APPLICATION-CONTROLLED FILE CACHING AND PREFETCHING

Pei Cao

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

January 1996

© Copyright by Pei Cao 1996
All Rights Reserved

Abstract

As disk performance continues to lag behind microprocessors and memory systems, the file system is increasingly becoming the bottleneck for many applications. This dissertation demonstrates that with appropriate mechanisms and policies, application-controlled file caching and prefetching can significantly improve the file I/O performance of applications in both single and multiple process cases.

In traditional file systems, the kernel controls file cache replacement and prefetching using fixed policies, with no input from user processes. For many applications, this has resulted in poor utilization of the file cache and little overlapping between I/O and computation. The challenges are to design a scheme that allows applications to control the management of their file cache, and to design algorithms for the kernel to coordinate the use of shared resources so that the performance of the whole system is guaranteed to improve.

This dissertation proposes two-level file cache management: the kernel allocates physical pages to individual applications, and each application is responsible for deciding how to use its physical pages. The dissertation addresses three issues: a global allocation policy that allows applications to control their own cache replacement while maintaining fair allocation of cache blocks among processes, integrated algorithms for caching and prefetching, and a low-overhead mechanism to implement the interactions between user processes and the kernel.

A prototype file system, ACFS, is implemented to experiment with application-controlled file caching and prefetching on a suite of I/O intensive applications. Experiments show that application-controlled file caching and prefetching combined with

disk scheduling significantly improves the file I/O performance for applications: individual applications' running times are reduced by 3% to 49% (average 26%), and multi-process workloads' running times are reduced by 5% to 76% (average 32%). Each technique provides substantial performance benefits: application-controlled file caching reduces the number of disk I/Os, carefully integrated caching and prefetching increase the overlap between CPU computation and disk accesses, and disk scheduling combined with prefetching reduces the average disk access latency. The combination of all three techniques provides the best performance.

Acknowledgements

I am very fortunate to have an advisor, Professor Kai Li, who taught me so much and who waited so patiently while I learned how to do research. He helped me discover the fun of kernel hacking. When I was very frustrated with finding a thesis topic, he suggested me to play with disks — indeed I had a lot of fun.

I am also very lucky to be able to work with Professor Edward Felten, who arrived in Princeton just before I started working on my thesis project. This dissertation would be a much poorer one without the countless discussions with him. I could talk to him about my ideas, and he would understand everything about them in fifteen minutes. He made the thesis work much more bearable and much more fun. It really was a privilege to work with him.

Professor Anna Karlin worked with me on some of the material in Chapter 3 of this thesis. My work benefited greatly from both her expertise and her friendship. I am grateful that because of this work, we became good friends and continued working with each other.

Professor Douglas Clark gave me many good advices, on things both related and unrelated to work. There was a time when I was seriously questioning whether I was PhD material. It was through his seminar on computer architecture, which was taught with an unusual approach, that I regained my confidence (though I do not remember what he taught). I wish there is someone like him wherever I work.

All the people in the CS building helped making my life in graduate school enjoyable. Students who came earlier than I did, in particular, Mark Greenstreet, Ayellet Tal, and Jenny Zhao, helped me go through the pre-general period. Students who came later than I did, in particular, Dan Boneh, Yuqun Chen, Drew Dean, Chris

Dunworth, Mariusz Jakubowski, Liviu Iftode, and everyone on the "dinner" mailing list, helped me go through the post-general period.

My deepest thanks to my American parents, Lee and Bob Gunther-Mohr. They gave me a home in Princeton, New Jersey. I cannot imagine what I would do if they were not here.

Finally, all these were made possible by my parents, who decided to send me away for a better education, and who helped me to stay on. I owe this thesis to them.

Contents

Abstract iii

Acknowledgements vi

1 Introduction 1

1.1 Motivation 1

1.2 Traditional File Cache Management 3

1.3 Incorporating Application Control 6

1.4 Thesis Contributions 8

1.5 Related Work 9

1.5.1 File Caching Studies 9

1.5.2 Prefetching Studies 10

1.5.3 Virtual Memory 12

1.5.4 Other Techniques 13

1.6 Outline of the Dissertation 14

2 Application-Controlled File Caching 16

2.1 Introduction: Why Application Control 16

2.1.1 An Example 16

2.1.2 Existing Approaches 18

2.2 Mechanisms and Policies 19

2.2.1 Design Challenge 19

2.2.2 Two-Level Cache Block Replacement 20

2.2.3 Criteria for the Kernel Allocation Policy 21
2.3 The Allocation Policy 22
2.3.1 Algorithm Design 22
2.3.2 Theoretical Analysis 27
2.4 Simulations 36
2.4.1 Simulation Environment 37
2.4.2 Results with Ultrix Traces 37
2.4.3 Results with Sprite Traces 43
2.5 Summary 44

3 Integrating caching and prefetching **46**
3.1 Introduction 47
3.1.1 An Example 47
3.2 Basics of Prefetching and Caching 49
3.2.1 Problem Definition 49
3.2.2 Rules for Optimal Prefetching and Caching 50
3.2.3 Theoretical Model Definition 52
3.2.4 An Optimal Dynamic Programming Algorithm 53
3.3 Aggressive and Conservative 54
3.3.1 Policy Alternatives 54
3.3.2 Analysis of Conservative 55
3.3.3 Analysis of Aggressive 56
3.4 Simulations 63
3.4.1 Existing Approaches 63
3.4.2 File Access Traces 65
3.4.3 Simulation Models 66
3.4.4 Simulation Results 69
3.5 Aggressive with Limited Lookahead 73
3.6 Summary 74

4 ACFS: A Prototype **78**
4.1 Design . . . 79
4.1.1 Two-Level File Cache Management . . . 79
4.1.2 Incorporating Disk Scheduling . . . 84
4.1.3 Application Programming Interface . . . 86
4.2 Implementation . . . 91
4.2.1 Application-Controlled Caching . . . 91
4.2.2 Integrated Prefetching . . . 94
4.3 Applications . . . 97
4.4 Experiments . . . 102
4.4.1 File System Configurations . . . 102
4.4.2 Single Process Performance . . . 103
4.4.3 Multiple Process Performance . . . 109
4.4.4 Analysis of LRU-SP . . . 112
4.5 Summary . . . 122

5 Conclusions **125**
5.1 Summary of Results . . . 125
5.2 Lessons . . . 126
5.3 Future Work . . . 127

List of Tables

1 Average and standard deviation of the reference times and disk fetch times in each Ultrix trace. Times are in milliseconds. 66

2 Comparison of simulated and measured elapsed time and number of disk fetches for the Ultrix traces with two cache sizes: 6.4MB and 12MB. 69

3 Elapsed times (in seconds) of multiple applications under LRU-SP, ALLOC-LRU and the original kernel. 113

4 Numbers of block I/Os of multiple applications under LRU-SP, ALLOC-LRU and the original kernel. 114

5 Results from experiments testing the effectiveness of placeholders. . 115

6 Effect of a foolish process on smart applications. 117

List of Figures

1 The file cache. 2
2 The global LRU replacement policy. 4
3 Two-level file cache management. 7
4 An example under the Least-Recently-Used (LRU) replacement policy. 17
5 Same example under the Most-Recently-Used (MRU) replacement policy. 17
6 Interaction between kernel and user processes in two-level replacement. 21
7 This example shows what's wrong with the first try and how to fix it with the swapping position mechanism. 23
8 This example shows why place-holders are necessary. 25
9 Simulated Miss Ratios for `postgres` 39
10 Simulated Miss Ratios for `cscope` 39
11 Simulated Miss Ratios for Linking Kernel 40
12 Simulated Miss Ratios for Linking with X11 41
13 Simulated Miss Ratios for a Multi-Process Workload 42
14 Benefit of Using Place-Holders 42
15 Averaged Results from Sprite Traces 43
16 An example under no-prefetch policy. 47
17 Same example under always-prefetch policy. 48
18 Performance of prefetching algorithms for the Sprite and Ultrix traces. 68
19 Individual Ultrix traces: elapsed times of the eight algorithms. 70
20 Individual Sprite traces: number of disk fetches performed by the eight algorithms. 71
21 How LRU-SP keeps the "LRU" list. 82

22 Structure of the implementation of ACFS. 92
23 Single application running time, and number of block I/Os, with a 6.4MB file cache. 104
24 Single application running time, and number of block I/Os, with a 12MB file cache. 105
25 Normalized elapsed times of applications as a function of batch size B (the elapsed time when $B = 1$ is 1.0). 106
26 Average disk access latency as a function of batch size B. 107
27 The portion of user and system CPU time for the single application experiments with a 6.4MB file cache. 108
28 Multiple application running time, and number of block I/Os, with a 6.4MB file cache. 110
29 Multiple application running time, and number of block I/Os, with a 12MB file cache. 111
30 Elapsed time and number of block I/Os of the Andrew benchmark, when running concurrently with the applications, with a 6.4MB file cache. 119
31 Elapsed time and number of block I/Os of the Andrew benchmark, when running concurrently with the application, with a 12MB file cache.120

Chapter 1

Introduction

1.1 Motivation

In the last decade, advances in hardware technology have created a wide performance gap between microprocessors and disks. On the one hand, rapid developments in semiconductor technology have improved microprocessor performance by about 50% each year [HP90, PH94]. As a result, microprocessors today are orders of magnitude faster than those in existence ten years ago. On the other hand, limitations from the mechanics of rotating magnetic media have constrained magnetic disks to improve only by a factor of 2 in speed during the last ten years. Thus, there has been a widening gap between the speed of microprocessors and that of disks, which is often called the performance gap.

Amdahl's law [Amd67] dictates that unless the performance of every part in a system is improved, the whole system's performance will ultimately be limited by the slowest part of the system. Therefore, as the performance gap increases, a computer system's performance is increasingly limited by the part that primarily involves disks, the file system. The challenge for today's file system is to bridge the gap and keep the performance of I/O dominant applications improving at almost the same rate as that of processor speeds.

To achieve the goal, today's file systems widely use a file cache, also called a

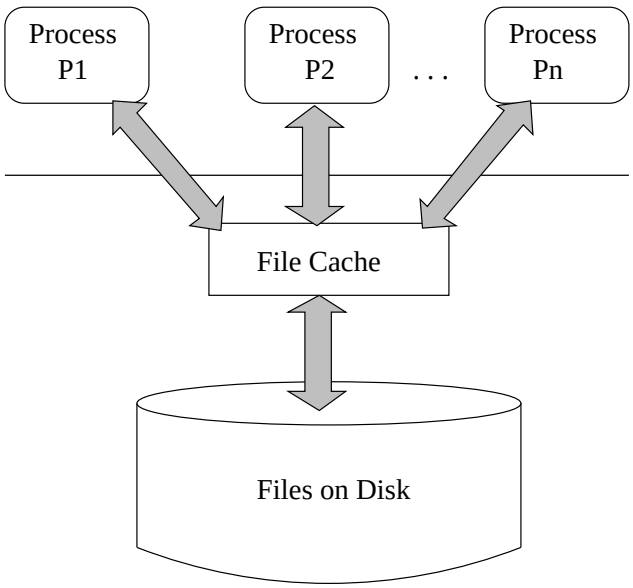

Figure 1: The file cache.

"buffer cache." A file cache is a portion of DRAM memory set aside by the operating system to store commonly accessed file data. If an application hits in the file cache, that is, if it reads or writes file data that are currently in the cache, the read or write proceeds at main memory speed, rather than disk speed. A file cache effectively "filters" out the access to cached data from the file access traffic, and only those that "miss" in the file cache need to access the disk (Figure 1). If misses are rare, file caching can decouple the performance of file systems from that of disks.

Clearly, the file cache is crucial to an application's file access performance. It reduces the traffic to disk. Studies [OCH*85, Bla93] have shown that file caches traditionally have a miss ratio between 10% to 40%, meaning that only 10% to 40% of file access traffic involves disk transactions. Ten years ago such a ratio was sufficient to keep many applications from being limited by disk speed.

Unfortunately, this is no longer the case, as the performance gap between microprocessor systems and disks increases. File system designers are now challenged to find new ways to improve the effectiveness of file caches.

1.2 Traditional File Cache Management

In a typical operating system, a portion of the main memory is allocated as file cache. Some operating systems allocate the portion at boot time, usually 10% to 20% of the total memory [Bac86]. Other operating systems dynamically adjust the file cache size [SBK*85, NWO88].

File data are cached by blocks, much like pages in virtual memory systems or lines in hardware processor caches. Blocks are chunks of continuous file data, usually 512-32K bytes. The size of blocks is configurable, and is usually chosen around 8K bytes to amortize the disk head seeking and positioning costs.

While the size of the file cache is based either on a number determined at configuration time or a number adjusted dynamically by the operating system, the *management* of the file cache involves two important decisions: the replacement policy and the prefetching algorithm.

Replacement policy Perhaps the most important decision in file cache management is the replacement policy. Since the file cache cannot possibly hold all file data on disk, some block must be displaced from the cache when an uncached block is brought in, e.g., when an application reads file data that are not in the cache. The replacement policy decides which block should be replaced. For a given cache size and access sequence, the replacement policy completely determines the hit ratio.

Traditional file systems generally employ global Least-Recently-Used (LRU) as the file cache replacement policy. The kernel usually keeps a list of all cached file blocks, ordering them by time of last access. For example, in Figure 2, the list is $A \rightarrow B \rightarrow C$, where A is the least recently used block and C is the most recently used block (the cache only holds three blocks in this example). And when a new block has to be brought in, the kernel looks at the LRU end of the list and replaces the block whose last access was earliest in the past. For example, in Figure 2, when block D, which is not cached, is accessed, block A is replaced, and the new list becomes $B \rightarrow C \rightarrow D$.

Some systems use approximations of global LRU for file cache replacement. This

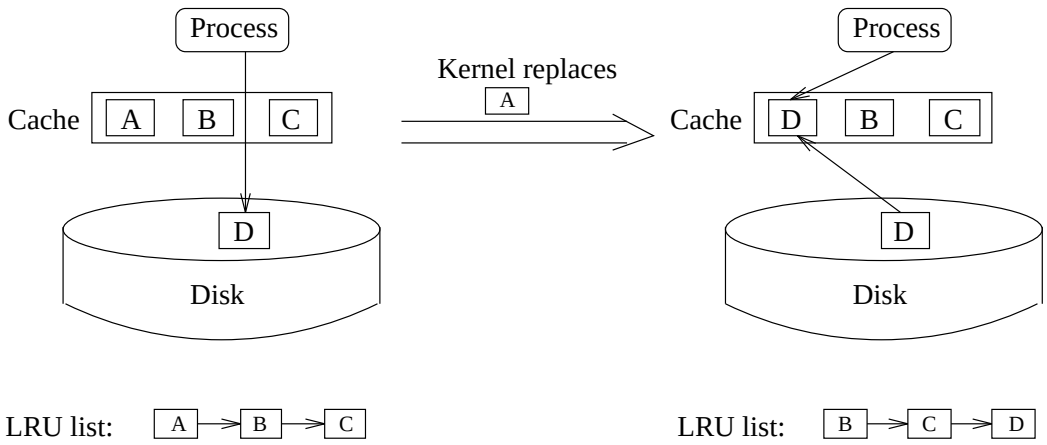

Figure 2: The global LRU replacement policy.

is because in these systems, the file cache and the virtual memory cache are under the same management module (called "unified cache") [GMS87]. And because of the difficulty in getting fine-grained access history in virtual memory, the management module often uses global LRU with a coarse access granularity. Typical algorithms are FIFO with second chance [GCD73] and two-hand clock [LL82].

In all these systems, the underlying design characteristic is that the kernel, not the applications, determines the file cache replacement policy.

Prefetching algorithm Another important aspect of file cache management is prefetching. Prefetching means reading file data from disks *before* they are used, rather than waiting until applications need them. The goal of prefetching is to overlap disk accesses with CPU computation, so that the latency of disk accesses can be hidden. Obviously, prefetching requires *a priori* knowledge or predictions of applications' file accesses.

Traditional file systems employ a simple heuristic, called sequential one-block lookahead (OBL) [Gin77, Smi78]. Whenever the last two accesses to a file were to consecutive blocks, e.g., blocks N and $N+1$, sequential OBL predicts that block $N+2$ will be accessed soon and prefetches it if necessary. Since most files are read sequentially most of the time [OCH*85, BHK*91], the heuristic has been quite successful at predicting the next read to a file and overlapping the disk access for it, if

necessary, with CPU computation.

Thus, the heuristic works well for large files that are accessed sequentially. In systems that are dominated by large sequential accesses, it is sometimes used more aggressively: it prefetches more than one block when sequential accesses are detected. This is especially useful when reading from a parallel disk array like RAID [PGG*95].

However, the heuristic does not work well for small files or for large files that are not accessed sequentially. The prediction of accesses within a file is not very useful when the file itself is only a few blocks long, and the prediction is often wrong for non-sequential access patterns.

Opportunities for Improvement

There are many opportunities to improve the traditional scheme.

For example, some applications can perform better with a tailored replacement policy, rather than the global LRU replacement policy. Researchers in the database community were among the first to notice this opportunity [Sto81]. The reason is that sometimes global LRU replaces hot blocks (blocks that are used very frequently, such as, index blocks) when there is a sequence of references to a large number of cold blocks (blocks that are not used frequently), or it replaces the wrong blocks and results in a hit ratio of zero, for example, when the file is accessed repeatedly in a sequential fashion and the cache is smaller than the file. In the former case, applications can perform much better if the frequently accessed blocks are retained in cache; in the latter case, a most-recently-used (MRU) replacement policy can have a much higher hit ratio.

Thus, applications often have special knowledge about their file access patterns and can, in principle, use their knowledge to obtain good replacement policies. If these application-specific policies are used instead of global LRU, applications can have a much higher hit ratio in file cache and consequently better I/O performance.

Another opportunity is that applications often know what file accesses they are going to make, though to a limited extent. For example, in the UNIX shell, the command arguments often include the list of files the program will access. Another

example is database applications: depending on the query, databases can often do some precomputation to find out what file blocks they will need for the execution of the query [Sto81].

The lookahead information can be used for a more aggressive method of prefetching than sequential one-block lookahead. Such a method of prefetching not only increases the overlap between disk accesses and CPU computations, but can also reduce the cost of average disk accesses by properly ordering the servicies of prefetch requests. The total effect would be a significant reduction in the amount of stall time due to file I/O.

The challenge is to design a scheme that enables applications to use their knowledge to improve their file I/O performance, and at the same time let the kernel allocate the resources and coordinate the use of file cache among competing applications, so that the performance of the whole system is improved.

1.3 Incorporating Application Control

In this dissertation, I propose a scheme to allow applications to exert control over file cache replacement and to prefetch file data. The scheme has the following characteristics. It integrates cache replacement and prefetching carefully so that neither harms the performance of the other. It applies disk scheduling to further improve the performance of prefetching. Finally, it operates in multi-process environments, so that applications can use these techniques without unduly harming each other or degrading the performance of the whole system. I call the scheme "application-controlled file caching and prefetching."

The underlying framework in application-controlled file caching and prefetching is two-level file cache management. Two-level file cache management splits the responsibility of managing the file cache between the kernel and user processes: the kernel allocates cache blocks to application processes, and each application process is responsible for the use of its cache blocks. This is shown in Figure 3. The kernel's allocation and the applications' management interact via a scheme called two-level

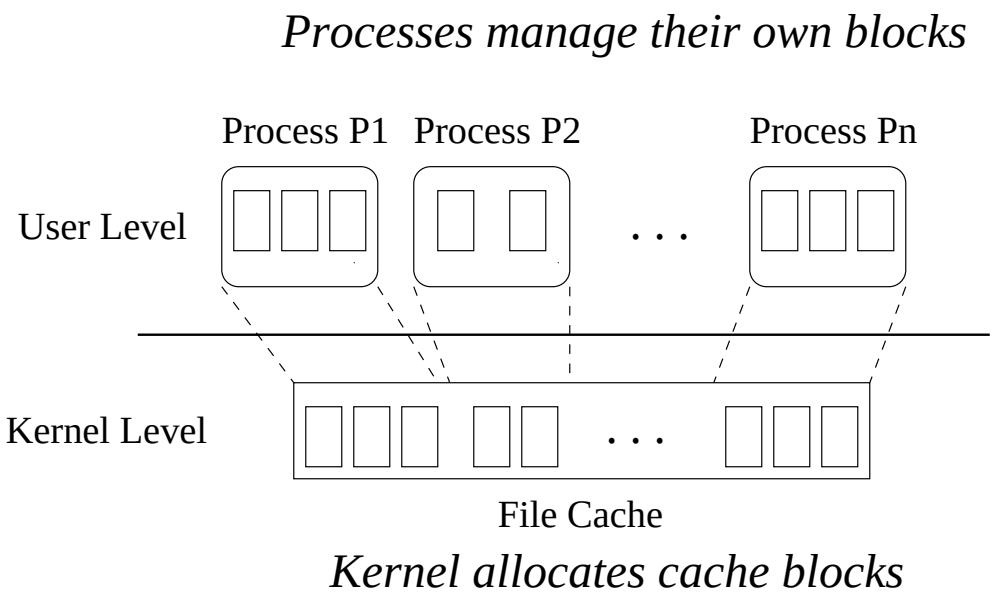

Figure 3: Two-level file cache management.

block replacement, which is addressed in detail in Chapter 2.

Two-level cache management requires that the kernel have a *global allocation policy* for deciding how to allocate cache blocks to processes, and that each user-level process have a *local management policy* for deciding how to use its blocks. Thus, two key problems must be solved:

- What should be the kernel's global allocation policy? The kernel's policy impacts the performance of the whole system and must be chosen carefully when applications are allowed to use *any* local policies they want.

- How should cache replacement and prefetching be integrated in a process's local management policy? If prefetching is not integrated with caching carefully, it may hurt performance.

This dissertation addresses both problems and gives solutions that are both theoretically sound and highly effective in practice.

1.4 Thesis Contributions

The main contributions of this thesis are:

- A solution of the global allocation problem in application-controlled file caching. The policy developed here, *Least-Recently-Used with Swapping and Placeholders (LRU-SP)*, guarantees that an application-tailored replacement policy may improve the overall file system performance and that a foolish replacement policy of one process will not degrade the file cache hit ratios of other processes.

- A solution of the integration problem in file caching and prefetching. The problem is analyzed from a theoretical perspective, policy alternatives are investigated, and one policy, *aggressive*, is found to be close to optimal. Simulation and implementation demonstrate that the policy works very well in practice.

- A simple and flexible application programming interface for application-controlled file caching and prefetching. The interface, *priorities and policies*, allows applications to specify priorities on files (file blocks) and specify replacement policies for each priority. It also allows applications to specify a list of file data for prefetching. It provides both flexibility of control for applications and fast implementation of common policies.

- Experiments with application-controlled file caching and prefetching by a suite of I/O intensive applications on the prototype file system, ACFS. The results show that application-controlled file caching and prefetching can significantly improve file access performance for the applications, and the above solutions guarantee that in most cases the performance of all applications is no worse than under traditional file systems. Careful integration of application-controlled file caching, prefetching and disk scheduling reduces individual applications' running times from 3% to 49% (average 26%), and multi-process workloads' running times from 5% to 76% (average 32%). The results also show that disk scheduling is particularly important for prefetching; when combined with disk scheduling, prefetching provides much greater performance benefits.

1.5 Related Work

1.5.1 File Caching Studies

Caches and memory hierarchies are old concepts. The idea of a cache was first introduced by Maurice Wilkes in [Wil65]. Today caches are used in every place where there is a performance gap: between the microprocessor and the main memory, between the main memory and the disk, between local and remote file systems, between the disks and the tertiary storage, and so on. This thesis focuses on the use of caches in file systems, i.e. between the main memory and the disk or the storage-server.

Research on distributed file systems has long investigated file caching strategies: client and server caching [SGN85, Che87, SBK*85, HKM*88b, NWO88], sharing memory among clients [DWAP94, ADN*95, FMP*95], as well as strategies for caching in disconnected environments [KS92]. However, these studies are not primarily concerned with the performance impact of different replacement policies. They still assume global LRU as the basic cache replacement policy. By contrast, application-controlled cache replacement can improve the cache hit ratio not only in local file systems, but also in distributed file systems.

The database research community has long studied access patterns and buffer replacement policies [Sto81, CD85, OOW93], though existing file systems do not support them. ACFS can support most of the policies a database system might want to use. Chapter 4 will use the relational database system `postgres` [The93] as an example to illustrate how database systems can control file caching in ACFS.

Database researchers have also studied global allocation issues, though in a different context. Their questions often are: how to allocate buffers among different types of queries, how to allocate buffers between index and data blocks, and so on. Their solutions often take advantage of the very rigid access patterns of query executions, and hence do not apply directly to file systems.

1.5.2 Prefetching Studies

Prefetching is an old concept as well. Gindele and Smith introduce the sequential one-block lookahead (also called next-block) strategy and discuss hardware implementations of the strategy in [Gin77] and [Smi78].

In recent years prefetching has been studied extensively in various areas, including, uni-processor and multi-processor architectures [RL92, CB92, MG91, CKP91, GGV90, Smi78, TE93], file systems [PGS93, GA94, TD90] and databases [CKV93, PZ91]. Few of these studies consider the interaction between prefetching and caching as explored in this dissertation. Below I discuss prefetching studies in file systems and databases, and then talk briefly about related work in processor architectures.

In file systems, perhaps the most primitive kind of prefetching is using large I/O units (i.e. blocks), as in extent-based or similar file systems [MK91]. Most file systems use sequential one-block lookahead as well. These approaches are often too limited and only benefit applications that make sequential references to large files [GA94].

Recently there have been a number of research projects on prefetching in file systems. Patterson's Transparent-Informed Prefetching (TIP) [PGS93, PG94, PGG*95] uses hints from applications to prefetch file data; their application programming interface is similar to ACFS's interface for prefetching. However, instead of supporting application-controlled cache replacement, TIP "infers" good replacement policies from the hints, and consequently does not benefit applications that know about good replacement policies but do not predict their file accesses well. TIP focuses on prefetching from parallel disks, and handles the interaction between cache replacement and prefetching by estimating the cost and benefit of prefetch decisions. ACFS focuses on prefetching from a single disk, and integrates cache replacement and prefetching with an algorithm that is proven to be close to optimal. The performance improvement in TIP on a single disk, reported in [PGG*95], is comparable to that of ACFS (Section 4.4).

Another project is Griffioen and Appleton's work [GA94] in using "probability graphs" to predict future file accesses from past ones and to prefetch accordingly. Their work demonstrates the benefits of prefetching, though it is mostly a simulation

study and does not address the interaction between caching and prefetching.

Other studies in parallel prefetching [EK89, WYT93] reveal the thrashing problem that occurs due to the lack of integration between caching and prefetching. Although the theoretical study in this thesis focuses on the case of a single disk or server, some of the policy principles in Chapter 3, namely, "do-no-harm" and "first-opportunity," apply to prefetching algorithms in the parallel context. In Chapter 3 I will show that these principles are important in avoiding the thrashing problem.

There have also been many studies in file systems focusing on the prediction of future accesses from past accesses. Tait and Duchamp's work [TD90] tries to detect users' file access patterns and exploit the patterns to prefetch files from servers. Palmer and Zdonik's work on Fido [PZ91] tries to train an associative memory to recognize access patterns in order to prefetch. Vitter and Krishnan's work [CKV93] tries to use compression techniques to predict future file accesses from past access history. All these studies generate promising results with respect to prediction. However, the models which they use assume that many block I/Os can be done in a single time step (i.e. at each reference), which unfortunately is not realistic for file systems. With such a model, cache management becomes substantially less important.

Some mobile file systems use the "stashing" approach [KS92]. Though similar to prefetching, stashing has a quite different purpose: stashing is concerned with the availability of files when the client is disconnected from the server, while prefetching is more concerned with latency hiding. Hence, policies and algorithms for these two techniques are quite different.

Finally, prefetching in uni-processor and multi-processor caches is similar to prefetching in file systems. However, in these systems there is little flexibility in cache management, as the cache is usually direct-mapped or has very limited associativity. In addition, it is not possible to spend more than a few machine cycles on each prefetch. In contrast, file systems can change their cache management algorithms freely and can spare more cycles for calculating a good replacement or prefetching decision, as the potential savings are substantial. On the other hand, Tullsen and Eggers's work [TE93] demonstrates the thrashing problem of prefetching in bus-based

multiprocessor caches, suggesting that the "do-no-harm" principle (Chapter 3) applies in those systems as well.

1.5.3 Virtual Memory

In virtual memory systems, application-controlled physical memory management can improve performance significantly. Unfortunately, traditional virtual memory systems only provide very limited facilities for applications to control memory management. For example, the Berkeley UNIX system calls for pinning pages, `mpin` or `mlock`, are only available to the superuser. Advisory calls like `vadvise` or `madvise` provide an interface for applications to advise the kernel about their access patterns over a memory object, including a mapped file. However, the interface is very limited, allowing specification of only a small number of basic patterns, and it does not support prefetching of pages specified by applications.

As a result, in the past few years there has been a stream of research papers on application control of caching in the virtual memory context [MA90, SP91, HC92, KLVA93]. My work differs from that described in these papers in three significant ways:

- None of these papers (except [HC92]) addresses the global allocation policy problem. The approach suggested in [HC92] requires an external (administrative) resource distribution policy, and it is not clear how the approach can be used in practice.

- None of these papers investigates prefetching or its interaction with cache replacement.

- Most of these papers rely on interprocess communications or upcalls for kernel-to-user communication, and consequently report overhead as high as 10% of the total execution time [MA90, SP91]. The implementation in this thesis provides a flexible interface for applications to issue primitives to exert control on cache replacement. Experiments show that this is adequate most of the time and requires low overhead.

On the other hand, my work shares some common purposes with the existing work on application-controlled virtual memory. With some minor modifications, the techniques used in application-controlled file caching and prefetching apply to virtual memory cache management as well.

First, the techniques used for my global allocation algorithm, swapping and placeholders, can be modified to work in virtual memory systems, as will be explained in Chapter 2. Second, the algorithm that integrates caching and prefetching applies to cache management in virtual memory as well. Finally, the application programming interface, as explained in Chapter 4, can be used for application-controlled physical memory management. Instead of files, the interface can specify a range of virtual addresses (or memory regions).

1.5.4 Other Techniques

Besides caching and prefetching, there are many other ways to improve file system performance. They fall roughly into the following categories:

- **Disk allocation**. Continuous allocation techniques [MJLF84, MK91, SS95] seek to lay a file continuously on the disk, so that logically consecutive blocks are physically adjacent. Clustering [KS86] tries to lay files that are accessed together close to each other on disk. Both are aimed at reducing the seek and rotational delays for average disk accesses.

- **Bandwidth and capacity**. Compression can reduce a file's disk transfer time and capacity requirement [BJLM92, WM91]. Mirrored disks [BG88, PBD93] or parallel disk arrays like RAIDs [SG86, PGK88, CLG*94] can improve bandwidth and reduce disk contention.

- **Handling writes**. Delayed writeback techniques [MJLF84, Mog94] cache file writes and reflect the modifications back to the disk at a later time. Log-structured file systems (LFS) [RO91, SBMS93] coalesce writes to a sequential log and write the log to the disk. All these techniques aim at reducing the cost and frequency of disk writes while maintaining file system integrity.

