Implementation and Performance of Application-Controlled File Caching

Pei Cao, Edward W. Felten, and Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA
{pc,felten,li}@cs.princeton.edu

Abstract

Traditional file system implementations do not allow applications to control file caching replacement decisions. We have implemented two-level replacement, a scheme that allows applications to control their own cache replacement, while letting the kernel control the allocation of cache space among processes. We designed an interface to let applications exert control on replacement via a set of directives to the kernel. This is effective and requires low overhead.

We demonstrate that for applications that do not perform well under traditional caching policies, the combination of good application-chosen replacement strategies, and our kernel allocation policy LRU-SP, can reduce the number of block I/Os by up to 80%, and can reduce the elapsed time by up to 45%. We also show that LRU-SP is crucial to the performance improvement for multiple concurrent applications: LRU-SP fairly distributes cache blocks and offers protection against foolish applications.

1 Introduction

File caching is a widely used technique in file systems. Cache management policy is normally centrally controlled by the operating system kernel. Recently, we have shown by simulation that application-controlled file caching can offer higher file cache hit ratios than the traditional approach [3]. This paper presents the design and implementation of an application-controlled file cache and reports its performance benefit under a real application workload.

The design of our application-controlled file cache is based on a two-level replacement scheme proposed in [3]. This method lets the kernel dynamically allocate cache blocks to user processes, and allows each user process to apply its favorite file caching policy to its blocks. A cache block allocation policy, called Least-Recently-Used with Swapping and Placeholders(LRU-SP), is used to guarantee the fairness of allocation.

We designed an interface of user-to-kernel directives to enable applications to control file cache replacement. The interface is designed to be sufficiently flexible for applications to express desired strategies in the common cases, and yet to have low overhead.

We implemented the application-controlled file cache for the Ultrix file system on the DEC 5000/240 workstation. We compared the performance of our file cache to that of the traditional, global LRU approach, using several real applications. First, we showed that a well-chosen caching policy can reduce cache misses (and hence disk I/Os) significantly. In our single-application experiments, good policies can reduce cache misses by between 10% and 80%. As a result, these policies can reduce the elapsed time of applications by up to 45%. Second, we measured the effectiveness and fairness of our LRU-SP allocation policy. Using various mixes of concurrent applications, we showed that our implementation can reduce their elapsed times by up to 30% over the global LRU approach.

We also compared our implementation with our earlier simulation study. This confirmed the result from our simulation study that the added features of LRU-SP (swapping and placeholders) are important for performance improvement. Although LRU-SP does not always provide *perfect* protection against foolish processes, it significantly reduces their effect, and it provides an easy way for the kernel to detect foolish or malicious behavior.

2 Two-Level Replacement and LRU-SP

This section presents some background material about two level block replacement and the LRU-SP global allocation policy. For a full discussion see [3].

The challenge in application-controlled file caching

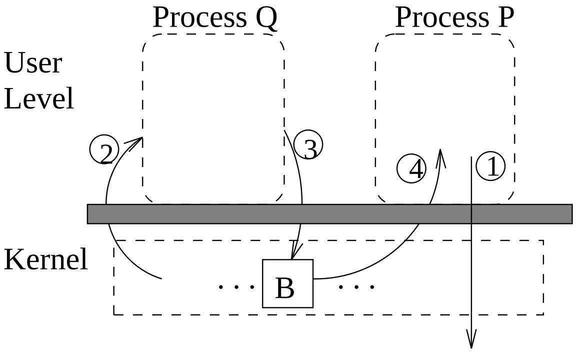

Figure 1: Interaction between kernel and user processes in two-level replacement: (1) P misses; (2) kernel consults Q for replacement; (3) Q decides to give up page B; (4) kernel reallocates B to P.

is to allow each user process to control its own caching *and* at the same time to maintain the dynamic allocation of cache blocks among processes in a fair way so that overall system performance improves.

To achieve this goal, we use *two level block replacement.* Two-level replacement splits the responsibility for allocation and replacement between the kernel and user level. The kernel is responsible for allocating cache blocks to processes, while each user process can control the replacement strategy on its share of cache blocks. When a user-level process misses in the cache, the kernel chooses a process to give up a cache block. The designated process is free to give up whichever block it likes. The kernel's *allocation policy* is used to decide which process will give up a block.

The interactions between the kernel and the controlling user process (called the manager process) are as follows: on a cache miss, the kernel first finds a candidate block to replace, based on its "global replacement" policy (step 1 in Figure 1). The kernel then identifies the manager process for the candidate. This manager process is given a chance to decide on the replacement (step 2). The candidate block is given as a hint, but the manager process may overrule the kernel's choice by suggesting an alternative block under that manager's control (step 3). Finally, the block suggested by the manager process is replaced by the kernel (step 4). (If the manager process does not exist or is uncooperative, then the kernel simply replaces the candidate block.)

Clearly, the kernel's "global replacement" policy is actually not a replacement policy at all (as it doesn't really decide which block to replace) but rather a global *allocation* policy. A sound global allocation policy is crucial to the success of two level replacement. It should satisfy these three criteria:

- Oblivious processes (those that do not want to manage their share of the cache) should do no worse than they would have done under the existing LRU policy.

- Foolish processes should not hurt other processes.

- Smart processes should perform better than under the existing LRU policy whenever possible, and they should never perform worse.

