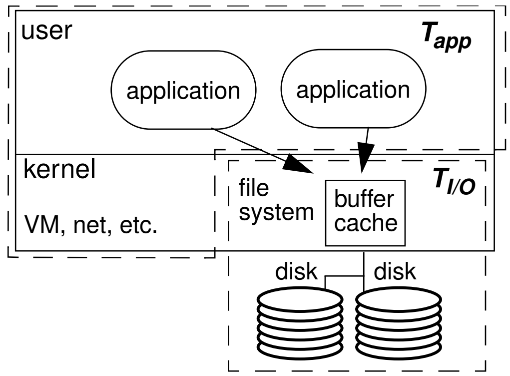


Figure 4.2. Components of system execution. In our simplified system model, application elapsed time, T, has two components, computation and I/O. The computational component, T_{app}, consists of user-level application execution plus time spent in kernel subsystems other than the file system. The I/O component, $T_{I/O}$, consists of time spent in the file system, which includes time for reading blocks, allocating blocks for disk I/Os, servicing disk interrupts, and waiting for a physical disk I/O to complete.

that all application I/O accesses request a single file block that can be read in a single disk access and that the requests are not too bursty. Further, I assume that system parameters such as disk access latency, T_{disk}, are constants. Lastly, as mentioned above, I assume enough disk parallelism for there never to be any congestion (that is, there is no disk queueing). As we shall see, distressing as these assumptions may seem, the policies derived from this simple system model behave well in a real system, even one with a single congested disk.

The elapsed time, T, for an application is given by

$$T = N_{I/O}(T_{app} + T_{I/O}) \ , \tag{4.1}$$

where $N_{I/O}$ is the number of I/O accesses, T_{app} is the inter-access application CPU time,[1] and $T_{I/O}$ is the time it takes to service an I/O access. Figure 4.2 diagrams the system model, and Table 4.1 gives the definitions for the model variables.

In the model, the I/O service time, $T_{I/O}$, includes some system CPU time. In particular, an access that hits in the cache experiences time T_{hit} to read the block from the cache. In the case of a cache miss, the block needs to be fetched from disk before it may be delivered to the application. In addition to the latency of the fetch, T_{disk}, these requests suffer the computational overhead, T_{driver}, of allocating a buffer, queuing the request at the

[1] Note that $T_{app} \equiv T_{CPU}$ in the terminology of the paper, "Informed Prefetching and Caching" [Patterson95].

symbol	meaning
T	total application elapsed time
$N_{I/O}$	total application I/O requests
T_{app}	average application CPU time between I/O requests
$T_{I/O}$	average I/O service time including file system CPU overhead
T_{hit}	time to service an I/O request that hits in the buffer cache
T_{driver}	CPU time to allocate a cache buffer and perform a disk I/O including interrupt servicing (does not include disk latency)
T_{disk}	latency of a disk access
T_{miss}	time to service an I/O request that misses in the buffer cache = $T_{hit} + T_{driver} + T_{disk}$
n	number of buffers in the LRU queue
$H(n)$	hit ratio for the LRU queue as a function of queue size, n
T_{LRU}	time to service an unhinted request through the LRU cache
x	prefetching depth, or number of accesses in advance that a prefetch is initiated
T_{stall}	time the CPU goes idle waiting for an I/O to complete
$T_{pf}(x)$	time to service a request for a prefetched block as a function of the prefetch depth, x
$P(T_{app})$	the *prefetch horizon*, or the minimum prefetching depth, as a function of T_{app}, that eliminates CPU stalls for I/O
$\Delta T_{eject}(x)$	change in service time that results from ejecting a hinted block as a function of the depth at which it is prefetched back, x
y	the number of accesses in advance that a hinted block is ejected
r_d, r_h	the rate of respectively demand and hinted accesses
s_i, $\|s_i\|$	segment i of the LRU queue, and the number of buffers in the segment
h_i	cache hits to buffers in LRU segment s_i
A	number of unhinted accesses; the denominator when computing $H(n)$
$\hat{P}$	fixed, system-wide, upper-bound prefetch horizon

Table 4.1. Performance model symbol definitions. These symbols are listed in the order they are defined in the text and will be used throughout this chapter.

drive, and servicing the interrupt when the disk operation completes. The total time to service an I/O access that misses in the cache, T_{miss}, is the sum of these times:

$$T_{miss} = T_{hit} + T_{driver} + T_{disk} \quad . \tag{4.2}$$

In the terms of this model, deallocating an LRU cache buffer makes it more likely that an unhinted access misses in the cache and must pay a delay of T_{miss} instead of T_{hit}. Allocating a buffer for prefetching can mask some disk latency. Ejecting a hinted block from the cache means an extra disk read will be needed to prefetch it back later. Piggybacking a prefetch on another access can save CPU overhead and avoid disk latency. In the next sections, I quantify these effects. But, first note that any delay in servicing a demand miss is added directly to the I/O service time of the request, so there can be no better use for a buffer than initiating a read to service the miss. Acknowledging this, the benefit of allocating a buffer for a demand miss is taken to be infinite, and requests for buffers for demand accesses are not denied.

4.2.2 The cost of shrinking the LRU cache

Over time, the portion of demand accesses that hit in the cache is given by the cache-hit ratio, $H(n)$, a function of the number of buffers in the cache, n. Given $H(n)$, the average time to service a demand I/O request, denoted $T_{LRU}(n)$, is

$$T_{LRU}(n) = H(n)T_{hit} + (1 - H(n))T_{miss} \ . \tag{4.3}$$

Taking the least-recently-used buffer from a cache employing an LRU replacement policy results in an increase in the average I/O service time of

$$\Delta T_{LRU}(n) = T_{LRU}(n-1) - T_{LRU}(n) \tag{4.4}$$

$$= (H(n) - H(n-1))(T_{miss} - T_{hit}) = \Delta H(n)(T_{miss} - T_{hit}) \ . \tag{4.5}$$

Because $H(n)$ varies as the I/O workload changes, the LRU cache estimator dynamically estimates $H(n)$ and the value of this expression as explained in Section 4.3.

Every access that the LRU cache is deprived of this buffer will, on average, suffer this additional I/O service time, so the bufferage freed for this increase in elapsed time is one buffer-access. Thus, in terms of the common currency, the cost of taking a buffer from the LRU queue is the magnitude of the change in I/O service time given by Equation 4.5, and

$$\mathrm{Cost}_{LRU} \equiv \frac{|\Delta T_{LRU}(n)|}{bufferage} = \Delta T_{LRU}(n) \ . \tag{4.6}$$

4.2.3 The benefit of prefetching

Prefetching a block can mask some of the latency of a disk read, T_{disk}. Thus, in general, an application accessing such a prefetched block will stall for less than the full T_{disk}. Suppose the system uses x buffers to prefetch x accesses into the future. Then, stall time is a function of x, $T_{stall}(x)$. Substituting this reduced stall for the disk service time in Equation (4.2), we find that the service time for a hinted read is also a function of x,

$$T_{pf}(x) = T_{hit} + T_{driver} + T_{stall}(x) \ . \tag{4.7}$$

Then, assuming that prefetches of blocks 1 to x-1 have already been initiated, the benefit of using an additional buffer to prefetch x instead of x-1 accesses in advance is the change in service time,[2]

$$\Delta T_{pf}(x) = T_{pf}(x) - T_{pf}(x-1) \tag{4.8}$$

$$= T_{stall}(x) - T_{stall}(x-1) \ . \tag{4.9}$$

A key observation is that the application's data consumption rate is finite. Even when all data are cached or when prefetching has completely masked disk access time, application computation, T_{app}, and the system overheads of servicing cache hits, T_{hit}, and performing disk accesses, T_{driver}, limit the rate at which the application issues requests. Thus, prefetches do not need to be started infinitely far in advance to be sure that they will complete before the application requests the prefetched data. Typically, the application reads a block from the cache in time T_{hit}, does some computation, T_{app}, and pays an overhead, T_{driver}, for future accesses currently being prefetched. Thus, even if all intervening accesses hit in the cache, the soonest one might expect a block x accesses into the future to be requested is $x(T_{app} + T_{hit} + T_{driver})$. Under the assumption of no disk congestion, a prefetch of this xth future block would complete in T_{disk} time. Thus, the stall time when requesting this block is at most

---

[2] This formulation is slightly different from that presented in the paper, "Informed Prefetching and Caching" [Patterson95] in that it compares prefetching x vs. x-1 accesses in advance instead of x+1 vs. x. This reformulation lets us ask: assuming blocks 1 … x-1 have been prefetched, what is the benefit of prefetching the next block x accesses in advance. The change results in (x-1) appearing in equations instead of (x+1), but has no other material effect.

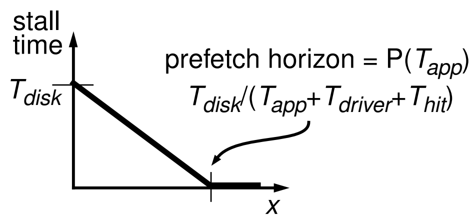


Figure 4.3. Worst case stall time and the prefetch horizon. Data consumption is limited by the time an application spends acquiring and consuming each block. This graph shows the worst case application stall time for a single prefetch *x* accesses in advance, assuming adequate I/O bandwidth, and therefore no disk queues. There is no benefit from prefetching further ahead than the prefetch horizon.

$$T_{stall}(x) \le T_{disk} - x(T_{app} + T_{hit} + T_{driver}) \quad . \tag{4.10}$$

Figure 4.3 shows this worst case stall time as a function of *x*.

This stall-time expression allows us to define the distance, in terms of future accesses, at which informed prefetching yields a zero stall time. I call this distance the *prefetch horizon*, $P(T_{app})$, recognizing that it is a function of a specific application's inter-access CPU time.

$$P(T_{app}) = \frac{T_{disk}}{(T_{app} + T_{hit} + T_{driver})} \quad . \tag{4.11}$$

When sufficient disk bandwidth is available, there is no benefit from prefetching more deeply than the prefetch horizon. Thus, it is easy to bound the impact of informed prefetching on effective cache size; prefetching a stream of hints will not lead informed prefetching to acquire more than $P(T_{app})$ buffers.

Equation (4.10) is an upper bound on the stall time experienced by the *x*th future access assuming that the intervening accesses are cache hits and do not stall. However, it overestimates stall time in practice. In steady state, multiple prefetches are in progress and a stall for one access masks latency for another so that, on average, only one in *x* accesses experiences the stall in Equation (4.10). Conceptually, a stall on the first block masks the stall for subsequent accesses which, by the assumption of no disk queues, are proceeding without delay on other disk drives. Figure 4.4 diagrams this effect. Thus, the average stall per access as a function of the prefetch depth, $P(T_{app}) > x > 0$, is

$$T_{stall}(x) = \frac{T_{disk} - x(T_{app} + T_{hit} + T_{driver})}{x} \quad . \tag{4.12}$$

access number	time (1 time-step = $T_{app} + T_{hit} + T_{driver}$)																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	I	⊝	⊝	⊝	⊝	⊗															
2	I	-	-	-	-	C	X														
3	I	-	-	-	-	C		X													
4						I	-	-	⊝	⊝	⊗										
5							I	-	-	-	-	⊗									
6								I	-	-	-	-	⊗								
7											I	-	-	⊝	⊝	⊗					
8												I	-	-	-	-	⊗				
9													I	-	-	-	-	⊗			
10																I	-	-	⊝	⊝	⊗

I : initiate prefetch - : prefetch in progress C : block arrives in cache X : consume block O : stall

Figure 4.4. Average stall time when prefetching in parallel. This figure illustrates informed prefetching as a pipeline. In this example, three buffers are used to prefetch three blocks concurrently and T_{app} is assumed fixed. At time T=0, the application gives hints for all its accesses and then requests the first block. Prefetches for the first three accesses are initiated immediately. The first access stalls until the prefetch completes at T=5, at which point the data is consumed and the prefetch of the fourth block begins. Accesses two and three proceed without stalls because the latency of prefetches for those accesses is overlapped with the latency of the first prefetch. But, the fourth access stalls for $T_{stall} = T_{disk} - 3(T_{app}+T_{hit}+T_{driver})$. The next two accesses don't stall, but the seventh does. The application settles into a pattern of stalling every third access. In general, when x prefetches occur in parallel, a stall occurs once every x accesses.

At $x = 0$, there is no prefetching, and $T_{stall}(0) = T_{disk}$. Similarly, for $x \geq P(T_{app})$, $T_{stall}(x) = 0$. Figure 4.5 shows that this estimate, although based on a simple model, is a good predictor of the actual stall time experienced by a synthetic application running on a real system.

We can now plug Equation (4.12) into Equation (4.9) and obtain an expression for the impact on I/O service time of acquiring one additional cache buffer to increase the prefetching depth,

$$\Delta T_{pf}(x) \approx \begin{cases} x = 1 & -(T_{app} + T_{hit} + T_{driver}) \\ x \leq P(T_{app}) & \dfrac{-T_{disk}}{x(x-1)} \\ x > P(T_{app}) & 0 \end{cases} . \tag{4.13}$$

The values are negative because the benefit of prefetching is a reduction in elapsed time. The benefit drops off roughly as the inverse square of the prefetching depth until the prefetch horizon is reached. Beyond that point, there is no benefit from deeper prefetching under the assumption of adequate disk bandwidth.

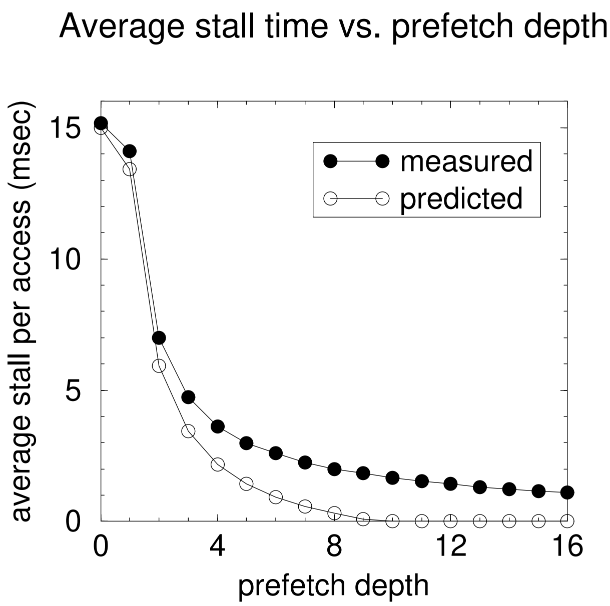


Figure 4.5. Predicted and measured per-access stall time. To verify the utility of Equation (4.12), I measured the stall time of a synthetic microbenchmark as a function of prefetch depth. The benchmark does 2000 reads of random, unique 8 KByte blocks from a 320 MByte file striped over 10 disks. It has 1 millisecond of computation between reads, so T_{app} = 1 msec, and, for the system described in Chapter 6, $T_{hit}+T_{driver}$ = 569 µsec and T_{disk} = 15 msec. Overall, Equation (4.12) has a maximum error of about 2 milliseconds, making it is a good predictor of actual stall time. The equation underestimates stall time because the underlying model neglects disk contention and variation in T_{disk}. Chapter 7 explores these issues in greater depth.

Every access that this additional buffer is used for prefetching benefits from this reduction in the average I/O service time. Thus, Equation (4.13) is the change in I/O service time per buffer-access, and the benefit of allocating a buffer for prefetching in terms of the common currency is the magnitude of this change. Thus,

$$\text{Benefit}_{pf} \equiv \frac{|\Delta T_{pf}(x)|}{bufferage} = |\Delta T_{pf}(x)| \quad . \tag{4.14}$$

4.2.4 The cost of ejecting a hinted block

Although there is no benefit from prefetching beyond the prefetch horizon, caching any block for reuse can avoid the cost of prefetching it back later. Thus, ejecting a block increases the service time for the eventual access of that block from a cache hit, T_{hit}, to the read of a prefetched block, T_{pf}. In determining the change in service time that results from ejecting a block, what matters, therefore, is not how far in the future the ejected block will be accessed, but the number of accesses in advance that it will be prefetched back. If the block is prefetched back x accesses in advance, then the increase in I/O service time caused by the ejection and subsequent prefetch is the differences between a prefetch and a hit,

$$\Delta T_{eject}(x) = T_{pf}(x) - T_{hit} \tag{4.15}$$

$$= T_{driver} + T_{stall}(x) \ . \tag{4.16}$$

Although the stall time, $T_{stall}(x)$, is zero when x is greater than the prefetch horizon, T_{driver} represents the constant CPU overhead of ejecting a block no matter how far into the future the block will be accessed.

This increase in service time from ejecting a block, $\Delta T_{eject}(x)$, does not affect every access; it is a one time cost borne by the next access to the ejected block. Thus, to express this cost in terms of the common currency, we must average this change in I/O service over the accesses that a buffer is freed. If the hint indicates the block will be read in y accesses, and the prefetch happens x accesses in advance, then ejection frees one buffer for a total of y–x buffer-accesses. Conceptually, if the block is ejected and its buffer lent where it reduces average I/O service time, then it will have y–x accesses to accrue a total savings that exceeds the cost of ejecting the block. Thus, the cost of ejecting a hinted block y accesses before its use is

$$\text{Cost}_{eject} \equiv \frac{\Delta T_{eject}(x)}{bufferage} = \frac{T_{driver} + T_{stall}(x)}{y - x} \ , \tag{4.17}$$

where $T_{stall}(x)$ is given by Equation (4.12). As we shall see in Section 4.3.3, the implementation simplifies this estimate further to eliminate the dependence on the variable x.

4.2.5 The benefit of informed clustering

Clustering multiple contiguous accesses into one large sequential access both eliminates the CPU overhead of performing multiple accesses and maximizes disk throughput for those accesses. Hints provide the opportunity to piggyback future accesses on current ones to take advantage of spacial locality even when the accesses are widely separated in time. What is the benefit of allocating a buffer to cluster a future prefetch with a current access?

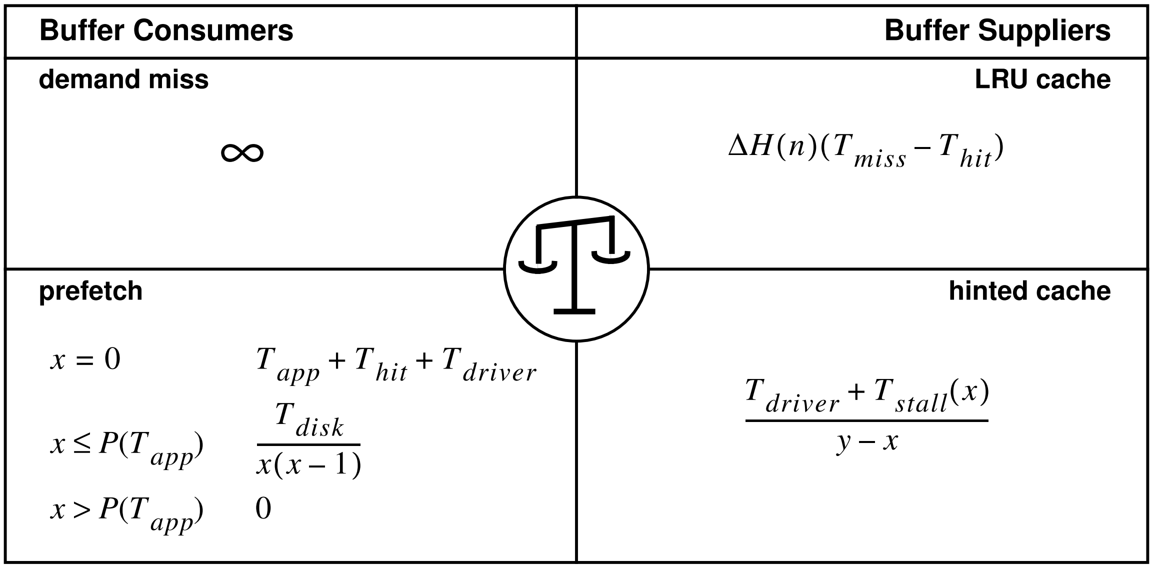


Figure 4.6. Local value estimates. Shown above are the locally estimated magnitudes of the change in I/O service time per buffer-access for the buffer consumers and suppliers of Figure 4.1. Because demand misses must be satisfied immediately, they are treated as having infinite value. The remaining three formulas are the absolute values of Equations (4.5), (4.14), and (4.17), for the LRU cache, hinted cache, and prefetch estimates, respectively.

If the decision to fetch or prefetch a block has already been made, then the cost, T_{driver}, of performing a disk read will be incurred. Any blocks that could piggyback on this read avoid most of the disk related CPU costs. If there are hinted blocks that can cluster with the first block, and they are not prefetched now in such a cluster, their later prefetch will incur the full CPU overhead of performing a disk access and possibly the cost of any unmasked disk latency. These are exactly the costs considered when deciding whether to eject a hinted block. Furthermore, clustering allocates a buffer for the same number of accesses that ejecting the block would free one. Thus, the benefit of informed clustering a prefetch of a block y accesses before it is accessed is the same as the cost of ejecting the same hinted block,

$$\text{Benefit}_{clpf} = \text{Cost}_{eject} = \frac{T_{driver} + T_{stall}(x)}{y - x} \quad . \tag{4.18}$$

4.2.6 Global buffer value and the min-max buffer

Figure 4.6 summarizes the value estimates in Equations (4.5), (4.14), and (4.17) which the various estimators use to determine the local value of a buffer. Even though these estimates are expressed in terms of the common currency, they are not yet ready for comparison at a global level because the unit of bufferage, a buffer-access, only has local meaning. For example, when considering ejecting a block *y* accesses before it is needed, accesses were counted within the single hint sequence, not at a global level. Fortunately, multiplying hint estimates by the rate of the hinted accesses, r_h, and the LRU estimate by the rate of unhinted demand accesses, r_d, normalizes the estimates to the same time basis and gives the estimates global meaning. Intuitively, if 95% of all accesses in the system are unhinted, the cost to overall performance of reducing the LRU cache hit ratio is much greater than it would be if only 5% of the accesses depended on the LRU cache. Thus, it makes sense to scale the estimates in proportion to their share of total system activity.

The buffer allocator uses the normalized estimates to decide when to take a buffer from a supplier and use it to service a request for a buffer. For example, deallocating a buffer from the LRU cache and using it to prefetch a block would cause a net reduction in aggregate I/O service time if $r_h \cdot \mathrm{Benefit}_{pf} > r_d \cdot \mathrm{Cost}_{LRU}$. For the greatest reduction, though, the globally least-valuable buffer should be allocated. Unfortunately, it is not obvious which is the least-valuable buffer. If there are multiple hints for the same block, which should be used to value that block's buffer? If a hint refers to a block that happens to be on the LRU list, which value estimate should be used?

To answer these questions, start by considering a single hint sequence that refers to the same block twice. If the block is ejected, it will have to be fetched back for the first hint. Ejected or not, prefetched back in time or not, after the first hinted access the block will certainly be in the cache. The first access pays T_{driver} and stalls and subsequent accesses find the block in the cache (unless the block is ejected again, which would be a separate decision). Thus, the increase in service time that would result from ejecting the block now is determined by the first access. Furthermore, ejecting one block only frees one buffer and it only frees it until the first access takes the buffer back. Therefore, the first hint alone determines the ejection cost for the block, and the ejection cost is not, for example, the sum of the cost estimates based on the two hints. Indeed, even if there are a hundred hints for the block, the cost of ejecting the block is determined by the cost of prefetching it back in for the first hint. Note that the first hint leads to the greatest of the cost estimates based on any of the hints.

This analysis extends directly to multiple hint sequences from multiple processes that refer to the same block. One of the processes will request the block first and the other processes will find the block in the cache. Again, ejecting a block only frees one buffer and only needs to be fetched back once. The cost of ejecting the block is determined by the next access to the block. Because, at a global level, cost estimates are scaled by the access rate to the hint sequence, the anticipated first access leads to the greatest normalized cost estimate.

In general, ejecting a block frees only one buffer and incurs at most one fetch cost to bring back in. These observations apply no matter what estimators are used. Thus, costs aren't additive nor are the freed resources. The cost of ejecting a particular block is the maximum of the various independent normalized cost estimates determined by the different estimators.

The globally least-valuable buffer is the one whose maximum valuation is minimal over all buffers. Hence, the replacement policy chooses this *min-max buffer* for ejection if the benefit exceeds the maximum estimated cost. Although this replacement policy seems to require every estimator to compute cost estimates for every buffer, in practice, as we shall see in Section 4.3.5, only a small number of cost estimates need to be computed to identify the min-max buffer and the overhead of this replacement policy is reasonable.

4.2.7 An example: emulating MRU replacement

As an aid to understanding how informed caching 'discovers' good caching policy, we show how it exhibits MRU (most-recently-used) behavior for a repeated access sequence. Figure 4.7 illustrates an example.

At the start of the first iteration through a sequence that repeats every N accesses, the cache manager prefetches out to the prefetch horizon. After the first block is consumed, it becomes a candidate for replacement either for further prefetching or to service demand misses. However, if the hit-ratio function, *H(n)*, indicates that the least-recently-used blocks in the LRU queue don't get many hits, then these blocks will be less valuable than the hinted block just consumed. Prefetching continues, replacing blocks from the LRU list and leaving the hinted blocks in the cache after consumption.

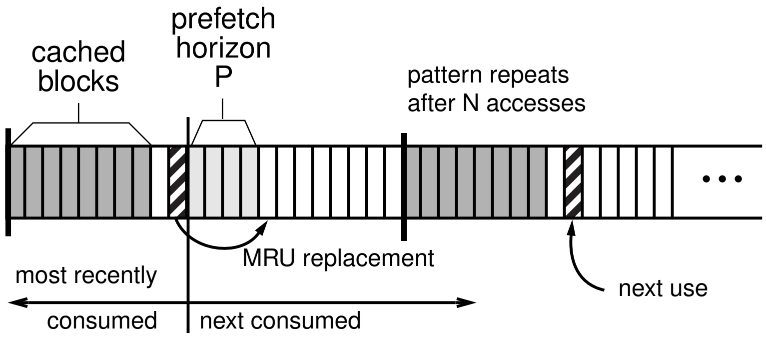


Figure 4.7. MRU behavior of the informed cache manager on repeated access sequences. The number of blocks allocated to caching for a repeated access pattern grows until the caching benefit is not sufficient to hold an additional buffer for the N accesses before it is reused. At that point, the least-valuable buffer is the one just consumed because its next access is furthest in the future. This buffer is recycled to prefetch the next block within the prefetch horizon. A wave of prefetching, consumption, and recycling moves through the accesses until it joins up with the blocks still cached from the last iteration through the data.

As this process continues, more and more blocks are devoted to caching for the repeated sequence and the number of LRU buffers shrinks. For most common hit-ratio functions, the fewer the buffers in the LRU cache, the more valuable they are. Eventually, the cost of taking another LRU buffer exceeds the cost of ejecting the most-recently-consumed hinted block. At the next prefetch, this MRU block is ejected because, among the cached blocks with outstanding hints, its next use is furthest in the future.

At this point, a wave of prefetching, consumption, and ejecting moves through the remaining blocks of the first iteration. Because the prefetch horizon limits prefetching, there are never more than the prefetch horizon, $P(T_{app})$, buffers in this wave. There is no risk that the cache manager will cannibalize the cached blocks to prefetch further into the future. Thus, the MRU behavior of the cache manager is assured. The cache manager effectively balances the use of buffers for prefetching, caching hinted blocks, and LRU caching.

The informed cache manager discovers MRU caching without being specifically coded to implement this policy. This behavior is a result of valuing hinted, cached blocks and ejecting the block whose next access is furthest in the future when a buffer is needed. These techniques will improve cache performance for arbitrary access sequences where blocks are reused with no particular pattern. All that is needed is a hint that discloses the access sequence.

4.3 Implementation of cost-benefit I/O management

My implementation of informed prefetching and caching, which is called TIP, replaces the unified buffer cache (UBC) in version 3.2c of the Digital UNIX operating system. To service unhinted demand accesses, TIP creates an LRU estimator to manage the LRU queue and estimate the value of its buffers. In addition, TIP creates an estimator for every process that issues hints to manage its hint sequence and associated blocks.

In the following sections, I describe how these implemented estimators arrive at their cost estimates and explain how they differ from the ideal estimates described in the previous section. I conclude with the description of an algorithm with a reasonable overhead that implements the min-max valuation of buffers. Chapter 5 describes the implementation in more detail.

4.3.1 The LRU estimator

LRU block replacement is a stack algorithm [Mattson70], which means that the ordering of blocks in the LRU queue is independent of the length of the queue. Consequently, cache hits occur at the same depth in the queue for all cache sizes. By observing where cache hits occur in a queue of N buffers, it is possible to make a history-based estimate of $H(n)$, the cache-hit ratio as a function of the number of buffers, n, in the cache for any cache size less than N, $0 < n < N$. Specifically, $H(n)$ is estimated by the sum of the number of hits with stack depths less then or equal to n divided by the total number of unhinted accesses, A.

In TIP, the number of buffers in the LRU queue varies dynamically. When the queue is short, TIP needs to know whether a larger queue would have achieved a higher hit ratio so that it can determine whether it would be beneficial to grow the LRU queue. To determine $H(n)$ for caches larger than the current size, TIP uses ghost buffers. Ghost buffers are dataless buffer headers which serve as placeholders to record when an access would have been a hit had there been more buffers in the cache. My use of ghost buffers was inspired by work by Maria Ebling on caching in a distributed file system [Ebling94]. The length of the LRU queue, including ghosts, is limited to the total number of buffers in the cache.

Unfortunately, efficiently determining where in an LRU queue hits occur is not easy. After every cache miss, the buffer is released to the tail of the queue. If all accesses

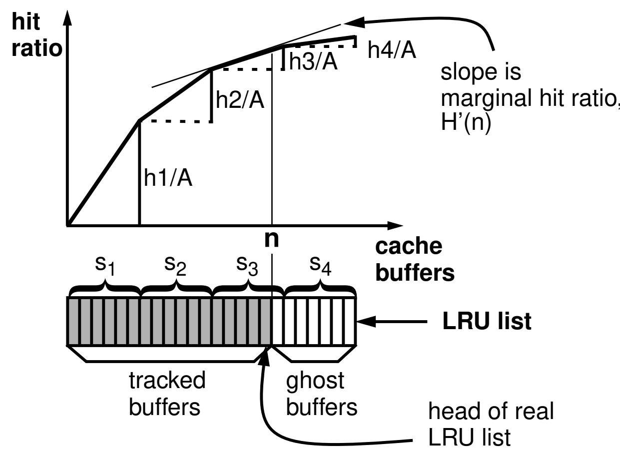


Figure 4.8. Piecewise estimation of *H(n)*. The LRU list is broken into segments, s_1, s_2, s_3, … Each buffer is tagged to indicate which segment it is in. The removal of the buffer on a cache hit lowers the buffer count for its segment below the target count. When a buffer is released to the tail of the LRU queue, the other buffers overflow from one segment to the next until an under-full segment stops the cascade. The tag on a buffer is updated when the buffer overflows from one segment to the next. When there is a cache hit in segment *i*, the segment hit count, h_i, is incremented. That segment's contribution to the hit ratio is then h_i/A, where *A* is the total number of unhinted accesses. See Section 5.2.6 for more detail.

missed, the position of a buffer in the queue would be equal to the number of accesses since the buffer was last referenced because every access would release a new buffer to the tail of the queue. However, when a hit occurs, the accessed buffer is promoted to the tail of the queue. The position of a buffer in the queue depends on how many cache hits there have been between the buffer in question and the tail of the queue. For example, consider a buffer that is second from the tail of the queue. After 100 accesses to the first buffer, the buffer is still second from the tail. On the other hand, 100 accesses to the 200th position move the second buffer to position 102. Depending on the particular sequence of hits and misses, the buffer could be anywhere in between.

The most obvious technique for determining the queue position of a cache hit is to run down the queue and count the buffers from the tail to the hit buffer. This could mean traversing a linked list of hundreds or thousands of buffers on every cache hit. Utilizing this approach would add substantial CPU overhead to the system.

To reduce overhead, hit counts are recorded not by individual queue depths, but by disjoint intervals of queue depths, called segments. Shown in Figure 4.8, this allows a piecewise estimation of *H(n)*. Such averaging of hits over a segment of the queue has the advantage of smoothing over little spikes and plateaus in the function *H(n)*.

The cost of losing an LRU buffer given in Equation (4.5) requires an estimate of $\Delta H(n) = H(n) - H(n-1)$. Because the piecewise estimate of $H(n)$ is linear within any segment, $\Delta H(n)$ is equal to the slope of $H(n)$ within a segment and is the same for all buffers in the segment if we ignore the discontinuity at the boundary between segments. This slope is the increase in $H(n)$ over the segment divided by the number of buffers in the segment. Thus,

$$\Delta H(n) \approx \frac{(h_i / A)}{|s_i|} = \frac{h_i}{A|s_i|} \tag{4.19}$$

where n falls within segment s_i, h_i is the number of hits in segment s_i, A is the total number of unhinted accesses, and $|s_i|$ represents the number of buffers in segment s_i. In TIP, $|s_i| = 100$.

In a running system, file deletions and other events may remove many buffers from some segments. The question arises: should buffers be shifted back from higher-numbered segments to fill the vacancies in the lower-numbered segments? The issue is which segment should get credit if there is a subsequent hit on one of the shifted buffers. Concretely, suppose buffer b is in segment s_3 when 20 blocks cached in segment s_2 are deleted and b could be shifted back into s_2. Should an access to b be scored a hit in s_2 or s_3? The key observation is that if there had not at one time been enough buffers in the LRU queue to fill segments s_1 and s_2 and push b into segment s_3, then the block in b would have been ejected, b recycled and the access to b would have been a miss instead of a hit. The LRU estimator asks the question: What would be my hit ratio if I had n buffers. Shifting b back to segment s_2 and scoring the hit there would mislead the estimator into believing that it needed fewer buffers than it did to get a hit to b. Leaving b in segment s_3, records the fact that enough buffers to push b into segment s_3 were needed to get a hit to b. Thus, TIP does not shift buffers back.

One consequence of not shifting buffers is that there is not always a direct correlation between the number of buffers currently in the LRU queue, n, and the segment number of the buffer at the head of the LRU queue. I just argued that, in fact, it is the segment number of a buffer and not the number of buffers that is the key parameter when estimating the cost of losing the buffer at the head of the LRU queue. Thus, properly speaking, cost of losing a buffer is a function of the segment number, i, and

$$\Delta H(s_i) = \frac{h_i}{A|s_i|} \tag{4.20}$$

should replace $\Delta H(n)$ in the expression for the cost of losing an LRU buffer.

A final complexity arises because, in general, $H(n)$ may not be similar to the smoothly increasing function suggested by Figure 4.8. There is often a large jump in the hit ratio when the entire working set of an application fits into the buffer cache. The value of $\Delta H(s_i)$ would be high for the segment that captures the working set. A large LRU queue would successfully defend its buffers and continue caching the working set. However, if the LRU queue is initially not large enough to hold the working set as the number of hits mount and it becomes clear that there is a working set to cache, then a strictly local estimate of $\Delta H(s_i)$ would fail to see the benefit of growing the LRU cache. Consequently, the LRU queue could fail to grow and might remain at its small initial size no matter how long the working set continued to exist. The end result is that TIP's buffer allocations would depend on initial conditions. To avoid this undesirable dependence, TIP's LRU estimator uses a simple mechanism to avoid being stuck in a local minima that ignores the benefit of a much larger cache: $\Delta H(s_i)$ is modified to be $max_{j \geq i}\{\Delta H(s_j)\}$ in the LRU value estimate. That is, the value of the marginal hit ratio is rounded up to the value of any larger marginal hit ratio occurring deeper in the LRU stack. If the LRU cache is currently small, but a larger cache would achieve a much higher hit ratio, this mechanism encourages the cache to grow.

Applying these modifications to Equation (4.6) yields the following expression for the cost of losing an LRU buffer when the head of the LRU list is in segment s_i:

$$\text{Cost}_{LRU} \approx \Delta T_{LRU}(i) \approx max_{j \geq i}\{\Delta H(s_j)\}(T_{miss} - T_{hit}) \tag{4.21}$$

$$\approx max_{j \geq i}\left\{\frac{h_j}{A|s_j|}\right\}(T_{miss} - T_{hit}) \quad . \tag{4.22}$$

4.3.2 The prefetching estimator

Equations (4.11) and (4.13) give precise expressions for, respectively, the prefetch horizon and for the benefit of prefetching. In practice, variations in model parameters can lead to either over- or underestimation of the actual stall. Specifically, variations in application execution time, T_{app}, can speed or slow the rate of data requests. Also, runs of cached blocks larger than the prefetch horizon can suspend prefetching which eliminates the CPU overhead of prefetching, T_{driver}, from the equations and reduces the interaccess period. On the other hand, blocks may need to be prefetched for other applications which may add additional T_{driver} overhead to the interaccess period. To eliminate the overhead of measuring T_{app} and increase tolerance to bursts of application requests and runs of cached blocks, TIP assumes T_{app} = 0 and discounts the overhead of prefetching other blocks, T_{driver}, to arrive at a static, system-wide upper-bound on the prefetch horizon,

$$\hat{P} = \frac{T_{disk}}{T_{hit}} \quad . \tag{4.23}$$

This is an upper-bound prefetch horizon for an application that does negligible computation, has most of its data cached, but an infinitely large array at it disposal. It is reasonable for TIP to be generous when deciding how deep to prefetch because, as we will see, the system parameters of T_{disk} and T_{hit} for the TIP testbed lead to a modest value of $\hat{P}$. On future systems with different values for these parameters, there may be more incentive to tighten these estimates. Section 7.2 explores this issue in more depth.

Finally, the assumption of negligible application CPU time leads to a very small benefit of T_{hit} for allocating the first buffer for prefetching, x=1 in Equation (4.5). In contrast, the much larger benefit of adding a second buffer is $T_{disk}/2$. Recognizing that multiple buffers will be used for prefetching, I define a benefit for allocating the first buffer for prefetching that leads to a smooth benefit function.

Together, these implementation considerations lead to this variant of Equation 4.6

$$\text{Benefit}_{pf} = \left|\Delta T_{pf}(x)\right| \approx \begin{cases} x = 1 & T_{disk} \\ x \le \hat{P} & \dfrac{T_{disk}}{x(x-1)} \\ x > \hat{P} & 0 \end{cases} \quad . \tag{4.24}$$

4.3.3 The hinted cache estimator

Equation (4.17) in Section 4.2.4 expresses the cost of ejecting a hinted block in terms of y, the number of accesses till the hinted read, and x, how far in advance the block will be prefetched back and is repeated here for easy reference:

$$\text{Cost}_{eject} \equiv \frac{\left|\Delta T_{eject}(x, y)\right|}{bufferage} = \frac{T_{driver} + T_{stall}(x)}{y - x} \quad . \tag{4.25}$$

The dependence on x poses two difficulties. First, it ties the estimate of the cost of ejecting a block to an estimate of when the block will be prefetched back. A precise determination would require knowledge of other costs and benefits that the prefetch would be bidding against for a buffer. Even if this could be done, it would destroy the independence of the estimators. To maintain estimator independence and eliminate the possibly substantial overhead of determining x, I simplify the expression for the cost of ejecting a hinted block by assuming that the prefetch back will occur at the (upper-bound) prefetch horizon, $\hat{P}$. The stall for such a prefetch is zero, so, for $y > \hat{P}$, we have,

$$\text{Cost}_{eject} = \frac{T_{driver}}{y - \hat{P}} \quad . \tag{4.26}$$

If the block is already within the prefetch horizon, $y \le \hat{P}$, I assume that the prefetch will occur at the next access, that is (y-1) accesses in advance. Then, to maintain consistency between this estimate of when the block will be prefetched back and the estimate of the stall that will result on the prefetch, I apply the assumptions of Section 4.3.2 used to compute $\hat{P}$, and set $T_{app} = 0$ and neglect T_{driver} in the expression for the stall:

$$T_{stall}(x) \approx \frac{T_{disk} - xT_{hit}}{x} = \frac{T_{disk}}{x} - T_{hit} \quad . \tag{4.27}$$

Thus, for $1 < y \le \hat{P}$, we have,

$$\Delta T_{eject}(y) = T_{driver} + \frac{T_{disk}}{y-1} - T_{hit} \quad . \tag{4.28}$$

Unfortunately, using this equation could lead to prefetching a block back shortly after ejecting it. Even though the expression assumes that the prefetch back will occur on the next access, ejecting a block only to prefetch it back immediately would be wasteful. To avoid this thrashing, there should be hysteresis in the valuations; that is, we need

$$|\Delta T_{eject}(y)| > |\Delta T_{pf}(y-1)| \quad \text{, or, substituting,} \tag{4.29}$$

$$T_{driver} + \frac{T_{disk}}{y-1} - T_{hit} > \frac{T_{disk}}{(y-1)(y-2)} \quad . \tag{4.30}$$

Unfortunately, this inequality does not hold for all possible values of T_{driver}, T_{disk}, and T_{hit}. To guarantee robustness for all values of these parameters greater than zero, I choose to add T_{hit} to $\Delta T_{eject}(y)$ for $1 < y < \hat{P}$.[3] Assembling the pieces, we have,

$$\text{Cost}_{eject} \approx \begin{cases} y = 1 & T_{driver} + T_{disk} \\ 1 < y \le \hat{P} & T_{driver} + \dfrac{T_{disk}}{y-1} \\ y > \hat{P} & \dfrac{T_{driver}}{y - \hat{P}} \end{cases} \quad . \tag{4.31}$$

4.3.4 Implementation of informed clustering

In Section 4.2.5, I argued that the benefit of informed clustering, where the prefetch of one block is piggybacked on the prefetch of another, is the same as the cost of ejecting the block if it were already cached. In keeping with this argument, TIP uses the same equations to implement both the benefit of informed clustering and the cost of ejecting a hinted block, namely Equation (4.31).

The informed clustering estimator is not like the others in that it is not constantly bidding for a buffer. It can only use a buffer when another prefetch is about to be sent to disk.

---

[3] It turns out that this inequality does hold within the prefetch horizon where it is applied. The T_{hit} term could be included and hysteresis would be preserved. I do not expect this to have a measurable impact because, in practice, blocks within the prefetch horizon are valuable and are rarely, if ever, ejected.

Buffer Consumers	**Buffer Suppliers**
demand miss ∞	**LRU cache** $max_{j \geq i}\left\{\frac{h_j}{A\lvert s_j \rvert}\right\}(T_{miss} - T_{hit})$
prefetch $x = 0$: T_{disk} $x \leq \hat{P}$: $\frac{T_{disk}}{x(x-1)}$ $x > \hat{P}$: 0	**hinted cache** $y = 1$: $T_{driver} + T_{disk}$ $1 < y \leq \hat{P}$: $T_{driver} + \frac{T_{disk}}{y-1}$ $y > \hat{P}$: $\frac{T_{driver}}{y - \hat{P}}$

Figure 4.9. Local value estimates in the implementation. Shown above are the local estimates of the value per buffer-access for the buffer consumers and suppliers of Figure 4.1. These estimates are easy-to-compute approximations of the exact estimates of Figure 4.6.

At that time, a check for hints for contiguous blocks is made, and if any are found, and the benefit of the clustered prefetch exceeds the value of the globally least-valuable buffer, then the buffer is allocated and a cluster built. This special allocation path may cause clustering to be allocated a buffer before a regular, higher-benefit prefetch of some other block within the prefetch horizon. Strictly speaking, buffers should be allocated to prefetches and clusters in order of greatest benefit. But, doing so in this case while still supporting clustering could require initiating multiple different prefetches and then going back to build clusters around these multiple prefetches. It might even require intermingling these two activities. Certainly, a system could be built to support this, but it would add substantial complexity. Instead, I choose to rely on the fact that clustered prefetches never allocate more than 7 blocks at a time and so are unlikely to deplete the supply of buffers available for regular prefetching.

Figure 4.9 summarizes the equations used to estimate buffer values in TIP. The benefit of informed clustering is not shown because it is not constantly bidding for buffers and because it is the same as for the hinted cache.

4.3.5 Identifying the min-max buffer

Section 4.2.6 identified the min-max buffer as being the globally least valuable in the cache and therefore the buffer that should next be replaced when a new buffer is needed[4]. Unfortunately, as presented there, it appears that identifying the min-max buffer and therefore making a replacement decision requires all estimators to estimate the value of all cached blocks and then sorting these blocks by value. This would entail thousands of calculations for every block allocation which would add far too much overhead to be practical. In this section, I show how only a small number of cost estimates need be computed to identify the min-max buffer. Indeed, when few data blocks are shared among processes, each replacement decision requires as few as one cost-estimate calculation.

The first observation is that individual estimators can easily rank blocks by value without calculating actual value estimates. The LRU estimator ranks blocks on the LRU list in the order they appear on the list and blocks not on the list have zero value. A hint estimator ranks blocks in order of their next appearance in the hint sequence and blocks that don't appear in the sequence have zero value. Thus, an estimator with appropriate data structures can quickly identify blocks of no value to it as well as the block with the minimal positive value without making any cost calculations.

If no estimator values a particular block, then the block has no global value, and it is a candidate for immediate replacement; it is the min-max buffer. If there are multiple such blocks, then the algorithm can choose randomly from among them. Such replacement decisions require no cost calculations.

In the more interesting case, every block is valued by some estimator. Suppose, as a restricted case, every block is valued by exactly one estimator. Then, the max valuation across estimators for a particular block is the value assigned by the one estimator that values it. Effectively, we can ignore the zero-value estimates of the other estimators. Among the blocks that an estimator values, one will have the minimal valuation for that estimator.[5] Clearly, the block with the globally minimal of the max valuations (the min-max block) will be one of these blocks that is minimally valued by an estimator. Then, to find the min-max block, it is sufficient for each estimator to compute the value of its minimally

[4] Recall that the min-max buffer is the one whose maximal valuation by any estimator is minimal among all of the buffers.

valued block and then, at a global level, to determine which of these minimally valued blocks is least valuable.

In the above case, it appears that each estimator needs to compute one value estimate (for its minimally valued block) per replacement decision. This is already a great improvement over computing value estimates for all blocks. In fact, it is not even necessary to make this many calculations. Value estimates only change when some state relevant to an estimator changes such as a block it values is ejected, a block it desires is prefetched, or an application consumes a hinted block. Therefore, the only estimator that needs to compute a new value estimate from one replacement decision to the next is the one whose minimally valued block was ejected. Thus, only one cost calculation is needed per replacement decision for this restricted case.

In general, multiple estimators may value the same block. For example, there may be a hint for a block that also happens to be on the LRU queue. In this case, there is no guarantee that the min-max block will be minimally valued by some estimator; each estimator's minimally valued block may be highly valued by some other estimator and all estimators may value some block less than they value the min-max block. Thus, if we only consider for replacement blocks that are minimally valued, we may not even consider the min-max block for replacement. How, then, can we find the min-max block without computing the max valuation for all blocks?

The solution is to assume that only one estimator values any block and then use lazy evaluation to catch violations to this assumption. Here's how this works. Each estimator computes the value of its minimally valued block and the globally least valuable of these is selected as the candidate for replacement. Thenceforth, the estimator whose block was selected acts as if the block had been ejected from the cache; it no longer considers the block when choosing its minimally valued block; the estimator ceases *tracking* the block[6]. Before the block is actually ejected, a check is made to see if any other estimator values

[5] In practice, an estimator may compute the same valuation for multiple blocks. For example, the LRU estimator computes the same value for all blocks within a queue segment. Nevertheless, it can pick one it thinks is least valuable: the one at the head the LRU list. In principle, it doesn't matter which of a set of equivalently valued blocks is replaced. For the allocation algorithm to work, all that is needed is some way, even random selection, to pick one block from the set of equivalently valued blocks.

[6] The notion of tracking is further clarified below.

```
start:
   /* all estimators have already computed the cost of their */
   /* minimally valued tracked block and these costs have */
   /* been normalized for global comparison */
   estimator with lowest global cost {
      names its least valuable block;
      ceases tracking the named block;
      computes value of new least valuable block it is tracking;
      submits new value for normalization for global comparison;
   }
   /* check that no estimator values named block more highly */
   foreach estimator {
      query to see if estimator values named block more highly;
      if yes {
         /* block is saved from replacement */
         valuing estimator {
            begins tracking saved block;
            if saved block is now its least valuable {
               computes value of new least valuable block;
               submits new value for normalization for
                     global comparison;
            }
         }
         goto start;
      }
   }
```

Figure 4.10. Algorithm for identifying the min-max buffer. The algorithm optimistically assumes that the globally least-valuable buffer, the min-max buffer, is the one valued least by the estimator with lowest global cost. Before reallocating the buffer, the algorithm checks that, in fact, no other estimator values the block more highly. In most cases, this approach requires just one or a few cost calculations per allocation decision.

the block more highly. If one does, the candidate block is saved from ejection, the valuing estimator begins *tracking* the block if it isn't already doing so, and the algorithm loops to pick a new candidate for replacement. If no other estimator values the block more highly, then this valuation is the maximal valuation for that block across all estimators. Furthermore, because this maximal valuation is less than at least some valuation by some estimator for every other block in the cache, this candidate block is the globally least valuable according to the min-max valuation of blocks and it should be ejected. Finally, note that because each iteration that picks a new candidate eliminates one block from consideration by one estimator, forward progress is guaranteed. Figure 4.10 presents pseudocode for the algorithm.

Observe that it is the mechanism of tracking that prevents an estimator from repeatedly picking the same block for replacement. When identifying its minimally valued block, an estimator only chooses from among the blocks it is tracking. Effectively, each estimator only sees the subset of all cached blocks that it is tracking. Once an estimator picks a buffer for replacement, the buffer disappears from its view of the cache. Because the algorithm just described for identifying the min-max block uses lazy evaluation to check estimators before ejecting a block, the algorithm correctly identifies the min-max buffer no matter which estimator is initially tracking the buffer. However, it is important that some estimator be tracking every buffer. Because estimators only choose from among buffers they are tracking when nominating buffers for replacement, a buffer untracked by any estimator will never be picked for replacement and will stay in the cache forever.

4.4 Conclusion

Disclosure hints provide a great opportunity for optimizing I/O performance through informed prefetching, clustering, and caching. Unfortunately, these optimizations compete with each other as well as traditional LRU caching for cache buffers. How should buffers be allocated to take maximum advantage of hints while preserving buffers for the LRU queue? In this chapter, I present my solution to this problem.

In approaching the problem, I wanted to find a reasoned solution and to avoid rule-of-thumb approaches that require fiddling with tunable parameters. The key insight was that because buffer managers most commonly reallocate blocks one-at-a-time, ejecting one block to load another, they present a splendid opportunity to apply cost-benefit analysis. If estimates for both the cost of ejecting a block and the benefit of giving the freed buffer to another block could be found, then it would be clear which block, if any, to eject and which block should replace it.

My framework for resource management by cost-benefit analysis has three key components. First, a collection of independent estimators dynamically estimate either (1) the benefit (reduction in application I/O service time) of allocating a buffer for prefetching or a demand access, or (2) the cost (increase in I/O service time) of taking a buffer from the LRU queue or the cache of hinted blocks. Because the estimators are independent, they are relatively simple and the system is extensible because the addition of a new estimator does not require changes in the existing ones.