- **Crash recovery**. Metadata logging [GMB81, RO91, SBMS93, Hag87, dJKH93] and asynchronous ordered writes for metadata updates [GP94] try to reduce the cost of metadata operation without sacrificing the integrity of file system on-disk data structures. Non-volatile RAM [BAD*92, WZ94] is sometimes used to improve response time and reliability.

Most of these techniques do not affect file cache management because they target different areas in attempts to improve file system performance. The approach described in this thesis is complementary to these approaches, such that, when used in conjunction with application-controlled file caching and prefetching, these approaches will improve file system performance even more.

Most of these techniques require little modification when integrated with application-controlled file caching and prefetching. The only exception is delayed writeback. Delayed writeback approaches can be integrated with application-controlled file caching and prefetching in a number of ways. One approach is to treat modified (dirty) blocks and unmodified (clean) blocks the same when choosing which block to replace. Another approach is to keep dirty blocks in cache and write them back to disks when the data are old enough (e.g. exceed 30 seconds), as done in the original Sprite LFS implementation [Ros92].

Yet another approach is to keep dirty blocks in cache but write them back to disk in the background more often, for example, when the number of dirty blocks exceeds a certain threshold (as done in a BSD LFS [SBMS93]), or when there is a cluster of dirty blocks to write to disk (as done in the Ultrix 4.3 file system [MK91]). The implementation of my prototype file system took this approach.

1.6 Outline of the Dissertation

The remainder of the thesis is organized in four chapters. I present the theoretical studies and simulation results first: Chapter 2 focuses on the global allocation policy, and Chapter 3 focuses on the integration algorithms for caching and prefetching. I turn to practical considerations in Chapter 4, presenting the complete design

of application-controlled caching and prefetching, the implementation of ACFS (my prototype file system), and experimental results. The results are summarized and the implications for future research are discussed in Chapter 5.

The materials in some chapters have been published as conference papers. Some materials in Chapter 2 have been presented in [CFL94a]. The materials in Chapter 3 have been reported in [CFKL95], which is also joint work with Anna Karlin. Part of Chapter 4 has been presented in [CFL94b].

Chapter 2

Application-Controlled File Caching

2.1 Introduction: Why Application Control

File caching is a widely used technique in today's file systems. One major challenge in file caching is to provide a high cache hit ratio, in other words, to make the majority of file accesses proceed at main memory speed.

The key issue determining cache hit ratio is the block replacement policy. Traditional file systems use Least-Recently-Used (LRU) as the criterion to choose a replacement among all cached blocks. Unfortunately, this global LRU policy does not always perform well.

2.1.1 An Example

Consider a program that writes a file, and then reads it, for example, a compiler that writes the intermediate codes to a file and then reads them back for further processing. Suppose the file is bigger than the file cache: in the example in Figure 4, the file has three blocks A, B and C, while the cache can only hold two blocks. Then the reference sequence (write A, write B, write C, read A, read B, read C) results in six disk transfers under the global LRU policy: three disk writes when the blocks are

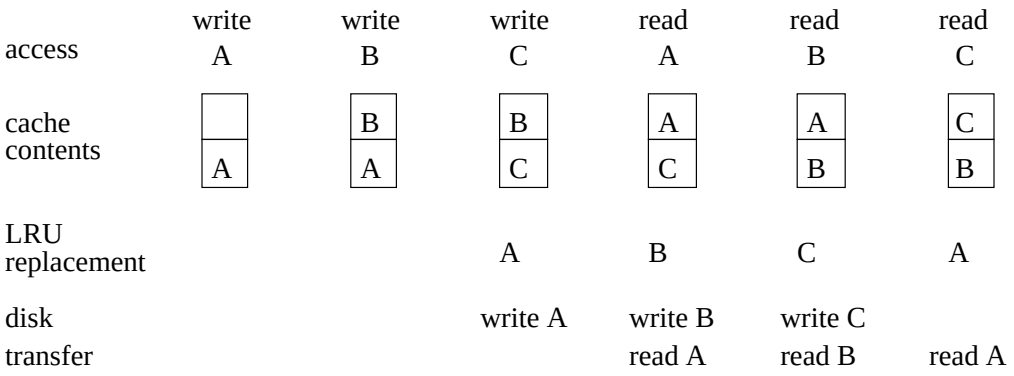

Figure 4: An example under the Least-Recently-Used (LRU) replacement policy.

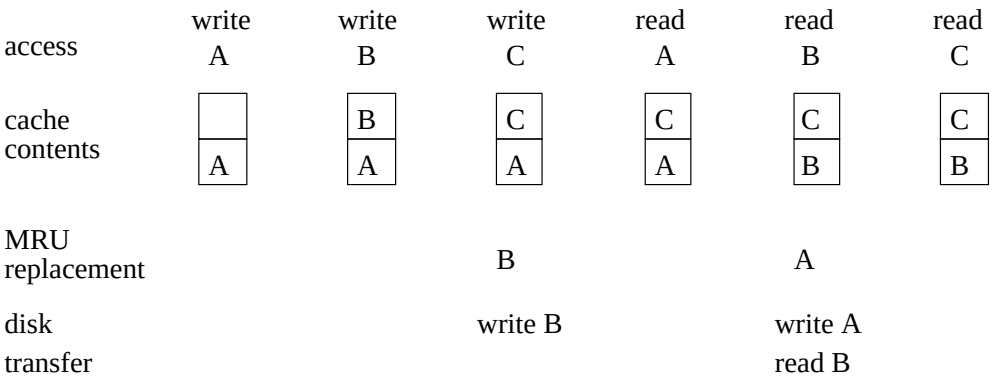

Figure 5: Same example under the Most-Recently-Used (MRU) replacement policy.

replaced and three disk reads, as shown in Figure 5.

On the other hand, the program may know that it writes and reads the file sequentially and it should use most-recently-used as the replacement policy. If the program can control the replacement of its files, it can have only three disk transfers (write B, write A and read B), instead of six (Figure 5). The writing of block C to the disk is deferred in this example and can be done in the background.

There are many situations in real systems which are similar to the above example. While the kernel may use a more sophisticated and complicated replacement policy, it is hard for one policy to satisfy the special needs of all applications. Essentially, "one size cannot fit all."

On the other hand, it is not hard for ordinary applications to make intelligent

replacement decisions. Following are some examples of common access patterns, and how applications can make good replacement choices based on these patterns.

- *sequential*: most files are accessed sequentially most of the time. Thus, for every file, the block just before the current file pointer should be replaced first, as it will be referenced latest among the blocks of the file;

- *filter*: if an application reads files one by one in the order of their names in the command line (like many UNIX programs), then a compiler or preprocessor can easily determine which file blocks should be kept in cache and when a file block is no longer needed;

- *same-order*: if a file or a group of files are repeatedly accessed in the same order (for example, when a "$*$" for file name expansion appears more than once in a shell script), the most-recently-used policy should be used for replacement among the blocks.

- *done-with*: sometimes a program (or its compiler) knows when it no longer needs to access certain file blocks and can mark those blocks to be replaced first.

Therefore, we need a scheme to let applications control the replacement of the file blocks that they use.

2.1.2 Existing Approaches

In traditional file systems, since applications cannot control replacement decisions of the file cache, many of them resort to buffering file data in user address space as a way of keeping data in main memory. Unfortunately, since the virtual memory system can still page out part of user address space, and applications have no control over the paging decisions, this approach does not give applications real control. Furthermore, since the file system caches file data as well, this approach leads to double buffering, which wastes space.

Applications can also map files into user address space and use virtual memory system calls like `madvise` [Mic90] to advise the kernel of the files' caching. Unfortunately these system calls cannot express which block or which file is a better candidate for replacement.

An alternative approach to application-controlled file caching would be to let applications map portions of file cache into their user address space and access the file cache directly. However, applications still have no control over the *replacement* of file blocks. Basically, such approach merely changes the problem from how to let applications control file cache replacement to how to let them control page replacement in the virtual memory system.

None of these approaches gives applications direct control over the replacement decisions of the file cache, and consequently they can perform poorly for some applications.

2.2 Mechanisms and Policies

The chief mechanism to support application control of file caching is two-level cache block replacement. Below I describe why we need the mechanism, how it works, and the criteria for its allocation policies.

2.2.1 Design Challenge

The goal is to allow user-level control over cache replacement decisions. However, we cannot simply move all responsibility for cache management to the user level. In a multiprogrammed system, the kernel must control the allocation of resources among competing user processes.

To reduce the miss ratio, a user-level file cache needs not only an application-tailored replacement policy but also a sufficient number of available cache blocks. In a multiprocess environment, the allocation of cache blocks to processes will affect the file cache miss ratio of the entire system. It is the kernel's job to ensure that the miss ratio of the whole system does not increase because of the user-level management of

cache replacement policies.

The challenge is to allow each user process to control its own caching, while simultaneously maintaining the dynamic allocation of cache blocks among processes in a fair way such that overall system performance improves. Below I describe a scheme that achieves the goal.

2.2.2 Two-Level Cache Block Replacement

The basic idea behind my scheme is to split responsibility between the kernel and the user level. The kernel is responsible for allocating cache blocks to processes. Each user process is free to control the replacement strategy on its share of cache blocks; if it chooses not to exercise this choice, the kernel applies a default policy (which is usually LRU). I call the approach *two-level cache block replacement.*

To be more precise, each file is assigned to a "manager" process, which is responsible for making replacement decisions concerning the file. Usually the process that opens the file acts as its manager; however, the process may designate another process to be the manager of a particular file. If several processes have the same file open simultaneously, it is up to the processes to agree on a manager; if they cannot agree, the kernel imposes the default LRU policy for that file. Processes that do not want to control their own replacement policy can abdicate their management responsibility, in which case the kernel applies the default LRU policy for the affected files.

The interactions between kernel and manager processes are as follows. On a cache miss, the kernel finds a candidate block to replace, based on its global replacement policy (step 1 in Figure 6). The kernel then identifies the manager process of the candidate. This manager process is given the opportunity to decide on the replacement (step 2). The kernel suggests the candidate block, but the manager process may overrule the kernel's choice by suggesting an alternative block under that manager's control (step 3). Finally, the block suggested by the manager process is replaced by the kernel (step 4). If the manager process is not cooperative (for example, it suggests an alternative block that is under some other manager's control), then the kernel simply replaces the candidate block.

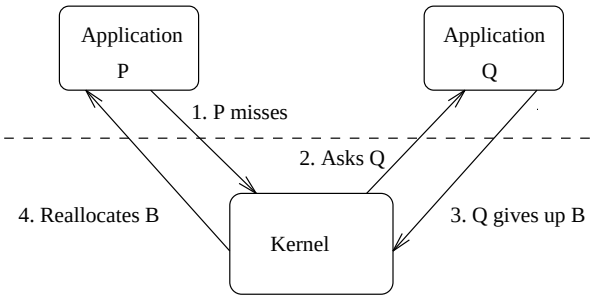

Figure 6: Interaction between kernel and user processes in two-level replacement.

The kernel's replacement policy is in fact an allocation policy. Suppose that process P's reference misses in the cache and the kernel finds a replacement candidate owned by process Q. Although process Q's user-level replacement policy decides *which* of its blocks will be replaced, the replacement will cause a deallocation of a block from process Q and an allocation of a block to process P.

2.2.3 Criteria for the Kernel Allocation Policy

The kernel allocation policy is the most critical part of two-level replacement. While it is the application's job to find a good replacement policy, it is the kernel's job to allocate cache blocks well among the processes. Giving too few blocks to processes that really need them, while at the same time giving too many blocks to processes that do not use them, will result in poor performance of the entire system.

To obtain best performance, it is known that allocation should follow the *dynamic partition* principle [GCD73]: when multiple processes compete for cache space, the number of cache blocks allocated to each process should vary dynamically in accordance with the process's working set size. Experience has shown that the allocation performed by global LRU or its approximations works relatively well; it approximates the dynamic partition principle or tend to follow processes' working set sizes.

Thus, the goal is to design an allocation policy for the kernel to guarantee that the two-level replacement method indeed improves system performance over the traditional global replacement method (when user processes are not making bad replacement decisions). To be more precise, let us call a decision to overrule the kernel *wise*

if the candidate block (the replacement suggested by the kernel) is referenced before the alternative (the replacement suggested by the user process), and *foolish* if the alternative will be referenced first. A decision not to overrule the kernel can be viewed as neutral. The kernel's allocation policy should satisfy three principles:

1. *A process that never overrules the kernel does not suffer more misses than it would under global LRU.* This ensures that ordinary processes, which are unwilling or unable to control their replacement policies, will perform at least as well as they did under the traditional file system.

2. *A foolish decision by one process never causes another process to suffer more misses than it would suffer under global LRU.* Of course, we cannot prevent a process from discarding its own valuable data. However, we must ensure that an errant or malicious process cannot harm the performance of others.

3. *A wise decision by one process never causes any process, including itself, to suffer more misses than it would suffer under global LRU.* This ensures that processes have an incentive to choose wisely. (It goes without saying that wise decisions should actually improve performance whenever possible.)

Essentially, the three principles require that application-controlled file caching not worsen the performance when compared with traditional file systems, unless applications make mistakes themselves.

2.3 The Allocation Policy

This section presents a global allocation policy that satisfies the three principles in Section 2.2.3. I first describe the algorithm, and then prove that it satisfies the principles.

2.3.1 Algorithm Design

I will describe the allocation policy in an evolutionary fashion. I will start with a simple but flawed policy, and then diagnose and fix two problems with it. The result

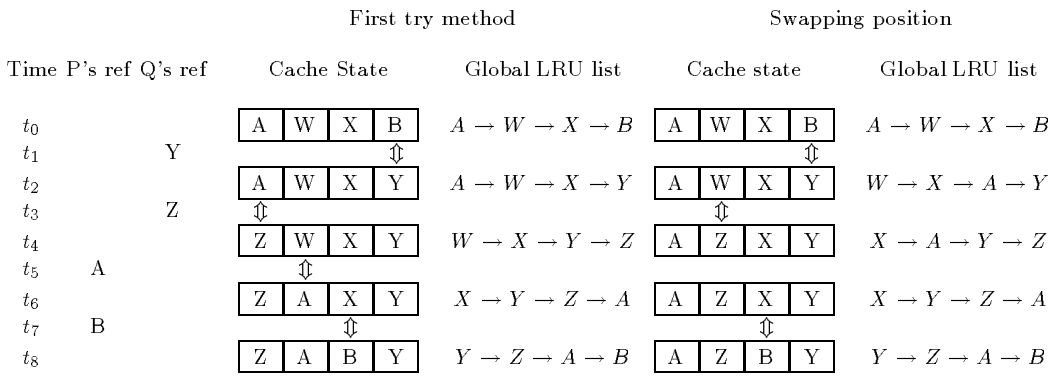

Figure 7: This example shows what's wrong with the first try and how to fix it with the swapping position mechanism.

will be a fully satisfactory allocation policy.

First Try

To start, we can have an allocation policy that simply uses the globally least-recently-used block to choose a process. The kernel maintains an LRU list of all blocks currently in the file cache. When a replacement is necessary, the block at the least-recently-used end of the list is suggested as a candidate, and its owner process is asked to give up a block.

The problem is that if the owner process overrules the kernel, the candidate block still stays at the least-recently-used end of the list. On the next miss, the same process will again be asked to give up a block.

The left side of Figure 7 shows an example. Two processes, P and Q, share a file cache with four blocks. Process P uses blocks A and B; process Q uses blocks W, X, Y, and Z. The reference stream is Y, Z, A, B.

The top line shows the initial LRU list, at time t_0. The first reference, to Y at time t_1, causes a replacement. The kernel consults its LRU list and suggests A for replacement. A's owner, process P, overrules the kernel. It decides to replace B, hoping to save the next miss to A. The LRU list is now as shown at time t_2. The next reference, to Z at time t_3, causes another replacement. Again, A is chosen as a

candidate. This time P has no other blocks in the cache and hence must give up A. The LRU list is now as shown at time t_4. At this point, the next two references, to A and B, both miss.

There are four misses in this example, two misses by P and two by Q. But note that under global LRU, there would be only three misses, one by P and two by Q. This violates Principle 3: a wise decision by process P causes P to suffer one extra miss.

Swapping Position

The problem in the above policy arises because the LRU list is maintained in strict reference order. Intuitively, the only use of the LRU list is to decide which process will give up a block upon a miss. To get the same allocation policy as the existing global LRU policy, our policy's LRU list should be in correspondence to the LRU list in the original algorithm. This can be achieved by swapping the blocks' positions in the LRU list.

Suppose the kernel suggests A for replacement, but the user-level manager overrules it with B. At this point, the previous policy would simply replace B. The new policy first swaps the positions of A and B in the LRU list, then proceeds to replace B. As a result of this swap, A is no longer at the LRU end of the list. Compared with the LRU list under global LRU (that is, if A is replaced), the only difference is that A is in B's position.

This fixes the problem with above example as in Figure 7. The right side of the figure shows what happens under the new policy. On the first replacement, A moves to the head of the LRU list before B is replaced. The result is that A is still in the cache when it is referenced. Process P is no longer hurt by its wise choice.

In general, swapping positions guarantees that if no process makes foolish choices, the global hit ratio is the same as or better than it would be under global LRU.

Unfortunately, this policy does not guard against foolish choices made by user processes. This is illustrated by the left side of Figure 8. Processes P and Q share a three-block cache. The top line shows the initial LRU list at time t_0; the reference

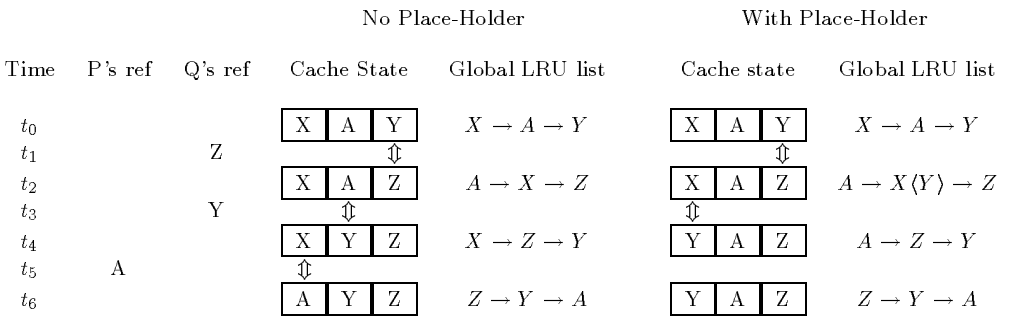

Figure 8: This example shows why place-holders are necessary.

stream is Z, Y, A. The first reference, to Z at time t_1, causes a replacement. The kernel suggests X for replacement. Now suppose process Q makes exactly the wrong choice: it decides to replace Y. After swapping X and Y in the LRU list, the kernel replaces Y, leading to the LRU list as shown at time t_2. The second reference, to Y at time t_3, misses. The kernel suggests A for replacement, and process P cannot overrule because it has no alternative to suggest. Thus A is replaced, leading to the LRU list as shown at time t_4. The third reference, to A at time t_5, misses.

There are three misses in this example, one miss by P and two by Q. Under global LRU, there would be only one miss, by Q. Had Q not foolishly overruled the kernel, the last two references would both have hit in the cache. Principle 2 is violated — process Q's foolish decision causes process P to suffer an extra miss.

Place-Holders

The problem in the previous example arises because Q's choice enables it to acquire more cache blocks than it would have had under global LRU. As a result, some of P's blocks are pushed out of the cache, which increases P's miss ratio. To satisfy Principle 2, we must prevent foolish processes from acquiring extra cache blocks. We achieve this by using *place-holders*.

A place-holder is a record that refers to a page. It records which block would have occupied that page under the global LRU policy. Suppose the kernel suggests A for

replacement, and the user process overrules it and decides to replace B instead. In addition to swapping the positions of A and B in the LRU list, the kernel also builds a place-holder for B to point to A's page. If B is later referenced before A, A's page can be confiscated immediately. This allows the cache state to recover from the user process's mistake.

The right side of Figure 8 illustrates how place-holders work. The top line shows the initial LRU list at time t_0. The first reference, to Z at time t_1, misses. Block X is chosen as a candidate for replacement, but process Q (foolishly) overrules the kernel and chooses Y for replacement. X and Y are swapped in the LRU list, and Y is replaced. At this point, a place-holder is created, denoting the fact that the block occupied by X would have contained Y under global LRU.

The second reference, to Y at time t_3, misses. The kernel notices that there is a place-holder for Y, which means that Y "should" have been in the cache, but was not, due to a foolish replacement decision by Y's owner. The kernel responds to this situation by correcting the foolish decision: it loads Y into the page that Y's place-holder pointed to, replacing X. Note that in this case the normal replacement mechanism is bypassed.

The LRU list is now as shown at time t_4. The third reference, to A, hits.

This example results in two misses, both by process Q. Under global LRU, there would have been only one miss, by Q. Process Q hurts itself by its foolish decision, but it does not hurt anyone else. Principle 2 is satisfied.

The Allocation Policy LRU-SP

Combining the above two fixes, here is the full allocation policy, LRU with Swapping and Placeholders (LRU-SP). If a reference to cache block b hits, then b is moved to the mostly-recently-used end of the global LRU list, and if there is a placeholder pointing to b, the placeholder is deleted. If the reference misses, then there are two cases. In the first case, there is a place-holder for b, pointing to t; in this case t is replaced and its page is given to b. If t is dirty, it is written to disk.

In the second case, there is no place-holder for b. In this case, the kernel finds

the block at the least-recently-used end of the LRU list. Say that block c, belonging to process P, is at the least-recently-used end of the LRU list. The kernel consults P to choose a block to replace, and suggests replacing c. Say that P's choice is to replace block x. If c and x are different, then the kernel swaps x and c in the LRU list. If there is place-holder pointing to x, it is changed to point to c; otherwise a place-holder is built for x, pointing to c. Finally, x's page is given to b, and x is written to disk if it is dirty.

In both cases, b moves to the most recently used end of the LRU list after the reference.

This algorithm satisfies all three design principles. The following is an intuitive argument, while a formal proof appears in the next section.

The algorithm has the property that it never asks a process to replace a block for another process more often than global LRU does. In other words, whenever a process is asked to give up a block for another process, it would also have already given up some block under global LRU.

To see why, first notice that the place-holder mechanism ensures that every process *appears*, in the view of other processes, never to unwisely overrule the kernel's suggestions. This is because whenever a process makes an unwise decision, only the errant process is punished and the state is restored as if the mistake were never made.

Hence, from the allocator's point of view, every process is either doing LRU or is doing something better than LRU. Therefore we need only guarantee that those that are doing better than LRU are not discriminated against. Swapping position serves this purpose. That is, the allocator does not care which of a process's cache blocks is holding which data, as long as its cache blocks occupy the same positions on the LRU list that they would have occupied under global LRU.

2.3.2 Theoretical Analysis

In this section I present a formal proof showing that LRU-SP satisfies the principles stated in Section 2.2.3. I also present an analysis showing that LRU-SP can be $2(p+1)$ competitive [ST85] (i.e. the miss ratio is no more than $2(p+1)$ times that of optimal),

where p is the number of concurrent processes.

Proof of LRU-SP satisfying the principles

Model definition Consider a known sequence $\omega = (r_1, r_2, \ldots, r_n)$ of accesses to file blocks. The sequence is an interleaving of accesses from p manager processes $P_1, P_2, \ldots, P_p$, with each block appearing in exactly one process's accesses. We assume that the sequence is given *a priori* and is not affected by the replacement decisions made.

The file cache can hold k blocks ($k < n$). Let $F_{global-LRU}(P_i, t)$ be the number of process P_i's accesses in $(r_1, r_2, \ldots, r_t)$ that miss in the file cache under global LRU replacement policy, and let $F_{LRU-SP}(P_i, Alg_i, t)$ be P_i's misses in $(r_1, r_2, \ldots, r_t)$ under two-level replacement with LRU-SP as allocation policy and Alg_i as P_i's replacement policy. In addition, let $M(P_i, Alg_i, t)$ be the number of mistakes Alg_i made that are caught by placeholders during the execution of $(r_1, r_2, \ldots, r_t)$, in other words, the number of times when it was discovered that Alg_i's replacement choice was worse than the kernel's candidate block.

My main theorem is the following:

Theorem 1 *Given any reference sequence $\omega = (r_1, r_2, \ldots, r_n)$, for every process P_i and any replacement algorithm Alg_i it uses, its number of misses under two-level cache block replacement with LRU-SP allocation is no more than that under the global LRU replacement policy plus Alg_i's mistakes. In other words, for any $\omega = (r_1, r_2, \ldots, r_n)$,*

$$F_{LRU-SP}(P_i, Alg_i, n) \leq F_{global-LRU}(P_i, n) + M(P_i, Alg_i, n).$$

Note that an important assumption in the theorem is that the reference sequence is not affected by replacement decisions.

It is trivial to see that it is a corollary of Theorem 1 that LRU-SP satisfies the three principles in Section 2.2.3.

To prove the theorem we need the following notations. Let $[g_1(t), g_2(t), \ldots, g_k(t)]$ be the global LRU policy's LRU list of blocks, at r_t of ω. In other words, $g_i(t)$ is

the block in the ith position in the LRU list, with $g_1(t)$ being the most recently used block, and $g_k(t)$ being the least recently used.

Similarly, let $[a_1(t), a_2(t), \ldots, a_k(t)]$ be the "LRU" list under two-level replacement with LRU-SP, as described in the previous section. If $a_i(t)$ has a placeholder pointing at it for another block $h_i(t)$, we mark $a_i(t)$ as $a_i(t)\langle h_i(t)\rangle$. Thus, the list may look like $[a_1(t), \ldots, a_{i-1}(t), a_i(t)\langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)]$. Let's call it the LRU-SP list.

For the proof we need to augment the LRU-SP list. Instead of deleting the placeholder $\langle h_i(t)\rangle$ when $a_i(t)$ is accessed, we leave $\langle h_i(t)\rangle$ in the same position in the list, and label it as a virtual block. In other words, if the list is

$$[a_1(t), \ldots, a_{i-1}(t), a_i(t)\langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)],$$

and $r_t = a_i(t)$, then the new list becomes

$$[a_i(t), a_1(t), \ldots, a_{i-1}(t), \langle h_i(t)\rangle, a_{i+1}(t), \ldots, a_k(t)].$$

Thus, virtual blocks are old placeholders that no longer point at any real block but stay in the list. The list can now grow beyond k entries.

As the list changes according to the LRU-SP rule, a virtual block stays in the same position as if it were a real block. It is deleted from the list either when it is hit by a reference (i.e., $r_t = h_i(t)$ and $\langle h_i(t)\rangle$ is a virtual block), or when it reaches beyond the first k positions of the list (i.e., $\langle h_i(t)\rangle$ with $i > k$). Obviously, if at reference r_t there are $N_v(t)$ virtual blocks, then the augmented list has $k + N_v(t)$ elements.

Note that the block at the LRU (right) end of the augmented LRU-SP list and that at the LRU end of the original LRU-SP list are always the same, because a virtual block can never reach the LRU end without being deleted first. Therefore, the LRU end of the augmented list is always the candidate replacement block in LRU-SP.

The purpose of augmenting the LRU-SP list is to establish the correspondence between the LRU-SP list and the global LRU policy's list. The following lemma shows that at any reference r_t of ω, if we take the first k elements from the augmented list, and substitute every $a_i(t)\langle h_i(t)\rangle$ by $h_i(t)$, and every $\langle h_j(t)\rangle$ by $h_j(t)$, then we obtain exactly the global LRU policy's list $[g_1(t), g_2(t), \ldots, g_k(t)]$.

Lemma 1 (Correspondence) *Given* $\omega = (r_1, r_2, \ldots, r_n)$, *at reference* r_t, *let* $G(t) = [g_1(t), g_2(t), \ldots, g_k(t)]$ *be the list of the global LRU replacement policy, and let*

$$A(t) = [a_1(t), \ldots, a_i(t)\langle h_i(t)\rangle, \ldots, h_j(t), \ldots, a_m(t)], m \geq k$$

be the augmented LRU-SP list. Let $A_{1-k}(t)$ *be the first* k *elements of* $A(t)$, *and let* H *be the operation that substitutes every* $a_i(t)\langle h_i(t)\rangle$ *by* $h_i(t)$, *and every* $\langle h_j(t)\rangle$ *by* $h_j(t)$. *Then at every* r_t, $1 \leq t \leq n$,

$$H(A_{1-k}(t)) = G(t). \tag{1}$$

Proof. The proof is carried out by induction. Trivially, Equation 1 is true at $t = 1$.

To facilitate the argument, notice that in two-level replacement, if an alternative block other than the candidate block is replaced, the resulting $H(A_{1-k}(t))$ is the same as if the candidate block had been replaced. The reason is the following. Suppose the candidate block is b, and an alternative block c is replaced. Since a placeholder for c is built to point to b, the element in the LRU-SP list is $b\langle c\rangle$. The H operation will substitute c for b in $H(A_{1-k}(t))$, resulting in the same list as if b, not c, were replaced.

Suppose Equation 1 is true at $t = q$, let's consider the cases at $t = q + 1$. There are four possibilities:

1. If r_t is in both $G(q)$ and $A(q)$, then r_t is in $A_{1-k}(q)$, and the changes from $G(q)$ to $G(q+1)$ and from $A_{1-k}(q)$ to $A_{1-k}(q+1)$ are the same. So Equation 1 is true at $t = q + 1$.

2. If r_t is in $G(q)$ but not in $A(q)$'s set of real blocks (that is, it hits in global LRU's cache but misses in LRU-SP's cache), then because $G(q) = H(A_{1-k}(q))$, r_t must be either a placeholder or a virtual block in $A(q)$:

   (a) If r_t is a placeholder $a_j(t)\langle r_t\rangle$, then a mistake by some user process is caught, and $a_j(t)$ is replaced;

(b) If r_t is a virtual block $\langle r_t \rangle$, then $A(q)$'s size $m > k$, and $a_m(t)$ is replaced.

In either case the replacement block ($a_j(t)$ or $a_m(t)$) is not in $H(A_{1-k}(q))$, and $H(A_{1-k}(q+1))$ is the same as $H(A_{1-k}(q))$ except that r_t is moved to the MRU end. Since $G(q+1)$ is also $G(q)$ with r_t moved to the MRU end, Equation 1 is still true.

3. If r_t is not in $G(q)$ but is in the set of real blocks of $A(q)$ (that is, it misses in global LRU's cache but hits in LRU-SP's cache), then because $G(q) = H(A_{1-k}(q))$, either r_t is $a_j(q), j > k$, or it has a placeholder, i.e., $a_j(q)\langle h_j(q) \rangle, j \leq k$:

   (a) If r_t is $a_j(q), j > k$, then it would move to the MRU end of the list. Meanwhile, $g_k(q)$ is replaced by global LRU, and its corresponding element in $A_{1-k}(q)$, $a_k(q)$, moves to the $k+1$ position of $A(q+1)$ (and therefore out of $A_{1-k}(q+1)$). Thus Equation 1 holds after r_t.

