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

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

## Abstract

Although file caching and prefetching are known techniques to improve the performance of file systems, little work has been done on intergrating caching and prefetching. Optimal prefetching is nontrivial because prefetching may require early cache block replacements. Moreover, the tradeoff between the latency-hiding benefits of prefetching and the increase in the number of fetches required must be considered.

This paper presents the design and implementation 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 policy [CFL94a] to allocate blocks to processes, and each process uses the *controlled-aggressive* policy, an algorithm previously shown in a theoretical sense to be near-optimal, for managing its cache. Each process then improves its disk access latency by submitting its prefetches in batches and schedules the requests in each batch to optimize disk access performance. Our measurements show that this combination of techniques greatly improves the performance of the file system. Average running time is reduced by 26% for single-process workloads, and by 46% for multi-process workloads.

## 1 Introduction

Caching and prefetching are two effective techniques for improving the performance of file systems. Although they have been studied extensively in the past, file caching has been studied in the absence of prefetching, and most studies on prefetching have been conducted in the absence of caching or for a fixed caching strategy. The integration of caching and prefetching has not been 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 hurt performance because new and possibly better replacement opportunities open up as the program proceeds.

In a recent paper, we studied application-controlled file caching and prefetching from a theoretical point of view [CFKL95]. We presented four rules that any optimal integrated algorithm must follow and then showed that a simple algorithm derived using these rules has performance that is provably close to optimal. Although our trace-driven simulation results validated the theoretical analysis, it remained unclear whether or not a real implementation would yield the predicted performance gains.

This paper presents the implementation and performance of a file system under the Ultrix operating system that integrates application-controlled caching with prefetching. The integrated prefetching unit takes advantage of knowledge of future application reference patterns for both prefetching and cache block replacement.

Prefetching provides opportunities for disk scheduling to optimize disk access performance. Since prefetch requests can be generated in groups, average disk latency can be improved by reordering disk accesses within a group. Our system implements batch prefetching combined with disk scheduling.

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

*Department of Computer Science, Princeton University, Princeton, NJ.

†Department of Computer Science, University of Washington Seattle, WA.

be taken to ensure that performance is not adversely affected by this uncertainty. Moreover, global allocation of cache space to competing processes should be fair. Our implementation uses a variant of the LRU-SP policy (described in [CFL94a]) in order to address these concerns.

We have measured the file system 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 46%.

## 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 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 that a no-prefetching policy (using the optimal offline algorithm) would hit on the first two references, then miss on the reference to C, discarding B, and finally hit on A. 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.

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; readers should refer to [CFKL95] for more details.

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.