For these estimates to be comparable at a global level they must be expressed in terms of a common currency, the second component of the framework. This common currency relates usage of the cache buffer resource to the system goal of reducing I/O service time. Expressed in terms of this common currency, estimates indicate how much service time will change per unit of resource freed or consumed, or, colloquially, how much bang for the buck.

The third and final component is an algorithm that finds the globally least valuable buffer and reallocates it for the greatest benefit when the estimated benefit exceeds the anticipated cost. The first issue is determining at a theoretical level that when multiple estimators value the same buffer, the buffer's global value is the maximum assigned by any estimator. Thus the globally least valuable buffer is the one whose maximal valuation is minimal across all buffers. Even with independent estimates of the cost or benefit of ejecting or gaining an individual buffer, this global valuation could require that value estimates be computed for all of the many hundreds or thousands of cache buffers. Fortunately, individual estimators can rank buffers without computing their value. I showed how to construct an algorithm that takes advantage of this fact, optimistically assumes that only one estimator values any buffer, and uses lazy evaluation to catch violations of this assumption. In the common case, this algorithm requires only one cost calculation by one estimator per buffer allocation event.

With this framework, I developed an analytical performance model and used it to derive estimates in terms of the common currency for the benefit of prefetching and building clusters of prefetches, and for the cost of ejecting a hinted block or shrinking the LRU cache. I then showed how to adapt the analytical estimates for use in an implementation. In particular, I showed how to dynamically estimate the hit ratio for LRU queue as a function of queue size.

This cost-benefit approach to resource management provides the reasoned solution I sought. Analytical performance models provide the basis for buffer allocation decisions. The cache manager dynamically applies the models to reallocate buffers to perform gradient descent on I/O service time as a function of the allocation of buffers among the competing demands.

Chapter 5

Implementation of Informed Prefetching and Caching

Application disclosures of future file reads provide opportunities for: aggressive parallel prefetching; clustering of multiple prefetches into fewer, larger accesses; disk scheduling of multiple prefetches; and caching data for reuse. Chapter 4 provided the theoretical and practical framework for a system that applies run-time cost-benefit analysis to exploit all of these opportunities while preserving buffers in the LRU queue for unhinted accesses. In this chapter, I show what is needed to implement a working system around this framework and I describe how my implementation, called TIP, meets these needs. I will also share some of the particular problems faced in implementing TIP inside the Digital UNIX operating system. I start with a brief overview of how TIP fits in the rest of the kernel and the functionality it must provide before going on to consider how it provides that functionality.

5.1 Overview

TIP replaces the Unified Buffer Cache (UBC) in version 3.2c of the Digital UNIX (DU) kernel as shown in Figure 5.1. TIP provides conventional file caching service for the several file systems that DU supports including UFS, a variant of the Fast File System (FFS) [McKusick84] and the Network File System (NFS) [Sandberg85]. Application file requests first pass through the Virtual File System (VFS) [Sandberg85, Kleiman86] which forwards them to the target file system (UFS, NFS, or some other). The target asks the

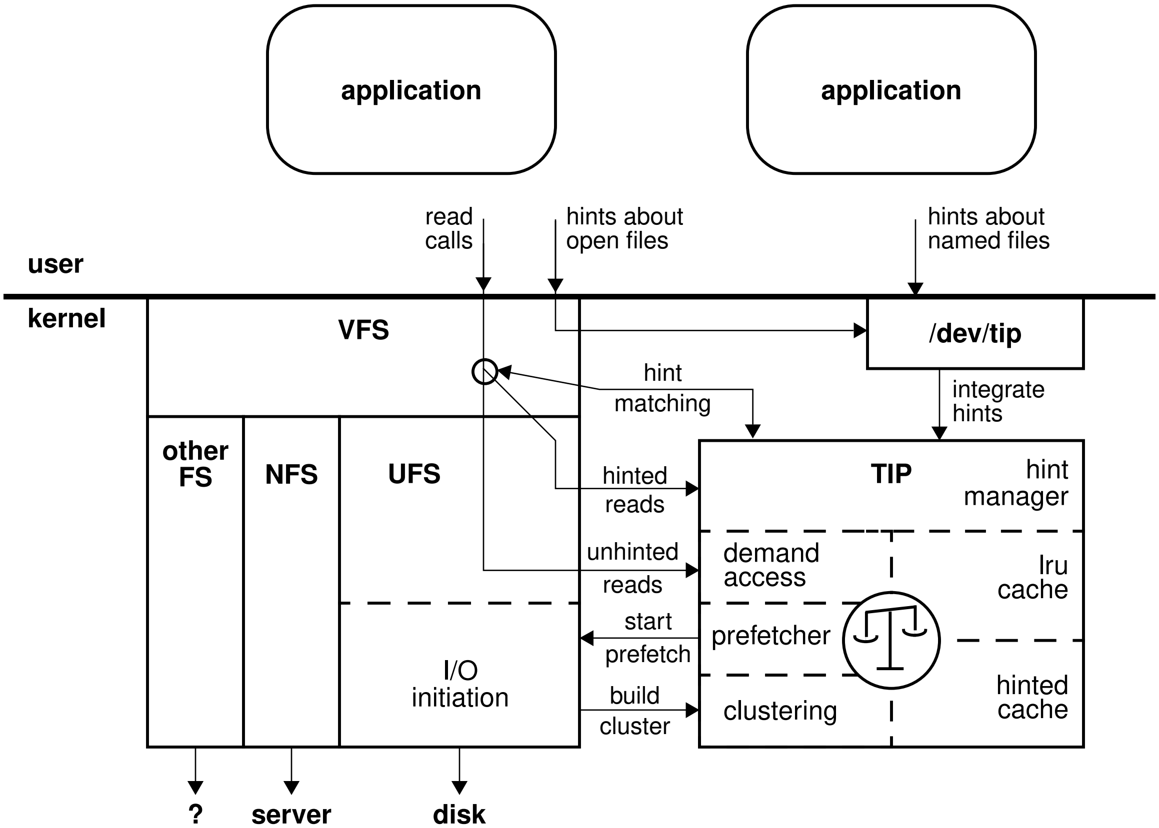


Figure 5.1. The TIP informed cache manager in the Digital UNIX operating system. When applications don't hint, TIP provides conventional caching service for the various file systems, including the UNIX File System (UFS) and the Network File System (NFS). Application requests for data first go through the Virtual File System (VFS) layer which forwards them to the appropriate file system which checks TIP to see if the requested data is cached. If not, a buffer is allocated for the missing block and the file system initiates an I/O to load the block into the buffer. Application hints for open files are delivered via VFS in ioctls on the file descriptor. Hints for unopened files are delivered in ioctls to the TIP pseudo-device (/dev/tip). See text for further details.

cache if it has the referenced block. If not, the cache allocates a new buffer which the file system fills with the requested block.

The cache is organized by *vnode* (the structure in VFS which describes a file) and logical offset within the file. Thus, when checking the cache for a block, file systems specify the vnode and offset of the block in question. To find blocks quickly, both TIP and the original UBC use a conventional hash table much like the one in the BSD 4.3 file system [Leffler89]. But, the cache in that system was organized by disk block, not logical file block. This logical-block organization has certain implications for access clustering which will be discussed in Section 5.2.2.

hint	target	description
TIPIO_SEG	/dev/tip	batch of <offset, length> segments for a named file
TIPIO_FD_SEG	open file descriptor	batch of <offset, length> segments for an open file
TIPIO_MFD_SEG	/dev/tip	batch of <fd, offset, length> segments for multiple open files
TIPIO_CANCEL	/dev/tip or open file descriptor	cancels segment at head of hint list; used when a hint turns out to be erroneous

Table 5.1. Ioctl calls in the disclosure hint interface. Disclosure hints describe future requests in the same terms as the existing file interface. Thus, they must specify the file, the starting offset of the access, and the length of the sequential access before a *seek* to a new offset. This information is relayed to the file system via *ioctl* system calls using one of the hints specified in this table. Hints specifying a file by name are given in *ioctl* calls to the /dev/tip pseudo-device, whereas *ioctls* giving hints about open files can target those files directly.

To conventional caching functionality, TIP adds the exploitation of hints for I/O optimizations. These hints disclose the file and byte range that future file reads will access. Table 5.1 summarizes the hint interface; see Chapter 3 for full details. The order in which hints are given indicates the order of the hinted accesses. Hints for named files are given to a pseudo-device, /dev/tip. VFS forwards hints on open file descriptors to the pseudo-device. The device stores both types of hints in kernel data structures which it forwards to the TIP hint manager.

A hook in the VFS layer lets the hint manager match read requests against the hints. In this way, TIP knows how the application is progressing through its hints. Monitoring reads is better done in the VFS layer than from within the cache manager because it allows matching of exact byte ranges; the cache only sees requests for whole blocks. If the read matches a hint, then the read takes a special hinted-read path to and through the relevant file system. This special path serves two purposes. First, it allows TIP to make a positive hand-off of the hinted data to the application. Without this, a data race could result if matching a hint in the VFS layer lead the cache manger to eject data it thought was no longer needed before the application had a chance to read the data from the UFS layer. The second purpose of the hinted-read path is that it can be optimized for the anticipated request. For example, cache lookup operations can be avoided because the matching hint already points directly to the requested block. How a hint points to a block and other aspects of hint management will be described in more detail in Section 5.3.1.

In addition to providing a hinted-read path, file systems that support TIP must also provide a routine for TIP to call when it wants to prefetch a block. TIP's decision to prefetch is governed by its cost-benefit buffer management which is described in Section 5.2. But, in the DU kernel structure, the cache manager does not have direct access to the disk. Instead, TIP calls a file-system-specific routine which performs such functions as mapping the logical file block to a physical disk block and building a request to send to the disk device driver. Before actually queuing a request at the disk, this routine calls back to TIP to give it the opportunity to build a cluster prefetch. I have only added such support for TIP to the UFS file system, but other researchers are working to add support to other file systems including NFS [Rochberg97].

As a last note on the relationship between TIP and the rest of the system, I should point out that the original UBC shares memory pages with the Virtual Memory (VM) system, and the partition between the two varies dynamically. TIP does not yet have an estimator for value of VM page usage and so it is not yet able to duplicate this functionality. Developing such an estimator is an interesting area for future research. For now, TIP sets a static partition of pages between the cache and VM and thus manages a fixed number of cache pages or buffers. Nevertheless, this unification of virtual memory and file objects means that memory objects backed by files, such as mapped files, reside in the cache and not virtual memory. TIP must manage such objects even if it does not accept hints for them. This issue is discussed further in Section 5.3.3.

5.2 Implementation of cost-benefit buffer allocation

The cost-benefit buffer allocator, as described in Chapter 4, relies on estimators for help when making allocation decisions. Buffer consumers (prefetches and demand accesses) estimate the benefit in terms of the common currency[1] that they would derive from a buffer. Meanwhile, buffer caches, which could supply a buffer, estimate the common-currency cost of giving up the least valuable of their tracked buffers. When the bene-

[1] The common currency is defined to be the change in I/O service time per buffer-access. See Section 4.1.2 for details.

Buffer Consumers	**Buffer Suppliers**
demand miss ∞	**LRU cache** $max_{j \geq i} \left\{ \frac{h_j}{A\lvert s_j \rvert} \right\} (T_{miss} - T_{hit})$
prefetch $x = 0 \quad T_{disk}$ $x \leq \hat{P} \quad \frac{T_{disk}}{x(x-1)}$ $x > \hat{P} \quad 0$	**hinted cache** $y = 1 \quad T_{driver} + T_{disk}$ $1 < y \leq \hat{P} \quad T_{driver} + \frac{T_{disk}}{y-1}$ $y > \hat{P} \quad \frac{T_{driver}}{y - \hat{P}}$

Figure 5.2. Local value estimates in the implementation. Shown above are the local estimates of the value per buffer-access for the buffer consumers and suppliers that were developed in Chapter 4 and first summarized in Figure 4.9. Recall that the benefit of adding a block to a cluster prefetch (not shown) is the same as the cost of ejecting a block from the hinted cache.

fit exceeds the cost, the allocator takes a buffer from the lowest-cost estimator and gives it to the greatest-benefit consumer.

For review, Figure 5.2 presents the equations developed in Chapter 4 that are used to estimate costs and benefits in the TIP implementation. In this section, I describe how TIP applies these equations.

Both the benefit of prefetching and the cost of ejecting a hinted block depend on the number of accesses until the block is accessed. Thus, the first step in applying these equations is converting the hints to a sequence of accesses. For simplicity, TIP treats every reference to an 8 KByte file block as one access as shown in Figure 5.3. When TIP matches an application read to a hint, it stores the index of the last access read in the variable *conIndex* which stands for consumption index. Then, when computing the benefit of prefetching a block for a given access or the cost of ejecting a block referenced by a given access, the value of, respectively, x or y needed for the equations is the difference between this *conIndex* and the index of the access in the sequence.

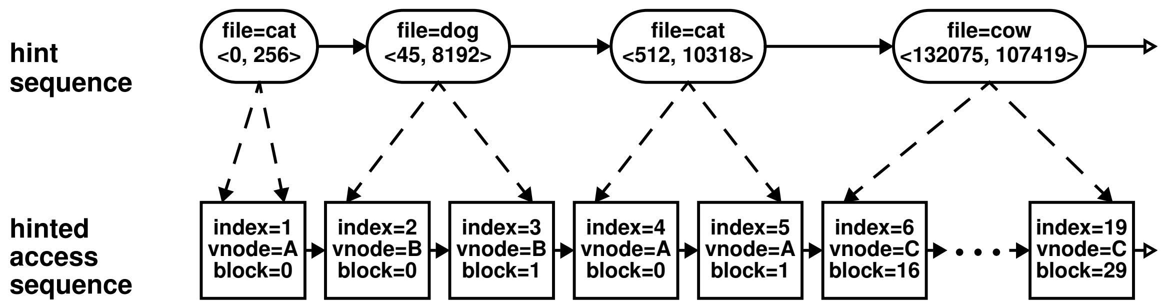


Figure 5.3. The hinted access sequence. TIP hint ejection-cost and prefetching-benefit estimates are parameterized by the number of single-block accesses until a hinted block is referenced. But, application hints specify a file and an <offset, length> couple for a byte range within the file; they do not indicate whether the application will read the byte range in one large access or multiple small ones. Thus, the hints do not directly correspond to a sequence of block accesses. TIP expands the hints to an access sequence by counting every reference to a block as an access. The index of an access is its position in this sequence. This figure gives an example of how TIP expands a hint sequence into a hinted access sequence.

The next step for a hint estimator is to declare the benefit it would derive from being given one buffer and the cost it would suffer from giving up one buffer. To do this, the estimator must determine which block it would most like to prefetch and which block is its least-valuable. Because the hinted access sequence is central to computing value estimates, the hint estimators are organized around accesses and not blocks or buffers. Thus, as will be described more later, the hint estimator maintains an ordered list of the accesses for which it would like to prefetch a block and not a list of missing blocks. Also, instead of tracking blocks in buffers, it tracks accesses. Thus, the estimator keeps a list of accesses it is tracking. Its least-valuable tracked buffer is the one caching the block referred to by the tracked access that is furthest in the future.

For its part, the LRU estimator uses the LRU queue as an ordered list of the blocks it is tracking. Its least valuable tracked buffer is the one that was least recently used which is the one at the head of the LRU list. The cost of giving up that buffer is determined by which segment of the queue that buffer is in as described in the last chapter in Section 4.3.1.

To match the estimator with greatest benefit with the estimator with lowest cost, the global allocator separately ranks consumer and supplier estimators by value as shown in Figure 5.4. There is one estimator for each process that issues a sequence of hints and one for the LRU queue. Note that hint estimators may be both a consumer of buffers for

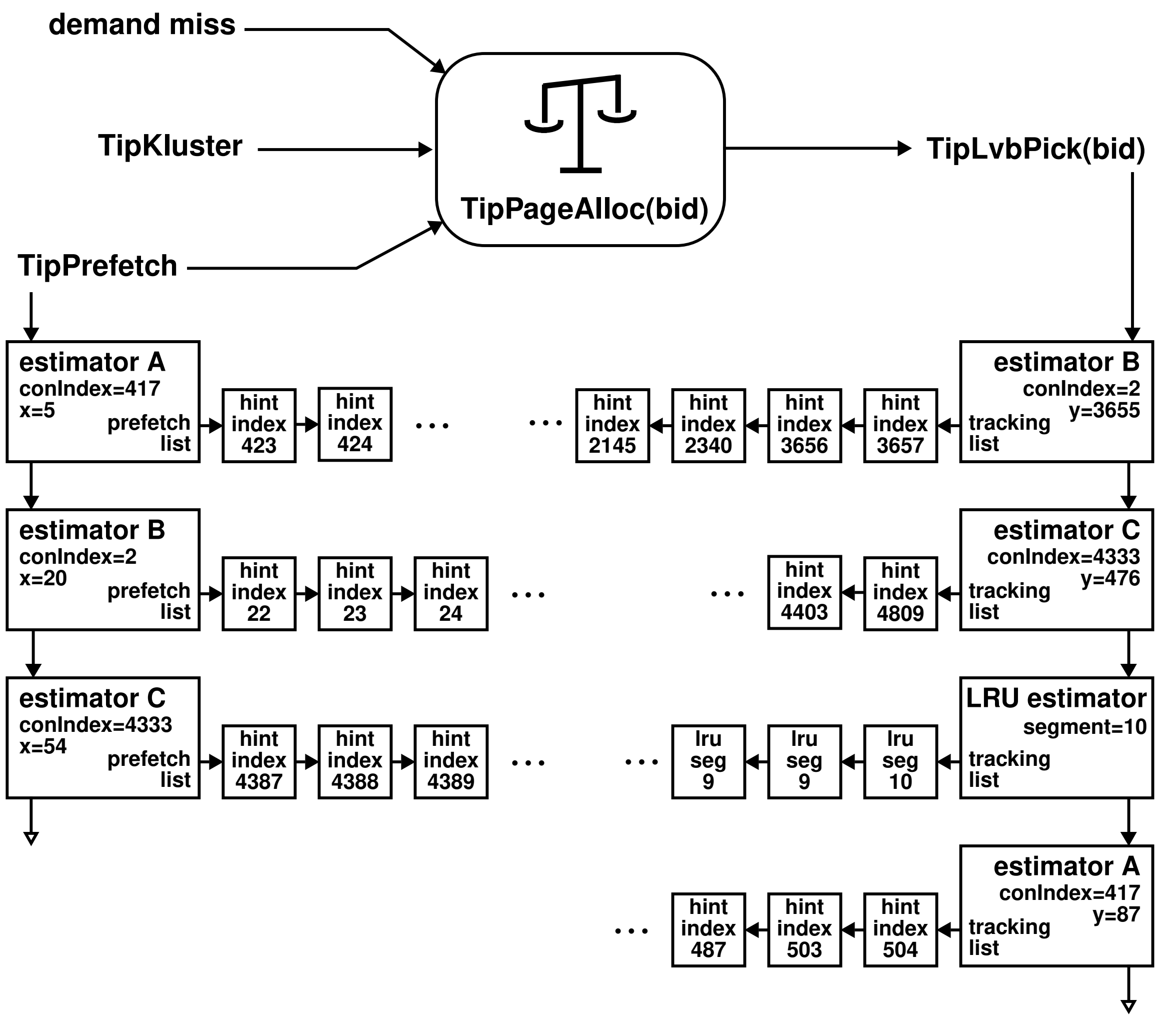


Figure 5.4. Schematic of cost-benefit buffer allocator. A procedure called *TipPageAlloc* serves as the mediator between buffer consumers and suppliers. When a demand miss, informed clustering or informed prefetching need a buffer, they call *TipPageAlloc* with a bid expressing the benefit they would derive from a buffer. *TipPrefetch* bids for the estimator ranked with the greatest benefit in a list of estimators that would like a buffer for prefetching. *TipLvbPick* picks a buffer from the estimator ranked least-valuable in a list of the estimators that can supply buffers if the bid exceeds the cost. See text for further details.

prefetching and a supplier of buffers from its cache of hinted blocks. The estimators maintain lists as needed to generate their estimates.

The prefetcher, invoked by calling *TipPrefetch*, calls *TipPageAlloc* to bid the greatest benefit for a buffer. *TipPageAlloc* first calls *TipLvbPick* to find the globally least valuable buffer (the min-max buffer) and then reallocates the buffer if the bid exceeds the cost. Other consumers only want buffers occasionally and they also call *TipPageAlloc* to bid for a buffer. For example, UFS does this when there is a demand miss for which it needs a buffer.[2] And, the cluster prefetcher does this when a disk read on which it can piggyback prefetches is about to be queued at the disk.

In Section 5.2.1, I describe the prefetcher and what is needed to generate prefetching benefit estimates. Section 5.2.2 identifies the requirements for clustered prefetching. Section 5.2.3 reviews the algorithm for identifying the min-max buffer. Section 5.2.4 identifies the functions estimators must provide to support the allocation algorithm. Section 5.2.5 describes the nexus data structure which ties estimators to buffers to make finding value estimates for blocks efficient. Finally, Sections 5.2.6 and 5.2.7 describe the different strategies that the LRU and hinted cache estimators use to take advantage of the nexus data structure and provide the functions needed by the allocation algorithm.

5.2.1 Informed prefetching

The prefetching module ranks hint estimators according to the normalized benefit of prefetching the first missing block in their hinted access sequence. The benefits are computed using the prefetch value estimate in Figure 5.2. This benefit estimate depends on x, the number of accesses until the missing block will be accessed. Thus, computing the benefit of prefetching for a hint estimator requires knowledge of which is the next missing block in the sequence, the block's location or index in the sequence, and the index of the hinted block last consumed by the application.

To make this information easily available, the hints are expanded into the hinted access sequence which is stored as a linked list associated with the estimator. TIP runs down the list to find the first missing block. It stops at the prefetch horizon because beyond that point, the benefit estimate is zero, and no buffer would be allocated to prefetch a missing block even if one were found. TIP removes accesses from the head of the list as it checks them to avoid rechecking them in the future.[3] The difference between the index of the hinted access to this first missing block and the application's current consumption index, *conIndex*, determines the parameter x in the benefit calculation.

---

[2] UFS also calls *TipPageAlloc* to obtain buffers for sequential readahead. These are allocated buffers with the same priority as demand misses. It would be useful to develop an estimator for the benefit of heuristic prefetching, but this dissertation focuses on prefetching according to hints and so leaves this as an area for future research.

[3] Should a cached or prefetched block within the prefetch horizon be ejected, the hinted accesses that reference it could be reinserted at the head of the list.

The *TipPrefetch* routine uses the greatest normalized benefit among the estimators to bid for a buffer. If it obtains one, it calls the file-system-specific routine to initiate a prefetch of the first missing block in the estimator's hinted access sequence. Before queuing the prefetch at the disk, the file-system-specific routine calls *TipKluster* to try and build a cluster prefetch as described in the next section. The estimator computes a new prefetching benefit for the next missing block. The benefit is normalized and the ranking of the estimators adjusted. *TipPrefetch* continues bidding for buffers until it fails to obtain one or no estimator has a positive prefetching benefit.

When a process performs a hinted read, all accesses in its hinted sequence move closer. This could change the benefit of prefetching for the processor. Thus, after a hinted read, the prefetching benefit estimate for the process is updated and TIP calls *TipPrefetch* to see if the system should prefetch some more blocks. The prefetching benefit does not change for the other processes, so their benefit does not need to be recomputed. However, after an application consumes some hinted blocks, new buffers may become available for prefetching. Or, the completion of writes of dirty buffers may make buffers available. In either case, the prefetcher gets another chance to bid for a buffer. In general, whenever an event occurs that may change either the benefit of prefetching or the cost of ejecting the globally least valuable buffer, the affected estimates are updated and a call to *TipPrefetch* is made to see if the prefetcher's bid for a buffer might now be successful.

5.2.2 Informed clustering

Clustering assembles separate, contiguous disk accesses into a single larger access that increases disk workload sequentiality and that decreases the number of disk access and therefore the CPU overhead of performing disk accesses. Informed clustering exploits hints to build clusters.

TIP builds clusters opportunistically on disk accesses[4] that are about to be queued at the drive. (Once a request has been queued at the disk, it cannot be changed, so clusters built around a request need to be built before the request is queued.) One could imagine scanning the entire hint sequence to determine the optimal clustering of requests. But, in

[4] TIP only builds informed clusters on other prefetches. In my benchmark suite, hinted accesses tend to be disjoint from unhinted accesses, so informed clustering opportunities on unhinted accesses are extremely limited or nil. DU with or without TIP builds clusters for heuristic sequential readahead.

keeping with the cost-benefit approach of one-at-a-time buffer allocation that tries to approximate gradient descent on I/O service time, TIP does not do this. Instead, it bids for buffers to add to a cluster one-at-a-time.

At its core, the procedure for building a cluster is as follows. Starting from a block about to be read from disk, work first to extend the request to include blocks after the requested one. If the next contiguous block is not already present in the cache, check if there are any hints for it. If so, compute the cluster prefetch benefit for the block and bid for a buffer. If one is obtained, add the block and buffer to the cluster and continue trying to extend the cluster by one more block. Otherwise, give up, and repeat the procedure going in the reverse direction from the original block. When the cluster can't be extended any further in that direction either, queue the cluster request at the disk.

The key operations in this algorithm are determining if a disk block is cached, determining if there are any hints for an uncached disk block, and determining where in the hinted access sequence these hints fall so that a benefit can be computed for adding the block to the cluster.

Unfortunately, because the file cache is organized by logical file block and not by disk block, the only way to determine if a particular disk block is already in the cache is to iterate over every block in the cache, mapping each to its disk block and checking if it is the desired block. Searching every cached block is bad enough, but hints are also given in terms of file byte range, not disk blocks, so either hints would all have to be mapped to a disk block and then indexed by disk block, or else hints, like cache buffers, would have to be searched exhaustively to see if any referred to the contiguous disk block.[5]

To get around this problem, TIP takes advantage of the fact that UFS tries to store logically contiguous file blocks in physically contiguous disk locations. This policy makes it likely that the blocks that are physically contiguous to a particular access are also logically

[5] I could have reorganized the whole cache around disk blocks, but a file-block organization has its own advantages. For example, the file-block organization makes it easy to use the same structure to cache blocks from file systems such as NFS which don't store their data on the local disk. It also means that hinted blocks can be found in the cache without first mapping the hints to disk blocks. Furthermore, changing the cache to a disk-block organization would have necessitated changes in all of the file systems as well as the virtual memory system which all assume a file-block organization. In implementing TIP, I wanted to keep changes as localized as possible. I could have added a disk-block structure on top of the file-block structure, but in addition to necessitating the mapping of all hints to disk blocks, it would have added CPU and memory overheads to maintain the duplicate structures. I did not seriously consider this option.

contiguous to it. In the UNIX file system, the disk addresses of neighboring logical blocks are stored in neighboring elements of an array [Leffler89], so the number of physically contiguous blocks before and after an initial block can be quickly determined when the initial block is mapped to disk. TIP limits its attempts to build clusters to those blocks which are both logically and physically contiguous.[6]

It is easy to check if these logically contiguous blocks are already present in the cache, but this does not address the issue of finding hints for the ones that are uncached. In providing this functionality, TIP optimizes for speed at the cost of memory. TIP allocates a ghost buffer, like those used in the LRU queue and described in Section 4.3.1, for hinted but uncached blocks. To this ghost buffer, TIP appends a list of all appearances of the block in any hinted access sequence. When TIP does the cache lookup to see if a contiguous block is cached, it finds the ghost buffer, which tells it the block is not already cached, and it has a list of all hints for the block. TIP then loops through this list of hinted accesses and calls the *estBid* function of each access's estimator to obtain a bid for inclusion of the block in the cluster. TIP then calls *TipPageAlloc* with the maximum of these bids and, if it gets a buffer, adds the block to the cluster. Shortly, there will be more on estimator functions and the nexus data structure that links hinted accesses to buffers.

5.2.3 Allocating the min-max buffer

The prefetcher, cluster prefetcher, and demand accesses call *TipPageAlloc*[7] with a bid to obtain a buffer. This routine finds the min-max buffer[8], determines if the cost of ejecting it is less than the bid, and if it is, reallocates the buffer. The algorithm for finding the min-max buffer was first described in Section 4.3.5. For convenience, Figure 5.5 is a reprise of the pseudocode for the algorithm. The crux of the algorithm is the lazy evaluation of the global value of a buffer. As described in Chapter 4, it is this lazy evaluation that allows the independent estimators to estimate the cost of only their least valuable buffer.

---

[6] TIP assumes that blocks at sequential disk addresses are stored contiguously on disk, even though this is not always true for SCSI disks. Actually, because the data is usually stored on a disk array, there is known to be a break in sequentiality at stripe-unit boundaries. TIP tries to build up clusters to full stripe units which are 64 KBytes or 8 blocks in size and does not build clusters that span multiple stripe units.

[7] The fact that in Digital UNIX all buffers are one page in size leads to this choice of name.

[8] This is the globally least-valuable buffer.

```
start:
   /* all estimators have already computed the cost of their */
   /* minimally valued tracked block and these costs have */
   /* been normalized for global comparison */
   estimator with lowest global cost {
      names its least valuable block;
      ceases tracking the named block;
      computes value of new least valuable block it is tracking;
      submits new value for normalization for global comparison;
   }
   /* check that no estimator values named block more highly */
   foreach estimator {
      query to see if estimator values named block more highly;
      if yes {
         /* block is saved from replacement */
         valuing estimator {
            begins tracking saved block;
            if saved block is now its least valuable {
               computes value of new least valuable block;
               submits new value for normalization for
                     global comparison;
            }
         }
         goto start;
      }
   }
```

Figure 5.5. Algorithm for identifying the min-max buffer (reprise). This algorithm lazily evaluates the global value of a buffer. Estimators nominate their least valuable buffer as the globally least valuable. The least valuable of the nominations becomes the candidate. To verify that the candidate is indeed the globally least valuable, each estimator is given the opportunity to save the buffer from replacement if it values the buffer highly. If one saves the buffer, the process repeats. The mechanism of tracking and only nominating blocks that the estimator is tracking guarantees that cycles do not occur and that the algorithm makes

In TIP, the estimators cooperate with a global 'least-valuable buffer' (LVB) module to implement the lazy-evaluation algorithm. *TipPageAlloc* takes care of the mechanics of reallocating a buffer from one block to another, but relies on *TipLvbPick* to identify the min-max buffer. Figure 5.6 shows the procedural flow from *TipPageAlloc* to *TipLvbPick* and the procedural interactions between the estimators and the LVB module.

The LVB module maintains the list of estimators that could supply a buffer that is sorted by normalized ejection cost shown in Figure 5.4. *TipLvbPick* takes advantage of the list to quickly find the estimator whose least valuable block is least valuable among all the estimators. Whenever an estimator computes a new value for its least-valuable buffer, it calls *TipLvbUpdate* to have the new value normalized for global comparison and the estimator's position in the list adjusted if necessary.

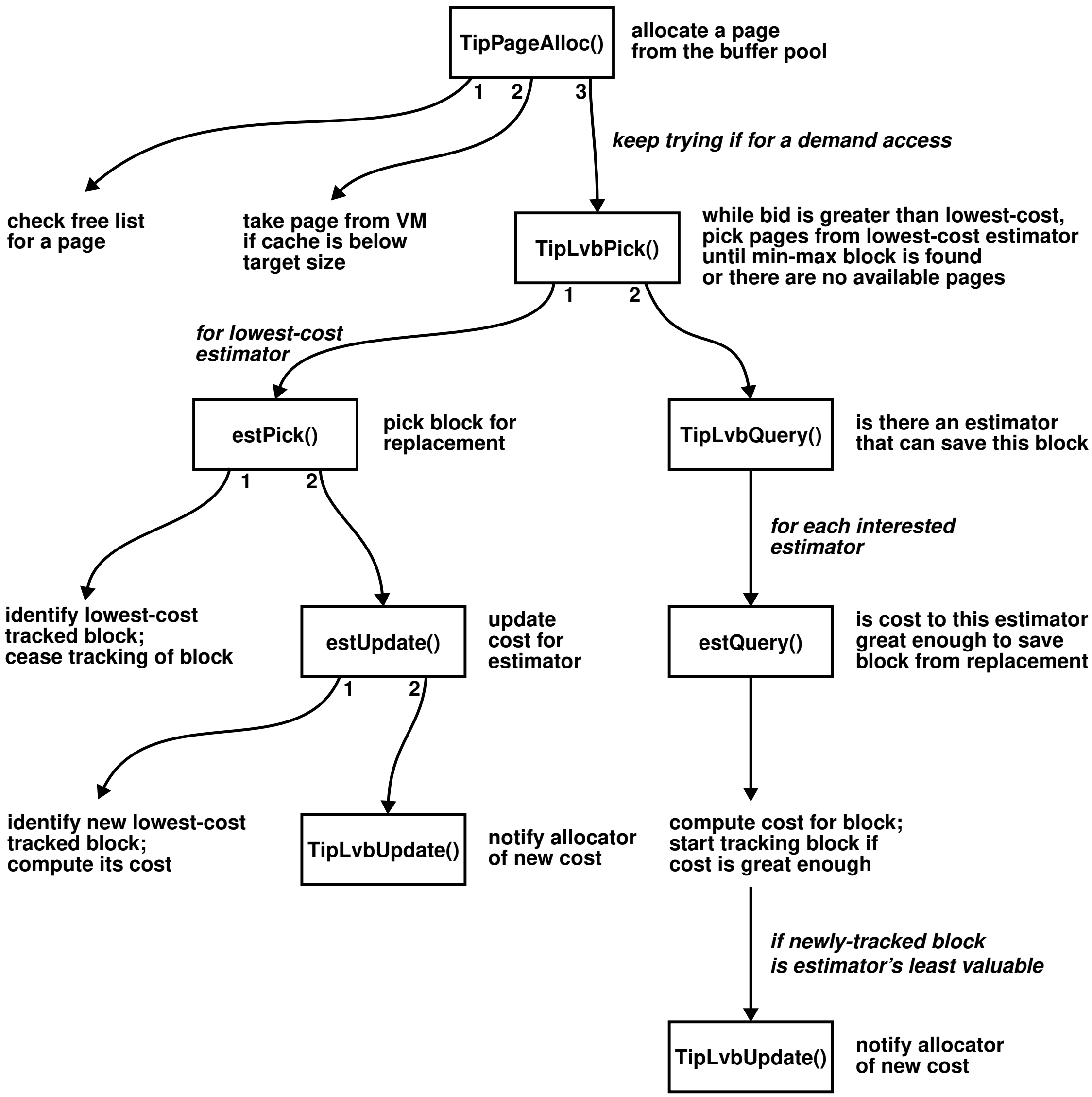


Figure 5.6. Procedural flow for page allocation. After checking for free pages and whether the cache should grow, *TipPageAlloc* calls *TipLvbPick* to ask the lowest-cost estimator to pick blocks for ejection and query other estimators to verify that the picked block is indeed the globally least-valuable, min-max block. The estimators call *TipLvbUpdate* to declare the cost of giving up of their new least-valuable buffer.

The LVB module requires estimators to provide two functions. First, *TipLvbPick* calls the *estPick* function of the least valuable estimator when it wants the estimator to give up its least-valuable block. Then, to see if any estimator wants to save the block from replacement, *TipLvbQuery* calls the *estQuery* function of every estimator that values the block to see if the estimator would like to save the block from replacement. Commonly, estimators also support an *estUpdate* function for internal use that updates the estimator's local estimate of ejecting its least-valuable buffer and therefore calls *TipLvbUpdate* to update its global value.

In implementing its *estPick* function, an estimator must be able to identify its least valuable tracked block. It must also be able to stop tracking the block when it gives up the block. To update their value estimate for their new least valuable buffer after a pick, estimators must also be able to identify the next to least valuable buffer, and ultimately every tracked buffer in turn. The LRU and hint estimators implement these operations differently, as will be described in Sections 5.2.6 and 5.2.7.

At a global level, implementing *TipLvbQuery* requires a method of finding every estimator that values a particular block. And, when *TipLvbQuery* calls an *estQuery* function, the called estimator needs to be able to estimate the value of the block it is being queried about. TIP accomplishes this by asking every estimator to attach a marker to every block it values. In the marker, estimators can store some data that helps them estimate the value of the block. Then, by calling the appropriate estQuery function for every marker attached to a block, TipLvbQuery can be certain that it has queried every estimator that values the block.

Hint estimators value blocks they have hints for. As was the case with clustering, TIP is faced with a need for finding all hints for a block. The list of hinted accesses attached to every buffer to generate cluster bids also serves as the list of markers for hint estimators. When *TipLvbQuery* calls the *estQuery* function for an access on the list, the hint estimator has only to look at the index of the access to compute its estimate of the value of the block for that access. Note that because a single hinted access sequence may reference a single block multiple times, and that because hint estimators add a marker to the list for every access to the block in the hinted sequence, *TipLvbQuery* may query the same estimator multiple times about the same block, once for each access to the block. As mentioned above, estimators track accesses, not blocks or buffers. If an estimator does not save a block based on an access in the distant future, *TipLvbQuery* will give it another chance to save it based on another access in the same sequence that is imminent.

The LRU estimator values buffers in its LRU queue. Conceptually, the LRU estimator adds a marker to each buffer in its queue and stores the segment number for the buffer so

name	function
estBid	generate bid to cache block; used to build clusters
estPick	pick a block for ejection; stop tracking block
estQuery	save & track block if valuable
estInval	stop tracking block; used when files are deleted
estCreate	create a new estimator
estDestroy	deallocate estimator
(estUpdate)	update cost estimate (optional; for internal use only)

Figure 5.7. Estimator operations. These are the six functions common to all estimators plus one that is commonly implemented for internal estimator use. See the text for details.

it can easily compute its value. The actual implementation differs from this slightly as will be discussed in Section 5.2.6.

5.2.4 Estimator functions

Scattered through the previous sections, are references to various functions that estimators must implement to support the prefetching and allocation algorithms. Figure 5.7 tabulates these and a few additional functions. These are the functions that a new estimator would have to support to be integrated into the TIP buffer allocation system.

TipLvbBid calls the *estBid* function to obtain bids for caching a block. The cluster prefetcher calls *TipLvbBid* to determine the benefit of adding a block to a cluster. *TipLvbPick* calls *estPick* when is wants the least-valuable estimator to name and stop tracking its least-valuable block. This pick starts the process of identifying the min-max block. *TipLvbQuery* calls *estQuery* to see if any estimator values an already cached block highly enough to save it from ejection and start tracking it. *TipLvbPick* uses the query operation for the lazy evaluation of buffer value. The cluster prefetcher also uses it to cause an estimator to start tracking a clustered block if the bid for a buffer for clustered prefetch turns out to be successful.

In addition to these functions which were mentioned earlier, estimators must support a few other functions. In particular, *estInval* is needed in case a file is deleted or otherwise becomes unavailable which causes blocks for the file to be removed from the cache. The *estCreate* and *estDestroy* functions are needed to create and destroy particular instances of an estimator. The LRU estimator is created at boot time and never destroyed, but hint estimators are created and destroyed for processes as needed.

5.2.5 The nexus data structure

From the foregoing algorithm descriptions, you should be getting the sense that the routine work of the TIP system is bookkeeping. The cache manager consists of many buffers, estimators, hints, hinted accesses, and the LRU list of unhinted accesses. The main challenge in generating estimates for cost-benefit buffer management is pulling together the estimator, block, and reference by the estimator to the block needed to apply the estimation equations.

In implementing TIP, I optimized for speed and created lists to accelerate the many lookup operations. Because it is the reference linking an estimator to a block that is most often needed to compute value estimates, these lists are most often lists of references, not blocks or estimators. The prefetcher needs to find the next missing block in a hinted sequence, so it has a list of the hinted accesses for each hint sequence. Clustering prefetches requires benefit estimates from all estimators that would like a particular block prefetched. Similarly, querying requires cost estimates from all estimators that would like to keep a block cached. So, attached to each buffer is a list of markers or references by estimators to the buffered block. Lastly, to ease picking the least valuable tracked block and finding the next least valuable tracked block, each estimator maintains a list of the references that are the basis of its value estimates for the blocks it is tracking.

In summary, estimators maintain multiple lists of references to blocks, and buffers maintain a list of estimator references to them. The TIP data structure that embodies an estimator reference to a buffer and links the two together is called a *nexus*. There is one nexus for every hinted access. And, there would also be one nexus for every block in the LRU list, but for historical reasons, in TIP, nexuses for the LRU list are embedded in the buffer header.[9] Multiple nexuses may link one estimator to a single block because an estimator may have multiple hints for the same block. Conversely, from every block, there is a separate nexus that links it to each estimator reference to it. Figure 5.8 shows how the nexus data structure links estimators and data blocks.

---

[9] There is also a memory advantage to having the LRU nexus embedded in the header. There, it occupies 20 bytes whereas a nexus requires 64. Most of the savings come from not needing links for the prefetching or hint lists of nexuses, nor for the buffer list of nexuses, nor from the nexus back to the buffer or the hint estimator.

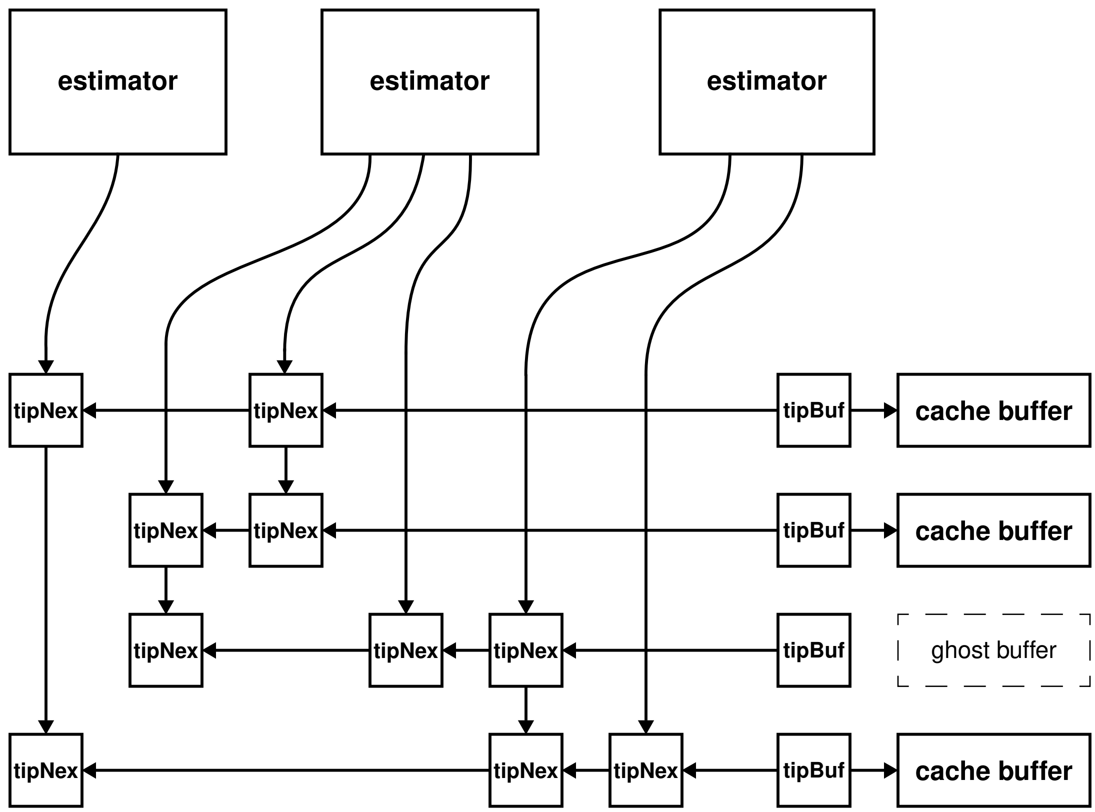


Figure 5.8. TIP data structure overview. The key data structures in the TIP system are the cache buffers, the estimators, and the *tipNex* nexus structures that link them together. Not shown in the figure are the links from every *tipNex* back to the corresponding estimator and *tipBuf*. There is one nexus for every hinted access, and one nexus list for every hinted byte range. Thus, a hint estimator with multiple hints for a block may have multiple nexuses for the block. Every block that has a nexus has a corresponding *tipBuf* buffer header whether there is physical buffer caching that block or not. If there is no physical buffer, it is referred to as a ghost buffer. The *tipBufs* are organized into a hash table so that for any block, it is easy to find both the block, and any estimators that value the block. The nexuses are strung together in different ways to form prefetching and tracking lists for hint estimators. The LRU list could also be composed of nexuses, but for historical reasons, a virtual ‘nexus’ for the LRU estimator is embedded in the *tipBuf* structure. A flag indicates whether this ‘nexus’ is part of the LRU list or not.

There are many references to blocks that are not currently cached. The whole idea of prefetching according to hints presumes that the hints may often refer to uncached blocks. Furthermore, the LRU estimator needs to detect that there would have been hits to blocks had the LRU queue been longer. TIP maintains headers for these ghost buffers. The headers are included in the file-block hash list so the LRU estimator can detect accesses to them and so that the cluster prefetcher can find hints for specific blocks quickly.

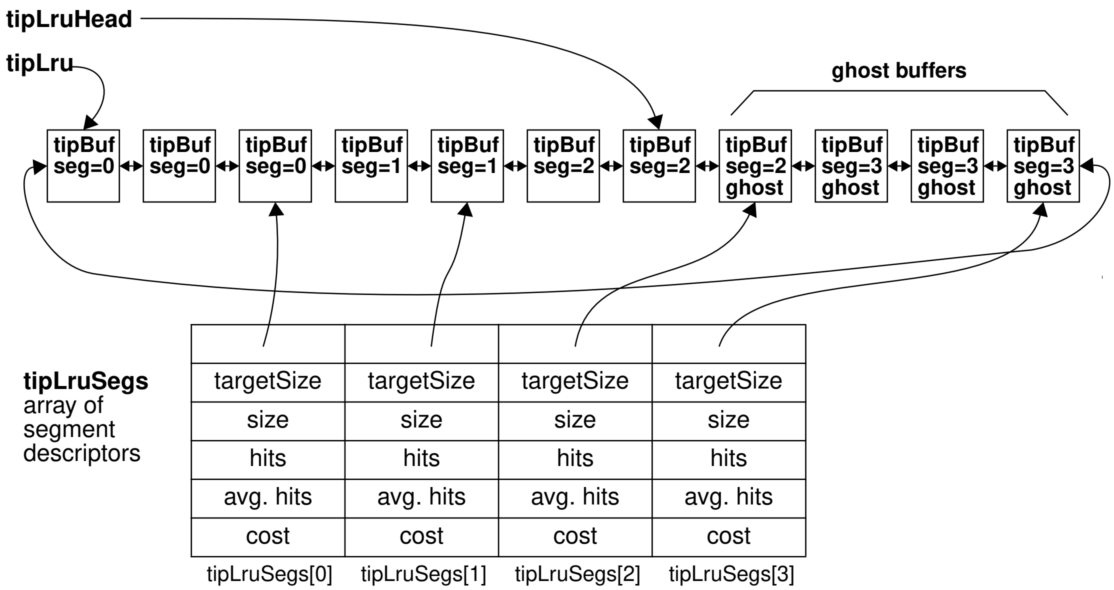


Figure 5.9. Data structures for the LRU estimator. The LRU queue is made up of a doubly-linked ring of *tipBuf* structures. Each *tipBuf* records which segment of the queue it is in as well as a status flag that indicates whether or not it is a ghost buffer. An array of *tipLruSegs* structures keeps track of per-segment data. When there is a cache hit, the segment number in the *tipBuf* is used to increment the hit count for the segment. When a *tipBuf* is released to the tail of the queue, buffers overflow from one segment to the next by shifting the segment pointers left so that no segment has more than *targetSize* buffers in it. The estimator periodically applies the LRU cost equation to each segment. The LRU estimator's least valuable buffer is the least-recently used non-ghost buffer in the queue which is pointed to by *tipLruHead*. The cost of losing it is the cost for the segment containing it.

5.2.6 The LRU estimator

Because, as mentioned above, the nexuses for the LRU estimator are embedded in the *tipBuf* buffer header structure, the LRU queue is a doubly-linked list of *tipBuf* structures as shown in Figure 5.9. Recall from Section 4.3.1 that to arrive at an approximation of *H(n)*, the hit ratio for the LRU queue as a function of cache size, the LRU queue is broken into segments and the number of hits in each segment is recorded. As shown in the figure, an array of segment descriptors keeps track of segment boundaries, counts the hits to the different segments, and records the cost of shrinking the LRU queue when the head of the queue is in that segment. An index in each *tipBuf* records which segment it belongs to. This segment index is incremented when the buffer overflows from one segment to the next.

To compute the cost estimate for each segment, the LRU estimator applies the equation shown in Figure 5.2, using a moving average of the hit counts for the segment for h_i

and the segment target size for $|s_i|$. It computes the costs for the segments from right to left so it can keep track of the maximum cost for higher numbered segments.

For the purposes of the lazy-evaluation algorithm, the LRU estimator's tracked buffers are those on the non-ghost portion of the LRU queue. Thus, the LRU estimator's least-valuable buffer is the one at the head of the non-ghost portion of the list which is pointed to by *tipLruHead*. The cost of losing the least-valuable buffer is the cost for the segment containing it. When *TipLvbPick* calls the LRU estimator's *estPick* function, it returns the value of the *tipLruHead* pointer, but leaves the buffer in the list. Instead of removing it, the estimator sets the buffer's ghost flag and moves the *tipLruHead* pointer to the buffer's left neighbor in the queue.[10] In this way the estimator ceases tracking of the block just picked and identifies its new least-valuable buffer.

When queried about a block, the LRU estimator only saves blocks it is already tracking; that is, it only saves non-ghost buffers. One could imagine saving any block on the queue that was in a segment whose cost of ejection was high enough to save the block. But, I chose not to do this because, as explained below, doing so would only very rarely improve application performance, and because doing so would often add extra overhead to the system.

Saving ghost buffers is unlikely to improve performance because there are few occasions in which saving such a buffer would lead to a cache hit. For the LRU queue to even consider saving a ghost buffer, it must first have picked the block for replacement to make it a ghost, something else must have saved the block from replacement, that something else must no longer want it cached, the block must still be on the LRU queue, and the cost of ejecting an LRU block must have grown relative to the cost of ejecting other blocks or the LRU estimator would not be able to save a block it previously picked for replacement. The something else must be either a hint estimator, or the fact that the block was dirty and could not be flushed. The dirty-block case, which is fairly common, is considered below. If a hint estimator saved it, but is now is willing to replace it, it is most likely that the hinted access occurred. It is unlikely that both the LRU queue's ghost buffer did not get pushed off the end of the LRU queue while the hint estimator tracked the block and that

[10] This ghost flag is used only by the LRU estimator to record that it is no longer tracking the buffer. True ghosts, *tipBuf*s with no associated buffer, have a nil buffer pointer.

the relative LRU cost of ejection grew in that same time period. But, even supposing that this unlikely chain of events occurred and the LRU estimator does save the block, doing so only helps performance when there is an unhinted access that finds it there in the cache. But, at least for the benchmark applications under study here, accesses to particular blocks tend to be either all hinted or all unhinted. Just the fact that five of the six applications hint over 85% of the block accesses shows that there is little overlap between hinted and unhinted accesses.