   (b) If r_t is $a_j(q)\langle h_j(q) \rangle, j \leq k$, then $a_j(q)$ is moved to the MRU end of the list, and $h_j(q)$, which is in $G(q)$, stays and becomes a virtual block $\langle h_j(q) \rangle$. Meanwhile, $g_k(q)$'s corresponding element in $A_{1-k}(q)$ moves to the $k+1$st position and is out of $A_{1-k}(q+1)$. Thus Equation 1 holds in this case.

4. If r_t misses in both $G(q)$ and $A(q)$, $g_k(q)$ is replaced from $G(q)$, and $a_m(q)$ is replaced from $A(q)$ with $a_k(q)$ moving out of $A_{1-k}(q+1)$. Hence Equation 1 is true at $t = q+1$.

■

The above case analysis also leads to the following lemma.

Lemma 2 *At every reference r_t of ω, let $N_v(P_i, Alg_i, t)$ be the number of virtual blocks in $A(t)$ that are in P_i's access sequence. Then for all i,*

$$F_{LRU-SP}(P_i, Alg_i, t) + N_v(P_i, Alg_i, t) \leq F_{global-LRU}(P_i, n) + M(P_i, Alg_i, n).$$

Proof. Similar to the above one by induction. Notice that any time a reference r_{q+1} hits in global LRU's cache but not in LRU-SP's, it either has a placeholder or is a virtual block. If it has a placeholder, then $M(P_i, Alg_i, q+1) = M(P_i, Alg_i, q) + 1$. If it has a virtual block, then $N_v(P_i, Alg_i, q+1) = N_v(P_i, Alg_i, q) - 1$.

The only case when $N_v(P_i, Alg_i, q+1) = N_v(P_i, Alg_i, q) + 1$ is when a placeholder becomes a virtual block, that is, r_{q+1} is in $A(q)$ and has a placeholder. Because of Lemma 1, r_{q+1} must miss in $G(q)$ in this case. Thus, $F_{global-LRU}(P_i, q+1) = F_{global-LRU}(P_i, q) + 1$.

■

Since $N_v(P_i, Alg_i, t) \geq 0$ for every t, this completes the proof of Theorem 1. ■

Extension of LRU-SP The above proof does not require that the traditional algorithm be global LRU, nor that the new algorithm be LRU-SP. Actually, the swapping and placeholder techniques can apply to any default global replacement algorithm and turn it into an allocation algorithm meeting the principles in Section 2.2.3. The only requirements are:

- maintain the set of virtual blocks as described above, that is, when there is a placeholder pointing at a block and the block is referenced, the placeholder becomes a virtual block and stays in the same position in the list;
- maintain the set of "saved" blocks, S. S is initially empty. Whenever a miss happens in the default algorithm but not in the application-controlled version, that is, if the reference is to a block which is pointed to by a placeholder, or a block which is in S, the replacement made by the default algorithm is put aside into S (we say that the replacement made by the default algorithm is "saved" by the application control).
- whenever a virtual block is accessed, replace a block in S.

In particular, the framework applies to various policies that approximate LRU, such as FIFO with second chance, or two-hand clock [LL82] used in virtual memory systems. For example, with FIFO with second chance, a FIFOSC-SP can be

constructed, where there will be a FIFO list with reference bits and a separate list S. The FIFO list is used to choose a candidate block, and the candidate's process is notified to choose a replacement block. Swapping and placeholders are used; the alternative block (the replacement chosen by the user process) is put at the candidate block's position, and a placeholder for the candidate block is built. Then, when the alternative or any block in S is accessed, the algorithm deletes from the FIFO list the block that would be replaced had the access been a miss, and inserts it in S. The conversion from a placeholder to a virtual block occurs when the alternative block is accessed. Finally, whenever a virtual block is accessed, the replacement is always chosen from S.

Limitations of the analysis An important assumption in the above proof is that the reference sequence ω is not affected by applications' replacement decisions. This is not necessarily true in practice. Replacement decisions affect future cache hits and misses, which determine when and how long a process is stalled for disk I/Os, and thus affect the CPU scheduling of processes. On the other hand, LRU-SP still satisfies the principles in Section 2.2.3 most of the time in a real system, as demonstrated in experiments on ACFS (Section 4.4.4).

Another limitation of the analysis in this section is that it is concerned with miss ratio only. In practice, where elapsed time is more important, the load on disk will affect an application regardless of its miss ratio, as will be shown in Section 4.4.4.

Proof of competitive ratio $2 * (p + 1)$

The above proof shows that LRU-SP is at least as good as the allocation done by global LRU. But how far is LRU-SP from optimal? Here I conduct a brief competitive analysis [ST85] of LRU-SP.

The main result of the analysis is that, if each process uses a good replacement algorithm, called a *phase-optimal* algorithm, then the number of all processes' misses under LRU-SP allocation is at most $2 * (p + 1)$ times the number of misses under optimal cache management, where p is the number of processes involved.

The phase-optimal algorithm is defined as follows. Suppose the cache can hold k blocks. Given $\omega = (r_1, r_2, \ldots, r_n)$, we partition it into phases $R_1, R_2, \ldots, R_l$. The first phase starts from r_1 and runs until it reaches a reference to the $k+1$st distinct block. The second phase starts at the end of the first phase, also running until it reaches a reference to the $k+1$st distinct block. All subsequent phases are derived similarly. The phase-optimal algorithm always replaces a process's block that is referenced furthest in the future, among all its blocks that have not been accessed yet in the current phase.

Conceivably, we can have an algorithm which uses LRU-SP as the allocation policy and phase-optimal as each process's replacement policy. Let's call it the LRU-SP with phase optimal (LRU-SP-PO) algorithm.

LRU-SP-PO is in between on-line and off-line algorithms. On-line algorithms assume that they do not have any knowledge about future accesses, while off-line algorithms assume that they know the entire access sequence before the first access starts. LRU-SP-PO is not strictly on-line because it has the knowledge of each process's future file accesses, abstracting the situations when applications have knowledge of their own access pattern. On the other hand, it is not strictly off-line because it does not have knowledge of how processes' access sequences will be interleaved in the future.

The off-line optimal cache replacement policy in this case always replaces the block that is referenced furthest in the future (called the RMIN global replacement algorithm).

Theorem 2 *Given an access sequence $\omega = (r_1, r_2, \ldots, r_n)$, which is an interleaving of accesses from p processes $P_1, P_2, \ldots, P_p$, with each block appearing in exactly one process's references, let $F_{LRU-SP-PO}$ be the number of cache misses under the LRU-SP-PO algorithm, and F_{RMIN} be the minimum number of cache misses. Then for any cache size k,*

$$F_{LRU-SP-PO} \leq 2 * (p+1) * F_{RMIN}$$

Let M_i be the set of file blocks accessed in phase i. Since each phase references

k distinct blocks, $\|M_i\| = k$, for every i. The following lemma is a well known fact [ST85].

Lemma 3 *The number of cache misses performed by the optimal replacement algorithm is at least half of the sum of the number of cache blocks that are referenced in phase i but not in phase $i-1$, for $i = 2, \ldots, l$. That is, let $M_i - M_{i-1}$ be the set of blocks that are referenced in phase i but not in phase $i-1$, then*

$$\sum_{i=2}^{l} \|M_i - M_{i-1}\| \leq 2 * F_{RMIN}$$

Proof. Because the number of different cache blocks that are accessed during phases $i-1$ and i is $k + \|M_i - M_{i-1}\|$, the optimal algorithm has to incur at least $\|M_i - M_{i-1}\|$ misses during phases $i-1$ and i, $i = 2, \ldots, l$. ■

Observe that the set of cache blocks of LRU-SP-PO at the end of phase i is exactly M_i, because during the execution of phase i, any block that has been accessed in phase i would not be chosen as a candidate, nor replaced by phase-optimal.

Let us define a cache miss in phase i that accesses, for the first time in phase i, a block that is not in phase $i-1$ a *new* cache miss. Let us further say that a cache miss to block A *immediately generates* a cache miss to block B if B is the replacement block for A and the cache miss to block B is the first access to B after it is replaced by A. Let us further say that one cache miss, a, *generates* another, b, if a immediately generates c and c immediately generates b, or a immediately generates c and c immediately generates d and d immediately generates b, and so on.

Thus, according to the above observation, LRU-SP-PO's misses in phase i are either new misses, or ones generated by a new miss. The number of new misses is exactly $\|M_i - M_{i-1}\|$. The following lemma bounds the number of generated misses.

Lemma 4 *Under LRU-SP-PO, in any phase i, a new miss can generate at most p cache misses during the same phase, where p is the number of processes.*

Proof. Suppose when a cache miss A happens, a process P_j is chosen to give up a block, and P_j replaces B. Since P_j uses phase-optimal, B is referenced furthest in the

future among all blocks of P_j that have not been accessed in phase i. That means if B is accessed again in phase i, all P_j's cached blocks have already been accessed. In other words, during phase i, when B is accessed again, P_j would no longer have a block appearing at the LRU end of the LRU-SP list, and therefore, would no longer be chosen for replacement. Since there are p processes, the number of cache misses generated by A can be at most p before the phase ends. ■

Thus, the number of generated misses in phase i can be at most $p * \|M_i - M_{i-1}\|$. Combining the above observations and Lemma 3 gives Theorem 2. ■

Related theoretical work on caching Researchers in the theory community have done a number of studies on caching. Sleator and Tarjan first introduced the concept of competitive analysis in [ST85]. [FKL*91] showed that the best attainable competitive ratio among all deterministic on-line algorithms is k (the size of the cache), and among randomized algorithms, H_k (the harmonic number of k). In practice, where $p << k$, LRU-SP-PO can achieve a much better competitive ratio. This demonstrates the potential of exploiting application-specific knowledge. In fact, [BGV95] presents a randomized algorithm which exploits application-specific knowledge to achieve a competitive ratio of $2 * H_{p-1} + 2$. Unfortunately, their algorithm does not satisfy the criteria for allocation policies in real systems and cannot deal with non-optimal replacement policies from applications.

2.4 Simulations

This section presents trace-driven simulation studies of application-controlled file caching. The purpose of the simulations is two-fold, to investigate the potential performance benefits of application-controlled file caching in two-level replacement, and to evaluate the performance of LRU-SP on file system traces and validate the theoretical results. The simulations use two sets of traces. One was collected locally by tracing various applications running on a DEC 5000/200 workstation. The other is the Sprite file system traces from the University of California at Berkeley [BHK*91].

2.4.1 Simulation Environment

I built a trace-driven simulator to simulate the behavior of the file cache under various replacement policies[1]. In my simulation I only considered accesses to regular files — accesses to directories were ignored for simplicity, the justification being that file systems often have a separate caching mechanism for directory entries. I also assumed that the file system has a fixed size file cache; in other words, I do not simulate the dynamic reallocation of physical memory between virtual memory and file system (which happens in some operating systems [Mic90, Nel88]). The block size is 8K bytes in the simulations.

The simulator was validated using Ultrix traces. The machine has a 1.6MB file cache. I measured the actual number of read misses using the UNIX `time` command. The simulation results were within 3% of the real result except for link-editing, for which the simulator predicted 7% fewer misses. This is because the simulator ignored directory operations, which were more common in the link-editing application.

To evaluate LRU-SP, I compared it with two policies: existing kernel-level global LRU without application control, and the ideal off-line optimal replacement algorithm. The former is used by most file systems, while the latter sets an upper bound on how much miss ratio can be reduced by improving the replacement policy. The performance criterion is miss ratio: the ratio of total file cache misses to total file accesses.

2.4.2 Results with Ultrix Traces

I instrumented the Ultrix 4.3 kernel to collect the first set of traces. When tracing is turned on, file I/O system calls from every process are recorded in a log, which is later fed to the simulator.

Traces were gathered for three application programs, chosen to represent I/O intensive applications that do not perform well under the LRU replacement policy:

[1]The traces and simulator are available via anonymous ftp from ftp.cs.princeton.edu: pub/people/pc.

- `postgres`: the program `postgres` is a relational database system developed at the University of California at Berkeley [The93]. I used version 4.1. The system is traced running a benchmark from the University of Wisconsin, which is included in the release package. The benchmark mainly consists of selection, join and aggregate queries; when there is no index built for a relation, queries involving the relation often access the tuple file sequentially. The benchmark takes about fifteen minutes on the workstation.

- `cscope`: the program `cscope` is an interactive tool for examining C sources [Ste85]. It first builds a database of all source files, then uses the database to answer the user's queries, such as locating all the places a function is called. I traced cscope when it was being used to examine the source code of the kernel. The trace recorded four queries, taking about two minutes.

- `link-editing`: The Ultrix linker is known as being I/O-bound, making many small and random file accesses. It never reuses data, which means that after it reads all data from a file block, it no longer needs the block. I collected the I/O traces when link-editing an operating system kernel twice, which takes about six minutes. I also collected traces when linking some programs with the X11 library.

I collected traces both when the applications were running separately and when they were run concurrently.

Benefits of Application Control

To see whether application specific policies can improve hit ratio, I experimented with the single application traces.

- `postgres`: even a simple heuristic like *sequential*, which, for each file, replaces the block before the current file pointer, can reduce `postgres`'s miss ratio (Figure 9).

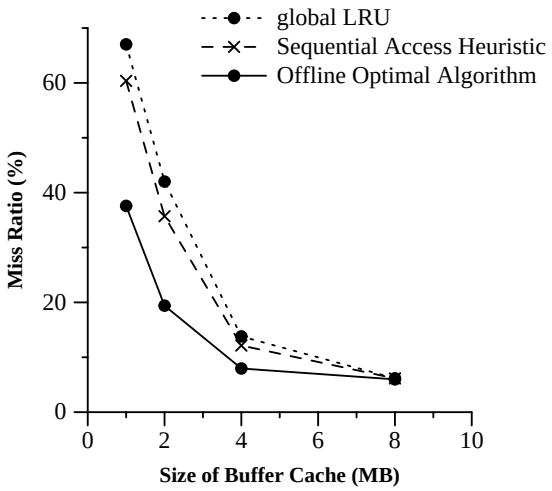

Figure 9: Simulated Miss Ratios for `postgres`

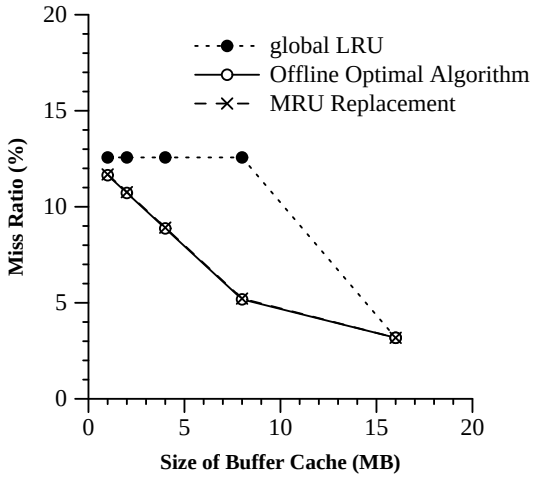

Figure 10: Simulated Miss Ratios for `cscope`

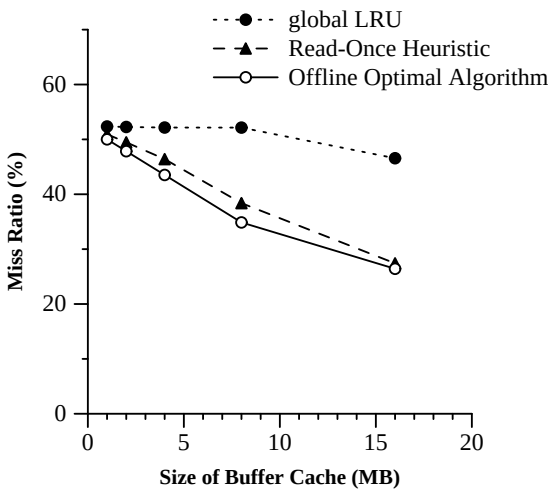

Figure 11: Simulated Miss Ratios for Linking Kernel

- `cscope`: `cscope` actually has a very simple access pattern. It reads the database file sequentially from beginning to end to answer each query. Using MRU on the database file reduces the miss ratio significantly (Figure 10). In fact, global LRU's miss ratio of 12.5% does not mean it performs well; the ratio is instead attributable to the fact that the size of file accesses is 1KB, and the file block size is 8KB. Global LRU is useless for caches smaller than 10MB because the database file in this trace is about 10MB.

- `link-editing`: the linker makes many small random file accesses. However, it knows that after it has read all data from a file block, it no longer needs the block and can replace the block first. This policy is called *read-once*. Figure 11 shows the benefit of read-once on the kernel linking trace. Figure 12 shows the effect of both *sequential* and *read-once* on linking with X11 library; this trace is small and a 4MB cache is large enough for it.

These results confirm the observation in Section 1.2 that, using intelligent replacement policies, applications can significantly reduce their miss ratios.

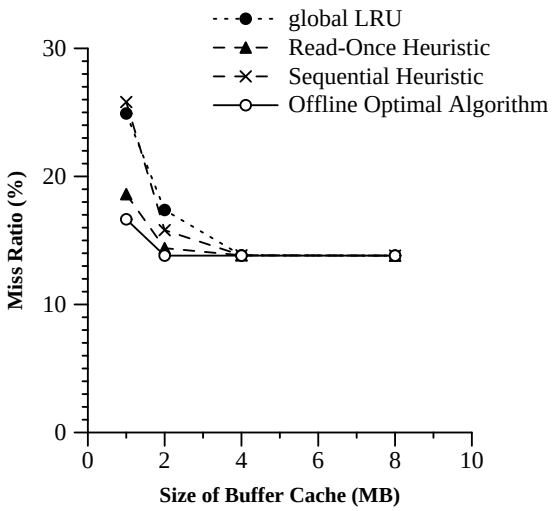

Figure 12: Simulated Miss Ratios for Linking with X11

Evaluating LRU-SP

I also collected a trace when the three applications (`postgres`, `cscope` and `link-editing` of the kernel) were run concurrently. In this trace, I simulated each application running its own user-level policy as discussed above. The result is shown in Figure 13. Since the applications' user-level policies are not optimal, I also simulated the case when each application uses the off-line optimal algorithm (RMIN) as its user-level policy. This yields the curve directly above the bottom curve.

As shown here, LRU-SP, coupled with appropriate user level policies, can improve the hit ratio for multiprocess workloads.

Effect of placeholders I also performed an experiment to measure the benefit of using placeholders. I collected a trace of `postgres` and `link-editing` of the kernel running concurrently, and simulated the miss ratio of `postgres` which occurs when `link-editing` makes the worst possible replacement choices and `postgres` simply follows LRU. I simulated three situations: the kernel uses LRU-SP as the allocation policy, the kernel uses LRU with swapping only (without placeholders) as the allocation policy, and the kernel uses global LRU replacement policy (i.e. no application control). Figure 14 shows the miss ratio of `postgres` in the three cases.

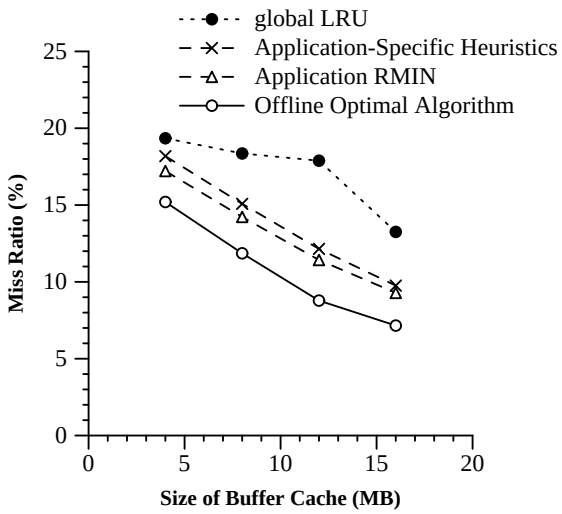

Figure 13: Simulated Miss Ratios for a Multi-Process Workload

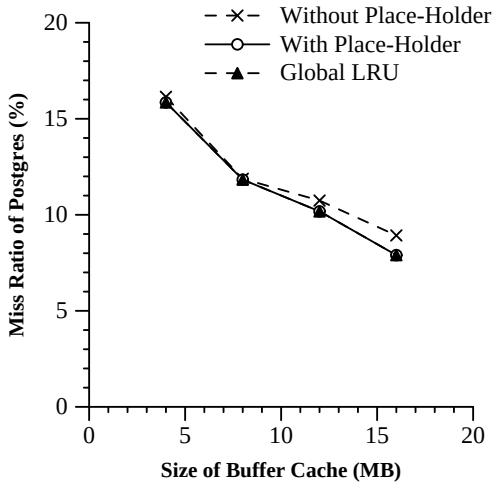

Figure 14: Benefit of Using Place-Holders

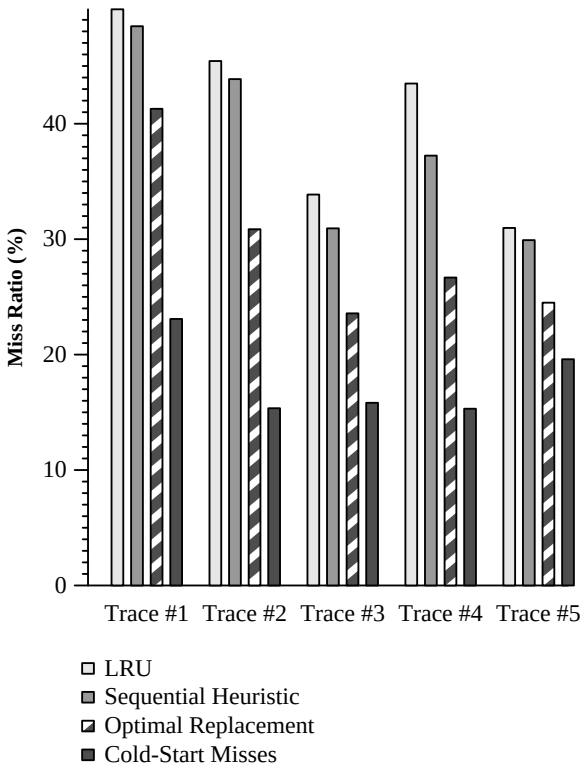

Figure 15: Averaged Results from Sprite Traces

Without placeholders, the miss ratio of `postgres` is noticeably increased by the other application's bad replacement decisions. In contrast, with placeholders, `postgres` has the same miss ratio as under global LRU (the two curves are in fact identical), as predicted by the theoretical analysis.

2.4.3 Results with Sprite Traces

The second set of traces is from the UC Berkeley Sprite File System Traces [BHK*91]. There are five sets of traces, recording about 40 clients' file activities over a period of 48 hours (traces 1, 2 and 3) or 24 hours (traces 4 and 5).

I focused on the performance of client caching. In a system with a slow network (e.g. Ethernet), client caching performance determines the file system performance on each workstation. Furthermore I ignored kernel file activities in these traces, because Sprite's virtual memory system swaps to remote files. In the simulation I set the

client cache size at 7MB, which is the average file cache size reported in [BHK*91].

Since these traces do not contain process-ID information, the simulation assumes that each file is managed by a separate pseudo process. Each process uses the *sequential* heuristic, which replaces the block with the largest logical block number first because it expects the file to be accessed sequentially.

Figure 15 shows average miss ratios for global LRU, LRU-SP with *sequential* heuristic, and optimal replacement. Average cold-start (compulsory) miss ratios are also shown.

As can be seen, LRU-SP with *sequential* improves the hit ratio for some traces. In fact simulations show that it improves the hit ratio for about 10% of the clients, and the improvements in these cases are between 10% and over 100%.

Overall, these results show that two-level replacement with LRU-SP is a promising approach to enable application control of replacement and to improve the hit ratio of the file buffer cache.

2.5 Summary

The key in application-controlled file caching is the allocation policy. This chapter proposes a two-level replacement scheme, where the kernel controls allocation of cache blocks and user processes control local replacements. This chapter describes an allocation policy for the kernel, called LRU with Swapping and Placeholders, and evaluates its performance using trace-driven simulations.

The policy, LRU with Swapping and Placeholders (LRU-SP), satisfies the criteria for allocation policies. Under the assumption that the interleaving of file references from multiple processes is not affected by replacement decisions, it guarantees that processes which use good replacement policies do not suffer more misses but reduce their miss ratio whenever possible. Under the same assumption, it also guarantees that processes which are unwilling or unable to manage their file caches will perform at least as well as they did under the traditional global LRU policy, and that a process which uses a bad replacement policy (worse than LRU) cannot cause other

processes to suffer more misses. The key benefit of LRU-SP is the *guarantee* that good application-controlled replacement policies will not degrade, but rather will improve, the file cache hit ratio of the whole system.

The chapter presents the proof of these guarantees under the assumption that the interleaving of accesses from processes is not affected by replacement decisions. The proof process also shows that the techniques, swapping and placeholders, are not unique to LRU, and can be applied to any default global replacement policies. An analysis of LRU-SP's competitive ratio shows that when applications use phase-optimal replacement algorithms, LRU-SP's elapsed time is at most $2(p + 1)$ times that of the optimal policy, where p is the number of processes.

To investigate the performance of application-controlled file caching and LRU-SP, I simulated the two-level replacement scheme and LRU-SP using two sets of file system traces: one collected locally by tracing several applications on a DEC 5000/200 workstation, and the other from the Sprite traces from University of California at Berkeley. The simulation results show that even simple application-tailored policies can reduce miss ratios significantly. In addition, simulations show that LRU-SP works well in multi-process environments: the combination of the allocation policy and application-tailored replacement policies can substantially reduce the overall file cache miss ratios over the traditional global file caching approach. The results also show that placeholders are necessary to prevent performance degradation caused by misbehaving processes.

Chapter 3

Integrating caching and prefetching

Prefetching is an effective way to overlap I/O and computation. As the gap between disk speed and microprocessor speed grows wider, prefetching becomes increasingly important. However, traditional file systems only employ limited one-block lookahead prefetching, which only benefits sequential I/O on large files.

There are two reason why more sophisticated prefetching is not used in traditional file systems. One is the lack of sufficient information on future file accesses. The other is that the interaction between caching and prefetching is complicated, and if not carefully handled, can cause prefetching to harm, rather than help, the applications' performance.

Recent advances in compiler and prediction technology have made user-level information on future file accesses more readily available [PGS93, CKV93, PZ91, TD90]. This makes possible the use of a more aggressive prefetching technique utilizing user-level information. Thus, the first problem is solved. On the other hand, the second problem, that of the interaction between caching and prefetching, needs careful theoretical study. That is the topic of this chapter.

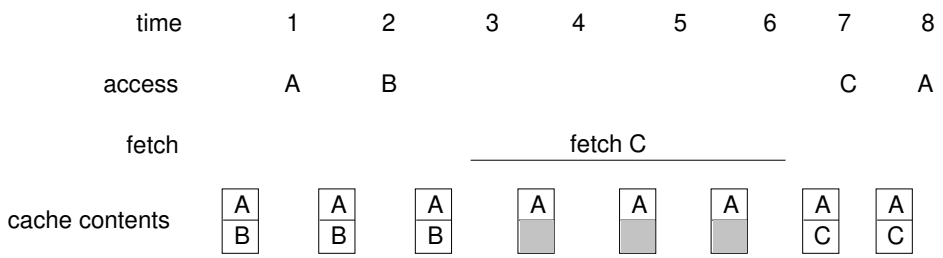

Figure 16: An example under no-prefetch policy.

3.1 Introduction

Prefetching and caching are two well-known approaches for improving the performance of file systems. Although they have been studied extensively, most studies on prefetching have been conducted in the absence of caching or for a fixed caching strategy. The interaction between prefetching and caching is not well understood.

The main complication is that prefetching file blocks into a cache can be harmful even if the blocks will be accessed in the near future. This is because a cache block needs to be reserved for the block being prefetched at the time the prefetch is initiated. The reservation of a cache block requires performing a cache block replacement earlier than it would otherwise have been done. Making the decision earlier may harm performance because new and possibly better replacement opportunities open up as the program proceeds.

3.1.1 An Example

Consider a program that references blocks according to the pattern "ABCA." Assume that the file cache holds two blocks, that fetching a block takes four time units, and that A and B are initially in the cache.

As shown in Figure 16, a no-prefetch policy (using the optimal off-line replacement algorithm) would hit on the first two references (A and B), then miss on the reference

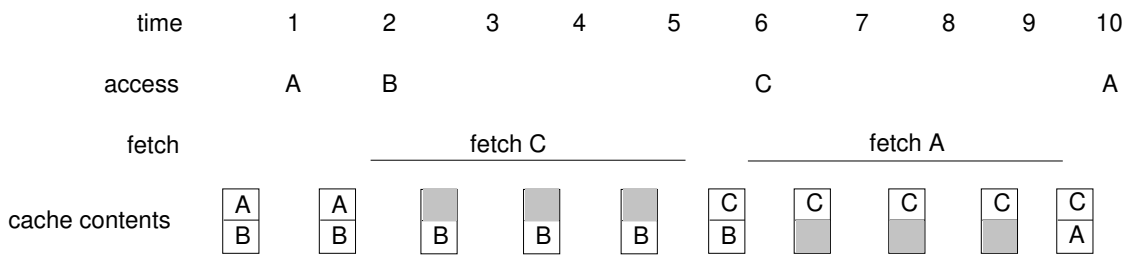

Figure 17: Same example under always-prefetch policy.

to C, discarding B, and finally hit on A. The reference to C is stalled for four time units while C is fetched and B is discarded. The execution time of the no-prefetch policy would therefore be *eight* time units (one for each of the four references, plus four units for the miss.)

By contrast, as Figure 17 shows, a policy that prefetches whenever possible (while making optimal replacement choices) takes *ten* time units to execute this sequence. After the first successful access to A, a prefetch of C is initiated, discarding A. This prefetch hides one unit of the fetch latency, so the access to C stalls for only three units. Once C arrives in memory, the algorithm initiates another prefetch, bringing in A and discarding B, while accessing C. The next reference, to A, stalls for three time units, waiting for the prefetch of A to complete. This algorithm uses ten time units, one for each of the four references, plus two stalls of three time units each.

This example illustrates that aggressive prefetching is not always beneficial. The no-prefetch policy fetched one block, while the aggressive prefetching algorithm fetched two. The price of performing an extra fetch outweighs the latency-hiding benefit of prefetching in this case. On the other hand, prefetching might have been beneficial under slightly different circumstances. If the reference stream had been "ABCB" instead of "ABCA," then aggressive prefetching would have outperformed the no-prefetch policy. Thus we see that aggressive prefetching is a double-edged sword: it hides fetch latency, but it may increase the number of fetches.

Although the above example is hypothetical, the problem exists in real systems [PGS93, WYT93]. Existing approaches to deal with the problem include: fetching only one or a few blocks ahead (for example, in traditional file systems), no prefetching if the number of prefetched but not yet referenced blocks exceeds a certain number [PGS93], no prefetching unless the time until actual access is below a certain threshold [PGG*95]. None of these approaches solves the problem.

The question that needs to be answered is: given detailed information about future file accesses, what is the optimal combined prefetching and caching strategy that gives the minimum elapsed time?

Below I phrase the problem in a more formal and theoretical setting, and give the solutions.

