Application-Controlled File Caching Policies

Pei Cao, Edward W. Felten, and Kai Li

Department of Computer Science
Princeton University
Princeton, NJ 08544 USA

Abstract

We consider how to improve the performance of file caching by allowing user-level control over file cache replacement decisions. We use two-level cache management: the kernel allocates physical pages to individual applications (*allocation*), and each application is responsible for deciding how to use its physical pages (*replacement*). Previous work on two-level memory management has focused on replacement, largely ignoring allocation.

The main contribution of this paper is our solution to the allocation problem. Our solution allows processes to manage their own cache blocks, while at the same time maintains the dynamic allocation of cache blocks among processes. Our solution makes sure that good user-level policies can improve the file cache hit ratios of the entire system over the existing replacement approach. We evaluate our scheme by trace-based simulation, demonstrating that it leads to significant improvements in hit ratios for a variety of applications.

1 Introduction

File caching is a widely used technique in today's file system implementations. Since CPU speed and memory density have improved dramatically in the last decade while disk access latency has improved slowly, file caching has become increasingly important. One major challenge in file caching is to provide high cache hit ratio.

This paper studies an application-controlled file caching approach that allows each user process to use an application-tailored cache replacement policy instead of always using a global Least-Recently-Used (LRU) policy. Some applications have special knowledge about their file access patterns which can be used to make intelligent cache replacement decisions. For example, if an application knows which blocks it needs and which it does not, it can keep the former in cache and reduce its cache miss ratio.

Traditionally such applications buffer file data in user address space as a way of controlling replacement. However, since the kernel tries to cache file data as well, this approach leads to double buffering, which wastes space. Furthermore, this approach does not give applications real control because the virtual memory system can still page out data in the user address space. Hence we need another way to let applications control replacement.

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

This paper describes a scheme that achieves this goal. Our approach, called "two-level block replacement", splits the responsibilities of allocation and replacement between kernel and user level. A key element in this scheme is a sound allocation policy for the kernel, which is discussed in section 3. This allocation policy guarantees that an application-tailored replacement policy can improve the overall file system performance and that a foolish replacement policy in one application will not degrade the file cache hit ratios of other processes.

We have evaluated our allocation policy using trace-driven simulation. In our simulations, we used several file access traces that we collected on a DEC 5000/200 workstation running the Ultrix operating system, and the Sprite traces from University of California at Berkeley. We have simu-

lated our allocation policy and various replacement policies for individual application processes. The simulations show that an application-tailored replacement policy can reduce an application's file cache miss ratio up to 100%, over the global LRU policy. In addition, in a multiprocess environment, the combination of our allocation policy and application-tailored replacement policies can reduce the overall file cache miss ratios, over the traditional global file caching approach, by up to 50%.

2 User Level File Caching

Our goal is to allow user-level control over cache replacement policy. In many cases, the application has better knowledge about its future file accesses than the kernel has. User level control of cache replacement enables the application to use its better knowledge to improve the hit ratio.

Despite the advantages of application control, we cannot simply move all responsibility for cache management to the user level. In a multiprogrammed system, the kernel serves a valuable function: managing the allocation of resources between users to guarantee the performance of the entire system.

2.1 Two-Level Replacement

We propose a scheme for file caching that splits responsibility between the kernel and user levels. The kernel is responsible for allocating cache blocks to processes. Each user process is free to control the replacement strategy on its share of cache blocks; if it chooses not to exercise this choice, the kernel applies a default policy (LRU). We call our approach *two-level cache block replacement.*

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

The interactions between kernel and manager processes are the following: On a cache miss, the kernel finds a candidate block to replace, based on its global replacement policy (step 1 in Figure 1). The kernel then identifies the manager process of 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 is not cooperative, then the kernel simply replaces the candidate block.)

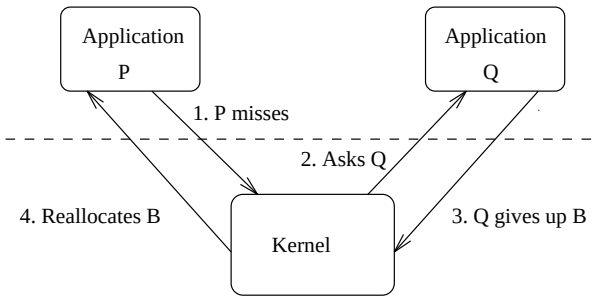

Figure 1: Interaction between kernel and user processes in two-level replacement: (1) P misses; (2) kernel consults Q for replacement; (3) Q decides to give up page B; (4) kernel reallocates B to P.