Being unlikely to improve performance would not by itself be a persuasive argument against saving ghost buffers, but saving them can also hurt performance. The most common opportunity to save blocks is when the LRU picked a dirty block for replacement and the write has completed. If the LRU saves the block, it will in all likelihood pick it for replacement almost immediately; the LRU already picked it once, so it cannot be too valuable. Having the LRU save and repick a block would not be bad except that saving ghost buffers would make it more expensive for the LRU estimator to find its least-valuable buffer because it would destroy the invariant that all buffers to the left of *tipLruHead* are real and all buffers to the right are ghosts. Specifically, if the saved block is in the same segment as the current head of the LRU list, the estimator may have to search the entire segment to see if the saved block is to the right or left of the current head and therefore is or is not the new head of the list. Further, if the LRU picks the head for replacement, it may have to search through many buffers to find the new head. Thus, saving ghosts adds CPU overhead and seems, on balance, likely to hurt, not help performance.

5.2.7 The hinted cache estimator

The value of a hinted block is a function of its position in the hinted access sequence. The estimator expands hints into a hinted access sequence as described earlier in Figure 5.3. As it expands hints, it builds a list of the sequence with *tipNex* data structures, marking each with the index of the access, and adding each to the list of nexuses for the block and to the list of accesses for the prefetcher. When the cluster prefetcher runs down the list of nexuses for the block, it asks the estimator that put each nexus on the list whether it wants to save the block for that nexus. Thus, if a block appears multiple times in a single hinted access sequence, then the estimator for that sequence may be called multiple times and asked if it wants to bid or save the block. But, each call will pass a pointer to the nexus that triggered the call. The estimator has only to look at the index stored in the nexus to compute its value and decide what it would bid for the cluster prefetcher to fetch the block or whether it can save the block from replacement.

If the estimator does save the block, or if the prefetcher prefetches the block, then the estimator must start tracking the block. The hint estimator maintains a linked list, not of the blocks it is tracking, but of the accesses that it is tracking. The list is sorted by the index number of the access. Thus, the estimator's least valuable block is the one that is last on its list of tracked nexuses. It is thus easy for the estimator both to compute the value of its least valuable block and to identify its next least-valuable block if it is asked to pick a block for replacement. If the picked block appears at another position in its access sequence, then the lazy evaluation will give it a chance to save the block based on these other hints.

5.3 Other implementation challenges

In the previous sections, I described the key aspects of the TIP implementation of cost-benefit buffer management. In the course of implementing the system, I ran into a number of problems. In this section, I describe some of those problems and my solutions to them. These problems include: the potential for the hint estimator data structures to consume an unbounded amount of memory when applications issue large numbers of hints; the lack of floating point arithmetic inside the kernel for the computation of value estimates; the existence of mapped pages in Digital UNIX's LRU queue which may be accessed without the knowledge of the LRU estimator; the existence of buffers in the cache which no estimator wants to track, but which must be pickable to not be lost to the cache forever; and the potential for priority inversion in the disk queue if demand accesses must wait for a large batch of prefetches to complete. I address each of these problems in turn.

5.3.1 Hint management and the caching horizon

When applications give hints, TIP immediately allocates kernel data structures which store them in essentially the same format in which they are issued. Before TIP's cost-benefit allocator can take advantage of the hints, it must resolve the names of the hinted file to obtain the vnode that describes the file if one is not readily available from the open file table, expand the hinted byte ranges into the hinted access sequence, allocate a *tipNex* structure for each access, and link the structure into the prefetching list and the per-block list of nexuses. If there is not already a *tipBuf* for each referenced block, then it must allocate one, thereby creating a ghost buffer, so that it can add the block to the cache's hash table. This whole process is called hint resolution.

When a hint is fully resolved, the nexuses, ghost buffers, and vnodes may consume a substantial amount of memory. Because a process may issue large numbers of hints at once, resolving all hints immediately could consume an arbitrarily large amount of memory.[11] Fortunately, it is not necessary to resolve an arbitrary number of hints to take full advantage of all of the hints that have been issued because even if resolved, the cost-benefit allocator would not devote any resources to hints for accesses that are very far in the future. The prefetcher only needs hints to be resolved out to the prefetch horizon. Beyond that, accesses which are candidates for informed caching or clustering need to be resolved. But, because the cost of ejecting a hinted block (which is the same as the benefit of cluster prefetching a block) decreases with the number of accesses until the block is accessed, there is some number of accesses beyond which any block, if found, would be the least valuable in the cache and therefore the next to be replaced. Specifically, if the cost of ejecting the block according to the index of the access in the sequence is less than the cost of ejecting the current least-valuable block, then the hinted cache estimator could not use that hinted access to save the block from replacement and the cluster prefetcher could not use it to add the block to a cluster. The cost-benefit allocator is not willing to devote any resources to that access or any access later in the sequence. The point in the sequence at which the cost falls below the cost of ejecting the currently least-valuable block is called the *caching horizon*. Exactly where the caching horizon lies depends dynamically on the cost of ejecting the currently least-valuable block in the cache. TIP only resolves hints out to the caching horizon.

[11] Even without resolution, the hints themselves could consume an unbounded amount of memory. The current TIP implementation puts a hard limit on the number of hints it stores and discards any additional hints. None of the benchmark applications exceed this limit. But, dropping hints on the floor discourages applications from giving as many hints as possible as early as possible. A better solution would be to store overflow hints on disk and bring them in as needed to be resolved. If an application issued so many hints that there was no available disk storage, TIP could return an error code that informed the application that further hints will be dropped on the floor until the application consumes some of the hints it has already issued.

In general, there is a moving window of the hinted access sequence that is resolved. As the application performs hinted accesses, TIP deallocates the nexus structures for those accesses. Meanwhile, the application moves closer to later accesses in the sequence and therefore pushes the caching horizon later in the sequence. TIP records which hint was last resolved so that it can quickly resume resolution.

5.3.2 Using integer arithmetic to compute cost and benefit estimates

The cost and benefit expressions in Figure 5.2 include division operations that could produce fractional, non-integer results. If implemented at user level, it would be natural to use floating point arithmetic to calculate the cost and benefit estimates. Unfortunately, use of the floating point registers is disallowed in the kernel so that these registers do not need to be saved and restored on every system call and interrupt. Thus, integer math must be used to compute cost and benefit estimates.

TIP applies a few simple techniques to adapt the estimates to integer arithmetic. Essentially, the approach is to use a loose form of fixed point arithmetic and take advantage of the fact that the estimates will only be compared to each other so only relative, not absolute values are important. First, TIP expresses the T_{hit}, T_{driver}, and T_{disk} parameters in an integer number of microseconds instead of fractions of a second. Second, to gain precision when dividing a time value by the potentially large number of accesses that a buffer may be tied up, the parameters are left-shifted by 10 which is equivalent to multiplying them by 1024. Finally, when normalizing local estimates for global comparison, instead of multiplying estimates by the number of access per second that an estimator represents, TIP eliminates the division by time and simply multiplies by the number of accesses in a time period. I now give example code fragments for the estimators to clarify these techniques.

When prefetching for a hint sequence, the benefit of prefetching a block x accesses in advance, but within the prefetch horizon, $0 < x \le \hat{P}$, is, from Figure 5.2,

$$\text{Benefit}_{pf} = \frac{T_{disk}}{x(x-1)} \quad . \tag{5.1}$$

TIP implements this as

```
benefit = (Tdisk<<10)/((x*x)- x).
```

The disks used for the testbed described in the next chapter have an average access time of 15 milliseconds, so expressed in microseconds the value of T_{disk} is 15000.

The cost of ejecting a block y accesses in advance of its use when y is beyond the prefetch horizon, $y > \hat{P}$, is

$$\text{Cost}_{eject} = \frac{T_{driver}}{y - \hat{P}} \quad . \tag{5.2}$$

This is also the benefit of adding the block to a cluster prefetch. TIP implements the computation as

```
cost = (Tdriver<<10)/(x-pfHorizon).
```

The disk driver overhead is on the order of hundreds of microseconds.

The cost of taking a block from the LRU estimator depends on the number of the segment which currently holds the head of the LRU queue.

$$\text{Cost}_{LRU} = max_{j \geq i}\left\{\frac{h_j}{A|s_j|}\right\}(T_{miss} - T_{hit}) \quad . \tag{5.3}$$

In this equation, h_j is the number of hits in a segment, A is the total number of unhinted accesses (which rely on the LRU queue), and $|s_j|$ is the target number of buffers for the segment. In the global normalization step, this estimate would normally be multiplied by the number of accesses, A. TIP takes advantage of the division by A followed by multiplication by A; it skips the division and multiplies LRU estimates by 1 to normalize them. It computes the cost in two steps. First, it computes the marginal cost for each segment and then runs through the segments to determine the maximum of the costs for higher numbered segments. The code for computing the marginal cost for one segment is

```
segCost = (segHits * ((Tmiss - Thit)<<10))/ targetSize.
```

The value for *segHits* is a moving average of the number of hits in 1024 accesses and therefore is a value between 0 and 1024. Computation of moving averages is described below. The time for a cache hit, T_{hit}, is on the order of hundreds of microseconds. The time for a cache miss, T_{miss}, is the sum of T_{hit}, T_{driver}, and T_{disk} and so the 15000 micro-second disk access time dominates its value.

As mentioned above, TIP normalizes value estimates by multiplying them by the number of accesses. Actually, it uses a weighted moving average of the number of accesses to each estimator out of a total of 1024 accesses to the system as a whole. TIP counts the number of accesses to each estimator, and after a total of 1024 accesses to all estimators, it recomputes the moving averages:

```
new_average = (old_average + (3 * current_count)) / 4.
```

When TIP updates the averages and therefore the normalization factors, it also updates the cost estimates for the LRU segments and uses this same weighting for the number of hits to a segment in a given period.

5.3.3 Managing mapped pages with the LRU annex

In Digital UNIX, data blocks from mapped files are kept in the file buffer cache and are not moved to the virtual-memory system. These mapped files pose a special problem for the LRU estimator because accesses to mapped blocks are unobservable memory references. Thus, the LRU estimator cannot know when mapped blocks are referenced and therefore cannot include accesses to them in its estimation of *H(n)*. How can the LRU estimator obtain an accurate estimate of *H(n)* that includes mapped blocks? Because mapped files are in regular use in Digital UNIX for such things as shared libraries, this is not a strictly academic question. Some answer is needed to build a working TIP system.

One approach is to bound the time since the last access by unmapping pages, thereby forcing page faults which go through the normal read code-path. Faults on cached pages are then equivalent to cache hits and unmapped pages which have not caused a fault are known not to have been referenced since the page was unmapped. This was the strategy in the initial implementation. The first few segments of the LRU queue were designated the active region of the queue and contained both mapped and unmapped blocks. Any block found to be mapped as it overflowed from the last active segment to the first inactive segment was unmapped and reinserted at the tail of the LRU list. The inactive region of the queue thus contained only unmapped blocks. Note that within the active region, the estimation of *H(n)* is poor and that, furthermore, the active-page remappings in the first inactive segment drive up the number of hits in that segment. For these reasons, the LRU estimator has less reliable information to use to shrink the LRU queue any smaller than the active region plus the first inactive segment. Thus, it is desirable to have a small active region and achieve a tight bound on the time since the last reference to a mapped page. Unfortunately, a small active region in the LRU queue results in the continual mapping and unmapping of active pages which can add substantial CPU overhead and slow the system down.

For low overhead, but an accurate estimate of $H(n)$, the system needs to identify active mapped pages and avoid unmapping them too frequently while keeping the size of the active region small. TIP achieves this with a separate queue of blocks called the LRU annex. When mapped blocks overflow from the active region, TIP moves them to the tail of the annex list instead of the general LRU list. Reads that hit a block in the annex leave the block in the annex. Periodically, the system examines a fraction of the blocks in the annex, and if they are no longer mapped, it moves them back to the regular LRU queue. Otherwise, it unmaps them and moves them back to the tail of the annex. The key benefit of the annex is that large numbers of regular file accesses do not cause mapped blocks to be unmapped at a high rate. The regular blocks move down the LRU list without disturbing the mapped blocks in the annex.

In the default configuration, the file cache has a total of 1536 8 KByte buffers, there are 14 segments, the active region of the LRU queue consists of just the first segment whose target size is 236, and the other 13 segments have a target size of 100. There is one additional segment that holds overflow *tipBufs*. The annex is limited in size to a maximum of 500 blocks although it is typically about half that size.

The LRU annex is essentially a crude approach to managing virtual-memory pages. VM pages are also mapped into a process' address space making accesses to them unobservable. The annex is crude in that it fixes the size of the active region, puts a hard limit on the size of the annex, and does not use any cost or benefit functions to determine how many buffers to leave in the annex. A better approach might be to split the annex into active and inactive regions and use cache hits that remap blocks in the inactive portion of the annex as an indication of the amount of activity for the mapped pages. If there were a lot of hits, there would be benefit in saved overhead to growing the active region. If there were not many hits there would be little additional overhead if the active region were shrunk. Furthermore, hits in the inactive region could be profiled just like the regular LRU queue to estimate the cost of shrinking the size of the annex's inactive region. In this way, the cost-benefit buffer allocator could be used to size the annex dynamically. This approach could also be applied to the active and inactive regions of the queue of VM pages. In this way, VM management could be integrated into TIP and cost-benefit analysis could be used to manage VM pages and cache buffers as a single memory resource.

5.3.4 The orphan estimator

Because *TipLvbPick* obtains candidate replacement blocks by asking estimators to nominate one of their tracked blocks for replacement, any cached block that is not tracked by some estimator will never be replaced. Thus, whenever a buffer is not busy, wired, or otherwise unavailable for replacement, at least one estimator must be tracking it. A special *orphan* estimator tracks blocks that neither the LRU nor any hint estimator wants to track. Examples of such blocks include those that are dirty when chosen for replacement and blocks tracked for a now aborted process. Whenever a previously busy (at the disk), held (by a process), or wired block is released back to the cache and that page is not tracked by any estimator, then the block is transferred to the orphan estimator which begins tracking the buffer. Similarly, at any other time, such as hint cancellation or process termination when orphan blocks are created, the orphans are transferred to the orphan estimator. Through the orphan mechanism, TIP ensures that all replaceable blocks are in fact pick-able and therefore not lost to the system.

Orphan blocks have no estimated value. Thus, whenever a buffer is needed, orphan blocks are the first to be replaced. The orphan estimator maintains its tracking list in FIFO order. Even when the orphan estimator picks a block for replacement, the system still calls *TipLvbQuery* to make sure there is still no estimator that values the block highly enough to save it from replacement.

5.3.5 Disk driver support for prefetching

The TIP system has two device driver enhancements to support prefetching. First, a striping driver makes possible I/O parallelism within a single file system. Second, a low-priority prefetch queue limits how much prefetch accesses can delay demand accesses.

The striping pseudo-device makes multiple physical disks appear to be a single disk to the rest of the system. The striper breaks the single linear block address space it exports into stripe units of eight 8-KByte blocks or 128 512-byte sectors. It then assigns these stripe units in a round-robin fashion to the disks that make up the array. When the striper receives an I/O request, it maps the request to the appropriate disk and blocks within the disk and then forwards the request to the disk's driver. If a request spans multiple disks, it is broken into multiple requests which are sent to the different disks. When the individual disk requests complete, the results are reassembled and returned as the results of the single request to the array. However, both TIP and the UFS file system are aware of the stripe-unit size and do not issue requests that straddle multiple stripe units, so such spanning requests do not occur in normal operation.

Because TIP may queue many prefetches at a time, demand requests could experience substantial delays if they were forced to wait for the prefetches ahead of them to complete. To avoid this problem, the striper maintains a special prefetch queue for each disk in the array. When the disk is idle, the striper issues up to two prefetch requests at a time to the disk. When one completes, it issues another unless a demand request arrives in the meantime. By having two requests outstanding at the disk, the disk can begin processing the second prefetch while the system services the interrupt for the first request and queues a new request at the disk. On the other hand, demand requests never find more than two requests queued in front of them at the disk. The striper sorts the prefetch requests by the CSCAN[12] scheduling discipline.

In the future, it would be beneficial if disks could support a lower-priority queue for prefetch requests. Then the system could let the disk take advantage of its intimate knowledge of the data layout to schedule both demand and prefetch requests. It would also be possible to abort servicing a prefetch request if a demand request arrived. In this manner, demand requests would experience even lower delays behind prefetch requests and all requests would receive more efficient service from the disk.

---

[12] CSCAN stands for circular scan. CSCAN services requests in increasing order of disk block address. When there is no queued request for a block address greater than the block last accessed, it next services the request with the lowest block address. Thus, CSCAN scans the disk surface in increasing order and then seeks back to begin a new scan. Some researchers refer to this algorithm as CLOOK [Worthington94].

5.4 Conclusion

Taking advantage of application disclosure of future file accesses for prefetching and caching is a bookkeeping challenge. To hold a block in the cache, a hint for the block must be found before the block is ejected. And, to prefetch a block, a hint must be found before the block is accessed. The main challenges in implementing a system that takes advantage of hints are performing these bookkeeping tasks efficiently and making replacement decisions without slow searches through large data structures.

TIP's nexus data structure provides the key bookkeeping support for allocation by cost-benefit analysis. It ties both hints and the LRU queue to buffer headers for the blocks they reference. Ghost buffers, which have a buffer header but no data buffer, serve as cache placeholders for referenced but uncached blocks. Real and ghost buffers are organized into a single hash table for quick access. And, the nexuses attached to each buffer header provide quick access to value estimates for the block whether it is currently cached or not. Thus, it is easy to check the value of a block before ejecting it, and it is easy to check whether a hint refers to real buffer or refers to a ghost that must be prefetched.

The nexus makes it easy to find value estimates for a block, but identifying the globally least-valuable block would still be costly if the global value of all blocks had to be determined before making a replacement decision. TIP's algorithm for the lazy evaluation of global buffer value avoids this need. It relies on the independent value estimators to rank the blocks they are tracking by their own local estimate of value. Both the LRU and hint estimator can do this without computing any cost estimates. For the LRU estimator, position in the LRU queue determines the value rank. For hint estimators, position in the hinted access sequence determines value rank. The algorithm then only needs to compare globally the values of each estimator's least-valuable block to determine which estimator's *estPick* function it should call to find a likely candidate for replacement. By calling the appropriate estimator's *estQuery* function for each nexus linked to the candidate block, the allocation algorithm ensures that the candidate block is indeed the least valuable before replacing it. It is this combination of quick local value comparisons and few global comparisons that makes the TIP implementation of buffer allocation by cost-benefit analysis efficient. In the next chapter, I will quantify the computational overhead of the algorithm.

In addition to being efficient, the two-tier allocation algorithm also makes it easy to add new value estimators to the system. Any buffer supplier that supports the *estPick* and *estQuery* operations, and declares the value of its least valuable block in terms of the common currency by calling *TipLvbUpdate* could be integrated into the system.

Chapter 6

TIP Performance Evaluation

The overall goal of informed prefetching and caching is to reduce the elapsed time required to run applications. Elapsed time is therefore the key metric of TIP system performance. In this chapter, I present the results of experiments that show that informed prefetching and caching in TIP are remarkably effective, reducing elapsed time by up to 84%. Further, to better understand the contributions of the different components of the system, these experiments explicitly isolate the effects of informed prefetching and caching. Detailed measurements of many aspects of system behavior are presented to shed additional light on why TIP performs the way it does.

Because the benefit of prefetching and caching is workload-dependent, it is important to evaluate system performance under as broad a range of realistic workloads as possible. To that end, the benchmark suite used in this evaluation is the collection of six real-world applications described in Chapter 3. I first consider the performance of each benchmark running alone and then go on to consider performance when multiple applications are running simultaneously.

Nothing is free, and this includes TIP. Achieving the results it does comes at the expense of some additional CPU and memory overheads. Detailed traces quantify the CPU overheads. Simple calculations quantify the memory overhead.

6.1 Experimental testbed

The testbed used to run the experiments described in this chapter is a Digital 3000/600 workstation (SPECint92=114; SPECfp92=162), containing a 175 MHz Alpha (21064) processor, 128 MBytes of memory and two KZTSA fast SCSI-2 adapters each hosting up to five HP2247 1 GByte disks. This machine runs version 3.2c of the Digital UNIX (DU) operating system. For comparison purposes, I present results for both the standard DU kernel and a kernel that has been modified to include the TIP buffer cache manager whose implementation was described in Chapter 5.

The system's 10 drives are bound into a disk array by a striping pseudo-device with a stripe unit of 64 KBytes. This striper maps and forwards accesses to the appropriate per-disk device driver. Demand accesses are forwarded immediately, whereas prefetch reads are forwarded whenever there are fewer than two outstanding requests at the drive. The striper forwards two prefetch requests to reduce disk idle time between requests, but doesn't forward more to limit priority inversion of prefetch over demand requests. The striper sorts queued prefetch requests according to CSCAN. This striper is described in more detail in Section 5.3.5.

The standard DU kernel has two features that make it a particularly strong base case for comparison: request clustering and aggressive sequential readahead. When DU performs multi-block reads or writes to contiguous disk blocks, it coalesces or clusters up to 8 contiguous disk accesses into one large request. Because the file block size is 8 KBytes, this means that individual disk requests can range in size up to 64 KBytes. This request clustering has the same performance advantages as informed clustering: a significant reduction in the CPU overhead of performing disk accesses and an increase in efficient, sequential disk accesses. The algorithm that checks blocks for contiguity was modified to make it aware of the 64 KByte disk-array stripe unit and ensure that all clusters fit within a single stripe unit.

DU's second notable feature is its very aggressive sequential readahead heuristic. In addition to prefetching data, the aggressive readahead serves to fetch clusters of blocks from the disk even when applications request only a single block at a time. Essentially, the file system initiates prefetches in proportion to the length of the current run of sequential accesses up to a maximum of 8 clusters of 8 blocks each. Thus, if an application sequentially reads 4 blocks, then the system initiates readaheads for the next 4 blocks. These readaheads are queued as low-priority prefetch requests by the striper. This same readahead algorithm is applied to unhinted accesses in the TIP system. As we will see, this algorithm is very effective for large sequential accesses. However, we will also see that such aggressiveness can hurt performance when accesses are more random.

Another notable feature of DU is that it includes a unified buffer cache (UBC) module that dynamically trades memory between its file cache and virtual memory (VM). Unfortunately, as mentioned in Section 5.1, because TIP does not yet have an estimator for the value of VM pages, it cannot dynamically size the cache. For meaningful performance comparisons between the DU and TIP systems, I needed to eliminate cache size as a factor differentiating the two systems. Therefore, as mentioned in Section 5.1, I fixed the cache size of both systems at 12 MByte (1536 8 KByte buffers).

6.2 Measuring cost-benefit model parameters

TIP's cost-benefit estimates depend on the model parameters defined in Chapter 4: T_{hit}, the time to read a block from the cache, T_{driver}, the CPU overhead of performing a disk read, and T_{disk}, the average disk access time. To determine values for these parameters, I used a synthetic application to run a number of micro-benchmarks. The synthetic application repeatedly reads a sequence of random, unique blocks from a large file (512 MBytes). The application determines the block sequence and then gives a hint disclosing the sequence for the first iteration. Hints for the next iteration are given at the start of the current iteration. Between each block read, the application computes for an amount of time given by the T_{app} parameter.

To measure T_{hit}, I set the sequence length to 1000, T_{app} to 0, and the number of iterations through the sequence to 10. Because the sequence length is less than the cache size of 1536 buffers, the cache can easily hold the entire sequence. After preloading the sequence into the cache, I measure the time to required to perform the 10,000 block reads, all of which are cache hits. Because the stall time is zero, T_{hit} is the elapsed time for the run divided by the 10,000 accesses. Using this method, I determined that T_{hit} = 203 microseconds.

To measure T_{driver}, I repeated the above experiments, except with a sequence length of 2000. Because this sequence does not fit in the cache (and the hints are not used for caching), there is a prefetch for every read. Thus, there is a total of $T_{hit} + T_{driver}$ of CPU time per read. In the experiments, the elapsed time less the time spent stalled for I/O is the total CPU time. From this measurement and the value of T_{hit} found above, I determined that T_{driver} = 366 microseconds.

Disk service time, T_{disk}, depends heavily on the length of seek required, which makes assigning a single value to this parameter difficult. From direct measurements on a variety of applications including this synthetic benchmark, I found that average service times for random accesses were about 15 milliseconds, so this is the value assigned to T_{disk}.

Based on these parameter values, at boot time, TIP applies Equation (4.23) to compute a system-wide, static, upper-bound prefetch horizon, $\hat{P}$, of 73 (using integer arithmetic).

6.3 Single application performance

To explore the contributions of the various optimizations to overall performance, I ran each of the benchmark applications in four configurations. First, as a baseline for comparison, I ran each application when it doesn't give hints on the standard Digital UNIX operating system constrained to a fixed cache size. To evaluate how well TIP performs for non-hinters, I ran the unhinting applications on it (*TIP, no hints*). TIP applies DU's clustered readahead heuristic to unhinted accesses so the comparison to DU is fair. To determine the effect of informed prefetching alone, I ran the hinting applications on a TIP system restricted to using the hints for prefetching and clustering within the prefetch horizon but not for informed caching (*TIP, no caching*). For these runs, the prefetch depth is statically set to the prefetch horizon, $\hat{P}$ = 73. Finally, I ran the hinting applications on a fully-functional TIP system which exploits the hints for informed prefetching, clustering, and caching. I measured elapsed time and stall time for each of these configurations when running with disk arrays of 1, 2, 3, 4, and 10 disks. A bar chart reports the average elapsed and stall time of five runs. A separate table presents the numbers for CPU and stall time along with the 95% confidence interval for the average as computed using the sample variance and the student-t distribution. The CPU time presented is the difference between the measured elapsed and stall times.

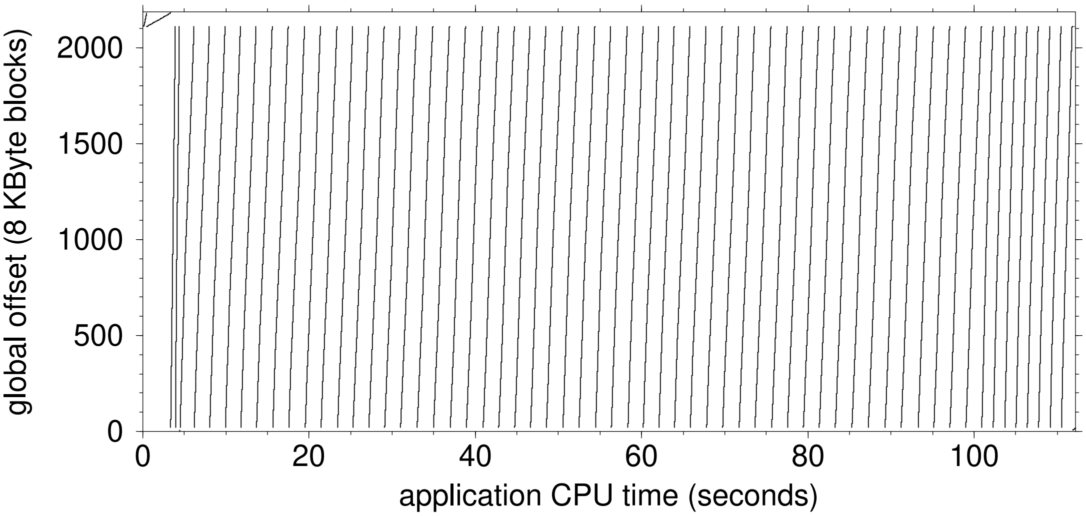
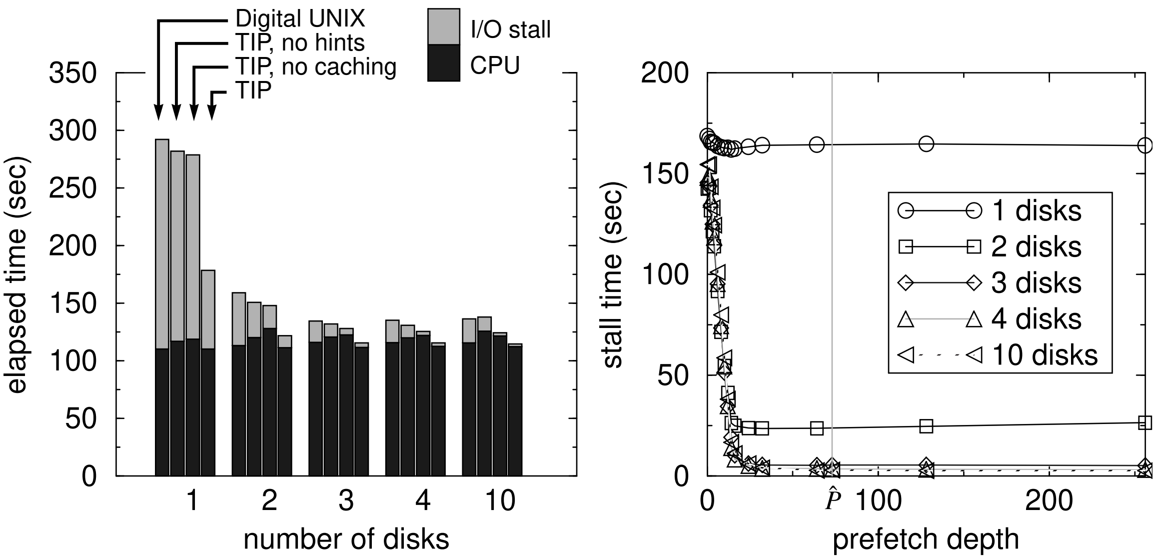


Figure 6.1. Davidson access pattern. For this figure, the several files Davidson accesses are logically concatenated to map their block numbers to a global block number space. Davidson's file accesses were traced and every access was mapped to the corresponding offset within the global space. This graph shows the global offsets of Davidson's accesses as a function of application CPU time. Davidson's repeated sequential accesses to the 2089-block matrix file are clearly visible.

To explore the effect of prefetch depth on application stall, a supporting graph shows stall for the *TIP, no caching* configuration when the prefetch depth varies from 0 to 256. Finally, to examine the effect of prefetching and caching on cache performance, I tabulate cache performance for all four configurations. The numbers are taken from the results on a single disk, but because prefetching and caching decisions in TIP depend primarily on the sequence of accesses and not on their timing, single-application cache performance is not sensitive to array size. Later, when I consider cache performance when multiple applications are running, I will report numbers for multiple array sizes because array size can affect the interleaving of accesses from the multiple applications and therefore caching decisions.

6.3.1 MCHF Davidson algorithm

The Davidson algorithm benchmark, described in Section 3.4.4, repeatedly reads a 16.3 MByte dataset sequentially in its entirety 60 times. Figure 6.1 is a graphical representation of its access pattern. Figure 6.2a presents the results for the four configurations and Table 6.1 gives the corresponding numerical data. Table 6.2 shows the cache performance for the systems. The high-level results are that, with or without hints, Davidson benefits significantly from the extra bandwidth of a second disk and becomes CPU-bound on



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.2. Davidson performance. This figure shows the performance of the Davidson computational-physics benchmark which repeatedly reads a large file sequentially on a range of disk array sizes. On the left, (a) shows elapsed time broken into CPU time and I/O stall time for four configurations. *Digital UNIX* is an unhinting Davidson running on the unmodified system. *TIP, no hints* is an unhinting Davidson running on TIP. *TIP, no caching* is Davidson giving hints which are used only for informed prefetching and not informed caching. Finally, *TIP* uses Davidson's hints for both prefetching and caching. Digital UNIX's aggressive readahead performs about the same as informed prefetching alone (*TIP, no caching*) for this access pattern. With informed caching, TIP reduces elapsed time by more than 30% on one disk by avoiding high-latency disk accesses. On more disks, prefetching masks disk latency, but informed caching still reduces elapsed time more than 10% by avoiding the overhead of going to disk. On the right, (b) shows I/O stall time as a function of prefetch depth for the TIP, no caching configuration. A prefetch depth of $\hat{P}$ is more than adequate to obtain the full benefit of informed prefetching for these *TIP, no caching* runs.

larger arrays. Because the hints only disclose sequential access in one large file, Digital UNIX's aggressive readahead is nearly as effective as informed prefetching alone as seen by comparing *TIP, no hints* and *no caching* performance. Informed caching, however, increases the cache hit ratio which reduces the number of disk accesses. When the I/O bandwidth of a single disk limits performance, fewer accesses translates into fewer stalls and faster elapsed time.

The first two bars in each quartet in the figure show that the performance of TIP without hints is similar to that of the unmodified Digital UNIX kernel over the full range of array sizes. Nevertheless, there are minor differences. In particular, the TIP system stalls a little less than the standard system but consumes a little more CPU time. The reduced stalls are the result of TIP's better paging performance due to the LRU annex which was described in Section 5.3.3. From Table 6.2, which gives the prefetching and caching per-

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
Digital UNIX	110.03 (0.66)	182.07 (0.65)	113.22 (0.41)	45.75 (0.42)	116.11 (0.35)	18.29 (0.35)	115.76 (0.16)	19.41 (0.17)	115.49 (0.23)	20.88 (0.32)
TIP, no hints	116.95 (1.23)	165.01 (1.19)	120.11 (0.41)	30.46 (0.39)	120.53 (1.84)	11.46 (0.73)	119.84 (0.73)	10.86 (0.40)	125.63 (11.3)	12.23 (1.09)
TIP, no caching	118.75 (7.03)	160.06 (6.98)	127.92 (6.89)	20.05 (3.76)	122.41 (1.23)	5.45 (0.29)	121.91 (1.49)	3.48 (0.11)	121.41 (0.58)	2.95 (0.17)
TIP	110.14 (1.34)	68.43 (1.61)	111.46 (1.40)	10.32 (1.87)	111.59 (0.43)	3.94 (0.13)	112.57 (0.87)	2.88 (0.13)	112.28 (1.29)	2.29 (0.09)

Table 6.1. Davidson elapsed time. This table presents the CPU and I/O stall times in seconds that are graphed in Figure 6.2. They are averages over five runs. The numbers in parentheses give the 95% confidence intervals.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
Digital UNIX	153149 (0)	7248 (0)	126592 (6)	16597 (1)	25780 (0)	126039 (6)	1329 (6)	630 (3)
TIP, no hints	147041 (0)	1140 (0)	124879 (6)	15980 (1)	21381 (0)	124847 (6)	812 (6)	195 (3)
TIP, no caching	147042 (0)	1140 (0)	125676 (0)	15874 (0)	21373 (0)	125569 (0)	100 (0)	77 (0)
TIP	147042 (1)	1140 (0)	50845 (1072)	8937 (328)	96159 (1082)	50782 (1082)	100 (0)	77 (0)

Table 6.2. Davidson prefetching and caching performance. This table shows the number of requests for file blocks made of the buffer cache, the prefetching aimed at maximizing cache hits, and how well the combination of caching and prefetching did at servicing the requests. The numbers in parentheses are the 95% confidence interval for the measurements. Requests for empty buffers, e.g. for writes, are not included. *Faults* are blocks requested in response to a fault on a mapped block and are included in the *total* column. Most faults are for shared library text pages. *Prefetches* includes heuristic readahead and informed prefetching where applicable. Because the system clusters requests for multiple blocks into a single disk I/O access, the number of prefetch and miss I/Os is smaller than the number of blocks requested. Cache performance is broken into three categories. *Reuse hits* are scored when a block services a second or subsequent request. *Prefetch hits* are requests that hit in the cache only because the requested block was prefetched. The difference between the number of blocks prefetched and the number of prefetch hits is the number of blocks that were prefetched, but never accessed. Finally, the number of *misses* is the number of blocks requested that weren't in the cache. Disk reads for these blocks are clustered into *miss I/Os* accesses. For Davidson's sequential accesses, prefetching is effective even without hints, but hint-based informed caching increases the number of reuse hits by a factor of 4.5. See the text for details.

formance for Davidson, the annex eliminates about 6100 page faults and reduces by about 2200 the number of blocks read from disk. Without the annex, the sequential read of the large matrix file runs the mapped pages through the LRU queue which causes many to be unmapped and some to be flushed from the cache.

The higher CPU time for TIP with no hints compared to DU is primarily a result of the overhead of estimating on an ongoing basis the cache hit rate profile of the LRU queue. Giving hints to TIP adds some overhead for managing the hints. But, when TIP can use the hints for informed caching to reduce the number of disk accesses, the reduction in the CPU overhead of performing disk accesses more than offsets the CPU overhead of the TIP system. I examine the CPU overhead of TIP in more detail in Section 6.6.2.

In the absence of hints, the TIP system uses DU's aggressive readahead policy for heuristic prefetching. By comparing the second and third bars in the figure, we can see that such readahead is nearly as effective as informed prefetching for this sequential workload.

For a closer look at I/O stall time as a function of prefetch depth on the TIP system with different array sizes, consider Figure 6.2b which presents stall time as a function of prefetch depth when TIP is configured to prefetch a fixed depth or number of accesses in advance. In this configuration, when TIP initiates a prefetch, it tries to cluster other hinted blocks up to a depth of 16 accesses beyond the prefetch horizon. It clusters beyond the prefetch horizon so that at a prefetch depth of 1, for example, the system still clusters the sequential accesses that DU would. The *TIP, no caching* bars in Figure 6.2a correspond to a prefetch depth of $\hat{P}=73$ in part (b) of the figure. At a prefetch depth of 0 there is neither heuristic nor informed prefetching, although when a block is missed and read in, its neighbors on the disk are read in as a cluster.

On a single disk, sequential readahead, no prefetching, and informed prefetching all perform equally well for Davidson's sequential accesses because there is sequential readahead within the disk drive; the drive continues reading sequentially on the surface of the disk after servicing a read request. The drive knows nothing of file structure, but as long as the file is laid out sequentially and read sequentially as it is in this case, the drive's readahead heuristic successfully anticipates the next request. On the single disk, the bandwidth off the media is the ultimate performance bottleneck. For sequentially laid-out and read data, the drive's readahead is sufficient to fully utilize the bandwidth of the drive and no file-system prefetching can further improve performance.

On two disks, the second drives' readahead further reduces stall only a little as the latency of initiating an access and transferring the data from the disk to the buffer cache starts to dominate performance in the absence of prefetching. In contrast, both sequential readahead and deep informed prefetching expose enough I/O concurrency to keep both drives busy, overlap the data transfer from one disk with computation on data read from the other and eliminate most of the I/O stall time. The bandwidth of two disks is nearly enough to keep up with Davidson.

On three or more disks, there is ample I/O bandwidth to keep the CPU busy. As long as the prefetch depth is deep enough to take advantage of array parallelism, it is possible to virtually eliminate I/O stalls.

Even though the model developed for stall as a function of prefetch depth in Chapter 4 does not include complexities such as clustering, the sequentiality of the I/O workload, or even array size, the basic notion of the prefetch horizon still holds for the prefetch-only experiments shown in Figure 6.2b. $\hat{P}$ was intended to be an upper-bound on the depth of prefetching needed to eliminate stall on a large array. For Davidson, which has non-zero application CPU time, T_{app}, and performs sequential accesses which complete in much less time than random accesses ($T_{disk} < 15$ msec), prefetching to a depth of 16 on three or more disks is enough to virtually eliminate stall. Smaller arrays don't have enough bandwidth to eliminate stall. Nevertheless, on these smaller arrays, stall reaches a minimum within the prefetch horizon. As we will see, this same observation holds to varying degrees for all of the benchmarks; $\hat{P}$ serves as an upper bound on the prefetch depth needed to minimize stall for all array sizes. However, recent work has shown that, in the presence of caching, there is benefit in taking advantage of disk idleness that may occur when accessing cached data for deeper prefetching. In Chapter 7, I will explore why $\hat{P}$ is a reasonable upper bound for all array sizes and discuss in more depth the relationship of these findings to the recent work on exploiting disk idleness.

As Table 6.2 shows, none of the systems without informed caching uses the 12 MBytes of cache buffers well. Because the 16.3 MByte matrix does not fit in the cache, the LRU replacement algorithm ejects all of the blocks before any of them are reused. Indeed, 18,780 of the 21,000 reuse hits that the LRU queue scores result from Davidson's non-block-aligned accesses; Davidson often ends a read in the middle of a file block, and scores a reuse hit when it reads the rest of the block with the next read system call. Most of the remaining reuse hits result from page faults serviced from the cache. (Digital UNIX's

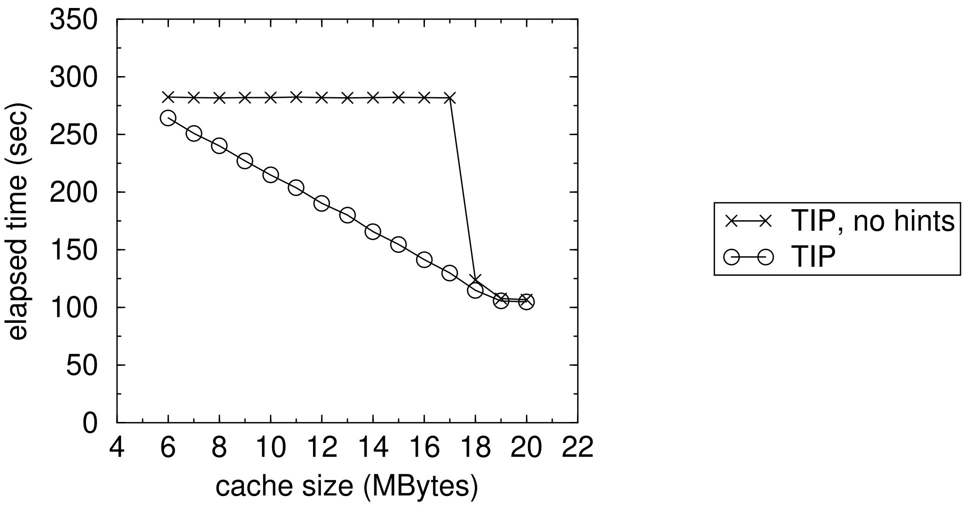


Figure 6.3. Davidson performance vs. cache size. This graph shows the elapsed time for the Davidson benchmark as a function of cache size on a single disk. Informed caching in *TIP* discovers an MRU-like policy which uses additional buffers to increase cache hits and reduce elapsed time. In contrast, LRU caching in *TIP, no hints* derives no benefit from additional buffers until there are enough of them to cache both the entire dataset and needed shared libraries which also occupy cache buffers. Informed caching's advantage leads, in the most dramatic case, to a 54% reduction in elapsed time with a 17-MByte cache.

higher reuse count results from its greater number of page faults.) Overall, these systems would do no worse if instead of hundreds of blocks from the large matrix, they only cached the single most recently used block.

With informed caching turned on, TIP effectively uses the cache buffers to score nearly 75,000 additional reuse hits in the large matrix which reduces the number of blocks fetched from disk by an equivalent number. On one disk, this reduces elapsed time by 36% compared to informed prefetching alone. When disk bandwidth is inadequate, improved caching avoids disk latency. Figure 6.3 shows Davidson's elapsed time with one disk on TIP with and without informed caching as a function of cache size. With standard LRU caching, extra buffers are of no use until the entire dataset fits in the cache. In contrast, informed caching with TIP's min-max global valuation of blocks yields the smooth exploitation of additional cache buffers that is expected from an MRU replacement policy.

On more disks, prefetching masks disk latency, but informed caching still reduces elapsed time an additional 8% by avoiding the CPU overhead of extra disk accesses. The prefetch horizon limits the use of buffers for prefetching and so avoids a pitfall of more aggressive prefetching strategies which may use excess I/O bandwidth to prefetch too deeply and flush all cached blocks [Kimbrel96]. TIP effectively balances the allocation of cache buffers between prefetching and caching.

In general, the benefit of informed caching is sensitive to the spacial locality of an application's I/O workload and how well conventional caching is working. If the cache is small relative to the number of distinct blocks that are repeatedly accessed, as it is towards the left in Figure 6.3, then not even optimal caching can reduce elapsed time by more than a small percentage. On the other hand, if the cache is large enough to hold all accessed blocks, then all block reuses are cache hits regardless of caching policy. This is the case towards the right in the figure. However, great gains are possible when the cache is large enough to hold a substantial portion of the blocks reused, but conventional caching techniques are failing to deliver cache hits.

6.3.2 XDataSlice

XDataSlice (XDS) is an interactive scientific visualization tool that allows scientists to view arbitrary slices through a 3-D dataset. The XDS benchmark simulates this behavior by rendering a sequence of 25 random slices from a dataset consisting of 512^3 32-bit floating point values and requiring 512 MBytes of disk storage. Figures 6.4 and 6.5 show the benchmark's access pattern which is a series of short sequential segments separated by some stride. The length of the sequential segments is roughly proportional to the projection of the slice being rendered onto the z-axis, the axis along which blocks are stored sequentially. Neither the sequences nor the strides are completely regular because of edge effects and discretization as arbitrary slice orientations are mapped to integer-sized data blocks.

Figure 6.6a shows the average elapsed time on the usual four configurations plus *TIP, no prefetching* which has hints but does not use them for informed prefetching, disables DU's heuristic readahead for the hinted data, and does not read hinted blocks in clusters. Thus, *TIP, no prefetching* represents performance when almost every access is a miss. Table 6.3 gives the corresponding raw numbers. As was the case with Davidson, Digital UNIX and TIP without hints have comparable performance with the exception of a small difference in faults revealed by the prefetching and caching numbers in Table 6.4.

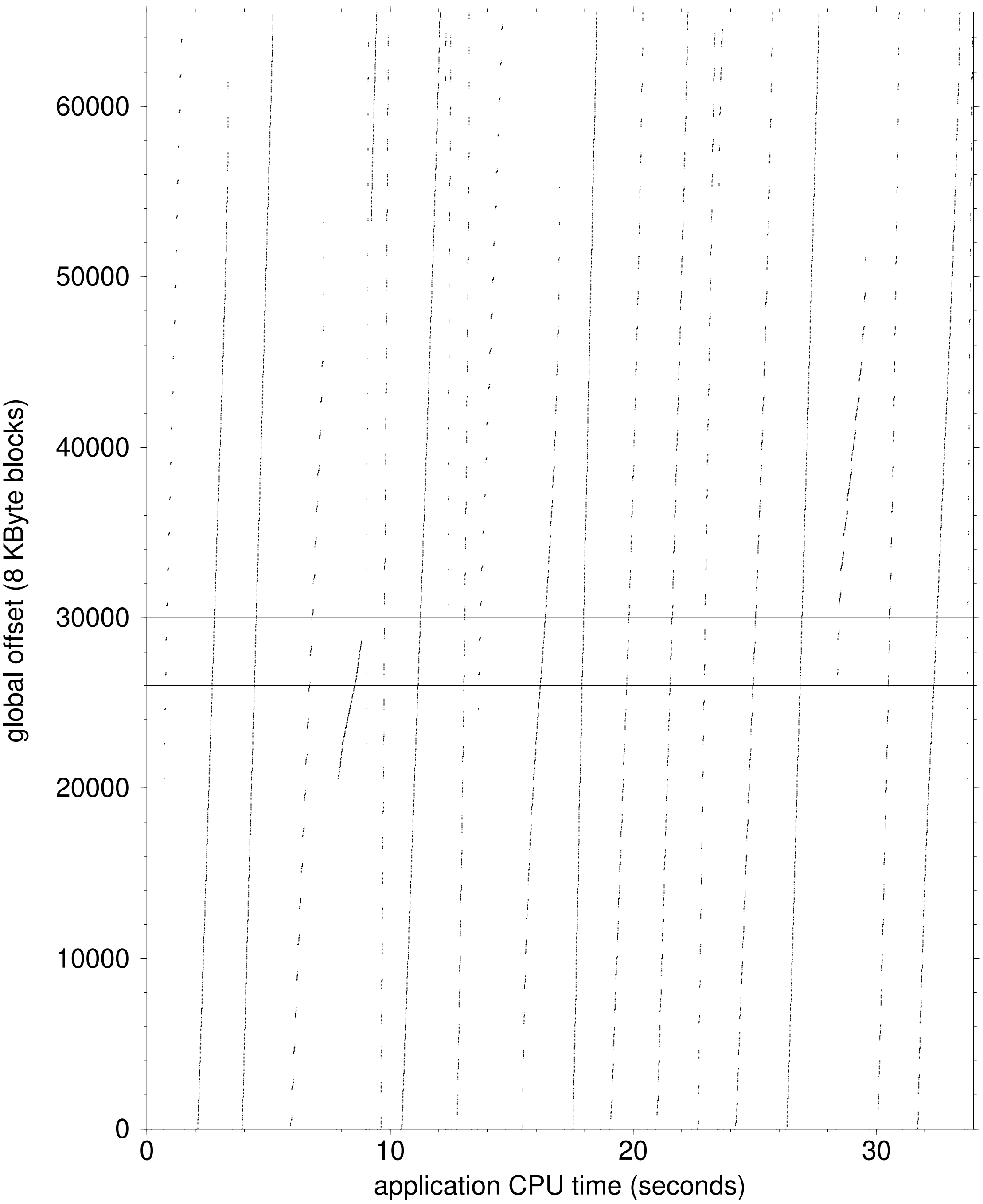


Figure 6.4. XDataSlice access pattern. This graph shows which blocks XDS accessed at different times during the benchmark. XDS reads blocks needed to render a slice in ascending order. Thus, each of the 25 slices fetched produces one monotonically increasing sequence of accesses in this graph. XDS delivers hints for one slice at a time. All of the access patterns may be loosely described as strided. But, not only do the strides vary from slice to slice, they vary within a single slice. In fact, this graph is misleadingly simple because the y-axis represents about 10,000 blocks per inch which is more than printers or the human eye can resolve. Most of the access which appear to be sequential in this graph are actually strided. To show this, Figure 6.5 graphs the region from y= 26,000 – 30,000 which is demarcated by the horizontal lines in this figure.