3.2 Basics of Prefetching and Caching

In this section, I explain the model on an intuitive level, describe four rules that an optimal policy must follow, and, after defining the problem formally, give an exponential optimal algorithm.

3.2.1 Problem Definition

Consider the execution of a program that makes a known sequence $(r_1, r_2, \ldots, r_n)$ of references to data blocks. The program executes one reference per time unit. The cache can hold k blocks, where $k < n$. If a reference hits in the cache, it can be satisfied immediately; otherwise, the missed block has to be fetched from the backing store.

The system can either fetch a block in response to a cache miss (*on-demand fetch*), or it can fetch a block before it is referenced in anticipation of a miss (*prefetch*). It takes F time units to fetch a block from the backing store into the cache. Only one fetch can be in progress at any given time. When a fetch is initiated, some block must be discarded from the cache to make room for the incoming block; while the fetch is in progress, neither the incoming block nor the discarded block is available for access.

When the program tries to access a block that is not available in the cache, the program stalls until the block arrives in the cache. The stall time is either F if the block is fetched on-demand, or $max(0, F - i)$ if the fetch was started i time units in advance. The total elapsed time for a reference string is the total reference time (or the number of references) plus the sum of stall times.

The goal of an integrated prefetching and caching policy is to make the decisions

- when to fetch a block from disk,
- which block to fetch, and
- which block to replace when the fetch is initiated,

so that the total elapsed time is minimized.

Clearly, this model corresponds to a system with one disk or one file server, and it considers only read references. The time unit represents the fact that there is generally CPU time spent between two consecutive file references — the CPU time includes the time to copy the accessed file data from kernel address space to user address space, and the time for the application to consume the file data. The model simplifies the real situation by assuming that the CPU time between every two file references (called "reference time") is the same, and it is one time-unit. As I will show later in my simulations, this simplification approximates real systems reasonably well.

3.2.2 Rules for Optimal Prefetching and Caching

This section presents four rules that an optimal prefetching and caching strategy must follow. These rules are mandatory in the sense that any off-line algorithm which does not comply may be easily transformed into one that does, with no increase in elapsed time. Thus, the search for optimal algorithms can be restricted to those that follow the rules. Correctness of these rules is easily proved.

1. *Optimal Prefetching: every prefetch should bring into the cache the next block in the reference stream that is not in the cache.*

2. *Optimal Replacement: every prefetch should discard the block whose next reference is furthest in the future.*

   The first two rules determine what to do, once the decision to prefetch has been made. However, they say nothing about when to fetch — the next two rules speak on that question.

3. *Do No Harm: Never discard block A to prefetch block B when A will be referenced before B.*

   A prefetch that disobeys this rule does more harm than good — it can only increase the program's running time. Unfortunately, existing prefetching algorithms do not always satisfy this requirement, because they separate caching from prefetching, and separate cache replacement decisions from prefetching decisions.

4. *First Opportunity: Never perform a prefetch-and-replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.*

   The algorithm must perform each operation at the first opportunity. A new opportunity may arise when either (a) a fetch completes, or (b) the block that would be discarded (according to the "optimal-replacement" rule) was just referenced in the previous time unit. Condition (b) is the only circumstance under which the "optimal-replacement" rule might change its recommendation about which block to discard.

Note that the above rules hold for the situation modeled here, i.e. file prefetching on a system with one disk or server. They may not apply to all situations in which prefetching is used; in particular, they do not apply to hardware prefetching, where the cache has limited associativity. However, variations of the above rules can be formulated for other situations.

3.2.3 Theoretical Model Definition

In this section, I review the model in a slightly more formal fashion. The input to a prefetching algorithm is a *reference string* $(r_1, r_2, \ldots, r_n)$ representing the ordered sequence of file-block accesses to be performed. Recall that the cache holds k blocks, that each access takes unit time, and that fetching a block from the backing store takes F time units. (In expressions involving F and k, it may be helpful to think of k as the time to make k references.)

Imagine that there is a cursor, which advances at a fixed rate along the reference string until it hits an access, say r_j, to a file block that is not present in the cache. The cursor then stays at reference r_j until the block arrives in cache. If the processor waits for block r_j to arrive, we say that *the cursor stalls at reference* r_j. The total amount of time the cursor spends stalled is called the *stall time*.

If the prefetch for r_j is initiated when the cursor is at reference r_i, and $i + F \leq j$, then the cursor does not stall at r_j, because the block will have already arrived in memory by the time it is referenced. Otherwise, the cursor stalls for $i + F - j$ time units.

The model assumes that prefetches are serialized; if a prefetch is initiated at time t, then the next prefetch cannot be initiated earlier than time $t + F$.

It also assumes that if block A is replaced in order to bring block B into the cache, then block A becomes unavailable for access at the moment the prefetch is initiated, and block B becomes available only when the prefetch terminates (F time units later).

Goal of Prefetching Strategy: To minimize the elapsed time of executing the reference string, where the elapsed time is the time to perform the n accesses in the reference string (n time units) plus the stall time.

3.2.4 An Optimal Dynamic Programming Algorithm

In principle, the optimal prefetching strategy is computable quite simply by dynamic programming as follows [1].

Let $opt(R_n, S, c)$ be the optimal cost of servicing the request sequence $R_n = (r_1, \ldots, r_n)$, ending up with the set of blocks S in the cache, and with the last prefetch terminating when the cursor is at position c in the sequence. Note that it is necessarily the case that $r_i \in S$ for each $c \leq i \leq n$.

Initially, $opt(\emptyset, S_0, 0) = 0$ ($\emptyset$ is the empty sequence) and $opt(\emptyset, S, 0) = \infty$ for $S \neq S_0$, where S_0 is the set of blocks initially in the cache.

$opt(R_{n+1}, S, c)$ is then computed inductively as follows.

- If $r_{c+1}, \ldots, r_{n+1}$ are not all in S, then $opt(R_{n+1}, S, c) = \infty$.
- If $r_{c+1}, \ldots, r_{n+1}$ are all in S, we have two cases:
  - If $c < n$, then $opt(R_{n+1}, S, c) = opt(R_n, S, c) + 1$, where the 1 accounts for the added cost of accessing r_{n+1}.
  - If $c = n$, then the prefetch for r_{n+1} terminated immediately before the access to it, and there may have been stall time associated with that fetch. The previous prefetch ended earlier, when the cursor was, say, at c', and the set of blocks in the cache was $S - p + q$. The prefetch for p was then initiated when the cursor was at c'', where $n - F \leq c'' \leq n$. It is necessarily the case that $r_{c'+1}, \ldots r_n \in S - p + q$ and that $r_{c''+1}, \ldots, r_n \in S - p$. Hence, $opt(R_{n+1}, S, c)$ is the minimum of

    $$opt(R(n), S - p + q, c') + F - (n - c'') + 1.$$

    over blocks $q \notin S$, and c' and c'', such that $c' \leq c''$, $n - F \leq c'' \leq n$, $r_{c'+1}, \ldots r_n \in S - p + q$ and $r_{c''+1}, \ldots, r_n \in S - p$.

Clearly, it is not feasible to compute the optimal schedule this way. The number of possibilities for S alone is $\Omega\left(\binom{n}{k}\right)$.

[1] Joint work with Anna Karlin.

3.3 Aggressive and Conservative

The goal of this study is to find a simple *near-optimal* policy for prefetching and caching that minimizes the total elapsed time of a known sequence of references. The dynamic programming algorithm in Section 3.2.4 is unfortunately too time-consuming. Below I describe two algorithms, *conservative* and *aggressive*, and show that they are close to optimal [2].

3.3.1 Policy Alternatives

The four rules in Section 3.2.2 provide some guidance about when to prefetch; once a decision to prefetch has been made, they uniquely specify what should be prefetched and what should be discarded.

Thus, we can describe a prefetching policy as answering a series of yes/no questions, with each question of the form "Should I prefetch now?" As the program executes, a series of opportunities to prefetch arise, and the policy is asked whether to take each opportunity or to let it pass. I now present two policies, *conservative* and *aggressive*. Both follow the four rules described in Section 3.2.2.

The "Conservative" Strategy

The *conservative* prefetching strategy tries to minimize the elapsed time while performing the minimum number of fetches. It performs exactly the same replacements as the optimal off-line demand paging strategy MIN, except that each fetch is performed at the earliest opportunity consistent with the four rules in Section 3.2.2.

The "Aggressive" Strategy

The *aggressive* prefetching strategy always prefetches the next missing block at the earliest opportunity consistent with the four rules of Section 3.2.2. In order to bring in the next missing block, aggressive replaces the block whose next reference is furthest

[2] Some of the results here are joint work with Anna Karlin.

in the future. Notice that aggressive is not mindlessly greedy — it waits until there is a block to replace whose next reference is after the request to the missing block.

3.3.2 Analysis of Conservative

The main results of the analysis of conservative are the following:

1. On any reference string R, conservative performs exactly the same number of fetches as the optimal off-line demand paging algorithm does on R.
2. On any reference string R, the elapsed time of conservative is at most twice that of the optimal prefetching algorithm.
3. The previous bound is nearly tight: there is a reference string R' on which the elapsed time of conservative is $2/(1 + (1/F))$ times that of the optimal prefetching algorithm.

Bound on Number of Fetches

By definition, conservative performs exactly the same number of fetches as the optimal demand paging algorithm RMIN does (RMIN is the algorithm that always replaces the block that is referenced furthest in the future). However, its elapsed time can be much smaller, since it may be able to overlap these fetches with the request sequence.

Bounds on Elapsed Time

Although conservative places the minimum possible load on the disk, its elapsed time can in some cases be nearly twice optimal. The following example shows this.

Example : Suppose that F divides k, that the algorithms start with blocks $b_1, \ldots, b_k$ in the cache and that the reference string is a cyclic pattern, $b_1, \ldots b_k, b_{k+1}, b_{k+2}, \ldots, b_{k+k/F}$ repeated many times. It is easy to see that in this case conservative will not be able to overlap fetches with references. Since k/F fetches will be done per pass through the cycle, and conservative will stall for F time units on each, its elapsed time will be at least $2k$ per pass through the cycle. On the other

hand, it is not difficult to see that by keeping the missing blocks at least F references apart, this sequence can be prefetched so that all fetches overlap references, and the elapsed time is $k(1 + (1/F))$ per pass through the cycle. Therefore, in this example, the elapsed time of conservative is at least $2/(1 + (1/F))$ times that of the optimal algorithm. ■

It is also easy to see that conservative's elapsed time is at most twice optimal.

Lemma 5 *On any reference string R, the elapsed time of conservative is at most twice the elapsed time of the optimal prefetching strategy.*

Proof. Any prefetching strategy, and in particular the optimal one, *opt*, performs at least as many fetches as conservative. Let the m be the minimum number of cache misses, and l be the length of R. Since *opt* performs the same number of references as conservative and at best entirely overlaps its fetches with references, *opt*'s elapsed time is at least $\max(l, m * F)$, while conservative's elapsed time is at most $l + m * F$. Therefore, *opt*'s elapsed time is at least half of that of conservative. ■

3.3.3 Analysis of Aggressive

The main results of the analysis of aggressive are the following:

1. On any reference string R, aggressive performs at most the number of fetches performed by LRU demand paging.

2. On any reference string R, and for $F \leq k$, the elapsed time of aggressive is at most $(1 + F/k)$ times that of the optimal prefetching algorithm. If $F > k$, the elapsed time of aggressive is at most twice that of the optimal prefetching algorithm.

3. The previous bound is nearly tight: there is a reference string R' on which the elapsed time of aggressive is $1 + (F - 2)/(k + 1)$ times that of the optimal prefetching algorithm.

Bound on Number of Fetches

It is possible to show that aggressive performs at most the number of fetches performed by the LRU replacement policy. The proof is essentially the same as the proof of the domination lemma (shown later in this section). This suggests that on reference patterns with good locality, aggressive may have near-optimal performance, since it will not perform an excessive number of fetches, and it will often be able to overlap those fetches with references.

Aggressive: Lower Bound on Elapsed Time

The following example shows that aggressive can have an elapsed time which is nearly $1 + F/k$ times that of the optimal prefetching strategy.

Example : Suppose that $F = k - 2$. Suppose that the algorithms start with blocks $b_1, \ldots b_k$ in the cache. The reference string begins with $b_1, b_2, \ldots b_k, b', b_1$. Aggressive will begin prefetching b' immediately after the request to b_1, replacing b_1. This prefetch will terminate immediately before the request to b', at which point aggressive will begin prefetching b_1.

The optimal decision would have been to prefetch b' after the request to b_2, since in the remainder of the sequence b_2 will never be requested again. Consequently, aggressive incurs $F - 2$ units of stall time on the prefetch of b_1, whereas the optimal algorithm incurs no stall time by waiting to prefetch b' until b_2 can be replaced. If we extend the sequence in a way such that b_2 is not accessed again, aggressive will replace b_2 on the prefetch of b_1, and the optimal algorithm and aggressive will both have the same cache state immediately before each of them references b_1 for the second time. Therefore, we can repeat such a subsequence again and again, where each time, by waiting 1 extra time unit, the optimal algorithm needs do one fewer prefetch per $k+1$ accesses than aggressive has to do.

Since the optimal algorithm incurs $k+1$ time units for every $k + F - 1$ time units incurred by aggressive, the above statement is true. Note that the specific choice of $F = k - 2$ here is not important. For any value of F, it is quite easy to construct reference strings on which aggressive has an elapsed time close to $\min(2, 1 + F/k)$

times that of the optimal algorithm. ■

I now show that this is essentially the worst case for aggressive.

Aggressive: Upper Bound on Elapsed Time

In what follows, we will need to divide the reference string into *phases*. The first phase starts with the first reference, and ends immediately before the reference to the $(k+1)$st distinct block. In general, the ith phase starts at the end of the $(i-1)$st phase, and ends immediately before the reference to the $(k+1)$st distinct block in the ith phase. (These are defined in the same way as in Section 2.3.2.) Note that phases are a property of the reference string, and hence the choice of prefetching strategy is irrelevant to their definition.

Theorem 3 *On any reference string R, the elapsed time of aggressive is at most the elapsed time of the optimal prefetching strategy plus F times the number of phases in the reference string.*

This leads to the following corollary.

Corollary 3.3.1 *On any reference string R, the elapsed time of aggressive is at most* $\min(1 + F/k, 2)$ *times the elapsed time of the optimal prefetching strategy.*

Proof. Suppose that $F \leq k$. Since each phase lasts for at least k references, the number of phases is at most the length of R, l, divided by k. Since the elapsed time of the optimal algorithm, *opt*, is at least l, Theorem 3 means that the elapsed time of aggressive is at most $(1 + F/k)$ times that of *opt*.

For the other case, $F > k$, observe that it follows from the definition of phases that *opt* has to perform at least one fetch per phase. This fetch takes F time units. Since in this case aggressive incurs at most one extra fetch per phase, its elapsed time is at most twice *opt*'s. ■

In practice, F/k is small (usually < 0.1). Thus, aggressive is very close to optimal.

Proof of The Upper Bound

The proof for Theorem 3 is complicated [3]; however, the idea is not difficult. Let *opt* be the optimal prefetching strategy. We show inductively that aggressive can fall behind *opt* by at most an additional F time units each phase. Consider the first phase. At time 0, both algorithms are missing the same subset of blocks, say i of them, from the first phase. Therefore, both algorithms have in their cache i blocks that will not be referenced in the first phase. Since both algorithms always replace the block whose first reference is furthest in the future, during the first phase neither will ever replace a block that will be referenced in the first phase. Consequently, since aggressive prefetches those i blocks at the earliest opportunity, aggressive will stay ahead of *opt* at least until the cursor reaches the first reference of the second phase. Here, aggressive may stall and *opt*'s cursor may pass it by, since aggressive may have prefetched some blocks in the first phase earlier than *opt* and therefore generated missing blocks in the second phase that are earlier than those generated by *opt*. Imagine now that we "stop" *opt* at the phase boundary for F time units. This gives aggressive enough time to complete the extra fetch. More importantly, it is possible to show that after the fetch, aggressive once again has its missing blocks in positions that are at least as good as those of *opt* (where "at least as good" means that aggressive's ith missing block occurs later in the sequence than *opt*'s ith missing block). This is all we need to complete the induction.

Preliminaries: Let $c_A(t)$ be the index of the next reference at time t when running prefetching algorithm A. Let $H_A(i)$ be the set of blocks *not* present in the cache when the next reference is r_i and we are running prefetching algorithm A. Let $b_A(i,j)$ be the block in $H_A(i)$ whose first occurrence after r_i is jth, and $h_A(i,j)$ be $b_A(i,j)$'s first occurrence after r_i. We will subsequently refer to $h_A(i,j)$ as *A's jth hole*, when the context is clear. Note that j varies between 1 and $\|H_A(i)\|$, which is $n-k$, where n is the number of distinct blocks in the reference string R.

Given two prefetching algorithms A and B, we say that *A's cursor at time t*

[3] Joint work with Anna Karlin

dominates B's cursor at time t' if $c_A(t) \geq c_B(t')$. We also say that *A's holes at time t dominate B's holes at time t'* if $h_B(c_B(t'), j) \leq h_A(c_A(t), j)$ for each j. Finally, we say that *A's state at time t dominates B's state at time t'* if A's cursor at time t dominates B's cursor at time t' and A's holes at time t dominate B's holes at time t'.

The following lemma will be useful.

Domination Lemma: Suppose that A (resp. B) initiates a prefetch at time t (resp. t'), and that both algorithms prefetch the next missing block and replace the block whose next reference is furthest in the future. Suppose further that A's state at time t dominates B's state at time t'. Then A's state at time $t + F$ dominates B's state at time $t' + F$.

Proof. Suppose that A's holes (the first occurrences of each of the blocks not in the cache) at time t are at cursor positions $a_1 < a_2 < \ldots < a_{n-k}$, and that B's holes at time t' are at cursor positions $b_1 < b_2 < \ldots < b_{n-k}$. Since A's state dominates B's, $a_i \geq b_i$ for each i. Suppose that in order to prefetch b_1, B replaces a block b whose first occurrence is between b_j and b_{j+1}, and that in order to prefetch a_1, A replaces a block a whose first occurrence is between a_r and a_{r+1}. If a'_i (resp. b'_i) is the position of A's (resp. B's) ith hole after the prefetch, then for $i \leq r$, $a'_i = a_{i+1}$, $a'_r = a$, and $a'_i = a_i$, for $i \geq r + 1$. Similarly, for $i \leq j$, $b'_i = b_{i+1}$, $b'_j = b$, and $b'_i = b_i$, for $i \geq j + 1$. In order to show that the domination of A over B is preserved, we must show that $a'_i \geq b'_i$ for all i at time $t + F$ (resp. $t' + F$) for A (resp. B).

Trivially, for $2 \leq i \leq \min(r, j)$ and $i \geq \max(j + 1, r + 1)$, $a_i \geq b_i$.

If $r \geq j$, we have $a > a_r \geq b_r$, $a_r > a_{r-1} \geq b_{r-1}$,..., $a_{j+2} > a_{j+1} \geq b_{j+1}$, and $a_{j+1} \geq b_{j+1} > b$, which are the remaining inequalities needed.

If $r < j$, we must show that $a_j \geq b$, $a_{j-1} \geq b_j$, ... $a_{r+1} \geq b_{r+2}$, $a \geq b_{r+1}$. Suppose that one of these inequalities is violated. Consider the largest index that violates the condition. Then we have $a'_{i-1} < b'_{i-1} < b'_i \leq a'_i$ for some $r + 1 \leq i \leq j + 1$, where a, the replaced block, satisfies $a \leq a'_{i-1}$. This means, however, that the block at cursor position b'_{i-1} was in A's cache at the time it issued the prefetch, and its first occurrence was later than that of a, the block it chose to replace, contradicting

the assumption that A always replaces the block whose first reference is furthest in the future. Therefore, none of these inequalities is violated, and hole domination is preserved.

As for the cursor, if A's cursor stalls, then it does so at A's first hole. But in that case, either B's cursor also stalls at B's first hole or it doesn't reach B's first hole. In either case, B's cursor remains behind A's. ■

We are now ready to present the proof of Theorem 3.

Proof. Let *opt* be the optimal prefetching strategy. We prove the theorem using the following invariant by induction on the number of phases.

Invariant: During phase i, there is a time t such that aggressive's holes dominate *opt*'s holes at time $t' \geq t - iF$, and aggressive's cursor dominates *opt*'s cursor, and neither aggressive nor *opt* is in the middle of a prefetch.

The proof is by induction on the number of phases. The base case is trivial, since at time 0, both algorithms are in exactly the same state.

Suppose that the invariant is true in phase i, i.e. there is a time t in phase i such that aggressive's holes dominate *opt*'s holes at time $t' \geq t - iF$.

The first observation is that from time t until the beginning of phase $i+1$, aggressive never evicts any block that will still be referenced in phase i. Indeed, if aggressive is prefetching a block that will still be requested in phase i, then there is some block in the cache which is not requested in phase i. On the other hand, if *aggressive* is prefetching a block p that will not be requested until phase $i+1$, then since it always replaces a block whose first occurrence is later than that of the block being prefetched, no new hole can be created within phase i.

This observation implies that at all times $T > 0$, such that aggressive is still in phase i at time $t+T$, $c_a(t+T) \geq c_{opt}(t'+T)$, where $c_a(t)$ is the cursor position of aggressive at time t. The argument is that the set of holes aggressive has within phase i at time t dominates the set of holes *opt* has within phase i at time t', and no new holes are added within the phase. Since aggressive prefetches as aggressively as possible, it eliminates those holes at the earliest possible time, and therefore, aggressive's cursor

cannot fall behind *opt*'s cursor within the rest of the phase.

Let $t + t_0, t + t_1 \ldots, t + t_j$ be the cursor positions at which aggressive initiates prefetches, after time t but within phase i. If *opt* is in the middle of a prefetch at time $t' + t_l$, then let $t' + t'_l$ be the time at which that prefetch was initiated, otherwise let $t'_l = t_l$. We prove by induction on l that aggressive's state at time $t + t_l$ dominates *opt*'s state at time $t' + t'_l$. The base case is easy. *Aggressive*'s state at time t dominates *opt*'s state at time t'. If $t_0 > 0$, then aggressive cannot prefetch anything between time t and time $t + t_0$, and hence is in the best possible state during this period. Consequently, its state at time $t + t_0$ dominates *opt*'s state at time $t' + t'_0$. Suppose by the inductive hypothesis that the claim is true for l. Then by the Domination Lemma, aggressive's state at $t + t_l + F$ dominates *opt*'s state at $t' + t'_l + F$. Consider two cases:

1. $t + t_{l+1} = t + t_l + F$: Since $t'_{l+1} \leq t_{l+1}$, $c_a(t + t_{l+1}) \geq c_{opt}(t' + t_{l+1}) \geq c_{opt}(t' + t'_{l+1})$. Furthermore, since $H_{opt}(t' + t'_{l+1}) = H_{opt}(t' + t'_l + F)$, aggressive's state at $t + t_{l+1}$ dominates *opt*'s state at $t' + t'_{l+1}$.

2. $t + t_{l+1} > t + t_l + F$: In this case aggressive stops prefetching for a while, which means that its holes are in an optimal state (i.e. there are k distinct requests to the k blocks in the cache prior to any request to a hole). Furthermore, its cursor cannot stall during such a period. Therefore, during the period between $t + t_l + F$ and $t + t_{l+1}$, *opt* can at best get into the same state as aggressive.

By the same argument, aggressive's state at time $t + t_j + F$ dominates *opt*'s state at time $t' + t'_j + F$. Since aggressive is in phase $i + 1$ at time $t + t_j + F$, and $t' + t'_j + F \geq t' + t_j$, then at time $T = t + t_j + F$ in phase $i + 1$, aggressive's state dominates *opt*'s state at time $t' + t_j \geq t - iF + t_j \geq T - (i + 1)F$.

Finally, from the invariant it follows that, if aggressive finishes processing the sequence at time T, *opt* cannot finish before time $T -$ (number of phases)F, which completes the proof of the theorem. ■

3.4 Simulations

I have measured the performance of *aggressive* and *conservative*, along with several existing algorithms, by trace-driven simulation. I first discuss the existing algorithms, then introduce the traces and simulation models, and finally present simulation results.

3.4.1 Existing Approaches

I compare the *aggressive* and *conservative* algorithms with six existing algorithms for caching and prefetching:

- *LRU-demand* is a demand-paging algorithm: it only fetches a block from disk when it is accessed (i.e. no prefetching). It always replaces the block that is least recently used.

- *OPT-demand* is also a demand-paging algorithm and does not perform prefetching. It uses optimal replacement: always replaces the block that will be referenced furthest in the future.

- *LRU-OBL* is the LRU replacement algorithm combined with sequential one-block lookahead (OBL) prefetching [Wil65, Gin77, Smi78]. Traditional file systems use OBL to take advantage of the fact that files are often accessed sequentially [OCH*85, BHK*91]. OBL prefetches block $N+2$ of a file whenever the last two references to the file were to block N and block $N+1$. In *LRU-OBL*, the prefetched block is inserted in the most-recently-used end of the LRU list when the fetch finishes. Most existing file systems use *LRU-OBL* as cache management policy.

- *OPT-OBL* combines sequential OBL prefetching with the optimal replacement algorithm. That is, it always prefetches block $N+2$ of a file whenever the last two references to the file were to block N and block $N+1$, and always replaces the block that is already in cache and will be referenced furthest in the future.

- *LRU-sensible* looks into the future reference string and fetches the next missing block at the earliest possible moment, subject to the "do-no-harm" rule. Among all the blocks that can be replaced for this fetch, it chooses the least-recently-used one. *LRU-sensible* performs the best among all prefetching algorithms that use LRU as replacement principle (the proof is trivial from the "First Opportunity" rule in Section 3.2.2), and I use it to approximate existing prefetching approaches that follow the "do-no-harm" rule.

- *LRU-throttled* models prefetching approaches that do not follow the rules in Section 3.2.2 but rather use "throttling" or similar measures to integrate caching and prefetching decisions. Throttling puts an upper limit on the number of blocks that have been prefetched but not yet accessed. It is used to limit the occurrence of "thrashing," in which more precious blocks are replaced in order to prefetch less precious blocks, or prefetched file blocks are replaced before they are accessed. Unfortunately, thrashing results from the algorithms not following the "do-no-harm" rule, and throttling is an *ad hoc* approach that does not always work well.

  Given the sequence of future accesses, the prefetcher of *LRU-throttled* walks down this sequence, allocates a buffer and issues a prefetch for every missing block. It uses LRU as the replacement algorithm. It stops when one-third of the cache contains blocks that have been prefetched but haven't been referenced yet (*LRU-throttled* attempts to simulate the algorithm described in [PG94], which sets the throttling limit to 150 blocks out of a cache of 400 blocks, thus the one-third limit).

Some of the algorithms are on-line ones, including *LRU-demand* and *LRU-OBL*. Some are off-line, including *OPT-demand* and *OPT-OBL*. *LRU-sensible* and *LRU-throttled* need a certain amount of lookahead into future accesses, but do not require knowing the entire sequence. And of course, *aggressive* and *conservative* are off-line algorithms.

Among the eight algorithms, four (*aggressive*, *conservative*, *OPT-demand* and *OPT-OBL*) make optimal replacement decisions every time, and four (*LRU-demand*, *LRU-OBL*, *LRU-sensible* and *LRU-throttled*) use the LRU principle. The results will show that, as the fetch cost increases, the performance difference between the two classes of algorithms increases.

3.4.2 File Access Traces

Two sets of traces were used. One set was collected by tracing several applications running on a DEC 5000/200 workstation under Ultrix 4.3. The other is from the Sprite file system traces from the University of California at Berkeley [BHK*91].

The Ultrix traces were collected by instrumenting the Ultrix 4.3 kernel to log file accesses. I traced a set of read-dominated applications on which existing file systems perform poorly. They are:

- `cscope[1-3]`: an interactive C-source examination tool written by Joe Steffen [Ste85], searching for eight symbols (`cscope1`) in a 18MB software package, searching for four text strings (`cscope2`) in the same 18MB software package, and searching for four text strings (`cscope3`) in a 10MB software package;
- `dinero`: a cache simulator written by Mark Hill [HP90], running on the `cc` trace;
- `glimpse`: a text information retrieval system [MW94], searching for four keywords in a 40MB snapshot of news articles;
- `postgres-join`: the `postgres` relational database system (version 4.0.1) developed at the University of California at Berkeley [The93], performing a join between an indexed 32MB relation and a non-indexed 3.2MB relation (the relations are those used in the Wisconsin Benchmark [Gra91]). Since the result relation is small, most of the file accesses are reads.

The Sprite traces consist of five sets, recording about 40 clients' file activities over a period of 48 hours (traces 1, 2 and 3) or 24 hours (traces 4 and 5). I simulated client

		dinero	cscope1	cscope3	cscope2	glimpse	pjoin
Reference Time	Average	11.8	3.13	2.89	4.13	1.17	10.4
	Stddev	0.12	1.07	1.54	1.56	3.83	1.27
Fetch Time	Average	8.7	8.0	11.8	14.3	7.6	17.9
	Stddev	0.38	0.40	0.69	0.53	0.95	0.26

Table 1: Average and standard deviation of the reference times and disk fetch times in each Ultrix trace. Times are in milliseconds.

prefetching because it is important in a system with a slow network like Ethernet. For each client, I extracted its file activities to Sprite's main file server.

In the Sprite traces, I eliminated those that are dominated by cold-start misses, keeping only those for which cold-start misses account for fewer than half of the misses of the LRU algorithm on a 7MB cache (7MB is the average file cache size for the Sprite clients). This resulted in 13 client traces, which are denoted as `T[1-13]`: client 29, 33, 53, and 81 of trace 1, client 60, 62, 75, 82 of trace 2, client 9, 77, 56, 18 of trace 4, and client 18 of trace 5.

The client traces are not read-dominated: about 10% to 30% of the accesses are writes. In the simulations, however, I treat all write accesses as if they were read accesses.

3.4.3 Simulation Models

The simulations used two models of file accesses: *simplified* and *realistic*. The simplified model corresponds to the theoretical model discussed in previous sections: it assumes that the "reference time" (i.e. the amount of CPU time spent by the program between two consecutive file accesses) is uniform and defined as one time unit, and that fetching data from disk takes F time units. The realistic model uses the actual time interval between references from the trace, and uses the average (measured) disk access time for the trace as the fetch time. Only the Ultrix traces may be simulated under the realistic model, as the Sprite traces lack the necessary timing data. The average reference time and disk access time for each Ultrix trace is

shown in Table 1.

There are still a number of differences between the realistic model and the actual file systems. Disk access times are not uniform in real systems. The model assumes that disk blocks are allocated in 8KB blocks, whereas real file systems allocate smaller blocks (e.g. 1KB blocks) for small files. Hence, the simulations tend to inflate file data set size for applications that use a lot of small files. The simulations also ignore accesses to meta-data (including directories and inodes).