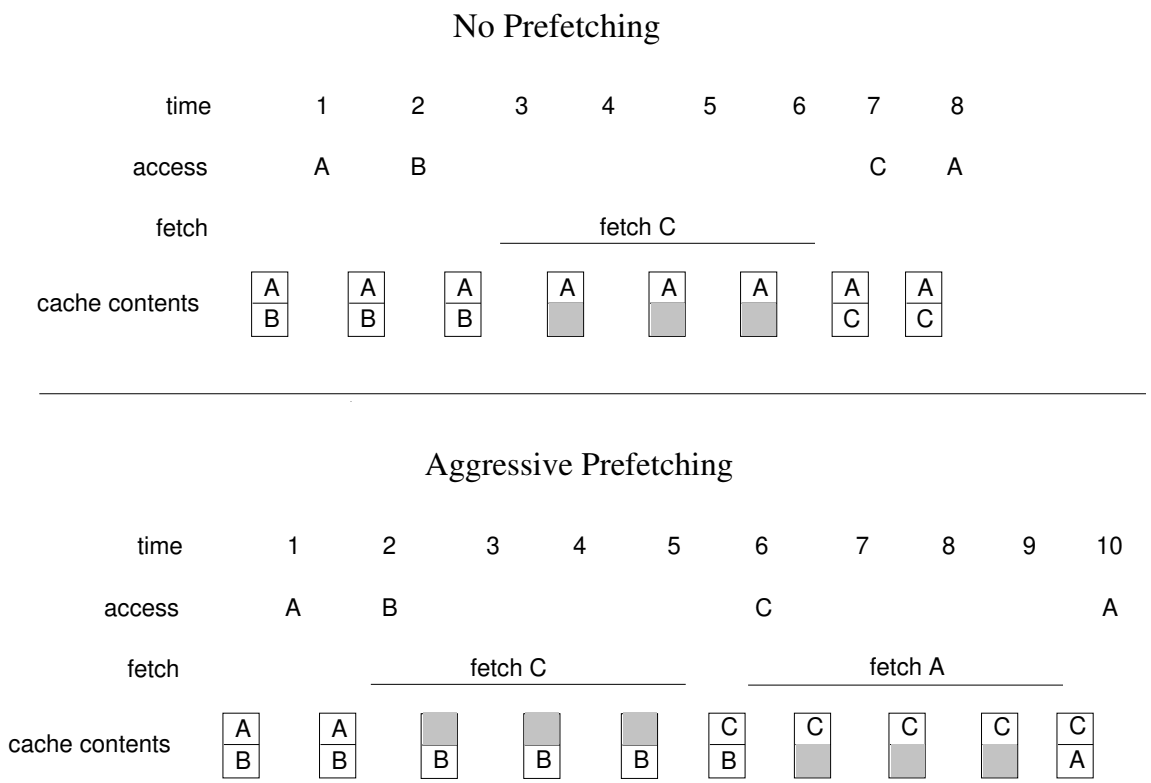


Figure 1: A file access stream for which prefetching hurts performance. Eight time units are required in the absence of prefetching; ten time units are required when prefetching is done.

In [CFKL95], we observed that any optimal prefetching algorithm must follow 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 there is a block in cache that will be accessed after the missing one and the disk is idle. 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

[1]In our previous paper 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.

time. In our case (i.e. prefetching from disk to buffer cache), $F/K$ is typically less than 0.02 [CFKL95], so *controlled-aggressive* is very close to optimal.

Simulations
described in our previous paper [CFKL95] confirm that controlled-aggressive does well for real reference streams.

Although *controlled-aggressive* relies on perfect knowledge of future access, it is not hard to see how to adapt it to the case where only limited-lookahead knowledge is available. The details will be addressed in section 4.

### 2.2 Incorporating Disk Scheduling

The final technique we use is disk scheduling. Because of the physical attributes of disks, careful scheduling of disk accesses can provide significant improvement in performance [SCO90]. 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.

We employ a simple scheduling heuristic: limited batch scheduling. The prefetcher waits until it has $B$ disk requests to issue. These requests are then sorted into order of increasing logical block number, and the fetches are performed in sorted order.

Logical-block-number ordering 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 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].

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 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 logical-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. We found by experience that $B = 16$ worked well for our system.

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 can 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 use LRU replacement without prefetching and disk scheduling (or only one-block-lookahead prefetching as in existing file systems). Processes that have knowledge on their future file accesses can use the *controlled-aggressive* policy that integrates caching, prefetching and disk scheduling on a single-process basis. 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.

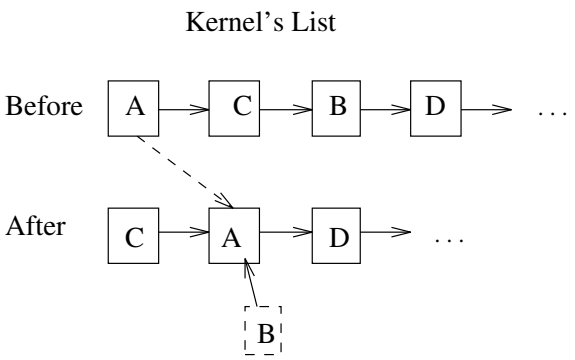


Figure 2: How LRU-SP keeps the "LRU" list. Here block A is at the "least recently used" end of the list. The process who owns A decides to replace block B. The figure shows the list before and after the replacement decision.

### 3.2 The LRU-SP Allocation Policy

In [CFL94a] and [CFL94b] we presented the LRU-SP (LRU with Swapping and Placeholders) policy, which solves this problem for the case of application-controlled caching without prefetching and disk scheduling. LRU-SP can easily be extended to incorporate prefetching and disk scheduling; we describe the extended version here.

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; it is up to the process which of its blocks to evict.