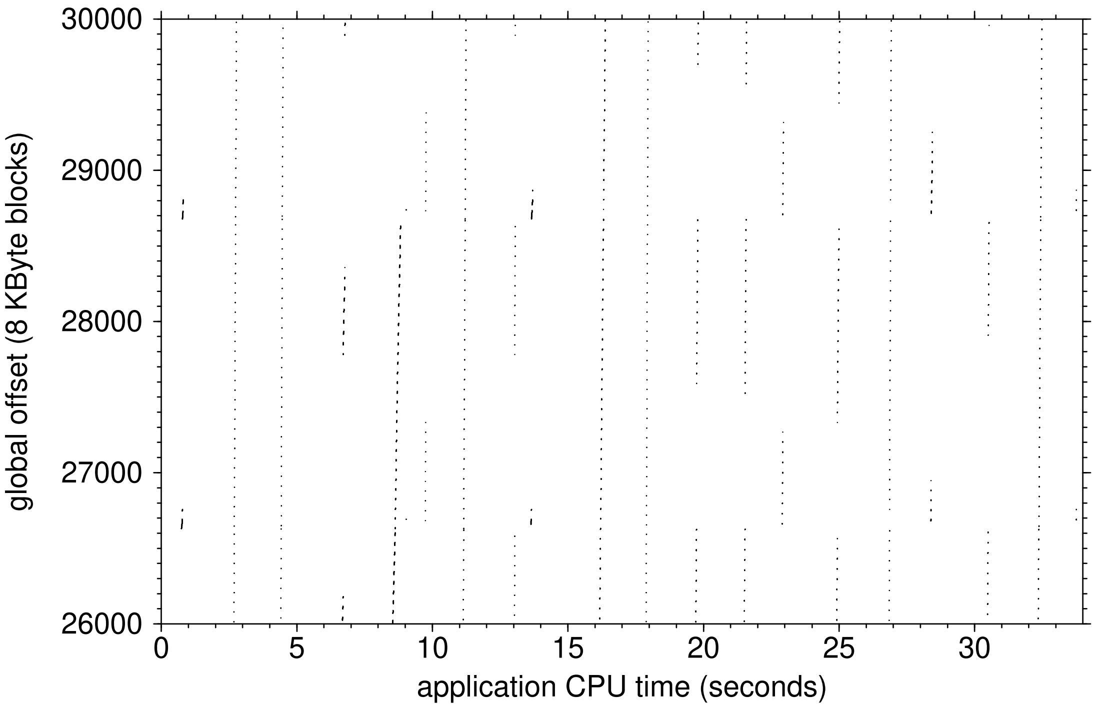
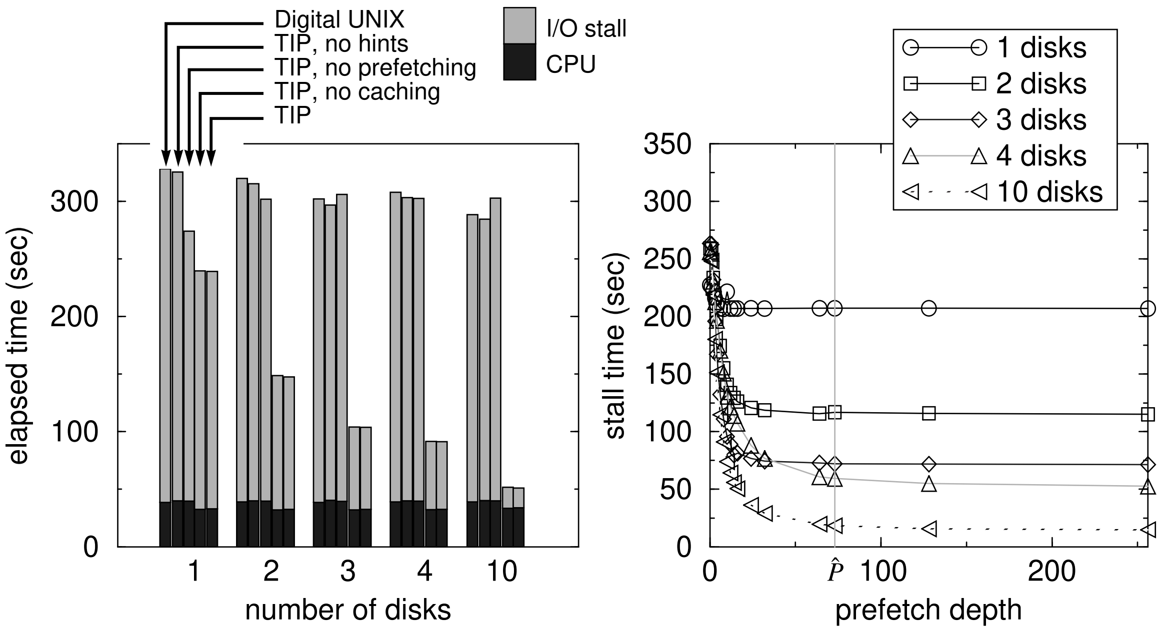


Figure 6.5. Close-up of XDataSlice's accesses to a small range of its dataset. This graph expands the region between the horizontal lines in Figure 6.4. It shows that accesses that appeared to be sequential in that figure are, in fact, themselves strided. In this graph, some of the accesses which appear to be single blocks, are in fact short sequential runs. For example, most of the dots near x=2.7 seconds represent reads of three sequential blocks which is enough to trigger DU's sequential readahead heuristic. Although strided, XDS' access pattern is complex. It would be a challenge to develop a heuristic algorithm that could quickly lock on to its pattern and prefetch deeply with high accuracy.

Although sequential readahead worked well for Davidson, it fails miserably for XDS's short sequential reads. First, it does not provide the concurrency needed to take advantage of disk array parallelism so, without hints, XDS performs little better on 10 disks than it does on 1. Indeed, when bandwidth is scarce on one disk, Digital UNIX's readahead actually hurts rather than helps XDataSlice as shown by the performance of *TIP, no prefetching*. From Table 6.4, of the roughly 60,000 blocks prefetched by the sequential readahead heuristic for the first two configurations, only about 25,000 ever become prefetch hits. Overall, Digital UNIX reads 1.7 times as many blocks from disk as are accessed. On a single disk, readahead increases elapsed time by about 50 seconds or 18%. On ten disks, there is bandwidth to spare, so the useless readaheads are less likely to delay a read for needed data and the 25,000 good readaheads have a chance to improve performance. The



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.6. XDataSlice performance. Graph (a) shows the elapsed time for rendering 25 random slices through a 512 MByte dataset. In *TIP, no prefetching*, TIP is not prefetching or using DU's heuristic readahead for the hinted data. Without TIP's informed prefetching, the system makes poor use of the disk array because it doesn't know what to prefetch. In fact, DU's heuristic readahead prefetches so many unused blocks that it hurts performance on one disk. But, informed by hints, TIP is able to prefetch in parallel, mask the latency of the many seeks, and reduce overall elapsed time by 82%. There is very little data reuse, so informed caching does not further decrease elapsed time. Graph (b) shows stall as a function of prefetch depth for the *TIP, no caching* case. For this application which performs many seeks and has little computation between reads, there is benefit in prefetching out to the prefetch horizon, but little beyond that. An unbalanced load (see Figure 6.7) on four disk causes the stall on four disks to exceed the stall on three disks at prefetching depths less than about 32.

net result is that sequential readahead on ten disks reduces elapsed time by about 20 seconds. The cross-over point is at three disks.

With hints, informed prefetching knows what to prefetch. Thus, TIP can prefetch aggressively in parallel and exploit the bandwidth of the disk array. On ten disks, the result is a 93% reduction in I/O stall time which leads to an 82% reduction in elapsed time. Because informed prefetching does not waste I/Os, the TIP system does not have to pay the CPU overhead, T_{driver}, of performing Digital UNIX's useless readaheads. Thus, informed prefetching reduces CPU time as well as I/O stall time.

Figure 6.6b shows the effect of informed prefetching on stall time as a function of prefetch depth. The curves reveal a number of interesting peculiarities in XDataSlice's performance. First, at a prefetch depth of 0, stall is greater on two or more disks than on

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
Digital UNIX	38.66 (0.11)	289.71 (1.05)	38.96 (0.36)	281.16 (0.77)	38.65 (0.07)	263.43 (2.08)	38.91 (0.29)	269.07 (1.18)	38.87 (0.05)	249.61 (1.07)
TIP, no hints	39.95 (0.21)	285.61 (0.57)	39.94 (0.14)	275.53 (0.69)	40.27 (0.32)	256.52 (1.47)	39.87 (0.08)	263.46 (0.44)	40.01 (0.28)	244.57 (0.82)
TIP, no prefetch	39.68 (0.29)	234.62 (0.42)	39.77 (0.59)	262.18 (1.28)	39.46 (0.23)	266.63 (2.43)	39.50 (0.27)	263.01 (0.76)	39.90 (0.57)	263.09 (1.44)
TIP, no caching	32.39 (0.12)	207.26 (0.20)	32.01 (0.08)	116.70 (1.51)	32.04 (0.09)	72.07 (0.19)	32.31 (0.20)	59.25 (0.17)	33.44 (0.15)	18.27 (0.41)
TIP	32.93 (0.18)	206.13 (0.13)	32.44 (0.11)	115.07 (0.44)	32.59 (0.17)	71.22 (0.33)	32.63 (0.33)	58.57 (0.62)	33.82 (0.13)	17.33 (0.12)

Table 6.3. XDataSlice elapsed time. This table presents the raw numbers graphically portrayed in Figure 6.6. Units are seconds and the numbers in parentheses are the 95% confidence intervals.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
Digital UNIX	55194 (41)	8829 (31)	62011 (4)	23171 (2)	7808 (31)	26480 (2)	20905 (10)	20593 (9)
TIP, no hints	48166 (14)	1799 (0)	60710 (3)	22681 (2)	2641 (4)	25381 (1)	20144 (9)	20101 (9)
TIP, no prefetch	48160 (0)	1799 (0)	179 (1)	65 (1)	2669 (0)	169 (1)	45321 (1)	45279 (1)
TIP, no caching	48160 (0)	1799 (0)	45328 (1)	14871 (1)	2662 (0)	45318 (1)	179 (1)	137 (1)
TIP	48160 (0)	1799 (0)	45307 (1)	14863 (0)	2688 (1)	45297 (1)	174 (0)	132 (0)

Table 6.4. XDataSlice prefetching and caching performance. Aggressive sequential readahead in Digital UNIX and TIP without hints works poorly for this workload of many short sequential runs. Of over 60,000 prefetches, only about 25,000 ever become prefetch hits. Informed prefetching knows what not to prefetch as well as what to prefetch and so can prefetch aggressively and accurately. There is very little reuse in this workload, so informed caching does not significantly increase reuse hits.

one. This is because the typical sequential run is four or five blocks. On one disk, XDS seeks once to the start of the run and then does efficient sequential accesses for the rest of the run. On two or more disks, however, a single sequential run often spills over into the next stripe unit so that the application incurs the latency of two seeks instead of one. As the prefetching depth increases, this second seek is overlapped with one or more other seeks and its impact is eventually offset by the larger array's higher transfer bandwidth and ability to fetch multiple sequential runs simultaneously.

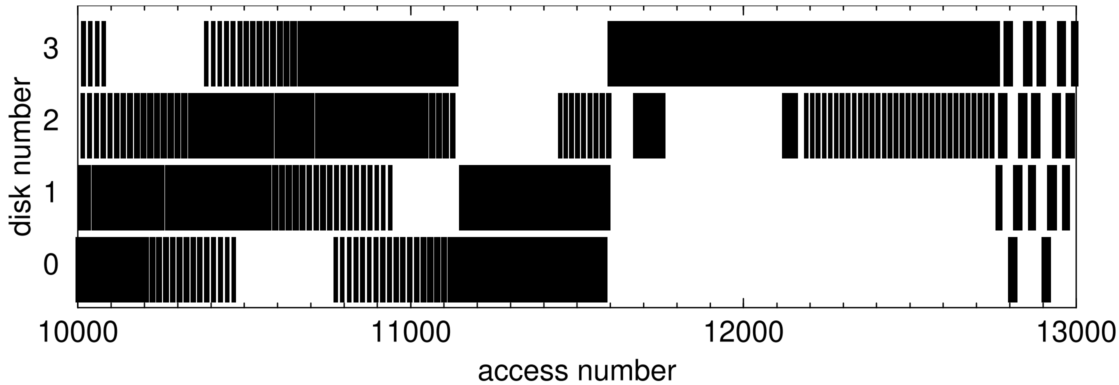


Figure 6.7. XDataSlice disk load distribution for a range of accesses on a four-disk array. This figure shows how activity is distributed across the disks in a four-disk array for a range of 3000 of XDS' total of about 50,000 accesses. Some slices and therefore sequences of accesses are reasonably well-balanced, but some, such as this one, are highly unbalanced. For example, all of the accesses from about 11,600 to 12,700 go to only two disks and most only go one disk. For this period of over 1000 accesses, disks 0 and 1 sit idle.

Another interesting result is that the I/O stall time on 4 disks is greater than on 3 disks up to a prefetch depth of about 32. Because the dataset dimensions, the block size, and the stripe unit are all powers of 2, some slices have a pathologically unbalanced workload on a 4-disk array as shown in Figure 6.7. The situation is analogous to the contention one can find in the interleaved memory of a supercomputer. Using a prime number of disks in the array, or randomizing the assignment of stripe units to disks could probably help alleviate the problem. The effect of the unbalanced load is that the full parallelism of the array is not always available and performance suffers. Prefetching deeply can smooth over these transient load imbalances which is why the performance of the four-disk array eventually caches up to and surpasses that of the three-disk array. Ideally, prefetching would be sensitive to load imbalances and be deeper when beneficial. This observation has led to work developing such adaptability [Kimbrel95, Kimbrel96a, Kimbrel96, Tomkins97]. I will discuss the relationship of that work to the approach described here in Chapter 7. But, such adaptability is beyond the scope of this dissertation which is limited to an investigation of prefetching and caching algorithms that do not rely on specific information about data layout.

Even with the unbalanced load, the prefetch horizon of $\hat{P}=73$ captures most of the potential stall reduction, as shown in Figure 6.6b, and stall decreases very little beyond $\hat{P}$ for any array size. At the same time, the prefetch horizon successfully captures the knee in the curve for 4 and 10 disks; there is benefit in prefetching out to the prefetch horizon. Davidson performs a significant amount of computation between sequential reads and therefore, in the absence of caching, does not benefit from increasing the prefetch depth beyond 16. But, XDS does little computation between reads (T_{app} is small), and its reads often require seeks to new locations (T_{disk} is large), so its application-specific prefetch horizon is close to the system-wide upper bound, $\hat{P}$.

Because XDataSlice is reading thin slices from a very large dataset, there is very little reuse in its workload. Consequently, there is little opportunity for informed caching to avoid I/O accesses. For similar reasons informed clustering does not play a substantial role. Standard clustering of sequential reads is important, but because hints are given for one slice at a time, and because accesses within a slice are in ascending order, there is little opportunity for informed clustering to combine multiple, widely-separated accesses into a single larger one.

XDataSlice originally required all data to be memory-resident to render slices interactively. These results show that with informed prefetching and a disk array, this application can run out-of-core and still render a slice from a very large dataset in about two seconds. Informed prefetching doesn't just improve performance; for XDataSlice, it is an enabling technology that provides important new out-of-core capability.

6.3.3 Sphinx

Sphinx is a high-quality, speaker-independent, continuous-voice, speech-recognition system. In our experiments, Sphinx is recognizing an 18-second recording commonly used in Sphinx regression testing. Figure 6.8 shows its access pattern which includes an initialization scan of its language models followed by a recognition phase during which it dynamically loads needed language data. Roughly 65,000 of Sphinx's 78,000 block reads occur during the initialization phase. Figure 6.9 shows the elapsed time for the benchmark, Table 6.5 gives the corresponding elapsed-time numbers, and Table 6.6 shows caching and prefetching performance.

Digital UNIX's sequential readahead and a two-disk array help Sphinx during its initialization scan even though the skips lead to some false readahead. But, with informed prefetching, it takes advantage of the array even for the many small accesses during the

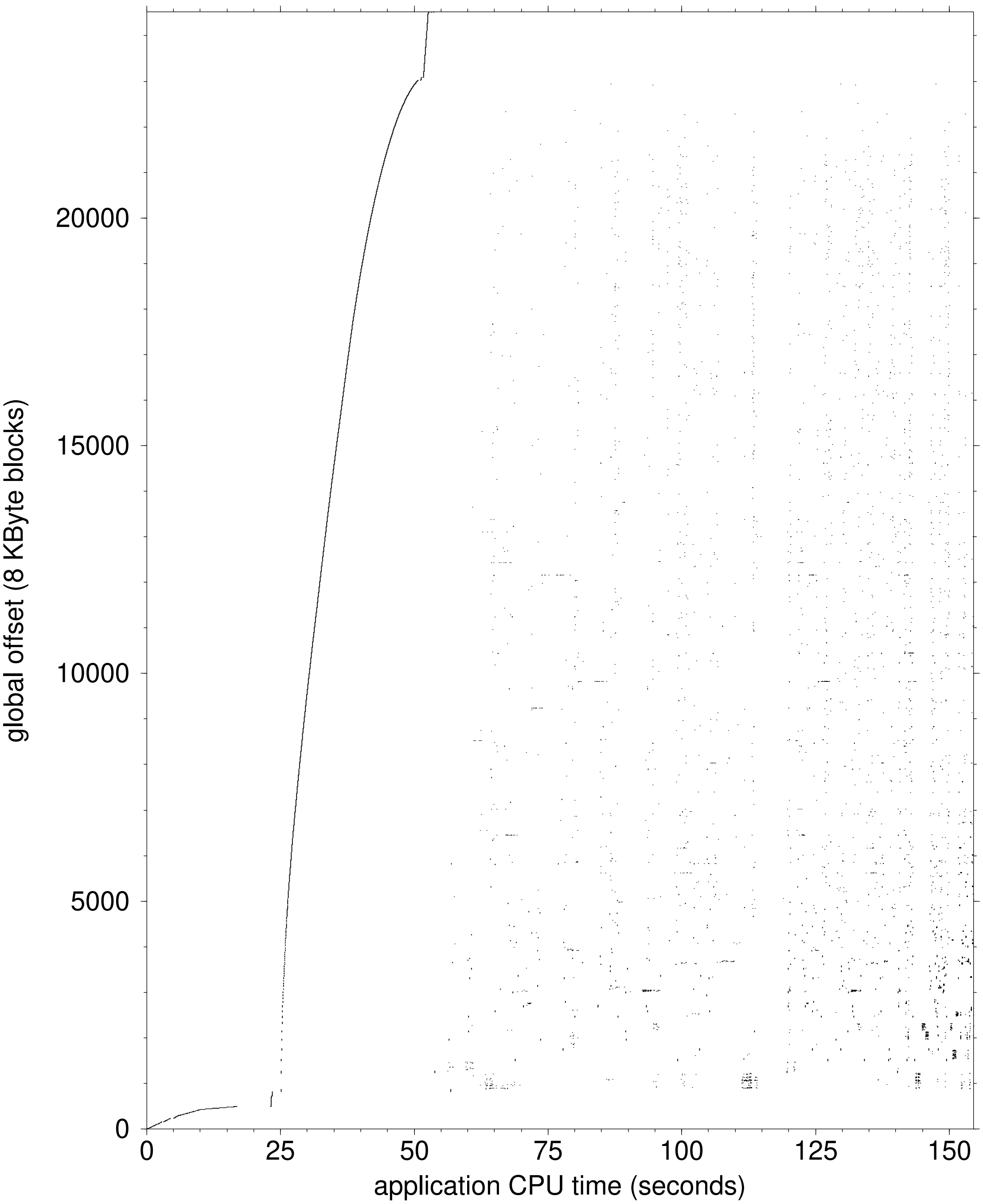
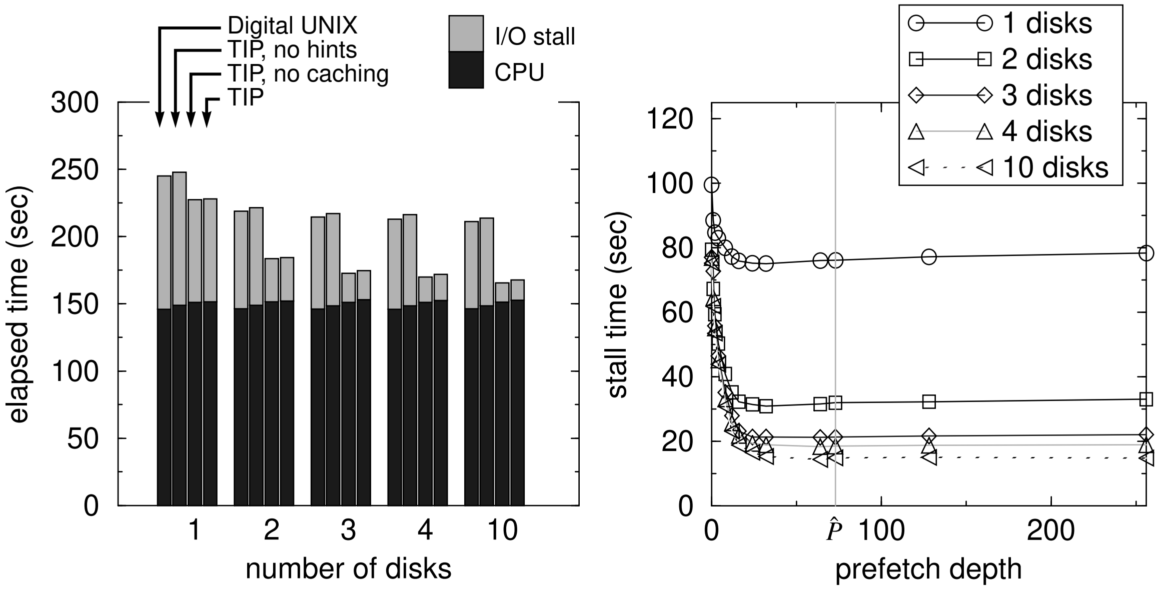


Figure 6.8. Sphinx access pattern. During the first 53 seconds, Sphinx initializes itself by scanning its language models and dictionaries to build internal tables. One language-model file accounts for all the blocks from 493–23,016 in the graph. The scans are in ascending order, but there are skips which decrease in size until, from global offset 13768, the skips are smaller than a block so all blocks are accessed. After this initialization phase, Sphinx dynamically hints and loads pieces of the language model as needed during the recognition phase. The file holding the digitized speech being recognized only occupies 8 blocks and so is not visible on the graph.



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.9. Sphinx performance. As (a) shows, without hints Sphinx derives only a small benefit from the disk array. With hints, TIP's informed prefetch takes advantage of the array for the random loads of pieces of the language model. Informed caching does not help because Sphinx gives mostly small bursts of hints and has poor access locality. Figure (b) shows the familiar relationship between stall time and prefetch depth. In this case, however, the advantage of deeper prefetching on a single disk is not better disk scheduling, but increased resilience to Sphinx's bursty I/O accesses.

recognition phase that dynamically load the needed parts of the language model. These small reads also lead to some false readahead, although to a much smaller extent than for XDataSlice. On just three disks, informed prefetching reduces I/O stall time by 69% which translates into a 21% reduction in elapsed time for this compute-intensive application.

Referring to Figure 6.9b, we see the familiar curve of stall time vs. prefetch depth. In this case, however, the results on one disk point to an interesting new phenomenon. Normally, when the stall time drops off on one disk, the expectation is that it is due to disk scheduling advantages of queuing multiple requests. In this case, however, virtually all of the hints during the initialization phase and over 85% of the hints during the recognition phase are given in ascending order of block address so there is little opportunity for disk scheduling to reduce access time. Instead, the greater prefetch depth is providing resiliency to Sphinx's bursty I/O workload. Sphinx often pauses to compute for ten or more milliseconds in the middle of a hinted burst of reads. By prefetching more deeply, TIP can take advantage of the computation-induced lulls in I/O activity, to get ahead of Sphinx's

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
Digital UNIX	145.88 (0.67)	99.29 (0.37)	146.31 (0.85)	72.73 (0.32)	146.16 (0.80)	68.55 (0.42)	145.94 (0.15)	67.17 (0.38)	146.29 (0.81)	64.98 (0.37)
TIP, no hints	148.89 (0.65)	98.15 (0.25)	149.34 (0.51)	72.49 (1.10)	149.02 (0.62)	68.34 (0.46)	149.18 (0.49)	67.37 (0.44)	149.49 (0.80)	65.08 (0.42)
TIP, no caching	152.87 (0.53)	76.15 (0.26)	153.10 (0.80)	32.01 (0.48)	150.77 (0.92)	21.31 (0.36)	153.19 (0.81)	18.55 (0.45)	150.81 (0.34)	14.93 (0.85)
TIP	152.26 (0.52)	76.36 (0.65)	152.13 (0.85)	31.55 (0.40)	153.02 (0.50)	20.94 (0.30)	152.09 (0.67)	18.14 (0.24)	152.56 (1.28)	14.20 (0.15)

Table 6.5. Sphinx elapsed time. These are the data graphed in Figure 6.9a. The numbers in parentheses are the 95% confidence intervals for the averages of the five runs.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
Digital UNIX	78879 (111)	1142 (113)	21135 (38)	4356 (12)	51125 (40)	17822 (22)	9930 (80)	4570 (24)
TIP, no hints	78665 (4)	929 (0)	21103 (19)	4343 (7)	50909 (49)	17808 (10)	9947 (47)	4554 (14)
TIP, no caching	78393 (0)	890 (0)	26849 (14)	7731 (4)	51307 (14)	26730 (14)	355 (3)	288 (3)
TIP	78363 (9)	860 (9)	26764 (92)	6487 (21)	51368 (97)	26648 (93)	346 (0)	279 (0)

Table 6.6. Sphinx prefetching and caching performance. These data show that informed prefetching achieves almost 50% more prefetch hits with only 26% more prefetches than sequential readahead because it can prefetch accurately. Although there is substantial buffer reuse by Sphinx, LRU queue profiling reveals that virtually all of these reuse hits occur in the most recently used segment of the LRU queue and that there is little opportunity for informed caching to improve performance.

data requests. Because these lulls are short, a prefetch depth of only 16 is sufficient to take full advantage of them.

For a number of reasons, informed caching does not help Sphinx. Although there are a good number of reuse hits, as shown in Table 6.6, most of these are a result of multiple partial-block reads that hit in the first segment of the LRU queue. Sphinx's internal cache and large datasets lead to little locality in its file accesses beyond this. Furthermore, Sphinx's small bursts of hints do not give TIP sufficient advance knowledge to significantly improve cache performance. In general, informed caching requires hints much further in advance than does informed prefetching.

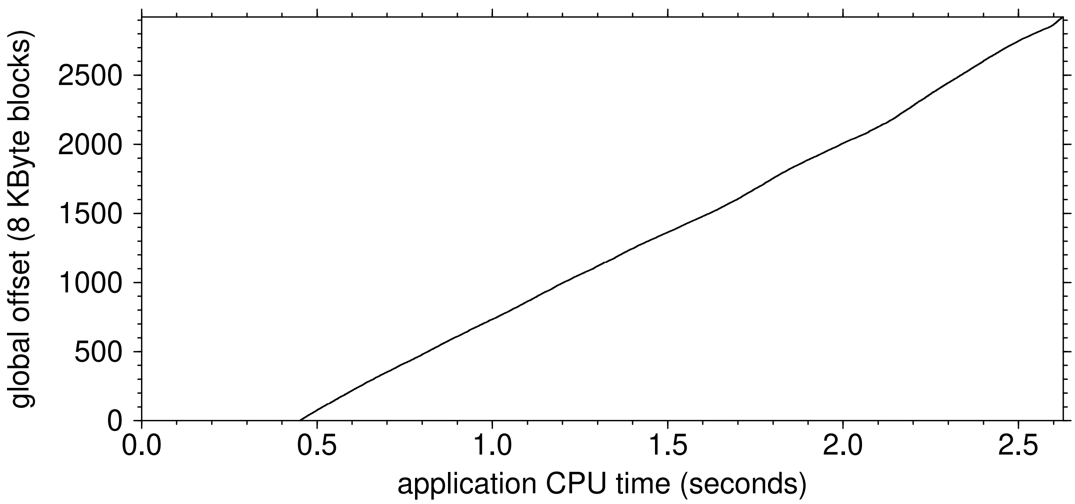
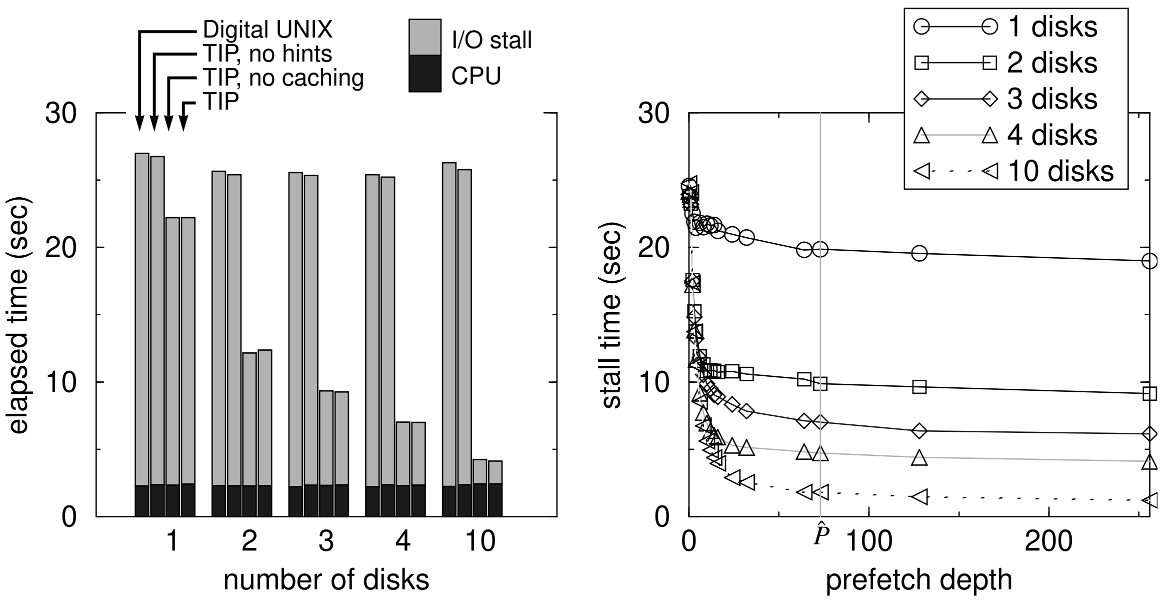


Figure 6.10. Agrep access pattern. In the benchmark, Agrep searches 1349 files sequentially. This results in the trivial access pattern shown here. But, because the single linear global block addresses used for the *y*-axis map to so many separate files, the file system does not observe the accesses as one single sequential read, but as many short, disjoint sequential reads for which sequential readahead is not too useful. Agrep's hints disclose the larger pattern and enable TIP to prefetch across files and not just within individual files. During the delay in starting the search, Agrep is checking each of its arguments to make sure they are valid file names. As it does so, it discloses the eventual sequential read of the file in a hint.

6.3.4 Agrep

In this benchmark, Agrep searches 1349 kernel source files occupying 2922 disk blocks for a simple string that does not occur in any of the files. Figure 6.10 shows the misleadingly simple global access pattern; misleading because the single large sequential access is actually a concatenation of sequential accesses to the many files. Over 80% of these files have only one or two blocks and are therefore too small for any sequential readahead. Over 94% consist of five blocks or less. Only four of the 1349 files are larger than 20 blocks and the largest has 38. The average number is 2.17 blocks.

Figure 6.11 shows the elapsed and stall times for this search and Table 6.7 gives the numbers for part (a) of the figure. As was the case for both XDataSlice and Sphinx, without informed prefetching there is little parallelism in Agrep's I/O workload. Even though the files are searched sequentially, because most are small, even aggressive sequential readahead successfully prefetches only about a third of the blocks (see Table 6.8) and does not achieve parallel transfer. However, Agrep's disclosure of future accesses exposes potential I/O concurrency not within individual files, but across multiple files. Arrays of as few as four disks reduce elapsed time by 72% and of ten disks reduced elapsed time by 84%.



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.11. Agrep performance. This figure shows the elapsed time (a) and stall time (b) for searches through 1349 files in three directories. Most of the files are not large enough for sequential readahead to expose concurrency and take advantage of the disk array. Disclosure exposes concurrency across files that informed prefetching uses to reduce elapsed time by up to 84%. Because there is no data reuse by this application, there is no opportunity for informed caching.

In this benchmark, all the files are read only once, so there is no data reuse and therefor no opportunity for informed caching benefits.

6.3.5 Gnuld

The Gnuld benchmark links the 562 object files of a TIP kernel. These object files comprise approximately 64 MBytes, and produce an 8.8-MByte kernel. Figure 6.12 shows the access pattern for the benchmark. Figure 6.13 presents the elapsed and I/O stall times for this test and Table 6.9 gives the numerical data for the elapsed time.

Gnuld is another example of a serial I/O workload that is unable to take advantage of disk-array parallelism as is seen from the flat performance across array sizes for both the *Digital UNIX* and *TIP, no hints* runs. For most of its accesses, Gnuld is looping over the object files reading small segments from each. As was the case for XDataSlice, sequential readahead actually hurts performance for this workload. As Table 6.10 shows, false readahead leads to the wasted prefetching of over 1100 blocks. But, here, there is an additional penalty of false readahead: more cache misses. Comparing the reuse hits for *Tip, no hints*

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
Digital UNIX	2.27 (0.03)	24.72 (0.30)	2.30 (0.11)	23.37 (0.73)	2.22 (0.03)	23.35 (0.96)	2.22 (0.03)	23.19 (0.70)	2.23 (0.04)	24.08 (0.88)
TIP, no hints	2.36 (0.02)	24.40 (0.34)	2.30 (0.05)	23.10 (0.84)	2.34 (0.03)	23.01 (0.68)	2.37 (0.15)	22.87 (0.59)	2.38 (0.04)	23.41 (0.66)
TIP, no caching	2.34 (0.03)	19.86 (0.32)	2.28 (0.02)	9.87 (0.19)	2.31 (0.05)	7.02 (0.20)	2.29 (0.03)	4.72 (0.13)	2.44 (0.18)	1.80 (0.13)
TIP	2.40 (0.03)	19.81 (0.45)	2.30 (0.02)	10.07 (0.20)	2.35 (0.02)	6.92 (0.25)	2.33 (0.05)	4.66 (0.12)	2.43 (0.01)	1.69 (0.02)

Table 6.7. Agrep elapsed time. These are the data graphed in Figure 6.11a. The numbers in parentheses are the 95% confidence intervals for the averages of the five runs.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
Digital UNIX	3425 (25)	497 (25)	1077 (367)	613 (261)	407 (19)	1065 (366)	1953 (358)	1933 (357)
TIP, no hints	3305 (13)	377 (13)	1050 (358)	598 (256)	335 (14)	1044 (358)	1925 (357)	1920 (358)
TIP, no caching	3309 (7)	381 (7)	2949 (1)	1771 (1)	341 (6)	2945 (1)	23 (3)	13 (1)
TIP	3307 (11)	379 (11)	2951 (1)	1771 (0)	337 (11)	2949 (3)	21 (4)	12 (1)

Table 6.8. Agrep prefetching and caching performance. Although Agrep searches the file sequentially, most of the files are short, so sequential readahead only prefetches about a third of the blocks. None of the blocks from the searched files are reused, so there is no opportunity for informed caching.

and *TIP, no prefetch*, we see that the false readahead in *no hints* reduces the number of reuse hits by 330. Whereas false prefetches delay useful prefetches less on larger arrays, these cache misses incur the full latency of a disk access no matter what the array size is.

Informed prefetching in *TIP, no caching* suffers from neither false prefetching nor a decrease in reuse hits. It takes advantage of a ten-disk array to eliminate 84% of the I/O stall time and reduce overall elapsed time by 74%.

Informed caching in the *TIP* runs increases reuse hits by 600 compared to prefetching alone in *TIP, no caching*. Most of the avoided misses are for hinted data which in *TIP, no caching* are prefetched back. On a single disk there is insufficient bandwidth to prefetch all of the misses without stall and informed caching reduces elapsed time by 6%.

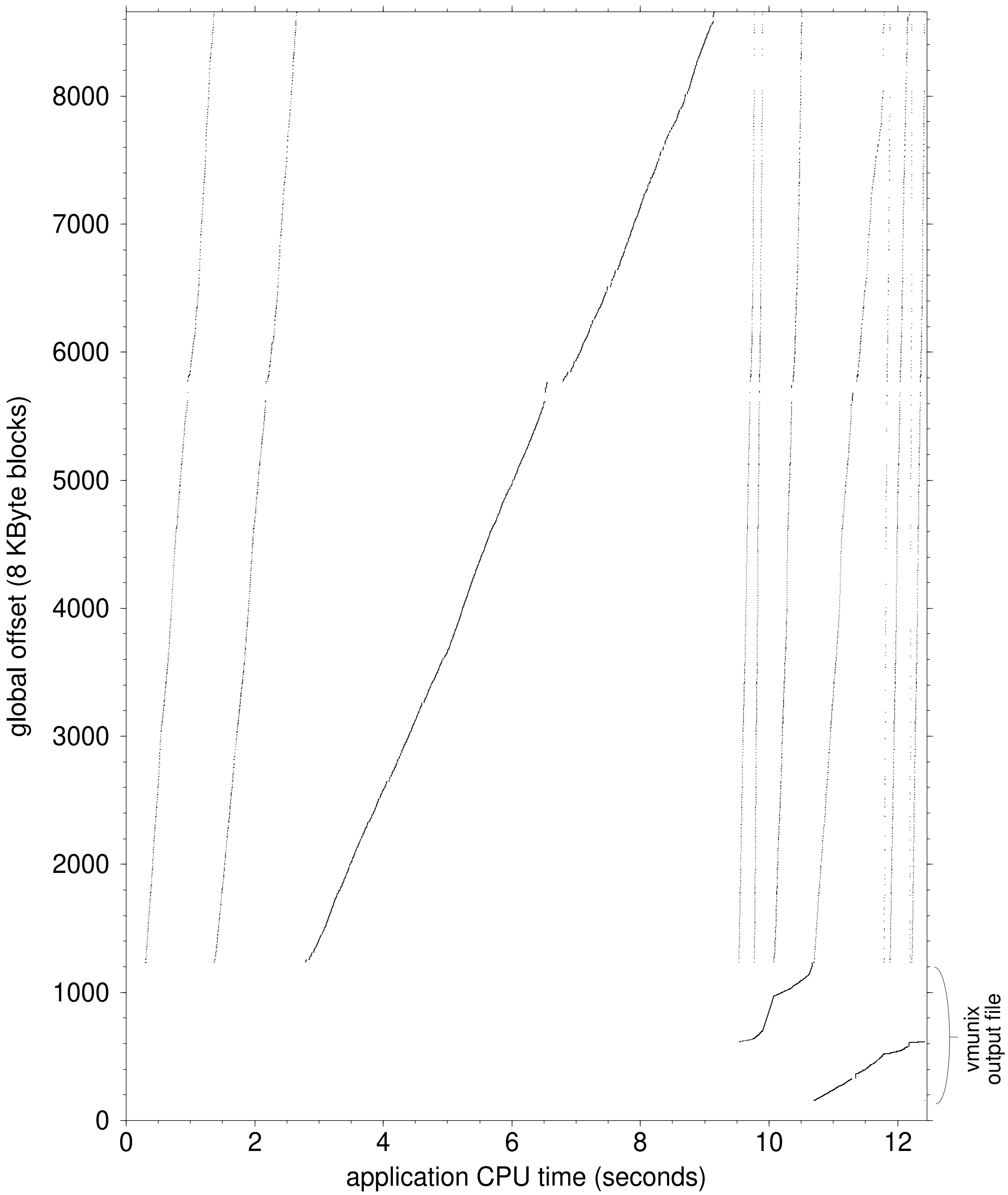
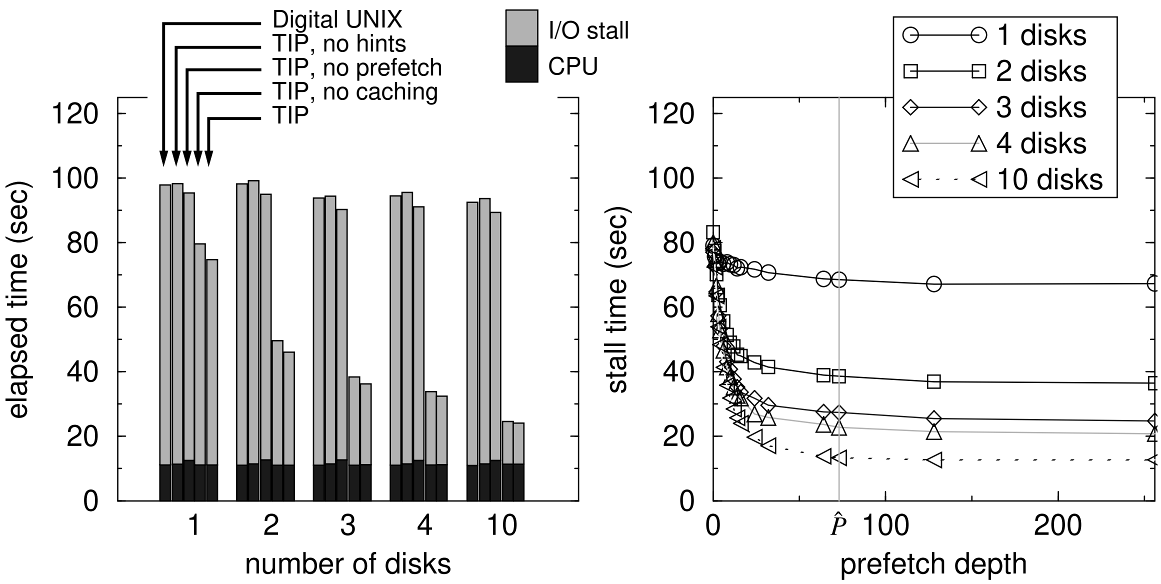


Figure 6.12. Gnuld access pattern. Gnuld makes many passes over the 562 object files it is linking to build a new kernel. It is only after it has read the headers and symbols from each file that it starts writing the new kernel, symbols at the end first and code at the beginning next. The blip at offsets 5620–5772 is the 152-block AFS file system library being loaded. The next largest object file is only 50 blocks in size.



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.13. Gnuld performance. This figure shows the elapsed time (a) and stall time (b) for Gnuld to link a Digital UNIX kernel. Informed prefetching again takes advantage of the disk array where readahead heuristics fail. TIP reduces elapsed time by up to 74%.

6.3.6 Postgres

The Postgres benchmarks are joins of two relations. The outer relation contains 20,000 unindexed tuples (3.2 MBytes) whereas the inner relation has 200,000 tuples (32 MBytes) and is indexed (5 MBytes). In the first benchmark, 20% of the outer-relation tuples find a match in the inner relation. In the second, 80% find a match. One output tuple is written sequentially for every tuple match. Recall from Section 3.4.3, that in the original code, Postgres loops over the outer-relation tuples, interleaving sequential accesses to the outer relation with random accesses to the index and the inner relation. To generate hints, the loop is split and Postgres passes over the outer-relation tuples twice. During the first pass, Postgres performs all the index lookups. It then issues hints for the reads of the matching inner-relation tuples which it performs during the second pass over the outer-relation tuples. Figures 6.14 and 6.15 show the access pattern for the two benchmarks after the loop is split. Figure 6.16 and Tables 6.11 and 6.12 give the results for the 20%-match case and Figure 6.17 and Tables 6.13 and 6.14 give the results for the 80%-match case.

Splitting the loop substantially reduces Postgres elapsed time even without giving hints. On a single disk, it reduces elapsed time by 25% and 35% for the 20%- and 80%-

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
Digital UNIX	11.10 (0.31)	86.77 (0.87)	11.04 (0.10)	87.19 (1.29)	11.05 (0.14)	82.77 (1.63)	10.99 (0.10)	83.52 (1.00)	10.92 (0.09)	81.61 (0.77)
TIP, no hints	11.35 (0.14)	87.04 (0.71)	11.41 (0.11)	87.79 (1.86)	11.46 (0.22)	83.00 (2.91)	11.43 (0.12)	84.19 (1.77)	11.39 (0.08)	82.29 (1.18)
TIP, no prefetch	12.49 (0.03)	82.95 (0.20)	12.70 (0.18)	82.34 (0.20)	12.67 (0.16)	77.65 (0.95)	12.51 (0.03)	78.63 (0.16)	12.54 (0.05)	76.89 (0.22)
TIP, no caching	11.08 (0.08)	68.56 (0.28)	11.03 (0.06)	38.60 (0.22)	11.04 (0.05)	27.37 (0.41)	11.09 (0.06)	22.77 (0.31)	11.32 (0.05)	13.32 (0.23)
TIP	11.11 (0.11)	63.65 (3.46)	11.00 (0.06)	35.11 (1.25)	11.16 (0.09)	25.08 (0.12)	11.23 (0.14)	21.24 (0.85)	11.36 (0.17)	12.70 (0.43)

Table 6.9. Gnuld elapsed time. These are the data graphed in Figure 6.13a. The numbers in parentheses are the 95% confidence intervals for the averages of the five runs.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
Digital UNIX	23734 (48)	1303 (48)	5816 (621)	2710 (321)	12056 (45)	4662 (111)	7016 (97)	5191 (89)
TIP, no hints	23525 (34)	1094 (34)	5784 (615)	2689 (319)	11924 (32)	4631 (109)	6970 (106)	5174 (101)
TIP, no prefetch	23538 (44)	1106 (44)	103 (3)	47 (1)	12252 (43)	82 (4)	11203 (5)	11164 (4)
TIP, no caching	23541 (44)	1109 (44)	11027 (3)	4880 (1)	12247 (43)	11006 (4)	287 (5)	247 (4)
TIP	23536 (32)	1104 (32)	10477 (384)	4431 (304)	12856 (312)	10402 (321)	277 (8)	239 (3)

Table 6.10. Gnuld prefetching and caching performance. Gnuld loops over the object files several times reading short segments from the files. This access pattern defeats sequential readahead and, in fact, leads to some false prefetching which displaces some useful blocks and decreases the number of reuse hits in *TIP, no hints* compared to *TIP, no prefetch*. Informed prefetching accurately prefetches more than twice as many blocks which exposes concurrency for the disk array and does not decrease reuse hits. Informed caching increases reuse hits by 600 which helps reduce stalls when bandwidth is limited by a small array.

Restructuring the code increases the locality of the index accesses which are no longer interleaved with inner-relation tuple accesses. The increased locality increases cache effectiveness and therefore the number of reuse hits as a percentage of total requests from 52% to 65% in the 20%-match case and from 47% to 63% in the 80%-match case. These caching gains dwarf the CPU penalty.

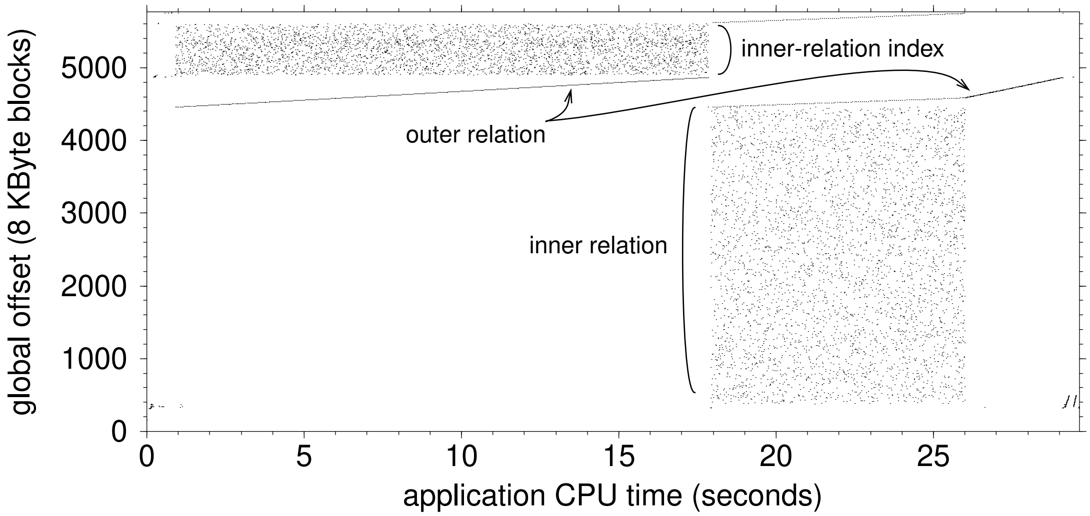


Figure 6.14. Postgres, 20% match, access pattern. On its first pass through the outer relation, Postgres tries to find a match for each record in the inner-relation index. It then discloses the blocks it will read in a hint. On the second pass, it performs 3976 hinted reads of matching records from the 4082-block inner relation. Before the loop-splitting, the accesses to the inner-relation index and data files were interleaved and the access locality was much lower.

