Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching and Disk Scheduling

Pei Cao * Edward W. Felten* Anna R. Karlin † Kai Li *

Abstract

As the performance gap between disks and microprocessors continues to increase, effective utilization of the file cache becomes increasingly important. Application-controlled file caching and prefetching can apply application specific knowledge to improve file cache management. However, supporting application-controlled file caching and prefetching is nontrivial because caching and prefetching need to be integrated carefully, and the kernel needs to allocate cache blocks among processes appropriately.

This paper presents the design, implementation and performance of a file system that integrates application-controlled caching, prefetching and disk scheduling. We use a two-level cache management strategy. The kernel uses the LRU-SP (Least-Recently-Used with Swapping and Placeholders) policy to allocate blocks to processes, and each process integrates application-specific caching and prefetching based on the *controlled-aggressive* policy, an algorithm previously shown in a theoretical sense to be near-optimal. Each process also improves its disk access latency by submitting its prefetches in batches so that the requests can be scheduled to optimize disk access performance. Our measurements show that this combination of techniques greatly improves the performance of the file system. We measured that the running time is reduced by 3% to 49% (average 26%) for single-process workloads, and by 5% to 76% (average 32%) for multi-process workloads.

1 Introduction

In the last decade, advances in hardware technology have created a wide performance gap between microprocessors and disks. As the gap increases, a computer system's performance is increasingly limited by the part that primarily

*Department of Computer Science, Princeton University, Princeton, NJ.
†Department of Computer Science, University of Washington, Seattle, WA.

involves disks, the file system. Thus, today's operating systems use a file cache, which is a portion of DRAM memory set aside to cache and prefetch file data.

Traditional file systems manage the file cache using fixed replacement and prefetching policies. The replacement algorithm is usually Least-Recently-Used or its approximation: if a cache miss occurs, the block whose last reference was earliest among all cached blocks is replaced. The prefetching algorithm is usually sequential prefetching: either one-block-lookahead (i.e. if block K and block K+1 of a file have been referenced, prefetch block K+2) or extent-based prefetching (when a cache miss happens, fetch not only the requested block but also a number of adjacent blocks).

Although these fixed algorithms generally perform well, application-specific policies can often perform much better. Many researchers have pointed out the advantage of application-specific replacement policies, in the context of both file caching and physical memory management [CD85, SP91, HC92]. However, devising fair and efficient policies for allocating file cache space among multiple competing processes remains a difficult problem. Recent studies have also demonstrated the benefits of application-specified prefetching [TD90, GA94, PGG+95]. However, the question of how aggressively to prefetch remains a difficult problem. Finally, how to integrate caching and prefetching policies for optimal performance is a difficult issue.

This paper presents the design, implementation and performance of "ACFS" (Application-Controlled File System), a file system that integrates application-controlled file caching with prefetching. ACFS uses a two-level cache management strategy to allow applications to exert control over file cache replacement and to prefetch file data, and retains for the kernel the allocation of cache blocks to concurrent processes. ACFS 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 allocates file cache space among multiple processes properly so that applications can use these techniques without unduly harming each other or degrading the performance of the whole system.

There are two challenges in the design of ACFS. The first is the proper integration of caching and prefetching. 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 hurt performance because new and possibly better replacement opportunities open up as the program proceeds.

ACFS handles the integration issue using an algorithm we call "controlled-aggressive". Previous studies showed that in a simplified theoretical model, the elapsed time of "controlled-aggressive" is always close to that of the optimal algorithm. In addition, since multiple prefetch requests can be generated at a time, average disk latency can be improved by reordering the multiple requests. Thus, ACFS issues prefetch requests in batches and uses disk scheduling to reorder service within a batch.

The second challenge in the design of ACFS is how to allocate cache space among multiple competing processes. There are a number of concerns. The allocation should be fair in the sense that a process receives its share of file cache space no matter what cache management policy it is using. The allocation should be robust in the sense that no process can manipulate the allocation by choosing bad cache management policies. Finally, when multiple processes execute concurrently, the complex interaction between caching, prefetching and CPU scheduling makes prediction of future reference patterns more difficult. Care must be taken to ensure that performance is not adversely affected by this interaction.

ACFS addresses these concerns using the LRU-SP (Least-Recently-Used with Swapping and Placeholders) allocation policy. Previous studies of application-controlled file caching [CFL94b] showed that LRU-SP not only performs well, but also is fair and relatively robust. In addition, to reduce the effect of false prefetching, ACFS matches an application's file accesses with its prediction and only allows the application to prefetch if its predictions have been reasonably accurate.

We have measured ACFS's performance with a suite of I/O intensive applications on a DEC 5000/240 workstation. We considered both sequential and concurrent executions of these applications with various combinations of application-controlled caching, prefetching and disk scheduling. Our results show that careful integration of these techniques greatly improves the performance of the file system: the elapsed times of the sequential executions can be reduced by up to 50% with average 26%, and those of the concurrent executions can be reduced by up to 76% with average 32%.

2 Integration for Single-Process Case

We first consider how to integrate application-controlled caching, prefetching, and disk scheduling when only a single process is using the file system. The next section will describe how to extend this integration to the multiple-process case.

We focus on applications that can predict their future access patterns. Research has shown that such predictions are often possible in practice [PG94, GA94, TD90, Smi78].

2.1 Integrating Prefetching with Caching

Since both application-controlled caching and prefetching rely on knowledge of future access patterns, it is natural to try to integrate them. While it may

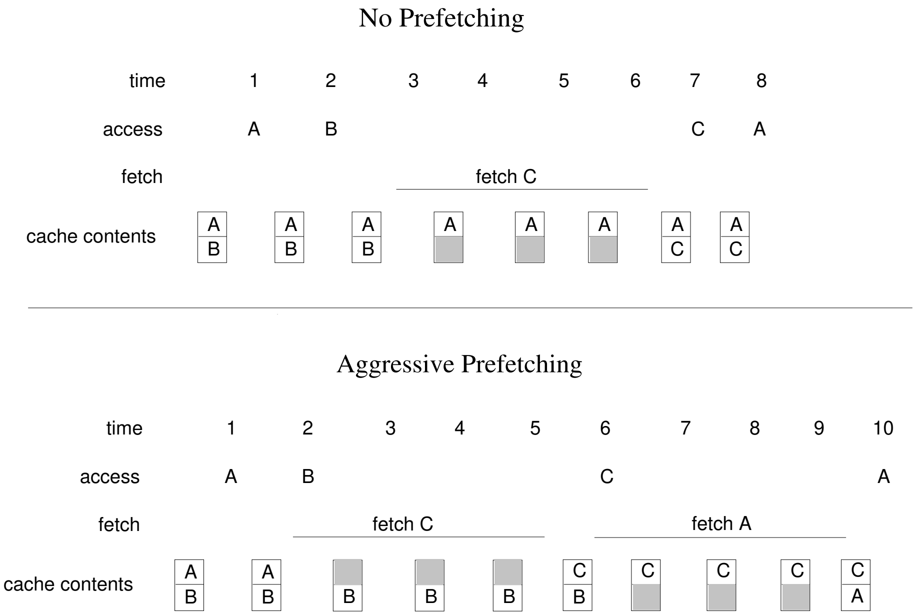

Figure 1: A file access stream for which prefetching hurts performance. Eight time units are required in the absence of prefetching; ten time units are required when prefetching is done. (Optimal replacement is assumed in both cases.)

seem at first glance that this is easy, this is not the case. The main reason is that issuing a prefetch requires that a cache block be reserved to hold the block being fetched. This may lead to earlier cache replacement than in the no-prefetching case. Thus, prefetching file blocks into a cache can be harmful even if the blocks will be accessed in the near future.