Despite the differences, the simulator is calibrated under the realistic model against real systems. I compared the simulated elapsed time and number of disk fetches of *LRU-OBL*, which is the algorithm used in Ultrix 4.3, to the measured elapsed time and number of disk fetches for the Ultrix traces. Table 2 shows the comparison for the Ultrix traces with two cache sizes: 6.4MB and 12MB. The simulated results are usually within 10-15% of the measured ones. The only exception is `cscope3`; the difference is due to the fact that `cscope3` accesses a lot of small files, and hence the assumption that files are allocated in 8KB blocks artificially inflates the size of the file data set. The results show that, in general, the simulation predicts real performance reasonably well.

In order to understand the accuracy of the simplified model, I also compared the simulation results under the simplified and the realistic models for Ultrix traces. I took the average reference time of each trace as one time unit, and set F to be the ratio of the average disk access time to the average reference time (remember that F is the fetch cost in terms of number of time units). The simulation results under this simplified model match those of corresponding simulations under the realistic model very well[4] — the differences are within 2%. Thus, the simplified model predicts real system performance reasonably.

[4]For the simulations under the realistic model, I scaled the disk fetch time to be an integer multiple of the average reference time, corresponding to the constraint that F be an integer.

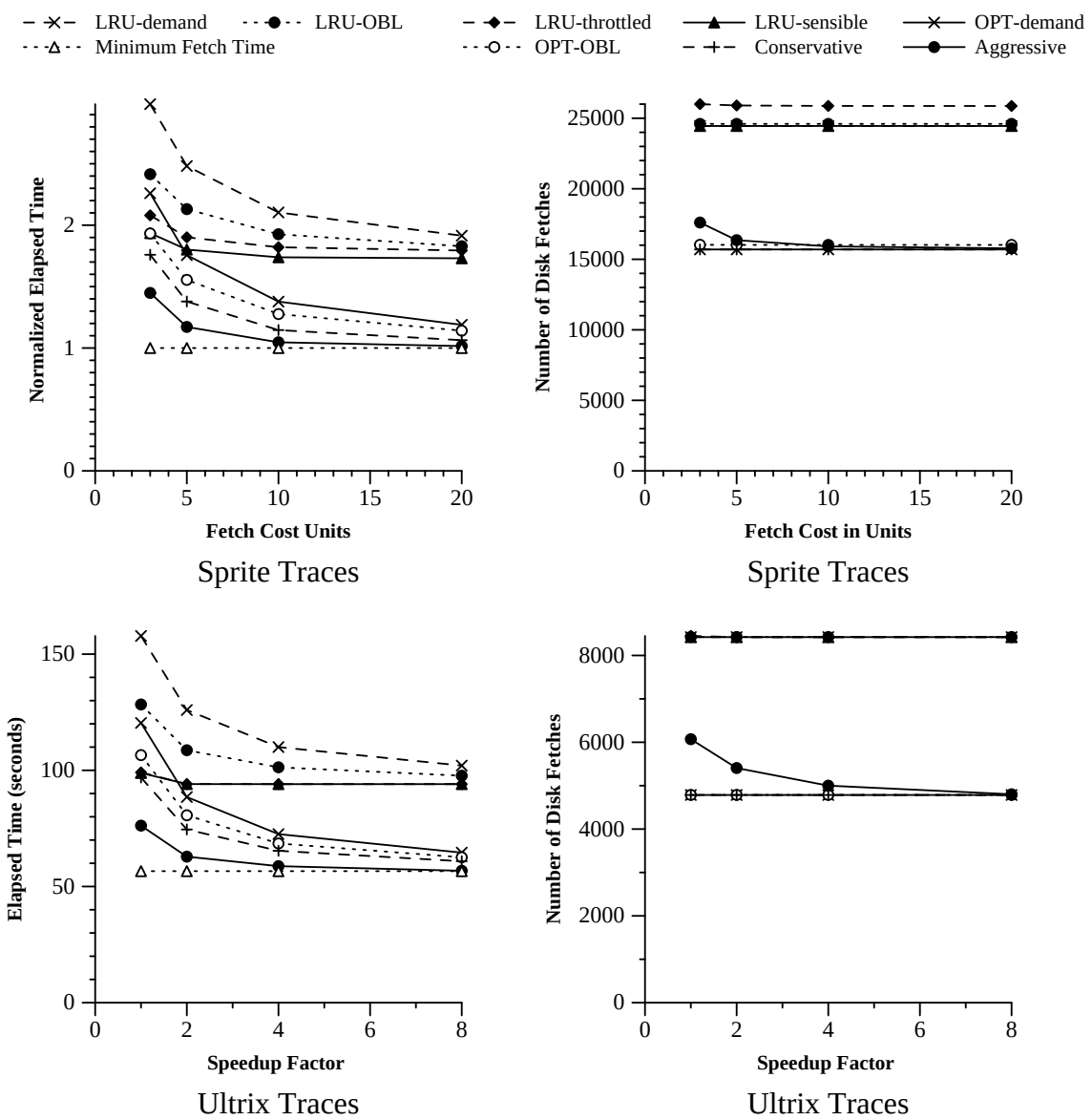

Figure 18: Performance of prefetching algorithms for the Sprite and Ultrix traces.

			dinero	cscope1	cscope2	cscope3	glimpse	pjoin
6.4MB	Elapsed	Simulated	105	70	92	194	101	207
6.4MB	Time	Measured	117	62	96	191	126	225
6.4MB	Disk	Simulated	8807	8518	6353	10769	9480	6624
6.4MB	Fetches	Measured	8888	8634	6576	11785	10435	6706
12MB	Elapsed	Simulated	105	31	92	194	101	183
12MB	Time	Measured	99	28	57	188	113	202
12MB	Disk	Simulated	988	1071	6353	10765	9466	5305
12MB	Fetches	Measured	997	1141	2815	11717	9720	5437

Table 2: Comparison of simulated and measured elapsed time and number of disk fetches for the Ultrix traces with two cache sizes: 6.4MB and 12MB.

3.4.4 Simulation Results

I simulated all eight algorithms for both Ultrix traces and Sprite traces. The simulation of the Ultrix traces used the *realistic* model, and the simulation of the Sprite traces used the *simplified* model. The simulations were run varying the cost of disk fetch time relative to the reference time.

For the Ultrix traces, the algorithms were simulated with the CPU time reduced by factors of 1, 2, 4, and 8 (corresponding to faster CPUs), and the elapsed time was reported in seconds.

For the Sprite traces, the algorithms were simulated with F set to 3, 5, 10 and 20, representing the range of fetch costs in today's systems (see Table 1). I use the minimum total fetch time M, which is the theoretically minimal number of disk fetches times the fetch time, as a baseline for comparison. For each F, I report the normalized elapsed time (i.e. the ratio between the elapsed time and M).

Figure 18 shows the averaged results over each set of traces (Ultrix and Sprite). The top two graphs are averaged elapsed time and number of disk fetches for the Sprite traces, with cache size 7MB; the bottom ones are for the Ultrix traces, with cache size 6.4MB. The elapsed time graphs (on the left) show the performance of the eight algorithms, as well as the lower bound on the elapsed time given by the minimum fetch time (i.e. the theoretically minimal number of disk fetches times the

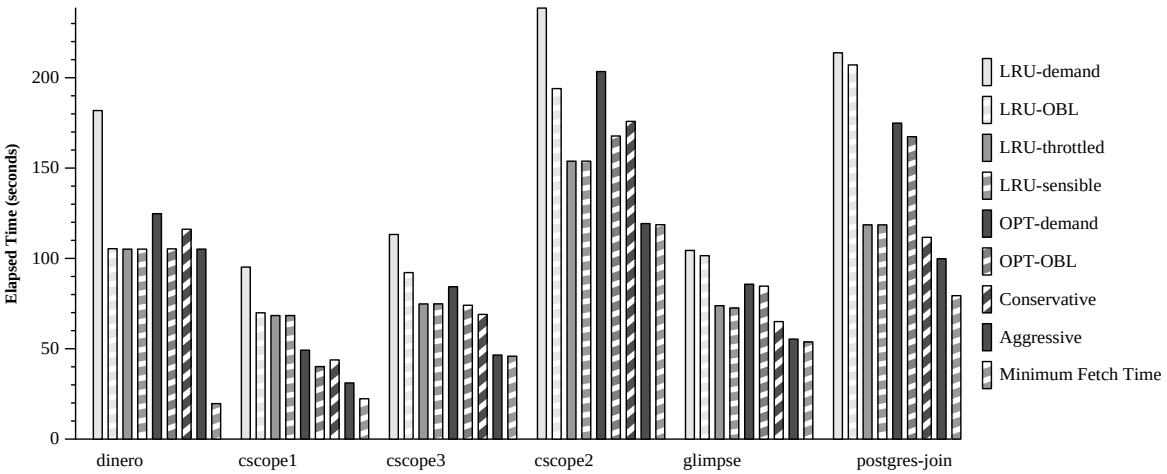

Figure 19: Individual Ultrix traces: elapsed times of the eight algorithms.

disk fetch time). In the graphs for numbers of disk fetches (on the right), the top group of lines contains *LRU-demand*, *LRU-OBL*, *LRU-throttled*, and *LRU-sensible*, and the bottom group of lines contains *aggressive*, *OPT-OBL*, *OPT-demand*, and *conservative*.

The results from individual traces differ only slightly from the averages. I select two graphs to show here. Figure 19 shows the elapsed times in seconds for individual Ultrix traces, under the 6.4MB cache, with the original CPU times (no speedup factor). Figure 20 shows the number of disk fetches for each Sprite trace, under the 7MB cache, with $F = 5$.

The major conclusion from these results is that *aggressive* performs the best among all these algorithms, confirming the theoretical results. In fact, *aggressive* is quite close to optimal for these traces. Theorem 3 says that the running time of *aggressive* is at most $1 + F/p$ times that of optimal on any reference stream, where p is the average phase length. Measuring the phase length of the traces, I find that as F varies, *aggressive*'s elapsed time is at most 1.024 times that of optimal for the Ultrix traces and at most 1.02 times that of optimal for the Sprite traces.

Conservative does slightly worse than *aggressive* most of the time. Among all LRU-based algorithms, *LRU-sensible* performs the best, as expected. The reason for all LRU-based algorithms performing about the same number of disk fetches on Ultrix

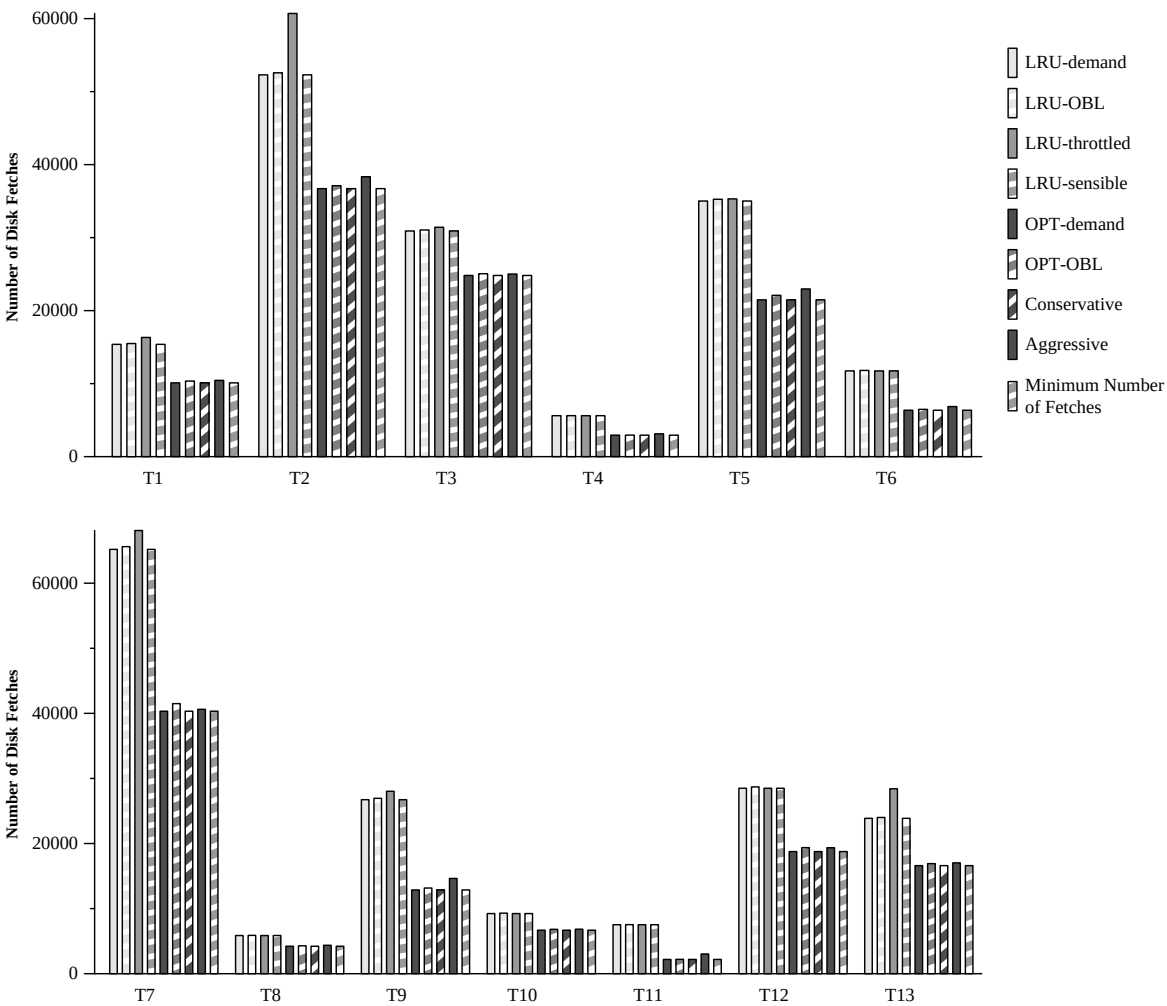

Figure 20: Individual Sprite traces: number of disk fetches performed by the eight algorithms.

traces is because these traces have such poor temporal locality that LRU replacement misses most of the time.

In addition, the following observations may be made:

- As the relative cost of disk accesses increases, replacement policies make a bigger difference than prefetching policies. This is because the systems become more disk fetch-bound as the relative fetch cost goes up, and the benefit from prefetching (overlapping computation time with fetch time) diminishes as the relative computation time gets smaller.

  Therefore, when the fetch cost is large, given any information on future file accesses, the first priority is to make replacement decisions as close to optimal as possible; making good prefetching decisions comes second. The *conservative* algorithm performs well in these cases since it performs the minimum number of disk fetches possible.

  Of course, this conclusion does not apply to all situations. Introducing parallel disks, or increasing the block size, would probably increase the benefit of prefetching. The results do imply, though, that prefetching alone is of limited value.

- The graphs for numbers of disk fetches confirm the theoretical results. As predicted, *LRU-sensible* and *LRU-demand* always perform the same number of disk fetches, and *conservative* and *OPT-demand* always perform the same number of disk fetches. In addition, the number of disk fetches performed by *aggressive* is always between that of *OPT-demand* and that of *LRU-demand*.

  The number of disk fetches made by *aggressive* decreases as the fetch cost increases. This is due to the self-adjusting property of *aggressive*. As the fetch cost gets larger, *aggressive* cannot prefetch fast enough, and thus it finds itself prefetching only a few references ahead. Under these conditions *aggressive* makes replacement choices close to those of the optimal replacement algorithm.

- The graphs show the weakness of approaches that do not follow the "do-no-harm" rule. For the Sprite traces, *LRU-throttled* makes noticeably more disk

fetches than LRU demand paging. This is because *LRU-throttled* ignores the "do-no-harm" rule, and "throttling" does not always eliminate the "thrashing" problem. *LRU-sensible*, on the other hand, always has fewer disk fetches and shorter elapsed time than *LRU-throttled*.

Essentially, these observations state that all four rules about optimal prefetching algorithms (Section 3.2.2), especially "optimal-replacement" and "do-no-harm," should be followed as much as possible by prefetching approaches used in practice.

3.5 Aggressive with Limited Lookahead

Both theoretical analysis (measuring worst case performance) and trace-driven simulations (measuring average case performance) show that *aggressive* performs close to optimal. Though both algorithms perform well, *aggressive* is generally better than *conservative*. Not only the approximation ratio of *aggressive*, $(1 + F/k)$, is usually much less than that of *conservative*, but also the simulations confirm that most of time *aggressive* outperforms *conservative*.

Although *aggressive* is presented as an off-line algorithm, it can be implemented *on-line* in application-controlled file caching and prefetching. Basically, *aggressive* needs two pieces of information: lookahead to the next cache miss, and the optimal replacement algorithm. In application-controlled file caching and prefetching, application-specific replacement policies can often serve as approximations to optimal replacement algorithms, and applications' prediction of future file access can serve as the lookahead.

To be more specific, here is how *aggressive* integrates prefetching and caching in practice. When a process would like to use prefetching, it gives the kernel a list of predicted file accesses. The kernel keeps track of whether the list is correct and where the process is in this list, i.e. the *cursor*. Every time a file access is made or a disk I/O finishes, the kernel checks which block is the next missing one from the list, and whether there is any block that is currently in cache and will be accessed after the missing block. If there is, a prefetch is issued for the missing block, and a replacement

is found among all cached blocks that will be referenced after the missing block.

To choose the replacement, the kernel uses both the application's control over file cache replacement and the list of predicted file accesses. The goal is to find the block referenced furthest in the future. If the list is long enough, it can uniquely identify the block. If it is not long enough, application controlled file cache replacement is used to help choosing the block. The cached blocks which no longer appear in the list are identified, and the replacement is chosen among them by applications using information on their access patterns.

The reason why it is necessary to use both application-controlled cache replacement and the list of predicted accesses in choosing the replacement is as follows. Applications often only predict their future accesses to a limited degree, for example, they may only know the access sequence for the next phase of the computation. The prediction do not always tell which block is the best replacement; it can tell which blocks are accessed next, but among all blocks which are not accessed soon, it does not know which one is the best to replace. Application-controlled cache replacement provides more information on that; for example, it may know which blocks, among those are not accessed soon, are generally accessed more often than others, and hence which are good replacements in the long run.

Thus, the two pieces of information complement each other. The application-specific replacement information captures long term patterns, while the prefetching prediction describes the short term accesses in detail. The two pieces of information can overlap each other as well. The algorithm described here lets the prefetching predictions override the application-specific replacement information, by using application-controlled cache replacement only among those blocks that do not appear in the predicted accesses.

3.6 Summary

In application-controlled file caching and prefetching, the key problem in a process's local cache management is the integration of cache replacement and prefetching. This

chapter studies this problem first from a theoretical perspective, proposing algorithms that are close to optimal in theory, then reports simulation results of the algorithms with real file access traces to confirm and validate the theoretical findings. The conclusion is that the *aggressive* algorithm performs close to optimal.

The theoretical investigation starts by defining a model for caching and prefetching in file systems. Though simple, the model is later shown by simulations to predict the performance of the algorithms on real systems quite well. The model enables the discovery of four rules for optimal algorithms and a dynamic-programming optimal algorithm. Since it is difficult to find a polynomial-time optimal algorithm (the dynamic-programming one is exponential-time), I focus on simple approximation algorithms. The four rules lead to two such algorithms, *conservative* and *aggressive*. The algorithms represent the two opposite ends of the spectrum in trading off the number of disk fetches and the amount of overlapping between disk fetches and CPU computation. *Conservative* performs the minimum number of disk fetches possible but tries to overlap the disk fetch and CPU computation as much as possible under the constraint. *Aggressive* always prefetches the next block not in cache at the earliest possible moment allowed by the four rules, replacing the block whose next request is furthest in the future. The number of disk fetches performed by *aggressive* is never more than that of the LRU replacement policy.

Analysis shows that the the elapsed time of *conservative* is at most twice of that of optimal. What is more important, the elapsed time of *aggressive* is at most $1 + F/k$ times that of optimal, where F is the cost of fetching a block from disk (in number of time units), and k is the cache size. In file systems, where F/k is usually less than 0.1, the theoretical analysis predicts that *aggressive* approximates the optimal algorithm closely.

To evaluate the algorithms and compare them with existing approaches, I built a trace-driven simulator for *aggressive* and *conservative*, as well as six other algorithms modeling traditional file prefetching approaches and strategies used in recent projects on file prefetching. Using two collections of file access traces, the simulations show that *aggressive* performs the best among all these algorithms and is very close to

optimal (its elapsed time is at most 1.03 times that of the optimal algorithm for these traces). Comparing *aggressive* and *conservative*, the simulations show that on average *aggressive* performs better.

The simulation results also demonstrate that the four rules on optimal prefetching algorithms, derived from the theoretical model, are quite important in practice. Algorithms which follow the "optimal-replacement" rule significantly outperform those which do not, especially as the fetch cost increases. Algorithms that do not follow the "do-no-harm" rule experience thrashing, in which case prefetching hurts, rather than helps, the performance. The results show that although the four rules are properties of the optimal algorithms, practical algorithms should follow them as well for good performance.

Several questions deserve further study. It is not known whether there is a polynomial time algorithm for computing the optimal off-line prefetching and caching strategy, or whether the problem is NP-complete. The studies presented here assume full knowledge of future file accesses; it is not clear how the algorithms perform when only inaccurate or probabilistic information is known, or what are the best algorithms in those cases. The theoretical model does not distinguish write accesses from read accesses. In reality, writes are different from reads in that dirty blocks need to write back to disk when replaced, and full-block writes may not need to bring the block into the cache. Treating write accesses differently from reads may produce more accurate results.

There are a number of limitations in the studies in this chapter. I have discussed most of the algorithms as off-line, i.e. they know the complete access sequence beforehand. The simulations only consider the single-process case (assuming all accesses come from one process) and ignore the often unpredictable context switches. The model used in the studies does not consider write accesses and represents only the single disk or single storage server case.

Some of the limitations are addressed in this dissertation. In Section 3.5 I have

described how *aggressive* can be implemented in a semi-on-line fashion, with only limited look-ahead into future file accesses and application-specific replacement policies. The next chapter will describe a detailed implementation of *aggressive* and evaluate its performance in multi-process environments. Studies of integrated prefetching algorithms in the parallel disk case are unfortunately beyond the scope of this thesis.

Chapter 4

ACFS: A Prototype

There are three reasons for a prototype implementation of application-controlled file caching and prefetching. The previous chapters have predicted that the techniques can improve applications' file access performance significantly. However, the results are obtained through theoretical analysis and simulation; whether they are true in practice can only be answered by an implementation in a real file system. The prototype also provides an opportunity to understand the interactions among application-controlled file caching, prefetching and disk scheduling. Finally, the prototype provides a testbed for experimenting with global allocation policies in multi-process environments.

This chapter describes a prototype file system that incorporates application-controlled file caching, integrated prefetching and disk scheduling. The prototype is implemented by modifying an Ultrix 4.3 file system (a variant of the UNIX file system), and is called Application-Controlled File System, or ACFS. The chapter first presents the design and implementation of ACFS, then describes experiments with a suite of I/O intensive applications.

4.1 Design

File cache management in ACFS follows the two-level cache management principle introduced in Chapter 1. Briefly, two-level cache management splits the responsibility of managing the file cache between the kernel and user processes; the kernel allocates cache blocks to application processes, and each application process is responsible for the use of its cache blocks. In ACFS, the kernel uses the policy of Least-Recently-Used with Swapping and Placeholders (LRU-SP) (Chapter 2) to allocate cache blocks among processes. Applications use ACFS's interface for application-controlled caching and prefetching to control the replacement and prefetching of their file blocks, with each application using *aggressive* (Chapter 3) to integrate cache replacement, prefetching and disk scheduling.

4.1.1 Two-Level File Cache Management

In multi-process or multi-user environments, a file system that supports application control of file cache management needs to prevent any single process or user from monopolizing the file cache resource and negatively affecting the performance of the whole system. In other words, the kernel needs to retain control of the allocation of cache blocks to applications.

The goal of ACFS is to allow user processes to manage their file cache, and at the same time, to maintain dynamic allocation of cache blocks among processes, such that good user level control can improve the overall system performance over the traditional scheme. To achieve this goal, ACFS uses the two-level file cache management scheme.

The interactions between kernel and user processes are the following. On a file cache miss or a prefetch for a file block, some block needs to be replaced from the cache. The kernel decides which process should replace a block (based on its allocation policy) and notifies the process, suggesting a replacement block. The process either takes the kernel's suggestion, if it does not want to control its cache replacement, or decides which block it wants to replace. The kernel then replaces whichever block the

process chooses (as long as it belongs to the process).

The user process can control not only the replacement of its cache blocks, but also the prefetching of its file blocks. Chapter 3 has shown that prefetching decisions should be integrated with replacement decisions using the *aggressive* policy. Due to limitations from implementation, in ACFS the kernel performs the integrated prefetching on behalf of the user process. In other words, the user process gives the kernel a list of its predicted file references, and tells the kernel to prefetch file blocks according to the list using the *aggressive* policy. Thus, the user process's control of prefetching is indirect; it decides when the prefetching of a list of blocks starts by specifying the list, but once it specifies the list, it leaves to the kernel the prefetching of necessary blocks.

The reason for the "indirect" control of prefetching is the following. Since the kernel allocates cache blocks dynamically, the user process needs to monitor the content of its portion of the cache constantly in order to make prefetching decisions following the *aggressive* policy. This not only increases the complexity of applications or user-level libraries, but also may incur high overhead due to the synchronization between applications and the kernel. On the other hand, Chapter 3 has shown through theoretical analysis and simulations that *aggressive* is close to optimal. Therefore, *aggressive* can be implemented in the kernel, which, knowing the content of the whole file cache at every moment, can identify missing blocks in the process's predicted access sequence and prefetch them at appropriate times. The performance of the integrated prefetching implemented in the kernel would be no worse than that implemented at the user level.

To support cache management of shared files among cooperating processes, ACFS allows a parent process to be the cache manager of its child processes. Usually a process owns the files that it accesses (i.e. opens and reads or writes) and manages those files' caching and prefetching. When multiple processes collaborate on a job and share files, however, they can let their parent process manage the files they access, in which case the parent process own all files accessed by the child process. This allows the parent process to take advantage of the information on the sharing among the

collaborating process to improve their file I/O performance. If a file belongs to more than one manager processes, the kernel takes over the control on it and applies the default policy.

Finally, if a process is unable or unwilling to manage its own cache blocks and does not have a parent process as its manager, it lets the kernel manage its cache using the default policy. In ACFS, the default policy is Least-Recently-Used (LRU) replacement with sequential one-block lookahead prefetching. The LRU replacement policy always replaces the block that is least recently used among all blocks of the process. The sequential one-block lookahead policy prefetches block $N+2$ of a file, if the block is not in cache, when the last two references to the file are to block N and block $N+1$.

The Global Allocation Policy

The choice of the kernel's global allocation policy is the key issue in two-level file cache management. Chapter 2 lists the criteria for global allocation policies. Essentially, the criteria state that under the global allocation policy, applications which use their control appropriately should not only benefit themselves, but should also maintain the performance of the whole system no worse than under traditional file systems. Briefly, the criteria are:

- *An oblivious process (one that lets the kernel manage cache blocks for it) does not suffer more misses than it would under global LRU (the cache management policy used by traditional file systems).*

- *A foolish process (one that uses a policy worse than the kernel's default policy) never causes another process to suffer more misses than it would suffer under global LRU.*

- *A smart process (one that uses a policy better than the default) never causes any process (including the smart process) to suffer more misses than it would suffer under global LRU.*

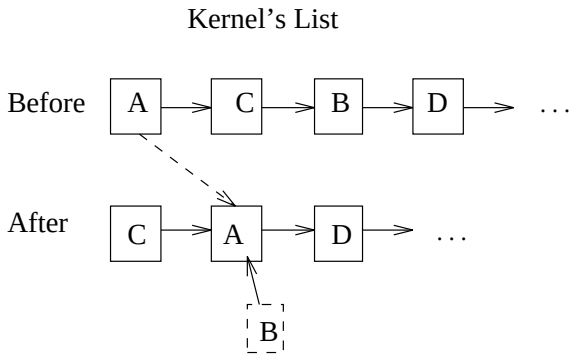

Figure 21: How LRU-SP keeps the "LRU" list.

Chapter 2 presented an allocation policy that satisfies these criteria in the context of application-controlled file caching without prefetching. The policy, called LRU with Swapping and Placeholders (LRU-SP), operates by maintaining an "LRU" list of all in-cache blocks. The list is in the order of the blocks' last references, except for two modifications (explained below). Those blocks which are prefetched but not yet referenced are put in the most-recently-used end of the list.

When a process *misses in the cache* or *issues a prefetch*, some block must be evicted from the cache in order to make room for the newcomer. LRU-SP looks at the LRU list to find out which process "owns" the block that is at the least-recently-used end of the list. That process is asked to give up one of its blocks; it is up to the process to decide which of its blocks to evict. (Blocks belonging to dead processes are considered to have no owner; they are simply replaced when they reach the least-recently-used end of the list, as they would have been under the global LRU replacement policy.)

LRU-SP makes two modifications to the LRU list, swapping and placeholders, as explained in Chapter 2. Swapping means that when a process chooses to give up a block which is not at the least-recently-used end of the list, the block at the least-recently-used end is moved into the evicted block's position in the LRU list. Placeholders are data structures which remember when a process chooses to overrule the kernel; if this choice turns out to be foolish (i.e. the block chosen by the user process is accessed before that chosen by the kernel), the placeholder can be used

to keep the foolish process from obtaining more cache blocks as a result of its bad choice. Figure 21 shows an example. Block A is at the least-recently-used end of the list. The process which owns A decides to replace block B. The figure shows the list before and after the replacement decision.

To prevent processes from abusing the facility for application-controlled prefetching, the kernel takes the following precaution. Prefetched blocks are generally added to the most-recently-used end of the LRU list, because they are predicted to be used soon. However, it is possible for an application process to prefetch blocks but subsequently not use them. Thus the process can deliberately do harm to the whole system by tying up the disk with its prefetches. To prevent this from happening, the kernel must always match the specified reference list against the actual file accesses and, whenever it finds blocks that are prefetched but not accessed as predicted, quickly flush them out of the cache.

LRU-SP does not provide the same theoretical guarantee in the presence of prefetching as it does with application-controlled file caching alone. Chapter 2 showed that under the assumption of fixed interleaving of references from multiple processes, LRU-SP is proven to satisfy all three criteria for global allocation policies. The proof no longer holds when there are processes which uses application-controlled prefetching.

On the other hand, the principle of the criteria is still upheld. The above check by the kernel makes sure that applications cannot monopolize resources by providing false information in prefetching. And *aggressive* makes sure that applications providing correct information can actually benefit from prefetching.

On important fact that has been ignored by the theoretical studies so far in this thesis is that disks are not uniform-access devices. In fact, appropriate scheduling of disk accesses can improve the throughput and the average latency significantly [SCO90, JW91]. And indeed, application-controlled file prefetching brings new opportunities for disk scheduling.

4.1.2 Incorporating Disk Scheduling

Another technique ACFS uses to improve file system performance is disk scheduling. Because of the physical attributes of disks, careful scheduling of disk accesses can provide significant improvement in performance [SCO90, JW91]. Effective scheduling requires a large number of requests in queue for service. In traditional file systems, the sources of multiple requests are mostly asynchronous disk I/Os (especially delayed disk writes) and multiple concurrent processes.