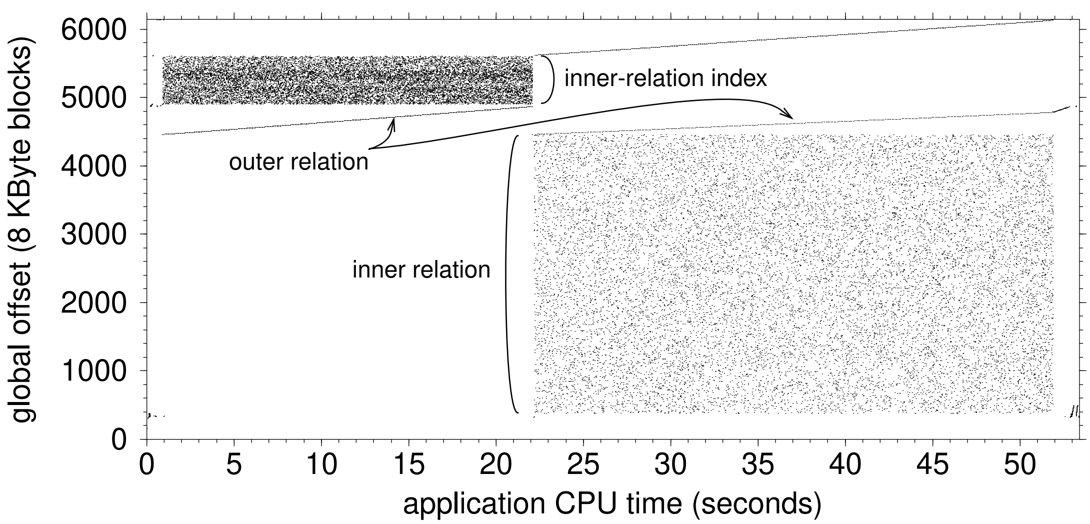
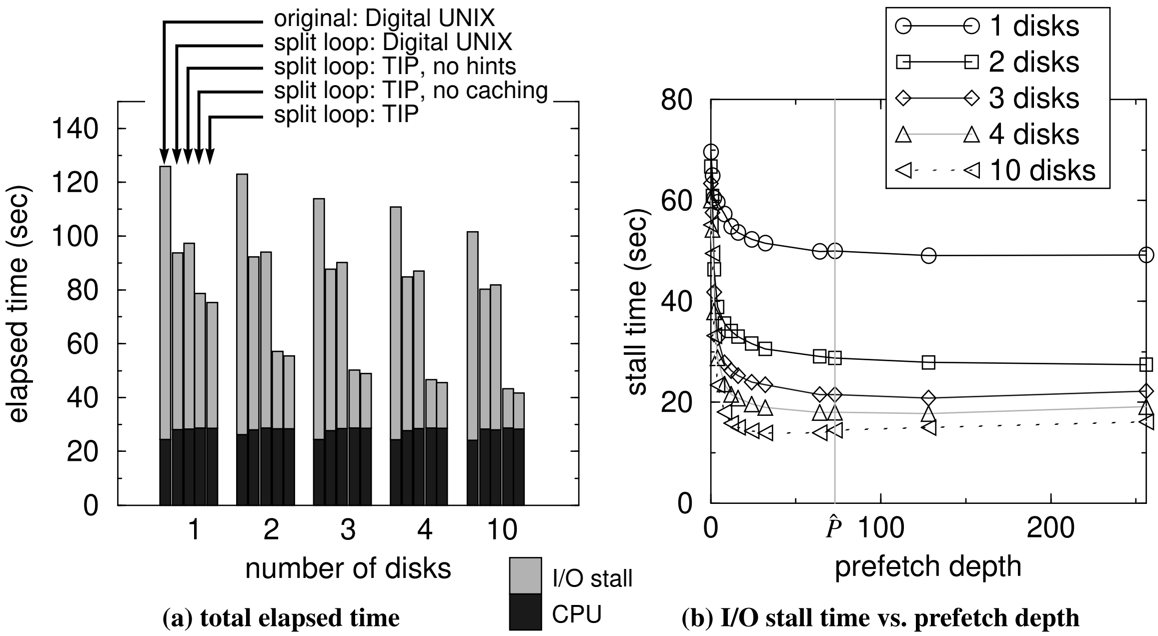


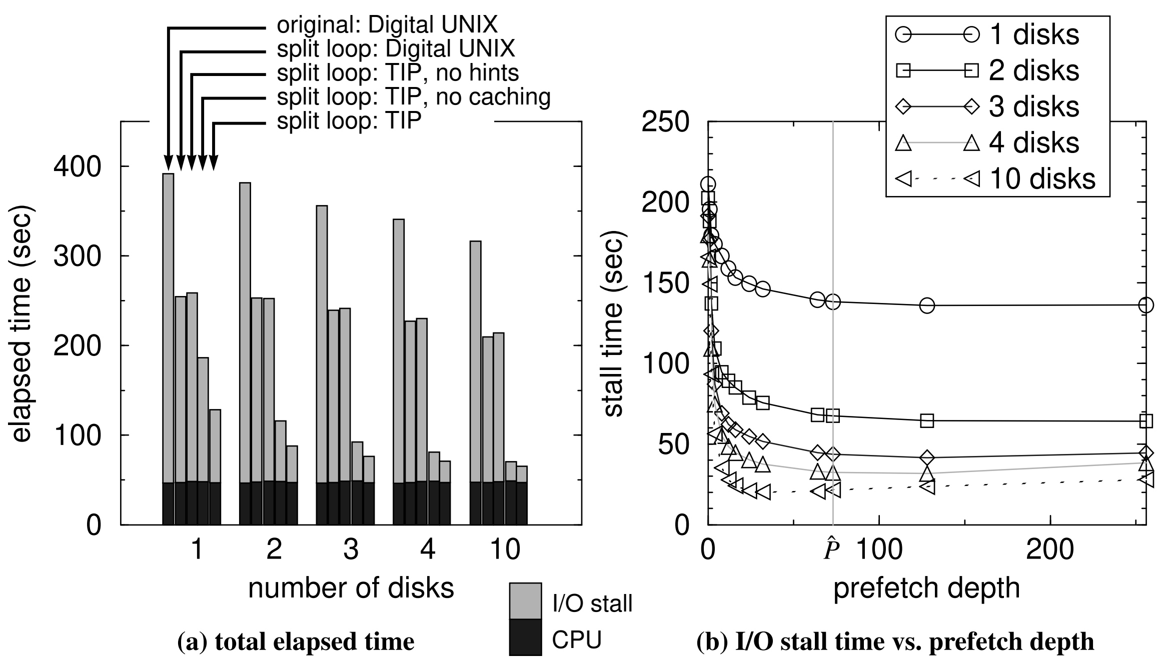
Figure 6.15. Postgres, 80% match, access pattern. The general access pattern for the 80% match case is identical to that of the 20% match case, except that because more outer-relation records have matches in the inner relation, it performs 15,674 hinted reads of matching records from the inner relation.

Postgres benefits little from Digital UNIX's sequential readahead. There are few sequential accesses apart from the 818 accesses to the outer relation. Thus, even though the 80%-match case has three times as many accesses as the 20%-match case, there are about the same number of prefetches in the two cases when Postgres does not give hints.



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.16. Postgres, 20% match, performance. In (a), the *original* runs show the performance before Postgres' loop is split to give hints. The *split loop* runs show the performance of the restructured Postgres running on the usual four configurations. The restructuring improves access locality and therefore cache performance, allowing it to run faster than the original Postgres even without hints. Informed prefetching further reduces I/O stall time. Graph (b) shows stall as a function of prefetch depth. On one disk, deep prefetching improves disk scheduling and reduces stall, but on ten disks, prefetching too deeply reduces cache effectiveness and increases stall. See text for details.



(a) total elapsed time **(b) I/O stall time vs. prefetch depth**

Figure 6.17. Postgres, 80% match, performance. Overall performance is very similar to the 20%-match case except that the larger number of matches leads to more random hinted reads of the matching inner relation tuples which lead to greater gains from informed prefetching. They also provide more opportunities for informed caching to reduce the number of blocks fetched from disk and for informed clustering to fetch adjacent blocks for widely separated accesses which improves access efficiency and reduces elapsed time by 31%.

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
original: Digital UNIX	24.50 (0.16)	101.45 (0.48)	26.24 (3.13)	96.78 (2.40)	24.41 (0.24)	89.55 (1.24)	24.30 (0.36)	86.57 (0.56)	24.13 (0.38)	77.45 (0.24)
split loop: Digital UNIX	28.08 (0.28)	65.77 (0.37)	28.06 (0.18)	64.18 (0.42)	27.72 (0.28)	60.04 (0.23)	27.71 (0.10)	57.09 (0.18)	28.29 (0.59)	51.93 (0.48)
split loop: TIP, no hints	28.35 (0.41)	69.00 (0.53)	28.70 (0.72)	65.37 (0.64)	28.50 (0.37)	61.69 (0.44)	28.45 (0.14)	58.56 (0.96)	28.03 (0.36)	53.84 (0.41)
split loop: TIP, no caching	28.65 (0.40)	50.01 (0.36)	28.42 (0.23)	28.78 (0.43)	28.75 (0.34)	21.50 (0.30)	28.66 (0.39)	18.05 (0.20)	28.78 (0.36)	14.44 (0.20)
split loop: TIP	28.60 (0.50)	46.72 (0.46)	28.40 (0.26)	27.08 (0.53)	28.60 (0.68)	20.38 (0.49)	28.58 (0.36)	17.06 (0.34)	28.34 (0.22)	13.37 (0.28)

Table 6.11. Postgres, 20% match, elapsed time. These are the data graphed in Figure 6.16a. The numbers in parentheses are the 95% confidence intervals for the averages of the five runs.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
original: Digital UNIX	11125 (47)	2395 (46)	724 (5)	174 (6)	5201 (49)	523 (6)	5401 (12)	5313 (13)
split loop: Digital UNIX	11254 (34)	2555 (35)	1121 (13)	233 (2)	6174 (31)	1039 (5)	4041 (11)	3961 (10)
split loop: TIP, no hints	10851 (24)	2151 (28)	1070 (16)	224 (4)	5744 (37)	901 (8)	4204 (13)	4139 (8)
split loop: TIP, no caching	10926 (31)	2096 (31)	4240 (20)	3340 (12)	5743 (20)	4074 (12)	1108 (6)	1042 (5)
split loop: TIP	10787 (20)	2082 (19)	4107 (17)	3086 (21)	5787 (17)	3934 (8)	1065 (4)	1002 (2)

Table 6.12. Postgres, 20% match, prefetching and caching performance. Splitting the loop adds 1000 reuse hits and decreases elapsed time even without hints. Informed caching and clustering reduce the number of prefetch I/Os by 250 which reduces stall by 7% on a single disk.

Because sequential readahead does not work for Postgres, there is little concurrency in Postgres' I/O workload and, without hints, Postgres is unable to take full advantage of the disk array. Indeed, what little advantage Postgres does find in the larger arrays is a result not of the disks working in parallel, but of the data being spread over less of an individual disk's surface which lowers the per-block access time from 16 msec on one disk to 13 msec on ten disk for the 80%-match case. We see again that disk arrays need parallel workloads to take advantage of the hardware parallelism they offer.

system	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
original: Digital UNIX	46.49 (0.49)	345.32 (2.27)	46.59 (0.60)	334.89 (1.63)	46.47 (0.49)	309.63 (1.56)	46.25 (1.06)	294.57 (1.01)	47.39 (4.68)	269.09 (3.81)
split loop: Digital UNIX	47.01 (0.35)	207.55 (0.21)	47.69 (1.44)	205.35 (0.38)	47.07 (0.31)	192.50 (1.06)	46.98 (0.30)	180.00 (0.71)	47.13 (0.82)	162.59 (0.67)
split loop: TIP, no hints	48.35 (0.72)	210.40 (0.91)	48.45 (0.45)	203.94 (1.62)	48.50 (0.68)	192.81 (2.15)	48.32 (0.57)	181.94 (2.66)	48.09 (0.68)	166.01 (0.67)
split loop: TIP, no caching	47.98 (0.40)	138.31 (0.42)	48.29 (0.37)	67.58 (0.48)	48.74 (0.32)	43.79 (0.21)	48.62 (0.32)	32.47 (0.15)	48.71 (0.69)	21.62 (0.31)
split loop: TIP	46.85 (0.25)	81.67 (1.88)	47.13 (0.42)	40.75 (1.19)	46.83 (0.42)	29.35 (1.18)	47.03 (0.40)	23.78 (0.30)	47.16 (0.88)	18.18 (0.27)

Table 6.13. Postgres, 80% match, elapsed time. These are the data graphed in Figure 6.17a. The numbers in parentheses are the 95% confidence intervals for the averages of the five runs.

system (1 disk)	requests		prefetches		cache			
	total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
original: Digital UNIX	35081 (59)	2422 (58)	792 (24)	196 (12)	16157 (86)	354 (14)	18570 (109)	18477 (108)
split loop: Digital UNIX	33820 (70)	2553 (71)	1174 (6)	248 (2)	20316 (64)	787 (3)	12715 (17)	12632 (17)
split loop: TIP, no hints	33788 (31)	2522 (26)	1188 (14)	256 (4)	20296 (97)	795 (8)	12696 (78)	12618 (78)
split loop: TIP, no caching	33921 (26)	2524 (26)	12593 (8)	11552 (9)	20263 (23)	12190 (9)	1467 (4)	1386 (4)
split loop: TIP	33787 (25)	2515 (25)	10300 (314)	5655 (304)	23377 (110)	9106 (112)	1303 (7)	1221 (6)

Table 6.14. Postgres, 80% match, prefetching and caching performance. Splitting the loop again increases reuse hits substantially. The impact of informed clustering is, again, to reduce reuse hits even while reducing elapsed time.

With informed prefetching, Postgres has I/O workload concurrency and takes advantage of the parallelism of the disk array to reduce elapsed time by up to 47% for the 20%-match case and up to 67% for the 80%-match case. Some stall remains because Postgres doesn't give hints for the index lookups. Nevertheless, because most of these lookups hit in the cache, informed prefetching eliminates up to 73% and 87% of the stall for the 20%- and 80%-match benchmarks of this I/O-bound application.

The results for stall as a function of prefetch depth shown in Figures 6.16b and 6.17b reveal an interesting effect on the ten-disk array: prefetching too deeply increases stall.

Using too many buffers for prefetching can reduce cache effectiveness and increase the number of cache misses. For example, for these experiments without informed caching, the number of cache misses for the 80%-match case increases from about 1280 at a prefetch depth of one to about 1880 at a prefetch depth of 256 (not shown). Although these are small numbers compared to the nearly 34,000 total requests, they each add about 12 msec of stall to the elapsed time of the benchmark on ten disks or a total of about 7 seconds.

On a single disk, more effective disk scheduling at the deeper prefetch depths reduces disk service time from almost 15 msec to under 12 msec per block for the 80%-match case. This reduction more than offsets the increase in stall from the larger number of misses. With a single disk, bandwidth is at premium, and the disk is the bottleneck on overall performance. Thus, the greatest gains come from maximizing disk performance. With ten disks, there is ample bandwidth and maximizing the performance of individual disks is less important; stall has already been masked, at least for hinted accesses. However, the unhinted misses cannot take advantage of array parallelism and therefore stall for the full latency of a disk access. The stall for these unhinted accesses determines the overall stall for the benchmark.

The upper-bound prefetch horizon, $\hat{P}$, strikes a good compromise in performance across array sizes. It obtains most of the benefit from improved disk scheduling on a single disk where it reduces total disk service time by 19% and 24% for the 20%- and 80%-match cases as will be seen in Table 6.15. On the other hand, stall time on ten disks is within, respectively, 4% and 6% of the minimum which occurs at a prefetch depth of 32 in both cases. The elapsed time when prefetching to a depth of $\hat{P}$=73 is, on one disk, within 1% of the elapsed time when prefetching to a depth of 256, and is, on ten disks, within 1% of the elapsed time when prefetching to a depth of 32.

Using the hints for informed caching and informed clustering in *TIP* reduces the elapsed time of the 20%- and 80%-match cases by 4% and 31% compared to *TIP, no caching* on a single disk. TIP is able to take advantage of the hints for the many random inner-relation reads to increase cache effectiveness and cluster together many widely separated accesses into a much smaller number of more sequential accesses. For the 80%-match case, informed caching in *TIP* increases the number of reuse hits by over 3000 or more than 15% compared to *TIP, no caching* as seen in Table 6.14. And, for the 80%-match case, informed clustering in *TIP* increases the average blocks per prefetch cluster from 1.09 in the *TIP, no caching* case to 1.82. Consequently, *TIP* needs 5900 fewer I/Os to prefetch only 2300 fewer blocks. These effects are more dramatic in the 80%- than the 20%- match case because the larger number of inner-relation accesses provides more opportunities for informed caching and clustering.

The impact on elapsed time is greatest on a single disk where bandwidth is most limited. As was the case for disk scheduling when prefetching without caching, optimizing disk performance is most important when the disk is the bottleneck. On ten disks, informed prefetching alone masks most of the stall for hinted accesses and there is little room for additional improvement. Nevertheless, the large reduction in the number of I/O accesses, especially in the 80%-match case, reduces disk-driver CPU overhead and therefore reduces CPU time in that case by about 1.5 seconds or a little over 3% on ten disks.

Unfortunately, informed clustering increases the number of prefetched blocks that are never accessed by about 800 to nearly 8% of the total prefetched in the 80%-match benchmark. There are two reasons why so many are ejected before they can be used. First, because TIP's local value estimates do not generate a full clustering and caching schedule, but instead build clusters opportunistically around prefetches that are about to occur, some blocks may be clustered that cannot be cached until they are accessed. For example, a prefetch may present the opportunity to cluster a prefetch for access number 2000 in the hinted access sequence. At a later time, an opportunity to cluster-prefetch for access number 1000 may arise. If the block for access 2000 is the least valuable, it will be ejected to cluster-prefetch the block for access 1000. A full schedule could have anticipated this and avoided cluster-prefetching block 2000 in the first place. Fortunately, cluster-prefetches are cheap, so the cost of ejecting some cluster-prefetched blocks is small compared to the benefit of the many successful cluster-prefetches.

The second reason clustered blocks are ejected is a result of a complex interaction between the hinted cache and the LRU cache. When the loop is split, Postgres must read the outer relation twice. The first time it reads the relation, it also performs index lookups which push the outer-relation blocks to the tail of the LRU queue. During this phase of the computation, there are few hits at the tail of the queue because the outer relation is being scanned sequentially, and so the tail of the LRU does not appear to be very valuable. Then, Postgres delivers thousands of hints for the reads of the inner-relation data blocks. TIP takes a few buffers from the tail of the LRU queue to begin prefetching these blocks, and more for clustered reads. As Postgres performs the join, it begins reaccessing the outer-relation tuples with unhinted reads while consuming some of the hinted inner-relation data blocks. This hint consumption leads to more prefetching and more informed clustering. Eventually, prefetched and clustered blocks completely displace the outer-relation blocks from the tail of the LRU queue. However, as Postgres continues to reaccess the outer-relation blocks, it scores hits on the ghost buffers at the tail of the LRU. Eventually, the tail of the LRU queue starts to look valuable and buffers for further prefetching and clustering come not from the LRU queue, but from the hinted cache which holds blocks for reuse and blocks that were prefetched as part of a cluster and are waiting to be accessed for the first time. But, because hinted blocks get put on the LRU queue after the hinted access occurs, the now-growing LRU queue saves some of these blocks from ejection. The LRU estimator does not save clustered blocks, however, and many of them are ejected, even if they will be accessed before a recently consumed hinted block will be reaccessed. Of course, growing the queue does not restore the data that the ghosts once held and the larger queue does not gain any additional cache hits. Fortunately, the outer relation is read sequentially, so heuristic readahead brings its blocks back into the cache without adding much stall.

This sequence of events highlights a weakness of using the LRU queue as a predictor of future behavior, especially at the boundary between phases in a computation when the characteristics of the workload are changing. It suggests that a workload that accesses its data exactly twice may be the worst-case scenario for the LRU estimator. Initially, there are no hits and the queue looks useless. Then there are ghost hits, and the queue grows. But, the data is never re-accessed a third time, so that larger queue does not actually increase the number of hits.

Despite these difficulties with the LRU estimator, informed caching and clustering delivers substantial gains for this application, including up to a 31% reduction in elapsed time. The primary goal of informed caching is to reduce the number of blocks requested from disk, and the primary goal of informed clustering is to reduce the disk service time per block by increasing disk workload sequentiality. Even with the ejection of some clustered prefetch blocks, informed caching in *TIP* reduces the number of blocks fetched from disk by 17% compared to *Tip, no caching* for the 80%-match case as shown in Table 6.15. And, informed clustering reduces the service time per block by 22%. So, not only does *TIP* perform fewer accesses, it services each more quickly. Clearly, these performance benefits far outweigh the cost of ejecting some of the clustered blocks before they are accessed.

6.3.7 The impact on disk service time

As just discussed in the context of the Postgres benchmark, TIP can improve disk performance through two mechanisms: more effective disk scheduling, and clustering prefetches. Table 6.15 summarizes the impact of these mechanisms on disk performance for all of the benchmarks on a single disk. I focus on single-disk performance because, as noted above, that is where the impact of improvements in disk performance on elapsed time is greatest.

To help clarify the individual impact of the two mechanisms, I consider the performance of three system configurations. *TIP, no caching (prefetch depth=0),* or simply *TIP, no caching(0)*, receives hints but does not use them for prefetching, although it does use them for clustering sequential accesses and avoiding false readahead. It shows performance when disk queues are short. I use it as the base case for comparison instead of *Digital UNIX* because it does not suffer from false readahead which can make accesses appear to be highly sequential, yet it does cluster sequential accesses which Digital UNIX's readahead does fairly well. Thus, it represents a sort of idealized base case. *TIP, no caching* shows the performance when hints are used to generate deep queues which can provide disk-scheduling opportunities, but hints are not used for caching or clustering except within the prefetch horizon. Thus, by comparing its disk performance with the *no caching(0)* case, it is possible to see the benefit of more effective disk scheduling that results from informed prefetching. Finally, *TIP* uses hints for prefetching, clustering and caching. Informed caching may reduce the number of blocks requested, but clustering is responsible for assembling these requests into larger clusters than sequential clustering alone can

benchmark	TIP, no caching (prefetch depth = 0)				TIP, no caching				TIP			
	total service time (sec)	I/Os	blocks	service time per block (ms)	total service time (sec)	I/Os	blocks	service time per block (ms)	total service time (sec)	I/Os	blocks	service time per block (ms)
Davidson	181.32 (0.85)	15980 (0)	125741 (0)	1.44 (0.01)	275.18 (0.27)	15993 (3)	125822 (3)	2.19 (0.00)	137.15 (2.93)	9053 (328)	50988 (1073)	2.69 (0.10)
XDataSlice	228.89 (0.20)	14889 (2)	45229 (5)	5.06 (0.00)	225.78 (0.11)	14893 (0)	45236 (0)	4.99 (0.00)	225.19 (0.15)	14880 (0)	45210 (1)	4.98 (0.00)
Sphinx	103.69 (0.36)	8406 (13)	27238 (37)	3.81 (0.01)	113.21 (0.08)	8069 (6)	27258 (15)	4.15 (0.00)	111.09 (0.39)	6815 (22)	27164 (93)	4.09 (0.01)
Agrep	27.97 (0.13)	1992 (6)	3155 (6)	8.87 (0.03)	25.00 (0.31)	1972 (20)	3135 (20)	7.97 (0.05)	25.02 (0.44)	1980 (28)	3143 (28)	7.96 (0.07)
Gnuld	80.70 (0.07)	5621 (4)	12669 (4)	6.37 (0.00)	78.56 (0.05)	5620 (6)	12673 (7)	6.20 (0.00)	73.38 (3.55)	5160 (310)	12111 (387)	6.06 (0.14)
Postgres, 20% match	69.91 (0.30)	4410 (17)	5375 (17)	13.01 (0.01)	57.03 (0.45)	4443 (21)	5441 (21)	10.48 (0.05)	53.88 (0.52)	4146 (29)	5261 (10)	10.24 (0.08)
Postgres, 80% match	213.17 (2.19)	13024 (143)	14445 (143)	14.76 (0.02)	162.43 (0.44)	13052 (15)	14516 (15)	11.19 (0.02)	104.62 (2.04)	6981 (294)	12050 (324)	8.69 (0.32)

Table 6.15. Summary of disk performance for the benchmarks running on a single disk. This table reports the impact of disk scheduling and request clustering on disk performance. The statistics were collected in the disk driver while running the benchmarks on three system configurations. *TIP, no caching (prefetch depth = 0)* or simply *TIP, no caching(0)* is the same configuration used to measure stall as a function of prefetch depth in the performance graphs for each benchmark with the prefetch depth set to 0. It shows disk performance when there are no deep request queues which can be used for disk scheduling. *TIP, no caching* is the familiar configuration that uses hints to prefetching out to the prefetch horizon. Comparison with the *no caching(0)* case, shows the impact of using long queues of prefetch requests for disk scheduling. Finally, *TIP* uses hints for prefetching, clustering and caching. Comparison with *TIP, no caching* is an indication of the impact of using large numbers of hints to cluster widely separated accesses. For each configuration, this table shows: the total time for the disk to service all accesses (in seconds); the number of distinct disk accesses; the number of 8-KByte blocks requested by those accesses; and the total service time divided by the number of blocks (in milliseconds).

achieve. By comparing *TIP*'s performance to *TIP, no caching*, one can see the benefit of using large numbers of hints to cluster widely separated accesses[1].

The first unexpected result is that service time for Davidson is higher when prefetching in *TIP, no caching* than it is in *TIP, no caching(0)*. Normally, I would expect the deeper queues in *TIP, no caching* to lead to a reduction in service time. Even though this is not the case here, the elapsed time does go down slightly (not shown). The service time is misleading in this case because of the disk's internal readahead. The wallclock time at which the sequential data requested by Davidson will be available is determined by the rotation of the disk under the read head. The service time in this case is just a measure of how far in advance of that time the request for the data is queued at the disk. A similar effect can be seen to a lesser degree for Sphinx.

The total service time for Davidson goes down substantially in *TIP* as informed caching reduces the number of blocks fetched from disk. However, the service time per block rises. This is because, through a dynamic of the implementation that I do not fully understand, Davidson does not end up with the single range of cached blocks suggested by Figure 4.7 for repeated sequential access, but instead ends up with one large range and many small sequential groups caching random blocks in the file[2]. Filling the gaps between these groups leads to disk requests that are non-sequential and may be smaller than a full cluster of 8 blocks. This experience suggests that informed caching might be improved if it were mindful of clusters when it estimates the cost of ejecting a block. It is cheaper, both in terms of disk driver overhead and disk access latency, to eject a block that could be prefetched as part of a cluster than to eject one that would require a separate access. Incorporating such cost estimates into the informed caching estimates could be an interesting area for future research.

---

[1] It must be noted that there is ambiguity in this comparison because informed caching may change the set of blocks fetched from disk and therefore the sequentiality of the disk workload before clustering. But, reducing the number of blocks read from disk most likely reduces, not increases the opportunities for sequential clustering. Thus, any increase in cluster size and reduction in service time per block for *TIP* compared to *TIP, no caching* is almost certainly the result of informed clustering. Thus, if this comparison does not provide definitive evidence, it does provide highly suggestive evidence of the impact of informed clustering.

[2] My hunch is that, periodically, when a batch of buffers is moved from the annex to the LRU queue, some buffers suddenly become available to cache whatever blocks were last accessed by Davidson. As the wave of prefetching and MRU replacement moves on, these blocks remain cached.

XDataSlice takes care to issue its requests in ascending order in the file, so there is little opportunity either for disk scheduling or informed clustering for this application.

Sphinx, during its long initialization phase which includes more than 80% of its accesses, issues requests in ascending order. Many of these requests benefit from sequential readahead and so, like Davidson, the service time in *TIP, no caching* is higher than in *TIP, no caching(0)*. Clustering, even just within the prefetch horizon in *TIP, no caching,* reduces the number of disk accesses by 340. Informed clustering in *TIP* eliminates more than 1100 more accesses, and, compared to *TIP, no caching*, reduces service time per block from 4.15 msec to 4.09 msec. One might expect a larger reduction given the many random accesses seen in Figure 6.8, but there are two reasons why this is not the case. First, the large number of nearly sequential accesses during the initialization phase do not provide much opportunity for substantial improvement and this brings the average down. Second, during the recognition phase when accesses are more random, hints are given in small bursts (see Figure 3.6). There is no opportunity to cluster across these bursts so the clustering opportunity is not as great as it appears. Informed clustering needs hints about many future accesses to be most effective.

Agrep obtains a 10% reduction in disk service time for disk scheduling because informed prefetching can sort requests across multiple files. But, because clustering only happens within files as explained in Section 5.2.2, and because Agrep reads files sequentially, even *TIP, no caching(0),* which only clusters for sequential accesses, builds all the clusters possible and there is no additional benefit from using more hints for informed clustering.

Gnuld derives a few percent benefit from better request sorting in *TIP, no caching* compared to *TIP, no caching(0)*. Gnuld also derives a small additional benefit from clustering a few access from one pass over its input files with accesses to contiguous blocks in subsequent passes.

Finally, as discussed at length in the previous section, Postgres' many random reads benefit most of all the applications from both disk scheduling and informed clustering. In the 20%-match case, disk scheduling in *TIP, no caching* reduces the service time per block by 19% and informed clustering in *TIP* further reduces it by another couple of percent. In the 80%-match case, disk scheduling reduces average service time 24% and informed clustering reduces it by another 22%.

Overall, most of the benchmark applications were able to organize their file requests into ascending order to start with. Consequently, for most of the benchmarks, gains from both scheduling and informed clustering are only a few percent. However, when applications perform reads that are scattered randomly as is the case for Postgres, disk scheduling can reduce average disk service time by up to 24%. Further, if applications can provide hints about many accesses, as Postgres can, informed clustering can provide up to a 22% reduction in service time.

6.4 Multiple-process results

Multiprogramming I/O-intensive applications does not generally lead to efficient use of resources because these programs eject each other's blocks from the cache and interpose disk accesses which disturbs each other's disk access sequentiality. However, it is inevitable that I/O-intensive programs will be multiprogrammed. In this section, I present the implications of informed prefetching and caching on multiprogrammed I/O-intensive applications.

When multiple applications are running concurrently, the informed prefetching and caching system should exhibit three basic properties. First and foremost, hints should increase overall throughput. Second, an application that gives hints should improve its own performance. Third, non-hinting applications should not suffer unfairly when a competing application gives hints. This last is a bit vague; what does it mean not to suffer unfairly?

In my view, it does not mean that a non-hinter should not suffer at all when another application hints. All applications suffer when forced to share a machine and the extent to which they suffer depends on how the other applications use the machine. Without hints, an I/O-intensive application may be blocked on the disk so often that it interferes little with a CPU-intensive application. When the I/O-intensive application gives hints, it may stall less on the disk, use more of the CPU, and slow down the CPU-intensive application. But, the same thing would have happened if the CPU-intensive application had been forced to share the machine with another CPU-intensive application in the first place.

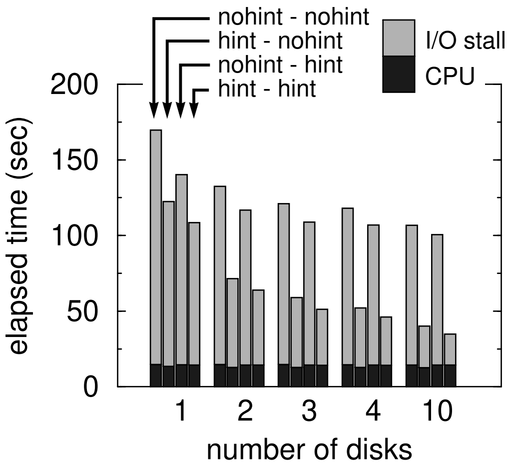
When an application gives hints, it changes the way it uses a machine's resources, but that does not mean it uses them unfairly.

The previous argument only asserts slowing down a competing application is not necessarily unfair; it does not define fairness. Fairness is a deep issue, especially when multiple resources are involved and a 'fair' allocation of the disk, for example, may lead to an 'unfair' allocation of the CPU. I do not attempt here to find answers to these hard questions. For the purposes of this dissertation, I take it to be fair for an application to suffer so long as overall throughput increases. That is, it would be unfair for an application to suffer so much that overall throughput suffers. From this perspective, the third desired property listed above is really just another facet of the first property: that hints should increase overall throughput. The cost-benefit model attempts to reduce the sum of the I/O overhead and stall time for all executing applications, and thus, I expect the resource management algorithms to benefit multiprogrammed workloads and have the desired properties.

To explore how well TIP meets these performance expectations, I report three pairs of application executions: Gnuld/Agrep, Sphinx/Davidson, and XDataSlice/Postgres. Here, Postgres performs the join with 80% matches and precomputes its data accesses even when it does not give hints. For each pair of applications, I ran all four hinting and non-hinting combinations on TIP starting the two applications simultaneously with a cold cache and measuring the elapsed time of each. Figures 6.18 through 6.20 show the results for Gnuld/Agrep, and Figures 6.21 through 6.23 show the results for Sphinx/Davidson and XDataSlice/Postgres.

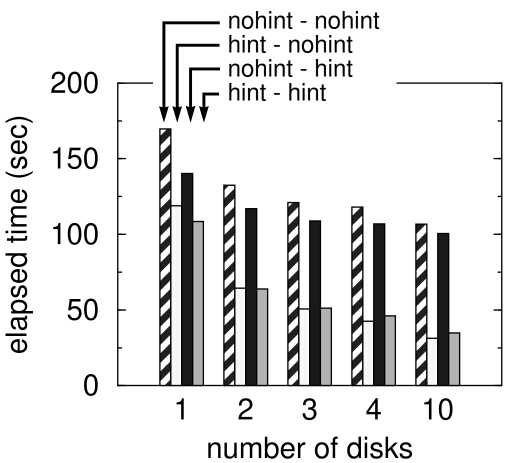
In both sets of figures, the upper graphs (Figures 6.18 and 6.21) show the impact of hints on throughput for the three pairs of applications. Tables 6.16 through 6.18 present these same results in tabular format. I report the time until both applications complete, broken down by total CPU time and simultaneous stall time during which the CPU is idle. In all cases, the maximum elapsed time decreases when one application gives hints, and decreases further still when both applications give hints. Simultaneous I/O stall time is virtually eliminated for two out of the three pairs when both applications give hints and the parallelism of 10 disks is available.

The middle and lower graphs in the two sets of figures show the elapsed time for individual applications when paired with another application (whose name is in parentheses).



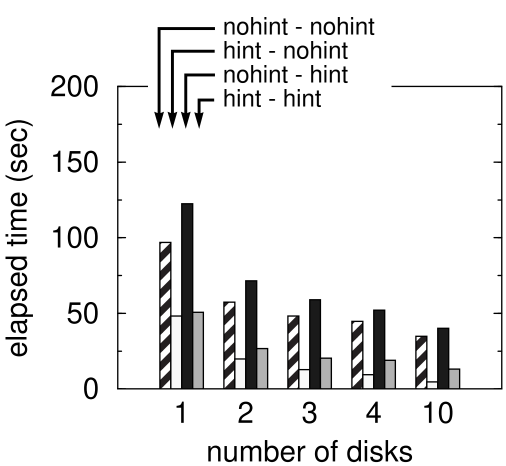
Gnuld and Agrep

Figure 6.18. Elapsed time for both Gnuld and Agrep to complete. The pair of workloads are run concurrently on *TIP* and the elapsed time of the last to complete is reported along with the total CPU busy time. For each number of disks, four bars are shown. These represent the four hint/nohint combinations. For example, the second bar from the left in any quartet of bars is Gnuld hinting and Agrep not hinting.



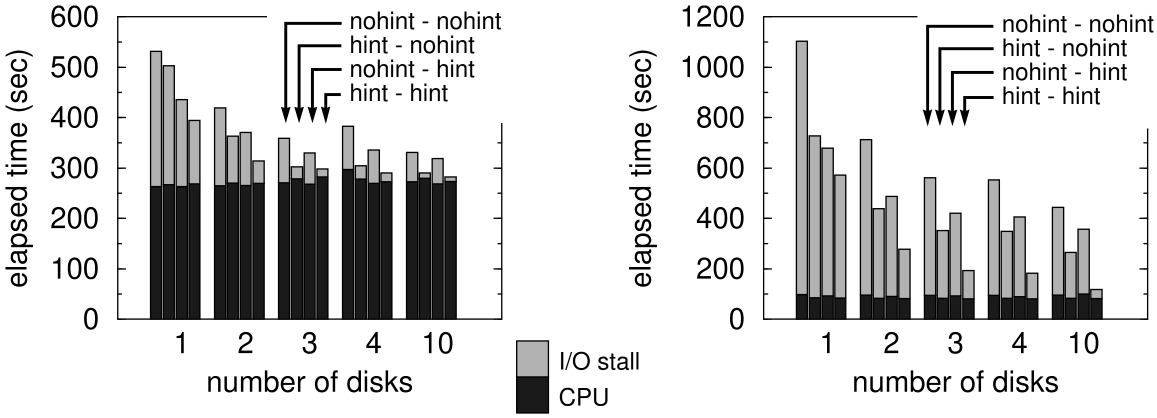
Gnuld (with Agrep)

Figure 6.19. Elapsed time for Gnuld when run with Agrep. These figures report data taken from the same runs on *TIP* as reported in Figure 6.18. However, the elapsed time shown represents only Gnuld's execution. The hint/nohint combinations are identical to Figure 6.18. Compare bars one and two or three and four to see the impact of giving hints when Agrep is respectively non-hinting or hinting. Compare bars one and three or two and four to see the impact of Agrep giving hints.



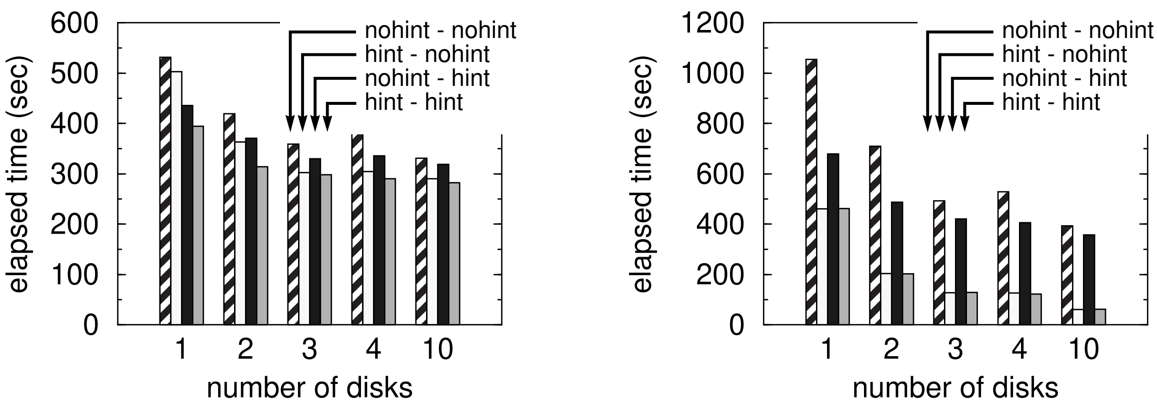
Agrep (with Gnuld)

Figure 6.20. Elapsed time for Agrep when run with Gnuld. These figures report data from the same set of runs as reported in Figures 6.18 and 6.19. However, the inner two bars are swapped relative to the inner two bars of the other figures. For example, the second bar from the left in any quartet is Gnuld not hinting and Agrep hinting. Compare bars one and two or three and four to see the impact of giving hints when Gnuld is respectively non-hinting or hinting. Compare bars one and three or two and four to see the impact of Gnuld giving hints.



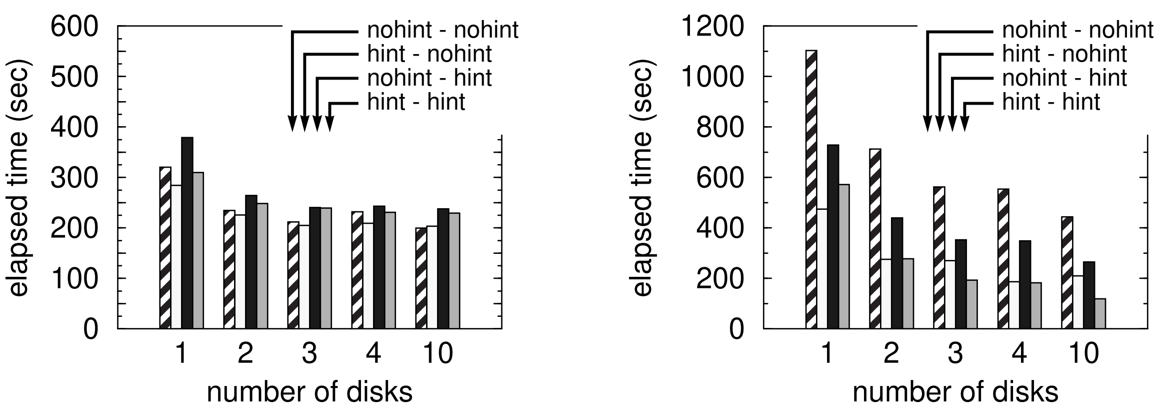
(a) Sphinx and Davidson **(b) XDataSlice and Postgres,80% match**

Figure 6.21. Elapsed time for both applications to compete. In a format identical to that of Figure 6.18, this figure shows the elapsed time for both of a pair of applications to complete. Results for Sphinx and Davidson running together are on the left, results for XDataSlice and Postgres, 80% match are on the right.



(a) Sphinx (with Davidson) **(b) XDataSlice (with Postgres, 80% match)**

Figure 6.22. Elapsed time for one of a pair of applications. These figures report data taken from the same runs as reported in Figure 6.21. However, in a format identical to that of Figure 6.19, the times shown are for only one of a pair of applications running. Sphinx is on the left and XDataSlice is on the right.



(a) Davidson (with Sphinx) **(b) Postgres, 80% match with XDataSlice**

Figure 6.23. Elapsed time for the other of a pair of applications. These figures report data from the same runs as Figures 6.21 and 6.22. However, the inner two bars of each quartet are swapped relative to the inner two bars of the other figures. Thus, in a format identical to Figure 6.20, they report the time for the other of a pair of applications. Davidson is on the left and Postgres, 80% match is on the right.

Gnuld - Agrep	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
no hint - no hint	14.72 (0.14)	154.96 (7.46)	14.61 (0.03)	117.86 (2.83)	14.61 (0.23)	106.40 (2.56)	14.52 (0.16)	103.52 (1.29)	14.42 (0.09)	92.27 (0.69)
hint - no hint	13.52 (0.22)	108.99 (3.75)	12.72 (0.14)	58.85 (1.49)	12.75 (0.14)	46.19 (0.83)	12.76 (0.10)	39.35 (1.11)	12.62 (0.04)	27.41 (0.60)
no hint - hint	14.54 (0.16)	125.77 (0.21)	14.37 (0.04)	102.46 (1.70)	14.37 (0.07)	94.47 (0.49)	14.43 (0.22)	92.41 (1.76)	14.39 (0.06)	86.12 (2.58)
hint - hint	14.30 (0.23)	94.22 (2.54)	14.23 (0.16)	49.64 (0.48)	14.19 (0.07)	36.92 (0.77)	14.22 (0.08)	31.79 (0.15)	14.42 (0.16)	20.50 (0.33)

Table 6.16. Elapsed time for both Gnuld and Agrep to complete. This table gives the complete data for Figure 6.18.

Sphinx - Davidson	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
no hint - no hint	263.11 (1.25)	268.61 (1.43)	265.03 (0.52)	154.08 (1.65)	270.54 (0.67)	88.48 (1.40)	296.91 (22.8)	85.72 (9.27)	272.74 (1.54)	58.27 (0.38)
hint - no hint	267.37 (0.52)	235.56 (1.34)	270.34 (1.45)	93.01 (3.33)	278.48 (1.46)	23.62 (1.13)	278.17 (1.04)	26.42 (1.43)	279.69 (1.37)	10.38 (0.36)
no hint - hint	263.03 (0.94)	172.55 (6.14)	265.32 (0.86)	105.52 (10.4)	267.98 (2.61)	62.25 (3.18)	269.69 (4.98)	66.02 (1.56)	268.56 (0.67)	50.49 (1.08)
hint - hint	268.70 (1.16)	125.76 (2.66)	270.06 (0.69)	44.20 (1.39)	282.23 (18.3)	16.28 (2.10)	272.97 (1.52)	17.21 (0.27)	273.21 (1.85)	8.73 (0.13)

Table 6.17. Elapsed time for both Sphinx and Davidson to complete. The data in this table corresponds to the left-hand graph in Figure 6.21.

XDataSlice - Postgres, 80%	1 disk		2 disks		3 disks		4 disks		10 disks	
	CPU	stall	CPU	stall	CPU	stall	CPU	stall	CPU	stall
no hint - no hint	97.62 (0.89)	1005.1 (2.12)	95.74 (0.39)	616.98 (4.39)	94.65 (0.77)	466.93 (1.99)	95.18 (0.73)	458.29 (2.70)	95.84 (1.36)	347.98 (5.05)
hint - no hint	86.03 (0.79)	642.12 (3.32)	82.76 (1.01)	356.67 (1.28)	82.37 (0.37)	270.33 (1.43)	83.23 (1.91)	265.44 (8.15)	82.88 (0.27)	182.24 (4.26)
no hint - hint	92.21 (0.95)	587.22 (12.6)	91.14 (0.34)	396.06 (2.93)	92.32 (0.41)	328.67 (3.07)	89.90 (1.31)	316.84 (14.1)	100.19 (8.61)	256.40 (4.65)
hint - hint	84.34 (0.72)	487.52 (3.98)	82.16 (1.40)	195.33 (2.04)	81.10 (0.68)	112.36 (1.10)	81.46 (0.50)	100.81 (6.70)	81.91 (0.47)	36.45 (5.15)

Table 6.18. Elapsed time for both XDataSlice and Postgres, 80% match, to complete. This is the complete data for the right-hand graph in Figure 6.21.

Although vertical columns of graphs in the figures correspond to the same test runs, the middle two bars in any quartet of the lower figures (Figures 6.20 and 6.23) are swapped relative to the middle two bars in the corresponding quartets of the middle and upper graphs. So, for example, in Figures 6.18 and 6.19, ‘hint-nohint’ means Gnuld hints while Agrep does not, whereas in Figure 6.20 ‘hint-nohint’ means Agrep hints while Gnuld does not. Tables 6.19 through 6.24 show the prefetching and caching performance for the individual applications in each pair of experiments.

A traditional goal of multiprogramming is to increase CPU utilization. These results show that multiprogramming I/O-intensive workloads has the opposite effect when applications must share a single disk. For example, running Gnuld and Agrep together instead of serially on a single disk reduces CPU utilization by 21% from 11% to 8.7% because the pair of applications uses the disk much less efficiently than do either of the applications when running alone. Both applications read multiple files from any one directory. Because Digital UNIX’s UFS file system tries to store individual files sequentially on the disk and to store multiple files from the same directory near each other on the disk surface, the disk workloads of both Gnuld and Agrep utilize the disk read head efficiently or, in the terminology of Chapter 2, have high sequentiality. But, because they read from different directories, when the applications are run together on a single disk, their accesses are interleaved and the workload sequentiality is greatly reduced. This increases average disk service time from 6.9 msec to 9.3 msec. Because the single disk is the bottleneck, this increase in service time leads to an increase in elapsed time and the reduction in CPU utilization noted above.

Sharing a single disk between XDataSlice and Postgres reduces disk workload sequentiality just as it did for Gnuld and Agrep. The problem is particularly acute for XDataSlice’s many false readaheads. When running alone, the false readaheads are at least sequential. But, when interleaved with Postgres’ accesses, these readaheads require a long expensive seek. Further exacerbating the problem is that when Postgres must share the cache with XDataSlice, its reuse hits drop from about 20,000 to about 10,000 as seen in the ‘no hint - no hint,’ single-disk row of Table 6.24. This reduction in cache effectiveness translates into an increase in disk load. Together these effects reduce CPU utilization by 41% from 15% to 8.9%.

Gnuld (with Agrep)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	23197 (13.7)	766 (13.7)	5747 (613)	2673 (317)	11563 (14.6)	4593 (106)	7040 (104)	5246 (96.0)
	10	23201 (7.55)	770 (7.55)	5728 (634)	2633 (319)	11581 (16.4)	4565 (103)	7054 (92.3)	5210 (89.3)
hint - no hint	1	23286 (7.55)	854 (7.55)	10427 (199)	4302 (80.2)	12709 (168)	10296 (173)	280 (6.57)	240 (1.82)
	10	23296 (6.17)	864 (6.17)	10310 (151)	4189 (116)	12847 (84.9)	10174 (82.5)	274 (6.87)	235 (3.31)
no hint - hint	1	23201 (12.3)	770 (12.3)	5512 (6.96)	2549 (3.19)	11608 (15.6)	4557 (5.14)	7036 (6.99)	5240 (5.18)
	10	23273 (18.5)	842 (18.5)	5496 (8.92)	2515 (4.70)	11667 (16.1)	4538 (8.00)	7068 (9.80)	5216 (10.0)
hint - hint	1	23253 (45.3)	821 (45.3)	10499 (276)	4331 (197)	12534 (201)	10447 (206)	271 (5.55)	240 (3.11)
	10	23248 (11.5)	816 (11.5)	10451 (123)	4145 (73.0)	12587 (84.1)	10399 (83.7)	261 (12.7)	231 (5.86)

Table 6.19. Gnuld prefetching and caching performance when run with Agrep. This table shows results for the Gnuld runs in Figure 6.19 on arrays of one and ten disks. The shaded rows correspond to the shaded rows in Table 6.20 below.

Agrep (with Gnuld)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	3041 (0.00)	113 (0.00)	1020 (367)	573 (268)	96 (0.00)	1013 (367)	1932 (367)	1926 (367)
	10	3043 (6.17)	115 (6.17)	1019 (367)	513 (257)	98 (5.94)	1012 (367)	1932 (367)	1926 (367)
hint - no hint	1	3036 (12.3)	108 (12.3)	2932 (5.29)	1764 (1.74)	94 (6.50)	2929 (5.14)	13 (7.55)	6 (3.08)
	10	3050 (6.17)	122 (6.17)	2931 (0.00)	1600 (0.00)	104 (5.65)	2927 (0.00)	18 (0.51)	8 (0.51)
no hint - hint	1	3037 (0.00)	109 (0.00)	868 (8.74)	465 (7.71)	98 (0.00)	860 (8.74)	2078 (8.74)	2077 (8.74)
	10	3037 (0.00)	109 (0.00)	863 (0.00)	406 (0.00)	98 (0.00)	855 (0.00)	2084 (0.00)	2083 (0.00)
hint - hint	1	3036 (12.3)	108 (12.3)	2930 (0.51)	1763 (0.51)	93 (6.17)	2927 (0.00)	15 (6.17)	6 (3.08)
	10	3041 (0.00)	113 (0.00)	2931 (1.03)	1600 (0.00)	96 (1.54)	2928 (3.08)	16 (4.63)	7 (2.06)

Table 6.20. Agrep prefetching and caching performance when run with Gnuld. This table shows results for the Agrep runs in Figure 6.20 on arrays of one and ten disks. Just as the middle two bars were reversed in that figure, so the middle pairs of rows are reversed relative to Table 6.19 above.

Sphinx (with Davidson)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	78745 (16.6)	1010 (16.3)	21260 (31.4)	4384 (8.84)	50800 (160)	17702 (27.0)	10243 (165)	4765 (53.6)
	10	78874 (15.1)	1142 (15.1)	21315 (24.9)	4359 (10.8)	50417 (43.7)	17704 (40.8)	10752 (21.7)	4982 (17.4)
hint - no hint	1	78630 (25.5)	1127 (25.5)	27256 (28.5)	6936 (94.9)	51175 (37.0)	26476 (120)	978 (125)	890 (126)
	10	78596 (22.6)	1093 (22.6)	28090 (41.8)	6955 (73.6)	50331 (50.6)	27521 (94.6)	744 (89.1)	666 (88.0)
no hint - hint	1	78874 (89.6)	1142 (89.6)	21294 (40.0)	4413 (11.2)	50603 (67.7)	17692 (30.3)	10578 (56.8)	4909 (32.2)
	10	78889 (14.8)	1157 (14.8)	21366 (53.7)	4382 (10.2)	50172 (58.7)	17629 (56.0)	11087 (63.4)	5189 (50.9)
hint - hint	1	78562 (68.9)	1059 (68.9)	27898 (108)	6931 (93.1)	50481 (81.8)	27386 (162)	694 (124)	609 (122)
	10	78595 (21.2)	1092 (21.2)	28321 (106)	7025 (72.2)	50165 (120)	27671 (149)	758 (54.6)	680 (58.2)

Table 6.21. Sphinx prefetching and caching performance when run with Davidson. This table shows results for the Sphinx runs in Figure 6.22a on arrays of one and ten disks. The shaded rows correspond to the shaded rows in Table 6.22 below.

Davidson (with Sphinx)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	147029 (0.63)	1128 (0.00)	124859 (5.65)	15977 (1.03)	21368 (0.63)	124828 (4.63)	832 (4.63)	254 (3.08)
	10	147563 (93.1)	1663 (93.1)	124859 (14.1)	16083 (7.55)	21796 (74.7)	124797 (12.2)	970 (9.06)	336 (6.24)
hint - no hint	1	147397 (2.86)	1495 (1.99)	64212 (537)	10185 (232)	83115 (526)	64073 (532)	208 (28.3)	136 (29.4)
	10	147446 (172)	1544 (172)	61913 (845)	10146 (488)	85442 (1010)	61794 (852)	209 (115)	166 (115)
no hint - hint	1	147330 (72.7)	1430 (72.5)	124964 (27.4)	16017 (10.0)	21566 (66.5)	124897 (15.7)	867 (8.43)	277 (3.49)
	10	147568 (87.9)	1668 (87.9)	124862 (16.6)	16085 (6.21)	21799 (74.1)	124796 (12.6)	972 (4.03)	336 (3.00)
hint - hint	1	147299 (153.	1397 (153)	61536 (1990)	9770 (90.3)	85698 (1890)	61414 (1950)	187 (40.8)	137 (30.7)
	10	147527 (142)	1625 (142)	60224 (571)	9576 (362)	87220 (627)	60143 (586)	163 (6.83)	118 (7.24)

Table 6.22. Davidson prefetching and caching performance when run with Sphinx. This table shows results for the Davidson runs in Figure 6.23a on arrays of one and ten disks. Just as the middle two bars were reversed in that figure, so the middle pairs of rows are reversed relative to Table 6.21 above.

