INFORMATION AND CONTROL IN FILE SYSTEM BUFFER MANAGEMENT

by

Nathan Christopher Burnett

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2006

ACKNOWLEDGMENTS

I would like to begin by thanking my advisers, Andrea and Remzi Arpaci-Dusseau. Had they not been at Wisconsin, I don't know who else I would have worked with. They have a unique combination of taking their research and their student's work very seriously, while not taking themselves too seriously. Among other things, they taught me how to ask the right questions when conducting research. Indeed it has been an honor and a privilege working with both of them.

I thank Jeff Naughton, Marv Solomon and Tehshik Yoon for serving on my thesis committee. They provided valuable advice and feedback. I thank Tehshik in particular for making it to my defense despite his illness that day so I could avoid delaying my graduation.

During my stay at Wisconsin, I have had the opportunity to collaborate with a number of talented graduate students including John Bent, Tim Denehy, Florentina Popovici, Vijayan Prabhakaran, Muthian Sivathanu, Todd Jones, James Nugent, Haryadi Gunawi, Nitin Agrawal, Lakshmi Bairavasundaram and Brian Forney. Their intellects have improved my work to a great degree and their individual personalities and friendships have made graduate school more enjoyable than it otherwise might have been.

The Computer Systems Lab consistently provides a reliable and modern computing environment for the research and teaching of the Computer Sciences Department. The CSL is particularly good at accommodating the sometimes esoteric needs of an operating systems research group. They truly provide a gold standard to which other IT groups should aspire. I also thank the Condor team for providing a seemingly unlimited number of compute cycles. Without Condor, Chapter 4 of this thesis would have taken many months longer than it did to complete.

When I leave Madison, I will miss most the friends I have made here. In addition to the people I've already mentioned, Erik Paulson, David Parter, Nathan Cunningham, William Annis, Eric

Oehler, Matt Lockett, Regan McKendry, Ken Ekern and Dr. Sam Pazicni have made Madison home. Erik and David in particular have provided immeasurable support and advice during my time at Wisconsin. Nathan, though an unlikely friend, has been one of my most loyal and beloved friends and I look forward to continuing our adventures in California. I also thank Glenn Jahns, Bob Mahr and the staff and patrons of the Shamrock Bar for providing my "third place" in Madison.

My mother, father and sister have been crucial to my success, not only in graduate school but throughout my life. Their high expectations and constant encouragement have been a powerful motivator and source of support throughout my life. Without them, none of this would have been possible.

Finally, I thank my partner, Flip, for his undying support and love. My triumphs have always been equally his triumphs and my disappointments, his disappointments. I hope I support him as he completes his own Ph.D. as well as he has supported me while I completed mine.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

By implementing file system caching within the operating system, applications are required to cede to the OS a degree of control over memory utilization and IO scheduling. This dissertation explores ways in which applications can rediscover *information* hidden by the file system buffer cache and reclaim some of the *control* ceded to it. We find that this can be achieved without a wholesale redesign of either the operating systems or applications concerned.

We present *Dust* a tool to automatically determine the buffer cache replacement policy of an operating system. We describe a cache-aware web server. Using the information gained through *Dust*, our cache-aware web server is able to infer the contents of the buffer cache. It uses this information to schedule web connections on an in-cache-first basis, improving throughput and response time.

Implicit information can be imprecise. To address this limitation, we modify Linux and NetBSD to expose a list of pages which are about to be evicted. This *explicit information* is always accurate. We present InfoReplace, a user library which observes that list and touches pages that should remain cached, allowing applications to transform the kernel policy into one of the application's choice.

Some applications, such as those that use write-ahead logging, require control over the order in which data is written to disk. We propose two new interfaces by which applications can express write ordering constraints to the operating system. *File system barriers* introduce the `barrier()` system call. The operating system guarantees that no write operations will be reordered across a barrier. *Asynchronous graphs* allows applications to specify ordering constraints on a per-write-operation basis. Both would be difficult to implement with only information.

Chapter 1

Introduction

1.1 Cache Management

On a typical system access time for data cached in main memory is several orders of magnitude lower than for data that is not cached, requiring a disk IO to access. Memory access times range from 1 to 20 clock cycles depending whether the access hits in the CPU caches, whereas the latency of a disk access ranges in the tens to hundreds of milliseconds [26]. Access to storage over a network is even slower. It is thus worthwhile to dedicate a portion of a system's main memory to caching file system data.