In order to prevent certain bad behaviors (which are described in more detail in [CFL94a]), two modifications to the "LRU" list, swapping and placeholders (figure 2), are used. Swapping means that when a process chooses to give up a block which is not at the end of the "LRU" list, the block at the end of the "LRU" list is moved into the evicted block's position in the "LRU" list. Placeholders are data structures which remember when a process chose to evict a non-"LRU" block; if this choice turns out to be foolish, the placeholder can be used to keep the foolish process from benefiting from its bad choice. Further details of LRU-SP appear in [CFL94a] and [CFL94b].

### 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 the *controlled-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 logical block number.

We now proceed to describe how we implemented this scheme in a real system.

## 4 Implementation

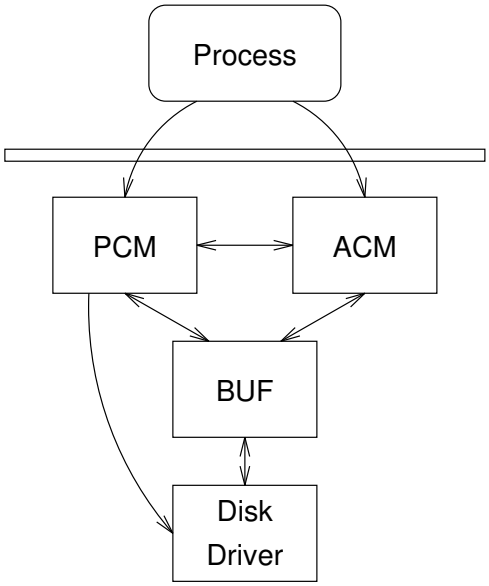


Figure 3: Structure of our implementation. Cache management is handled by three modules: BUF manages the allocation of blocks to processes, ACM implements the replacement policy on behalf of each application, and PCM manages prefetching and disk scheduling.

We have implemented a file system that integrates application-controlled caching with prefetching. The implementation is realized by breaking the existing buffer cache module of the file system in the Ultrix 4.3 operating system into two modules: a buffer cache module (BUF) and an application control module (ACM), and adding a prefetch control module (PCM). The structure is shown in Figure 3.

BUF handles the file accesses to the buffer cache 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. ACM and BUF do roughly the same jobs that they did in our system that implemented only application-controlled file caching [CFL94b]. In the current implementation, ACM and BUF have been modified in order to integrate with prefetching.

The PCM module implements the application prefetching interface and disk scheduling. It interfaces with the BUF, ACM and disk driver modules to optimize prefetching performance.

## Application Interface of PCM

Since prefetching is implemented in the kernel, application processes need to give the kernel information about future accesses. The PCM module implements an interface that allows applications to specify file accesses.

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.

## Implementation Details

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, issues prefetches in accordance with the *controlled-aggressive* algorithm, and performs disk scheduling for batched prefetches to optimize disk access performance.

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. The most important one is the `accessList`, the ordered list of file block numbers to be accessed. To prevent malicious applications from building a list that is too long, the size of `accessList` is limited.

In order to prefetch using the *controlled-aggressive* policy, the PCM must have four pieces of information: where the process is in the access stream, what the next block that has to be fetched is, 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 four variables: the `cursor`, 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`; and `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, our goal becomes to keep the `cursor` at the correct position with high probability in the face of possible inaccuracies in the access list. In our 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.

A new batch of prefetching is triggered 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 the number of `freeBlocks` is still greater than 4, the PCM module asks the BUF module for an empty cache block, queues a prefetch request for the block pointed to by `nextHole` and advances `nextHole`, changing `freeBlocks` if necessary. After queueing all requests issued in this batch, the PCM module reorders the requests according to their disk access locations, and passes these requests on to the disk driver. The disk driver then services these asynchronous requests[2].

[2]The disk driver in Ultrix 4.3 services requests first-come-first-served, therefore we schedule requests in the PCM to avoid changing the 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, *controlled-aggressive* could inadvertently issue a prefetch violating the "Do No Harm" rule. Fortunately, dramatic and rapid fluctuations in global allocation are unlikely. Our implementation causes prefetches to be triggered only if there are at least 4 `freeBlocks`. 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 ACM module is responsible for cache block replacement. Its interaction with BUF and PCM integrates prefetching with application-controlled caching. As described above, when a process misses in the cache, or issues a prefetch, the global allocation procedure of LRU-SP (in the BUF module) is responsible for determining which process must relinquish a block. The ACM module is then responsible for choosing the optimal block belonging to that process to replace. In addition, it notifies the PCM of its replacement choice (so that `freeBlocks` may be changed accordingly).