2.1.1 An Example

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

The top half of Figure 1 shows a no-prefetching policy using the optimal replacement algorithm. The first two references hit in the cache. The third reference (to C) misses in the cache, thus triggers a fetch of C, replacing B at time 3. Finally, the fourth reference hits in the cache. 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, the bottom half of Figure 1 shows that a policy that prefetches whenever possible (while making optimal replacement choices) takes *ten* time units to execute this sequence. The policy decides to prefetch C at time 2, resulting in the replacement of A because B is in-use at the time, thus the fourth reference (to A) misses in the cache.

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.

2.1.2 Theoretical Results

The example raises the following natural question. Given some amount of lookahead in the file block reference pattern, what is the optimal combined prefetching and caching strategy? In [CFKL95] we derived several theoretical results about this problem. These results are summarized here.

We abstracted the real problem to the following theoretical model:

- The cache holds K blocks.
- The entire sequence of future block references is known in advance.
- The program references one block per time unit. If the block to be referenced is not in the cache, the program must wait until the block arrives.
- Fetching a block from disk requires F time units, and only one fetch may be in progress at any time.
- When a fetch is initiated, a block must be discarded from the cache; the discarded block becomes unavailable at the moment the fetch is initiated.
- The goal is to minimize the total running time, or equivalently to minimize the time spent waiting for fetches to complete.

We observed that any optimal prefetching algorithm must obey the following four rules. We state the rules here without proof.

- **Rule 1: Optimal Prefetching** *Every prefetch should bring into the cache the next block in the reference stream that is not in the cache.*
- **Rule 2: Optimal Replacement** *Every prefetch should discard the block whose next reference is furthest in the future.*

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

- **Rule 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 first two rules uniquely determine what to do, once the decision to prefetch has been made. The problem is reduced to the question of *when* to prefetch. Thus, we can imagine a good strategy as answering a series of yes/no questions of the form "Should I prefetch now?"

The last two rules provide some guidance about when to prefetch. Specifically, they allow us to detect some situations when the answer to the "Should I prefetch now?" question must be "no." We can think of these rules as reducing the number of times the question must be asked.

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.

The *controlled-aggressive* policy is the one that always answers "yes" whenever Rules 3 and 4 allow it[1]. In other words, it tries to fetch the next block in the access stream that is not in cache, as soon as the disk is idle and there is a block in cache that will be accessed after the missing one. In addition, it always does optimal replacement.

We proved in [CFKL95] that the running time under *controlled-aggressive* is never more than $1 + F/K$ times the optimal running time. In the case of file prefetching, F can be thought of as the ratio of the average disk access time over the average CPU time in between file accesses (the CPU time includes the time to copy file data from kernel address space to user address space), and K is the number of blocks (i.e. unit of caching, usually 8K bytes) in the file cache. In many systems, the file cache is big enough such that F/K is typically less than 0.02. Thus, *controlled-aggressive* can perform very close to optimal.

We also performed simulation studies in [CFKL95] comparing *controlled-aggressive* and six other existing prefetching approaches, including the one-block lookahead prefetching used in many file systems. Using real file reference traces and actual computation time between file accesses, the simulation showed that "controlled-aggressive" performs the best among all algorithms, and is quite close to optimal in terms of applications' elapsed times.

2.1.3 Controlled-Aggressive with Limited Lookahead

The above discussion assumed that the knowledge of future access is perfect. In practice, however, only imperfect and limited lookahead knowledge is available. Fortunately, even though the near-optimality of "controlled-aggressive" depends

[1] In [CFKL95] this policy was called *aggressive* to denote that it is the most aggressive of all reasonable policies. We chose to rename it here because we felt the name *aggressive* is misleading when taken out of context.

on availability of perfect knowledge, the algorithm itself can be adjusted to handle the situation when only limited lookahead knowledge is available.

Applications' knowledge of their future references is often incomplete in three aspects: some file accesses are not predicted; some predicted accesses do not happen; or an application cannot predict all of its future accesses but only the first N of them. Occasionally, an application might give a list of future file accesses that is completely bogus.

ACFS deals with these inaccuracies by attempting to match the application's actual references against its prediction. In the case of an unpredicted file reference, the reference is simply serviced as a regular file access — if the request hits in the file cache, it reads from the cache; otherwise, ACFS chooses an optimal replacement block, issues the demand read request, and does not issue any prefetch until the demand request finishes. In the case of a predicted access that did not actually occur ACFS simply "pretends" that all of the skipped accesses already happened, and adjusts the replacement priority of the involved blocks according to the next accesses to them (for example, if an involved block will not be accessed again, it will be replaced first). To handle the case of a completely bogus list, if an application's list of predicted accesses is too often wrong, the list is discarded.

If an application only predicts the next N file accesses, "controlled-aggressive" will prefetch only the missing blocks in those accesses. This means that "controlled-aggressive" will not be able to fetch some blocks as early as it could under complete lookahead. However, a more serious problem is that the limited access list may not reveal what optimal replacement choices are .

To alleviate this problem, "controlled-aggressive" seeks help from the application's control over file cache replacement in ACFS. The goal is to make replacement choices as close to optimal as possible. If the list of predicted file accesses is long enough to uniquely identify the optimal replacement, then the block is replaced. Otherwise, application-controlled file cache replacement is used; the cached blocks which no longer appear in the list are first identified, then the replacement is chosen among them by the application using information on its access pattern.

In general, application-controlled file caching policies and the list of predicted file accesses may give conflicting information on what is the best replacement choice. How to best resolve the conflict depends on one's assumption of why conflicts occur. Our approach is based on the assumption that the list of predicted accesses is more accurate about the application's near-term behavior, while the file caching policy is more accurate about the long-term behavior. The issue, however, deserves more careful study.

Our prototype implementation deviates from the above design and relies more heavily on application-controlled file caching for making replacement choice (see section 4.3). This is purely for the ease of implementation. Thus, in some sense we only implemented an approximation of our design.

Finally, note that our specific way of handling inaccuracies in applications' predictions is not necessarily optimal. Study of optimal algorithms in this case requires modelling the probability of the occurrence of various inaccuracies, and is beyond the scope of this paper. Thus, although "controlled-aggressive" is close to optimal given perfect knowledge, we do not claim that our implementation is close to optimal in real systems.

2.2 Incorporating Disk Scheduling

Because of the physical attributes of disks, careful scheduling of disk accesses can provide significant improvement in performance [SCO90, JW91]. Without prefetching, scheduling opportunities only come from asynchronous I/O activities or multiple processes. Prefetching provides new opportunities for disk scheduling, because prefetch requests can be generated in groups.

However, scheduling prefetching 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 overlapping between I/O and CPU computation. For example, suppose the application computes, reads block X, computes again, then reads block Y. If the scheduling algorithm is first-come-first-serve, the computation before block X will overlap with the I/O for block X, and the computation between X and Y will overlap with Y's I/O time. However, if the scheduling algorithm reads Y before X, the computation before the reading of X will have to overlap with both X and Y's I/O time, resulting in the CPU being stalled longer for I/O. Hence, scheduling prefetches is a nontrivial problem, and we know of no theoretical study of this problem yet.

We employ a simple heuristic: limited batch scheduling. Every time the disk becomes idle, the prefetcher tries to issue a batch of prefetch requests, instead of just one request. There is a limit, B, on the batch size; that is, the prefetcher will not issue more than B requests at a time. These requests are issued to the disk driver, which then sorts them and all other requests into order of increasing logical block number, so that disk fetches are performed in sorted order.