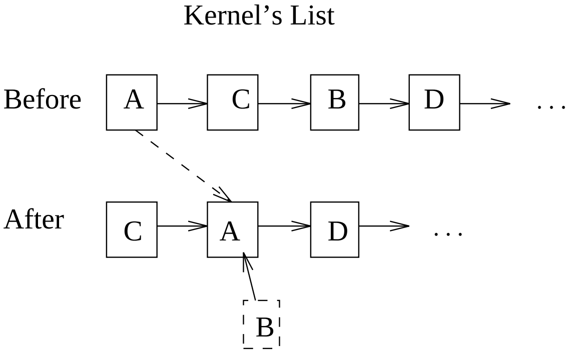

Figure 2: How LRU-SP keeps the "LRU" list. Here the kernel suggests block A as a candidate for replacement; the application overrules the kernel, and causes block B to be replaced instead. The figure shows the list before and after the replacement decision.

In [3] we proposed a kernel policy called LRU-SP that satisfies these criteria. LRU-SP operates by keeping an LRU list of all cache blocks and having two extensions to the "basic LRU" scheme:

- If the kernel suggests block A for replacement and the user-level manager chooses to replace 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 manager's choice. (See Figure 2.)

- If a user process misses on a block B, and a placeholder for B exists, then the block pointed to by that placeholder is chosen as the candidate for replacement. Otherwise, the block at the end of the LRU list is picked as the candidate.

If an application never overrules the kernel's suggestions, then no swapping or placeholders are ever used for that application's blocks, so the application sees an LRU policy. If it uses a better policy than LRU to overrule the kernel's suggestion, then swapping is necessary so that it won't be penalized by the kernel. If it uses a worse policy than LRU, then placeholders are necessary to prevent it from hurting other processes. For more details please see [3].

3 Supporting Application Control

There are a variety of ways to implement the user-kernel interaction in two level block replacement, as discussed in [3]. 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.

Our approach was to first consider common file access patterns reported in the literature (e.g. [24, 1, 6]) as well as patterns that we observed in the applications discussed below. We then chose a simple interface that was sufficient to compose caching strategies to support these access patterns.

An Interface for Application Control

The basic idea in our interface is to allow applications to assign priorities to files and to specify file cache replacement policies for each priority level. The kernel always replaces blocks with the lowest priority first. (This rule applies only within the blocks of a single process. Inter-process allocation decisions are made using LRU-SP, as explained above.) The calls for applications are:

- `set_priority(file, prio)` and `get_priority(file)` set and get the long-term cache priority of a file.
- `set_policy(prio, policy)` and `get_policy(prio)` set and get the file cache replacement policy of a priority level. At present, we offer only two policies: least-recently-used (LRU) and most-recently-used (MRU). The default policy is LRU.

Since files 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 apply a single policy to all files, can apply different policies for different pools of files, and can change priorities to tell the kernel to replace some files before others.

Our interface also has a primitive for applications to assign temporary priorities to file blocks:

```
set_temppri(file,startBlock,endBlock,prio)
```

This is a mechanism to allow applications to temporarily change the priority of a set of blocks within a file. This temporary 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. When the temporary priority ends, the block reverts to its long-term priority as set by `set_priority`.

These five operations are multiplexed through a single new `fbehavior` system call, in the same way that the Unix `ioctl` system call multiplexes several operations.

Supporting Common Access Patterns

We now illustrate how our interface can be used to support a variety of common file access patterns. Further examples are found in section 5, in which we discuss how we used our interface to control caching for a set of real applications.

Sequential Applications often read a file sequentially from beginning to end. If the file is accessed sequentially repeatedly, it can be assigned a low priority and MRU replacement policy. If the file is accessed only once, it can be assigned priority -1 to flush its block from cache quickly.

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

Access to many small files Some applications access many small files. Since our interface treats all files of the same priority as being in a single "pool" when making replacement decisions, these applications can control the caching of these blocks. (An interface that allowed control only within individual files would be useless for tiny files.)

Hot and cold files Some applications have particular files that are accessed unusually often, or with unusually good temporal locality. Such files can be assigned high priority.

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

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

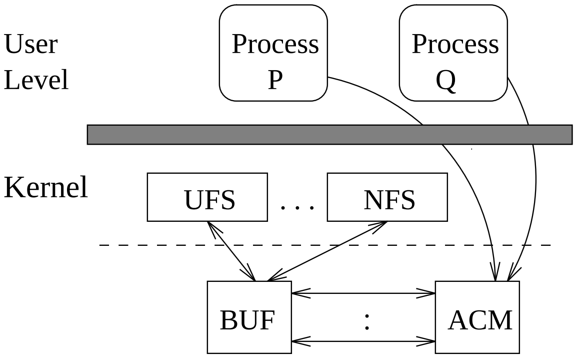

Figure 3: Implementation of Application Controlled Caching.

4 Kernel Implementation

The kernel implementation of two-level replacement is done by breaking the existing buffer cache code into two modules. The buffer cache module (BUF) handles cache management and bookkeeping and implements the allocation policy. The application control module (ACM) implements the interface calls and acts as a proxy for the user-level managers. Both of these modules sit below the VFS interface and communicate via procedure calls. The implementation is done on a Ultrix 4.3 kernel. [1]

When a block must be replaced, BUF picks a candidate block and asks ACM which block to replace, giving the candidate block as a suggestion. ACM then acts on behalf of the appropriate user-level manager to make the decision. The structure is shown in Figure 3.