The motivation for decoupling the prefetching decision from the replacement decision within the implementation is the following. Lookahead within the `accessList` for a process is typically long enough to give information about what to prefetch, but too short to give precise information about optimal replacement. (Recall that the optimal replacement decision is to replace that block whose next reference is furthest in the future.) Replacement decisions can be aided by other general cache replacement primitives that applications provide to the ACM module [CFL94b]. The PCM does, however, attempt to ensure that the ACM does not replace blocks that will be referenced before a block that is currently being prefetched. This information combined with knowledge of a process' `accessList` and general access pattern information, is then used by the ACM to make a near-optimal replacement decision.

# 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 and they have different file access patterns. 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. 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 read source files sequentially. Our policy for these searches is to use MRU replacement, and to prefetch files in their order of access.

### 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" 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. 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.

We use a slightly modified LRU caching strategy. We determine by examining the index which blocks in tuple files will be accessed more than once. These blocks are given higher priority than ordinary tuple blocks during the execution of the selection query. For prefetching, a block list is generated for each index block; these block lists are then passed to the kernel. Instead of modifying 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 other applications is 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. *AC caching*: the baseline system with the addition of two-level cache allocation and application-controlled caching;
3. *Global LRU + prefetching*: the baseline file system with the addition of application-controlled prefetching: applications can tell the kernel what to prefetch and the kernel uses global LRU as the cache replacement policy;
4. *AC caching + prefetching*: an integration of application-controlled caching and prefetching, using the *controlled-aggressive* cache management policy;
5. *Global LRU + prefetching + scheduling*: the same as configuration 3, with disk scheduling added;
6. *AC caching + prefetching + scheduling*: the fully-integrated system which incorporates all of the techniques.

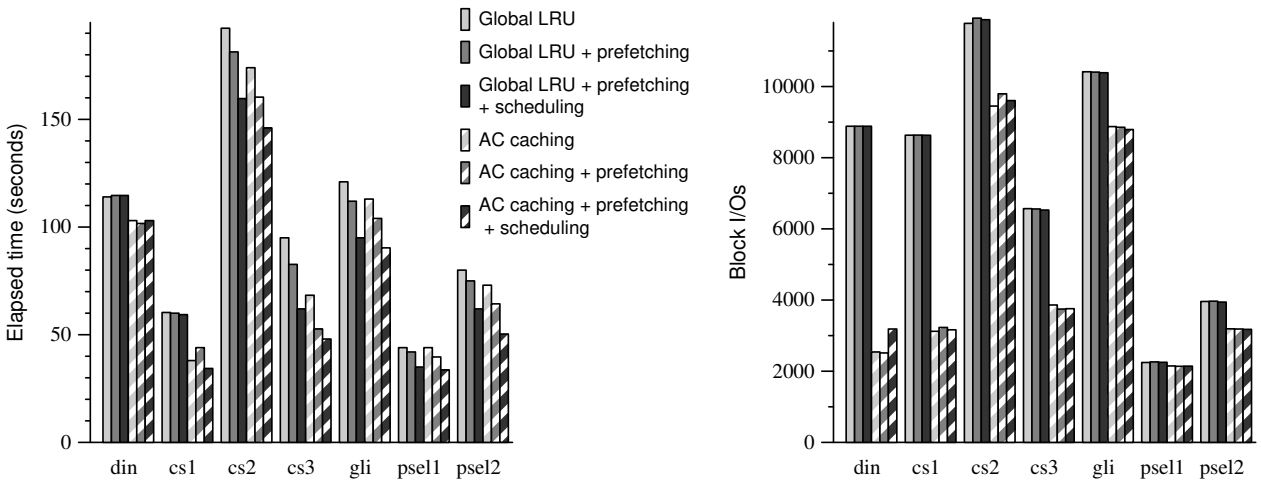


Figure 4: Single application running time, and number of block I/Os, with a 6.4MB file cache.