Application-controlled prefetching is a new source of multiple requests for disk scheduling. By giving a list of predicted file accesses, the application process informs the kernel of all the blocks that are currently not in cache and thus need to be prefetched. The kernel can choose to issue more than one prefetch for these blocks at a time, as long as the *aggressive* policy (especially the "do-no-harm" rule) is followed. The prefetch requests can then be scheduled to reduce the average access latency.

However, scheduling prefetch requests is different from scheduling asynchronous I/O requests or requests from multiple processes. Prefetching requests are issued in the order that the data will be consumed by the CPU. By reordering the requests, disk scheduling can cancel some of the overlapping between I/O and CPU computation. For example, suppose the application computes, reads block X, computes again, then reads block Y. If there's no scheduling, the computation before block X will overlap with the fetch for block X, and the computation between X and Y will overlap with the fetch for block Y. But if the scheduling algorithm reads Y before X, then the computation before X will have to overlap with the fetches for both X and Y, which might result in the CPU being stalled longer for I/O. Hence, scheduling prefetches is a nontrivial problem.

ACFS employs a simple heuristic: limited batch scheduling. The prefetcher for each application process (in the kernel) waits until it has B disk requests to issue. These requests are then sorted by order of increasing disk block number, and the fetches are performed in sorted order.

I chose the value of B empirically. In general, if B is too small, disk scheduling will be ineffective because there is not much latitude to reorder fetches. However, if

B is too large, then aggressive reordering of fetches will lead to some fetches being moved much later in the request sequence, thereby losing the opportunity to overlap CPU computation with disk fetches. For example, if the first missing block that the program will access has the $B/2$-th disk-block-number in the batch, then the CPU will have to wait for $B/2$ disk accesses to complete until the first missing block arrives in the cache. The proper choice of B balances these two factors. Through the implementation of ACFS I found by experiment that $B = 16$ works well (Section 4.4.2).

The batches of prefetch requests from all processes should be sent to the disk driver and scheduled together with other synchronous and asynchronous requests in the disk driver's queue. The disk driver can then use its best scheduling algorithm to maximize throughput. The implementation of ACFS, however, schedules each batch separately before issuing it to the disk driver, to avoid changing the SCSI disk driver of the Ultrix 4.3 kernel. The disk driver then simply serves the requests in the order that they are issued.

For each batch, ACFS's scheduling policy is disk-block-number ordering, as explained above. The requests are served in increasing order of their disk addresses. This policy is a crude approximation to the optimal fetching order. Computing the optimal pattern is a difficult problem even for idealized disks; for real disks, with track buffering, uncertainty in measuring head and rotational positions, and a complex mapping from disk addresses to physical sectors, it is virtually impossible. Nevertheless, studies show that fetching in order of disk block numbers tends to work well [WGP94].

To summarize, ACFS uses a strategy for file cache management integrating application-controlled caching, prefetching, and disk scheduling. It is a two-level strategy, in which the kernel allocates cache blocks to processes, and each process manages its own blocks. The kernel uses the LRU-SP policy to allocate blocks to processes, and each process uses the *aggressive* policy for managing its cache. Each process improves its access efficiency by submitting its prefetches in batches, with each batch sorted into order by disk block number.

4.1.3 Application Programming Interface

ACFS uses two sets of system calls as the interface for application-controlled file caching and prefetching. One allows applications to control file cache replacement and the other allows applications to specify predicted file accesses for prefetching.

Application Control of Cache Replacement

There are a variety of ways to implement the interaction between applications and kernel with respect to file cache replacement. Simple schemes perform well, but do not give applications sufficient flexibility in controlling cache replacement. A more general and flexible scheme, on the other hand, can incur high overhead. The main design challenge is to devise an interface that allows applications to exert the control they need, without introducing the overhead that would result from a totally general mechanism.

The simplest implementation is to provide a fixed set of replacement policies (the "menu") in the kernel and let user processes choose from them. Unfortunately, this is not flexible: every time a new policy is added, the kernel has to be changed.

The most general scheme is let the kernel make an upcall to the manager process every time a replacement decision is needed. This is implemented in some systems that support application control of physical memory management in virtual memory systems [MA90]. Although the scheme does provide full flexibility, it can incur significant overhead in current operating systems. Experiments reported overhead as high as 10% of the application's running time [MA90, SP91].

There are a number of other hybrid approaches, trading generality and flexibility for performance. One is to let applications specify priorities for cache blocks, so that lower priority blocks are replaced first. This approach provides roughly the same generality and flexibility as the upcall approach, but does not incur overhead as high as the upcall scheme. However, to implement some common policies (e.g. most-recently-used), applications have to change priorities frequently, which still results in significant overhead.

The strategy for ACFS was to first consider common file access patterns reported

in the literature (e.g., [OCH*85, BHK*91, CD85]), as well as patterns that are observed in many applications. I then chose a simple interface that was sufficient to compose caching strategies to support these access patterns.

Priorities and Policies

The basic idea in ACFS's interface is to allow applications to assign priorities to their own files or file blocks, and for each priority level, specify file cache replacement policies. Within a process, all the files or blocks with the same priority are treated as a single pool. The kernel always replaces blocks from the pool with the lowest priority first; among all file blocks with the same priority, the kernel chooses the victim block based on the replacement policy for that priority.

Priorities and policies applies only to all the blocks of a single process (or in fact, a manager process). Inter-process (or, inter-manager-process) allocation decisions are made using LRU-SP, as explained in the previous section.

Priorities and policies provides both full flexibility for application control of cache replacement and high performance for common application policies. The priority of a particular file or file blocks is not static; it can be changed dynamically during the application's execution. Thus, the interface can be as flexible as the pure priority scheme described above. That is, if wanted, the priorities can be sufficiently fine-grained that each priority level has only one block and the kernel merely uses priorities to choose replacement without looking at policies. The interface can also be as fast as a fixed menu scheme for some common policies, when the application simply specifies that all its blocks are of one priority, and chooses the policy for the priority.

In general, the appropriate use of this interface allows applications to implement a large variety of replacement policies at low overhead. Since within a process, files (file blocks) with the same priority belong to the same caching "pool" with the same replacement policy, application or library writers can use a combination of priorities and policies to deal with various file access patterns. They can support access patterns among files by applying different policies for different pools of files, or changing priorities to tell the kernel to replace some files or blocks before others.

The calls in the interface are:

- `set_priority(file, prio)` sets the long-term cache priority of a file;
- `get_priority(file)` gets the long-term cache priority of a file;
- `set_policy(prio, policy)` sets the file cache replacement policy of a priority level;
- `get_policy(prio)` gets the file cache replacement policy of a priority level;
- `set_temppri(file,startBlock,endBlock,prio)` temporarily changes the priority of blocks between `startBlock` and `endBlock` of the `file` to priority `prio`.

The current implementation of ACFS offers only two replacement policies: Least-Recently-Used (LRU) and Most-Recently-Used (MRU). The default priority for any block is priority 0, and the default policy for any priority level is LRU.

In the current implementation of ACFS, the temporary priority change specified by `set_temppri` has the following semantics. It affects only those blocks that are presently in the cache (since the priorities apply to replacement). The temporary change of a block's priority only lasts until the next time the block is either referenced or replaced; when the temporary priority ends, the block reverts to its long-term priority as set by `set_priority`. Thus, applications can conveniently use temporary priorities to flush blocks out of the cache. By setting the temporary priority of a block to be the negative of the distance until the next access to the block, the application can implement the optimal replacement policy when the list of future file accesses is known.

The priorities and policies interface can support a variety of common file access patterns. Below I list some examples. Section 4.3 will describe how real applications use the interface to control file caching.

- Sequential: applications often read a file sequentially from beginning to end. If the file is accessed sequentially repeatedly, it can be assigned a low priority and

MRU replacement policy. If the file is accessed only once, it can be assigned priority -1 to flush its block from cache quickly.

- Cyclic access: some applications repeat the same sequence of accesses several times. These applications can assign a low priority and MRU replacement policy to the relevant file or files.

- Access to many small files: some applications access many small files. Since the interface treats all files of the same priority in a process as being in a single "pool" when making replacement decisions, these applications can control the caching of the blocks. (An interface that do not support inter-file access patterns would be useless for tiny files.)

- Hot and cold files: some applications access particular files unusually often, or with unusually good temporal locality. Such files can be assigned high priority.

- Future access prediction: some applications can predict that certain blocks are especially likely to be accessed in the near future. These blocks can be given a temporary increase in priority, to keep them in the cache longer.

- Done-with blocks: an application may know that it is finished with a block, and will not access it again for a long time. For example, this can happen in temporary files, which are often written once and then read once. When such a file is being read, the priority of a block can be decreased temporarily when the file pointer moves past the end of the block. This will flush the block quickly from the cache.

Thus, the interface is quite flexible and support access patterns both within a file and among a group of files.

Application Specification of Prefetching

By comparison, the interface for application-controlled prefetching is more straightforward. Since prefetching is implemented in the kernel, application processes only

need to give the kernel information about future accesses. ACFS provides an interface that allows applications to specify their prediction of future file accesses.

An application's predicted file accesses are specified by either a list of files or a list of blocks. A list of files allows users to define an access sequence between files, where each file is read from beginning to end. The alternative specification, a list of blocks, allows applications to specify the accesses at the block level. ACFS provides the following system calls:

- `FileListStart(size)` and `FileListEnd()` specify a sequence of files that are predicted to be accessed in the future, where `size` is the number of files to be specified. The `FileListStart` call marks the beginning of the list and `FileListEnd` terminates a list. The kernel then builds an ordered list of file accesses based on the sequence of file `open` calls between the two markers.

  An application can use this interface by inserting a piece of code at the beginning of its execution, which calls `FileListStart`, then opens and closes a set of files in sequence, and finally calls `FileListEnd`. This tells the kernel which files the application will access. Use of this interface requires minimal modification of applications.

- `BlockList(list, size)` specifies a list of file blocks that will be accessed in the future. `List` is a list of file chunks. A file chunk is specified by a file descriptor, a starting offset and an ending offset. The kernel then translates each file chunk into standard sized file blocks and builds an access list where the file blocks of each chunk are accessed in order.

The interface is deliberately simple so that not only applications can use it, but also compilers and libraries can use it to communicate to the kernel while providing automated prediction or better interfaces to applications.

There are other alternatives for the application programming interface. As described above, ACFS uses one set of system calls for application-controlled cache replacement, and another for prefetching. A more unified interface may be used, for

example, an interface that combines priority information with the list of predicted accesses. I have chosen ACFS's interface because some applications only have information about their access patterns and cannot predict their future accesses. A separate set of system calls for application-controlled cache replacement allows such applications to take advantage of good replacement policies, without having to give a list of predicted accesses. Furthermore, ACFS combines the information from the two sets of system calls to make good replacement decisions, as described in Section 3.5 and in the next section.

4.2 Implementation

I have implemented ACFS, a prototype file system that supports application-controlled file caching and integrated prefetching. The implementation is realized by breaking the existing buffer cache module of the Ultrix 4.3 file system into two modules: a Buffer Cache Module (BUF) and an Application Control Module (ACM), and adding a Prefetch Control Module (PCM) and a Batch Scheduling Module (BSM). BUF manages the allocation of blocks to processes, and ACM implements the replacement policy on behalf of each application. PCM manages prefetching and sends disk I/O requests to BSM, which batches the requests, sorts them and dispatches them to the disk driver. The structure is shown in Figure 22.

4.2.1 Application-Controlled Caching

ACFS implements the priorities and policies interface by multiplexing the calls through a single new `fbehavior` system call, in the same way that the Unix `ioctl` system call multiplexes several operations.

In the kernel, the buffer cache module is broken into two modules. The buffer cache module (BUF) handles cache management and bookkeeping and implements the allocation policy. The application control module (ACM) implements the interface calls and acts as a proxy for the user-level managers. Both of these modules sit below the VFS interface and communicate via procedure calls.

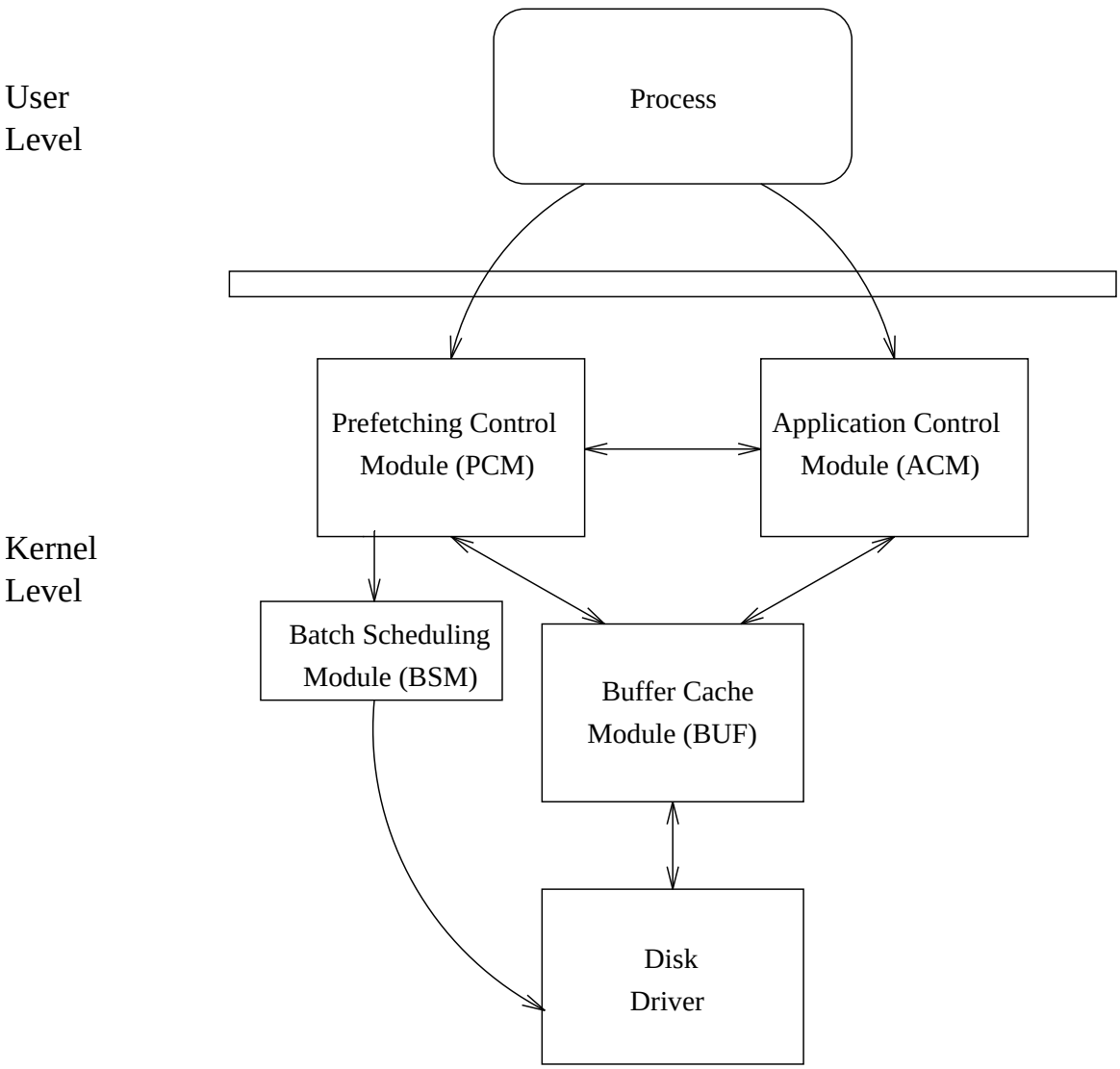

Figure 22: Structure of the implementation of ACFS.

When a block must be replaced, BUF picks a candidate block and asks ACM which block to replace, giving the candidate block as a suggestion. ACM then acts on behalf of the appropriate user-level manager to make the decision.

ACM implements the calls from user level in a straightforward way. It allocates a "manager" structure for any process that wants to control its own caching. Then for each priority level it allocates a header to keep the list of blocks in that level. It also allocates a file record if a file has a non-zero long-term priority. The implementation imposes a limit on kernel resources consumed by these data structures and fails the calls if the limit would be exceeded.

Every block, upon entering the cache, is linked into the appropriate list based on its file's long-term priority. The lists are always kept in LRU order, and LRU (MRU, respectively) chooses blocks from the least-recently-used end (most-recently-used end) of the list. Blocks may move among lists by `set_priority` or `set_temppri`. Blocks moving into a list are put at the end that causes them to be replaced later (the MRU end if the policy is LRU, or the LRU end if the policy is MRU). The opposite effect can be achieved by appropriate use of `set_temppri`.

BUF and ACM communicate using five procedure calls. These calls notify ACM about replacement decisions and mistakes, inform ACM about changes in cache state and accesses, and ask ACM for replacement decisions. The calls are:

- `new_block(block)` informs ACM that the block was loaded into cache buffer;
- `block_gone(block)` informs ACM that the block was removed from the cache;
- `block_accessed(block, offset, size)` informs ACM that the block was accessed;
- `replace_block(candidate, missing_block)`
  asks ACM which block to replace;
- `placeholder_used(block, placeholder)` informs ACM that a previous decision to replace the block was erroneous.

Changes are needed in the replacement procedure in BUF to implement LRU-SP. Instead of replacing the least-recently-used block, the procedure first checks if the missing block has a placeholder, then takes the least-recently-used block or the block pointed to by the placeholder (if there is one) as the candidate. BUF calls `replace_block` if the candidate block's caching is application-controlled, and finally BUF swaps block positions and builds a placeholder. The interface is well defined, and the procedures are called with no lock held.

The current implementation adds negligible overhead to file accesses. For processes that do not control their own caching, there is no added overhead on cache hits. On a cache miss, there are two sources of overhead: the kernel has to keep track of which file's data are in a cache block because the existing Ultrix file system does not remember it, and if the candidate block belongs to a process that manages its own cache, `replace_block` and `block_gone` have to be called. Since these procedures are cheap and are implemented in the kernel, both overheads are small; indeed they are negligible compared to the disk latency (Section 4.4.2).

For processes that control their own caching, the extra overhead on a cache miss is similar to the above case except that `new_block` is called as well. The overhead on a cache hit depends on the work done in `block_accessed`, which in the implementation only involves updating counters and moving a few blocks around in lists. These overheads are often smaller than the potential reduction in system CPU time caused by reduced number of misses.

Finally, the interface between BUF and ACM could be used if one wanted to implement two-level replacement using upcalls or dropped-in-kernel "user-level handlers" [BCE*94]. In particular user-level handlers could know which blocks are in cache by keeping track of `new_block` and `block_gone` calls.

4.2.2 Integrated Prefetching

ACFS uses the Prefetching Control Module (PCM) to implement the application prefetching interface, and the Batch Scheduling Module (BSM) to implement disk scheduling for prefetch requests. Together, PCM and BSM interface with the BUF,

ACM and the disk driver modules to optimize prefetching performance. Specifically, PCM keeps track of the file access streams specified by the applications and issues prefetches to BSM, in accordance with the *aggressive* algorithm. BSM batches prefetch requests, performs disk scheduling among them to optimize disk access performance, and issues them to the disk driver.

To accomplish these tasks, the PCM module builds and maintains a prefetch object for each application that is prefetching. The prefetch object consists of several data structures providing information necessary for PCM to implement prefetching under the *aggressive* policy. In general, in order to prefetch using the *aggressive* policy, the PCM must have five pieces of information:

- the predicted list of file accesses (the access stream),
- where the process is in the access stream,
- which is the next block that has to be fetched,
- how many blocks can be replaced for that fetch without violating the "do-no-harm" rule, and
- whether the previous prefetches are finished.

These are maintained in the prefetch object via the following data structures:

- `accessList`: the ordered list of file block numbers predicted to be accessed soon. Each entry in the `accessList` consists of the file's i-node and a pair of integers indicating the range of block numbers. Thus `accessList` takes roughly 136 bytes per entry. To prevent malicious applications from building a list that is too long, the size of `accessList` is limited (5000 entries).
- `cursor`: a pointer which keeps track of the most recently referenced block in the `accessList`.
- `nextHole`: the next missing block in the `accessList`.

- `freeBlocks`: the number of blocks in the cache whose next access is after `nextHole`.
- `pCount`: the number of outstanding prefetch requests.

In principle, keeping track of the `cursor` is straightforward: whenever the application accesses a file block, the PCM module matches the block against `accessList` to see whether the `cursor` should be advanced. In practice, `accessList` might not agree precisely with the true references. Therefore, the goal becomes to keep the `cursor` at the correct position with high probability in the face of possible inaccuracies in the access list. In the implementation, I used a very simple algorithm that looks ahead at some small constant number of references in `accessList` (two in the current implementation) and advances only if there is a match. The algorithm works fine in my experiments, because applications often predict less, not more, file accesses than they actually perform. However, if an application predicts two consecutive accesses but does not perform them later on, the cursor will get stuck and the kernel will ignore the list.

The PCM triggers a new batch of prefetching when the `cursor` is advanced, `freeBlocks` reaches at least 4 and the previous prefetches are finished (i.e. `pCount==0`). As long as fewer than B prefetch requests have been issued and `freeBlocks` is greater than 4, the PCM module asks the BUF module for an empty cache block, sends a prefetch request for the block pointed to by `nextHole` to the BSM module, and advances `nextHole`, changing `freeBlocks` if necessary.

The BSM module queues the requests issued by PCM. When the number of requests reaches B, or when the PCM module asks it to send the batch, BSM reorders the requests according to their disk addresses, and passes them on to the disk driver. The disk driver then services these asynchronous requests first-come-first-served (Section 4.1.2).

The motivation for issuing prefetches only when `freeBlocks` is at least 4 is as follows. In the multi-process case, the portion of the cache allocated to a single process varies with time depending on the relative frequency with which LRU-SP asks that process to relinquish blocks. Therefore, if the process' allocation shrinks

unexpectedly during a disk fetch, blocks more valuable than the prefetched block may have to be replaced, violating the "do-no-harm" rule. Fortunately, dramatic and rapid fluctuations in global allocation are unlikely. The implementation causes prefetches to be triggered only if there are at least 4 `freeBlocks` (i.e. there are at least four blocks less valuable than the prefetched block). This policy ensures that in the case that a process issues a prefetch and is asked to replace one of its own blocks, there is very likely to be a block that can be replaced without violating the "do-no-harm" rule.

The PCM module cooperates with the ACM module to make cache block replacement decisions. It passes the `accessList`, `cursor` and `nextHole` to the ACM module; the ACM module is responsible for choosing the best replacement according to the application's replacement policy and the given access stream, as described in Section 3.5. To be more specific, when a process misses in the cache, or issues a prefetch, the global allocation procedure of LRU-SP (in the BUF module) determines which process must relinquish a block. The ACM module then chooses the block by first looking at the `accessList` to find blocks that are referenced after the next missing block (`nextHole`), and then among all such blocks, using the application's primitives on caching policies to find the victim. This scheme is an approximation to the algorithm described in Section 3.5 in that it does not take full advantage of the information in `accessList`; however, it can perform almost as well as the algorithm in Section 3.5 as long as applications use good replacement policies. After it chooses the replacement block, the ACM notifies the PCM of its choice so that `freeBlocks` may be changed accordingly.

4.3 Applications

Beneficiaries of application controlled caching and prefetching include applications that use large file data sets and hence do not fit in cache, because LRU is often not the right policy for them, and applications that use mostly small files, because simple sequential prefetching does not benefit them. Examples include: database systems,

including text retrieval software; trace-driven simulations, as the trace files are often large; graphics and image-processing applications; and I/O-intensive UNIX utilities. I chose a set of applications that are representives of these categories. Most of the applications are publicly available and are commonly used as program development tools, teaching aids, or free database system packages.

Below I describe the applications and the replacement and prefetching strategies they use.

`cscope [cs1-3]`

The program `cscope` is an interactive C-source examination tool written by Joe Steffen [Ste85]. It builds a database of all source files, then uses the database to answer queries about the program. There are two kinds of queries for `cscope`: symbol-oriented ones, mostly questions about where C-symbols occur; and egrep-like text search, mostly asking where patterns of text occur. I used `cscope` on two software packages (in fact, two kernel sources), one about 18MB and one about 10MB, and did two sets of queries: searching for eight symbols, and searching for four text patterns. The three runs are:

- `cs1`: symbol search on the 18MB source;
- `cs2`: text search on the 18MB source;
- `cs3`: text search on the 10MB source.

Strategy: Symbol-oriented queries always read the database file "cscope.out" sequentially to search for records containing the requested symbols. Therefore for this kind of query, the right policy is to use MRU on "cscope.out:"

```
set_priority("cscope.out", 0);
set_policy(0, MRU);
```

For prefetching, the strategy is to prefetch "cscope.out" in sequential order.

Text searches involve reading all the source files in the same order on every query, so the right policy is MRU on all the source files. Since all source files have the default priority 0, the only necessary call is:

```
set_policy(0, MRU);
```

(When there is a mix of these queries, cscope can keep "cscope.out" in or discard it from cache when necessary by raising or lowering its priority.) The prefetching strategy is to prefetch all source files in their order of access.

`dinero [din]`

The program `dinero` is a cache simulator written by Mark Hill and used in Hennessy and Patterson's architecture textbook [HP90]. The distribution package for the course material includes the simulator and several program trace files. I chose the "cc" trace (about 8MB) from the distribution package, and ran a set of simulations, varying the cache line size from 32 to 128 bytes, and set associativity from 1 to 4.

Strategy: The program `dinero` reads the trace file sequentially on each simulation. Hence the right policy is MRU on trace file and the call is:

```
set_priority(trace, 0);
set_policy(0, MRU);
```

For prefetching, the application simply passes the trace file name to the kernel.

`glimpse [gli]`

The program `glimpse` is a text information retrieval system [MW94]. It builds approximate indexes for words to allow both relatively fast search and small index files. I took a snapshot of news articles in several comp.* newsgroups on May 22, 1994, about 40MB of text. Then I `glimpseindex`ed it, resulting in about 2MB of indexes. The searches are for lines containing these keywords: scheduling, scheduling and disk, cluster, rendering and volume, DTM.

Strategy: The index files and data files naturally lead to two priority levels because index files are always accessed first on every query, but data files are not. Hence `glimpse` gives the index files long-term priority 1, and the articles the long-term priority 0 (the default). Since index files are always accessed in the same order, and several groups of articles (called partitions [MW94]) are accessed in the same order, MRU is chosen for both levels. The calls are:

```
set_priority(".glimpse_index", 1);
set_priority(".glimpse_partitions", 1);
set_priority(".glimpse_filenames", 1);
set_priority(".glimpse_statistics", 1);
set_policy(1, MRU);
set_policy(0, MRU);
```

The program `glimpse` always starts its search with its indices. It first finds the partitions of files that match the pattern, then searches files in these partitions. It uses "agrep" to perform pattern matching on a list of files generated by indexing. As with `cscope`, only five lines of code were needed to pass the file list to the kernel for prefetching.

`link editor [ldk]`

I used the Ultrix link-editor to build the Ultrix 4.3 kernel from about 25 MB of object files, and labeled the application `ldk`.

Strategy: The link editor almost never accesses the same file data twice, but it does many small accesses, so the caching strategy is to free a block whenever its data have all been accessed, by calling:

```
set_temppri(file, blknum, blknum, -1);
```

Since the source of the link editor is not available, I implemented this policy in the kernel to simulate what the program would do and called it "access-once."

`postgres [psel1-2]`

The program `postgres` is a relational database system from the University of California at Berkeley [The93]. I used version 4.0.1 for Ultrix 4.3. The program `postgres` uses the file system for I/O operations, and since it only has a small internal buffer, relies heavily on file caching. I chose two indexed selection queries to show how it might benefit from application control of file caching and prefetching.

I used a scaled-up Wisconsin benchmark [Gra91]. The relations are a 200,000 tuple one, `twohundredk`, and a 20,000 tuple one, `twentyk`. There is a non-clustered index

on `unique1` in `twohundredk`, called `twohundredk_unique1`. The size of `twentyk` is roughly 3.2MB, `twohundredk` 32MB, and index `twohundredk_unique1` 5MB.

The queries are the following:

- `psel1`: query 1 in the Wisconsin Benchmark, a 1% selection on `unique1`, with value between 0 and 2,000.
- `psel2`: a 2% selection query, with `unique1` between 186,660 and 190,660.

The database system uses indices whenever possible to optimize query execution. A selection query, for example, has two possible access patterns: indexed or sequential. If there is an index, the selection query will use the index to find all tuples satisfying the queried condition. If there is no index file, the query will scan the relation tuple file sequentially to find the matching tuples.

Strategy: The execution of the selection queries uses the B+ tree index on `unique1`, and the access patterns are well understood. Both queries traverse the index blocks that cover the selection range to perform selections. They first search the index to find the index block containing the low end of the range (i.e. 1 for `psel1` and 186,660 for `psel2`), starting the search there and ending it at the block beyond the high end of the range. For each index block, the selection query checks all entries. If the data value of an entry falls in the selection range, it will read its indexed tuple.

Therefore, the prefetching algorithm for the selection queries is rather straightforward. After searching the indices, the list of tuple blocks that will be read is known. The application then simply passes this list to the kernel using the `BlockList` interface. For caching, it can be determined from this list which tuple blocks will be accessed more than once. These blocks are given higher priority than ordinary tuple blocks during the execution of the selection query. For convenience, instead of modifying the `postgres` sources, I implemented a very simple preprocessor to do the job and to invoke `postgres` queries. I believe, however, that modifying `postgres` as was done with the other applications would be easy.

All of these programs are I/O intensive and have different file access patterns.

Note that only four applications use prefetching: `cscope`, `dinero`, `glimpse`, and `postgres`. The link-editor application cannot predict its future file references very well because the references are data dependent.

4.4 Experiments

Various combinations of application-controlled caching, prefetching and disk scheduling can be used to improve file system performance. To understand the effect of each technique and how each works with the others, I have compared the performance of six file system implementations incorporating various combinations of the three techniques. I measured the performance of both single applications and multi-application mixes from the suite of I/O intensive applications described in Section 4.3.

4.4.1 File System Configurations

In order to see the performance advantages of the proposed techniques over traditional file system designs, I implemented the ACFS in such a way that it can be configured to use six different combinations of application-controlled caching, prefetching and disk scheduling techniques.

1. *Global LRU*: the traditional Ultrix 4.3 file system consisting of a global LRU file cache with sequential one-block lookahead prefetching on file blocks;

2. *Global LRU + prefetching*: the baseline file system with the addition of application-controlled prefetching: applications can tell the kernel what to prefetch; the kernel uses default LRU for each application's replacement policy and uses LRU-SP for the allocation policy;

3. *Global LRU + prefetching + scheduling*: the same as the previous configuration, with disk scheduling added;

4. *AC caching*: the baseline system with the addition of two-level cache allocation and application-controlled caching;

5. *AC caching + prefetching*: an integration of application-controlled caching and prefetching, using the *aggressive* cache management policy;

6. *AC caching + prefetching + scheduling*: the fully-integrated system which incorporates all of the techniques.