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 head (tail) of the list. Blocks may move among lists by `set_priority` or `set_temppri`. The implementation uses the rule that blocks moving into a list will be 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 by using five procedure calls. They serve the purpose of notifying ACM about replacement decisions and mistakes, informing ACM about changes in cache state and accesses, and asking ACM for replacement decisions. The calls are:

- `new_block(block)`: informs ACM that block was loaded into cache buffer;
- `block_gone(block)`: informs ACM that block was removed from the cache;
- `block_accessed(block, offset, size)`: informs ACM that 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 block was erroneous.

Changes are needed in the replacement procedure in BUF to implement LRU-SP. Instead of replacing the LRU (Least-Recently-Used) block, the procedure first checks if the missing block has a placeholder, then takes the LRU 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.

Since BUF and ACM sit below the VFS interface, our implementation works for both UFS [18] and NFS [26]. In all our following experiments, however, we put all files on local disks to avoid impact on performance from network traffic.

We believe this interface could be used if one wanted to implement two level replacement using upcalls or dropped-in-kernel "user-level handlers" [2]. In particular user-level handlers could know which blocks are in cache by keeping track of `new_block` and `block_gone` calls.

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

For processes that control their own caching, the extra overhead on a cache miss is similar to the above case except that `new_block` is called as well. The

[1] Contact the authors for portions of the code that are not vendor-proprietary.

overhead on a cache hit depends on the work done in `block_accessed`, which in our current implementation only involves updating counters and moving a few blocks around in lists. These overheads are often smaller than the potential reduction in system CPU time caused by reducing the number of misses.

5 Performance Results

Beneficiaries of application controlled caching include applications that use large file data sets (i.e. they do not fit in cache), because LRU is often not the right policy for them. Examples include: database systems, including text retrieval software; trace-driven simulations, as the trace files are often large; graphics and image-processing applications; and I/O-intensive UNIX utilities.

We chose a set of such applications for our experiments. Below we first describe the applications that we chose and the replacement strategies they use; then we present performance results for single and multiple application experiments.

5.1 Applications

We chose the following applications:

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 for cscope: symbol-oriented ones, mostly questions about where C-symbols occur; and egrep-like text search, mostly asking where patterns of text occur. We used cscope on two software packages (in fact, two kernel sources), one about 18MB and one about 10MB, and did two sets of queries: searching for eight symbols, and searching for four text patterns. The three runs are: **cs1** — symbol search on the 18MB source; **cs2** — text search on the 18MB source; **cs3** — text search on the 10MB source.

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

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

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

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

(When there is a mix of these queries, cscope can keep or discard "cscope.out" in cache when necessary by raising or lowering its priority.)

dinero [din] Dinero is a cache simulator written by Mark Hill and used in Hennessy and Patterson's architecture textbook [13]. 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, and set associativity from 1 to 4.

Strategy: Dinero reads the trace file sequentially on each simulation. Hence the right policy is MRU on trace file and the call is:

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

glimpse [gli] Glimpse is a text information retrieval system [17]. It builds approximate indexes for words to allow both relatively fast search and small index files. We took a snapshot of news articles in several comp.* newsgroups on May 22, 1994, about 40MB of texts. Then we `glimpseindex`ed it, resulting in about 2MB of indexes. The searches are for lines containing these keywords: scheduling, scheduling and disk, cluster, rendering and volume, DTM.

Strategy: The index files and data files naturally lead to two priority levels because index files are always accessed first on every query, but data files are not. Hence glimpse gives the index files long-term priority 1, and the articles the default long-term priority 0. Since index files are always accessed in the same order, and several groups of articles (called partitions [17]) are accessed in the same order, MRU is chosen for both levels. The calls are:

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

link editor [ldk] Ld is the Ultrix link-editor. We used it to build the Ultrix 4.3 kernel from about 25 MB of object files.

Strategy: Ld almost never accesses the same file data twice, but it does lots of small accesses, so the right thing to do is to free a block whenever its data have all been accessed by calling [2]:

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

[2] We can't obtain the source of the DEC MIPS link editor, so we implemented this policy in the kernel to simulate what the program would do and called it "access-once".

postgres join [pjn] Postgres is a relational database system from the University of California at Berkeley. We used version 4.0.1 for Ultrix 4.3. It uses the file system for I/O operations, and since it only has a small internal buffer, it relies heavily on file caching. We chose one query operation: a join between an indexed and a non-indexed relation, to illustrate how it can use application control on file caching.

The relations are a 200,000 tuple one, `twohundredk`, and a 20,000 tuple one, `twentyk`, from a scaled-up Wisconsin benchmark[10]. The join is on field `unique1`, which is uniquely random within 1-200,000 in `twohundredk`, and uniquely random within 1-1,000,020 in `twentyk`. The size of `twentyk` is roughly 3.2MB, `twohundredk` 32MB, and index `twohundredk_unique1` 5MB.

Strategy: Since there is a non-clustered index on `unique1` in `twohundredk`, and no index in `twentyk`, Postgres executes the join by using `twentyk` as the outer relation and using the index to retrieve tuples from `twohundredk`. The index blocks have a much higher probability of being accessed than data blocks. Therefore postgres uses two priority levels: the index file has long-term priority 1, while data files have default priority 0. LRU is used for both priority levels. The only call necessary is:

```
set_priority("twohundredk_unique1", 1);
```

sort [sort] Sort is the external sorting utility in UNIX. We used a 200,000-line, 17MB text file as input, and sorted numerically on the first field.

Sort has two phases: it first partitions the input file into sorted chunks that are stored in temporary files, then repeatedly merges the sorted files. Its access pattern has the following characteristics: input files are read once; temporary files are written once and then read once; merging is done eight files at a time, and temporary files are merged in the order in which they were created.