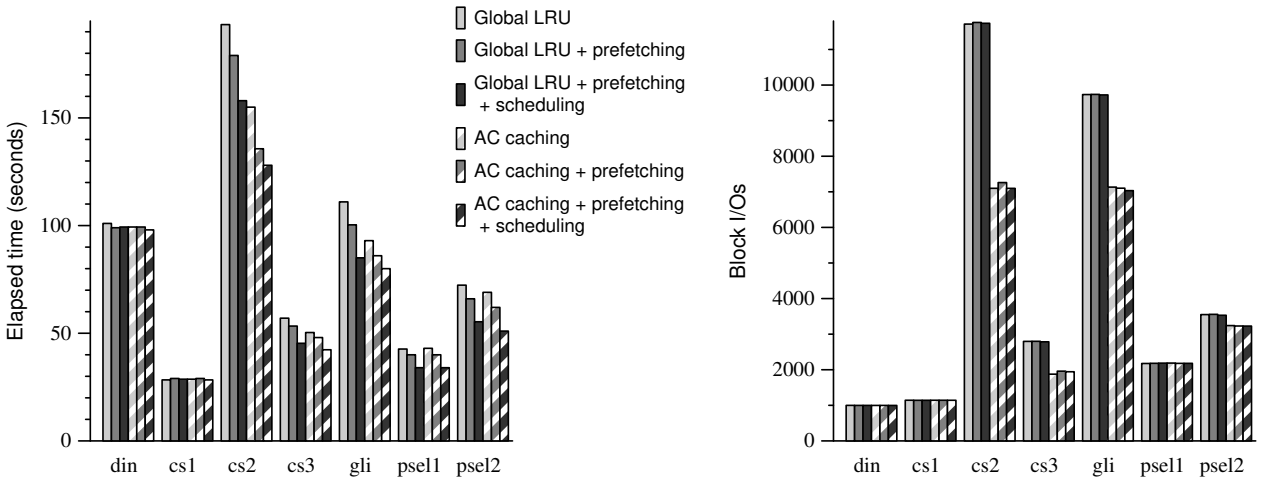


Figure 5: Single application running time, and number of block I/Os, with a 12MB file cache.

### 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%.

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

Looking at the data in more detail, we see that the applications seem to fall into two classes. `Din` and `cs1` make up the first class; they are programs that do primarily sequential accesses to large files. 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.

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

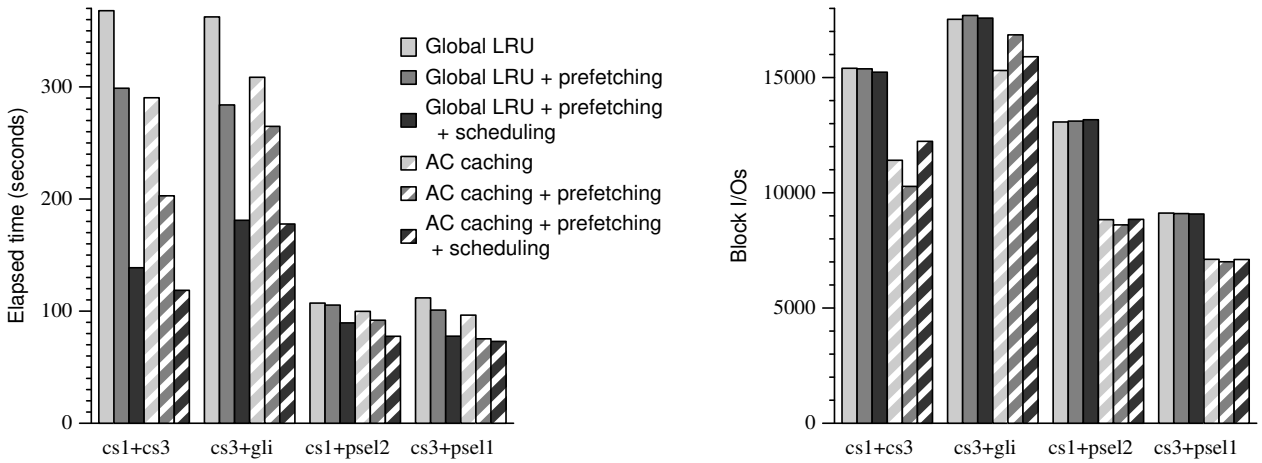


Figure 6: Multiple application running time, and number of block I/Os, with a 6.4MB file cache.