The platform is a DEC 5000/240 workstation. I used file cache sizes of 6.4MB (the default size under Ultrix) and 12MB. The workstation has two disks, an RZ56 and an RZ26. The RZ56 is a 665MB SCSI disk, with average seek time of 16ms, average rotational latency of 8.3ms, and peak transfer rate of 1.875MB/s; the RZ26 is a 1.05GB SCSI disk, with average seek time of 10.5ms, average rotational latency of 5.54ms, and peak transfer rate of 3.3MB/s. The two disks are connected to one SCSI bus. The `cscope`, `dinero` and `glimpse` experiments used the RZ56 disk, while the `postgres` experiments used the RZ26 disk. This set-up allows me to use both disks in some of the multi-process experiments.

4.4.2 Single Process Performance

Figures 23 and 24 show the performance of the seven programs under the six file system configurations with cache size 6.4MB and 12MB. All data points are the average of three runs; variances are mostly less than 2%, and all less than 3.5%.

The figures show that application-controlled caching, prefetching, and disk scheduling each provide significant benefit for at least some of the applications, and that adding these features almost never hurts performance. The runs under the fully integrated system, with all three techniques in use, clearly show the best performance.

Application-controlled caching significantly reduces the number of I/Os. Adding prefetching and disk scheduling under the *aggressive* algorithm does not affect much the total number of disk accesses performed. The only exception is `din` with a 6.4MB cache, which is due to a relatively small fetch cost compared with CPU time and is predicted in the simulation study of Chapter 3. Figures 23 and 24 show that the main factor influencing the number of I/Os is whether or not application-controlled caching is being used.

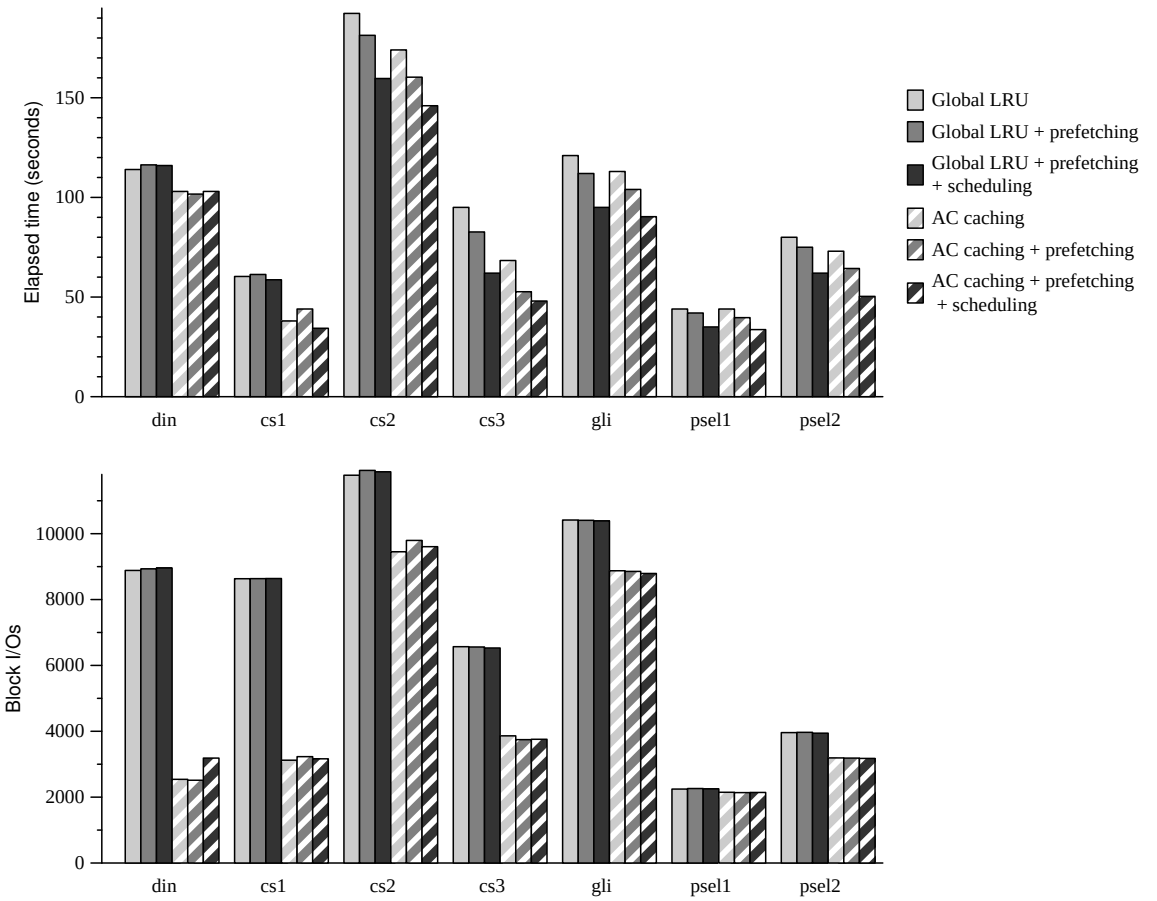

Figure 23: Single application running time, and number of block I/Os, with a 6.4MB file cache.

The applications fall into two classes. The first class consists of `din` and `cs1`; they are programs that primarily access large files sequentially. The second class is made up of the remaining applications: `cs2, cs3, gli, psel1`, and `psel2`. These programs follow more complex access patterns, such as access to a series of small files, or the use of index blocks to choose which data blocks to access.

The applications `din` and `cs1` benefit very little from the *aggressive* prefetching. Since these programs access large files sequentially, one-block lookahead prefetching is very effective for them. Since the traditional file system already implements one-block lookahead, additional prefetching does not help much. These applications also do not benefit much from disk scheduling, because sequential accesses already have

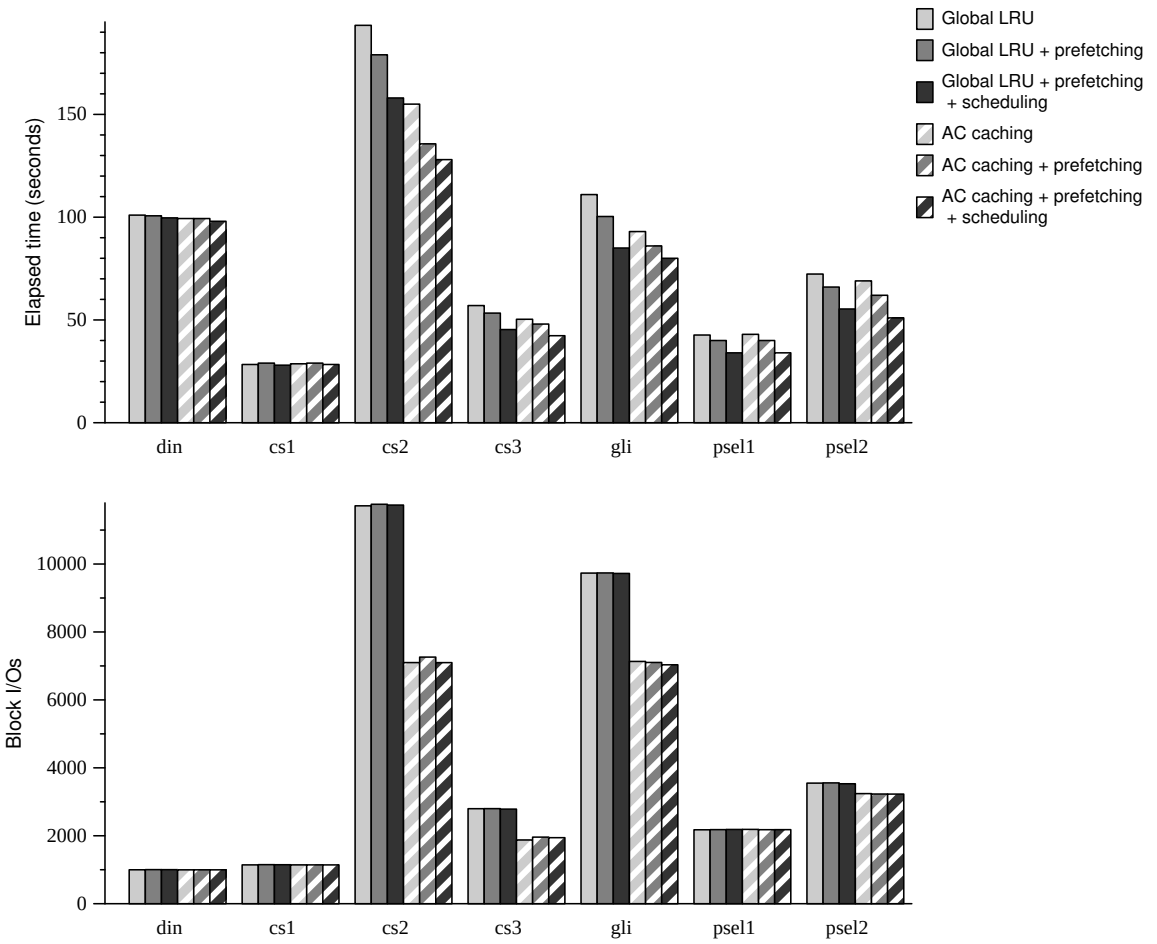

Figure 24: Single application running time, and number of block I/Os, with a 12MB file cache.

excellent locality. For these applications, application-controlled caching is successful in reducing the number of block I/Os, and hence in improving performance.

The behavior of `cs1` under caching+prefetching is surprising. When compared to `cs1`'s performance under caching only, adding prefetching increases the running time but has little effect on the number of block I/Os. Detailed examination of execution traces reveals that the slowdown is due to an increase in the average disk access latency, because aggressive prefetching is disrupting the locality of the replacement blocks, and hence is destroying the locality of future disk fetches. Adding disk scheduling restores the lost locality, and the slowdown vanishes.

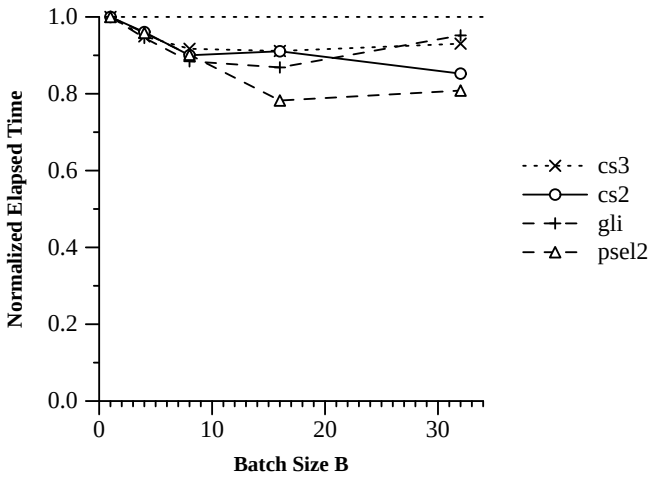

Figure 25: Normalized elapsed times of applications as a function of batch size B (the elapsed time when $B = 1$ is 1.0).

For the remaining six programs, which are not dominated by sequential accesses to large files, the three techniques of application-controlled caching, prefetching, and disk scheduling all provide roughly equal benefits. The techniques appear to complement each other, in the sense that using more than one of them leads to a performance gain which is roughly the combination of the "bonuses" attributable to the individual techniques.

To determine the batch size B used in batch-scheduling of prefetch requests (Section 4.1.2), I measured the elapsed times of several applications (`cs2`, `cs3`, `gli` and `psel2`) in the fully integrated system, varying B. Figure 25 shows the elapsed times of these applications normalized to those when $B = 1$ (i.e. no disk scheduling). The results show that the choice of $B = 16$ works well for most cases.

To understand the source of the performance gains, I measured the average latency of disk accesses (duration from when a request is issued to the disk until it is finished) as a function of B for `cs3`. Figure 26 shows the results. Thus, for applications accessing small files or performing random accesses to large files, disk scheduling substantially reduces the average disk I/O latency and improves performance.

I also measured the user and system CPU times for the applications in these runs,

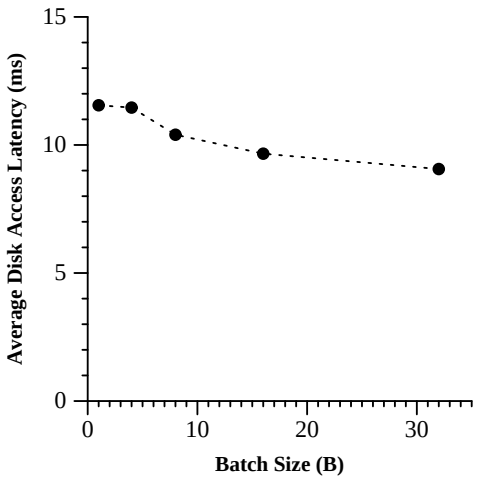

Figure 26: Average disk access latency as a function of batch size B.

to see the CPU overhead of the three techniques. Figure 27 shows the portion of user and system CPU time in the experiments with a 6.4MB cache (that is, those shown in Figure 23). The case with a 12MB cache is similar. Clearly, the techniques have little effect on the user CPU time, as expected. Their effects on the system CPU times, however, are mixed.

Application-controlled file caching usually reduces the system CPU time, but sometimes can increase it. The reason is because, on the one hand, application-controlled file caching incurs CPU overhead to implement application control (explained in Section 4.2.1), but on the other hand, it reduces the number of cache misses and thus reduces the CPU time for handling cache misses. To factor out the effect of fewer cache misses and to measure the processing overhead of application-controlled file caching, I ran the applications using application-controlled cache replacement with LRU as their user-level replacement policy. This way, applications have roughly the same numbers of block I/Os as they do under the traditional file system, and the increase in the system CPU time reflects the CPU processing overhead for application-controlled file caching. My experiments show that application-controlled file caching can increase the system CPU time by about 6% to 14%. However, the increases in the system CPU time have little effect on the elapsed times (only increase them by less than 2.5%), since the system CPU time is a small percentage of the elapsed time

for these applications.

Prefetching can increase the system CPU time for some applications. Figure 27 shows that the increase in the system CPU time can be as high as 30% of the original system CPU time (i.e. in the case of `gli`), though the total execution time is reduced by prefetching. The CPU overhead mostly comes from maintaining the data structures in the prefetch object. Batch scheduling reduces the CPU overhead for prefetching, even though it adds the sorting step. The reason might be that when batch scheduling is used, some procedures (for example, the procedure checking whether a prefetch can be issued, and the procedure issuing the request to the disk driver) are called successively in a loop, thus the locality of the instruction fetches is improved and the hit ratio to the CPU instruction cache is increased. It would require detailed instrumentations and experiments to determine the exact cause.

Although the implementation is not tuned and the techniques incur some CPU overhead, the increase in CPU time is almost always offset by the reduction in I/O time from prefetching and disk scheduling. As the microprocessor performance continues to improve dramatically, the additional CPU overhead for prefetching will not be a concern.

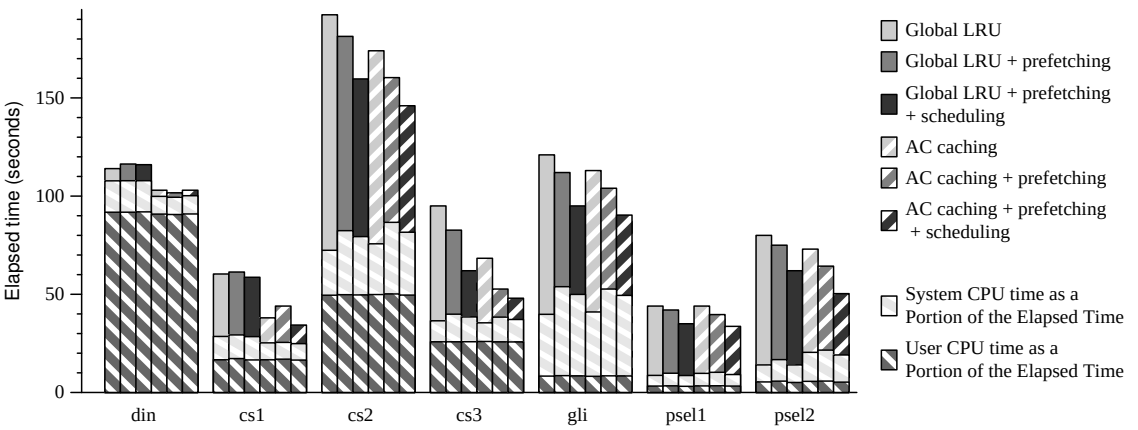

Figure 27: The portion of user and system CPU time for the single application experiments with a 6.4MB file cache.

4.4.3 Multiple Process Performance

I also measured the performance of the file system configurations for multi-process workloads. I ran several pairs of applications, chosen to represent combinations of access patterns: sequential large files and small files (`cs1+cs3`), sequential large files and random large files (`cs1+psel2`), small files and small files (`cs3+gli`), and small files and random large files (`cs3+psel1`). The `cs1+cs3` and `cs3+gli` experiments were run with one disk; the `cs1+psel2` and `cs3+psel1` runs were done with two disks. I also ran the applications concurrently with a copy of Andrew file system benchmark [HKM*88a]: `din+andrew`, `cs2+andrew`, `gli+andrew` and `psel2+andrew`. The Andrew benchmark runs on the RZ56 disk, so only `psel2+andrew` involves both disks.

The results are summarized in Figure 28 and Figure 29. All data points are the average of three runs, except for `cs1+cs3` with 6.4MB cache (15 runs) and `cs3+gli` with 6.4MB cache (6 runs), which are run more times to account for greater variances. Variances are all less than 3.5%, except for `cs1+cs3` under *AC Caching + prefetching* with a 6.4MB cache (15%) and `cs3+gli` under *AC caching + prefetching + scheduling* with a 6.4MB cache (9%).

The results show that the techniques work well in the multi-process case. Application-controlled caching, prefetching, and disk scheduling improves performance individually. Combining all three techniques provides the best performance.

In the single-disk experiments, we see an interesting effect: disk scheduling is particularly important for performance. To understand this, we have to consider how the processes' disk accesses are interleaved.

Without disk scheduling, the accesses of the two processes are finely interleaved, because the processes' execution is interleaved. Indeed, every cache miss causes a context switch, so if misses are common, the execution time is interleaved on a fine scale. In the worst case, the disk must service an access for process A, then one for process B, then one for A, then B, and so on.

If each process has reasonable locality of reference on the disk, the fine interleaving

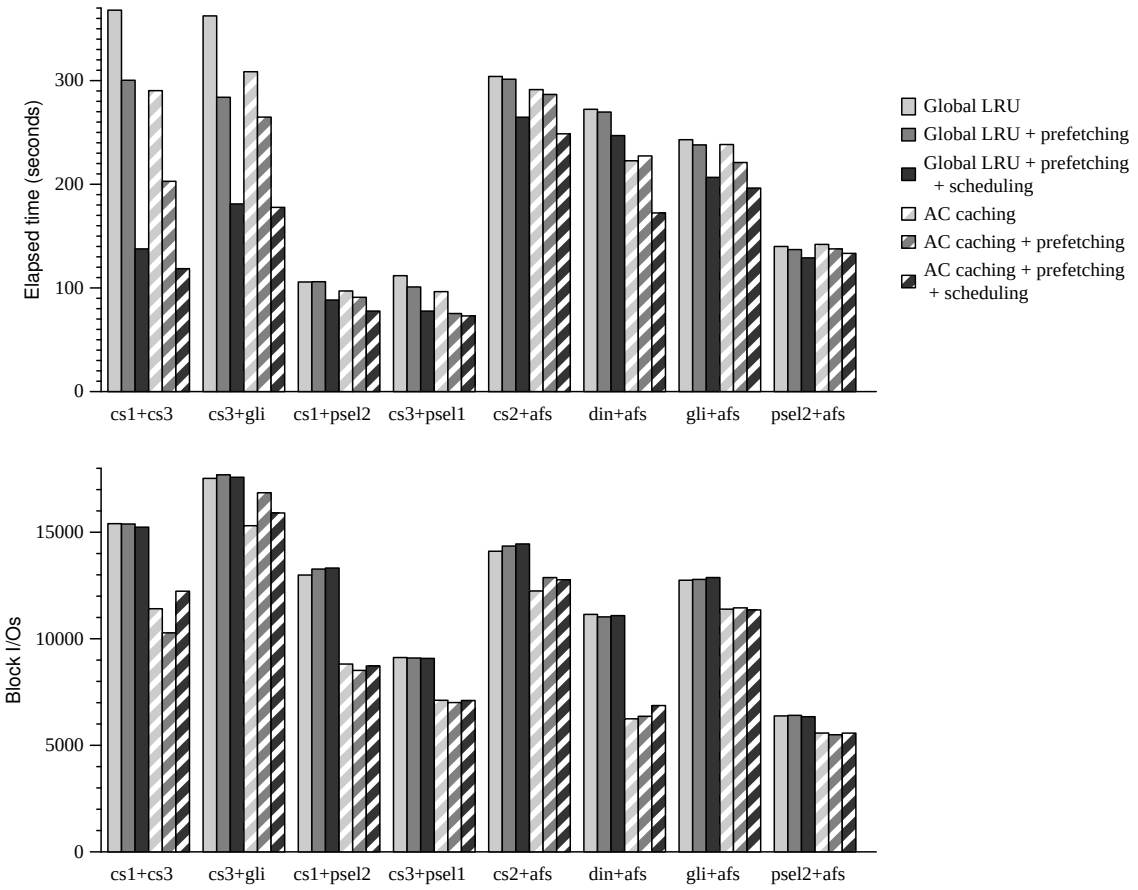

Figure 28: Multiple application running time, and number of block I/Os, with a 6.4MB file cache.

of the reference streams will destroy the benefit of the locality: the disk head will move back and forth between process A's local region and process B's local region. Disk scheduling reduces this effect by batching; the disk will service an entire batch of requests from one process before switching to the other process. Thus, disk scheduling restores most of the locality that the individual reference streams originally had.

Prefetching without disk scheduling also has a similar effect, though it is not nearly as large. Prefetching reduces the number of misses and hence the number of context switches. Fewer context switches mean that each process can, on average, do more disk accesses before giving up the CPU. Thus, the locality of disk accesses is improved.

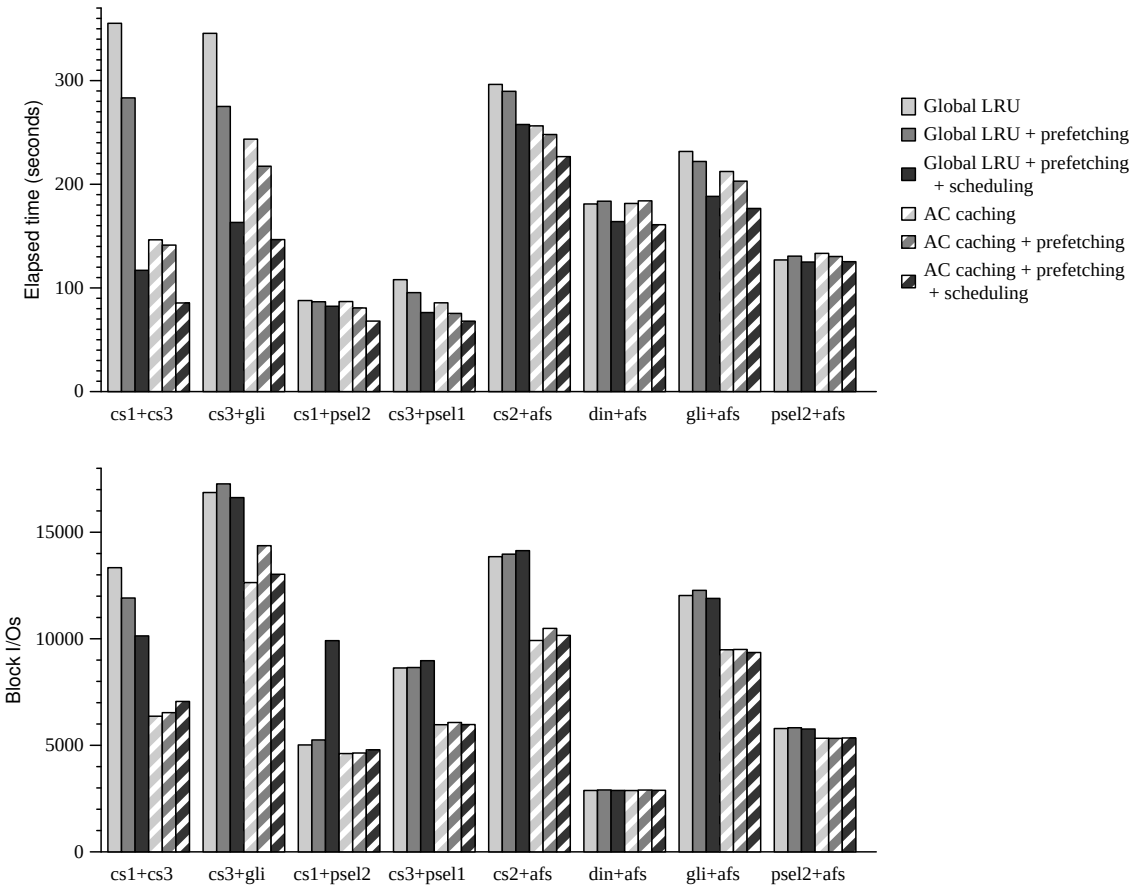

Figure 29: Multiple application running time, and number of block I/Os, with a 12MB file cache.

The results on the number of block I/Os show that the determining factor is whether application-specific replacement policies are used. When applications use good replacement policies, LRU-SP indeed reduces the cache miss ratio of the whole system.

The addition of prefetching and disk scheduling again have little impact on the number of block I/Os, except for the case of `cs1+psel2` under 12MB file cache. I fed the traces of `cs1+psel2` to the calibrated simulator (described in Chapter 3), and found that, because prefetching and scheduling changed the interleaving of accesses from the two applications, global LRU will not always hold the working set of `cs1` in cache. Since `cs1` uses the default LRU replacement policy, its cache misses are very

sensitive to whether the working set fits in cache (no cache miss if it fits, otherwise miss on every access). This shows that future studies must consider potential changes in the interleaving of process execution by prefetching and disk scheduling. Note that when `cs1` uses the MRU replacement policy, which is not sensitive to whether the working set fits in cache, the problem goes away.

Finally, it appears that some runs have high variances. It is not exactly clear why this happens. My hypothesis is that it is the result of a combination of factors: non-uniform disk access latency, prefetching, and the changes in context switches and the interleaving of requests.

4.4.4 Analysis of LRU-SP

Chapter 2 showed, with theoretical analysis and simulations, that both the swapping and placeholders techniques in LRU-SP are necessary for good performance, and that LRU-SP satisfies the allocation criteria described in Section 2.2.3. ACFS provides a testbed to see whether these results are true in practice. In particular, I would like to answer the following questions:

- Is swapping necessary?
- Are placeholders necessary?
- Under LRU-SP, do foolish processes hurt other processes?
- Under LRU-SP, do smart processes hurt other processes?

Below I report experiments designed to answer these questions. Although the experiments are limited, they do help us understand the performance of LRU-SP in real systems.

Is swapping necessary?

LRU-SP keeps a basic LRU list of cache blocks to choose victim processes, but makes two variations, swapping and placeholders, to the list. Swapping means exchanging

Applications		Buffer Cache Size			
		6.4MB	8MB	12MB	16MB
cs2+gli	original	513	510	506	497
	ALLOC-LRU	526	516	494	496
	LRU-SP	479	460	439	402
cs3+ldk	original	279	277	276	267
	ALLOC-LRU	274	271	267	255
	LRU-SP	261	254	234	214
din+cs2+ldk	original	688	688	686	683
	ALLOC-LRU	693	652	562	412
	LRU-SP	593	569	508	450
cs1+gli+ldk	original	608	608	533	475
	ALLOC-LRU	566	521	465	357
	LRU-SP	520	504	444	410
din+cs3+gli+ldk	original	837	832	827	773
	ALLOC-LRU	821	801	793	649
	LRU-SP	747	720	661	615

Table 3: Elapsed times (in seconds) of multiple applications under LRU-SP, ALLOC-LRU and the original kernel.

the position of the candidate block and that of the alternative block in the LRU list (described in detail in Section 2.3.1). To see whether swapping is necessary, let us compare LRU-SP with an allocation scheme that does not perform swapping.

Let's call the basic scheme that simply uses the LRU list to choose victim processes the ALLOC-LRU (ALLOCation by LRU order) policy. With ALLOC-LRU as the allocation policy in two level replacement, I randomly chose the following multi-application combinations: `cs2+gli`, `cs3+ldk`, `din+cs2+ldk`, `cs1+gli+ldk`, and `din+cs3+gli+ldk`, and compared the results to those of LRU-SP. The results of this comparison are shown in Table 3 and Table 4. All numbers are averages of three cold-start runs; variances are less than 2.5%.

The results shows that in most cases, ALLOC-LRU performs worse than LRU-SP. Thus, global allocation policies do affect the performance of the whole system when applications are executed concurrently. Table 3 also shows that in some cases (for example, `cs2+gli` with the 6.4MB cache), under ALLOC-LRU, applications are

Applications		Buffer Cache Size			
		6.4MB	8MB	12MB	16MB
cs2+gli	original	22851	22798	22642	22499
	ALLOC-LRU	23164	22850	22348	21981
	LRU-SP	21701	20969	19322	17640
cs3+ldk	original	12264	12222	12168	11224
	ALLOC-LRU	11755	11605	11143	10458
	LRU-SP	11044	10723	9883	9049
din+cs2+ldk	original	26595	26572	26525	26498
	ALLOC-LRU	26805	25651	23236	17987
	LRU-SP	23653	22746	20893	18493
cs1+gli+ldk	original	25057	25003	22791	20751
	ALLOC-LRU	23909	22767	20978	17635
	LRU-SP	22546	21803	19464	18302
din+cs3+gli+ldk	original	32492	32387	32151	30618
	ALLOC-LRU	31965	31430	30790	27083
	LRU-SP	29702	28610	27040	25181

Table 4: Numbers of block I/Os of multiple applications under LRU-SP, ALLOC-LRU and the original kernel.

better off not using smart policies — smarter replacement does not improve, but actually may degrade, their performance. This is because ALLOC-LRU uses straight LRU order without swapping, and hence penalizes any process that does not replace the LRU block. In the two cases where ALLOC-LRU performs slight better than LRU-SP (`din+cs2+ldk` and `cs1+gli+ldk` with the 16MB cache), it is because ALLOC-LRU allows one process to fit all its file data in cache by penalizing other processes, and thus reduces cache misses of multiple applications. Unfortunately this only happens under particular combinations of application access patterns and file cache sizes.

Therefore, in general, swapping the positions of the candidate and alternative blocks is necessary for good performance.

Are placeholders necessary?

Placeholders are used to prevent foolish applications from adversely affecting the performance of other applications. But do foolish applications indeed harm other

Settings	Elapsed Time of "ReadN"			
	Read390	Read400	Read490	Read500
Oblivious	53	58 (7%)	59	72 (14%)
Unprotected	73 (23%)	89 (24%)	76	122 (27%)
Protected	75	75	72	91 (12%)
Settings	Block I/Os of "ReadN"			
	Read390	Read400	Read490	Read500
Oblivious	1172	1181	1176	1481 (11%)
Unprotected	1300 (19%)	1538 (21%)	1465	2294 (27%)
Protected	1170	1170	1199	1580 (10%)

Table 5: Results from experiments testing the effectiveness of placeholders.

applications if placeholders are not used? If so, are placeholders effective at protecting other applications? To answer these questions, I need an application that can have a foolish replacement policy that performs much worse than LRU, and an application whose performance is sensitive to the number of file cache blocks allocated to it. I wrote a program "`ReadN`" that served both purposes, and performed the following experiments.