Since memory is a finite, and often constraining, resource that is shared among all applications running on a system, file system caching is normally implemented within the operating system. In this way, applications are provided with the benefits of caching transparently and the operating system serves as an arbitrator to determine how much of main memory will be used for caching and which data will be cached.

Implementing file system caching within the operating system requires applications to cede to the OS a degree of control over memory utilization and IO scheduling. The operating system, not the application, determines how much memory is devoted to file system caching. The operating system, not the application, determines what data is cached. The operating system, not the application, determines when updates to data are flushed to disk. Thankfully, most applications only suffer minimally from this loss of control.

Applications that are file system intensive, however, can suffer greatly due this lack of control [2, 63, 79]. Web servers, file servers and data base management systems are a few examples.

They each have knowledge about their workloads that the operating system lacks that can help optimize I/O. Web servers and file servers know the granularity at which requests can be rescheduled. Database management systems know what data they are likely to access in the near future. If this knowledge can be combined with the control that the operating system has over buffer management, there is an opportunity for better cooperation between the application and the OS.

The situation is especially notable for servers, since they are often the only large application running on a machine. If there is only one application running on a machine, it makes sense to give it the maximum amount of control over memory usage. At the same time, to enable application portability, and simplify application development, it is desirable to leverage the existing, standardized interfaces that modern operating systems provide and to continue to use those operating system facilities that adequately suit the applications in question.

For decades, operating systems have been designed to provide strong abstractions, such as network sockets and the open/close/read/write interface to the storage subsystem, to user-level applications. Abstractions provide a common framework for application development, enabling portability between platforms. They hide operating system data structures and prevent applications from seeing each other's data, providing security through isolation. They also hide the details of the underlying hardware, greatly simplifying application development.

These abstractions though, also hide a great deal of useful *information* from applications. First, user-level applications do not know what data of theirs is being cached, or even how much of it is being cached. Second, they have no way of knowing which of their data is likely to be evicted from the cache soon, and which can be expected to remain cached for a while. For applications whose performance is bounded by the speed at which they can access data from the file system, both pieces of information can be quite valuable.

In addition to hiding information, operating system abstractions also reduce the level of *control* available to applications. The file system cache not only stores data that has been read, it buffers updates to data. When an application updates data stored in the file system, that update is applied to a cached copy of the data. At some later point determined by the operating system, the on-disk copy of the data will be updated. Most applications don't care when this happens, as long as it happens eventually. Some, however, care a great deal for data integrity reasons. User-level applications normally have no control over, nor knowledge of when their data updates reach the disk. For applications with strong data-integrity requirements, such as database management systems and user-level file systems [37], this ordering is of critical importance.

One of the primary challenges in designing a caching system is the management of the available space. The cache has some finite amount of memory to manage; it must determine which data to keep in memory, and which to evict and when to evict it. The narrow interface that exists between the operating system and applications often makes it difficult to determine what choices will yield the greatest benefit to applications, both individually and collectively. Information available to the operating system to make cache management decisions is limited. In general, the operating system knows when data is accessed, at the time it is accessed, and nothing more. In a perfect world, the operating system would like to know the future access pattern of applications. Unfortunately, even when this knowledge exists within the application, it is difficult to get that information into the OS so it can be used. Some interfaces exist, such as `madvise()` on some systems, but these tend to be limited in that they are coarse grained and inconsistently implemented. For example, `madvise()`, lets an application give the OS *advice* such as whether to expect random or sequential accesses to a file (to assist in prefetching) and allows the application to say if a particular page will or will not be needed in the future. It does not, however, allow the application to assign relative priority to data pages. The difficulty in making good cache management decisions is evidenced by the enormous body of work on how to make such decisions effectively [13, 14, 15, 18, 21, 27, 29, 34, 41, 42, 44, 45, 49, 53, 60, 75, 78, 79].

1.2 Information and Control

This dissertation explores ways in which applications can rediscover *information* hidden by the file system buffer cache and reclaim some of the *control* ceded to it. We find that this can be achieved without a wholesale redesign of either the operating systems or applications concerned. Since one of our goals is to avoid large-scale changes to the operating system, we explore options for exposing information and gaining control in order of the degree of alterations to the operating system they require.

Some information, such as the contents of the buffer cache, can be discovered without any modifications to the operating system at all by leveraging *implicit information*. By its nature, implicit information is sometimes inaccurate. Despite this inherent inaccuracy, we will show that this information can still be leveraged for significant performance improvements.