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

2.2 Kernel Allocation Policy

The kernel allocation policy is the most critical part of two-level replacement. To obtain best performance, it is known that allocation should follow the *dynamic partition* principle [11]: each process should be allocated a number of cache blocks that varies dynamically in accordance with its working set size. Experience has shown that global LRU or its approximations perform relatively well; they approximate the dynamic partition principle or tend to follow processes' working set sizes.

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

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

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

2. *A foolish decision by one process never causes another process to suffer more misses.* Of course, we cannot prevent a process from discarding its valuable pages. However, we must ensure that an errant or malicious process cannot hurt the performance of others.

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

The main contribution of this paper is to propose and evaluate an allocation policy that satisfies these design principles.

3 An Allocation Policy

We will describe our allocation policy in an evolutionary fashion. We will start with a simple but flawed policy, and then diagnose and fix two problems with it. The result will be a fully satisfactory allocation policy.

3.1 First Try

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

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

The left side of Figure 2 shows an example. Two processes, P and Q, share a file cache with four blocks. Process P uses blocks A and B; process Q uses blocks W, X, Y, and Z. The reference stream is Y, Z, A, B.

The top line shows the initial LRU list, at time t_0. The first reference, to Y at time t_1, causes a replacement. The kernel consults its LRU list and suggests A for replacement. A's owner, process P, overrules the kernel. It decides to replace B, hoping to save the next miss to A. The LRU list is now as shown at time t_2. The next reference, to Z at time t_3, causes another replacement. Again, A is chosen as a candidate. This time P has no other blocks in the cache and hence must give up A. The LRU list is now as shown at time t_4. At this point, the next two references, to A and B, both miss.

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

3.2 Swapping Position

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

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

This fixes the problem with above example as in Figure 2. The right side of the figure shows what happens under the new policy. On the first replacement, A moves to the head of the LRU list before B is replaced. The result is that A is still in the cache when it is referenced. Process P is no longer hurt by its wise choice.

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

Unfortunately, this scheme does not guard against foolish choices made by user processes. This is illustrated by the left side of Figure 3. Processes

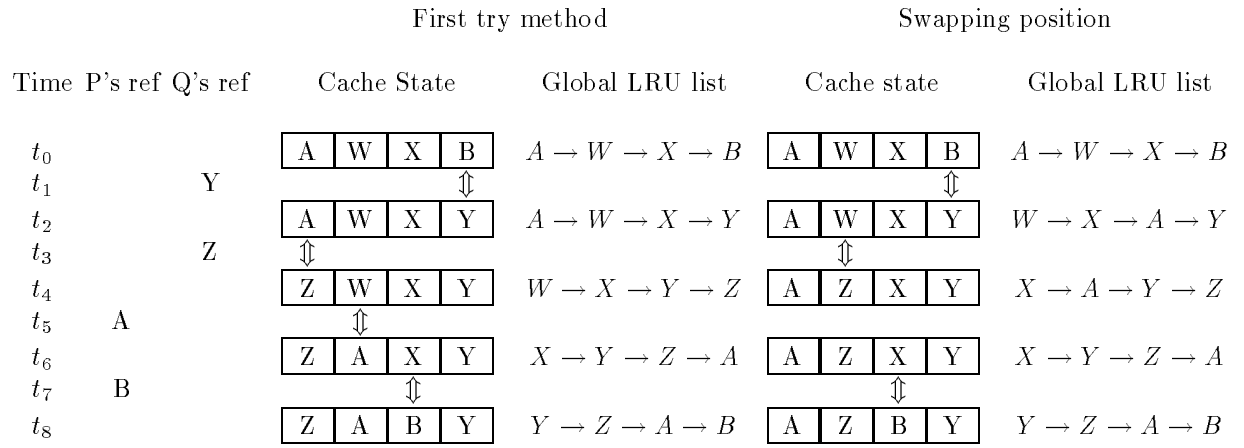

Figure 2: This example shows what's wrong with the first try and how to fix it with the swapping position mechanism.

P and Q share a three-block cache. The top line shows the initial LRU list at time t_0; the reference stream is Z, Y, A. The first reference, to Z at time t_1, causes a replacement. The kernel suggests X for replacement. Now suppose process Q makes exactly the wrong choice: it decides to replace Y. After swapping X and Y in the LRU list, the kernel replaces Y, leading to the LRU list as shown at time t_2. The second reference, to Y at time t_3, misses. The kernel suggests A for replacement, and process P cannot overrule because it has no alternative to suggest. Thus A is replaced, leading to the LRU list as shown at time t_4. The third reference, to A at time t_5, misses.