`ReadN` sequentially reads the first N 8K-byte blocks from a file in sequence, repeating this sequence four times, then reads the next N blocks five times, and so on. When the cache size k is bigger than N, the LRU replacement policy can keep the most-recently-accessed N blocks in cache and let the repeated accesses hit, while the MRU replacement policy keeps blocks 1 through k in cache and lets any access to block $k+i$ ($i > 0$) miss. Therefore, for `ReadN`, the MRU replacement policy can be much worse than LRU. In my experiments, I use `ReadN` with the MRU replacement policy as the "foolish" application.

`ReadN` also has the characteristic that, under LRU, the cache miss ratio is low when it has at least N cache blocks but very high when it has less than N cache blocks.

Therefore I can compare the number of block I/Os of `ReadN` in different situations as a way of measuring what share of the cache blocks it is getting.

The experiments are the following. I ran an application `Read300` (`ReadN` with N=300) concurrently with another version of `ReadN`. `Read300` is run in two modes: oblivious (letting the kernel apply the default LRU policy for cache replacement), and foolish (controlling its cache replacement with MRU replacement policy). The other version of `ReadN` is used to detect changes in allocation of cache blocks. It is run as an oblivious application (i.e. letting kernel control its cache) and with various values of N: `Read390`, `Read400`, `Read490`, and `Read500`. These values are chosen because the cache size used in this experiment is 6.4MB, or 819 blocks, so `Read300` and `Read390` can comfortably fit in cache together with plenty of space to spare, while `Read300` and `Read500` can just barely fit in cache.

I measured the performance (running time and number of I/Os) of the various `ReadN`'s when the background `Read300` uses both the oblivious replacement policy (i.e. the LRU policy) and the foolish policy (i.e. the MRU policy). If processes are protected from the foolishness of others, then the performance of the `ReadN`'s should not get much worse when the background Read300's policy is changed from oblivious to foolish.

In the case in which the background `Read300` is foolish, I ran two experiments. In the "protected" experiment, the kernel uses the LRU-SP allocation policy, with placeholders protecting the various `ReadN`'s from `Read300`. In the "unprotected" experiment, the kernel uses LRU-SP without placeholders. Thus there are three cases overall:

- Oblivious: the background `Read300` uses the oblivious policy (i.e. the LRU policy);
- Unprotected: the background `Read300` uses the foolish policy (i.e. the MRU policy), and the kernel uses LRU-SP *without placeholders* (i.e. LRU-S);
- Protected: the background `Read300` uses the foolish (MRU) policy, and the kernel uses LRU-SP.

"Read300" Policy	Elapsed Time				Block I/Os			
	din	cs2	gli	ldk	din	cs2	gli	ldk
Oblivious	155	225	156	112	3067	9760	9086	5201
Foolish	202	339	261	208	3495	10542	9759	5374

Table 6: Effect of a foolish process on smart applications.

The results are shown in Table 5. The numbers are averaged over five runs. Variances are shown in parentheses; when omitted they are less than 5%.

The data clearly shows that placeholders protect the oblivious `ReadN` from losing its share of cache blocks. This confirms the results in Chapter 2. However, the data also shows that placeholders did not prevent the increase in elapsed times of `ReadN` (explained below).

Do foolish processes hurt other processes?

The previous experiment has shown that placeholders limit the harm done to oblivious processes by foolish processes. To see whether smart processes are protected as well, I ran each of `din`, `cs2`, `gli` and `ldk`, all using smart policies, concurrently with `Read300`. The results when `Read300` is oblivious and when it is foolish are shown in Table 6.

The table shows the performance of various processes when run concurrently with a `Read300` process, depending on whether the `Read300` is oblivious (using LRU) or is foolish (using MRU). The first four columns shows elapsed time in seconds, and the next four the number of block I/Os. Numbers are averages of three runs with all variances $< 3\%$.

The data show that there are degradations in both the number of block I/Os and the elapsed time. This contradicts results from the simulation study. The reasons include the following:

- Foolish processes generate more I/O requests, and thus put a heavier load on the disk. This leads to longer queueing time for the disk, so non-foolish processes'

I/O requests wait longer to be served.

- Foolish processes take longer to finish and therefore occupy cache blocks for a longer time. Processes that run concurrently with them and finish after them normally get an increase in available cache blocks after they are finished. If they finish later, this increase comes later.

The best way to provide protection from foolish processes is probably for the kernel to revoke the cache-control privileges of consistently foolish applications. Placeholders allow the kernel to tell when an application is foolish, so the kernel can keep track of the mistakes made by the application's policy. If it turns out that a certain percentage of the application's decisions are wrong, the kernel can revoke its right to control its caching.

Do smart processes hurt other processes?

Figures 30 and 31 show the elapsed time and the number of block I/Os of the Andrew file benchmark program when it is run concurrently with several applications (experiments `din+andrew`, `cs2+andrew`, `gli+andrew` and `psel2+andrew`). The total elapsed time and number of block I/Os of these experiments have been reported in Figure 28 and Figure 29 in Section 4.4.3. In these experiments, the Andrew file benchmark program lets the kernel to manage its cache using the default policy; in other words, it is an oblivious application. Under the different file system configurations, with the smart applications running concurrently, the Andrew benchmark's elapsed time and number of block I/Os show the effect of smart applications on oblivious ones.

The results show that when smart applications only use application control on file caching (the fourth bar in each set of results, when only application-controlled file caching is used), they generally do not hurt oblivious applications — the graphs on number of block I/Os show that the Andrew benchmark does not perform more I/Os than under the traditional file system. This confirms the results in Chapter 2. In terms of elapsed time, sometimes smart processes help oblivious ones, for example, in the experiments where the applications run on one disk (`din+andrew` under the

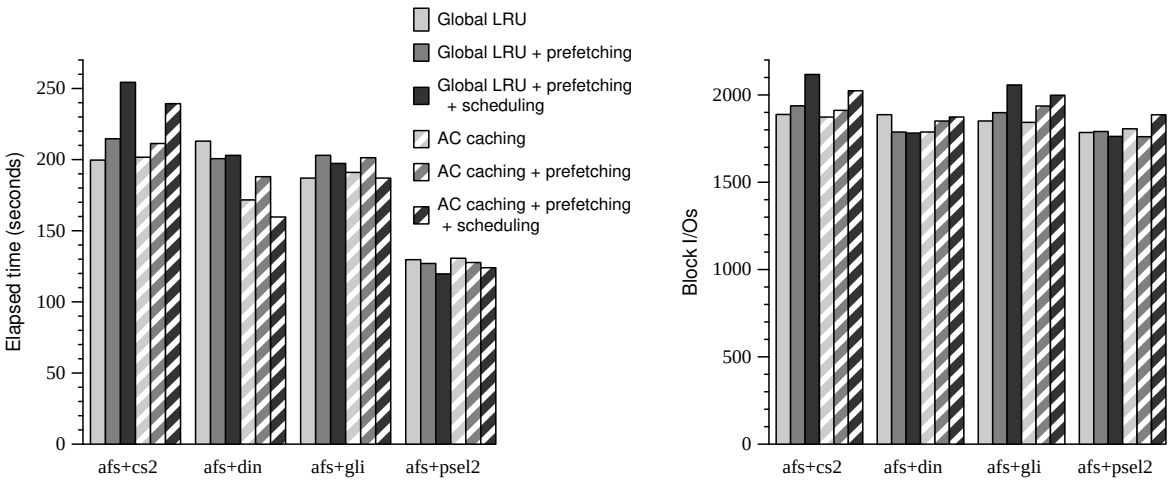

Figure 30: Elapsed time and number of block I/Os of the Andrew benchmark, when running concurrently with the applications, with a 6.4MB file cache.

6.4MB cache, and `cs2+andrew` and `gli+andrew` under the 12MB cache). This is because the reduction in the number of I/Os from smart processes reduces the load on the disk, and speeds up the requests from the oblivious process.

When the two other techniques, prefetching and disk scheduling, are used by smart applications to improve their performance, the oblivious application is affected in mixed ways: the Andrew benchmark's elapsed time is sometimes reduced, sometimes increased, or sometimes not affected much.

In general, a number of factors are involved. Application-controlled caching tends to reduce the elapsed time of oblivious applications because it reduces the number of disk I/Os from smart applications. On the other hand, prefetching with disk scheduling can either increase the elapsed times of oblivious applications because it issues a batch ($B = 16$) of prefetch requests at one time, making oblivious applications' I/O requests wait longer in the queue, or reduce the elapsed times of oblivious applications because it reduces the average I/O cost for requests from smart processes and speeds up these requests, and therefore reduces the waiting time for oblivious applications' I/O requests. Applying all three techniques together produces mixed results, sometimes reducing the Andrew benchmark's elapsed time by up to 25% (`din+andrew`), and sometimes increasing it by up to 20% (`cs2+andrew`).

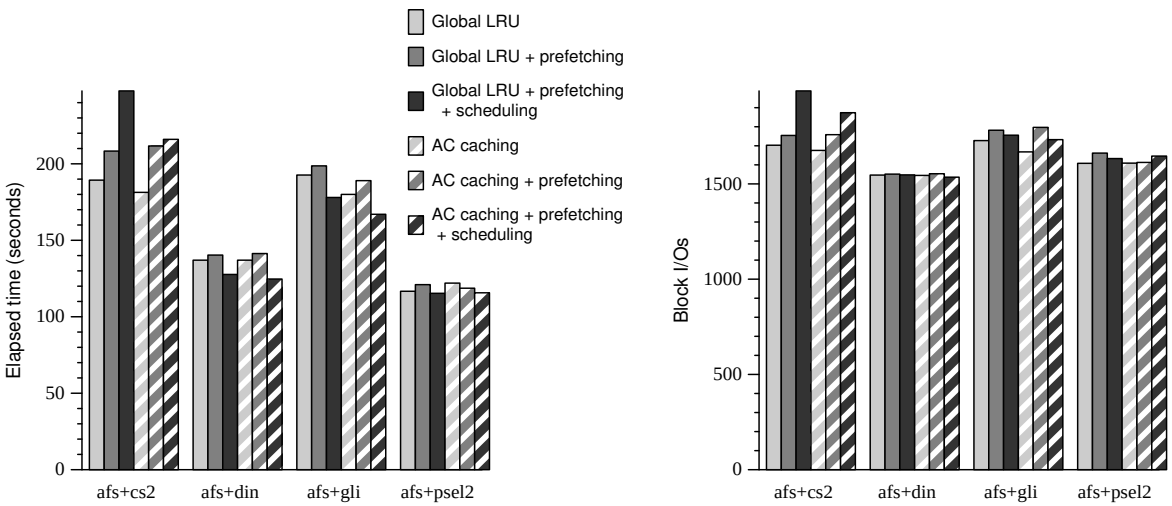

Figure 31: Elapsed time and number of block I/Os of the Andrew benchmark, when running concurrently with the application, with a 12MB file cache.

Note that although the Andrew benchmark's elapsed time is affected in different ways, the total elapsed time when all concurrent applications finish is always reduced by these techniques, as shown in Figure 28 and Figure 29.

When the smart applications use prefetching and batch scheduling, they increase the number of block I/Os of the Andrew benchmark. There can be a number of reasons. Prefetching and batch scheduling from smart applications makes block replacements happen earlier than the traditional file system's limited one-block lookahead prefetching, which might replace more of the Andrew benchmark's blocks. Through prefetching and disk scheduling, smart applications reduce the number and duration of stalls that they suffer because of disk reads, and hence execute faster relative to the oblivious application. This changes the interleaving of file accesses, moving the oblivious application's blocks more toward the least-recently-used end of the LRU list, and therefore might reduce the oblivious application's portion of the file cache. Exactly what causes the increase in Andrew benchmark's number of block I/Os is not clear. Note that the increase in block I/Os does not mean an increase in the elapsed time (due to multiple processes and disk scheduling).

Although the results appear to contradict the predictions from Chapter 2, note that the studies in Chapter 2 did not consider prefetching or replacement of file blocks

due to prefetch requests. The proofs in Chapter 2 also assumed that the interleaving between file accesses from concurrent processes is not affected by the resource management of individual processes. Unfortunately, this is not true in real systems; in particular, prefetching and batch scheduling can affect quite noticeably the CPU scheduling of concurrent processes, and hence the interleaving of file accesses. Prefetching fetches the next missing blocks before they are used to avoid I/O stalls or context switches. Prefetching and disk scheduling reduce the average latency of disk accesses, and thus reduce the duration of I/O stalls, so that the process get rescheduled sooner. Therefore, LRU-SP does not strictly satisfy the criteria for allocation policies in a system that fully integrates all three techniques (application-controlled file caching, prefetching and disk scheduling), due to the differences between the theoretical model and real systems.

Summary of the Results

In conclusion, we see that global allocation policies are important for file system performance in multi-process environments, and in general, LRU-SP performs well. Both the swapping and placeholders techniques are necessary for good performance. Although placeholders do not completely eliminate the harm done by foolish processes, they at least help the kernel to identify foolish processes and take administrative measures to solve the problem.

The experiments show that, in general, LRU-SP satisfies the criteria for allocation policies when applications only control the cache replacement decisions, though it does not eliminate the harm done by foolish applications. However, LRU-SP does not strictly satisfy the criteria in a system that fully integrates application-controlled file caching, prefetching and disk scheduling. The reason is that there are differences between the theoretical model in Chapter 2 and the behavior of the real system that fully integrates all these techniques. In order to find an allocation policy that provides a rigorous theoretical guarantee in the presence of prefetching and disk scheduling, the theoretical model must take into account the interactions among prefetching, disk scheduling, and CPU process scheduling. These interactions are not yet clearly

understood.

4.5 Summary

This chapter describes ACFS, a prototype file system supporting application-controlled file caching and prefetching. ACFS integrates three techniques, application-controlled file cache replacement, application-controlled file prefetching, and disk scheduling to improve file I/O performance.

ACFS uses two-level file cache management to allow application control of file cache management. In two-level file cache management, the kernel allocates cache blocks to processes, and each process manages the use of its cache blocks. In ACFS, the kernel uses LRU-SP as the global allocation policy, and each process applies its own cache replacement policy and *aggressive* prefetching. In addition, each process's prefetching is integrated with disk scheduling under the batch-scheduling algorithm.

To support application control of cache replacement, ACFS provides an interface called priorities and policies. By providing both priority mechanism and implemented common policies, the interface gives applications flexibility to control their caching, and at the same time keeps overhead low for policies commonly used by applications. ACFS also provide a set of system calls for applications to specify their prediction of future file accesses for prefetching.

Using ACFS as a testbed, I experimented with various combinations of application-controlled caching, prefetching and disk scheduling. I used a suite of I/O intensive applications including UNIX utilities, database systems, text information retrieval packages and simulation software. I investigated the performance of both single-process and multi-process workloads.

Experiments show that:

- Careful integration of the three techniques greatly improves file system performance: individual applications' running times are reduced by 3% to 49% (average 26%), and multi-process workloads' running times are reduced by 5% to 76% (average 32%).

- Each technique (application-controlled caching, integrated prefetching and batch scheduling) provides significant performance benefit. Application-controlled file caching reduces the number of block I/Os, while prefetching and batch scheduling overlap the CPU computation with disk accesses and reduce the average latency of disk I/Os.

  Since the techniques target different areas of the I/O bottleneck, they complement each other nicely and the fully integrated system provides the best performance.

- In multi-process environments, disk scheduling combined with prefetching (under the batch scheduling algorithm) increases the locality of disk accesses from concurrent applications and significantly improves performance.

- LRU-SP is crucial to the performance improvement for multiple concurrent applications. Using swapping and placeholder techniques, LRU-SP distributes cache blocks among concurrent applications in a way that allows smart applications to improve their performance and limits the harm done by foolish applications.

As the performance gap between disks and microprocessors becomes wider, the performance of file systems becomes increasingly important. ACFS demonstrates that application-controlled file caching and prefetching combined with disk scheduling can be implemented with low overhead, and can significant improve applications' file I/O performance.

There are a number of limitations in ACFS. ACFS focuses on file accesses and uses default policies for metadata I/Os (i.e. inode and directory I/Os); applying application-controlled file caching and prefetching to metadata blocks can further improve the file system performance. The current implementation of ACFS targets local disk file systems, although it can be extended to network file systems with moderate effort. The operating system in which ACFS is implemented, Ultrix 4.3, separates the file cache from the virtual memory cache (physical memory in the virtual

memory system); it would be interesting to implement ACFS in operating systems that unify the two caches. Finally, ACFS uses the same write-back policy as the Ultrix 4.3 file system; incorporating techniques like log-structured file systems will further improve the performance of ACFS.

Chapter 5

Conclusions

File system performance is becoming an increasingly important part of system performance. This dissertation proposes two techniques to alleviate the file I/O bottleneck: application control of file cache replacement and integrated prefetching of file blocks. It describes a framework to support application control, a solution to the global allocation policy problem, and a solution to the problem of integrating cache replacement and prefetching. A prototype implementation, ACFS, demonstrates that these techniques can greatly improve file system performance.

5.1 Summary of Results

The most important contributions of this dissertation are:

- *Global allocation policy LRU-SP.* This is the first published allocation algorithm that provides both a theoretical guarantee on the performance in multi-process environments and good performance in practice.
- *Integrated algorithms for caching and prefetching.* The study of the interaction between caching and prefetching yields an algorithm, *aggressive*, that is provably close to optimal in the theoretical model, simple, and very effective in practice.

- *A complete design integrating application-controlled file caching, prefetching and disk scheduling.* The scheme, two-level file cache management, separates the problem into local management policies of user level processes and the global allocation policy in the kernel. It carefully integrates file caching, prefetching and disk scheduling, and fully exploits the performance benefits of and the synergy among these techniques. In addition, the "priorities and policies" interface demonstrates that it is possible to design an application programming interface for application control that is both fast and flexible.

- Experimental results demonstrating the performance benefits of application-controlled file caching, prefetching and disk scheduling.

With ACFS as a testbed, experiments demonstrate that careful integration of application-controlled file caching, prefetching and disk scheduling greatly improves file system performance: individual applications' running times are reduced by 3% to 49% (average 26%), and multi-process workloads' running times are reduced by 5% to 76% (average 32%). Each of the techniques provides substantial performance benefits. In particular, disk scheduling incorporated in aggressive prefetching reduces the average disk access latency, improves the locality of disk accesses in multi-process environments, and significantly improves file system performance. The three techniques complement each other nicely, and the fully integrated system provides the best performance.

5.2 Lessons

The most important lesson I learned from my thesis work is about the feedback loop between theoretical studies and system developments. In the thesis work I have been applying theoretical techniques to problems that arise in practical systems. For both the global allocation problem and the integrated prefetching problem, I was able to construct abstract models simple enough that I could attack them first from a theoretical perspective, from which I gained insight into the problems and even found

solutions to them. I then used simulations to evaluate the solutions on real traces and to perform a preliminary validation of theoretical results.

Yet theoretical and simulation studies are often limited. Using ACFS, I experimented with the algorithms and compared the results in practical systems with those predicted by theory. I found that the two set of results do not always agree with each other. In the process of finding what causes the discrepancy, I discovered many differences between the real system and the theoretical or simulation model. The differences then led to more accurate models and simulations, and more studies to improve the algorithms for better performance.

Through this process I learned more about the various aspects of the I/O subsystems in a computer, about differences between theory, simulations, and practice, and how studies in these areas complement each other. I also discovered that many performance-related problems, if modeled appropriately, are amenable to the approach used here, an approach that combines theoretical analysis, simulations and implementation validation.

5.3 Future Work

A recent trend in kernel structures has been toward providing applications more capability to control the usage of their own resources. The results in this dissertation support this approach: application-controlled file caching and prefetching can substantially improve the performance of the file system. Other studies have shown the benefit of application control in multi-process operating systems [And91] and in physical memory management in virtual memory [MA90, SP91, HC92]. As the diversity of applications increases, it is becoming more important for the operating system to allow applications to manage their resources for best performance.

An important area for future work is the development of tools for applications or runtime systems to automatically detect access patterns and find good replacement policies. In my work so far, the application-specific policies have been discovered by program inspections and trace simulations. It might be possible for a runtime system

to automatically find good cache management policies for applications with help from compile-time analysis.

There are many theoretical questions left unanswered. It is not clear what is the optimal algorithm for integrating prefetching and disk scheduling, or whether the problem is NP-complete. How prefetching will interact with delayed writebacks is not clearly understood. The integration algorithms for caching and prefetching in the presence of multiple memory hierarchies need more study as well. Finally, the experience with ACFS suggests that process scheduling policies should be integrated with cache allocation and disk scheduling policies for maximum throughput; how these policies interact with each other and how they should be integrated are not clearly understood.

Chapter 2 described how swapping and placeholder techniques apply to application-controlled physical memory management in virtual memory systems. The algorithm that integrated caching and prefetching, *aggressive*, applies to virtual memory systems as well. In addition, although the implementation of ACFS has focused on file systems on local disks, the implementation can be extended easily to client caching and prefetching in network file systems; applying application-control for server caching in network file systems needs more careful study. As discussed in Section 1.5.4, there are many other techniques in addition to application-controlled file caching and prefetching, including log-structured file systems, non-volatile RAM, and compression. It would be interesting to build a file system that incorporates all of these techniques for high performance.

Finally, although this thesis focuses on uniprocessor file systems, the techniques developed here have broader applications. One area of particular interest is parallel file systems. Prefetching, and the integration of prefetching, caching and disk scheduling, are especially important on parallel disks. The parallel integrated prefetching problem is similar to the integrated prefetching problem studied in this thesis. However, parallel prefetching also involves data distribution on disks, and hence is much more complicated than prefetching in the single disk case. Application-controlled file

caching should improve buffer cache hit ratio in parallel file systems as well. Parallel file systems need to accomodate multiple processes' demand on the buffer cache resource, and the buffer allocation problem in parallel file systems is similar to the one studied in this thesis, though more complicated. Since many parallel applications today demand high-performance I/O, it is important to investigate application-controlled file caching and prefetching in parallel file systems.

Bibliography

[ADN*95] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In *To appear in Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[Amd67] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[And91] Thomas E. Anderson. *Operating System Support for High Performance Multiprocessing.* PhD thesis, University of Washington, August 1991. Also published as technical report no. CSE 91-08-10.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall Software Series, 1986.

[BAD*92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[BCE*94] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan MaNamee, Przemystaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN — An Extensible Microkernel for Application-Specific Operating

System Services. Technical Report TR 94-03-03, Dept. of Computer Science and Engineering, University of Washington, 1994.

[BG88] D. Bitton and J. Gray. Disk Shadowing. In *Proceedings of the 1988 VLDB Conference*, pages 331–338, September 1988.

[BGV95] Rakesh D. Barve, Edward F. Grove, and Jeffrey Scott Vitter. Application-Controlled Paging for a Shared Cache. In *Proceedings of the 1995 FOCS Conference*, 1995.

[BHK*91] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 198–211, October 1991.

[BJLM92] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line Data Compression in a Log-structured File System. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–9, October 1992.

[Bla93] Matthew A. Blaze. *Caching in Large-Scale Distributed File Systems.* PhD thesis, Princeton University, January 1993.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. Also available as U. Washington CS TR 92-06-03.

[CD85] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 127–141, August 1985.

[CFKL95] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. 1995 ACM SIGMETRICS*, pages 188–197, May 1995.

[CFL94a] Pei Cao, Edward W. Felten, and Kai Li. Application-Controlled File Caching Policies. In *Proc. USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[CFL94b] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. First USENIX Symposium on Operating Systems Design and Implementation*, pages 165–178, November 1994.

[Che87] David Cheriton. Effective Use of Large RAM Diskless Workstations with the V Virtual Memory System. Technical report, Dept. of Computer Science, Stanford University, 1987.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[CKV93] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 257–266, May 1993.

[CLG*94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, pages 26(2):145–187, June 1994.

[dJKH93] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, pages 15–28, December 1993.

[DWAP94] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[EK89] Carla Schlatter Ellis and David Kotz. Prefetching in File System for MIMD Multiprocessors. In *1989 International Conference on Parallel Processing*, pages 306–314, August 1989.

[FKL*91] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. On Competitive Algorithms for Paging Problems. *Journal of Algorithms*, pages 12:685–699, 1991.

[FMP*95] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, Henry M. Levy, and Chandramohan A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *To appear in Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[GA94] Jim Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[GCD73] Edward G. Coffman, Jr. and Peter J. Denning. *Operating Systems Theory*. Prentice-Hall, Inc., 1973.

[GGV90] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ACM International Conference on Supercomputing*, pages 354–368, June 1990.

[Gin77] J. D. Gindele. Buffer block prefetching method. *IBM Technical Disclosure Bulletin*, 20(2):696–697, July 1977.

[GMB81] J. N. Gray, P. McJones, and M. Blasgen. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, pages 13(2):223–242, June 1981.

[GMS87] R. Gingell, J. Moran, and W. Shannon. Virtual Memory Architecture in SunOS. In *Proceedings of the USENIX Conference*, Summer 1987.

[GP94] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, pages 49–60, November 1994.

[Gra91] Jim Gray. *The Benchmark Handbook*. Morgan-Kaufman, San Mateo, Ca., 1991.

[Hag87] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 155–162, November 1987.

[HC92] Kieran Harty and David R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, October 1992.

[HKM*88a] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, pages 6(1):51–81, February 1988.

[HKM*88b] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, pages 6(1):51–81, February 1988.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[JW91] David Jacobson and John Wilkes. Disk Scheduling Algorithms based on Rotational Positioning. Technical report, Hewlett Packard Laboratories, HPL-CSP-91-7, February 1991.

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Tom Anderson. Tools for the Development of Application-Specific Virtual Memory Management. In *OOPSLA '93 Conference Proceedings*, pages 48–64, October 1993.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill International Editions, Computer Science Series, 1986.

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, pages 6(1):1–25, February 1992.

[LL82] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, pages 35–41, March 1982.

[MA90] Dylan McNamee and Katherine Armstrong. Extending The Mach External Pager Interface To Accommodate User-Level Page Replacement Policies. In *Proceedings of USENIX Mach Symposiumi '91*, pages 17–29, 1990.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[Mic90] Sun Microsystems. *SUNOS 4.1 Programmer's Manual.* Sun Microsystems, Inc. Mountainview, California, 1990.

[MJLF84] M. Mckusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, August 1984.

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *1991 Winter USENIX*, pages 33–43, 1991.

[Mog94] Jeffrey C. Mogul. A Better Update Policy. In *Proceedings of 1994 Summer USENIX*, pages 99–111, June 1994.

[MW94] Udi Manber and Sun Wu. GLIMPSE: A Tool to Search Through Entire File Systems. In *Conference Proceedings of the USENIX Winter 1994 Technical Conference*, pages 23–32, January 1994.

[Nel88] M. N. Nelson. *Physical Memory Management in a Network Operating System.* PhD thesis, University of California, Berkeley, November 1988. Also published as Technical Report UCB/CSD 88/471.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite File System. *ACM Transactions on Computer Systems*, pages 6(1):134–154, February 1988.

[OCH*85] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proc. 1993 ACM-SIGMOD Conference on Management of Data*, pages 297–306, May 1993.

[PBD93] Christos A. Polyzois, Anupaam Bhide, and Daniel M. Dias. Disk Mirroring with Alternating Deferred Updates. In *Proceedings of the 19th International Conference on Very Large Data Bases*, 1993.

[PG94] R. Hugo Patterson and Garth A. Gibson. Exposing I/O Concurrency with Informed Prefetching. In *Proc. Third International Conf. on Parallel and Distributed Information Systems*, September 1994.

[PGG*95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *To appear in Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD Conference on the Management of Data*, pages 109–116, 1988.

[PGS93] Hugo Patterson, Garth Gibson, and M. Satyanarayanan. Transparent Informed Prefetching. *ACM Operating Systems Review*, pages 21–34, April 1993.

[PH94] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* Morgan Kaufmann Publishers, Inc., 1994.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September 1991.

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.

[RO91] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[Ros92] Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System.* PhD thesis, University of California at Berkeley, June 1992. Also published as technical report no. UCB/CSD 92/696.

[SBK*85] R. Sandberg, D. Boldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Summer Usenix Conference Proceedings*, pages 119–130, June 1985.

[SBMS93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of 1993 Winter USENIX*, January 1993.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proc. of USENIX Winter 1990 Technical Conference*, pages 313–324, 1990.

[SG86] K. Salem and H. Garcia-Molina. Disk Striping. In *Proceedings of the 2nd International Conference on Data Engineering*, pages 336–342, February 1986.

[SGN85] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. *ACM Operating Systems Review*, pages 19(5):35–50, December 1985.

[Smi78] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978.

[SP91] Stuart Sechrest and Yoonho Park. User-Level Physical Memory Management for Mach. In *Proceedings of the USENIX Mach Symposium*, pages 189–199, 1991.

[SS95] Margo Seltzer and Keith A. Smith. File System Logging Versus Clustering: A Performance Comparision. In *Proceedings of 1995 Summer USENIX*, June 1995.

[ST85] D. D. Sleator and R. E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, pages 28(2):202–208, February 1985.

[Ste85] Joseph L. Steffen. Interactive Examination of a C Program with Cscope. In *USENIX Dallas 1985 Winter Conference Proceedings*, pages 170–175, 1985.

[Sto81] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM, v. 24, no. 7*, pages 412–418, July 1981.

[TD90] Carl D. Tait and Dan Duchamp. Detection and Exploitation of File Working Sets. Technical Report CUCS-050-90, Computer Science Department, Columbia University, 1990.

[TE93] Dean M. Tullsen and Susan J. Eggers. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. In *Proceedings of the 1993 International Symposium on Computer Architecture*, pages 278–288, May 1993.

[The93] The Postgres Group. POSTGRES Version 4.1 Release Notes. Technical report, Electronics Research Lab, University of California, Berkeley, February 1993.

[WGP94] Bruck L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling for Modern Disk Drives and Non-Random Workloads. Technical Report CSE-TR-194-94, Dept. of Electrical Engineering and Computer Science, University of Michigan, March 1994.

[Wil65] Maurice Wilkes. Slave Memories and Dynamic Storage Allocation. *IEEE Transactions on Electronic Computers*, pages EC–14(2):270–271, April 1965.

[WM91] D. L. Whiting and R. A. Monsour. Data Compression Breaks Through to Disk Memory Technology. *Computer Technology Review*, 1991. Reprints also available from Stac Electronics, 5993 Avenida Encias, Carlsbad, California 92008.

[WYT93] Kun-Lung Wu, Philip S. Yu, and James Z. Teng. Performance Comparison of Thrashing Control Policies for Concurrent Mergesorts with Parallel Prefetching. In *Proceedings of 1993 ACM SIGMETRICS*, pages 171–182, May 1993.

[WZ94] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.