In some situations, timely, accurate information is necessary. If by acting on incorrect information, the application might significantly degrade its own performance, then implicit information may be inadequate. In cases where implicit information isn't enough, operating systems can be modified to provide interfaces to expose *explicit information*. These modifications are surprisingly simple and unintrusive to the overall structure of the OS.

Implicit and explicit information both allow applications to exercise *implicit control* over file system caching. By knowing the policies the OS is using to manage the cache and the current state of the cache, an application can both alter its behavior to better suit the policies of the OS and game the operating system into changing its behavior to better fit the needs of the application.

Finally, sometimes it isn't the application that needs more information about the operating system, but the OS that needs more information about the needs of the application. When implicit control proves insufficient, it is possible to alter the OS to allow the application to apply *explicit control* over the operating systems behavior. Explicit control can be seen as the reverse of explicit information. Instead of moving information from the operating system into the application, we provide an interface to move information from the application into the operating system. Moving information into the operating system is necessary when the mechanisms that can act on that information exist only within the OS.

1.2.1 Implicit Information

Operating systems are large, complex pieces of software. Altering them is difficult, time consuming and error prone. Further, the operating systems we would most like to improve are those that are the most widely deployed. These systems have large developer communities and larger numbers of users. Adding new functionality to popular operating systems is difficult precisely because they are popular; a new feature must be accepted by a great many people before it will be integrated into the codebase. So it is desirable to avoid modifying the operating system.

By not modifying the operating system, we limit ourselves to acquiring information about the OS *implicitly*. That is, we obtain information by probing and observing the OS through the existing interface. For example, an application might issue a `read()` call to a particular piece of data. If the application measures the execution time of that system call, it can implicitly determine whether or not that data was cached at the time of the read. Chapter 2 describes *Dust*, a tool which uses this idea to determine the replacement policy that the buffer cache is using.

Once an application knows the replacement policy that the buffer cache is using, it can use that knowledge to predict the behavior of the cache. Chapter 2 also describes a *cache-aware web server* that using knowledge of cache behavior to predict which data are currently cached. Our cache-aware web server then uses those predictions for connection scheduling [17]. By servicing those requests asking for cached data first, the web server improves its utilization of the buffer cache and in turn, improves its own performance.

1.2.2 Explicit Information

Using only implicit techniques limits the accuracy of the information the application can obtain. For some optimizations, such as scheduling HTTP requests, having somewhat inaccurate information is acceptable. If the web server's cache prediction is occasionally wrong, it simply means that the scheduling decision will be slightly sub-optimal. On the whole the performance will still be better than cache-oblivious scheduling. A tiny amount of performance improvement is sacrificed to gain the portability benefits of using only implicit techniques.

If an application is using information for more aggressive optimizations, the consequences of inaccurate information could be severe. Chapter 3 presents a method by which applications can alter the buffer cache replacement policy. Since this technique involves issuing additional `read()` calls, it is critical that the application know which data is in the buffer cache and only issue extra reads to cached data. Some applications can predict their own IO patterns to a sufficient degree as to be able to determine a good caching strategy in advance, that is, before performing the IO. Database management systems are one such example [16]. If an application is able to alter the buffer cache's replacement policy, better performance can be achieved.

Most common caching policies base their decisions at least in part on how recently each buffer has been accessed. Accessing a piece of data increases its priority in the cache and reduces its chances of being evicted in the near future. Thus if the application knows that a particular portion of its data should remain cached, those data's priority can be raised by simply reading the data.

If an application uses this technique to manipulate the behavior of the cache, it could easily degrade performance rather than improve it if care isn't taken to avoid performing extra reads on data that has already been evicted from the cache. Incurring even a small number of extra disk IOs could outweigh the benefit of manipulating the cache. The missing component is, again, information. The application needs to know what data is still cached, so extra reads to uncached data can be avoided, and it needs to know what data is likely to be evicted soon, so it knows for what data extra read calls will be most productive. If the information the application has is inaccurate, extra disk traffic and bad performance will be the result.