Strategy: Sort has two priority levels, -1 and 0. Input files have priority -1 as they are read once. Temporary files have default priority 0. MRU is chosen for both level -1 and level 0 (because temporary files created earlier will be merged first). The calls are:

```
set_policy(-1, MRU);
set_policy(0, MRU);
set_priority(input_file, -1);
```

In addition, the "readline" routine is changed to keep track of when the end of an 8K block is reached, and at the end of reading a block to free the block by calling:

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

We experimented with other strategies, and we found that the ones given here performed the best.

5.2 Single Applications

We compared the performance of each application under application-controlled file caching to that under the kernel-controlled file caching (the original, unmodified kernel). We used a DEC 5000/240 workstation with one RZ56 disk and one RZ26 disk. The RZ56 is a 665M 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.

Applications **cs[1-3]**, **din**, **gli** and **ldk** were run on the RZ56 disk, and **pjn** and **sort** were run on the RZ26 disk.

We measured both the number of block I/Os, reflecting the cache misses, and the elapsed time, for several configurations of the buffer cache size (6.4MB[3], 8MB, 12MB, 16MB). Figure 4 shows the elapsed times and the number of block I/Os *normalized* to those under the original kernel. The raw data for this experiment appear in the appendix[4].

Application-specific policies reduce the number of block I/Os by an amount ranging from 9% to over 80%. This confirms our simulation result [3] that miss ratio is reduced by application controlled caching. The reduction in elapsed time ranges from 6% to over 45%. Elapsed time is not directly proportional to block I/Os because elapsed time is also affected by the amount of CPU computation time, and because the disk access latency is not uniform.

The performance improvement often depends on the relative size of the cache versus the data set. For example, for `cs1`, the improvement increases until the 9MB database file "cscope.out" fits in cache, at which point all policies perform the same. In some cases LRU makes a bigger cache useless — there is no benefit until the entire data set can fit in the cache. Application specific policies, on the other hand, can appropriately take advantage of a bigger cache and improve application performance. As DRAM density improves, we will see ever-larger caches, so the benefit of application control will become more significant.

5.3 Multiple Applications

In our previous paper we used simulations to show that our global allocation policy LRU-SP can properly take advantage of application control on replacement to improve the hit ratio of the whole system. Now

[3] This is 10% of the memory of our workstation, which is the default cache size under Ultrix.

[4] Raw data for all of our experiments are available at ftp.cs.princeton.edu: /pub/pc/OSDI/.

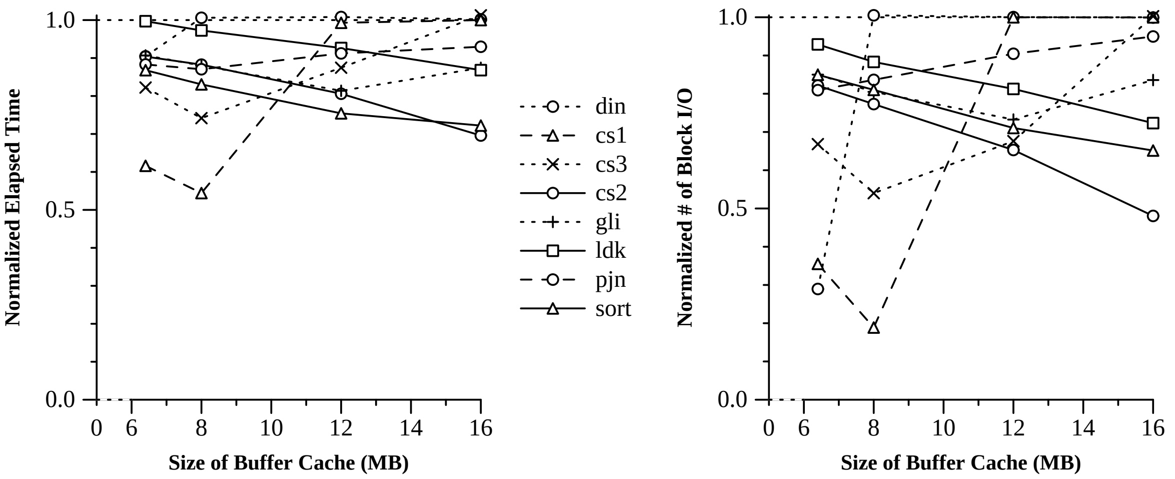

Figure 4: Normalized elapsed time and block I/Os of individual applications under LRU-SP, versus file cache size. Elapsed times under the original kernel are set to 1.0. All numbers are averages of five cold-start runs; variances are less than 2% with a few exceptions that are less than 5%.

that we've implemented the system, we'd like to see whether this is true in practice.

We ran several combinations of two or more of the above applications, with each application using its appropriate user-level replacement policy. The combinations were chosen to have various mixes of access patterns. For this purpose, we put the applications into categories based on their access patterns. We put **cs[1-3]** and **din** into the "cyclic" category, **gli** and **pjn** into the "hot/cold" category, and **ldk** and **sort** in two separate categories. We ran six two-application combinations, chosen to cover all combinations of the four categories. We also randomly chose two three-application combinations and one four-application combination.

The combinations are: **cs2+gli**, **cs3+ldk**, **gli+sort**, **din+sort**, **sort+ldk**, **pjn+ldk**, **din+cs2+ldk**, **cs1+gli+ldk**, and **din+cs3+gli+ldk**. Note that **gli+sort**, **din+sort**, **sort+ldk**, **pjn+ldk** are run using two disks, while all others are run using one disk.