We 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 might 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 logical-block-number in the batch, then the CPU will have to wait for $B/2$ disk accesses to complete before the first missing block arrives in the cache. The proper choice of B balances these two factors. We found by experience (see Section 5.3) that $B = 16$ worked well for our system.

The scheduling algorithm at the disk driver should try to minimize the average disk access latency. We use logical-block-number ordering. It is an approximation to the optimal scheduling algorithm. Computing the optimal fetching order 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 logical to physical block numbers, it is virtually impossible. Nevertheless, studies show that fetching in order of logical block numbers tends to work well [WGP94].

The mechanisms we have described so far work well in the single-process case. We now consider what to do when multiple processes are running at the same time.

3 Integration for Multi-Process Case

The situation is more complicated in the multi-process case than in the single-process case. Although the future file references of each process may be known, the interaction between file caching, prefetching, and CPU scheduling is complex, so we cannot predict how the processes' reference streams will be interleaved. Therefore, even if we have complete knowledge of each process's future accesses, we only have partial knowledge of the global access stream seen by the file cache. We address this problem by localizing prefetching and replacement decisions to each process, and relying on the kernel for allocating cache blocks to processes.

3.1 Two-Level Cache Management

We use a *two-level* cache management strategy. The kernel decides how many cache blocks each process may use. Each process decides how to use its own blocks for caching and prefetching.

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.

The choice of per-process local management policies is relatively simple. Processes that are unable or unwilling to manage their own cache blocks can let the kernel manage their cache space, and the kernel will use its default policy, for example, LRU replacement with one-block lookahead prefetching. Processes that have knowledge of their future file accesses can use the *controlled-aggressive* policy that integrates caching, prefetching and disk scheduling on a single-process basis, as described above. In other words, each process examines its cache contents and decides to prefetch or not based on its own future accesses following the *controlled-aggressive* algorithm, while attempting to make optimal replacements.

The choice of the kernel's global allocation policy is the key issue to be addressed. We would like a policy that allocates cache blocks to the processes that need them, but treats all processes fairly. In particular, processes that manage their own cache well should benefit from this scheme, and processes that do not want to manage their own cache blocks should not suffer.

3.2 The LRU-SP Allocation Policy

Ideally, the kernel's allocation policy should satisfy the following criteria:

- *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 itself) to suffer more misses than it would suffer under global LRU.*

The LRU-SP (LRU with Swapping and Placeholders) allocation policy is designed based on the above principles. It can be proven that, in a simplified theoretical model that describes the case of application-controlled caching without prefetching and disk scheduling, LRU-SP satisfies all three criteria [Cao96].

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). 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. If the process manages its own cache, then it can decide which of its blocks to evict at this time; otherwise, the block at the "least recently used" end of the list is replaced. (Blocks belonging to dead processes, or to more than one running process, are considered to have no owner; they are kept around until they reach the end of the system-wide LRU list, at which point they are discarded, as they would have been under the ordinary LRU policy.)

Two modifications need to be made to the "LRU" list:

- If block A is at the end of "LRU" list and the user process chooses to replace block B instead, the kernel swaps the positions of A and B in the LRU list ("swapping"), then builds a record (a "placeholder") for B, pointing to A, to remember the process's choice. (See Figure 2.)
- If a user process misses on a block B, and a place-holder for B exists, then the block pointed to by that placeholder is replaced. Otherwise, the process that owns the block at the end of the LRU list is chosen to give up a block.

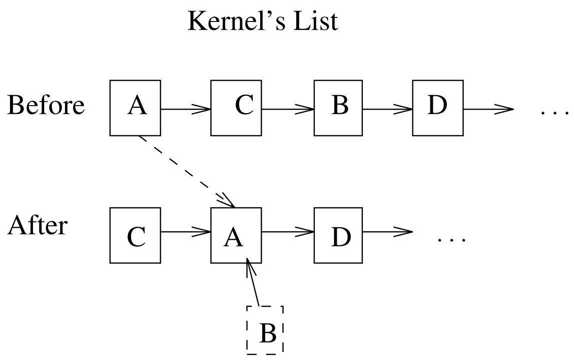

Figure 2: How LRU-SP keeps the "LRU" list. Here block A is at the "least recently used" end of the list. The process that owns A decides to replace block B. The figure shows the list before and after the replacement decision.

In addition, the rules for place-holder bookkeeping are as follows: if a user process hits on block A, then any placeholder pointing to A is deleted; if block C is replaced while block A is at the end of the "LRU" list, then after A is put in C's position in the list, any placeholder pointing to C is changed to point to A.

The reason swapping and placeholders are necessary is the following. If the positions of A and B are not swapped, then A will remain at the LRU end of the list, and the process that owns A will be repeatedly asked to give up a block, until it either replaces A or accesses A again. Thus, without swapping, a process that uses policies different from LRU would be penalized. Similarly, placeholders keep track of the differences between the replacement choices made by an application's policy and those that would have been made by the default LRU policy. Placeholders allow the system to detect when the application's choice is not as good as that of the default policy (e.g. B is accessed before A); in this case, the erroneous process will give up a block for the extra cache miss, instead of other processes giving up a block. (More details can be found in [CFL94a, Cao96].)

Early experiments with application-controlled file caching showed that LRU-SP performs quite well, and the swapping and placeholders mechanisms are crucial to performance improvement in multi-process environments [CFL94b]. For example, if the allocation scheme does not use swapping but simply chooses victim processes based on strict LRU order, it not only performs worse than LRU-SP most of the time, but also penalizes some smart applications to the extent that they are better off not using good replacement policies. Similarly, if placeholders are not built, a foolish process can hurt other processes by taking away cache blocks from them. By using both swapping and placeholders, LRU-SP fairly distributes cache blocks and offers protection against foolish applications.

In addition, LRU-SP contains the effect of false prefetching by always comparing the specified reference list with the actual file accesses. In LRU-SP, prefetched blocks are added to the "most-recently-used" end of the "LRU" list, because they are predicted to be used soon. However, if the prediction is not accurate, some prefetched blocks may not be used at all, wasting the cache space that could have been used for more valuable blocks. To detect such a situation, every time the application accesses file data, the prefetcher tries to match the specified reference list against the actual file accesses it has seen so far (there are many algorithms to compute the match; section 4 describes one). If the prefetcher finds some file accesses that were predicted but did not happen, it checks whether the corresponding blocks will be used again, and if not, flushes them out of the cache. Thus, a false prefetch occupies a cache block until the predicted access to it passes.

3.3 Summary

We now have a complete strategy for file cache management integrating application-controlled caching, prefetching, and disk scheduling. We use 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 application-controlled file caching and prefetching, integrated by the *controlled-aggressive* policy. Each process improves its access efficiency by submitting its prefetches in batches, which are scheduled by the disk driver to reduce disk access latency.

4 ACFS: A Prototype Implementation

We have implemented a prototype file system, called "Application-Controlled File System" (ACFS), that integrates application-controlled caching with prefetching and disk scheduling. ACFS is implemented by modifying the Ultrix 4.3 file system codes. Below we first describe the application programming interfaces, then present the details of our implementation.

4.1 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.

4.1.1 Application Control of Cache Replacement

There are a variety of ways to implement the interaction between applications and the 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 scheme, for example, "upcalls", can give applications complete control over cache replacement, but can incur high overhead [ABLL92, MA90]. 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 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. Interprocess allocation decisions are made using LRU-SP, as explained in the previous section.

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`. This change affects only those blocks that are presently in the cache, and a block's priority change only lasts until the next time the block is either referenced or replaced.

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 general, the appropriate use of this interface allows applications to implement a large variety of replacement policies with 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.