Chapter 3 introduces interfaces by which applications can learn the current state of the buffer cache [5]. We provide an interface that exposes to the application a list of the next N file system pages that will be evicted to the cache, and an efficient measure of how quickly that list is changing. These new interfaces allow applications to perform cache usage optimizations without fear that the knowledge they have of the cache state is inaccurate. By only exposing information, rather than providing new mechanisms, we reduce the changes that need to be made to the operating system. Most importantly, we avoid perturbing any of the OS's pre-existing data structures, and thus reduce the likelihood of introducing new bugs or performance problems into the kernel.

1.2.3 Explicit Control

Applications using explicit interfaces have access to precise information through the additional interfaces provided. However, they still depend on the existing interfaces to exercise control over the operating systems behavior. Utilizing this type of implicit control relies on making assumptions about the behavior of the operating system and may also make implicit assumptions about the usage patterns of other applications running in the system. For instance, the example mentioned in the previous section assumes that issuing a read to a piece of data will increase that data's priority in the buffer cache. If the operating system implements First-In-First-Out cache replacement, that assumption doesn't hold and attempts at manipulating cache replacement will fail. In short, even with exact information, control is still approximate.

Chapter 4 examines a situation where precise control is required. For applications that make incremental updates to complex on-disk data structures, the order in which updates are written to the disk can be important. A common example is write-ahead logging in database management systems. For each transaction executed, the log entries describing the updates to the data must be written to the disk strictly before the actual data is updated.

Traditional means of controlling write-ordering either utilize synchronous IO calls such as `fsync()` which perform poorly [50], or use direct access to storage which has unsatisfactory implications for system management [28]. Chapter 4 proposes two new interfaces by which applications can express write ordering constraints to the operating system. The first interface, *file system barriers* allows the application to insert barriers into the write request stream. The operating system then guarantees that no write operations will be reordered across a barrier. The second, *asynchronous graphs* allows applications to specify for each write operation exactly which write operations, if any, must have their data committed to disk *before* the current one.

1.3 Contributions

The buffer cache provides a valuable service to applications. However, utilizing the buffer cache requires applications to cede to the operating system some control over memory usage and IO management. This dissertation shows that by examining the flow of information across the OS/application interface, operating systems and applications can be made to cooperate more effectively.

When possible, we use *implicit information*, information gained through the existing interface, and avoid modifying the operating system in any way. While the implicit approach has advantages in that it is easily ported between operating systems and requires no OS modifications, the information that can be acquired using implicit techniques is limited. Specifically, it can be imprecise. While this is sufficient for some optimizations, such as request scheduling based on cache residency, it is ill suited for situations where bad information can lead to performance degradation.

Some of the limitations of implicit information can be overcome using *explicit information*, information gained by an interface provided by the operating system. This technique provides information that is always accurate and thus can be relied on for more aggressive optimizations, such as those which use extra IO calls to manipulate the buffer cache. Using implicit information to manipulate OS behavior, however, requires that actions of the application have a predictable effect on the OS behavior. Cache replacement policies fit into this category since most policies react to the `read()` call by increasing the read buffers cache priority.

Information alone isn't always enough. When the operating system behavior that we need to modify is completely hidden from the application, such as the flushing of delayed writes, direct modification of the operating system is our last resort. This can be viewed either as providing *explicit control* to applications or as a reversal of the previous techniques. Rather than moving information from the operating system to the application, we move it from the application to the operating system by providing an interface for the application to inform the operating system what orderings are safe orderings in which to flush dirty buffers.

1.4 Organization

Chapter 2 describes *Dust*, a tool that uses high level assumptions about operating system buffer cache behavior, probes and observation to determine the buffer cache replacement algorithm, then uses that information to predict the state of the buffer cache dynamically. Those predictions are then given to a web server which uses them to schedule requests on an in-cache-first basis. In this way, the application moulds its own behavior to better fit the policies of the operating system. Chapter 3 presents interfaces by which cache state information is explicitly exposed to applications.

Applications can use that information to not only change their own behavior to fit that of the operating system, but alter the cache replacement policy to better fit their workload. Chapter 4 proposes two new interfaces, file system barriers and asynchronous graphs, by which application-level write-ordering requirements can be explicitly expressed to the operating system. Chapter 5 reviews previous related work and Chapter 6 concludes.

Chapter 2

Discovering Buffer Cache State with Implicit Information

2.1 Introduction

Operating systems are large, complex pieces of software. Changing them to suit a particular application is time consuming and error prone. Further, getting proposed changes adopted and integrated into an already popular OS isn't easy. It requires convincing others of the value and safety of the new code. In the case of an open-source system, a potentially large number of developers need to be convinced. In the case of a closed source system, major corporation needs to be convinced that adding the new code is in their best interests. For these reasons, it is advantageous to find ways to extend the system without actually modifying it.