XDataSlice (with Postgres)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	48467 (41.6)	2105 (41.6)	60815 (10.3)	22722 (2.97)	2836 (32.9)	25417 (8.26)	20213 (6.97)	20160 (3.15)
	10	48322 (326)	1961 (326)	60873 (53.8)	22691 (19.6)	2705 (290)	25412 (38.2)	20205 (25.0)	20155 (16.9)
hint - no hint	1	48253 (82.3)	1892 (82.3)	45287 (32.9)	14862 (15.7)	2794 (50.9)	45272 (31.3)	186 (14.2)	142 (11.9)
	10	48191 (48.3)	1830 (48.3)	44965 (19.8)	14905 (14.5)	3061 (42.2)	44951 (18.9)	179 (2.44)	136 (2.38)
no hint - hint	1	48770 (204)	2409 (204)	60860 (46.7)	22738 (18.1)	3068 (155)	25460 (29.9)	20241 (20.1)	20178 (14.2)
	10	48598 (37.3)	2228 (38.0)	60937 (31.6)	22715 (10.5)	2889 (65.2)	25465 (22.9)	20243 (8.85)	20181 (7.06)
hint - hint	1	48251 (54.3)	1890 (54.3)	45323 (45.6)	14878 (13.2)	2758 (48.0)	45309 (43.7)	184 (12.3)	140 (10.3)
	10	48211 (40.7)	1850 (40.7)	44986 (11.4)	14909 (21.3)	3057 (37.9)	44973 (12.2)	180 (4.81)	137 (4.19)

Table 6.23. XDataSlice prefetching and caching performance when run with Postgres, 80% match. This table shows results for the XDataSlice runs in Figure 6.22b on arrays of one and ten disks. The shaded rows correspond to the shaded rows in Table 6.24 below.

Postgres (with XDataSlice)	disks	requests		prefetches		cache			
		total	faults	blocks	I/Os	reuse hits	prefetch hits	misses	miss I/Os
no hint - no hint	1	33697 (44.6)	2430 (44.2)	1173 (14.9)	253 (9.36)	10440 (87.6)	369 (9.81)	22887 (88.1)	22791 (86.4)
	10	33702 (29.0)	2435 (30.7)	1308 (10.8)	271 (7.68)	10140 (385)	429 (26.9)	23132 (404)	23039 (403)
hint - no hint	1	33719 (8.11)	2449 (6.17)	10072 (157)	6139 (98.3)	17018 (344)	9305 (97.7)	7395 (247)	7299 (246)
	10	33654 (105)	2384 (106)	11019 (751)	6206 (573)	15082 (738)	9772 (560)	8800 (470)	8721 (468)
no hint - hint	1	33750 (7.33)	2475 (7.55)	1133 (1.74)	248 (1.26)	12582 (121)	421 (5.83)	20747 (115)	20647 (115)
	10	33732 (52.3)	2457 (52.8)	1172 (6.05)	244 (3.51)	18168 (341)	679 (4.22)	14885 (360)	14786 (367)
hint - hint	1	33727 (21.3)	2449 (22.6)	10117 (84.3)	6010 (339)	15207 (192)	9261 (153)	9258 (100)	9156 (97.7)
	10	33764 (39.7)	2486 (38.2)	10619 (283)	5616 (200)	21298 (407)	9040 (107)	3425 (414)	3334 (424)

Table 6.24. Postgres, 80% match, prefetching and caching performance when run with XDataSlice. This table shows selected results for the Postgres runs in Figure 6.23b. Just as the middle two bars were reversed in that figure, so the middle pairs of rows are reversed relative to Table 6.23 above.

Sphinx and Davidson are well-behaved when they run together on a single disk and CPU utilization is unchanged. Neither of them use much cache when not hinting, so sharing the cache does not cause extra misses. And, both of them do a fair number of multi-block sequential accesses which get clustered into large disk accesses. These large accesses are fairly efficient so that interleaving them only increases aggregate disk service time by 7%. The two applications do enough computation for multiprogramming to mask most of this modest increase in disk service time, and CPU utilization remains roughly constant at about 50%.

In contrast to the single-disk performance, multiprogramming on disk arrays can not only increase processor utilization, but also expose I/O concurrency. Qualitative examination of the ‘no hint - no hint’ bars in the graphs shows that, in contrast to single-application performance, larger arrays reduce elapsed time for multiprogrammed, I/O-intensive applications. Quantitatively, on a three-disk array and for all three pairs of applications, the CPU utilization when multiprogramming is within 5% of the utilization when executing the applications serially. A ten-disk array turns Gnuld/Agrep’s 21% reduction in CPU utilization into a 17% increase and XDataSlice/Postgres’ 41% reduction into a 21% increase. In an age when processor cost is declining rapidly, the greater CPU utilization when multiprogramming these applications compared to their serial execution is not a compelling argument for multiprogramming. However, for a machine that is multiprogrammed, these results do provide compelling evidence of the benefit of using a disk array.

The real goal of these experiments, though, is to understand how informed prefetching and caching behaves in a multiprogrammed environment. To see the impact of giving hints on an individual application’s elapsed time when a second non-hinting application is run concurrently, compare bars one and two in Figures 6.19/6.20 and 6.22/6.23. Compare bars three and four to see the impact when the second application is giving hints.

In most cases, giving hints substantially reduces an application’s elapsed time. In fact, the reductions on one disk are generally greater than when the applications are running by themselves. By queuing multiple requests, prefetching allows better request scheduling which compensates for the loss of disk workload sequentiality that results when the accesses of two applications are interleaved. An exception is Davidson when running with

Sphinx as shown on the left in Figure 6.23. Without hints, Davidson's aggressive non-hinting readahead lets it monopolize a single disk and Sphinx spends a lot of time waiting for Davidson's accesses to complete. When Davidson gives hints, informed caching increases reuse hits which reduces the load on the disk and Sphinx's I/Os complete more quickly leading Sphinx to demand more of the CPU. This reduces the benefit Davidson sees from its hints when it is multiprogrammed with Sphinx. Effectively, Davidson ends up sharing some of the benefit of its hinting with Sphinx.

This brings us to the effect of hints on other applications running in the system. To see the impact on a non-hinting application of another application giving hints, compare the first and third bars in Figures 6.22 and 6.23. Comparing the second to fourth bars shows the impact on a hinting application. As described above, and is clear from Figure 6.22, Sphinx derives substantial benefit from Davidson's hints. On the other hand, when Sphinx hints, it is able to compete more effectively against Davidson's readaheads. Davidson no longer dominates resource usage and consequently Davidson slows down when Sphinx hints. A reverse effect befalls Agrep when Gnuld hints. In that case, when Gnuld hints, it becomes an aggressive user of disk resources which delays Agrep. But, in most cases, the non-hinting application benefits from the hinter's more efficient usage of resources which leaves more resources for the non-hinter. Non-hinters may also benefit simply because the hinter completes more quickly and relinquishes resources. Postgres, for example, benefits when XDataSlice completes and leaves it the cache buffers for its index lookups as is evident from the large number of reuse hints in the last two 10-disk rows of Table 6.24 which show performance when XDataSlice is hinting.

Stepping back from the details of the dynamics of these pairs of applications, the overall conclusion is that when one application hints, throughput increases. And when the second application also hints, throughput increases further. When an application hints, it may become a more aggressive consumer of system resources at the expense of competing applications. However, as the analysis of the performance when neither application hints showed, applications suffer when they must share resources, and how much they suffer depends on which other application they must share with. But, because the TIP informed prefetching and caching system allocates resources to reduce overall I/O service time, it only takes a resource from one application and allocates it to another if the second application will make better use of the resource. The first application may suffer, but overall throughput increases. As these experiments show, TIP does take advantage of hints to reduce I/O service time and improve overall performance. Thus, TIP achieves its stated goals: for a single application, TIP reduces elapsed time; and when multiprogramming multiple applications, TIP increases throughput.

6.5 Lessons from prefetching and caching experiments

In this section, I distill the experience gained from the experiments into a number of general lessons about informed prefetching and caching and about the performance of I/O-intensive applications.

1. *Serial workloads need prefetching to take advantage of array parallelism.* This insight was one of the original motivations for this work. It is important enough to restate here and observe that, without informed prefetching, five of the six benchmark applications studied in this dissertation derive little benefit from even a ten-disk array as shown way back in Figure 2.2. Sequential readahead is able to take advantage of parallel transfer from an array for Davidson's large sequential accesses. But, without some form of effective prefetching, disk arrays do not significantly reduce elapsed time for applications with serial workloads.

2. *Informed prefetching obtains its greatest performance gains from prefetching in parallel, not from overlapping I/O and CPU.* Prefetching is most commonly thought of as a technique for overlapping I/O and CPU. But, by far the greatest reduction in elapsed time comes when TIP takes advantage of an array to prefetch in parallel for a serial workload. From Table 6.25, prefetching to overlap I/O and computation for these benchmarks (*TIP, no caching* on one disk) reduces elapsed time by up to 28% or an average of 17%. But, prefetching in parallel from a ten-disk array (*TIP, no caching* on ten disks) reduces elapsed time by up to 84% or an average of 63%. This latter performance gain is well in excess of the 50% maximum possible gain for overlapping I/O and computation which was described in Figure 2.3.

3. *Informed caching can increase cache effectiveness.* When applications repeatedly access more unique blocks than fit in the cache, informed caching can increase the

benchmark	1 disk		2 disks		3 disks		4 disks		10 disks	
	TIP, no cache	TIP	TIP, no cache	TIP	TIP, no cache	TIP	TIP, no cache	TIP	TIP, no cache	TIP
Davidson	0.99	0.63	0.98	0.81	0.97	0.88	0.96	0.88	0.90	0.83
XDataSlice	0.74	0.73	0.47	0.47	0.35	0.35	0.30	0.30	0.18	0.18
Sphinx	0.93	0.93	0.83	0.83	0.79	0.80	0.79	0.79	0.77	0.78
Agrep	0.83	0.83	0.48	0.49	0.37	0.37	0.28	0.28	0.16	0.16
Gnuld	0.81	0.76	0.50	0.46	0.41	0.38	0.35	0.34	0.26	0.26
Postgres, 20%	0.81	0.77	0.61	0.59	0.56	0.54	0.54	0.52	0.53	0.51
Postgres, 80%	0.72	0.50	0.46	0.35	0.38	0.32	0.35	0.31	0.33	0.31
geom. mean	0.83	0.72	0.59	0.55	0.51	0.48	0.46	0.44	0.37	0.36

Table 6.25. Performance summary for all the benchmarks. This table shows, for each benchmark and each array size, the elapsed time for *TIP, no caching* and *TIP* as a fraction of the elapsed time for *TIP, no hints* on the same array size. I use *TIP, no hints* instead of *Digital UNIX* as the base case to eliminate factors such as the LRU annex and focus instead on the impact of prefetching, caching, and clustering. The averages in the last row are the geometric mean of the numbers in each column.

number of reuse hits and reduce dependence on the disk. When coupled with informed clustering in *TIP*, informed caching reduces elapsed time by up to 36% or an average of 13% compared to *TIP, no caching* on a single disk where informed prefetching is least effective.

4. *Informed prefetching enables more effective disk scheduling that increases workload sequentiality when accesses are not already in ascending block-address order.* Informed prefetching uses hints to build larger disk queues which provides more opportunity for disk scheduling to sort requests and reduce average disk service time per block by up to 24%. But, sorting requests into ascending block-address order with the CSCAN algorithm only increases sequentiality if the requests are not already in ascending order. Many of the benchmarks do not perform sequential accesses, but do issue requests in ascending order and do not benefit significantly from more effective disk scheduling. However, when any of the benchmarks are multiprogrammed, the aggregate workload of the interleaved accesses is not in ascending order and disk scheduling can increase disk performance.

5. *Informed clustering can substantially reduce disk service time for random accesses.* Clustering sequential reads can reduce the CPU overhead servicing disk accesses, but because most modern disks perform their own sequential readahead, clustering

does not deliver sequential data any sooner. But, when clustering turns many random requests into fewer, larger ones, clustering increases disk workload sequentiality which, as seen in the Postgres benchmarks, can reduce disk service time by up to 22%.

6. *Informed clustering can increase the number of blocks transferred from disk.* Informed clustering only fetches blocks that are valuable enough to cache at the time they are fetched. But, value estimates are dynamic and better uses of the buffers holding cluster-prefetch blocks may arise, including the opportunity to cluster-prefetch for an access that will occur sooner. Any ejected cluster-prefetch blocks will have to be fetched from disk a second time, increasing the total blocks transferred from disk. In none of the experiments did this effect lead to a net increase in elapsed time because clustering accesses decreases per-block service time more than enough to offset the cost of refetching some ejected clustered-prefetch blocks.

7. *Cache replacement decisions affect disk workload sequentiality.* Ejecting a hinted block implies a subsequent prefetch of the block. If the ejected block is contiguous to another uncached, hinted block, it may be possible to prefetch both in a single cluster. On the other hand, if the ejected block is not contiguous to any other hinted block, prefetching it will require a separate disk access. Clearly, the sequentiality of these prefetches is greater in the former case. Developing ejection cost estimates that are sensitive to the difference in cost between clustered prefetch and non-clustered prefetches would be an interesting area of future research.

8. *Optimizing disk performance is most beneficial on a single disk.* More effective disk scheduling and request clustering which both increase workload sequentiality and therefore disk read-channel utilization only have a significant impact on elapsed time when the disk is the bottleneck on system performance. The disk bottleneck is most acute on a one-disk array and so that is where these techniques are most beneficial. When the disk is not the bottleneck, improvements in disk performance do not help improve the performance of whatever other system component is the bottleneck and so have a much smaller impact on overall performance.

9. *Informed prefetching from an array can compensate for poor workload sequentiality.* This is the dual of the previous rule. The cheapest way to obtain high I/O throughput is with sequential accesses to a small number of disks. A single modern disk drive can deliver about 10 MBytes/sec for a purely sequential workload. But, when forced to perform random 8-KByte accesses, its throughput drops to about 1 MByte/sec. Informed prefetching can take advantage of an array of ten disks to deliver 10 MBytes/sec despite poor workload sequentiality. It is still cheaper to obtain needed throughput with sequential accesses, but when sequential accesses are difficult or impossible to generate, informed prefetching from an array provides an alternative strategy that can compensate for poor access sequentiality and deliver the needed storage bandwidth.

10. *Multiprogramming I/O-intensive workloads on one disk reduces throughput.* When the limited bandwidth of a single disk is the bottleneck in a system, maximizing utilization of the disk maximizes system throughput. Interleaving accesses from multiple I/O-intensive applications, in general, reduces the sequentiality of the aggregate disk workload which reduces disk-head utilization and therefore disk throughput. This loss of disk throughput reduces system throughput by as much as 41% compared to serial execution of the applications. Informed prefetching mitigates this effect through improved disk scheduling, but system throughput still drops.

11. *Multiprogramming increases I/O concurrency and therefore the throughput of a disk array.* I already noted above that individual applications with serial disk workloads cannot exploit array parallelism. Multiprogramming such applications can generate I/O concurrency which increases array throughput. But, if the array is too small, interleaving accesses reduces the throughput of the individual disks which can negate the increase in throughput from I/O concurrency. In the two-application experiments, the break-even point was at about three disks; multiprogramming two applications on four or more disks increases throughput relative to serial execution. In contrast to informed prefetching which increases I/O concurrency and reduces elapsed time for individual applications, multiprogramming only increases aggregate

<table>
<tr><th rowspan="2">benchmark</th><th colspan="2">1 disk</th><th colspan="2">2 disks</th><th colspan="2">3 disks</th><th colspan="2">4 disks</th><th colspan="2">10 disks</th></tr>
<tr><th>1 hinter</th><th>both hint</th><th>1 hinter</th><th>both hint</th><th>1 hinter</th><th>both hint</th><th>1 hinter</th><th>both hint</th><th>1 hinter</th><th>both hint</th></tr>
<tr><td>Gnuld with</td><td>0.72</td><td rowspan="2">0.64</td><td>0.54</td><td rowspan="2">0.48</td><td>0.49</td><td rowspan="2">0.42</td><td>0.44</td><td rowspan="2">0.39</td><td>0.38</td><td rowspan="2">0.33</td></tr>
<tr><td>Agrep</td><td>0.83</td><td>0.88</td><td>0.90</td><td>0.91</td><td>0.94</td></tr>
<tr><td>Sphinx with</td><td>0.95</td><td rowspan="2">0.74</td><td>0.87</td><td rowspan="2">0.75</td><td>0.84</td><td rowspan="2">0.83</td><td>0.80</td><td rowspan="2">0.76</td><td>0.88</td><td rowspan="2">0.85</td></tr>
<tr><td>Davidson</td><td>0.82</td><td>0.88</td><td>0.92</td><td>0.88</td><td>0.96</td></tr>
<tr><td>XDataSlice with</td><td>0.66</td><td rowspan="2">0.52</td><td>0.62</td><td rowspan="2">0.39</td><td>0.63</td><td rowspan="2">0.34</td><td>0.63</td><td rowspan="2">0.33</td><td>0.60</td><td rowspan="2">0.27</td></tr>
<tr><td>Postgres, 80%</td><td>0.62</td><td>0.68</td><td>0.75</td><td>0.73</td><td>0.80</td></tr>
<tr><td>geom. mean</td><td>0.76</td><td>0.63</td><td>0.73</td><td>0.52</td><td>0.74</td><td>0.49</td><td>0.71</td><td>0.46</td><td>0.73</td><td>0.42</td></tr>
</table>

Table 6.26. Performance summary for the multiprogramming experiments. This table shows, for each array size, the elapsed time until both benchmarks complete when one of a pair and when both give hints as a fraction the elapsed time when neither gives hints. In the *1 hinter* columns, the hinter is the benchmark that appears on the same row as the ratio. The *average* row gives the geometric average of each column.

I/O concurrency which increases system throughput but does not reduce elapsed time for individual applications.

12. *On a single disk, informed prefetching and caching are even more beneficial when multiprogramming than when executing applications serially.* Table 6.26 summarizes the results of the multiprogramming experiments. When both of a pair of benchmarks give hints, *TIP* reduces elapsed time by an average of 37% compared to 28% when the benchmarks run alone. Managing the disk and cache well is even more important when multiprogramming is reducing the effectiveness of their use.

13. *Informed prefetching can derive a large benefit from even a small number of outstanding hints.* Because prefetching out to the prefetch horizon effectively eliminates stall for hinted accesses when an array is available, it follows that informed prefetching does not require hints beyond the prefetch horizon. In fact, as can be seen from the graphs showing stall time as a function of prefetch depth, large reductions in stall are possible at prefetch depths much smaller than the prefetch horizon. It is important that applications disclose multiple accesses at once, but it is not necessary to disclose thousands at once. Nevertheless, I do not recommend that programmers tune their hint-giving to the prefetch horizon. Giving as many hints as possible will help ensure that there are enough hints for future machines with larger prefetch horizons.

Furthermore, deep hints are beneficial for informed clustering and caching, as noted below.

14. *To be useful, hints do not need to be given far in advance if they are given in batches and an array is available.* Hints given far in advance are useful for overlapping I/O with computation. But, achieving the much larger performance gains of parallel prefetching requires multiple outstanding hints at a time but not much advance notice. XDataSlice, for example, discloses its hints and immediately starts reading data, but still derives a huge benefit from I/O parallelism.

15. *Deep hints are needed for informed clustering and caching.* Informed clustering can merge widely separated accesses into one. Informed caching can hold onto blocks for hundreds or thousands of accesses. But, clustering and caching thousands of accesses in advance requires hints for thousands of accesses. The more accesses disclosed, the greater the opportunity for clustering and caching. Because these optimizations are most important on a single disk or small arrays, it follows that deep hints are most important there as well.

16. *Heuristic prefetching needs to be more adaptive than existing readahead strategies.* Digital UNIX has an adaptive sequential readahead strategy that scales its depth of prefetching in proportion to the number of blocks read sequentially. This strategy works well for Davidson's sequential workload, but hurts performance when applications such as XDataSlice and Gnuld read a few blocks sequentially and then seek to a new offset. The readahead strategy could possibly reduce the number of these harmful, false readaheads if it monitored the success of its prefetches and adapted its aggressiveness accordingly.

17. *The LRU queue is an imperfect predictor of future behavior.* The LRU algorithm is the most common heuristic for determining which blocks to cache and which to eject. Yet, it is an imperfect estimator for the value of caching blocks for unhinted accesses that is especially vulnerable to phase transitions. Postgres, for example, grows the LRU queue in fruitless attempt to achieve cache hits on the already ejected

outer-relation blocks. Its twice-accessed workload is particularly bad for LRU estimation.

18. *Pathologically unbalanced disk loads exist.* Striping effectively balances the load for most of the applications, but XDataSlice demonstrates that simple striping is not a universal solution. Randomized striping could help balance the load within a single device, but there will inevitably be imbalances among devices. Ideally, prefetching and caching should be sensitive to such imbalances and adapt accordingly. Recent work has shown how this can be done [Kimbrel96, Tomkins97].

6.6 System overhead

TIP's cost-benefit cache management adds both CPU and memory overheads to the system. In this section, I quantify these overheads.

To measure the CPU overheads of the different components of the system, I added hand-coded trace points to the entry and exit of selected functions and collected traces of five runs of each of the benchmark applications. I post-processed the traces to determine how much time was spent in the different components of the system.

6.6.1 Tracing infrastructure

Each trace record contains 8 bytes. The time stamp, occupying 4 bytes, is the current value of the Digital Alpha processor cycle counter which has a resolution of 1/175 MHz. Two bytes are used for a tag that uniquely identifies each trace point. The last two bytes are available for a parameter which is used only when switching from one task to another to record the process id of the old and new processes.

When tracing is on, trace records are stuffed into a statically-allocated in-kernel buffer 64 MBytes in size which is large enough for about 8 million trace records. When the benchmark run is finished, tracing is turned off and the contents of the buffer are read and stored in a file for later processing. Because the buffer is so large, it disturbs the normal paging behavior of some of the benchmarks. To compensate, an additional 64 MBytes of RAM were added to the system during tracing runs.

To minimize tracing overhead, there is no locking on the trace buffer. Instead, conflicts for the buffer are detected and compensated for during post-processing by checking for backwards-moving time stamps. Also, rather than checking for buffer overflow, the size of the buffer is restricted to be a power of two, and the high-order bits of the index into the buffer are masked off, effectively implementing a circular buffer. At the end of a tracing run, all of the bits of the index into the trace buffer are checked to ensure that there has been no wrap-around. The number of records collected during a run ranges from a low of about 130,000 for Agrep running on Digital UNIX to a high of about 8,400,000[3] for Davidson when hinting on TIP. Each trace record adds about 40 cycles of overhead.There is some variation due to cache effects. This overhead is substantially lower than automated techniques such as Digital's ATOM [Eustace95] package which would have added a few hundred cycles per record.

6.6.2 CPU overhead

Table 6.27 analyzes the CPU activity of the seven benchmarks into six categories. *User* is the time spent at user level between system calls. It does not include time spent servicing disk interrupts, but it does include untraced interrupts, such as the clock. *System total* reports the total CPU time spent in system calls by the application plus disk interrupts plus idle-process I/O-completion activity. Because these are I/O-intensive benchmarks, almost all system time is spent in the *file system*. The table reports the *total* time for the file system and breaks this into four sub-categories, *copy*, *I/O*, *TIP*, and *other*. It gives the 95% confidence interval based on the five runs in parentheses and the percentage of the total file-system time spent in each of these sub-categories. *Copy* is the time spent moving data between user space and the kernel cache buffers. *I/O* is time spent marshalling buffers for disk accesses plus time spent actually performing the accesses including queuing requests, initiating them at the drive, and servicing interrupts. Idle-process I/O activity is included here. I/O interrupts serviced by other processes are not included, but this time is only about 1-3% of the total time for I/O. It is interesting, for example, to note the large reduction in the time XDataSlice spends on I/O when it gives hints and does not suffer from false readahead. *Other* includes all other file-system activity in the unmodified system, including time spent going through the vnode layer, finding blocks in the cache,

[3] This one run requires a little more than 64 MBytes of RAM, so I enlarged the buffer a little, but lied to the tracing code, telling it there was a 128 MByte buffer, and kept my fingers crossed that the run completed before the buffer was overrun. Fortunately, it did.

benchmark	system	user	system total	file system				
				total	copy	I/O	TIP	other
Davidson	Digital UNIX	82.01 (0.63)	26.83 (0.21)	26.65 (0.19) 100.0%	14.68 (0.12) 55.1%	5.91 (0.09) 22.2%	0.00 (0.00) 0.0%	6.07 (0.08) 22.8%
	TIP, no hints	83.47 (0.93)	31.64 (0.46)	31.45 (0.45) 100.0%	14.89 (0.26) 47.3%	5.61 (0.19) 17.8%	4.39 (0.09) 14.0%	6.57 (0.08) 20.9%
	TIP	81.19 (1.12)	29.43 (0.23)	29.24 (0.23) 100.0%	14.27 (0.14) 48.8%	3.11 (0.20) 10.6%	7.60 (0.04) 26.0%	4.26 (0.05) 14.6%
XDataSlice	Digital UNIX	11.46 (0.01)	26.72 (0.09)	25.83 (0.07) 100.0%	7.73 (0.01) 29.9%	12.21 (0.03) 47.3%	0.00 (0.00) 0.0%	5.89 (0.05) 22.8%
	TIP, no hints	11.48 (0.06)	29.38 (0.15)	28.38 (0.15) 100.0%	7.70 (0.01) 27.1%	11.81 (0.02) 41.6%	2.72 (0.02) 9.6%	6.15 (0.12) 21.7%
	TIP	11.43 (0.02)	21.73 (0.36)	20.72 (0.35) 100.0%	7.74 (0.01) 37.3%	4.74 (0.31) 22.9%	3.76 (0.02) 18.1%	4.48 (0.02) 21.6%
Sphinx	Digital UNIX	133.67 (0.46)	11.18 (0.05)	9.89 (0.06) 100.0%	3.08 (0.03) 31.1%	2.90 (0.03) 29.3%	0.00 (0.00) 0.0%	3.92 (0.03) 39.6%
	TIP, no hints	135.15 (1.30)	13.52 (0.39)	12.03 (0.34) 100.0%	3.10 (0.04) 25.7%	2.98 (0.04) 24.8%	1.33 (0.01) 11.1%	4.62 (0.28) 38.4%
	TIP	136.88 (0.86)	14.84 (0.07)	13.32 (0.09) 100.0%	3.09 (0.03) 23.2%	2.27 (0.02) 17.1%	3.21 (0.03) 24.1%	4.75 (0.03) 35.7%
Agrep	Digital UNIX	0.60 (0.01)	1.64 (0.08)	1.24 (0.08) 100.0%	0.25 (0.03) 20.2%	0.68 (0.04) 54.9%	0.00 (0.00) 0.0%	0.31 (0.03) 24.9%
	TIP, no hints	0.60 (0.02)	1.73 (0.03)	1.31 (0.03) 100.0%	0.24 (0.01) 18.0%	0.68 (0.03) 51.8%	0.09 (0.00) 7.1%	0.30 (0.01) 23.0%
	TIP	0.59 (0.00)	1.81 (0.02)	1.35 (0.02) 100.0%	0.23 (0.00) 16.9%	0.47 (0.01) 34.7%	0.34 (0.00) 25.2%	0.31 (0.01) 23.2%
Gnuld	Digital UNIX	5.23 (0.03)	5.37 (0.11)	4.88 (0.10) 100.0%	1.14 (0.01) 23.3%	2.20 (0.07) 45.0%	0.00 (0.00) 0.0%	1.55 (0.04) 31.8%
	TIP, no hints	5.21 (0.01)	6.14 (0.10)	5.57 (0.10) 100.0%	1.14 (0.00) 20.4%	2.24 (0.07) 40.2%	0.55 (0.01) 9.8%	1.64 (0.02) 29.5%
	TIP	5.27 (0.02)	6.01 (0.11)	5.47 (0.11) 100.0%	1.15 (0.00) 21.1%	1.39 (0.05) 25.5%	1.31 (0.02) 24.0%	1.61 (0.05) 29.5%

Table 6.27. CPU profile by benchmark. This table shows the CPU time in seconds that each benchmark spends in *user* and *system* code and the system time spent in the *file system*. *File system* time is broken into four categories: copying data between user and system space (*copy*); initiating and servicing disk requests (*I/O*); in TIP-specific activities (*TIP*); and in all other activities such as reassigning buffers from one block to another (*other*). The numbers in parentheses are the 95% confidence intervals based on five runs. Table 6.28 summarizes these numbers.

benchmark	system	user	system total	file system				
				total	copy	I/O	TIP	other
Postgres, 20% match	Digital UNIX	24.43 (0.10)	3.22 (0.08)	3.05 (0.06) 100.0%	1.02 (0.01) 33.5%	1.22 (0.03) 40.1%	0.00 (0.00) 0.0%	0.81 (0.03) 26.4%
	TIP, no hints	24.82 (0.18)	3.62 (0.03)	3.44 (0.03) 100.0%	1.07 (0.01) 31.1%	1.26 (0.02) 36.6%	0.27 (0.01) 7.9%	0.84 (0.02) 24.4%
	TIP	24.45 (0.59)	3.69 (0.04)	3.51 (0.03) 100.0%	1.09 (0.01) 31.0%	1.03 (0.02) 29.4%	0.51 (0.02) 14.5%	0.88 (0.02) 25.0%
Postgres, 80% match	Digital UNIX	36.01 (0.41)	10.26 (0.12)	9.74 (0.10) 100.0%	3.64 (0.06) 37.4%	3.65 (0.03) 37.5%	0.00 (0.00) 0.0%	2.45 (0.04) 25.1%
	TIP, no hints	36.95 (0.34)	11.71 (0.12)	11.06 (0.12) 100.0%	3.58 (0.01) 32.4%	3.67 (0.06) 33.2%	1.02 (0.04) 9.2%	2.79 (0.13) 25.2%
	TIP	35.88 (0.34)	11.27 (0.14)	10.63 (0.13) 100.0%	3.79 (0.03) 35.6%	1.96 (0.04) 18.5%	2.16 (0.07) 20.3%	2.72 (0.07) 25.6%

Table 6.27. CPU profile by benchmark. This table shows the CPU time in seconds that each benchmark spends in *user* and *system* code and the system time spent in the *file system*. *File system* time is broken into four categories: copying data between user and system space (*copy*); initiating and servicing disk requests (*I/O*); in TIP-specific activities (*TIP*); and in all other activities such as reassigning buffers from one block to another (*other*). The numbers in parentheses are the 95% confidence intervals based on five runs. Table 6.28 summarizes these numbers.

reallocating buffers from one block to another, and time spent initiating heuristic readahead. To facilitate comparison between the overhead of heuristic readahead and of informed prefetching, the time TIP spends initiating informed prefetches is also reported in *other*. However, the time TIP spends calculating cost and benefit estimates to determine whether it should prefetch is included in the *TIP* category which includes all activities that are unique to the TIP system. Thus, *TIP* includes all cost and benefit calculations, profiling the LRU cache, running the min-max algorithm to pick pages for replacement, and tracking blocks in the cache of hinted blocks.

Table 6.28 summarizes the file-system results for all of the benchmarks. The total file-system time is the geometric mean of the ratios between *TIP, no hints* or *TIP* and the base UNIX system. For the four file-system components, I wanted one set of numbers that would simultaneously give a feel for both the component-by-component relative performance of the systems, and the portion of total time that each system spends in each component. Thus, for each component, I report the arithmetic average percentage of file-system time it represents, scaled by the previously computed ratio for the total file-system

system	file system				
	total	**copy**	**I/O**	**TIP**	**other**
Digital UNIX	1.000	0.33	0.39	0.00	0.28
TIP, no hints	1.135	0.33	0.40	0.11	0.30
TIP	1.089	0.33	0.25	0.24	0.27

Table 6.28. File system CPU overhead summary. This table shows the geometric average ratio of time spent in the TIP file system with and without hints relative to the base UNIX system. For the four components, this ratio is multiplied by the average fraction of filesytem time each represents (see text for further explanation). Overall, TIP adds a 13.5% CPU overhead to the file system when not given hints. When hints are available, TIP I/O and caching optimizations partially offset TIP overhead and reduce the net overhead to 8.9%.

times. For example, to compute the relative time *TIP, no hints* spends in *I/O*, I first compute the arithmetic average of the percentages of time *TIP, no hints* spends in *I/O* in each of the benchmarks,

$$\frac{17.8\% + 41.6\% + 24.8\% + 51.8\% + 40.2\% + 36.6\% + 33.2\%}{7} = 0.35,$$

and then scale this by the total time spent in *TIP, no hints,*

$$0.35 \times 1.135 = 0.40.$$[4]

The summary in Table 6.28 shows that when the applications do not give hints, TIP adds an average overhead of 13.5% to the file system, most of which is spent in the *TIP* category. When the applications do give hints, TIP's overall overhead drops to 8.9% even though the time spent in TIP functions doubles. Several factors contribute to this drop. First, TIP's I/O optimizations reduce the time spent on I/O by an average of over 35%. Applications such as Davidson, which benefit from informed caching, spend less time on I/O because they read fewer blocks from disk when they give hints. Others, such as XDataSlice, read fewer blocks because informed prefetching reduces false readahead.

[4] I like this approach better than two possible alternatives. One would have been to report ratios for each component just as I do for the total. Unfortunately, the ratio is not defined for the *TIP* component which has the value 0.00 for the base Digital UNIX system. Another alternative would have been to compute the geometric average of the percentages of time spent in each component. But, I prefer the arithmetic averages because they sum to 100% for each system whereas the geometric averages do not. Using the arithmetic average is equivalent to implicitly weighting equally each benchmark's contribution to overall system performance and taking the total time to run all benchmarks as the metric of interest. In this case, the total file-system time and the total time spent in each component would be meaningful numbers in and of themselves and then, according to Jain [Jain91, p. 190], the arithmetic average, and not the geometric average, would be the best estimate of the portion of time spent in each component. So, reporting the arithmetic average is not unreasonable. Moreover, scaling the arithmetic averages by the ratios for the total file-system time allows both component-by-component comparison and comparison among the components for each system.

Finally, other applications, such as Postgres, benefit from informed clustering of multiple reads into one disk access. An understanding of where time is spent in TIP, and how giving hints reduces *other* file-system overhead by nearly 10% requires further analysis.

Table 6.29 breaks time in the *TIP* column of Table 6.27 into five categories and Table 6.30 uses the same approach as Table 6.28 to summarize the results except that *TIP, no hints* is used as the base. *Hint bookkeeping* includes the time to give hints, resolve names of hinted files into a file handle, and build the internal data structures that represent the hints in the cache manager. It also includes a check on every read for a matching hint. This is why there is non-zero hint bookkeeping even when applications don't give hints. *LRU profiling* is the time spent recording where in the LRU queue cache hits occur. The LRU estimator uses this information to generate its estimate of the cost of shrinking the LRU queue. Because the LRU annex was created to enable efficient profiling of the LRU queue, time spent moving buffers to and from the annex is included here. *LRU profiling* represents by far the largest portion of TIP overhead without hints. *Hinted-block tracking* refers to the time spent by hint estimators updating their data structures when they begin or end 'tracking' a block as part of the min-max algorithm described in Section 4.3.5. The current implementation's use of a simple insertion sort when it starts tracking a block appears not to add too much overhead in most cases. *Hint cost/benefit estimates* is the time spent simply computing cost and benefit estimates for prefetching or hinted caching whenever the min-max algorithm needs one to make allocation decisions. The overhead here is substantial because of the relatively slow divide operations. Reimplementing these cost calculations as a table lookup, at least within the prefetch horizon, could probably reduce this time substantially. Finally, *pick, query, update* is the core of the min-max algorithm. It includes the time spent picking the least valuable buffer, updating the new value of the estimator that gave up the buffer (except the actual cost estimate), querying other interested estimators, and updating the cost for an estimator that starts tracking the picked block. It also includes benefit updates when an application consumes a hinted block or when the prefetcher issues a new prefetch.

When the system is running without hints, two-thirds of the time is spent profiling the LRU queue. To reduce this overhead, it would certainly be possible to dynamically size the LRU-queue segments to use larger ones during periods when the system had few hints

benchmark	system	TIP total	hint book-keeping	LRU profiling	hinted-block tracking	hint cost/benefit estimates	pick, query, update
Davidson	TIP, no hints	4.39 (0.09) 100.0%	0.04 (0.00) 0.8%	3.16 (0.09) 71.9%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	1.20 (0.03) 27.3%
	TIP	7.60 (0.04) 100.0%	0.76 (0.01) 10.0%	2.29 (0.02) 30.1%	0.65 (0.01) 8.6%	1.16 (0.02) 15.2%	2.74 (0.03) 36.1%
XDataSlice	TIP, no hints	2.72 (0.02) 100.0%	0.06 (0.01) 2.2%	1.73 (0.00) 63.6%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.93 (0.00) 34.1%
	TIP	3.76 (0.02) 100.0%	0.70 (0.00) 18.7%	1.05 (0.01) 28.0%	0.24 (0.01) 6.4%	0.61 (0.01) 16.2%	1.15 (0.02) 30.7%
Sphinx	TIP, no hints	1.33 (0.01) 100.0%	0.07 (0.00) 5.2%	0.91 (0.01) 68.2%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.35 (0.01) 26.6%
	TIP	3.21 (0.03) 100.0%	0.79 (0.01) 24.7%	0.76 (0.01) 23.7%	0.06 (0.00) 1.8%	0.61 (0.01) 19.1%	0.99 (0.03) 30.7%
Agrep	TIP, no hints	0.09 (0.00) 100.0%	0.01 (0.00) 6.2%	0.06 (0.00) 64.5%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.03 (0.00) 29.1%
	TIP	0.34 (0.00) 100.0%	0.17 (0.00) 50.1%	0.05 (0.00) 16.1%	0.01 (0.00) 2.1%	0.04 (0.00) 11.3%	0.07 (0.00) 20.5%
Gnuld	TIP, no hints	0.55 (0.01) 100.0%	0.02 (0.00) 3.1%	0.37 (0.01) 67.6%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.16 (0.01) 29.4%
	TIP	1.31 (0.02) 100.0%	0.49 (0.01) 37.3%	0.31 (0.00) 23.4%	0.06 (0.01) 4.9%	0.16 (0.00) 11.9%	0.30 (0.01) 22.6%
Postgres 20% match	TIP, no hints	0.27 (0.01) 100.0%	0.01 (0.00) 4.1%	0.20 (0.01) 72.4%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.06 (0.00) 23.5%
	TIP	0.51 (0.02) 100.0%	0.07 (0.00) 13.6%	0.20 (0.01) 39.4%	0.03 (0.00) 6.2%	0.07 (0.01) 13.9%	0.14 (0.01) 26.8%
Postgres 80% match	TIP, no hints	1.02 (0.04) 100.0%	0.09 (0.00) 9.0%	0.71 (0.04) 69.5%	0.00 (0.00) 0.0%	0.00 (0.00) 0.0%	0.22 (0.01) 21.5%
	TIP	2.16 (0.07) 100.0%	0.38 (0.06) 17.8%	0.61 (0.01) 28.2%	0.42 (0.03) 19.4%	0.29 (0.01) 13.5%	0.46 (0.02) 21.1%

Table 6.29. TIP CPU overhead. This table breaks the time spent in *TIP* in Table 6.27 into five categories. *Hint bookkeeping* includes storing hints and checking whether accesses match hints which occurs even when an application does not hint. *LRU profiling* is time spent estimating the hit ratio as a function of LRU queue length and represents by far the largest share of TIP overhead in the absence of hints. *Hinted block tracking* is the cost of tracking blocks in the hinted cache. *Hint cost/benefit estimates* is the cost of computing cost and benefit estimates. *Pick, query, update* is the core of the min-max buffer allocation algorithm. The numbers in parentheses are the 95% confidence intervals. The percentages are of total *TIP* time.

system	TIP					
	total	hint book-keeping	LRU profiling	hinted-block tracking	hint cost/benefit estimates	pick, query, update
TIP, no hints	1.000	0.04	0.68	0.00	0.00	0.27
TIP	2.130	0.52	0.57	0.15	0.31	0.57

Table 6.30. TIP CPU overhead summary. This table shows the geometric average ratio between the total times spent in *TIP* with and without hints. For the five *TIP* components, this ratio is multiplied by the average fraction of time spent in each component. Without hints, about two-thirds of TIP overhead is profiling the LRU queue. With hints, LRU profiling overhead drops slightly because, for example, false readaheads no longer go through the queue, but all other overheads increase which doubles overall TIP CPU overhead.

and therefore little need for precise profiling of the queue. Then, when hints did arrive, the segments could be shrunk to gain a more precise estimation of the value of LRU-queue buffers. A more radical solution would be to change the way LRU cost estimates are generated. In Chapter 7, I suggest a possible alternative.

To explore the effect giving hints has on the *other* parts of the file system (from Table 6.27), Table 6.31 details how time is spent within this category and Table 6.32 summarizes the results in the usual way. The tables show the time spent in the different layers of code traversed by a read request as it goes through the system. At the highest level, *system call to copyout loop* includes the time from the invocation of a read or write system call down through the VFS layer to the copyout loop in the UFS layer. The copyout loop includes three main steps: get a buffer, copy its contents to/from user space, and release the buffer. The time to actually copy the data was separately accounted for in Table 6.27. The other two steps are accounted for in this table by *get data buffer* and *release hold on buffer*.

Get data buffer includes the time to call the cache manager with a request for a block and if the requested block is not cached, to allocate a new buffer and initiate a disk access. Recall that the time to actually perform the I/O was separately accounted for in Table 6.27. The time to do the cache lookup is *cache lookup*. When the TIP system has hints, this lookup step is avoided because the block is found directly from the hint (see Section 5.1 for details). This is why the lookup time for the TIP system is so much lower than for *Digital UNIX* and *TIP, no hints*. If the buffer is not cached and a buffer must be allocated, then the time to allocate a new buffer and reassign it from the old to the new block is *allocate buffer*. Note that *allocate buffer* does not include the time to run the TIP allocation

benchmark	system	other							
		total	system call to copyout loop	get data buffer	cache lookup	allocate buffer	cluster I/O requests	release hold on buffer	build prefetch requests
Davidson	Digital UNIX	6.07 (0.08) 100.0%	0.97 (0.07) 15.9%	1.09 (0.04) 17.9%	0.73 (0.01) 12.0%	1.53 (0.04) 25.3%	0.48 (0.00) 7.9%	0.61 (0.01) 10.1%	0.65 (0.01) 10.7%
	TIP, no hints	6.57 (0.08) 100.0%	1.00 (0.04) 15.2%	1.21 (0.02) 18.5%	0.99 (0.05) 15.1%	1.61 (0.01) 24.5%	0.63 (0.01) 9.6%	0.52 (0.01) 7.9%	0.61 (0.02) 9.3%
	TIP	4.26 (0.05) 100.0%	1.00 (0.01) 23.5%	1.50 (0.01) 35.2%	0.01 (0.00) 0.3%	0.36 (0.01) 8.5%	0.35 (0.00) 8.3%	0.37 (0.01) 8.7%	0.66 (0.03) 15.4%
XDataSlice	Digital UNIX	5.89 (0.05) 100.0%	1.11 (0.01) 18.9%	1.35 (0.02) 22.9%	0.42 (0.01) 7.2%	1.24 (0.01) 21.1%	0.39 (0.00) 6.7%	0.41 (0.00) 7.0%	0.94 (0.01) 15.9%
	TIP, no hints	6.15 (0.12) 100.0%	1.33 (0.03) 21.6%	1.42 (0.04) 23.0%	0.48 (0.00) 7.8%	1.24 (0.04) 20.1%	0.45 (0.01) 7.4%	0.36 (0.00) 5.8%	0.87 (0.01) 14.2%
	TIP	4.48 (0.02) 100.0%	1.43 (0.01) 31.9%	0.99 (0.01) 22.2%	0.01 (0.00) 0.3%	0.35 (0.01) 7.9%	0.33 (0.01) 7.4%	0.32 (0.00) 7.1%	1.04 (0.01) 23.1%
Sphinx	Digital UNIX	3.92 (0.03) 100.0%	1.51 (0.02) 38.5%	0.94 (0.02) 23.9%	0.34 (0.02) 8.8%	0.47 (0.01) 11.9%	0.13 (0.00) 3.2%	0.30 (0.00) 7.7%	0.23 (0.00) 6.0%
	TIP, no hints	4.62 (0.28) 100.0%	1.84 (0.12) 39.9%	1.11 (0.01) 24.0%	0.45 (0.02) 9.7%	0.44 (0.01) 9.4%	0.17 (0.00) 3.8%	0.37 (0.16) 8.0%	0.23 (0.00) 5.1%
	TIP	4.75 (0.03) 100.0%	1.72 (0.02) 36.3%	1.52 (0.05) 31.9%	0.03 (0.00) 0.5%	0.21 (0.00) 4.4%	0.19 (0.00) 4.0%	0.27 (0.00) 5.6%	0.82 (0.01) 17.2%
Agrep	Digital UNIX	0.31 (0.03) 100.0%	0.10 (0.01) 33.4%	0.06 (0.01) 19.9%	0.02 (0.01) 7.1%	0.07 (0.01) 23.1%	0.01 (0.00) 3.8%	0.02 (0.00) 6.7%	0.02 (0.00) 5.6%
	TIP, no hints	0.30 (0.01) 100.0%	0.12 (0.00) 38.7%	0.07 (0.00) 22.5%	0.03 (0.01) 8.4%	0.05 (0.00) 15.5%	0.01 (0.00) 4.2%	0.02 (0.00) 5.8%	0.02 (0.00) 5.1%
	TIP	0.31 (0.01) 100.0%	0.11 (0.00) 36.2%	0.07 (0.00) 22.8%	0.00 (0.00) 0.3%	0.03 (0.00) 9.2%	0.02 (0.00) 6.4%	0.02 (0.00) 5.4%	0.06 (0.01) 20.0%
Gnuld	Digital UNIX	1.55 (0.04) 100.0%	0.41 (0.01) 26.5%	0.52 (0.01) 33.5%	0.12 (0.00) 7.7%	0.25 (0.01) 16.1%	0.05 (0.00) 3.3%	0.12 (0.00) 7.5%	0.07 (0.01) 4.8%
	TIP, no hints	1.64 (0.02) 100.0%	0.48 (0.01) 29.3%	0.55 (0.01) 33.3%	0.15 (0.00) 9.3%	0.22 (0.01) 13.2%	0.07 (0.00) 4.1%	0.10 (0.00) 6.0%	0.07 (0.01) 4.4%
	TIP	1.61 (0.05) 100.0%	0.47 (0.01) 29.0%	0.55 (0.01) 34.1%	0.04 (0.00) 2.4%	0.12 (0.00) 7.3%	0.09 (0.01) 5.4%	0.09 (0.00) 5.6%	0.25 (0.01) 15.8%

Table 6.31. CPU overhead of the *other* part of the file system. This table shows how time is spent in the *other* category of file-system time in Table 6.27 for each of the benchmarks. The numbers in parentheses are the 95% confidence intervals and percentages are of total *other* time. Table 6.32 summarizes these data.

benchmark	**system**	**other**							
		total	**system call to copyout loop**	**get data buffer**	**cache lookup**	**allocate buffer**	**cluster I/O requests**	**release hold on buffer**	**build prefetch requests**
Postgres, 20% match	Digital UNIX	0.81 (0.03) 100.0%	0.23 (0.01) 29.1%	0.28 (0.01) 35.2%	0.06 (0.02) 7.3%	0.13 (0.01) 15.9%	0.02 (0.00) 2.4%	0.05 (0.00) 5.8%	0.03 (0.00) 3.7%
	TIP, no hints	0.84 (0.02) 100.0%	0.28 (0.01) 33.2%	0.32 (0.01) 38.5%	0.05 (0.00) 6.5%	0.09 (0.00) 10.4%	0.02 (0.00) 2.4%	0.04 (0.01) 5.2%	0.03 (0.00) 3.1%
	TIP	0.88 (0.02) 100.0%	0.27 (0.01) 30.6%	0.25 (0.01) 29.0%	0.03 (0.00) 3.9%	0.06 (0.00) 6.9%	0.05 (0.01) 5.1%	0.05 (0.00) 5.2%	0.16 (0.00) 18.7%
Postgres, 80% match	Digital UNIX	2.45 (0.04) 100.0%	0.83 (0.03) 34.1%	0.93 (0.03) 38.1%	0.15 (0.00) 6.1%	0.31 (0.01) 12.7%	0.04 (0.00) 1.7%	0.14 (0.00) 5.7%	0.03 (0.00) 1.3%
	TIP, no hints	2.79 (0.13) 100.0%	1.08 (0.06) 38.6%	1.08 (0.08) 38.7%	0.18 (0.01) 6.5%	0.24 (0.01) 8.7%	0.04 (0.00) 1.5%	0.13 (0.01) 4.6%	0.03 (0.00) 1.2%
	TIP	2.72 (0.07) 100.0%	0.97 (0.04) 35.5%	0.82 (0.04) 30.2%	0.11 (0.01) 4.1%	0.15 (0.02) 5.4%	0.11 (0.01) 4.1%	0.16 (0.04) 5.7%	0.40 (0.01) 14.7%

Table 6.31. CPU overhead of the *other* part of the file system. This table shows how time is spent in the *other* category of file-system time in Table 6.27 for each of the benchmarks. The numbers in parentheses are the 95% confidence intervals and percentages are of total *other* time. Table 6.32 summarizes these data.

system	**other**							
	total	**system call to copyout loop**	**get data buffer**	**cache lookup**	**allocate buffer**	**cluster I/O requests**	**release hold on buffer**	**build prefetch requests**
Digital UNIX	1.000	0.28	0.27	0.08	0.18	0.04	0.07	0.07
TIP, no hints	1.072	0.33	0.30	0.10	0.16	0.05	0.07	0.06
TIP	0.973	0.31	0.29	0.02	0.07	0.06	0.06	0.17

Table 6.32. Summary of the CPU overhead of the *other* part of the file system. Performance in this part of the file system is comparable in the TIP and base UNIX systems when there are no hints. With hints, the TIP optimizations lead to a net reduction of 10% in the overhead of the *other* component of the file system.

algorithm which is accounted for by the *pick, query, update* component of Table 6.29. Before an I/O to fill the buffer is initiated, the system attempts to cluster requests for contiguous blocks into one access which takes time *cluster I/O requests*. Once the hi-level copyout loop copies the data, the buffer is released back to the cache in time *release hold on buffer*. After the system services read hits or misses, it executes the readahead heuristic for unhinted accesses or checks the hint lists for something to prefetch which takes time *build prefetch requests*.