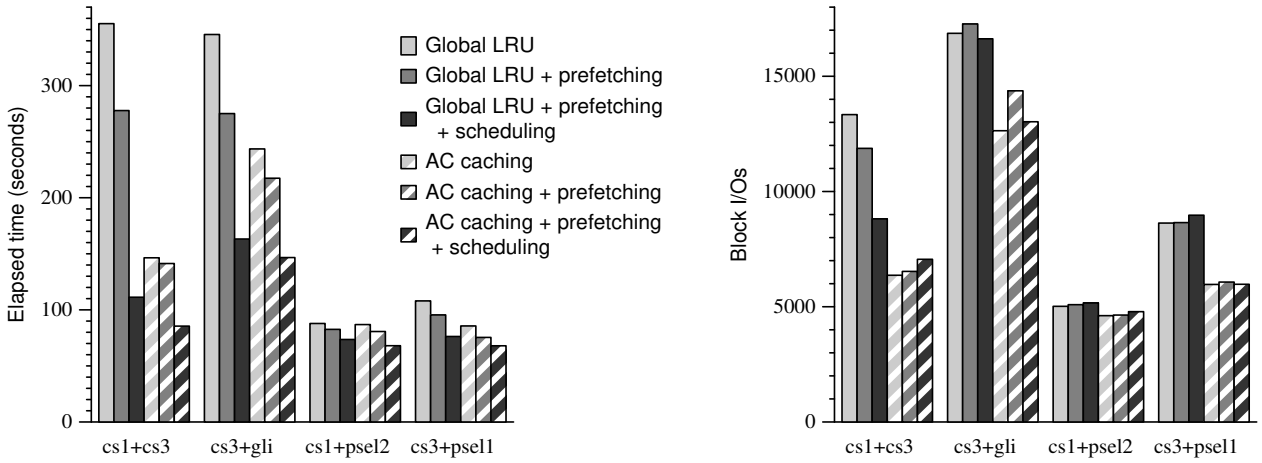


Figure 7: Multiple application running time, and number of block I/Os, with a 12MB file cache.

creases 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.

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 sum of the "bonuses" due to the individual techniques.

We note that *controlled-aggressive* prefetching usually has little effect [3] on the total number of disk accesses performed. Figures 4 and 5 show that the main factor influencing the number of I/Os is whether or not application-controlled caching is being used.

As predicted by recent studies, disk scheduling substantially improves performance for applications accessing small files and performing random accesses to large files. Figure 8 shows the average latency of disk accesses in `cs3` as a function of batch sizes, revealing the source of the performance gains.

### 5.4 Multiple Process Performance

We also measured the performance of our file system implementations for multi-process workloads, running several pairs of applications. The combina-

[3] The exception is din with a 6.4MB cache. This behavior is due to a relatively small fetch cost compared with CPU time, and is predicted in our simulation study in [CFKL95].

tions were 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. The results are summarized in figures 6 and 7. 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%, 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. As above, application-controlled caching, prefetching, and disk scheduling each provides a separate benefit, and combining all three techniques provides the best performance of all.

In the single-disk experiments, we see an interesting effect: disk scheduling seems to be 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 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.

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 means that each process can, on average, do more disk accesses before giving up the CPU. Thus, the locality of disk accesses is improved.

Finally, we notice that some runs have high variances. We are not exactly 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.

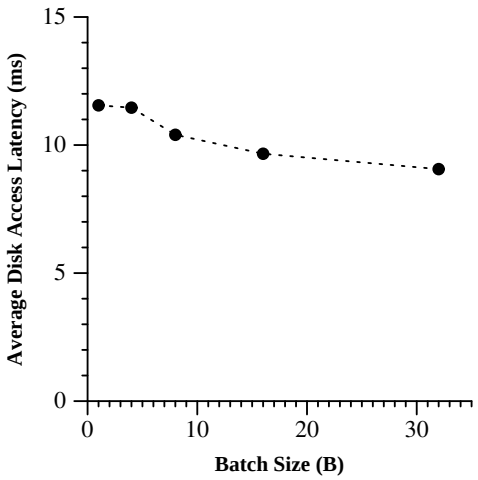


Figure 8: Average disk access latency as the batch size increases, for `cs3`. Note that the reduction in disk latency need not improve the elapsed time — the reordering of requests and resultant dis-overlapping of CPU computation and disk access must be taken into account.