In addition, `set_temppri` allows applications to control file cache replacement at the block level. Applications can raise or lower a block's priority at any time, to keep it in the cache, or to flush it out of the cache. Furthermore, 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.

4.1.2 Application Specification of Prefetching

Due to similar considerations of implementation cost versus available flexibility, we choose to implement prefetching in the kernel, and let application processes provide the kernel with information about future accesses. The kernel implements an interface that allows applications to specify file accesses. The interface is deliberately simple so that compilers and libraries can use it to communicate to the kernel while providing automated prediction or better interfaces to applications.

An application's future 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. There are two calls for this purpose: `FileListStart(size)` and `FileListEnd()`, 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 then calls `FileListEnd`. This tells the kernel which files the application will access. Use of this interface requires minimal modification of applications.

The alternative specification, a list of blocks, allows applications to specify future accesses at the block level. The call provided for this purpose is `BlockList(list, size)`, where `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.

4.2 Implementation Details

We implemented ACFS by replacing the file buffer cache module of the Ultrix 4.3 operating system with three modules, as shown in Figure 3: a buffer cache module (BUF), an application control module (ACM) and a prefetch control module (PCM).

BUF handles the file accesses to the buffer cache, does bookkeeping, and implements the cache block allocation policy. ACM implements the application-controlled caching interface calls and acts as a proxy for the user-level managers. The PCM module implements the application prefetching interface and issues prefetch requests. It interfaces with the BUF, ACM and disk driver modules to optimize prefetching performance.

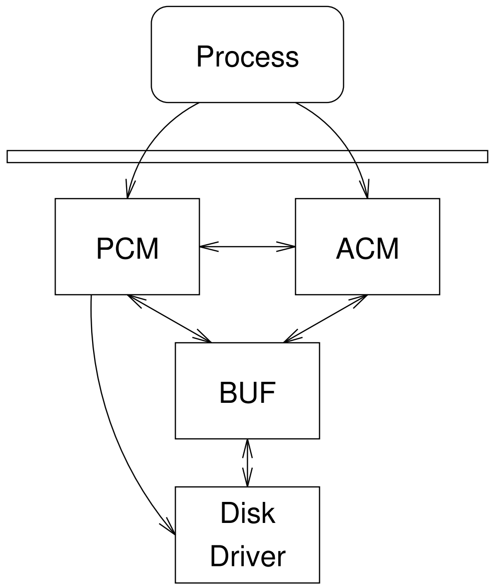

Figure 3: Structure of our implementation. Cache management is handled by three modules: BUF manages the allocation of blocks to processes, ACM implements the replacement policy on behalf of each application, and PCM manages prefetching.

4.2.1 Application-Controlled File Caching

The main modules supporting application-controlled file caching are BUF and ACM, which sit below the VFS interface and communicate via procedure calls.

When a block must be replaced, BUF picks the block at the end of its global "LRU" list as a candidate block and asks ACM which block to replace. ACM then checks whether the corresponding process manages its file cache. If it does not, the candidate block is replaced; otherwise, ACM acts as a proxy for the user-level manager and makes the decision based on the priorities and policies specified by the user-level process.

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 the file cache;
- `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. Experiments show that the implementation adds negligible overhead to file accesses.

4.3 Integrated Prefetching

The PCM module provides an interface between application processes and the BUF module. It keeps track of the file access streams specified by the applications, and issues prefetches in accordance with the *controlled-aggressive* algorithm.

To accomplish these tasks, the PCM module builds and maintains a prefetch object for each application that is prefetching. In general, in order to prefetch using the *controlled-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 to 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, as discussed in Section 2.1.3. to keep the `cursor` at the correct position with high probability in the face of possible inaccuracies in the access list. In the implementation, we used a very simple algorithm that looks ahead at some small constant number (two in the current implementation) of references in `accessList` and advances only if there is a match.

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 a scheduling module, and advances nextHole, changing freeBlocks if necessary.

Ideally, the batch of prefetch requests should be sent directly to the disk driver, which would schedule their services together with other disk requests in its queue. Unfortunately, the disk driver in Ultrix 4.3 does not schedule requests but rather services them first-come-first-served. To avoid changing the driver, our compromise is to schedule the requests in PCM first before sending them to the driver. This compromise generally performs poorer than a good scheduling strategy implemented in the disk driver.

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 rare. Our 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 when a process issues a prefetch and is asked to replace one of its blocks, it has 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 2.1.3. 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 all of the blocks that are referenced after the next missing block (nextHole), and then choosing among these blocks using the priorities and policies specified by the application. This scheme is an approximation to the algorithm described in Section 2.1.3 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 2.1.3 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.

5 Performance Measurements

We have compared the performance of six file system implementations incorporating various combinations of application-controlled caching, prefetching and disk scheduling. The performance of both single applications and multi-application mixes from a suite of I/O intensive applications was measured.

5.1 Applications

We selected four applications for our measurements: **CScope**, **Dinero**, **Glimpse**, and **Postgres**. All of these programs are I/O intensive (see Figure 4 and Figure 5 for a breakdown of elapsed time in terms of I/O and CPU times), and they have different file access patterns. All applications can predict their file accesses in the next phase of computation, though the predictions are not accurate (mostly missing file accesses). We begin by describing the applications, their file access patterns, and the application-controlled caching and prefetching strategies used for each.

CScope [cs1-3]

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: symbol-oriented queries and egrep-like text search. We used cscope on two operating system kernel sources of about 18 Mbytes and 10 Mbytes, respectively. We did three runs:

- **cs1**: searching for 8 symbols in the 18MB source,
- **cs2**: text search for 4 patterns in the 18MB source, and
- **cs3**: text search for 4 patterns in the 10MB source.

Symbol-oriented queries always read the database file "cscope.out" sequentially to search for records containing the requested symbols. Our cache management policy is to prefetch "cscope.out" in sequential order, and to use MRU replacement.

Text searches in CScope sequentially read all source files in an order specified in "cscope.files". Our policy for these searches is to use MRU replacement, and to prefetch files in their order of access. Only a few lines of code were needed to issue these directives to the kernel.

Dinero [din]

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. We 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, with set associativity ranging from 1 to 4.

Dinero reads the trace file sequentially for each simulation. Thus, the caching strategy is MRU on the trace file. For prefetching, we simply pass the trace file name to the kernel.

Glimpse [gli]

Glimpse is a text information retrieval system [MW94]. It builds approximate indices for words to allow relatively fast search with small index files. We took a snapshot of news articles in several comp.* newsgroups on May 22, 1994, consisting of about 40 MBytes of text in all. We built indices with the Glimpse tool, resulting in about 2 MBytes of indices. We then did searches for lines containing these keywords: scheduling, scheduling and disk, cluster, rendering and volume, and DTM.

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. Index files are more frequently accessed than data files. Thus, the caching strategy is to have two cache replacement priority levels: index files are at priority 1 and data files are at priority 0. Both priority levels use the MRU replacement policy.

Glimpse uses "agrep" (a variant of "grep") 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.

Postgres [psel1-2]

Postgres is a relational database system from the University of California at Berkeley [Pos93]. Postgres 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.

We used a 200,000 tuple relation from a scaled-up Wisconsin benchmark [Gra91]. There is an index on attribute `unique1`, which is uniquely random in the range 1-200,000. We used two selection queries to see whether our integrated prefetching approach can benefit database applications:

- **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 execution of these two 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**), and then start the search there until 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 is rather straightforward: after searching the indices, we know the list of tuple blocks that will be read. We then simply pass this list to the kernel using the `BlockList` interface. For caching, we determine 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, we implemented a very simple preprocessor to do the job and to invoke Postgres queries. We believe, however, that modifying Postgres as we did with the other applications would be easy.