Knowing what is currently in the buffer cache can be useful to applications that make heavy use of the file system. We will describe a storage server we have modified to serve requests for cached data first, thus improving throughput and response time. The primary challenge in using this sort of performance optimization is determining which data is currently cached. In this chapter we demonstrate that discovering this information is possible without any modification to the operating system whatsoever.

We observe that an application can model (or simulate) the state of the buffer cache if it knows the replacement policy used by the OS and can see most file accesses. The application can then use such a model to infer the current contents of the buffer cache and make application-level decisions based on that information.

Server applications typically dominate the use of the file system. Often, they are the only application running, apart from the various system maintenance daemons. Thus, these applications already see most of the file accesses on the system. If they also know the size of the buffer cache and the policy used to manage it, they would, by inference, have complete knowledge of the buffer cache contents.

Although the specific algorithms used to manage the buffer cache can significantly impact the performance of I/O-intensive applications [14, 34, 60], this knowledge is usually hidden from user processes. Currently, to determine the behavior of the buffer cache, implementors are forced to rely on available documentation, access to source code, or general knowledge of how buffer caches behave.

Rather than relying on these *ad hoc* methods, we propose the use of *fingerprinting* to automatically uncover characteristics of the OS buffer cache. This chapter describes *Dust*, a simple fingerprinting tool that is able to identify the buffer-cache replacement policy; specifically, we identify whether it uses initial access order, recency of access, frequency of access, or historical information.

Fingerprinting can be described as the use of microbenchmarking techniques to identify the algorithms and policies used by the system under test. The idea behind fingerprinting is to insert *probes* into the underlying system and to observe the resulting behavior through visible outputs. By carefully controlling the probes and matching the resulting output to the fingerprints of known algorithms, one can often identify the algorithm of the system under test. The key challenge is to inject probes to create distinctive fingerprints such that different algorithmic characteristics can be isolated.

There are several significant advantages to using fingerprints for automatically identifying internal algorithms. First, fingerprinting eliminates the need for a developer to obtain documentation or source code to understand the underlying system. Second, fingerprinting enables all programmers, not just those with sophisticated experience, to use algorithmic knowledge and thus improve performance. Third, fingerprinting can uncover bugs, or hidden complexities, in systems either under development or already deployed. Finally, fingerprinting can be used at run-time, allowing an adaptive application to modify its own behavior based on the characteristics of the underlying system.

We investigate a new use of algorithmic knowledge: its use in exposing the current contents of the OS buffer cache. Recent work has shown that I/O-intensive applications can improve their performance given information about the contents of the file cache [4, 71]; specifically, applications that can handle data from disk in a flexible order should first access those blocks in the buffer cache and then those on disk. However, current approaches suffer from one of two limitations: they either require changes to the underlying OS to export this information or cannot accurately identify the presence of small files in the buffer cache.

A dedicated web server can greatly benefit from knowing the contents of the buffer cache and servicing first those requests that will hit in the buffer cache. We have implemented a cache-aware web server based on the NeST storage appliance [9] and show that this web server improves both average response time and throughput.

This chapter describes the following contributions:

- We introduce *Dust*, a fingerprinting tool that automatically identifies cache replacement policies based upon how they prioritize between initial access order, recency of access, frequency of access, and historical information.

- We demonstrate through simulations that *Dust* can distinguish between a variety of replacement policies found in the literature: FIFO, LRU, LFU, Random, Clock, Segmented FIFO, 2Q, and LRU-K.

- We use our fingerprinting software to identify the replacement policies used in several operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, and Solaris 2.7.

- We show that by knowing the OS replacement policy, a cache-aware web server can first service those requests that can be satisfied within the OS buffer cache and thereby obtain substantial performance improvements.

2.2 Fingerprinting Methodology

We now describe *Dust*, our software for identifying the page replacement policy employed by an operating system. By manipulating how blocks are accessed, forcing evictions, and then observing which blocks are replaced, *Dust* can identify the parameters used by the page replacement policy and the corresponding algorithm.

Dust relies upon probes to infer the current state of the buffer cache. By measuring the time to read a byte within a file block, one can determine whether or not that block was previously in the buffer cache. Intuitively, if the probe is "slow", one infers that the block was previously on disk; if the probe is "fast", then one infers that the block was already in the cache.