## 6 Related Work

There have been many studies on caching in file systems (e.g. [SGN85, Che87, SBK+85, HKM+88, NWO88, KS92]), but these investigations were not primarily concerned with the performance impact of different caching and prefetching policies. Prefetching has also been studied extensively in various domains, ranging from uni-processor and multiprocessor architectures [RL92, CB92, CKP91, Smi78, TE93], to file systems [PG94, GA94, TD90] to databases [CKV93, PZ91] and beyond. There have also been many studies focusing on how to predict future accesses from past accesses [TD90, PZ91, CKV93]. Few of these studies considered the interaction between prefetching and caching.

Recently there have been a number of research projects on prefetching in file systems. Patterson's Transparent-Informed Prefetching [PG94] showed that prefetching using hints from applications is an effective way of exploiting I/O concurrency in disk arrays. Griffioen and Appleton's work [GA94] tries to predict future file accesses based on past accesses using "probability graphs," and prefetches accordingly. These papers demonstrated the benefits of prefetching. However they did not address the interaction between caching and prefetching and did not investi-

gate the combined cache management problem.

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 did not address the integration of caching, prefetching and disk scheduling in multi-process environments.

There are a number of papers 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].

Prefetching in uni-processor and multi-processor computer architectures 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.

Recently several research projects have tried to improve file system performance in a number of other ways, including log-structured file systems [Hag87, RO91, SBadCS93], 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 did not address how to integrate prefetching and disk scheduling with application-controlled file caching. Our work is complementary to these approaches: in other words, with application-controlled file caching, these approaches will improve file system performance even more.

Several detailed studies of disk scheduling have been done [SCO90, WGP94]. These studies typically considered a wide variety of scheduling policies under timesharing Unix workloads. Although they considered more complex policies than we do, they did not consider the question of how to integrate disk scheduling with other performance improvement techniques like application-controlled caching.

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* cache management 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.

- This paper is the first to present policies for integrated prefetching and caching in the multi-process case — [CFL94a] and [CFL94b] did not consider prefetching, and [CFKL95] discussed only the single-process case.
- This paper presents the first working implementation of our ideas for integrating prefetching and application-controlled caching — our previous implementation described in [CFL94b] did not address prefetching, and our studies of prefetching in [CFKL95] were based on theoretical analysis and simulation only.
- This paper is the first to consider the interaction of disk scheduling with application-controlled caching and prefetching.
- The experiments described in this paper are the first real measurements of the interaction between application-controlled caching and prefetching.

## 7 Conclusions

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

Our experimental results show that these techniques each provide significant benefit for most of the applications. Experiments also show that the fully integrated system, with all three techniques in place, provides the best performance of all for both single application and multi-process workloads. The performance gains from prefetching and disk scheduling are particularly pronounced when running a multi-process workload; the batching of prefetch requests and the increase in time between context switches

allows each process to do more disk accesses before relinquishing the CPU and hence to benefit from locality on the disk.

Many open questions remain. From an experimental point of view, we need to measure the performance of our system on a greater variety of benchmarks and on multi-process workloads consisting of larger numbers of concurrently executing applications. From a theoretical point of view, we are trying to extend our results on prefetching to a model which incorporates nonuniform fetch time. We are also planning to study optimal prefetching strategies in the presence of multiple disks. It would be nice to provide some theoretical backing for our experimental finding that a batch-size of 16 is best for batch prefetching with disk scheduling. Finally, we are trying to understand the interaction between prefetching, cache management and CPU scheduling, so as to come up with an integrated strategy that provides the same kind of theoretical fairness and performance guarantees that LRU-SP provides for application-controlled caching alone.

## References

## References

[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 da tabase 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.

[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 *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.

[Hag87] Robert B. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of 11th ACM Symposium on Operating Systems Principles*, pages 155–162, November 1987.

[HKM+88] 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 distribuetd file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

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

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

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like performacne 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.

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

[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.

[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.

[SBadCS93] Margo Seltzer, Keith Bostic, and Marshall Kirk McKusick an d Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of 1993 Winter USENIX*, January 1993.

[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.

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

[SGN85] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. *Operating System 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.

[SS95] Margo Seltzer and Keith A. Smith. File system logging versus clustering: A performance comparision. In *Proceedings of 1995 Summer USENIX*, June 1995.

[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. Detectino 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.