5.2 File System Configurations

In order to see the performance advantages of the proposed techniques over traditional file system designs, we experimented with six file system implementations, including a traditional file system, on the DEC 5000/240 workstation. We used file cache sizes of 6.4MB (the default size under Ultrix) and 12MB. The implementations are:

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

2. *Global LRU + prefetching*[2]: 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 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 *controlled-aggressive* cache management policy;

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

[2]For those who are familiar with our early paper [CFKL95], *Global LRU + prefetching* is roughly the equivalent of the *LRU-sensible* algorithm.

Our 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 us to use both disks in some of the multi-process experiments.

5.3 Single Process Performance

Figures 4 and 5 show the performance of our seven programs under the six file system configurations. All data points are the average of three runs; variances are mostly less than 2%, and all less than 3.5%.

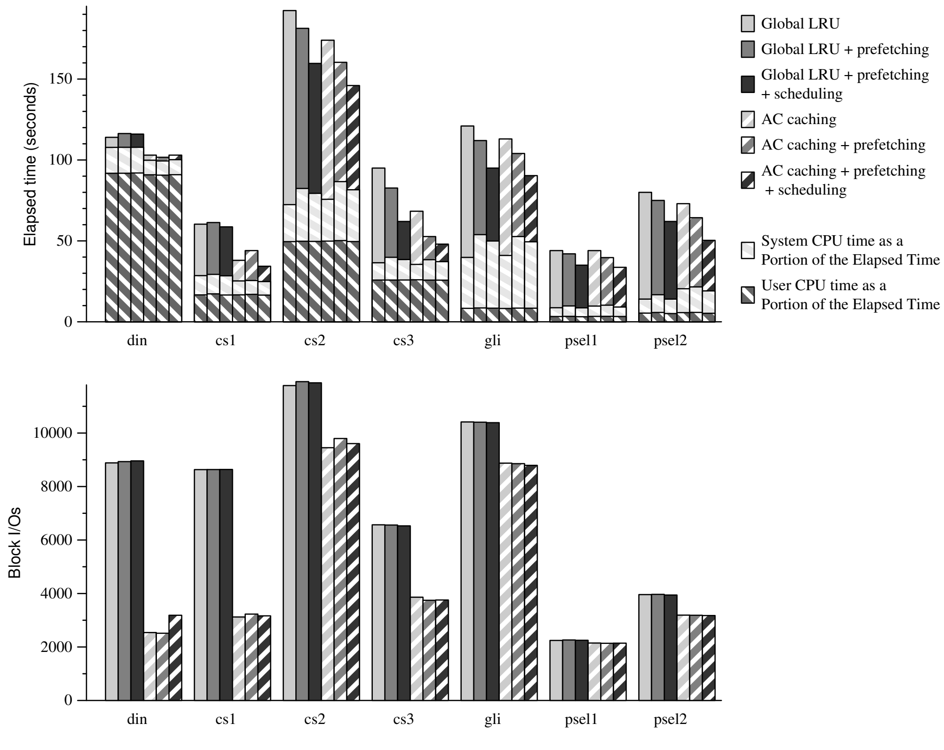

Figure 4: Single application running time, and number of block I/Os, with a 6.4MB file cache.

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

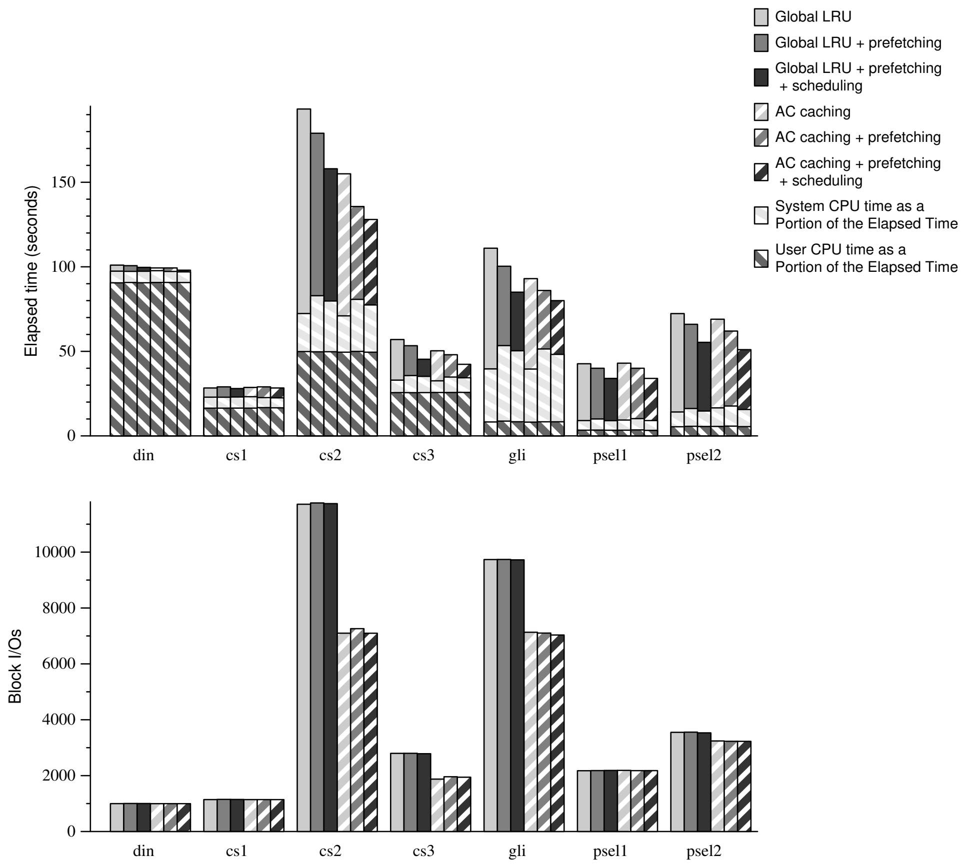

Figure 5: Single application running time, and number of block I/Os, with a 12MB file cache.

our fully integrated system, with all three techniques in use, clearly show the best performance.

To understand the source of the performance gains, note that the running time of an application is the CPU time (including both user and system CPU time) plus the I/O time (which is the number of disk I/Os times the average disk access latency), minus the portion of the I/O time that is overlapped with the CPU time. The top figures in Figure 4 and Figure 5 also show the decomposition of the running times. The various shaded portions of each bar, from bottom to top, shows the the user CPU time, the system CPU time, and the portion of I/O time that is *not* overlapped with the CPU times. The figures show that: application-controlled file caching improves performance mainly through reducing the number of disk I/Os, disk scheduling improves performance mostly by reducing the average disk access latency, and prefetching improves performance by increasing the portion of the I/O time that overlaps with the CPU time.

Different techniques contribute differently to the performance improvement in each application. For example, `din` and `cs1` mostly access large files sequentially, thus, the one-block-lookahead prefetching employed by traditional file systems works quite well for them. This is particularly evident for `din`: its portion of non-overlapped I/O time is quite small. In addition, since sequential accesses already have excellent locality, disk scheduling would not reduce the disk access latency either. Thus, for `din` and `cs1`, the main source of running-time reduction is application-controlled caching, which reduces the number of cache misses, thus reducing the I/O time that is not overlapped with computation. However, the reduction in the number of block I/Os does not lead to proportional reduction in the elapsed time, because of the CPU times.

The remaining applications, `cs2, cs3, gli, psel1`, and `psel2`, 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. Since they 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 sum of the "bonuses" due to the individual techniques.

The behavior of `cs1` under *AC caching + prefetching* is surprising. When compared to `cs1`'s performance under *AC caching*, 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.