As the summary data in Table 6.32 show, the TIP and base UNIX system are comparable except that when TIP has hints, it can reduce the *cache lookups* and *allocate buffers* components which more than offsets the increase in the *build prefetch requests* component. The net effect is about a 10% reduction in the overhead for the *other* part of the file system. Because *other* accounts for 25-30% of total file-system overhead, the overall reduction in file-system overhead is 2.5-3.0%.

6.6.3 Memory overhead

The TIP system consumes some memory for the data structures that keep track of the buffers, store hints, and link hints to buffers as hints are resolved.

Even when there are no hints, TIP allocates a *tipBuf* structure (see Figure 5.8) for each cache page which supports functions such as profiling the LRU queue. The structure consumes 104 bytes per page which amounts to a 1.2% overhead.

When an application issues a hint, TIP allocates one 568-byte *tipHnt* structure plus one 24-byte *tipSeg* for each hinted segment or sequential byte range (see Section 3.2). The amount of data actually consumed by hint structures is very much application-dependent. As Table 3.3 showed, there may from one to thousands of segments per hint and from one to thousands of blocks per segment. The aggregate memory consumed can be substantial. The Postgres (80% match) benchmark issues a hint with 15916 segments which occupies a total of almost 47 pages of memory. For the system to support truly vast numbers of hints, it may become necessary to store distant hints on disk.

Of more concern might be the tipHnt structure if many applications give separate hints for very small amounts of data. If this becomes a problem, then some simple optimizations could significantly reduce the size of the tipHnt structure. Of the 568 bytes in the structure, 456 are devoted to recording permissions data to aid name resolution. This data could be stored per-process instead of per-hint to eliminate 80% of this overhead. It was not a problem in our experiments, so this data was stored with the hint for convenience.

The final major memory overhead comes when the hints are resolved. There is one 64-byte *tipNex* structure for each resolved block.[5] And, if the hinted block is not either

[5] It would be possible to dispense with the separate tracking and prefetching links, have one list for the whole access sequence, and rely on a flag to indicate that the buffer was being tracked. This would save 24 bytes per nexus, but would add CPU overhead for sequentially searching the list for tracked blocks.

already resident, an LRU ghost buffer, or the target of a resolved hint, the system allocates a tipBuf structure for the block. In the worst case, where all hints are for unique, uncached blocks, this amounts to an overhead of 64+104=168 bytes per resolved block. How many such structures are simultaneously allocated is application dependent. Certainly there are no more of them than there are outstanding hints, and most of the benchmarks only issue a portion of their hints at a time. But, Postgres, for example, issues many hints simultaneously. In this case, the caching horizon, described in Section 5.3.1, limits how much memory may be devoted to any single hint stream to at most 10,000 blocks and usually less. At 10,000 resolved blocks, the tipNex and tipBuf structures could consume as much as 205 pages which is the largest single potential overhead.

A substantial portion of the TIP data structures are memory pointers which on the Digital Alpha CPU are 8 bytes in size. A 32-bit architecture would suffer half the overhead for these pointers which would reduce a tipBuf from 104 to 60 bytes and a tipNex from 64 to 36 bytes. This would reduce the worst-case 205 pages of resolved-hint overhead to 117 pages.

These data structures occupy the space they do because they explicitly and exhaustively enumerate the outstanding, resolved hints. An interesting area of future work would be to develop more compact representations of hints that yet could support the required pick, query, and bid functions without adding too much CPU overhead.

6.7 Conclusion

The experimental results presented in this chapter support three primary conclusions:

1. many I/O-intensive applications do not benefit from a disk array;
2. informed prefetching's greatest gains come from prefetching in parallel; and
3. informed caching and the disk optimizations deliver their greatest gains on small arrays.

Together, informed prefetching and caching are hugely successful at reducing application elapsed time on any array size.

On a single disk, TIP reduces elapsed time for the suite of application benchmarks by 7% to 50% or an average of 28%. When multiprogramming two applications on a single disk, TIP's informed resource management is even more beneficial, reducing elapsed time for the pair of applications by 26% to 48% or an average of 37% when both applications give hints. When it can exploit the parallelism of a ten-disk array for parallel prefetching, TIP reduces elapsed time by 17% to 84% or an average of 64%. Because multiprogramming on a ten-disk array increases both CPU and array utilization even without hints, there is less opportunity for TIP to improve performance. Nevertheless, TIP reduces elapsed time for a pair of applications running on a ten-disk array by 15% to 73% or an average of 58% when both applications give hints.

A combination of optimizations is responsible for producing these overall results. Among these, the experiments showed that informed prefetching, which takes advantage of hints within the prefetch horizon, reduces elapsed time by up to 28% or an average of 17% on one disk and by up to 84% or an average of 63% on a ten-disk array. The experiments measuring stall time as a function of prefetch depth clearly show that when the bandwidth of a disk array is available, not even a full prefetch horizon worth of outstanding hints is required to deliver huge performance gains. Programmers and researchers into techniques for automatic hint generation can be confident that disclosing even a limited number of accesses at a time can still lead to large performance gains.

When hints disclose many accesses in advance, they can be used for informed caching and clustering. Compared to prefetching alone, these techniques together reduced elapsed time for individual applications by up to an additional 36% or an average of 13% on a single disk. On large arrays, informed caching and clustering reduced elapsed time by up to 8% or an average of 3%. These gains are less dramatic because, as the bandwidth of the storage subsystem increases with larger array size, informed prefetching virtually eliminates stalls for hinted accesses. The only opportunity for informed caching to further improve performance is to reduce the number of accesses and therefore the CPU overhead of performing I/O. Managing the cache well and maximizing disk performance is most important when cache and disk resources are in short supply. Thus, TIP's informed caching and clustering see their greatest gains on a single disk.

In practice, most of the single-disk informed caching and clustering gains were realized for just two applications, Davidson and Postgres, 80% match, although Gnuld also benefitted to a lesser degree. All of these applications were able to disclose a significant amount of reuse many accesses in advance. Other applications, such as Agrep and XDataSlice reaccess little data. Still others, such as Sphinx, either reuse blocks immediately, for which LRU caching is effective, or don't give enough hints to capture widely-separated reuse. In particular, applications that access their files in ascending order on any single pass, even if not fully sequential because of strides, gain little from informed caching and clustering if they don't disclose hints about multiple passes over the data. Informed caching and clustering can only help workloads that provide both caching and clustering opportunities and hints that span these opportunities.

With regard specifically to optimizing the performance of individual disks, these experiments showed that informed prefetching's longer disk queues can reduce disk service time by up to 24% and informed clustering can reduce per-block service time by up to 22% on a single disk. On larger arrays, the impact of these gains can be small because, as noted above in the case of informed caching and clustering, informed prefetching masks stalls for hinted accesses. But, the impact of disk scheduling is further reduced on larger arrays because prefetches are spread over a larger number of disks resulting in a shorter queue at each drive. Shorter queues mean smaller reductions in disk service time from scheduling.

Informed prefetching, caching, clustering, and disk scheduling all require cache buffers to improve performance. The original goal in developing the framework for resource management based on cost-benefit analysis was to find a way to balance the use of cache buffers to take advantage of all of these optimizations. The results presented in this chapter show that cost-benefit analysis is indeed an effective mechanism for allocating cache buffers. With regard to informed prefetching, the experiments measuring stall as a function of prefetch depth show that the upper-bound prefetch horizon captures most of the potential stall reduction from both prefetching and disk scheduling without significantly cutting into cache performance, at least for a single application. Further, in the single-application experiments, informed caching and clustering always increase the number of reuse hits, reduce the number of blocks fetched from disk, and reduce the number of I/Os needed to fetch the blocks. Significantly, in no experiment, single-application or multiprogramming, did *TIP*'s application of these resource-demanding optimizations reduce performance. Never was prefetching so deep that losses in cache effectiveness offset prefetching gains. Never was informed caching so aggressive that reductions in LRU cache effectiveness for unhinted accesses more than offset its gains. Never did informed clustering reduce cache effectiveness enough to lead to an increase disk service time. On the contrary, TIP consistently achieved its stated goal of reducing application elapsed time in the single-application case and improving throughput in the multiprogramming case. TIP's cost-benefit buffer allocation effectively balanced the use of buffers for the several optimizations.

The cost-benefit estimations do not consider the overhead of the TIP implementation. But, any serious evaluation must. The overheads measured are noticeable, but the overall performance of the TIP system without hints is comparable to the standard Digital UNIX system. And, when TIP has hints, the performance gains in all the experiments presented here more than offset the losses due to these overheads. Still, there is room for improvement. The single largest CPU overhead in the TIP system is for LRU profiling. Thus, innovations in the algorithms for estimating the cost of shrinking the LRU queue offer the greatest opportunity for CPU overhead reduction. Finding more compact representations for hints that do not add significantly to CPU overhead offer the greatest opportunity for reducing the memory overhead of the system.

This performance evaluation of TIP shows that a system can take advantage of application disclosure of future accesses for prefetching, caching, clustering, and disk scheduling and dramatically reduce the elapsed time required to run a broad range of important I/O-intensive applications. Further, it shows that a system based on cost-benefit analysis can effectively manage cache resources to obtain substantial performance gains from all four of these optimizations.

Chapter 7

Generalizing the Results and Future Work

The experiments presented in Chapter 6 clearly demonstrated the utility of application disclosure of future accesses and the effectiveness resource management based on cost-benefit analysis. And yet, the experiments raised or left open a number of questions. In this chapter, I will explore some of these in more depth. Where possible, I will provide answers, or point to work that provides answers. But, in many cases, answers are unknown and I raise the questions here only to point to them as areas for future work.

In Section 7.1, I take up the question of why TIP performs well on a single, congested disk even though the prefetching benefit estimate is based on the assumption of no disk congestion. To answer this question, I develop a performance model that takes the number of disks into account. The model assumes a workload that is evenly balanced across the array and it neglects the effects of disk scheduling. But, through experiments with a synthetic application, I show that the model is useful for understanding TIP's performance on smaller arrays.

Next, in Section 7.2, I take up the question of what happens when, over time, processors get faster and the prefetch horizon grows to hundreds or thousands of accesses? Will the upper-bound prefetch horizon, $\hat{P}$, still be useful? I find that some of the simplifications of the current implementation may no longer be useful, and that finite bandwidth and the effects of disk scheduling may have to be accounted for explicitly to guarantee robustness. But, I argue that the fundamental framework is sound.

In Section 7.3, I discuss other recent work in this area and describe its relationship to this work. Specifically, in Section 7.3.1, I describe work that has shown that when prefetching to a fixed depth, runs of cached blocks or an unbalanced disk load can cause a disk to go idle. This work has shown that when, in the long term, the bandwidth of the storage subsystem is the performance bottleneck, these periods of disk idleness should be exploited for deeper prefetching. The result is a prefetching algorithm, *forestall*, that prefetches to a fixed depth when the disks are busy or when they are not the long-term bottleneck, but prefetches more deeply to take advantage of idleness when the disks are the bottleneck. The considerations in Section 7.2 apply to the near-term, fixed depth prefetching, but the deep prefetching usefully extends that work.

Then, in Section 7.3.2, I describe the results of a recent collaboration with Andrew Tomkins that showed how to incorporate the deep prefetching lessons of *forestall* into TIP's cost-benefit framework. The resulting system, TIPTOE, not only performs deep prefetching when appropriate to take advantage of disk idleness, but also incorporates the fact that a disk is a bottleneck into its estimate for the cost of ejecting a hinted block from that disk. A simulation study compares the performance of the cost-benefit approach to buffer allocation to an alternative based on the LRU algorithm.

Both *forestall* and TIPTOE rely on detailed knowledge of the layout of data on an array. In Section 7.3.3, I briefly discuss how to adapt TIPTOE to a world where data layout is unknown. I conclude that TIPTOE could be much more effective if disk array interfaces included a few minimum features.

Finally, in Section 7.4, I describe a number of areas of future work. These range from specific TIP implementation issues to broad areas of systems research that would help expand the usefulness of the informed prefetching and caching approach.

7.1 The impact of the no-congestion assumption

The performance model which served as the basis for the cost and benefit estimates developed in Chapter 4 makes certain simplifying assumptions. One of these is that there is never any disk congestion; that is, that disk requests never suffer any queuing delays

access number	time (1 time-step = T_{app} + T_{hit} + T_{driver})																				
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	I	⊝	⊝	⊝	⊝	⊗															
2	I	-	-	-	-	C	✕														
3	I	-	-	-	-	C		✕													
4						I	-	-	⊝	⊝	⊗										
5							I	-	-	-	-	⊗									
6								I	-	-	-	-	⊗								
7											I	-	-	⊝	⊝	⊗					
8												I	-	-	-	-	⊗				
9													I	-	-	-	-	⊗			
10																I	-	-	⊝	⊝	⊗

I : initiate prefetch - : prefetch in progress C : block arrives in cache ✕ : consume block ○ : stall

Figure 7.1. Average stall time when prefetching in parallel. This figure illustrates informed prefetching as a pipeline. In this example, three buffers are used to prefetch three blocks concurrently and T_{app} is assumed fixed. At time T=0, the application gives hints for all its accesses and then requests the first block. Prefetches for the first three accesses are initiated immediately. The first access stalls until the prefetch completes at T=5, at which point the data is consumed and the prefetch of the forth block begins. Accesses two and three proceed without stalls because the latency of prefetches for those accesses is overlapped with the latency of the first prefetch. But, the fourth access stalls for $T_{stall} = T_{disk} - 3(T_{app}+T_{hit}+T_{driver})$. The next two accesses don't stall, but the seventh does. The application settles into a pattern of stalling every third access.

(see Section 4.2.1). Clearly, this assumption is often violated, even in the experiments presented in Chapter 6. In this section, I will address the questions of what is the impact of this assumption, and why does the system perform so well even when the assumption is grossly violated?

7.1.1 The ideal model

Recall from Section 4.2.3 that the change in I/O service time that results from using one buffer to prefetch x instead of x-1 accesses in advance is given by Equation 4.9 which I repeat here:

$$\Delta T_{pf}(x) = T_{stall}(x) - T_{stall}(x-1) \quad . \tag{7.1}$$

Thus, the key problem is finding an expression for stall as a function of prefetch depth. I used a pipeline model for the servicing of prefetch requests which is first described in Figure 4.4 and is repeated here in Figure 7.1 to arrive at the following expression for stall time (Equation 4.12),

$$T_{stall}(x) = \frac{T_{disk} - x(T_{app} + T_{hit} + T_{driver})}{x} \tag{7.2}$$

$$= \frac{T_{disk}}{x} - (T_{app} + T_{hit} + T_{driver}) \ . \tag{7.3}$$

The key parameter that determines the behavior of the pipeline in Figure 7.1 is the number of prefetches that are serviced in parallel. Under the no-congestion assumption, all x of the prefetches are serviced in parallel which leads to the appearance of x in the denominator of Equation 7.3. But, if there are fewer than x disks, then it is impossible for all x prefetches to proceed in parallel, and the no-congestion assumption no longer holds.

Suppose there are d disks. At very large prefetching depths, $x \gg d$, the number of disks in the array ultimately limits prefetching parallelism. On the other hand, when the prefetching depth is smaller than the array size, $x < d$, prefetching depth limits parallelism. It is clear, then, that the prefetching parallelism, p, can never be more than the minimum of the prefetching depth and the array size, d, and we have

$$p \le min\{x, d\} \ . \tag{7.4}$$

In the ideal case, requests are perfectly balanced over the array and this expression can be rewritten as an equality. Taking the next step, we can use this ideal expression for the prefetching parallelism to rewrite Equation 7.3 in terms of prefetching depth and array size

$$T_{stall}(x, d) = \begin{cases} x = 0 & T_{disk} \\ x \ge 1 & \dfrac{T_{disk}}{min\{x, d\}} - (T_{app} + T_{hit} + T_{driver}) \end{cases} \ . \tag{7.5}$$

Note that under the no-congestion assumption, it is always possible to eliminate stall if prefetching is deep enough, but that, when this assumption is removed, it is no longer always possible to eliminate stall. Once all the disks are busy, no further increase in parallelism is possible and stall is minimized, at least if we neglect the impact of request sorting on disk access time as this model does. Under these conditions, the prefetch horizon is not the point at which stall is eliminated, but the point at which stall is minimized.

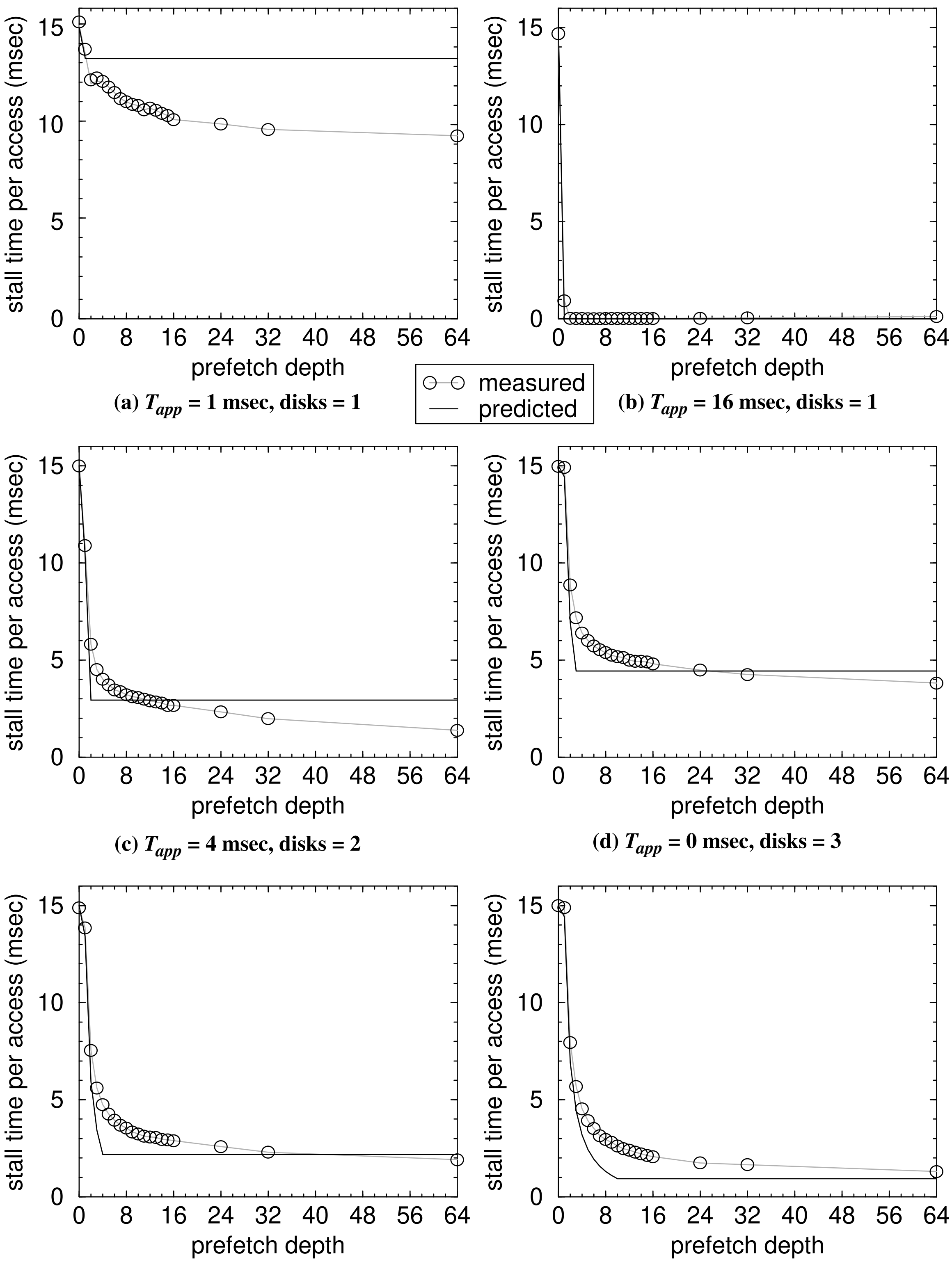
7.1.2 Experiments with a synthetic application

How well does this new, ideal model predict performance? In this section, I present the results of a number of experiments with the synthetic application used to determine the system parameters which was first described in Section 6.2. In these experiments, the application iterates twice over 2000 unique random blocks from a file striped over an array of from one to ten disks. Because this is more than the 1536 blocks in the cache, and because informed caching is turned off (the *TIP, no caching* configuration), this results in 4000 8 KByte disk reads. The file size is scaled with array size to a constant 32 MBytes per disk to keep the average disk access time roughly constant across array sizes. T_{app}, the application elapsed time between read calls, is either 0, 1, 4 or 16 milliseconds. Disk queues are sorted with the CSCAN algorithm.

Figure 7.2 compares the measured per-access stall time for this synthetic application to the stall predicted by Equation 7.5 for a selection of array sizes and values of T_{app}. The model successfully captures the general shape of the curves. However, at small prefetching depths it tends to underpredict stall and at large prefetching depths it tends to overpredict stall. These two discrepancies result from two different effects.

At small prefetching depths, the key factor in determining performance is the amount of prefetching parallelism. The ideal model assumes that increasing prefetch depth increases prefetching parallelism up to the limit imposed by the array size. In practice, because the accesses are chosen at random from the whole file, multiple accesses may go to the same disk while another disk stays idle. The effect is much like the memory contention that may occur in the interleaved memory banks of a supercomputer. This disk contention reduces the effective prefetching parallelism and consequently increases the stall relative to the predicted value.

At large prefetching depths, multiple prefetches are queued at individual disk drives. When the queued accesses are sorted according to the CSCAN scheduling algorithm, the average disk access time drops. The deeper the queue, the smaller the average access time. Naturally, this reduction in disk access time reduces the stall time experienced by the application and leads to the model's overestimation of stall time at large prefetching depths.



(a) T_{app} = 1 msec, disks = 1

(b) T_{app} = 16 msec, disks = 1

(c) T_{app} = 4 msec, disks = 2

(d) T_{app} = 0 msec, disks = 3

(e) T_{app} = 1 msec, disks = 4

(f) T_{app} = 0 msec, disks = 10

Figure 7.2. Measured per-access stall and stall predicted by the ideal model. This figure shows that for a broad range of application compute times and disk array sizes, the ideal model is a good overall predictor of stall time for a synthetic application. Nevertheless, for small prefetch depths, the model tends to underpredict stall, and for large prefetching depths, the model tends to overpredict stall.

To confirm that these two factors do indeed account for the discrepancies between the measured and predicted stall times, I ran some experiments in which these factors are eliminated. To eliminate disk contention, instead of choosing blocks randomly from the whole array, I cycled over the disks in the array and chose blocks randomly from within each disk. To eliminate the reduction in disk access time for deep queues, I used first-come-first-served (FCFS) instead of CSCAN disk scheduling.

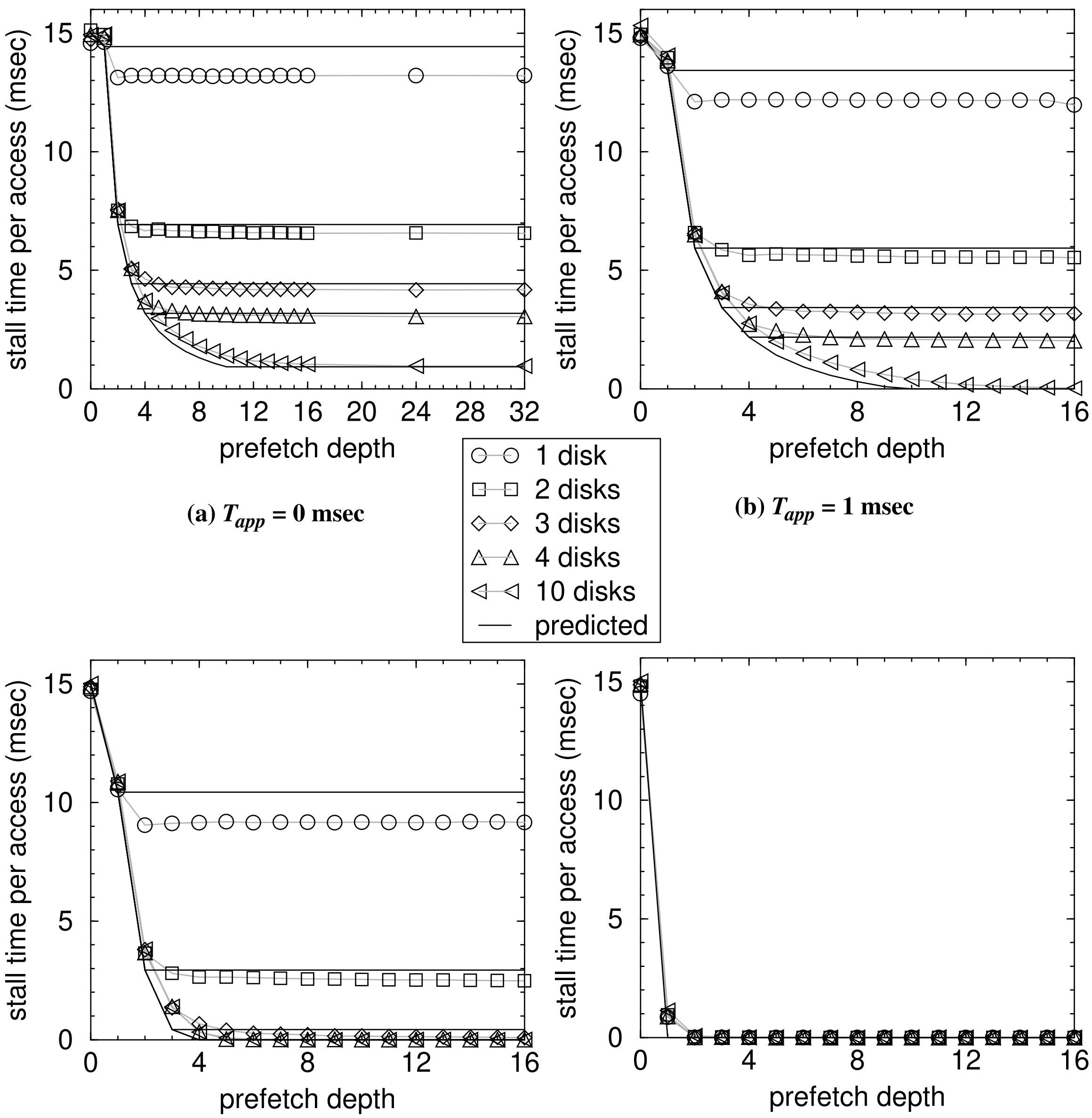
Figure 7.3 shows that excepting these two factors, the ideal model is an extremely good predictor of actual performance. The only remaining significant discrepancy is the stall on a single disk. Here we see that having a second I/O queued at the drive itself allows some of the SCSI and interrupt servicing overheads to be overlapped with the actual disk access.

The ideal model successfully captures the first-order effects, but its overestimation of parallelism on larger arrays and of disk access latency at large queue depths leads to significant differences between predicted and actual performance. A more accurate model would require better estimates of both parallelism and access times. To be most accurate, such estimates would have to take the specific workload into account. For example, the parallelism of a purely sequential workload is roughly the prefetch depth divided by the number of blocks in each stripe unit, 8 in these experiments. On the other hand, the access time for sequential accesses is much less than for random accesses. Developing a workload-dependent model is an interesting area for future research; it is beyond the scope of this dissertation which is limited to generating estimates for prefetching or ejecting blocks without considering the broader workload.

7.1.3 Analysis

Let us now return to the original question: why does TIP perform well on small numbers of disks even though it's prefetching model assumes that there are enough disks to avoid any disk congestion?

Intuitively, many people suspect that the lower I/O bandwidth of a smaller array would necessitate deeper prefetching. But, the results in Figure 7.3 show that, from the perspective of I/O parallelism, a smaller array requires prefetching less deeply, not more. For example, in Figure 7.3b, a prefetch depth of only two minimizes stall on a single disk,



(a) T_{app} = 0 msec

(b) T_{app} = 1 msec

(c) T_{app} = 4 msec

(d) T_{app} = 16 msec

Figure 7.3. Stall time when disk contention and disk scheduling are eliminated as factors. With these factors eliminated from the experiments, the ideal model becomes an excellent predictor of performance for all prefetching depths, application compute times, and array sizes. The only remaining discrepancy is on one disk where we see that having a second request queued at the drive allows the overlap of SCSI command processing with the actual disk access.

whereas a prefetch depth of about twelve is required to minimize stall on ten disks. Effectively, once the prefetch depth is great enough to keep all disks active, prefetching more deeply cannot further increase I/O parallelism; it takes fewer prefetches to keep fewer disks active. Thus, from the perspective of I/O parallelism, the assumption of no disk congestion, which is equivalent to assuming a very large array, led to a high estimate of the benefit of prefetching deeply. Thus, when TIP ran with a small array, its prefetching was deep enough to keep the array busy in most cases.[1]

Was TIP's prefetching therefore too deep? Consider stall on a single disk in Figure 7.2a. There, disk scheduling is not eliminated as a factor (as it is in Figure 7.3) and stall continues to drop at least to a prefetch depth of 64. On smaller arrays, there is no increase in I/O parallelism from prefetching more deeply, but there is a reduction in the average disk access time. When bandwidth is most limited, it is most important to maximize bandwidth by reducing the average access time through disk scheduling. Thus, from the perspective of disk scheduling, prefetching should be deepest on a single disk.

TIP performs well on all array sizes because its upper-bound prefetch horizon, $\hat{P}$, strikes a reasonable balance across array sizes. On larger arrays, it is sufficient to deliver the parallelism needed to mask stall. On smaller arrays, the prefetching depth not used for parallelism serves to reduce access time through disk scheduling.

7.2 Tightening the bound on prefetch depth

The applicability of a fixed, upper-bound prefetch horizon, $\hat{P}$, to all array sizes is fortuitous. But, because the use of $\hat{P}$ was a simplification to ease implementation (see Section 4.3.2), and because its applicability to scheduling on small arrays was not derived from the performance model, there is some concern that the use of a such a static prefetch horizon will not be robust in the face of the inevitable changes in system performance parameters. As CPU performance increases, the time to service a hit, T_{hit}, will shrink relative to the disk access time, T_{disk}. This could push $\hat{P}$ to hundreds or thousands of accesses. Alternatively, when prefetching from a remote server, access latencies could be quite high which could also increase $\hat{P}$. Will it still make sense to prefetch to that depth for all hinting processes?

On the current TIP testbed, $\hat{P}=73$. This is a small number compared to the cache size of 1536. $\hat{P}$ is an upper bound, and therefore larger than it needs to be in some cases, but there is little pressure to prefetch less deeply; using fewer buffers for prefetching would

[1] It turns out that caching can cause the array to go idle and that deeper prefetching can improve performance in such cases. I will get to this issue in Section 7.3.1.

not significantly increase cache performance in most cases. One could imagine putting pressure on the prefetch depth by running a large number of hinting applications simultaneously, but I have not performed experiments of this sort. However, if $\hat{P}$ grows faster than memory sizes and becomes a large portion of the cache, then it will be time to reconsider some of the simplifications that were made in the current implementation; the system will no longer have the luxury of being so generous with prefetch buffers. It will have to distinguish more accurately between those occasions when prefetching deeply is beneficial and those when it is not. Three factors could be considered to arrive at a more conservative, accurate estimate of the benefit of using buffers for prefetching.

First, instead of assuming that application CPU time, T_{app}, is negligible, and using a single system-wide prefetch horizon, the system could monitor application inter-request compute time and determine a per-application prefetch horizon from Equation 4.11. As shown by Figure 7.2b, the prefetch horizon shrinks dramatically when applications perform a significant amount of computation. To accommodate high degrees of multiprogramming, it may be useful to take the sharing of the processor into account and, instead of using the single process compute time, T_{app}, in the benefit equation, use the inter-access non-idle time which would include other processes' compute time. This would scale back the prefetch horizon for any single process when the processor is shared among many processes. The upper-bound prefetch horizon really applies to the system as a whole, not individual processes; multiple processes cannot consume data any faster than one.

Second, congestion and finite bandwidth which affect I/O parallelism, p from Equation 7.4, could be incorporated into the prefetching benefit estimate as suggested by Equation 7.5. The ideal model tells us that once there are enough outstanding prefetches to keep all disks busy, queuing additional requests does not increase I/O parallelism, assuming an evenly distributed workload and that caching does not let disks go idle.[2] From this perspective of I/O parallelism, the size of the array determines how deeply to prefetch. However, as mentioned above in Section 7.1.2, the most accurate estimate of parallelism as a function of queued prefetches would depend on the specific workload. Nevertheless, the

[2] When disks go idle either because the load is unbalanced, or because a run of cache hits leads to a lull in disk activity, recent work, which I discuss in Section 7.3, shows how to take advantage of that idleness for deep prefetching. In Section 7.3.3, I return to this simple model and suggest ways to apply those lessons here.

key point is that it is not necessary for the prefetching depth to scale with the performance disparity between processors and disks, it is only necessary for it to scale with the size of the attached array. A balanced system should have enough buffers to keep all disks busy.

Finally, if finite bandwidth considerations scale back prefetching depth, then a better estimate of the benefit of disk scheduling should also be included to earn buffers for deeper prefetching when it would reduce stall. This benefit depends primarily on the workload and the length of the queue at each disk, and so, as was the case for parallelism, the optimal prefetching depth does not scale directly with processor performance. However, a thorough study of disk scheduling in the presence of hints remains an area for future research. Here are two problems that need to be addressed.

The first is determining the impact on average access time, T_{disk}, of queuing additional requests. There is no point in depriving the cache of buffers to queue requests if doing so will not reduce average access time, and, ultimately, I/O service time. But, as Lesson 4 in Section 6.5 pointed out, sorting requests does not reduce access time if the requests are already in ascending order. On the other hand, sorting can significantly reduce the access time of random accesses. If it were possible to scan upcoming requests and estimate access time reduction as a function of queue depth, it would be possible to estimate the scheduling benefit of queuing additional requests.

The second, more subtle problem is related to the fact that the requests are ordered. If 1000 prefetches are queued at once, the device driver is free to completely reorder them. If the prefetch for the first read were sorted to the last position in the queue, then the application would block until all 1000 disk requests had completed. If the disk is the bottleneck, and the reordering would reduce the aggregate service time for all the requests by a factor of ten, then forcing the application to wait for all accesses to complete could be the right course to take. However, if the disk is not the bottleneck when prefetching, then reordering the first request to the end would remove the chance to overlap any computation with I/O; queuing 1000 requests would increase elapsed time, not reduce it.

One approach, known empirically to be effective [Cao96], is to issue prefetches in batches. A new batch of prefetches could be issued just before the disk completes servicing the previous batch. Batches limit reordering while providing a disk scheduler the opportunity to sort requests. If batches are used, the key problem becomes estimating the benefit of adding a buffer to increase the batch size. Developing such an estimate remains an area for future work.

7.3 Comparison with other systems

Although the experiments in Chapter 6 clearly demonstrate the effectiveness of the TIP informed prefetching and caching system, they leave open the possibility that some other system might make even better use of the application's disclosure hints. In a recent collaboration with Andrew Tomkins and other researchers, I endeavored to compare the cost-benefit approach to another proposed algorithm, and to extend the cost-benefit framework to include the dynamic load on the disk in its cost-benefit estimations. This extension is beyond the scope of this dissertation and neither it nor the experiments evaluating the extensions will be described in detail here. However, in this section, I will briefly summarize the results of this work and direct readers to other sources for more information.

The primary alternative to cost-benefit analysis is an approach developed by Pei Cao, Anna Karlin and other collaborators. Their approach was to decompose the problem into two sub-problems. The first is how to prefetch and cache for a single process that discloses all of its accesses. The second is how to allocate buffers globally among multiple processes. This decomposition led to two studies that explored alternative solutions to the two sub-problems.

7.3.1 Prefetching and caching for a single process

The *aggressive* algorithm was designed to prefetch and cache in the presence of complete knowledge of all future accesses [Cao95]. The algorithm is as follows: whenever the disk is free, eject the block whose next reference is furthest in the future to prefetch the block whose next reference is soonest, provided that the prefetched block will be referenced before the ejected one. The algorithm was developed with a single disk in mind which it uses to pace prefetching. Extended to multiple disks, whenever any disk is free, the algorithm prefetches the next-referenced block from that disk subject to the same ejection constraint as for the single disk. In an implementation of the algorithm [Cao96], prefetches are issued in batches of 16 to provide the opportunity for disk scheduling to reduce average access time.

In comparing the *cost-benefit* and *aggressive* algorithms, it is useful to note that in the single-process, complete-knowledge case, the two algorithms make very similar replacement decisions. The block with the lowest ejection cost is the one whose next access is furthest in the future and the block with the greatest prefetching benefit is the next missing block. Furthermore, the benefit of prefetching a block never exceeds the cost of ejecting a block that will be referenced before the ejected block. The key differences between the algorithms are (1) that *cost-benefit* only prefetches out to the prefetch horizon whereas *aggressive* may fill the cache with prefetches, (2) that *cost-benefit* initiates new prefetches as data are consumed whereas *aggressive* initiates prefetches when the disk is idle, and (3) that the hysteresis in the cost-benefit estimates means that a block is only ejected to prefetch another that is referenced substantially before the ejected block (here, substantial means many tens to hundreds of blocks) whereas *aggressive* has no such hysteresis.

A large collaboration, which included the developers of both algorithms, used trace-driven simulation to compare the performance of the *aggressive* and *cost-benefit* algorithms when all accesses are known in advance [Kimbrel96].[3] Also studied was a third algorithm, *reverse aggressive*, which was designed to take disk load into account when making ejection/prefetching decisions. The study found: that all three algorithms provided large benefits compared to a non-prefetching system; that *aggressive* sometimes out-performed *cost-benefit* on small arrays; that *cost-benefit* out-performed aggressive on large arrays; and that *reverse aggressive* performed about as well as any algorithm in all cases.

Aggressive outperformed *cost-benefit* on small arrays for benchmarks that had substantial reuse, such as repeated sequential access, or highly unbalanced disk loads. When there is high reuse, *cost-benefit* may cache long subsequences of accesses, for example, for Davidson's repeated sequential access of the same file (see Section 4.2.7 for details on how this occurs). When the application is accessing blocks in such a subsequence, *cost-benefit* may let the disk go idle because all blocks within the prefetch horizon are already

[3] The study actually used a variant of the *cost-benefit* algorithm called *fixed-horizon*. It prefetches a fixed distance into the future whereas *cost-benefit* scales back prefetching when prefetching would eject cached blocks that will be reaccessed soon. For the cache sizes studied, and when there is only one stream of hints and therefore one set of cached blocks, there are always blocks available that will not be reaccessed until far in the future, and *cost-benefit* is equivalent to *fixed-horizon* with the horizon set to the upper-bound prefetch horizon, $\hat{P}$.

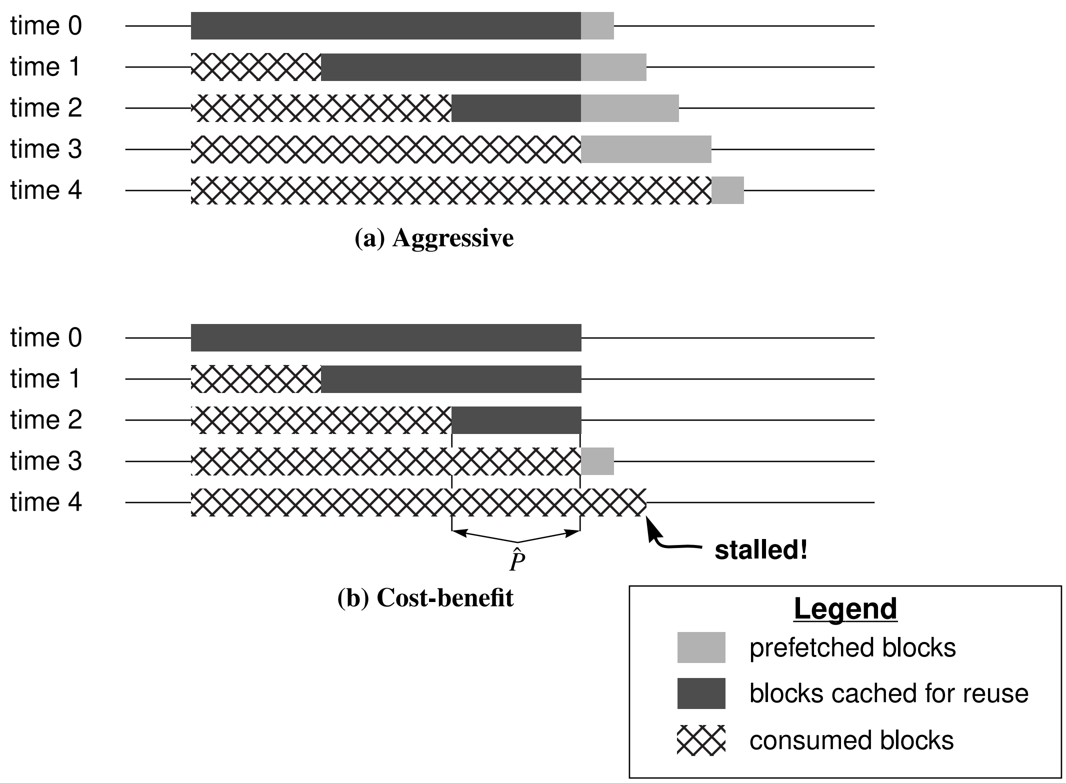
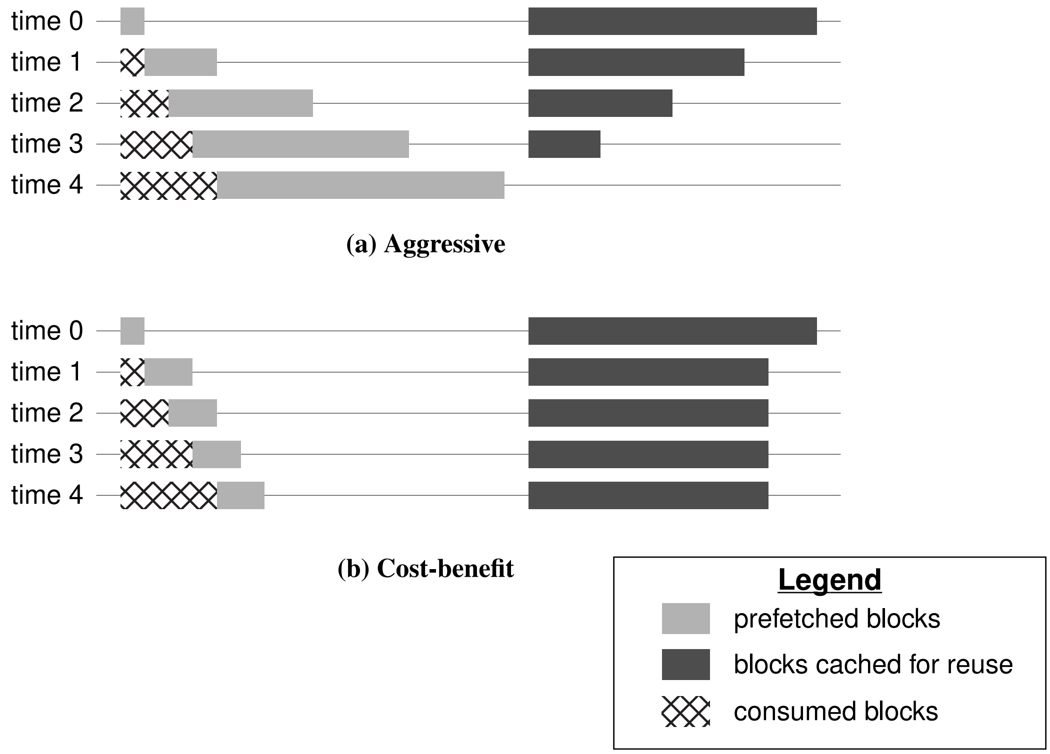


Figure 7.4. The lost opportunity of not prefetching during idleness on a small array. When an application is consuming a long sequence of cached data, the disk is not needed to service the application's immediate requests and the disk may go idle. Figure (a) shows how *aggressive* takes advantage of this idleness to prefetch as far into the future as possible. In contrast, Figure (b) shows how *cost-benefit*'s bounded prefetching lets disks stay idle. *Cost-benefit* does not resume prefetching until consumption is within the prefetch horizon, $\hat{P}$, at time 2. When bandwidth is limited, prefetching can't keep up with consumption, and the application stalls sooner than it would have had prefetching continued throughout.

cached as shown in Figure 7.4. *Aggressive* takes advantage of these lulls in disk activity to prefetch very far in advance. Similarly, when the disk load is unbalanced, *aggressive* takes advantage of lulls in activity on one disk to prefetch more deeply on that disk. *Aggressive* therefore maximized utilization of a single disk and I/O parallelism on an array. When the bandwidth of a single disk or small array is the performance bottleneck, or when an unbalanced load reduces the number of active disks and therefore the effective size of an array, *aggressive* can eliminate some stall and increase performance.

On larger arrays, *aggressive* used the high bandwidth available to flush the cache and fill it with prefetched data as shown in Figure 7.5. In contrast, *cost-benefit*, which assumes ample bandwidth, prefetches only deeply enough to eliminate stalls. Consequently, even though neither algorithm suffers significant stalls on larger arrays, *aggressive* performs



(a) Aggressive

(b) Cost-benefit

Figure 7.5. The wasted effort of prefetching too aggressively on a large array. *Aggressive* always ejects a cached block if it can take advantage of an idle disk to prefetch a closer block. Figure (a) shows how, when sufficient parallelism exists so there are often idle disks, *aggressive* flushes distant, cached blocks and fills the cache with prefetched blocks. In applications with significant reuse, this will incur unnecessary driver overhead by performing a disk access for each request which can have a significant impact on the elapsed time. Figure (b) shows how, in contrast, *cost-benefit*'s bounded prefetching retains the distant bocks for reuse but because there is enough bandwidth for prefetching to keep up with consumption, no stall ensues.

substantially more disk accesses than *cost-benefit*. These additional accesses incur the CPU overhead, T_{driver}, of performing an access which adds to the elapsed time for the *aggressive* algorithm.

The lesson from these experiments was that prefetching should be sensitive to the long-term load on the disks. When disk bandwidth is the constraining resource, prefetching during periods of transient disk idleness can avoid stalls far in the future and reduce elapsed time. On the other hand, when disk bandwidth is not the constraining resource, prefetching beyond the prefetch horizon can unnecessarily flush the cache and add CPU overhead to an application's elapsed time.

Unfortunately, although *reverse aggressive* was already sensitive to disk load, it had too much computational overhead to run on-line. However, the collaboration developed a new algorithm, *forestall*, which is sensitive to disk load and has reasonable overhead. The algorithm uses *fixed-horizon* for near-term prefetching and a disk-load sensitive algorithm for deep prefetching. The basic idea is to look forward in the hint sequence, estimating for each access when the disk will be able to perform the hinted read and when the application will issue the actual read request. If the disk will have no problem servicing the prefetch in time, then the prefetch may be delayed until the access reaches the prefetch horizon. On the other hand, if the request is anticipated before the disk will be able to service the prefetch, then the disk is *constrained* and prefetching from that disk should begin immediately. The simulation results showed that *forestall*'s performance for any benchmark on any array size ranges from only 2% slower to as much as 5.8% faster than the better of *aggressive* and *cost-benefit* on each configuration.

Sensitivity to disk load, such as that found in *forestall,* is not incorporated into the prefetching-benefit estimate in Chapter 4 because the scope of this dissertation is limited to estimates that are independent of both the layout of hinted data on disk and the current contents of the cache. The slower performance of *cost-benefit* on small disks and for unbalanced loads is the cost of these simplifications. However, as described below, recent extensions to this work show how to incorporate disk load not only into the prefetching-benefit estimate, but also into the ejection-cost estimate.

7.3.2 Allocating resources among multiple processes

A second comparative study investigated the second sub-problem: how to prefetch and cache when there are multiple processes and when not all accesses are hinted [Tomkins97]. The study compared using the time-tested LRU algorithm to make global allocation decisions to the cost-benefit approach.

Pei Cao showed how to adapt the LRU algorithm to partition the cache buffers among competing processes while using an algorithm such as *aggressive* or *forestall* to decide within a partition when to prefetch and what to eject [Cao96]. The idea is to maintain a global LRU queue with each buffer being owned by the process that last accessed it, and, instead of simply ejecting the block at the head of the LRU list, to give the owner of the head block the opportunity to hold onto that block and eject a different one of its blocks instead. She showed how swapping and placeholders could be used assure fairness and robustness in the face of poor replacement decisions. The resulting algorithm is called *LRU-SP*.

The cost-benefit approach does not explicitly partition the cache, but instead uses independent estimators and the common currency to make allocation decisions. Recent work has shown how to adapt the disk-load-sensitive *forestall* algorithm to the cost-benefit approach to build a modified TIP system called TIPTOE (TIP with temporal overload estimators) [Tomkins97, Tomkins97a]. The adaptation requires generating a benefit estimate in terms of the common currency.

The fundamental modeling insight of Chapter 4 remains the basis of TIPTOE: the benefit of prefetching is the reduction of stall. However, the *forestall* algorithm showed the superiority of a stall estimate that takes transient disk load into account. The *forestall* techniques can be used to detect constrained disks that will cause stall. A disk is *constrained* if it cannot service all prefetches in time even if prefetching non-stop starting immediately. Detection of constraint involves estimates of how quickly the application is consuming data and how quickly the disk can service prefetches. The TIPTOE work determined the change in stall that results from deep prefetching beyond the prefetch horizon on a constrained disk and also the change in buffer usage or bufferage required to obtain that reduction in stall. Dividing the one by the other produces the following common-currency benefit of prefetching a block x accesses in advance from a constrained disk:

$$\text{Benefit}_{deep\_pf} = \frac{\Delta T_{deep\_pf}(x)}{\Delta bufferage} = \frac{T_{disk}}{x} \quad . \tag{7.6}$$

Within the prefetch horizon, TIPTOE applies TIP's benefit estimate which, from Equation 4.24, is,

$$\text{Benefit}_{pf} = \frac{T_{disk}}{x(x-1)} \quad , \tag{7.7}$$

for $1 < x \leq \hat{P}$. The difference is the roughly factor of x in the denominator which occurs because Equation 7.6 estimates stall on a single constrained disk, whereas Equation 7.7 supposes that stall on one of x other accesses may mask stall for another.

Constrained disks also affect ejection decisions. Recall from Equation 4.16 in Section 4.2.4 that the cost of ejecting a hinted block is the additional CPU overhead of prefetching the ejected block back plus any stall that will be incurred on the eventual access. The same stall estimate used to compute the benefit of deep prefetching can also be used to refine the cost estimate for ejecting blocks from a constrained disk. Doing so leads TIPTOE to this equation for the cost of ejecting a block from a constrained disk that will be accessed beyond the prefetch horizon:

$$\mathrm{Cost}_{eject\_constrained} = \frac{\Delta T_{eject\_constrained}(x)}{\Delta bufferage} = \frac{T_{driver} + T_{disk}}{x} \quad . \tag{7.8}$$

TIPTOE uses TIP's estimate of the cost of ejection from unconstrained disks which, from Equation 4.31, is,

$$\mathrm{Cost}_{eject} = \frac{T_{driver}}{y - \hat{P}} \quad . \tag{7.9}$$

for $y > \hat{P}$. The essential difference is that TIPTOE anticipates a stall for a full disk access for a block ejected from a constrained disk whereas TIP assumes that the prefetch will not stall.