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

3.3 Place-Holders

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

A place-holder is a record that refers to a page. It records which block would have occupied that page under the global LRU policy. Suppose the kernel suggests A for replacement, and the user process overrules it and decides to replace B instead. In addition to swapping the positions of A and B in the LRU list, the kernel also builds a place-holder for B to point to A's page. If B is later referenced before A, A's page can be confiscated immediately. This allows the cache state to recover from the user process's mistake.

The right side of Figure 3 illustrates how place-holders work. The top line shows the initial LRU list at time t_0. The first reference, to Z at time t_1, misses. Block X is chosen as a candidate for replacement, but process Q (foolishly) overrules the kernel and chooses Y for replacement. X and Y are swapped in the LRU list, and Y is replaced. At this point, a place-holder is created, denoting the fact that the block occupied by X would have contained Y under global LRU.

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

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

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

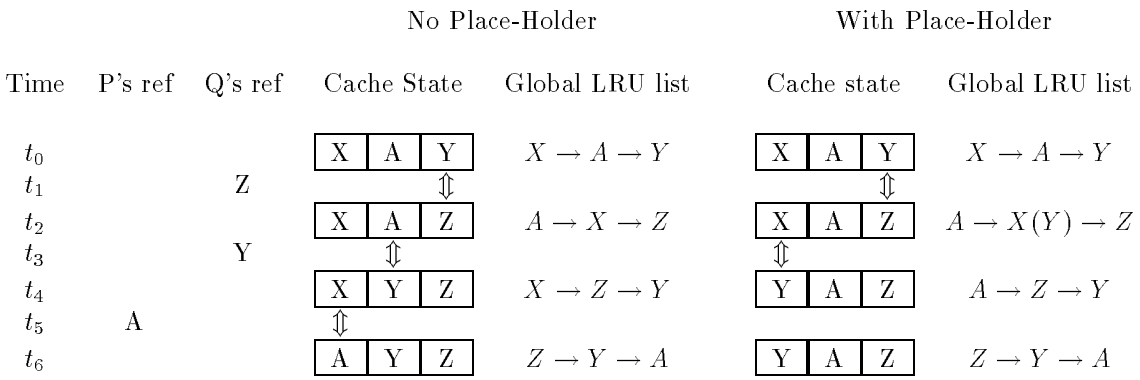

Figure 3: This example shows why place-holders are necessary.

3.4 Our Allocation Scheme

Combining above two fixes, here is our full allocation scheme: If a reference to cache block b hits, then b is moved to the head of the global LRU list, and the place-holder pointing to b (if there is one) is deleted. If the reference misses, then there are two cases. In the first case, there is a place-holder for b, pointing to t; in this case t is replaced and its page is given to b. (If t is dirty, it is written to disk.)

In the second case, there is no place-holder for b. In this case, the kernel finds the block at the end of the LRU list. Say that block c, belonging to process P, is at the end of the LRU list. The kernel consults P to choose a block to replace. (The kernel suggests replacing c.) Say that P's choice is to replace block x. The kernel then swaps x and c in the LRU list. If there is place-holder pointing to x, it is changed to point to c; otherwise a place-holder is built for x, pointing to c. Finally, x's page is given to b. (x is written to disk if it is dirty.)

We can prove that this algorithm satisfies all three of our design principles. (A detailed formal proof appears in [4].) Our scheme has the property that it never asks a process to replace a block for another process more often than global LRU. In other words, whenever a process is asked to give up a block for another process, it would have already given up that block under global LRU.

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

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

In summary, our framework for incorporating user level control into replacement policy is: consulting user processes at the time of replacement; swapping the positions of the block chosen by the global policy and the block chosen by the user process in the LRU list; and building "place-holder" records to detect and recover from user mistakes. This framework is also applicable to various policies that approximate LRU, such as FIFO with second chance, and two-hand clock[16].

4 Design Issues

This section addresses various aspects of our scheme, including possible implementation mechanisms, treatment of shared files and interaction with prefetching.

4.1 User-Kernel Interaction

There are several ways to implement two-level replacement, trading off generality and flexibility versus performance.

The simplest implementation is to allow each user process to give the kernel hints. For example, a user process can tell the kernel which blocks it no longer needs, or which blocks are less important than others. Or it can tell the kernel its access pattern for some file (sequential, random, etc). The

kernel can then make replacement decisions for the user process using these hints.

Alternatively, a fixed set of replacement policies can be implemented in the kernel and the user process can choose from this menu. Examples of such replacement policies include: LRU with relative weights, MRU (most recently used), LRU-K[21], etc.