As can be seen from Figure 4 and Figure 5, the main factor influencing the number of I/Os is whether or not application-controlled caching is being used. Adding prefetching and disk scheduling under the *controlled-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 our simulation studies in [CFKL95].

We chose the batch size B in ACFS through experiments. We measured the elapsed times of several applications (`cs2`, `cs3`, `gli` and `psel2`) in the fully integrated system, varying B. Figure 6 shows the elapsed times of these applications normalized to those when $B = 1$ (i.e. no disk scheduling). From the time-stamped execution traces of the application `cs3`, we also extracted 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 this particular application. The

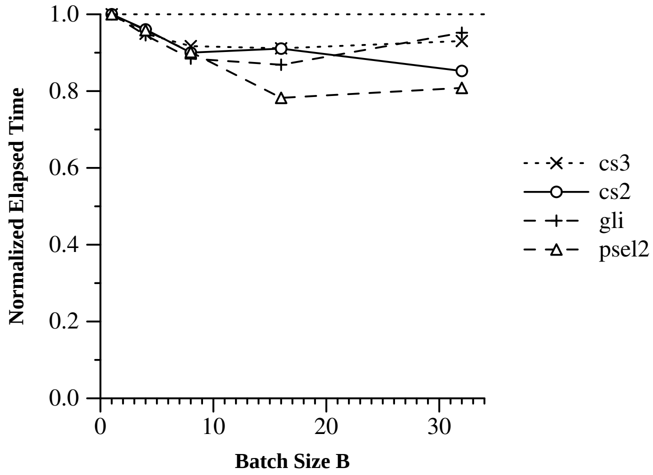

Figure 6: Normalized elapsed times of applications as a function of batch size B (the elapsed time when $B = 1$ is 1.0).

results show that the average disk access latency decreases as the batch size increases, but the reduction need not reduce the elapsed time — the reordering of requests and resultant dis-overlapping of CPU computation and disk access can increase the elapsed time. From these experiments we chose $B = 16$, since it works well for most cases.

On the other hand, we have not done enough experiments to thoroughly understand the effect of batch size on the elapsed time. The optimal batch size depends on both the scheduling algorithm and the characteristics of the reference stream. It is not even clear what the optimal disk scheduling algorithm is in this context. In addition, the behavior of `cs1` under *AC caching + prefetching* as described above, and our experiments on batch size, clearly showed that the non-uniform access latency of disks has a significant impact on applications' performance, and the simple "controlled-aggressive" algorithm derived from the uniform-access-latency assumption is no longer optimal in real systems. We plan to incorporate the non-uniform access latency in our theoretical model and study the issues.

Finally, Figure 4 and Figure 5 show that 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. As microprocessor performance continues to improve dramatically, the additional CPU overhead for these techniques will be less of a concern.

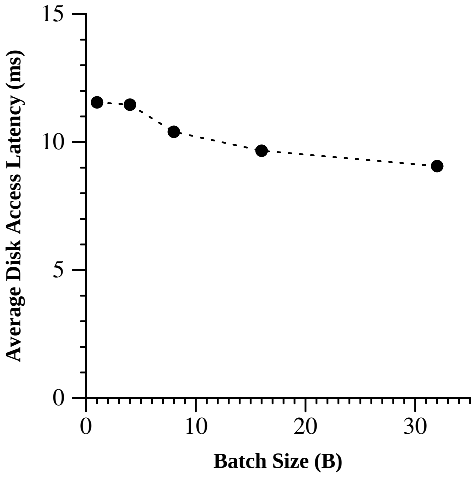

Figure 7: Average disk access latency (duration from when request is issued to the disk until it is finished) as the batch size increases, for `cs3`.

5.4 Multiple Process Performance

We also measured the performance of our file system implementations for multi-process workloads. We ran several pairs of applications[3], 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 done with one disk; the `cs1+psel2` and `cs3+psel1` runs were done with two disks. We also ran our applications concurrently with a copy of the Andrew file system benchmark [HKM+88]: `din+andrew`, `cs2+andrew`, `gli+andrew` and `psel2+andrew`. The Andrew benchmark runs on the RZ26 disk, so only `psel2+andrew` involves both disks.

The results are summarized in figures 8 and 9. 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); 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 our techniques work well in the multi-process case. Again, application-controlled caching, prefetching, and disk scheduling each provides a separate benefit, and combining all three techniques provides the best performance of all.

[3] We actually ran more combinations, but the results are similar to the ones reported here. More detailed information is available at ftp.cs.princeton.edu:/pub/people/pc/.

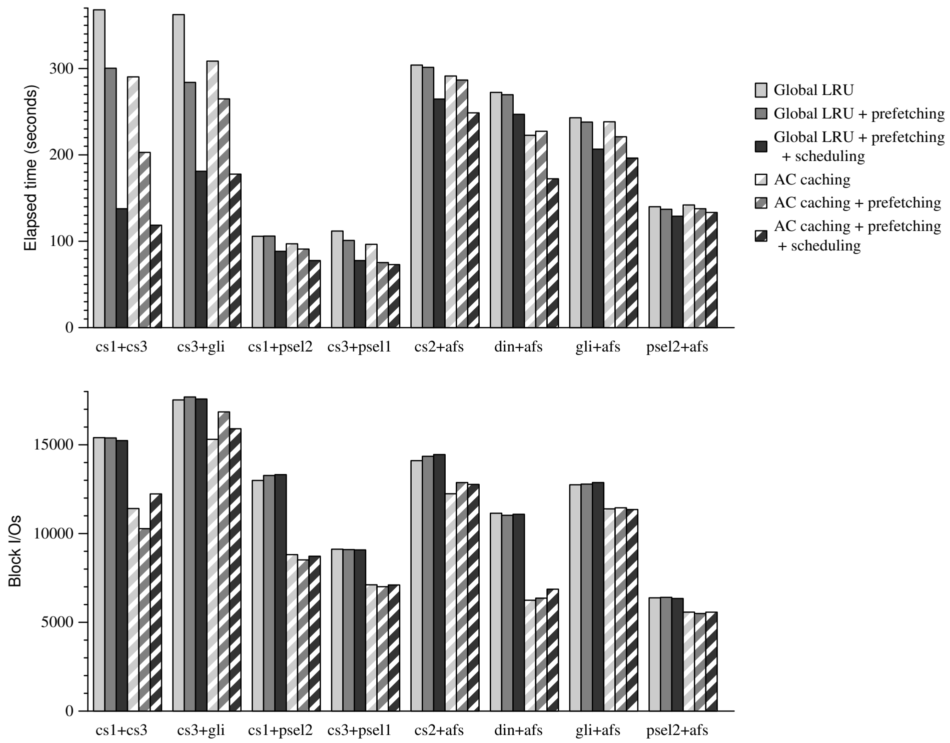

Figure 8: Multiple application running time, and number of block I/Os, with a 6.4MB file cache.

In the single-disk experiments, we see an interesting effect: disk scheduling seems to be particularly important for performance. Detailed examination of execution traces allow us to explain this effect. To understand the effect, 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 also interleaved on a very 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 of the reference streams will destroy this 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.