For *Dust* to correctly distinguish between different replacement polices, we must first identify the file block attributes used by existing policies to select a victim block for replacement. From a search of the OS and database research literature and the documentation of existing operating systems, we have identified four attributes that are often used for replacement: the order of initial access to the block (*e.g.*, FIFO), the recency of accesses (*e.g.*, LRU), the frequency of accesses (*e.g.*, LFU) and historical accesses to blocks (*e.g.*, 2Q [29]). Thus, we can correctly identify the use of combinations of these four attributes within a replacement policy.

We note that some operating systems use replacement policies that consider attributes beyond what *Dust* considers. For example, some replacement policies consider whether or not pages are dirty [39], the size of the file the page is from, or replacement cost [21]. Further, replacement of pages can be performed on either a global or per process basis [35]. Finally, in real systems, not only are file pages cached, but file meta-data as well, and some systems prefer to evict pages from files whose meta-data is no longer cached. It is also possible that future replacement policies may use new attributes that we do not currently fingerprint. Although *Dust* cannot currently identify these parameters, we believe that the basic framework within *Dust* can be extended to do so.

Given our goal of identifying replacement policies, there are three primary components to *Dust*. First, the size of the buffer cache is measured with a simple microbenchmark; this value is used as input to the remaining steps. Second, the short-term replacement algorithm is fingerprinted, based upon initial access, recency of access, and frequency of access. Third, *Dust* determines whether or not long-term history is used by the replacement algorithm.

2.2.1 Microbenchmarking Buffer Cache Size

To manipulate the state of the buffer cache and interpret its contents, *Dust* must first know the *size* of the buffer cache. Since this information is not readily available through a common interface on most systems, *Dust* contains a simple microbenchmark. The algorithm is shown in Figure 2.1. *Dust* accesses progressively larger amounts of file data until it notices that some blocks no longer fit the cache. For each increase in the tested size, there are two steps. In the first step, *Dust* touches the file blocks up through the newly increased size to fetch them into the buffer cache. In the second step, *Dust* probes each block again, measuring the time per probe to verify if the block is still in the cache. This technique is similar to the technique used to determine available memory in NOW-Sort [6].

There are two important features of this approach. First, by probing *every* file block in the second step, this algorithm is independent of the replacement policy used to manage the buffer cache. Second, this algorithm works even when the buffer cache is integrated with the virtual memory system, assuming that *Dust* uses little memory and the buffer cache is able to grow to its maximum size. Further, as we will show, our fingerprinting algorithm is robust to slight inaccuracies in our estimation of the buffer cache size.

2.2.2 Fingerprinting Replacement Attributes

Once the buffer cache size is known, *Dust* determines the attributes of file blocks that are used by the OS short-term replacement policy. This fingerprinting stage involves three simple steps. First, *Dust* reads file blocks into the buffer cache while simultaneously controlling the replacement attributes of each block (*e.g.*, by accessing blocks in different initial access, recency, and frequency orders). Second, *Dust* forces some of these blocks to be evicted from the buffer cache by accessing additional file data. Finally, the contents of the buffer cache are inferred by probing random sets of

```
fd = open("some_huge_file", O_RDONLY, 0);
mean = 0;

for (i = min; i < max; i+=blocksize) {
  for (j = 0; j < i; j +=blocksize) {
    read(fd, c, blocksize);
  }

  gettimeofday(&tp1,NULL);
  lseek(fd, 0, SEEK_SET);
  for (j = 0; j < i; j +=blocksize) {
    read(fd, c, blocksize);
  }
  lseek(fd, 0, SEEK_SET);
  gettimeofday(&tp2,NULL);
  elapsed = timedif(&tp1, &tp2);
  if( elapsed > mean*10) {
    slowcount++;
  }
  mean = (mean*samples + elapsed)/(++samples);
  if (slowcount > 5) {
    fprintf(stderr,"Effective Cache Size is: %ld\n", i*blocksize);
    exit(0);
  }
}
close(fd);
```