For full flexibility, the kernel can make an upcall to the manager process every time a replacement decision is needed, as in [18].

Similarly, each manager process can maintain a list of "free" blocks, and the kernel can take blocks off the list when it needs them. The manager would be awakened both periodically and when its free-list falls below an agreed-upon low-water mark. This is similar to what is implemented in [25].

Combinations of these schemes are possible too. For example, the kernel can implement some common policies, and rely on upcalls for applications that do not want to use the common policies. In short, all these implementations are possible for our two-level scheme. We are still investigating which is best.

4.2 Shared Files

As discussed in Section 2.1, concurrently shared files are handled in one of two ways. If all of the sharing processes agree to designate a single process as manager for the shared file, then the kernel allows this. However, if the sharing processes fail to agree, management reverts to the kernel and the default global LRU policy is used.

4.3 Prefetching

Under two-level replacement, prefetches could be treated in the same way as in most current file systems: as ordinary asynchronous reads.

Most file systems do some kind of prefetching. They either detect sequential access patterns and prefetch the next block[17], or do cluster I/O[19].

Recent research has explored how to prefetch much more aggressively. In this case, a significant resource allocation problem arises — how much of the available memory should the system allocate for prefetching? Allocating too little space diminishes the value of prefetching, while allocating too much hurts the performance of non-prefetch accesses. The prefetching system must decide how aggressively to prefetch.

Our techniques do not address this problem, nor do they make it worse. The kernel prefetching code would still be responsible for deciding how aggressively to prefetch. We would simply treat the prefetcher as another process competing for memory in the file cache. However, since the prefetcher would be trusted to decide how much memory to use, our allocation code would provide it with a fresh page whenever it wanted one.

Recent research on prefetching focuses on obtaining information about future file references[22]. This information might be as valuable to the replacement code as it is to the prefetcher, as we discuss in the next section. Thus, adding prefetching may well make the allocator's job easier rather than harder.

To facilitate the use of a sophisticated prefetcher, there can be more interaction between the allocator and the prefetcher. For example, the allocator could inform the prefetcher about the current demand for cache blocks; the prefetcher could voluntarily free cache blocks when it realized some prefetched blocks were no longer useful, etc. The details are beyond the scope of this paper.

5 Simulation

We used trace-driven simulation to evaluate two-level replacement. In our simulations the user-level managers used a general replacement strategy that takes advantage of knowledge about applications' file references. Two sets of traces were used to evaluate the scheme.

5.1 Simulated Application Policies

Our two-level block replacement enables each user process to use its own replacement policy. This solves the problem for those sophisticated applications that know exactly what replacement policy they want. However, for less sophisticated applications, is there anything better than local LRU? The answer is yes, because it is often easy to obtain knowledge about an application's file accesses, and such knowledge can be used in replacement policy.

Knowledge about file accesses can often be obtained through general heuristics, or from the compiler or application writer. Here are some examples:

- Files are mostly accessed sequentially; the suffix of a file name can be used to guess the usage pattern of a file: ".o" files are mostly accessed in certain sequences, ".ps" files are accessed sequentially and probably do not need to be cached, etc.

- Compilers may be able to detect whether there is any `lseek` call to a file; if there is none, then it is very likely that the file is accessed sequentially. Compilers can also generate the list of future file accesses in some cases; for example, current work on TIP (Transparent Informed Prefetching)[22] is directly applicable.

- The programmer can give hints about the access pattern for a file: sequential, with a stride, random, etc.

When these techniques give the exact sequence of future references, the manager process can apply the offline optimal policy RMIN: replace the block whose next reference is farthest in the future. Often, however, only incomplete knowledge about the future reference stream is known. For example, it might be known that each file is accessed sequentially, but there might be no information on the relative ordering between accesses to different files. RMIN is not directly applicable in these cases. However, the principle of RMIN still applies.

We propose the following replacement policy to exploit partial knowledge of the future file access sequence: when the kernel suggests a candidate replacement block to the manager process,

1. find all blocks whose next references are definitely (or with high probability) after the next reference to the candidate block;

2. if there is no such block, replace the candidate block;

3. else, choose the block whose reference is farthest from the next reference of the candidate block.

Depending on the implementation of two-level replacement, this policy may be implemented in the kernel or in a run-time I/O library. Either way, the programmer or compiler needs to predict future sequences of file references.