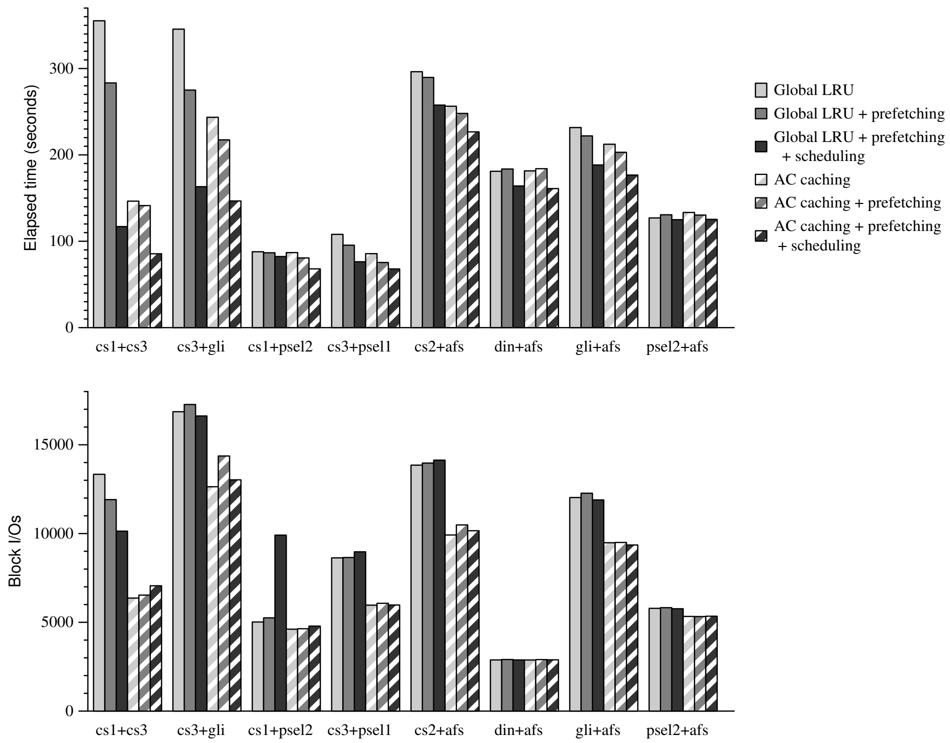

Figure 9: Multiple application running time, and number of block I/Os, with a 12MB file cache.

Prefetching without disk scheduling also has a similar effect, though it is not nearly as large. Since prefetching may bring a block into the cache before the application accesses the block, the access which would have been a cache miss now hits in the cache. Since a context switch happens every time an application misses in the file cache, prefetching reduces the number of context switches by reducing the number of misses. Fewer context switches means that each process can, on average, do more disk accesses before giving up the CPU. Thus, the locality of disk accesses is improved.

Looking at the number of disk I/Os, we see that prefetching and disk scheduling have little impact except for the case of `cs1+psel2` under 12MB file cache. We fed the traces of `cs1+psel2` to our calibrated simulator (described in [CFKL95]), 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

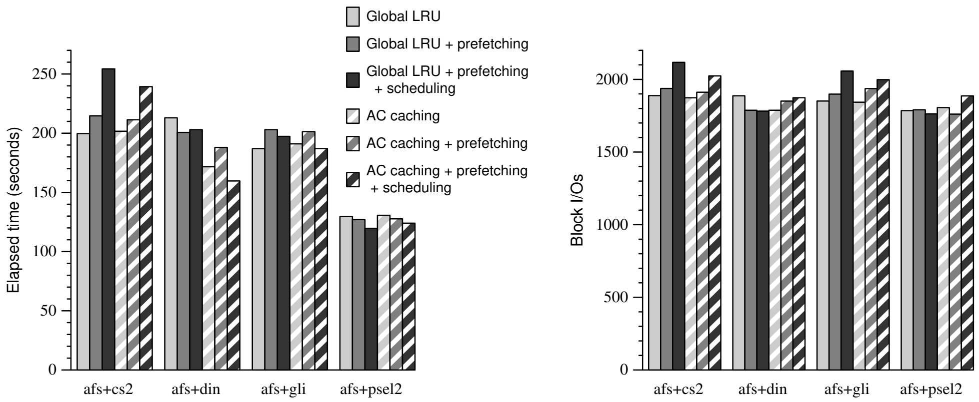

Figure 10: Elapsed time and number of block I/Os of Andrew benchmark, when running concurrently with our applications. File cache size is 6.4MB.

(no cache miss if it fits, otherwise miss on every access). Although the increase in the number of disk I/Os did not affect the elapsed time in this case, it does show that future studies must consider potential changes in the interleaving of process execution by prefetching and disk scheduling. Note that the problem goes away when cs1 uses a better replacement policy (MRU).

We are also interested in how application-controlled caching and prefetching might affect oblivious applications (i.e. those not using these techniques). Since the Andrew benchmark is oblivious, we examine its running time (figures 10 and 11). In general, there are several factors involved. Application-controlled caching tends to reduce the elapsed time of oblivious applications because it reduces the number of disk I/Os done by smart applications. On the other hand, prefetching with scheduling can either reduce the elapsed times of oblivious applications because it reduces the average I/O cost, or increase them because it issues a batch (sixteen) of prefetch requests at a time, making oblivious applications' I/O requests wait longer in the queue. Applying all three techniques together produced 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`). Notice, however, that these techniques always improve the total elapsed time required for both applications to finish (figures 8 and 9).

Finally, we notice that some runs have high variances. We are not sure why this happens. Our 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. Still, we observe that even with the variances our techniques provide substantial performance benefits.

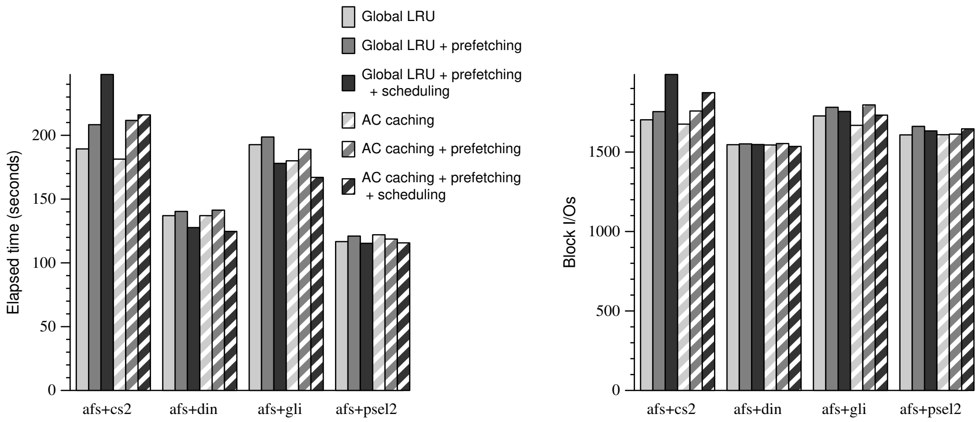

Figure 11: Andrew benchmark's elapsed time and number of block I/Os when running concurrently with our application, under 12MB file cache.

6 Related Work

The work described in this paper builds on our previous work. In [CFL94a] we introduced the LRU-SP kernel allocation policy and simulated its performance on applications. In [CFL94b] we implemented LRU-SP without prefetching or disk scheduling, and showed by experiment that it works well in practice, confirming the simulation results of [CFL94a]. In [CFKL95] we presented theoretical results on policies for integrating prefetching with application-controlled caching. We introduced the *controlled-aggressive* policy, proved that it is theoretically close to optimal, and verified its good performance by simulation.

This paper improves on our previous work in several ways. It presents the issues, our design and implementation of integrating application-controlled caching, prefetching and disk scheduling in the multi-process environment. Using real applications it shows that the techniques together generate considerable synergy and provide significant improvement on file cache performance.