Figure 2.1 **Psuedocode for cache size algorithm.** The algorithm opens a file that is known to be larger than the buffer cache. On each iteration, a larger section of the file is read twice. When the measured bandwidth drops, the cache size has been found.

```
stripe_size = cache_size/10;
leftseek = 0;
freqleft = 1;
rightseek = cache_size/2;
freqright = 10;

open(filename, O_RDONLY, 0);

/* an initial scan of the file to set a FIFO distribution
   that differs from the LRU distribution */
lseek(fd,0,SEEK_SET);
for (i = 0; i < cache_size; i += READSIZE) {
  read(fd, c, READSIZE);
}
lseek(fd, 0, SEEK_SET);

/* set up the frequency/recency distribution */
for (i = 0; i < cache_size/2; i+= stripe_size) {
  /* do the left side reads */
  for (j = 0; j < freqleft; j++) {
    pos =  lseek(fd, leftseek, SEEK_SET);
    for (k = 0; k < stripe_size; k += READSIZE) {
      read(fd, c, READSIZE);
    }
  }
  leftseek += stripe_size;
  freqleft++;

  /* do the right side reads */
  for (j = 0; j < freqright; j++) {
    pos = lseek(fd, rightseek, SEEK_SET);
    for (k = 0; k < stripe_size; k += READSIZE) {
      read(fd, c, READSIZE);
    }
  }
  rightseek += stripe_size;
  freqright--;
}
```

Figure 2.2 **Dust short-term attribute setting algorithm.** This algorithm performs a series of reads to set the initial access order, access recency and access frequency of each block in the buffer cache.

```
  /* fill the cache and cause evictions */
  page_size = getpagesize();
  scansize = ((cache_size - size) + (cache_size * 0.5));
  for (i = 1; i <= (scansize/page_size); i++) {
    pos = lseek(fd, size + (i * page_size), SEEK_SET);
    for (j = 0; j < freqleft * 1.5; j++) {
      read(fd, c, 1);
      lseek(fd, -1, SEEK_CUR);
    }
  }

  /* sample parts of the file to determine the cache state */
  newpos = ((double)rand()/RAND_MAX)*(size/20);
  pos = lseek(fd, newpos, SEEK_SET);
  for (i = 0; i < 20; i++) {
    gettimeofday(&tp1, NULL);
    read(fd,c,1);

    gettimeofday(&tp2, NULL);
    etime = timedif(&tp1, &tp2);
    printf("%d %ld\n", pos, etime);
    pos = lseek(fd, size/20, SEEK_CUR);
  }
  exit(0);
}
```

Figure 2.3 **Dust eviction and cache probing algorithm.** This algorithm evicts half of the data in the buffer cache by reading in the appropriate amount of new data. The new data is read multiple times to ensure eviction in the case of a frequency based replacement policy. The algorithm then issues probe reads to determine which of the original data is still cached.

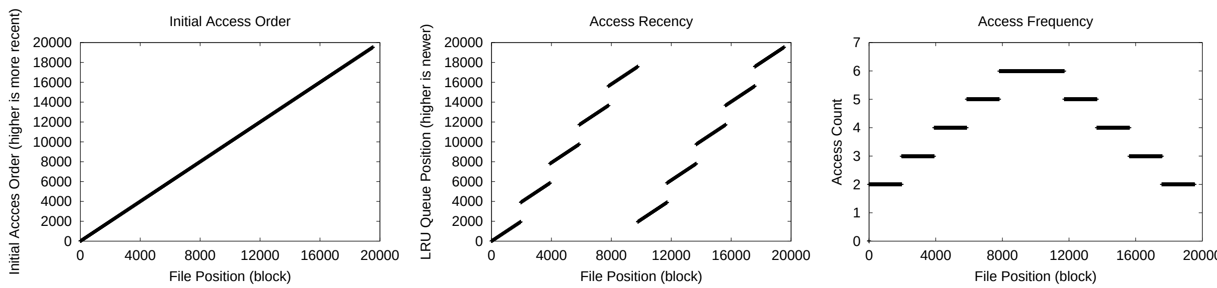


Figure 2.4 **Short-Term Attributes of Blocks.** The three graphs show the priority of each block within the test region according to the three metrics: order of initial access, recency of access, and frequency of access. The x-axis indicates the block number within the file forming the test region. The y-axes indicates the initial accesses order (left), recency of access (center) and frequency of access (right).

blocks; the cache state of these file blocks is then plotted to illustrate the replacement policy. We now describe each of these three steps in detail.