This strategy can be applied to common file reference patterns. For general applications, common file access patterns include:

- *sequential*: Most files are accessed sequentially most of the time;

- *file-specific sequences*: some files are mostly accessed in one of a few sequences. For example, object files are associated with two sequences: 1) sequential; 2) first symbol table, then text and data (used in link editing);

- *filter*: many applications access files one by one in the order of their names in the command line, and access each file sequentially from beginning to end. General filter utilities such as grep are representative of such applications;

- *same-order*: a file or a group of files are repeatedly accessed in the same order. For example, if "*" (for file name expansion) appears more than once in a shell script, it usually leads to such an access pattern; and

- *access-once*: many programs do not reread or rewrite file data that they have already accessed.

Applying our general replacement strategy, we can determine replacement policies for applications with these specific access patterns. Suppose the kernel suggests a block A, of file F, to be replaced. For *sequential* or *file-specific*, the block of F that will be referenced farthest in the future is chosen for replacement; for *filter*, the sequence of future references are known exactly, and RMIN can be applied; for *same-order*, the most recently accessed block can be replaced; and for *access-once*, any block of which the process has referenced all the data can be replaced.

5.2 Simulation Environment

We used trace-driven simulations to do a preliminary evaluation of our ideas. Our traces are from two sources. We collected the first set ourselves, tracing various applications running on a DEC 5000/200 workstation. The other set is from the Sprite file system traces from University of California at Berkeley[2].

We built a trace-driven simulator to simulate the behavior of the file cache under various replacement policies[1]. In our simulation we only considered accesses to regular files — accesses to directories were ignored for simplicity, the justification being that file systems often have a separate caching mechanism for directory entries. We also assume that the file system has a fixed size file cache[2], with a block size of 8K.

We validated our simulator using Ultrix traces. Our machine has a 1.6MB file cache. We can measure the actual number of read misses using the Ultrix "time" command. Our simulation results were

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

[2] That is, we do not simulate the dynamic reallocation of physical memory between virtual memory and file system that happens in some systems.

within 3% of the real result except for link-editing, for which the simulator predicted 7% fewer misses. This is because the simulator ignores directory operations, which are more common in the link-editing application.

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

5.3 Results for Ultrix Traces

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

Traces were gathered for three application programs, both when they were running separately and when they were run concurrently. The applications are:

- *Postgres*: Postgres is a relational database system developed at University of California at Berkeley[28]. We used version 4.1. We traced the system running a benchmark from the University of Wisconsin, which is included in the release package. The benchmark takes about fifteen minutes on our workstation.
- *cscope*: cscope is an interactive tool for examining C sources. It first builds a database of all source files, then uses the database to answer the user's queries, such as locating all the places a function is called. We traced cscope when it was being used to examine the source code for our kernel. The trace recorded four queries, taking about two minutes.
- *link-editing*: The Ultrix linker is known as being I/O-bound. We collected the I/O traces when link-editing an operating system kernel twice, taking about six minutes. We also collected traces when linking some programs with the X11 library.

Single Application Traces First we'd like to see how introducing application control can improve each application's caching performance:

- *Postgres*: it is often hard to predict future file accesses in database systems. To see whether