We measured the total elapsed time and the total number of block I/Os for these concurrent applications. Figure 5 shows the *normalized* elapsed time and the number of block I/Os. As can be seen, LRU-SP indeed improves the performance of the whole system. The improvement becomes more significant as the file cache size increases.

6 Analysis of LRU-SP

In our simulation study [3] we found that both the **swapping** and **placeholders** techniques in LRU-SP are necessary, and that LRU-SP satisfies our allocation criteria as described in Section 2. In this section, we investigate these questions again with our real implementation.

6.1 Comparison with ALLOC-LRU

LRU-SP keeps a basic LRU list of cache blocks to choose victim processes, but makes two variations, **swapping** and **placeholders**, to the list, as described in Section 2. In the following we first see whether swapping is necessary, then check whether placeholders are necessary.

Is swapping necessary? Let's call the basic scheme that simply uses the LRU list to choose victim processes the ALLOC-LRU (ALLOCation by LRU order) policy. With ALLOC-LRU as allocation policy in two level replacement, we randomly chose the following from our above experiments: **cs2+gli**, **cs3+ldk**, **din+cs2+ldk**, **cs1+gli+ldk**, and **din+cs3+gli+ldk**, and compared the results to those of LRU-SP. The results of this comparison are shown in Figure 6.

In most cases ALLOC-LRU performs worse. In fact, in a few cases, under ALLOC-LRU applications are better off not using smart policies — smarter allocation hurts their performance[5]! The problem is that ALLOC-LRU uses straight LRU order without swapping, and hence penalizes any process that does not replace the LRU block. These results show that swapping positions of candidate and alternative blocks is necessary.

Are placeholders necessary? To see whether placeholders are also necessary, we need an application that can have a replacement policy that does much

[5] For detailed data, see ftp.cs.princeton.edu: /pub/pc/OSDI.

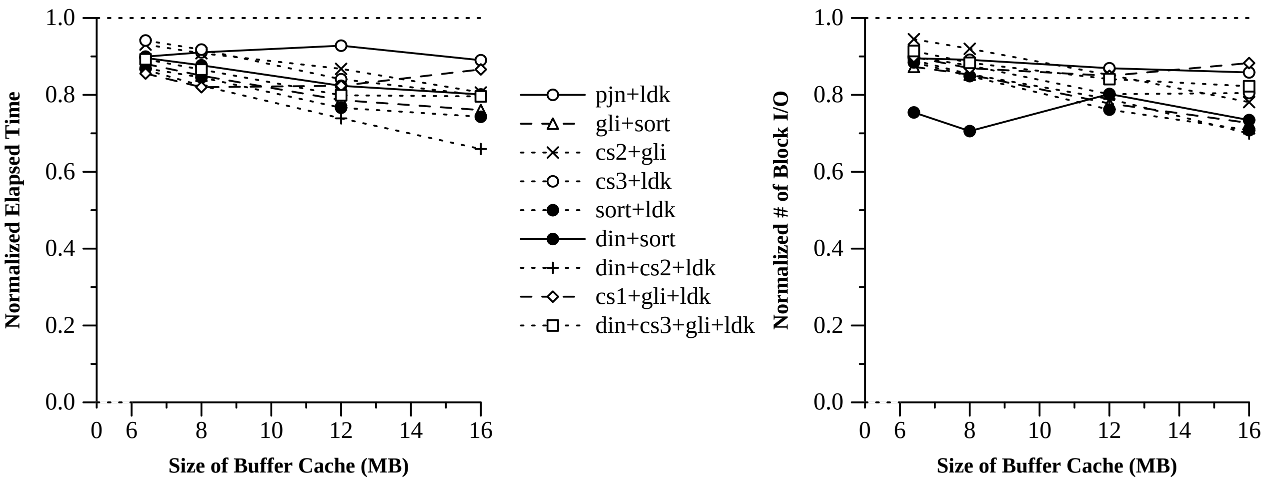

Figure 5: Normalized total elapsed time and block I/Os of multiple concurrent applications. Values under the original kernel are normalized to 1.0. All numbers are averages of three cold-start runs; variances are less than 2%.

worse than LRU [6], and an oblivious application whose performance is sensitive to the number of cache blocks it is given. We found a program "ReadN" that can serve both purposes.

ReadN sequentially reads the first N 8K-byte blocks from a file in sequence, repeating this sequence five times, then reads the next N blocks five times, and so on. MRU replacement policy is much worse than LRU for ReadN, so ReadN with MRU can be used as an example of a foolish application. ReadN also has the characteristic that under LRU, the cache miss ratio is low when it has at least N cache blocks, but is high when it has less than N cache blocks. Therefore we can compare the number of block I/Os done by ReadN in different situations as a way of measuring what share of the cache blocks it is getting.

To answer our question about the effectiveness of placeholders, we ran two versions of ReadN concurrently. One version, with N=300, is run with various replacement policies: the oblivious LRU policy (which is good but not optimal), and MRU (which is terrible). The other version of ReadN is used to detect changes in allocation of cache blocks. This version is run with various values of N: Read390, Read400, Read490, and Read500. These values are chosen because our cache size is 6.4MB, or 819 blocks, so Read300 and Read390 can comfortably fit in cache together with plenty of space to spare, while Read300 and Read500 can barely just fit in cache together.