The multiple-process study used traces of the benchmark suite used in Chapter 6 to drive simulations of four algorithms: *LRU-SP* coupled with both the original *aggressive* algorithm and *forestall*, the *TIP* system described in this dissertation, and the *TIPTOE* system just described. Overall, the study found cost-benefit prefetching and caching to be somewhat better than LRU, reducing elapsed time from 5% to 8% over a broad range of combinations of two or three hinting and non-hinting applications. To first order, the LRU queue allocates buffers to processes in proportion to their rate of data consumption. But, rate of consumption is not a good indicator of data reuse. The study showed that the cost-benefit approach can take advantage of disclosure hints to cache the blocks whose reuse at a global level will be soonest and not waste buffers caching for low-reuse but high data rate applications. TIP's comparison of independent estimates in terms of the common currency allows such a global assessment of value to be made efficiently.

The single-process experiments that led to the invention of the *forestall* algorithm showed that it is fruitful to push beyond the cost and benefit estimates presented in this dissertation to arrive at estimates that are sensitive to transient disk load. However, the work that incorporated the *forestall* lessons into the cost-benefit framework and led to the TIPTOE system shows the fundamental soundness of the cost-benefit approach. TIPTOE reconfirms the analysis that uncovered the basic relationships described in Chapter 4, namely that the benefit of prefetching is reduced stall, and that the cost of ejecting a hinted block is the CPU overhead of prefetching it back plus any stall that will be incurred. The fact that the disk-sensitive stall estimates could be incorporated into the framework highlights the basic extensibility of the cost-benefit framework. Finally, the performance results demonstrate the superiority of the cost-benefit resource allocation over the conventional LRU algorithm.

7.3.3 Applying TIPTOE to arrays that hide data layout

Detecting an unbalanced load requires knowledge of the layout of data on the disk. What approach should be pursued when the interface to the storage subsystem hides these details from the file system?

The first step is eliminating unbalanced loads as a problem. I believe this is largely possible if the storage subsystem randomizes the assignment of stripe units to disks and accepts hints so that it can prefetch internally to smooth out transient load imbalances. I discuss other support the storage subsystems could provide for informed prefetching and caching in Section 7.4.7. A remaining issue is determining the number of outstanding prefetches needed to achieve a desired level of parallelism. If the prefetcher knows how much raw parallelism is available and that addresses are randomized, it should be possible to estimate the parallelism achieved by a set of outstanding prefetches. But, this remains an area for future work.

Assuming that the above techniques successfully eliminate load imbalance, deep prefetching is still desirable to take advantage of the idleness induced by runs of cache hits. But, if the system cannot prefetch from individual disks, it must prefetch from the array as a unit. I suspect that where the current model assumes that a storage device can perform one access in time T_{disk}, an extended model could treat an array of d disks as being capable of servicing d requests in time T_{disk}. Working out the details remains an area for future work.

7.4 Future work

In the previous sections, I highlighted a number of areas for researchers to extend this work. But, I have not yet had the chance to touch on all such areas. In this section, I summarize areas that could benefit from additional work. These are organized loosely from the most TIP-specific to the most general.

7.4.1 Implementation optimizations

In Section 6.6, I identified LRU profiling as the biggest CPU overhead in TIP. The largest part of the overhead of LRU profiling, accounting for about a 3% overhead on the file system, is overflowing buffers from one segment of the LRU queue to the next (see Section 5.2.6 for a description of this process). This operation is required to determine the queue position of buffers that are the target of a cache hit.

It might be possible to avoid this overhead by using a completely different approach to estimating the cost of ejecting an LRU buffer based on access numbers. Instead of breaking the queue into segments, each buffer could be stamped with the number, in a global count of accesses, of the access that is releasing the buffer to the tail of the queue. When a cache hit occurs, the difference between the buffer's stamp and the number of the current access would indicate how many accesses had passed since the buffer was last referenced. To assess the value of buffers in the LRU queue, a histogram of hits vs. number of accesses in the queue could be kept on a running basis. Given the number of accesses that the buffer at the head of the LRU queue has been in the queue, it may be possible to consult this histogram and arrive at an expected value for the number of accesses until that buffer will produce a cache hit. From that, it should be possible to arrive at an estimate of the cost of ejecting the block at the head of the list. Clearly, much work remains to turn this sketch of an idea into a practical LRU estimator.

7.4.2 Cluster-sensitive caching

Informed clustering builds efficient sequential accesses out of smaller, possibly random accesses. As pointed out in Lesson 7 in Section 6.5, replacement decisions affect the opportunity for clustered prefetches to refetch ejected blocks. A useful area for future research would be developing an estimator for the cost of ejection that was sensitive to clustering opportunities for the subsequent prefetch.

7.4.3 Protecting the unhinted cache from hinted blocks: the post-hint estimator

In TIP, all blocks are placed on the LRU queue after they have been accessed whether the access was hinted or not. This is the behavior of the unmodified system, and at the time of the original implementation, I had no good reason to implement a different policy. However, a consequence of this policy is that the LRU queue is shared between blocks that were hinted and unhinted. Effectively, hinters get their normal share of the LRU queue and then, if their hints disclose reuse, they take additional buffers from the LRU queue to cache their hinted data. Hints only increase an application's share of the cache, they do not decrease it. No similar mechanism lets unhinted blocks gain a larger share of the cache; unhinted blocks must always share the LRU queue with blocks that have been hinted and read.

In many cases, this policy works well. Some applications perform unhinted accesses to previously hinted blocks and rely on the LRU queue for cache hits. Also, sometimes hints for a second hinted access appear long after the first hinted access. For example, Gnuld issues hints for some of its passes only after the previous pass has completed. If the LRU cache did not hold on to these blocks, these reaccesses would not be cache hits.

On the other hand, many hinted blocks are never or seldom reaccessed, as in the case of Agrep or XDataSlice, or are reaccessed only according to hints which the system already has available, as in the case of Davidson or Postgres' outer-relation data accesses. When these applications are running alone, this is not a problem; the LRU estimator correctly discerns that there are few hits in the LRU queue, and the queue shrinks, leaving all of the buffers for hinted accesses. However, if a hinting and non-hinting application are running together, or if a single application interleaves hinted and unhinted accesses, the many unneeded hinted blocks dilute the effectiveness of the LRU queue for caching unhinted data.

The LRU caching behavior for hinted and unhinted blocks should be adaptive to perform well in both cases. One possible way to achieve this is to maintain a separate LRU estimator for hinted blocks. If hinted blocks are reaccessed or rehinted later, then the queue in such a post-hint estimator will grow. However, if unhinted blocks are reaccessed more often, then the original LRU queue, which is no longer diluted with unhinted blocks, will grow.

In my recent collaboration with Andrew Tomkins, we found, in simulation, that such a post-hint estimator could reduce elapsed time for a pair of applications by as much as 30%, and that the average reduction for a set of seven single-process and 11 multi-process experiments on a range of array sizes was nearly 5% [Tomkins97].

Implementing a post-hint estimator remains an area for future work. The only stumbling block I anticipate is the active region of the LRU queue (see Section 5.3.3 for a description of the active region). Because the active region in the LRU queue is protected, its buffers would be unavailable for caching post-hint blocks if these were sent to a separate post-hint queue. Consequently, a post-hint queue would have a smaller effective size than the current single queue. One way around this would be to send all buffers through the active region and only send buffers to the post-hint queue as they overflowed from the active to the inactive region of the queue.

7.4.4 Generalized estimators

A post-hint estimator is just one example of new estimators that could be added to the TIP system. Generalizing, the cost-benefit framework allows the system designer to identify subclasses of a resource, such as post-hint buffers, and then build an estimator for the value of allocating resources to that subclass. All that is required is that the estimated values be accurately expressed in terms of the common currency, and that the estimator support the required pick, query, update and bid operations described in Section 5.2.4. An interesting area for future work would be exploring what sorts of different subclasses might be useful in practice. Here are some possibilities:

- Currently, there is a separate estimator for every hinting process. Should each non-hinting process have its own LRU queue? Or, should there only be one estimator for each process group? If separating hinted from unhinted blocks is a good idea, perhaps separating the blocks from all processes would be beneficial.
- Heuristic prefetching has the advantage of not requiring any application modifications. Perhaps sequential readahead and more sophisticated heuristics could be embodied in prefetching estimators. If they were successful at predicting future accesses, they would merit buffers for prefetching.
- Virtual memory shares the same memory resource with the file buffer cache. The

two could be managed as a single resource with the addition of a virtual memory estimator.

7.4.5 The hint interface

The disclosure-hint interface described in Chapter 3 is simple and straight-forward. One could imagine many possible enhancements. In many cases, supporting such enhancements would require substantial extensions to the TIP system. Here are some examples.

The current interface only allows a process to give a single linear stream of hints about its own accesses. There are times when an application may not know the exact interleaving of its requests and so desire to create multiple parallel hint streams. Postgres could have used such a facility to give a hint for the second sequential read of the inner relation which occurred in parallel with the outer-relation accesses (see Figure 6.14).

In some cases, it may be desirable for one process to give hints about the accesses of another. For example, a C compiler can scan source code for '#include' directives, but it cannot know what files these included header files will themselves include. However, the *make* program could know all of the header files if the makefile included a full list of dependencies. In such a case, the *make* program could give hints about what files the C compiler will include. One challenge in supporting hints from multiple sources is recognizing when the system has received duplicate hints for the same accesses.

The current interface requires that all hinted accesses either occur or be cancelled. Some applications may not be able to deliver such accuracy; they may inadvertently skip some hinted accesses. The system could be made resilient to minor inaccuracies. However, as discussed in Section 3.2, such resiliency may complicate the programming model and be undesirable for that reason. Is there a way to add such resiliency without complicating the programming model?

Inaccurate hints are hints that are wrong. Imprecise hints are correct, but do not disclose full information. For example, Postgres was unable to give precise hints about its accesses to the inner-relation index. However, it would have been easy for Postgres to disclose that it was going to perform about 4000 random accesses to the index. Note that, in contrast to hints that advise the system to cache index blocks with high-priority, such an imprecise hint adheres to the disclosure hint model; it discloses what the application is going to do. An informed prefetching and caching system could use this information to cache at high priority. But, if enough buffers were available, it could also decide to prefetch the whole index with efficient, sequential accesses and then service the random requests from the cache. The disclosure hint gives the system the knowledge it needs to make such a decision.

A variant of the imprecise hint could be an exclusive-or hint which discloses that one of several files will be accessed. For example, if users are looking at a menu of files they could view, the system could hint that with high probability one of the files on the menu will be read.

Imprecise hints are incompatible with rigorous matching of hints to accesses. To help the system's hint matching stay synchronized with the application, it may be desirable for applications to insert markers in a hint stream. For example, an application could disclose that it will perform several hundred random accesses and then a 100-block sequential access. If the application could put a marker between the two hints, it could later inform the system that the random reads were over and the sequential accesses were about to begin by indicating that it had consumed all hints before the maker. These markers could also help the system stay synchronized with an interactive application that may need to abruptly change course.

7.4.6 Automatic hint generation

In Chapter 3, I showed that many applications can be annotated to give a substantial number of precise hints without too much difficulty. However, I am sure that many more programs would give hints if annotations could be added automatically. We have already seen that compilers can generate hints for scientific applications [Mowry96], but much remains to be done for irregular programs.

On a more speculative level, it might be possible to combine simple compiler techniques with access profiling. For example, it might be possible to augment Lei and Duchamp's access pattern trees [Lei97] with the disclosure of the arguments passed to a program to arrive at a more accurate prediction of accesses. Such a simple disclosure would not be hard to generate automatically. At a finer granularity, it might be possible to profile procedures or modules within a program and correlate accesses with the parameters passed when invoking the procedure.

7.4.7 Disk subsystem enhancements

In the course of building TIP and experimenting with its performance, it became clear that the disk subsystem could do more to support informed prefetching.

First, support for low-priority requests needs to be added to the SCSI command set. The current interface supports high-priority, head-of-line requests. One could imagine queuing demand requests at this high-priority, and queuing prefetch requests at normal priority. But, these high-priority requests are serviced in-order and so don't benefit from on-disk scheduling. It would be better to have separate class of low-priority requests that would benefit from scheduling, would not starve, and could be promoted from a low-priority to a high-priority request.

Second, storage subsystems, which often hide the details of data layout, should export an interface that allows file-system and application clients to optimize their workload for performance. The SCSI interface, for example, makes no guarantees about data placement, but there is a common understanding that blocks stored in sequential linear block addresses will tend to be stored in sequential locations on the disk surface. Further, blocks stored at close logical addresses will tend to be stored near each other on the disk surface so that seeks between them are short. The SCSI interface hides details of rotational position, but, through convention, exposes the most important features: sequentiality and proximity. The Logical Disk interface makes these two characteristics explicit [de Jonge93].

Disk arrays have a third important performance parameter, parallelism. Ideally, the interface to an array would expose all three parameters. File-system and application clients should know when an access will be sequential within one stripe unit on a single disk. They should be able to specify in some way that blocks should be near each other. They should be able to issue multiple requests and be reasonably confident that they are fully exploiting the parallelism of the array. And, if there are any important parity update optimizations, such as the large write optimization (see Section 2.1), clients should be able to exploit them. It is probably not possible or even desirable to expose the details of the layout, and consequently truly optimal performance will not be possible. But, the interface should expose enough information for clients to take advantage of the key performance characteristics of the subsystem. It would be optimal if clients could know that they were exploiting all ten disks in a ten-disk array, but it would be acceptable if they could at least be confident they were exploiting eight or nine of the ten.

Informed prefetching and caching systems need to know how to issue requests to maximize parallelism as discussed in Sections 7.1.2 and 7.3.3. The first step is to make sure clients know how many requests, on average, need to be queued at the subsystem to keep all disks utilized. But, workloads can be unbalanced. If the actual data layout is hidden, there is no way for the prefetching file system to know that its requests are generating an unbalanced load. Two mechanisms could avoid this problem. First, randomizing the assignment of stripe units to disks would reduce the likelihood that a workload is pathologically unbalanced. Second, the subsystem could itself accept hints about future accesses. Then the subsystem could prefetch more deeply when necessary to smooth out transient load imbalances.

7.4.8 A disk array for everyone

This dissertation has clearly demonstrated the utility of disk arrays for serial workloads when hints are available for informed prefetching. But, this is not a lesson just for data centers and expensive workstations; everyone could use a disk array, even desktop personal computers. Although many PC applications are not particularly I/O-intensive when running, almost all of them are during launch.

The problem is that current arrays are not cheap. Clearly, it does not make economic sense to attach a private 10-disk array to a 16 MByte PC. And yet, I believe PCs could use the bandwidth of such an array. The challenge, then, is to develop architectures for shared storage that can deliver the array performance at an affordable cost. If such an infrastructure became available, I suspect application writers would find a way to take advantage of it, and I/O-intensive PC applications would become commonplace.

7.5 Conclusion

In this chapter, I carried the analysis of the previous chapters one step further. I explored the impact of removing the assumption of no congestion and used that analysis to shed light on TIP's performance on small arrays where the assumption is clearly violated.

I showed that from the perspective of I/O parallelism, smaller arrays require less deep prefetching. But, deep prefetching on small arrays produces greater opportunities for scheduling to reduce the average disk access time. The upper-bound prefetch horizon, $\hat{P}$, works well because it is a reasonable compromise across array sizes.

Through time, the growing performance disparity between disks and processors will increase $\hat{P}$ and the time will come to drop some of the simplifications of the current implementation and include computation time, T_{app}, in benefit estimates. Further, by explicitly modeling the benefit of parallelism and disk scheduling, tighter bounds on the number of buffers required for prefetching should be obtainable.

I discussed recent related work that showed that the TIP prefetching benefit model works well when high bandwidth is available. But, that work also shows that when ample bandwidth is not available, and when unbalanced workloads or runs of cached blocks result in idle disks, the system should take advantage of that idleness to prefetch beyond the prefetch horizon. I described joint work with Andrew Tomkins in which we developed an estimator for the benefit of using buffers for such deep prefetching and incorporated it into the cost-benefit framework and so built TIPTOE.

I went on to discuss other possible extensions to the cost-benefit framework including a post-hint estimator to protect the LRU queue for unhinted accesses. Many other extension are possible.

The resilience of the cost-benefit framework to changing conditions and parameters as well as the many opportunities for extensions show that the fundamental approach adopted for TIP is sound. Cost-benefit analysis provides a durable, extensible framework for resource management.

Chapter 8

Conclusion

In the late eighties and early nineties, researchers argued that storage device parallelism was required for secondary storage performance to balance increasing processor performance and proposed Redundant Arrays of Inexpensive Disks (RAID) to provide that parallelism [Patterson88, Gibson92a]. Since then, the processor and storage performance trends they identified have continued. In my analysis of the four principal virtues of storage workloads that maximize performance (ASAP or avoidance, sequentiality, asynchrony, and parallelism), I again found that only parallelism could satisfy the demand for storage throughput. The other virtues help maximize the throughput of arrays of all sizes.

Unfortunately, many computer applications have serial I/O workloads that access only one disk at a time and are therefore unable to take advantage of disk-array parallelism. How can systems deliver the performance of parallel I/O to such applications? The key performance insight is that aggressive prefetching can do for serial reads what buffering does for serial writes: mask latency with asynchrony and expose parallelism for throughput. No longer should prefetching be viewed simply as a technique for overlapping I/O with computation; I/O parallelism is prefetching's greatest benefit.

How can such aggressive prefetching be achieved given the difficulty of predicting future accesses and the performance penalty of prefetching unneeded data? In this dissertation, I show that many applications can disclose their future file requests in hints, and that a system can use these hints to decide when and how much to prefetch, and what to cache. Formally, the thesis of this dissertation is that many important, I/O-bound applications can provide accurate hints about their future accesses, that operating system prefetching and caching according to these hints can substantially reduce application wall-clock elapsed time, and that run-time cost-benefit analysis can be the basis of effective resource management that balances the use of cache buffers for prefetching, clustering prefetches, caching for hinted accesses, and caching in a traditional LRU queue.

The proof of the thesis is in three steps. First, I develop techniques for annotating applications to give hints about their future file requests and show they can be used to annotate a suite of six important, I/O-intensive applications. Second, I develop a framework for resource management based on the run-time application of cost-benefit analysis and build an informed prefetching and caching system, called TIP, based on this framework. Finally, through measurements of the performance of the annotated applications running on TIP, I show that the operating system can use application hints to allocate resources and deliver the promised performance gains.

The vision set forth in this dissertation is that serial applications need only disclose their future accesses to obtain high-performance, parallel I/O. The implication is that applications need not be rewritten to be more parallel — often a difficult task. Nor need they manage a private buffer pool and asynchronous I/O requests. Nor need they be concerned with the number and timing of prefetches and how these might vary on different machines. Nor need they violate the modularity of the file system by controlling specific implementation actions. Instead, applications need only disclose in advance the requests they will make of the file system. Further, they can use the same terms that already define the file-system interface to disclose this information and thereby respect the modularity of the system. And, in doing so, they free the operating system to optimize resource usage globally because they provide the evidence for a policy decision.

The hope is that some day, application disclosures will be generated automatically. For the purposes of proving my thesis, however, it is sufficient to show that applications can be annotated by hand. In Chapter 3, I describe three techniques for annotating applications to give disclosure hints: in-line hints, loop duplication, and loop splitting. I then apply these techniques to annotate a broad suite of I/O-intensive applications which includes: Davidson computational physics, XDataSlice 3D scientific visualization, Gnuld object code linker, Sphinx speech recognition, Agrep text search, and two queries to the

Postgres relational database. Thus, I have shown that important and diverse applications can provide hints about their future accesses.

For the vision of high-performance, parallel I/O through application disclosure to become reality, I have to show how a system can use disclosure hints to deliver the promised performance. In this dissertation, I described a system that takes advantage of disclosure hints for four primary I/O optimizations:

1. informed caching to hold on to useful blocks and outperform LRU caching independent of prefetching;
2. informed clustering of multiple accesses into one larger access;
3. informed disk management that better schedules accesses to increase access efficiency; and,
4. informed prefetching to parallelize the disk workload and mask access latency.

All of these optimizations require use of the cache buffers already employed for traditional LRU caching. The primary challenge in automatically applying these optimizations is building a mechanism that can balance the use of cache buffers for all of these optimizations as well as LRU caching.

The thesis posits that run-time cost-benefit analysis can be the basis of a mechanism that effectively balances the use of cache buffers. The motivation for using cost-benefit analysis is two-fold. First, cost-benefit analysis provides a rational basis for allocating buffers that does not depend on the proper adjustment of a number of tuning knobs. Second, cost-benefit analysis is a general technique that should easily accommodate estimates for new resources, such as virtual memory or remote files, as well as improved estimates for resources already being managed. The thesis claims neither of these assertions, but they did guide me in my design.

In Chapter 4, I develop a framework for the run-time application of cost-benefit analysis to resource management. The framework includes three key components. First, independent cost and benefit estimates of the impact on I/O service time of ejecting a block or allocating a buffer for a prefetch avoid the need to consider all possible replacements and thereby limit the complexity of the system and ease the integration of new estimates into the system. Second, a common currency for the expression of cost and benefit estimates relates consumption of the cache buffer resource to the system goal of reducing I/O service time and enables the global comparison of the independently generated estimates. Finally, an allocation algorithm accepts the many independent estimates, expressed in terms of the common currency, scales them in proportion to their contribution to overall performance, and compares them at a global level to identify the replacement that would produce the greatest net reduction in I/O service time.

The first step in building an informed prefetching and caching system on this framework is developing independent cost and benefit estimates. Chapter 4 shows how to use a model of I/O performance to estimate the cost of ejecting a hinted block or taking a buffer from the LRU queue, and to estimate the benefit of using a buffer to service a demand miss or prefetch a block. It goes on to suggest modifications to the estimates to ease implementation. And it presents an efficient algorithm that takes advantage of the independent estimates to find the globally least-valuable block so that it can be ejected and its buffer reallocated to prefetch new data when doing so would reduce I/O service time. Chapter 5 describes the details of TIP, my implementation of informed prefetching and caching based on this framework.

The evaluation, in Chapter 6, of TIP's performance when running the suite of annotated applications shows that an operating system can indeed use disclosure hints to deliver the promised performance benefits. Figure 8.1 summarizes the results. Quantitatively, TIP reduced elapsed times for the benchmarks on a single disk by up to 50%, with an average of 28%. On a ten-disk array, TIP took advantage of parallelism to reduce elapsed time by up to 84%, with an average of 64%. When multiprogramming on a single disk, where resource contention is at its worst, TIP reduced elapsed time for pairs of applications by up to 48%, with an average of 37%. On a ten-disk array, TIP reduced elapsed time for pairs of applications by up to 73%, with an average of 58%. Further, all experiments with both a single application and when multiprogramming demonstrated a reduction in elapsed time.

All by themselves, these results argue that TIP must be allocating buffers effectively to optimize I/O performance. To further strengthen the argument, I measured TIP's cache and disk performance when hints were unavailable, when using hints for prefetching and

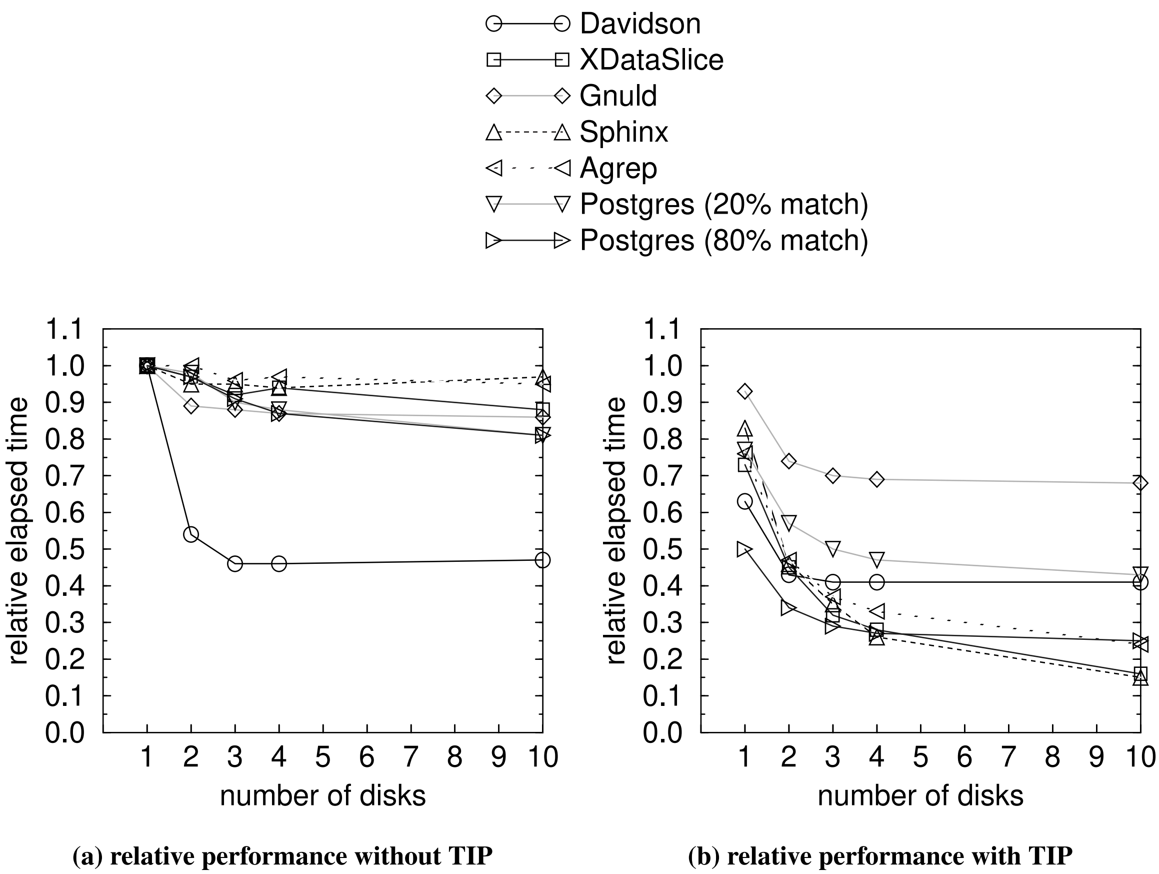


Figure 8.1. Elapsed time vs. array size with and without TIP. These graphs show elapsed time on multi-disk arrays as a fraction of elapsed time on a single disk without TIP for the suite of I/O-intensive applications. Graph (a), a reprise of Figure 2.2, shows that without informed prefetching and caching, only Davidson's sequential accesses benefit from array parallelism. Graph (b) shows that TIP's informed prefetching and caching can take advantage of array parallelism for all of the applications. On a sufficiently large array, all become compute bound. Further, most perform better on a single disk with TIP than they do on a ten-disk array without TIP.

clustering within the prefetch horizon, and when also using deep hints for informed clustering and caching. These experiments showed that the use of deep hints for informed clustering and caching could reduce application elapsed time by as much as 36% compared to prefetching alone. They also showed on a single disk that the longer disk queues generated by informed prefetching could reduce disk service time by up to 24%, and that informed clustering could reduce per-block service time by up to 22%. These specific results together with the elapsed time results demonstrate that TIP balances the use of buffers for all four I/O optimizations.

In recent joint work with Andrew Tomkins, we provide additional evidence in support of the claim that allocation based on cost-benefit analysis is effective by showing that, in simulation, the cost-benefit approach outperforms a competing approach which uses an LRU queue to allocate buffers at a global level [Tomkins97].

As described in Chapter 7, the same recent study and another [Kimbrel96], showed how to improve the specific prefetching-benefit and hinted-block-ejection-cost estimates proposed in this dissertation in Chapter 4. No claim is made that the estimators proposed here are optimal. To the contrary, my hope was that a framework for resource management based on cost-benefit analysis would be flexible and extensible. The fact that the improved estimators could be integrated into the existing framework argues that this is indeed the case. Recent work by David Rochberg extending TIP to prefetch from a distributed file system further strengthens this argument [Rochberg97].

Collectively, these results show that disclosure hints are a feasible and effective mechanism for passing I/O optimization information across the file-system interface that frees applications from the burden of buffer management and scheduling their own disk accesses. Further, they show that run-time cost-benefit analysis can be the basis of effective cache resource management that takes advantage of disclosure hints for informed prefetching and caching. Together, disclosure hints and cost-benefit based I/O optimization provide a powerful solution to the problem of delivering the scalable throughput of disk arrays to the many important applications with serial storage workloads.

Bibliography

[Accetta86] Accetta, M.J., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, June, 1986, pp. 93-112.

[Akyürek92] Akyürek, S., Salem, K., "Placing Replicated Data to Reduce Seek Delays," *Proceedings of the USENIX File System Conference*, May, 1992. Also available as Computer Science Technical Report CS-TR-2746, University of Maryland, August, 1991.

[Akyürek93] Akyürek, S., Salem, K., "Adaptive Block Rearrangement," *Proceedings of IEEE International Conference on Data Engineering*, April, 1993, pp. 182-189. An expanded version of the paper is available as Computer Science Technical Report CS-TR-2854, University of Maryland, March, 1992.

[Akyürek93a] Akyürek, S., Salem, K., "Adaptive Block Rearrangement under UNIX," *Proceedings of the USENIX Summer Technical Conference*, June, 1993, pp. 307-321. Also available as Computer Science Technical Report CS-TR-3054, University of Maryland, April , 1993.

[Amdahl67] "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *American Federation of Information Processing Societies (AFIPS) Spring Joint Conference*, V 30, Atlantic City, NJ, April 18-20, 1967, pp. 483-485.

[Anderson92] Anderson, T.E., Bershad, B.N., Lazowska, E.D. and Levy, H.M., “Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,” *ACM Transactions on Computer Systems (TOCS)*, V 10 (1), February, 1992, pp. 53-79.

[Baker91] Baker, M. G., Hartman, J. H., Kupfer, M.D., Shirriff, K.W., Ousterhout, J.K., “Measurements of a Distributed File System,” *Proceedings of the 13th Symposium on Operating System Principles (SOSP)*, Pacific Grove, CA, October, 1991, pp. 198-212.

[Cabrera91] Cabrera, L.-F., Long, D.D.E., “Swift: Using Distributed Disk Striping to Provide High I/O Data Rates,” *Computing Systems*, V 4 (4), 1991, pp. 405-439.

[Cao93] Cao, P., Lim, S.B., Venkataraman, S., Wilkes, J., "The TickerTAIP Parallel RAID Architecture," *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, May, 1993, pp. 52-63. Available from http://www.cs.wisc.edu/~cao/publications.html.

[Cao94] Cao, P., Felten, E.W., Li, K., “Application-Controlled File Caching Policies,” *Proceedings of the Summer 1994 USENIX Conference*, Boston, MA, June 6-10, 1994, pp. 171-182. Available from http://www.cs.wisc.edu/~cao/publications.html.

[Cao94a] Cao, P., Felten, E.W., Li, K., “Implementation and Performance of Application-Controlled File Caching,” *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, November, 1994, pp.165-178. Available from http://www.cs.wisc.edu/~cao/publications.html.

[Cao95] Cao, P., Felten, E.W., Karlin, A., Li, K., “A Study of Integrated Prefetching and Caching Strategies,” *Proceedings of the Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Ottawa, Canada, May, 1995, pp. 188-197. Available from http://www.cs.wisc.edu/~cao/publications.html.

[Cao96] Cao, P., Felten, E.W., Karlin, A., Li, K., “Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching and Disk Scheduling,” *ACM Transaction on Computer Systems (TOCS)*, V 14 (4), November, 1996, pp. 311-343. Available from http://www.cs.wisc.edu/~cao/publications.html.

[Chen93] Chen, C-M. M., Roussopoulos, N., "Adaptive Database Buffer Allocation Using Query Feedback," *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB),* Dublin, Ireland, 1993, pp. 342-353.

[Chen96] Chen, P.M., Ng, W.T., Chandra, S., Aycock, C., Rajamani, G., Lowell, D., "The Rio File Cache: Surviving Operating System Crashes," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1-5, 1996, pp. 74-83.

[Chou85] Chou, H.T., DeWitt, D.J., "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB)*, Stockholm, 1985, pp. 127-141.

[Corbett95] Corbettt, P.F., Feitelson, D.G., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.-P., Snir, M., Traversat, B., Wong, P., "Overview of the MPI-IO Parallel I/O Interface," *IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, April, 1995, pp. 1-15.

[Corbett96] Corbett, P.F., Feitelson, D.G., "*The Vesta Parallel File System*," *ACM Transactions on Computer Systems (TOCS)*, V 14 (3), August, 1996, pp. 225-264.

[Cornell89] Cornell, D. W., Yu, P. S., "Integration of Buffer Management and Query Optimization in Relational Database Environment," *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB)*, Amsterdam, August 1989, pp. 247-255.

[Cray93] Cray Research, *Advanced I/O User's Guide SG-3076 8.0*, Cray Research, Inc., Order desk phone number (612) 683-5907, Mendota Heights, MN, 1993.

[Curewitz93] Curewitz, K.M., Krishnan, P., Vitter, J.S., “Practical Prefetching via Data Compression,” *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, Washington, DC, May 1993, pp. 257-266.

[de Jonge93] de Jonge, W., Kaashoek, M.F., Hsieh, W.C., “The Logical Disk: A New Approach to Improving File Systems,” *Proceedings of the Fourteenth ACM Symposium on Operating Systems Priciples (SOSP)*, Asheville, NC, December 5-8, 1993, pp. 15-28.

[del Rosario94] del Rosario, J.M., Choudhary, A., "High Performance I/O for Parallel Computers: Problems and Prospects," *IEEE Computer*, V 27 (3), March, 1994, pp. 59-68.

[Denning67] Denning, P.J., “Effects of Scheduling on File Memory Operations,” *American Federation of Information Processing Societies (AFIPS) Spring Joint Conference*, V 30, Atlantic City, NJ, April 18-20, 1967, pp. 9-21.

[Dibble88] Dibble, P., Scott, M., Ellis, C., "Bridge: A High-Performance File System for Parallel Processors," *Proceedings of the Eighth International Conference on Distributed Computing Systems*, San Jose, CA, June, 1988, pp. 154-161.

[Ebling94] Ebling, M.R., Mummert, L.B., Steere, D.C., “Overcoming the Network Bottleneck in Mobile Computing,” *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December, 1994.

[Engler95] Engler, D.R., Kaashoek, M.F., O’Toole, Jr., J., “Exokernel: An Operating System Architecture for Application-Level Resource Management,” *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, December 3-6, 1995, pp. 251-266.

[Eustace95] Eustace, A., Srivastava, A., “ATOM: a Flexible Interface for Building High Performance Program Analysis Tools,” *Proceedings USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 1995, pp. 303-314.

[Ganger97] Ganger, G.R., Kaashoek, M.F., "Embedded Inodes and Explicit Groupiing: Exploiting Disk Bandwidth for Small File," *Proceedings of the Winter 1997 USENIX Technical Conference*, January, 1997, pp. 1-17.

[Geist87] Geist, R., Daniel, S., "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems*, V 5 (1), February, 1987, pp. 77-92.

[Gibson92] Gibson, G. A., Patterson, R. H., Satyanarayanan, M., "Disk Reads with DRAM Latency," *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE, Key Biscayne, FL, April, 1992, pp. 126-131. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Gibson92a] Gibson, G., "Redundant Disk Arrays: Reliable, Parallel Secondary Storage," Ph. D. thesis, MIT Press, Cambridge, MA, 1992.

[Gibson97] Gibson, Garth; Nagle, Deavid F.; Amiri, Khalil; Chang, Fay W.; Feinberg, Eugene M.; Gobioff, Howard; Lee, Chen; Ozceri, Berend; Riedel, Erik; Rochberg, David; Zelenka, Jim, "File Server Scaling with Network-Attached Secure Disks," *Proceedings of the 1997 ACM Sigmetrics International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Seattle, WA, June 15-18, 1997, pp. 272-284. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Griffioen93] Griffioen, J., Appleton, R., "Automatic Prefetching in a WAN," *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton, NJ, October, 6, 1993, pp. 8-12. Available from http://www.dcs.uky.edu/~griff/papers/mybib.html.

[Griffioen94] Griffioen, J., Appleton, R., "Reducing File System Latency using a Predictive Approach," *Proceedings of the 1994 Summer USENIX Conference*, Boston, MA, 1994. Available from http://www.dcs.uky.edu/~griff/papers/mybib.html.

[Griffioen95] Griffioen, J., Appleton, R., "Performance Measurements of Automatic Prefetching," *Proceedings of the International Society for Computers and their Applications (ISCA) International Conference on Parallel and Distributed Computing Sys-*

tems, Orlando, FL, October, 1995, pp. 165-170. Available from http://www.dcs.uky.edu/~griff/papers/mybib.html.

[Griffioen96] Griffioen, J., Appleton, R., “The Design, Implementation, and Evaluation of a Predictive Caching File System,” Technical Report CS-264-96, Department of Computer Science, University of Kentucky, June, 1996. Available from http://www.dcs.uky.edu/~griff/papers/mybib.html.

[Grimshaw91] Grimshaw, A.S., Loyot Jr., E.C., “ELFS: Object-Oriented Extensible File Systems,” Technical Report TR-91-14, Computer Science, University of Virginia, 1991.

[Grochowski96] Grochowski, E.G., Hoyt, R.F., “Future Trends in Hard Disk Drives,” *IEEE Transactions on Magnetics*, V 32 (3), May, 1996, pp. 1850-1854.

[Hartman93] Hartman, J.H., Ousterhout, J.K., “The Zebra Striped Network File System,” *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP)*, Ashville, NC, December, 1993, pp. 29-43.

[Harty92] Harty, K., Cheriton, D.R., “Application-Controlled Physical Memory Using External Page-Cache Management,” *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, MA, October, 1992, pp. 187-199.

[Haskin96] Haskin, R., Schmuck, F., "The Tiger Shark File System," *Proceedings of IEEE 1996 Spring COMPCON*, Santa Clara, CA, February, 1996.

[Hennessy96] Hennessy, J.L., Patterson, D.A., *Computer Architecture A Quantitative Approach*, 2nd ed., Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[IDC96] International Data Corporation, “1996 Worldwide Disk Subsystems Market Review and Forecast,” International Data Corporation publication number IDC #11365, June, 1996.

[Jacobson91] Jacobson, D.M. and Wilkes,J., “Disk Scheduling Algorithms Based on Rotational Position,” Technical Report HPL-CSP-91-7, Hewlett-Packard Laboratories, February, 1991.

[Jain91] Jain, Raj, *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, New York, ISBN 0-471-50336-3, 1991.

[Kiczales] Kiczales, G., “Towards a New Model of Abstraction in the Engineering of Software,” *Proceedings of the IMSA `92 Workshop on Reflection and Meta-level Architectures*, 1992.

[Kimbrel96] Kimbrel, T., Tomkins, A., Patterson, R.H., Bershad, B., Cao, P., Felten, E.W., Gibson, G.A., Karlin, A.R., Li, K., “A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching,” *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 28-31, 1996, pp. 19-34. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Kistler93] Kistler, J.J., “Disconnected Operation in a Distributed File System,” Ph. D. thesis, Technical Report CMU-CS-93-156, School of Computer Science, Carnegie Mellon University, 1993. Available from http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda/Web/docs-coda.html.

[Korner90] Korner, K., “Intelligent Caching for Remote File Service, *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990, pp. 220-226.

[Kotz90] Kotz, D., Ellis, C.S., “Prefetching in File Systems for MIMD Multiprocessors,” *IEEE Transactions on Parallel and Distributed Systems*, V 1 (2), April, 1990, pp. 218-230.

[Kotz91] Kotz, D., "Prefetching and Caching Techniques in File Systems for MIMD Multiprocessors,” Ph. D. thesis, Technical Report CS-1991-16, Department of Computer Science, Duke University, 1991.

[Kotz93] Kotz, D., Ellis, C.S., "Practical Prefetching Techniques for Multiprocessor File Systems," *Distributed and Parallel Databases*, V 1 (1), January, 1993, pp. 33-51.

[Kotz94] Kotz, D., "Disk-directed I/O for MIMD Multiprocessors," *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, Monterey, CA, November, 1994, pp. 61-74.

[Krieger94] Krieger, O., "HFS: A Flexible File System for Shared Memory Multiprocessors," Ph. D. thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

[Kroeger96] Kroeger, T.M., Long, D.D.E., "Predicting File System Actions from Prior Events," *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 22-26, 1996, pp. 319-328.

[Lampson83] Lampson, B.W., "Hints for Computer System Design," *Proceedings of the 9th Symposium on Operating System Principles (SOSP),* Bretton Woods, N.H., 1983, pp. 33-48.

[Lee96] Lee, E.K., Thekkath, C.A., "Petal: Distributed Virtual Disks," *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1-5, 1996, pp. 84-92.

[Lee90] Lee, K.-F., Hon, H.-W., Reddy, R., "An Overview of the SPHINX Speech Recognition System," *IEEE Transactions on Acoustics, Speech and Signal Processing*, V 38 (1), January, 1990, pp. 35-45.

[Lei97] Lei, H., Duchamp, D., "An Analytical Approach to File Prefetching," *Proceedings 1997 USENIX Annual Technical Conference*, January, 1997. Also available from http://www.cs.columbia.edu/~lei/resume.html.

[Madhyastha97] Madhyastha, T.M., Reed, D.A., “Input/Output Access Pattern Classification Using Hidden Markov Models,” *Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)*, November, 1997.

[Mattson70] Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L., “Evaluation Techniques for Storage Hierarchies,” *IBM Systems Journal*, V 9 (2), 1970, pp. 78-117.

[McKellar69] McKellar, A.C., Coffman, Jr., E.G., “Organizing Matrices and Matrix Operations for Paged Memory Systems,” *Communications of the ACM*, V 12 (3), March 1969, pp. 153-165.

[McKusick84] McKusick, M.K., Joy, W.J., Leffler, S.J., Fabry, R.S., “A Fast File System for UNIX,” *ACM Transactions on Computer Systems*, V 2 (3), August 1984, pp. 181-197.

[McVoy91] McVoy, L.W., Kleiman, S.R., “Extent-like Performance from a UNIX File System,” *Proceedings of the Winter 1991 USENIX Conference*, Dallas, TX, January, 1991, pp. 33-43.

[Microsoft93] Microsoft Corporation, “Microsoft Windows & MS-DOS 6 User’s Guide,” Microsoft Press, Redmond, WA, 1993.

[Mogi94] Mogi, K., Kitsuregawa, M., “Dynamic Parity Stripe Reorganizations for RAID5 Disk Arrays,” *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, TX, September 28-30, 1994, pp. 17-26.

[Mowry96] Mowry, T., Demke, A., Krieger, O., “Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications,” *Proceedings of Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 28-31, 1996, pp. 3-17.

[NCSA89] National Center for Supercomputing Applications. “XDataSlice for the X Window System,” http://www.ncsa.uiuc.edu/, Univ. of Illinois at Urbana-Champaign, 1989.

[Ng91] Ng, R., Faloutsos, C., Sellis, T., "Flexible Buffer Allocation Based on Marginal Gains," *Proceedings of the 1991 ACM Conference on Management of Data (SIGMOD)*, pp. 387-396.

[Ousterhout85] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., Thompson, J.G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th Symposium on Operating System Principles (SOSP),* Orcas Island, WA, December, 1985, pp. 15-24.

[Ousterhout89] Ousterhout, J., Douglis, F., "Beating the I/O Bottleneck: A Case for Log-Structured File Systems," *ACM Operating Systems Review*, V 23 (1), January, 1989, pp. 11-28. Also available as Technical Report UCB/CSD 88/467, University of California-Berkeley, 1988.

[Palmer90] Palmer, M.L., Zdonik, S.B., "Predictive Caching," Technical Report CS-90-29, Computer Science, Brown University, 1990.

[Palmer91] Palmer, M.L., Zdonik, S.B., "FIDO: A Cache that Learns to Fetch," Technical Report CS-90-15, Computer Science, Brown University, 1991.

[Parsons97] Parsons, I., Unrau, R., Schaeffer, J., Szafron, D., "PI/OT: Parallel I/O Templates," *Parallel Computing*, V 23 (4-5), June, 1997, pp. 543-570.

[Patterson88] Patterson, D., Gibson, G., Katz, R., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the 1988 ACM Conference on Management of Data (SIGMOD)*, Chicago, IL, June, 1988, pp. 109-116.

[Patterson93] Patterson, R.H., Gibson, G.A., Satyanarayanan, M., "A Status Report on Research in Transparent Informed Prefetching," *ACM Operating Systems Review*, V 27 (2), April, 1993, pp. 21-34. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Patterson94] Patterson, R.H., Gibson, G.A., "Exposing I/O Concurrency with Informed Prefetching," *Proceedings of the 3rd IEEE International Conference on Parallel and*

Distributed Information Systems (PDIS), Austin, TX, September 28-30, 1994, pp. 7-16. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Patterson95] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D., Zelenka, J., "Informed Prefetching and Caching," *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain Resort, CO, December 3-6, 1995, pp. 79-95. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[RAB96] Raid Advisory Board, *The RAIDbook, A Source Book for Disk Array Technology*, 5th ed., edited by Paul Massiglia, ISBN 1-879936-90-9, The RAID Advisory Board, 13 Marie Lane, St. Peter, MN, 1996.

[Peacock88] Peacock, J.K., "The Counterpoint Fast File System," *Proceedings of the USENIX Winter Conference*, Dallas, TX, February 9-12, 1988, pp. 243-249.

[Rochberg97] Rochberg, D., Gibson, G., "Prefetching Over a Network: Early Experience with CTIP," *ACM SIGMETRICS Performance Evaluation Review*, V 25 (3), December, 1997, pp. 29-36. Also available at http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Rosenblum92] Rosenblum, M., Ousterhout, J.K., "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, V 10 (1), February, 1992, pp. 26-52.

[Rozier88] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., Neuhauser, W., "CHORUS Distributed Operating System," *Computing Systems*, V 1 (4), 1988, pp. 305-370.

[Ruemmler91] Ruemmler, C., Wilkes, J., "Disk Shuffling," Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories, October, 1991.

[Sacco82] Sacco, G.M., Schkolnick, M., "A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model," *Proceedings of the 8th Inter-*

national Conference on Very Large Data Bases (VLDB), September, 1982, pp. 257-262.

[Salem86] Salem, K. Garcia-Molina, H., “Disk Striping,” *Proceedings of the 2nd IEEE International Conference on Data Engineering*, 1986.

[Seltzer90] Seltzer, M. I., Chen, P. M., Ousterhout, J. K., “Disk Scheduling Revisted,” *Proceedings of the Winter 1990 USENIX Technical Conference*, Washinton, DC, January, 1990.

[Smith78] Smith, A.J., “Sequentiality and Prefetching in Database Systems,” *ACM Transactions on Database Systems*, V 3 (3), September, 1978, pp. 223-247.

[Smith85] Smith, A.J., “Disk Cache — Miss Ratio Analysis and Design Considerations,” *ACM Transactions on Computer Systems*, V 3 (3), August 1985, pp. 161-203.

[Staelin90] Staelin, C., Garcia-Molina, H., “Clustering Active Disk Data to Improve Disk Performance,” Technical Report CS-TR-283-90, Computer Science, Princeton University, September, 1990.

[Stathopoulos94] Stathopoulos, A., Fischer, C.F., “A Davidson Program for Finding a Few Selected Extreme Eigenpairs of a Large, Sparse, Real, Symmetric Matrix,” *Computer Physics Communications,* V 79, 1994, pp. 268-290.

[Steere97] Steere, D.C., “Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency,” *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 5-8, 1997, pp. 252-263.

[Stodolsky93] Stodolsky, D., Gibson, G., Holland, M., “Parity Logging: Overcoming the Small Write Problem in Redundant Disk Arrays,” *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, May, 1993, pp. 64-75. Available at http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Stonebraker86] Stonebraker, M., Rowe, L, “The Design of Postgres,” *Proceedings of 1986 ACM International Conference on Management of Data (SIGMOD)*, Washington, DC, May 28-30, 1986.

[Stonebraker90] Stonebraker, M., Rowe, L.A., Hirohama, M., “The implementation of POSTGRES,” *IEEE Transactions on Knowledge and Data Engineering*, V 2 (1), March, 1990, pp. 125-142.

[Sun88] Sun Microsystems, Inc., *Sun OS Reference Manual*, Part Number 800-1751-10, Revision A, May 9, 1988.

[Tait91] Tait, C.D., Duchamp, D., “Detection and Exploitation of File Working Sets,” *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, TX, May, 1991, pp. 2-9.

[Terry87] Terry, D.B., “Caching Hints in Distributed Systems,” *IEEE Transactions on Software Engineering*, V SE-13 (1), January, 1987.

[Thekkath97] Thekkath, C.A., Mann, T., Lee, E.K., "Frangipani: A Scalable Distributed File System," *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 5-8, 1997, pp. 224-237.

[Tomkins97] Tomkins, A., Patterson, R.H., Gibson, G.A., “Informed Multi-Process Prefetching and Caching,” *Proceedings of the 1997 ACM Sigmetrics International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS),* Seattle, WA, June 15-18, 1997, pp. 100-114. Available from http://www.pdl.cs.cmu.edu/Publications/publications.html.

[Tomkins97a] Tomkins, A., Ph. D. thesis, Technical Report CMU-CS-97-181, School of Computer Science, Carnegie Mellon University, 1997. Available from http://www.cs.cmu.edu/~andrewt.

[Trivedi79] Trivedi, K.S., “An Analysis of Prepaging”, *Computing*, V 22 (3), 1979, pp. 191-210.

[Vitter91] Vitter, J.S., Krishnan, P., "Optimal Prefetching via Data Compression," Technical Report CS-91-46, Computer Science, Brown University, July, 1991. An extended abstract appears in *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, Puerto Rico, October, 1991.

[Vongsathorn90] Vongsathorn, P., Carson, S.D., "A System for Adaptive Disk Rearrangement," *Software - Practice and Experience* (UK), V 20 (3), March 1990, pp. 225-242.

[Wilkes96] Wilkes, J., Golding, R., Staelin, C., Sullivan, T., "The HP AutoRAID Hierarchical Storage System," *ACM Transactions on Computer Systems (TOCS)*, V 14 (1), February 1996, pp. 108-136.

[Wolfe96] Wolfe, M.J., *High Performance Compilers for Parallel Computing*, Addison-Wesley, Redwood City, CA, 1996.

[Worthington94] Worthington, B. L., Ganger, G. R., Patt, Y. N., "Scheduling algorithms for modern disk drives," *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, May, 1994, pp. 241-51.

[Wu92] Wu, S. and Manber, U. "AGREP - a Fast Approximate Pattern-Matching Tool," *Proceedings of the 1992 Winter USENIX Conference*, San Francisco, CA, January, 1992, pp. 20-24.