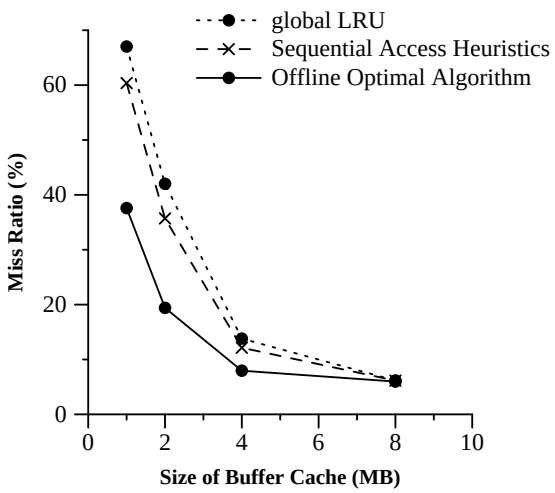

Figure 4: Performance for Postgres

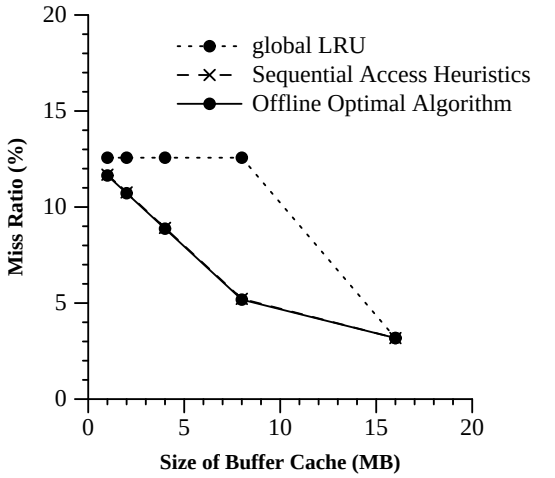

Figure 5: Performance for cscope

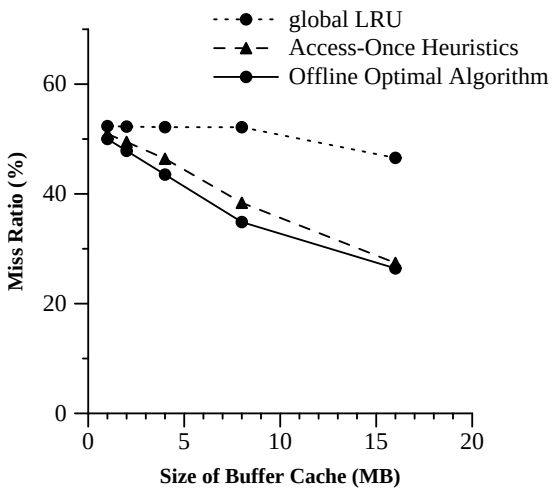

Figure 6: Performance for Linking Kernel

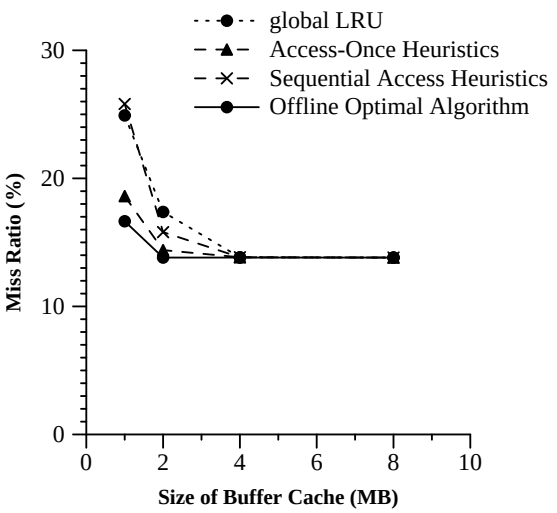

Figure 7: Performance for Linking with X11

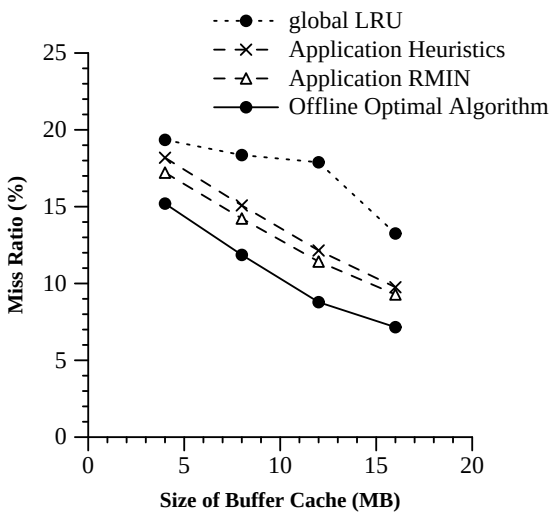

Figure 8: Performance for a Multi-Process Workload

user-level heuristics may reduce miss ratio, we tried the policy for *sequential* access pattern. Indeed, the miss ratio is reduced (Figure 4). We think that the designer of the database system can certainly give a better user-level policy, thus further improving the hit ratio.

- *cscope*: cscope actually has a very simple access pattern. It reads the database file sequentially from beginning to end to answer each query. The database file used in our trace is about 10MB. For caches smaller than 10MB, LRU is useless. The reason that the miss ratio is only 12.5% is that the size of file accesses is 1KB, while the file block size is 8KB. However, if we apply the right user-level policy (noticing that the access pattern is *same-order*), the miss ratio is reduced significantly (Figure 5).

- *link-editing*: the linker in our system makes a lot of small file accesses. It doesn't fit the *sequential* access pattern. However, it is *read-once*. Even though the linker is run twice in our traces, during each run its user level policy can still be that of *read-once*. The result is shown in Figure 6. For linking with X11 library, we tried both the policy for *sequential* and the policy for *read-once* at user-level (Figure 7). *read-once* seems to be the right policy. (Note that this trace is small and a 4MB cache is actually enough for it.)