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

In the case where the background Read300 is foolish, we actually ran two experiments. In the "protected" experiment, the kernel uses our LRU-SP allocation policy. In the "unprotected" experiment, the kernel uses LRU-SP *without placeholders*. Thus there are three cases overall:

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

The results are shown in Table 1. The data clearly shows that place-holders are necessary to protect the oblivious **readN** from losing its share of cache blocks. (For a detailed explanation of why placeholders provide this protection, see [3].) However, the data also shows that placeholders did not prevent the increasing in elapsed times of ReadN. The next subsection explains why.

6.2 Criteria Test

We have seen in Section 5 that smart processes improve their performance. The next two questions are: do foolish processes hurt other processes, and do smart processes hurt oblivious processes?

[6] We do not use the previous applications here because LRU works poorly for them.

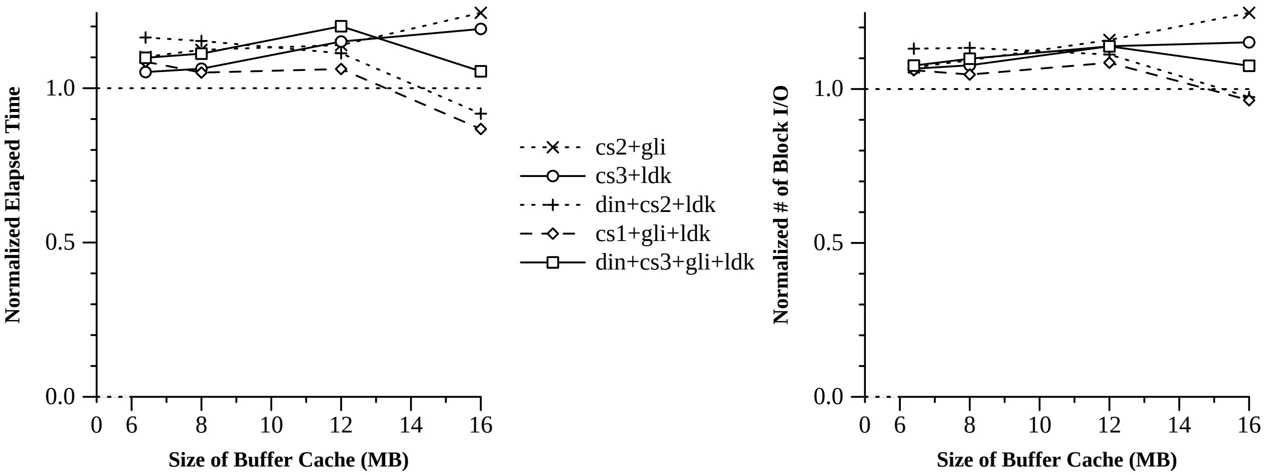

Figure 6: Normalized total elapsed time and block I/Os of multiple concurrent applications under **ALLOC-LRU**. Values under two level replacement with **LRU-SP** are normalized to 1.0. All numbers are averages of three cold-start runs; variances are less than 2%.

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

Table 1: Results from experiments done to test the effectiveness of placeholders. For descriptions of these experiments please see text. The first four columns show the running time in seconds, and the next four the number of block I/O's, for ReadN. Results are averaged over five runs. Variances are shown in parentheses; when omitted they are less than 5%.

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

Table 2: Effect of a foolish process on smart applications. This table shows performance of various processes when run concurrently with a Read300 process, depending on whether the Read300 is oblivious (using LRU) or is foolish (using MRU). The first four columns shows elapsed time in seconds, and the next four the number of block I/Os. Numbers are averages of three runs with all variances < 3%.

Application Policies	Elapsed time of "Read300"			
	w. din	w. cs2	w. gli	w. ldk
Oblivious	87	88	60(7%)	78
Smart	67(8%)	83	64(9%)	76

Table 3: Elapsed time of "Read300" when run concurrently with oblivious and smart versions of other applications on one disk. Numbers are in seconds, averages of three runs with variances <4% except those shown in parenthesis. Read300's numbers of block I/Os are the same in all cases (about 1310) as they are all compulsory misses.

Do foolish processes hurt other processes? We have that seen placeholders at least limit the harm done to oblivious processes by foolish processes. To see whether smart processes are protected as well, we ran each of **din**, **cs2**, **gli** and **ldk**, all using smart policies, concurrently with "Read300". The results when "Read300" is oblivious and when it is foolish are shown in Table 2.

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

- Foolish processes generate more I/O requests, and thus put a heavier load on the disk. This leads to longer queueing time for the disk, so non-foolish processes' I/O requests wait longer to be served.
- Foolish processes take longer to finish and therefore occupy cache blocks for a longer time. Processes that run concurrently with them and finish after them normally get an increase in available cache blocks after they are finished. If they finish later, this increase comes later.

The best way to provide protection from foolish processes is probably for the kernel to revoke the cache-control privileges of consistently foolish applications. Placeholders allow the kernel to tell when an application is foolish, so the kernel can keep track of how good the application's policy is. If it turns out that a certain percentage of the application's decisions are wrong, the kernel can revoke its right to control its caching [7].

Do smart processes hurt oblivious ones? To answer this we use "Read300" again, this time as an oblivious application. We run it with each of **din**, **cs2**, **gli**, **ldk**, both when they are oblivious and when they are smart. The results are summarized in Table 3.

In most cases smart processes do not hurt but rather help oblivious processes. Since these experiments are run on one disk (RZ56), the reduction in the number of I/Os from smart processes reduces the load on the disk, and speeds up the requests from "Read300".