Recently there have been a number of research projects on prefetching in file systems. Patterson and Gibson's informed prefetching and caching [PG94, PGG+95] is similar to our study. However, there are a number of major differences. Instead of providing a mechanism for applications to control file cache replacement, informed prefetching and caching tries to "infer" good replacement decisions from the reference list passed from the application to the kernel. Thus, access patterns such as hot and cold blocks cannot be easily expressed in this framework. In addition, instead of using two-level cache management and LRU-SP, informed caching and prefetching handles cache allocation using a cost-benefit estimate. At this time it is not clear which allocation approach is better for prefetching. Also, it is not clear how the "cost-benefit" approach would handle application-controlled file caching. Finally, our algorithms to integrate caching and prefetching are different: ACFS uses "controlled-aggressive", and the informed caching and prefetching approach uses a method called "prefetch-horizon." The two algorithms are designed for different environments: "controlled-aggressive" is designed for systems with a small number of disks and limited I/O bandwidth, while "prefetch-horizon" is designed mainly for parallel disk arrays with abundant I/O bandwidth. The comparison of the two algorithms remains future work. Despite these differences, both projects utilize disk scheduling, employ similar application interfaces for prefetching, and arrive at similar conclusions about the benefits of prefetching and disk scheduling.

There have also been many studies focusing on how to predict future accesses from past accesses [TD90, PZ91, CKV93, GA94]. In particular, Griffioen and Appleton's work [GA94] tries to predict future file accesses based on past accesses using "probability graphs," and prefetches accordingly. Few of these studies, however, considered the interaction between prefetching and caching or investigated the combined cache management problem.

Several detailed studies of disk scheduling have been done [SCO90, JW91, WGP94]. These studies typically considered a wide variety of scheduling policies under timesharing Unix workloads. However, the policies are for requests queued from multiple processes, not for requests specified in a prefetch-list from a single process. Thus, they are not concerned with the effect of reordering on the overlapping of computation and disk accesses. As we discussed in section 2.2 and section 5.3, optimal disk scheduling in the prefetching context is a complicated problem, and is quite different from disk scheduling in the timesharing context. We plan to continue working on this problem.

Several recent research projects have tried to improve file system performance in a number of other ways, including log-structured file systems [RO91], disk block clustering [MK91, SS95], and delayed writeback [Mog94]. Most of these papers still assume global LRU as the basic cache replacement policies and sequential one-block lookahead or large I/O units as the primary prefetching techniques. They do not address how to integrate prefetching and disk scheduling with application-controlled file caching.

On the other hand, our work is complementary to these approaches. The techniques used in ACFS can be incorporated in a log-structured file system to improve further the performance of the file system. Disk block clustering can be combined with our techniques: for applications that do not want to manage their caches, the default policy can be LRU replacement with disk-block-cluster prefetching, instead of one-block-lookahead prefetching. Our techniques can be combined with any delayed writeback policy, since the policy only decides when to write back a dirty block to disk (a dirty block is always written back to disk when it is replaced). On the other hand, choosing the best writeback algorithm in the context of application-controlled file caching and prefetching is an interesting problem that has not been solved yet.

There are numerous studies on prefetching in parallel I/O systems [EK89, WYT93]. Although our work focuses on prefetching with a single disk or server, the "Do No Harm" and "First Opportunity" principles apply to prefetching algorithms in the parallel context as well. We believe these principles are important to avoid the thrashing problem [WYT93].

The database community has long studied access patterns and buffer replacement [Sto81, CD85, OOW93] and prefetching [CKV93, PZ91] policies. However, most of these studies focus on buffer management and prefetching in database storage management systems rather than in file systems, and they do not address the integration of caching, prefetching and disk scheduling in multi-process environments.

Finally, prefetching in uni-processor and multi-processor computer architectures [RL92, CB92, CKP91, Smi78, TE93] 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 feasible to spend more than a few machine cycles on each prefetching decision. File systems, on the other hand, 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 [TE93] showed that thrashing is a problem when prefetching in bus-based multiprocessor caches, suggesting that the "Do No Harm" rule applies in those systems as well.

7 Conclusions

We have presented the design, implementation and performance of a file system incorporating integrated application-controlled file caching, prefetching and disk scheduling.

Our experimental results show that careful integration of these techniques greatly improves the 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%). Not only does each technique provide significant performance benefit, but they complement each other nicely by targeting different areas of the I/O bottleneck, and the resulting integrated system provides the best performance.

Our study shows that *controlled-aggressive* not only performs close to optimal in the theoretical model, but also performs well in practice. We also find that LRU-SP performs well in the presence of prefetching and disk scheduling. On the other hand, our experiments showed that prefetching and disk scheduling may change the interleaving of process executions significantly, and this should be taken into account when studying resource allocation.

What we did not foresee in our previous studies is the significant performance gains from combining disk scheduling with batch prefetching. This helps not only applications with non-sequential disk accesses, but also multi-process workloads. The batching of prefetch requests and the increase in time between context switches allow each process to do more disk accesses before relinquishing the CPU and to benefit from locality on the disk. It also leads to an interesting theoretical problem: what is the best algorithm for prefetching in the presence of nonuniform fetch time, for example, fetching from a real disk.

An issue that was not addressed in our study is the effect of disk scheduling on process scheduling. Ideally, disk scheduling algorithms should differentiate demand requests, which are synchronous, from prefetch requests, which are asynchronous, for fairness concerns. However, the interactions can be quite complex and we do not yet fully understand the issues involved.

Many other questions remain. We do not yet understand the best approaches to handle various inaccuracies in an application's prediction of future file accesses, including conflicts with application-controlled file caching policies. We need to evaluate the performance of our system on real-life workloads. It would also be useful to extend our results on prefetching to a model that incorporates multiple parallel disks. Finally, we need to understand the interaction between prefetching, cache management and CPU scheduling, in order to find an allocation strategy that provides both performance and fairness guarantees.

References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Cao96] Pei Cao. *Application-Controlled File Caching and Prefetching*. PhD thesis, Princeton University, January 1996. Also published as technical report CS TR-522-96.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.

[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 Ver y 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.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of 4th 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 Conf. on Management of Data*, pages 257–266, May 1993.

[EK89] Carla Schlatter Ellis and David Kotz. Prefetching in file systems for MIMD multiprocessors. In *Proc. 1989 Intl. Conf. on Parallel Processing*, pages 306–314, August 1989.

[GA94] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proc. of USENIX Summer 1994 Technical Conference*, pages 197–208, June 1994.

[Gra91] Jim Gray. *The Benchmark Handbook*. Morgan Kaufman, 1991.

[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+88] 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.

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

[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.

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proc. of 1991 Winter USENIX Symposium*, pages 33–43, 1991.

[Mog94] Jeffrey C. Mogul. A better update policy. In *Proc. of 1994 Summer USENIX Technical Conference*, pages 99–111, June 1994.

[MW94] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of USENIX Winter 1994 Technical Conference*, pages 23–32, January 1994.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffer ing. In *ACM SIGMOD Conference on the Management of Data*, pages 297–306, May 1993.

[PG94] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency witn informed prefetching. In *Proc. Third Intl. 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 *Proceedings of 15th ACM Symposium on Operating Systems Principles*, December 1995.

[Pos93] Postgres Group. POSTGRES version 4.1 release notes. Technical report, Electronics Research Lab, University of California, Berkeley, February 1993.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proc. of 17th Intl. Conf. on Very Large Data Bases*, pages 255–264, September 1991.

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *Proceedings of 5th 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.

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

[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 1991 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.

[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 Dept., Columbia University, 1990.

[TE93] Dean M. Tullsen and Susan J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of 20th International Symposium on Computer Architecture*, pages 278–288, May 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.

[WYT93] Kun-Ling Wu, Philip S. Yu, and James Z. Teng. Performance comparison of thrashing control policies for concurrent mergesorts with parallel prefetching. In *Proc. of 1993 ACM SIGMETRICS*, pages 171–182, May 1993.