Multi-Process Traces Having seen that appropriate user-level policies can really improve the cache performance of individual applications, we would like to see how our scheme performs in a multi-process environment.

We collected traces when the three applications (Postgres, cscope, linking the kernel) are run concurrently. In this trace, we simulated each application running its own user-level policy as discussed above. The result is shown in Fig.8. Since the applications' user-level policies are not optimal, we also simulate the case of each application using an offline optimal algorithm as its user-level policy. This yields the curve directly above RMIN.

As can be seen, our scheme, coupled with appropriate user level policies, can improve the hit ratio for multiprocess workloads.

We also performed an experiment to measure the benefit of using place-holders. We collected a trace of the Postgres and kernel-linking applications running concurrently, and simulated the miss ratio of Postgres when kernel-linking makes the worst possible replacement choices and Postgres simply follows LRU. We simulated our full allocation algorithm and our algorithm without place-holders. Figure 9 shows the result. Without place-holders, Postgres is noticeably hurt by the other application's bad replacement decisions. (With place-holders, Postgres has the same miss ratio as under global LRU.)

5.4 Results for Sprite Traces

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

We focused on the performance of client caching.

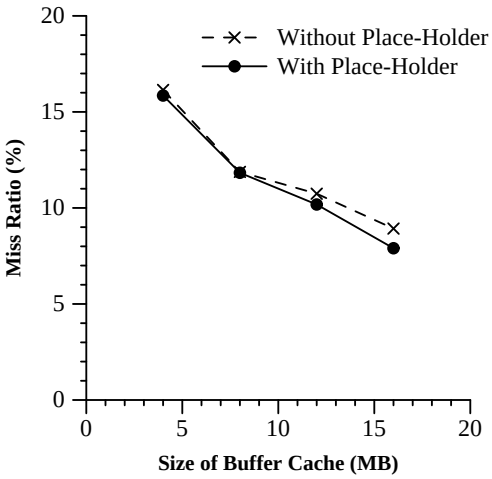

Figure 9: Benefit of Using Place-Holders

In a system with a slow network (e.g. ethernet), client caching performance determines the file system performance on each workstation. Furthermore we ignored kernel file activities in these traces, because Sprite's VM system swaps to remote files. In our simulation we set the client cache size to be 7MB, which is the average file cache size reported in [2].

These traces do not contain process-ID information, so we cannot simulate application-specific policies as with Ultrix traces. However, since most file accesses are sequential [2], the *sequential* heuristic can be used. Figure 10 shows average miss ratios for global LRU, sequential heuristic and optimal replacement. Average cold-start (compulsory) miss ratios are also shown.

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

Overall, these results show that two-level replacement is a promising scheme to enable application control of replacement and to improve the hit ratio of the file buffer cache. We believe that two-level replacement should be implemented in future file systems.

6 Related Work

There have been many studies on caching in file systems (e.g. [24, 5, 23, 3, 13, 20]), but these investigations were not primarily concerned with cache replacement policies. The database community has long studied buffer replacement policies[26, 8, 21],

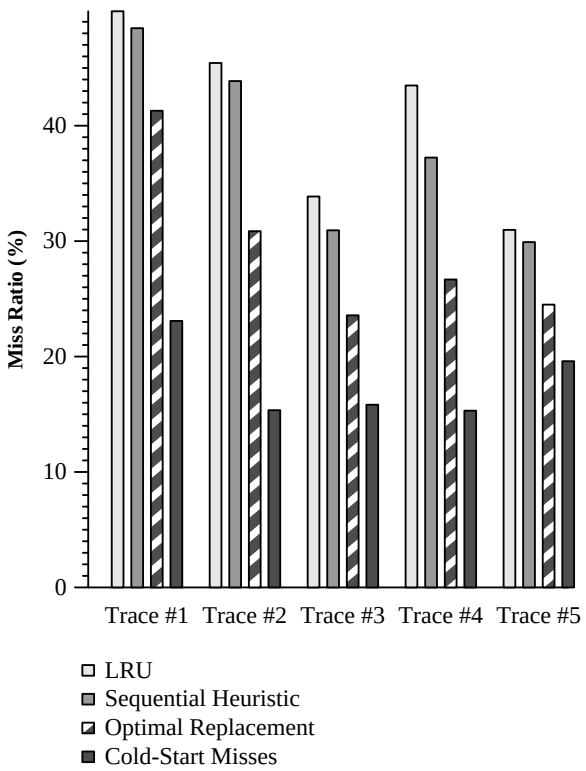

Figure 10: Averaged Results from Sprite Traces

but existing file systems do not support them. Although user scripts were introduced in caching in a disconnected environment[14], they were used to tell the file system which files should be cached (on disk) when disconnection occurs. In most of these systems, the underlying replacement policy is still LRU or an approximation to it.