Application Policies	Elapsed time of "Read300"			
	w. din	w. cs2	w. gli	w. ldk
Oblivious	20	18	19	17
Smart	20	17.5	18	17

Table 4: Elapsed time of "Read300" when run concurrently with oblivious and smart versions of other applications on two disks. Numbers are in seconds, averages of three runs with variances <3%. Read300's number of block I/Os is the same in all cases (about 1310).

The only exception is the experiments with **gli**, which have a high variance in the elpased time. Since the number of block I/Os from "Read300" did not increase, nor did its user and system times, we suspect that this is due to the RZ56 disk's internal scheduling or buffering. To test this, we also run this experiments on two disks, with "Read300" using RZ26 and others using RZ56. The results are summarized in Table 4.

As can be seen, the anomaly goes away, which suggests that the previous problem was due to the disk contention. The improvements in the elapsed times, however, are not as noticeable as before because "Read300" is using a seperate disk.

From these results we believe that both swapping and placeholders are necessary. Although placeholders do not completely eliminate the harm done by foolish processes, they at least help the kernel take administrative measures to solve the problem.

7 Related Work

There have been many studies on caching in file systems (e.g. [27, 4, 26, 14, 22, 15]), but these investigations were not primarily concerned with the performance impact of different replacement policies. Recently several research projects have tried to improve file system performance in a number of other ways, including prefetching[30, 25, 7, 5, 11], delayed writeback[21] and disk block clustering[20, 29]. Most of these papers still assume global LRU as the basic cache replacement policy, and they do not address how to use application control over replacement to improve cache hit ratio. Our work is complementary to these approaches; in other words, with application-controlled file caching, these approaches will improve file system performance even more.

The database community has long studied access patterns and buffer replacement policies [31, 6, 23]. In our experiments we used **pjn** (Postgres join) as an example of how database systems may use our inter-

[7] We are adding this in our implementation.

face to control file caching. We believe our interface is flexible enough to implement most of the policies a database system might want to use.

A very similar problem is application controlled physical memory management in virtual memory systems. Current operating systems provide limited facilities that are not adequate for application control. For example, the Berkeley UNIX system calls for pinning pages, `mpin` or `mlock`, are often only available to the superuser. Advisory calls like `vadvise` or `madvise` provide an interface for applications to advise the kernel about their access patterns over a memory object, including a mapped file. However, this interface is much more limited than ours, allowing specification of only a small number of basic patterns and no priorities.

As a result, in the past few years there has been a stream of research papers on application control of caching in the virtual memory context [19, 28, 12, 16]. Our work differs from that described in these papers in three significant ways:

- None of these papers (except [12]) addresses the global allocation policy problem. By contrast, we discuss this problem in detail, and provide a solution, LRU-SP, which we have simulated [3] and now have implemented.
- Most of these papers relied on RPC or upcalls for kernel-to-user communication, and consequently reported overhead as high as 10% of the total execution time [19, 28]. We provide a flexible interface for applications to issue primitives to exert control on cache replacement; we found that this is adequate most of the time and requires low overhead.
- Most of these papers do not adequately consider which replacement policy an application should use. In [3] we proposed that application replacement policies be based on the optimal replacement principle [8], and in this paper we discussed which policies to use for our example applications.

On the other hand, our work shares some common purposes with the existing work on application-controlled virtual memory. We believe that, with some minor modifications, our approach applies to virtual memory cache management as well.

First, swapping and placeholder techniques apply to the virtual memory caching problem — one can swap positions of pages on the two-hand-clock list, and can build placeholders to catch foolish decisions. Second, our interface can be modified to apply to virtual memory context, i.e. instead of files, we use a range of virtual addresses (or memory regions).

One important difference is that in the virtual memory context, the implementation cannot capture the exact reference stream as it does in the file caching context. It is still not clear to us what are the common policies applications may want to use for virtual memory caching and whether not capturing exact references is a serious problem.

Our implementation is for Ultrix 4.3 where there is a fixed amount of DRAM memory allocated for file caching. In modern operating systems the virtual memory system and the file system often share a common page buffer pool. We believe our approach to file caching fully applies to these systems as well, with only some minor changes in data structures.

Finally, the difference between this paper and the work described in [3] is that in [3] we proposed and simulated LRU-SP, and in this paper we proposed a concrete scheme for user-kernel interaction and implemented both the interaction scheme and application-controlled caching in a real kernel.

8 Conclusion and Future Work

Our main conclusion is that two level replacement with LRU-SP works. It can be implemented with low overhead, and it can improve performance significantly.

Applications with large data sets often do not perform well under traditional file caching. We have identified common access patterns of these applications, and have designed an interface for them to express control over cache replacement. The allocation policy LRU-SP fairly distributes cache blocks among applications. As a result, the number of disk I/Os can be significantly reduced, and the application's elapsed time and the system's throughput can improve.

We are working on improving our user-kernel interface, and on supporting user-level control over caching of concurrently shared files[3]. In addition, our current implementation ignores metadata blocks like inodes, partly because there is a separate caching scheme for them inside the file system. We plan to look more into metadata caching performance and investigate what should be done.

We also plan to look into the interaction between caching, prefetching, write-back and disk block clustering.

The search for better global allocation policies requires a clear definition of the goal and an examination of the interactions between caching and scheduling/management policies in other parts of the system, particularly process scheduling and disk scheduling[9]. This is a fruitful area for future work.

Acknowledgements

We are grateful to our paper shepherd Jay Lepreau and the anonymous referees for their valuable comments.

Keith Bostic and Randy Appleton also provided helpful feedback.

References

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

[2] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan MaNamee, Przemystaw Pardyak, Stefan Savage, and Emin Gun Sirer. SPIN — an extensible microkernel for applicatoin-specific operating system services. Technical Report TR 94-03-03, Dept. of Computer Science and Engineering, University of Washington, 1994.

[3] Pei Cao, Edward W. Felten, and Kai Li. Application-controlled file caching policies. In *Conference Proceedings of the USENIX Summer 1994 Technical Conference*, pages 171–182, June 1994.

[4] David Cheriton. Effective use of large RAM diskless workstations with the V virtual memory system. Technical report, Dept. of Computer Science, Stanford University, 1987.

[5] Khien-Mien Chew, A. Jyothy Reddy, Theodore H. Romer, and Avi Silberschatz. Kernel support for recoverable-persistent virtual memory. Technical Report TR-93-06, University of Texas at Austin, Dept. of Computer Science, February 1993.

[6] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the Eleventh International Conference on Very Large Databases*, pages 127–141, August 1985.

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

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

[9] Gregory R. Ganger and Yale N. Patt. The process-flow model: Examing I/O performance from the system's point of view. In *Proceedings of SIGMETRICS 1993*, pages 86–97, May 1993.

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

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

[12] 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.

[13] John L. Hennessy and David A. Patterson. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, Inc., 1994.

[14] John H. Howard, Michael Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, pages 6(1):51–81, February 1988.

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

[16] Keith Krueger, David Loftesness, Amin Vahdat, and Tom Anderson. Tools for the development of application-specific virtual memory management. In *OOPSLA '93 Conference Proceedings*, pages 48–64, October 1993.

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

[18] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, August 1984.

[19] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. In *Proceedings of the USENIX Association Mach Workshop*, pages 17–29, 1990.

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

[21] Jeffrey C. Mogul. A better update policy. In *Proceedings of 1994 Summer USENIX*, pages 99–111, June 1994.

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

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

[24] J. K. Ousterhout, H. Da Costa, D. Harrison, J.A. Kunze, M. Kupfer, and J. G. Tompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[25] Hugo Patterson, Garth Gibson, and M. Satyanarayanan. Transparent informed prefetching. *ACM Operating Systems Review*, pages 21–34, April 1993.

[26] 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.

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

[28] Stuart Sechrest and Yoonho Park. User-level physical memory management for Mach. In *Proceedings of the USENIX Mach Symposium*, pages 189–199, 1991.

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

[30] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[31] Michael Stonebraker. Operating system support for database management. *Communications of the ACM, v. 24, no. 7*, pages 412–418, July 1981.

Appendix

These tables show raw performance data for our single application experiments.

Application		Buffer Cache Size			
		6.4MB	8MB	12MB	16MB
din	original	117	99	99	99
	LRU-SP	106	99	100	100
	ratio	0.90	1.01	1.01	1.00
cs1	original	62	61	28	28
	LRU-SP	38	33	27	28
	ratio	0.62	0.54	0.99	1.00
cs3	original	96	96	57	47
	LRU-SP	79	71	50	48
	ratio	0.82	0.74	0.87	1.01
cs2	original	191	190	188	184
	LRU-SP	172	168	152	128
	ratio	0.90	0.88	0.81	0.70
gli	original	126	123	113	97
	LRU-SP	114	108	92	84
	ratio	0.91	0.88	0.81	0.87
ldk	original	66	65	65	65
	LRU-SP	66	64	60	56
	ratio	1.00	0.97	0.93	0.87
pjn	original	225	220	202	187
	LRU-SP	199	192	185	174
	ratio	0.88	0.87	0.91	0.93
sort	original	339	338	339	336
	LRU-SP	294	281	256	243
	ratio	0.87	0.83	0.75	0.72

Table 5: Elapsed time in seconds with/without application-controlled cache. The numbers are average of five runs. Variances are less than 2% with a few exceptions that are less than 5%.

Application		Buffer Cache Size			
		6.4MB	8MB	12MB	16MB
din	original	8888	998	997	998
	LRU-SP	2573	1003	997	997
	ratio	0.29	1.01	1.00	1.00
cs1	original	8634	8630	1141	1141
	LRU-SP	3066	1628	1141	1141
	ratio	0.36	0.19	1.00	1.00
cs3	original	6575	6571	2815	1728
	LRU-SP	4394	3548	1903	1733
	ratio	0.67	0.54	0.68	1.00
cs2	original	11785	11762	11717	11647
	LRU-SP	9680	9091	7650	5597
	ratio	0.82	0.77	0.65	0.48
gli	original	10435	10321	9720	7508
	LRU-SP	8870	8308	7120	6275
	ratio	0.85	0.81	0.73	0.84
ldk	original	5395	5389	5397	5390
	LRU-SP	5011	4760	4385	3898
	ratio	0.93	0.88	0.81	0.72
pjn	original	7166	6738	5897	5257
	LRU-SP	5800	5635	5334	4993
	ratio	0.81	0.84	0.90	0.95
sort	original	14670	14671	14639	14520
	LRU-SP	12462	11884	10400	9460
	ratio	0.85	0.81	0.71	0.65

Table 6: The numbers of block I/Os with/without application-controlled cache. The numbers are average of five runs, variances are less than 2% with one exception which is less than 3%.