In the past few years, there has been a stream of research papers on mechanisms to implement virtual memory paging at user level. The external pager in Mach [29] and V [6] allows users to implement paging between local memory and secondary storage, but it does not allow users to control the page replacement *policy*. Several studies [18, 25, 12, 15] proposed extensions to the external pager or improved mechanisms to allow users to control page replacement policy. These schemes do not provide resource allocation policies that satisfy our design principles to guarantee replacement performance. Furthermore, they are not concerned with file caching.

Previous research on user-level virtual memory page replacement policies [1, 9, 12, 15, 27] shows that application-tailored replacement policies can improve performance significantly. With certain modifications, these user-level policies might be used as user-level file caching policies in our

two-level replacement. Recent work on prefetching [22, 10, 7] can be directly applied in user-level file caching policies using our general replacement strategy. These systems can take advantage of the properties of our allocation policy to guarantee performance of the entire system.

7 Conclusions

This paper has proposed a two-level replacement scheme for file cache management, its kernel policy for cache block allocation, and several user-level replacement policies. We evaluated these policies using trace-driven simulation.

Our kernel allocation policy for the two-level replacement method guarantees performance improvements over the traditional global LRU file caching approach. Our method guarantees that processes that are unwilling or unable to predict their file access patterns will perform at least as well as they did under the traditional global LRU policy. Our method also guarantees that a process that mis-predicts its file access patterns cannot cause other processes to suffer more misses. Our key contribution is the *guarantee* that a good user-level policy will improve the file cache hit ratios of the entire system.

We proposed several user-level policies for common file access patterns. Our trace-driven simulation shows that they can improve file cache hit ratios significantly. Our simulation of a multiprogrammed workload confirms that two-level replacement indeed improves the file cache hit ratios of the entire system.

We believe that the kernel allocation policy proposed in this paper can also be applied to other instances of two-level management of storage resources. For example, with small modifications, it can be applied to user-level virtual memory management.

Although the kernel allocation policy guarantees performance improvement over the traditional global LRU replacement policy, there is still room for improvement. We plan to investigate these possible improvements and implement the two-level replacement method to evaluate our approach with various workloads.

Acknowledgments

We are grateful to our paper shepherd Keith Bostic and the USENIX program committee reviewers for their advice on improving this paper. Chris Maeda, Hugo Patterson and Daniel Stodolsky provided helpful comments. We benefited from discussions with Rafael Alonso, Matt Blaze, and Neal Young. Finally we'd like to thank the Sprite group at the University of California at Berkeley for collecting and distributing their file system traces.

[1] Rafael Alonso and Andrew W. Appel. An Advisor for Flexible Working Sets. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 153–162, May 1990.

[2] 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.

[3] Andrew Braunstein, Mark Riley, and John Wilkes. Improving the efficiency of UNIX file buffer caches. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 71–82, 1989.

[4] Pei Cao. Analysis of Two-Level Block Replacement. Working Paper, January 1994.

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

[6] David R. Cheriton. The Unified Management of Memory in the V Distributed System. Draft, 1988.

[7] 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.

[8] 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.

[9] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, June 1992.

[10] 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.

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

[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 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.

[14] 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.

[15] 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.

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

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

[18] 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.

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

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

[27] Indira Subramanian. Managing discardable pages with an external pager. In *Proceedings of the USENIX Mach Symposium*, pages 77–85, October 1990.

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

[29] M. Young, A. Tevanian, R. Rashid, D Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, pages 63–76, November 1987.

Author Information

Pei Cao graduated from TsingHua University in 1990 with a BS in Computer Science. She received her MS in Computer Science from Princeton University in 1992 and is currently a Ph.D. candidate. Her interests are in operating systems, storage management and parallel systems. Email address: pc@cs.princeton.edu.

Edward W. Felten received his Ph.D. degree from the University of Washington in 1993 and is currently Assistant Professor of Computer Science at Princeton University. His research interests include parallel and distributed computing, operating systems, and scientific computing. Email address: felten@cs.princeton.edu.

Kai Li received his Ph.D. degree from Yale University in 1986 and is currently an associate professor of the department of computer science, Princeton University. His research interests are in operating systems, computer architecture, fault tolerance, and parallel computing. He is an editor of the IEEE Transactions on Parallel and Distributed Systems, and a member of the editorial board of International Journal of Parallel Programming. Email address: li@cs.princeton.edu.