2.2.2.1 Configuring Attributes

The first step moves the buffer cache into a known and well-controlled state – both the data blocks that are resident and the initial access, recency, and frequency attributes of each resident block. This control is imposed by performing a pattern of reads over blocks within a single file; we refer to these blocks as the *test region*. To ensure that all of this data is resident, the size of this test region is set slightly smaller than the estimate of the buffer cache size (precisely, we use only 90% of the estimated cache size and adjust the size such that each of ten stripes discussed below are page aligned).

Controlling the initial access parameter of each block allows *Dust* to identify replacement policies that are based on the initial access order of blocks (*e.g.*, FIFO). To exert this control, our access pattern begins with a sequential scan of the test region. The resulting initial access queue ordering is shown in the first graph of Figure 2.4; specifically, the blocks at the end of the file are those that are given priority (*i.e.*, remain in the buffer cache) given a FIFO-based policy.

Dust is able to identify replacement policies that are based on temporal locality (*e.g.*, LRU) by controlling how recently each block is accessed and ensuring that this ordering does not match the initial access ordering. To ensure this criteria, a pattern of reads across ten *stripes* within the file are performed. Specifically, two indices into the file are maintained: a left pointer, which starts at the beginning of the file, and a right pointer, which starts at the center of the test region. The workload alternates between reading one stripe as indicated by the left pointer and then one stripe as indicated by the right pointer. The pattern continues until the left pointer reaches the center of the test region and the right pointer reaches the end. This controlled pattern of access induces the recency queue order shown in the middle graph of Figure 2.4; specifically, the blocks at the end of the left and right regions are those given priority with an LRU-based policy.

Finally, to identify policies that have a frequency based component, *Dust* ensures that stripes in the test region have distinctive frequency counts. When reading stripes for recency ordering, *Dust* touches each stripe multiple times for a frequency ordering as well. In our pattern, stripes near the center of the test region are read the most often, and those near the beginning and end of the test region are read the least. The number of reads for each area of the test region is shown in the right-most graph of Figure 2.4, where blocks in the middle are given priority with an LFU-based policy.

The need to impose different frequencies on different parts of the file is part of the motivation for dividing the test region into a fixed number of stripes. If, for instance, each block of the test region were given a different frequency count, the runtime of *Dust* would be exponential in the size of the file. In our simulation experiments, we determined ten to be a good number. The more stripes used, the more precise the fingerprint becomes since there is a greater variety of frequency and recency regimes. However, a greater number of stripes makes each stripe smaller thus making the data more susceptible to noise.

2.2.2.2 Forcing Evictions

Once the state of the buffer cache is configured, *Dust* performs an *eviction scan* in which more file data is read to cause some portion of the test region to be evicted from the cache. Since the goal of evicting pages is to give us the most information and ability to differentiate across replacement policies, *Dust* tries to evict approximately half of the cached data.[1]

We note that the eviction scan must read each page multiple times such that the frequency counts of its pages are higher than those of the pages in the test region. Otherwise, *Dust* would not able to identify frequency-based replacement policies since the eviction region would replace its own pages. This illustrates one of the limitations of our approach: we do not differentiate between LIFO, MRU, and MFU replacement policies, since all replace the eviction region with itself. However, we feel that this limitation is acceptable, given that such policies are used when streaming through large files and all tend to behave similarly under such conditions.

2.2.2.3 Probing File-Buffer Contents

To determine the state of the buffer cache after the eviction scan, we perform several probes, measuring the time to read one byte from selected pages. If the read call returns quickly, we assume the block of the file was resident in the cache; if the read returns slowly, we assume that a disk access was required. As noted elsewhere [4], it is not possible to perform a probe of every block to determine its state since this changes the state of the buffer cache; specifically, if *Dust* probes a block that was on disk, then this block will replace a block previously in the buffer cache, changing its state. Thus, we perform probes selectively.

To obtain an appropriate number of samples, we probe each stripe two times, for a total of twenty probes. The probes are spaced evenly across the test region, but the location of the first is chosen randomly from the first half of the first stripe. By keeping the probes relatively far apart, we ensure that they do not interfere with a later probe due to prefetching. Choosing a random offset for the probes allows one to run the benchmark multiple times to generate a better picture of the cache state. By running *Dust* multiple times on a platform, one is then able to accurately determine how the cache replacement policy chooses victim pages based on initial access, recency of access, and frequency of access.

[1]Precisely, the size of the eviction scan is set equal to the difference between the size of the cache and the size of the test region (*i.e.*, 0.1*cache size) plus one half the size of the cache.

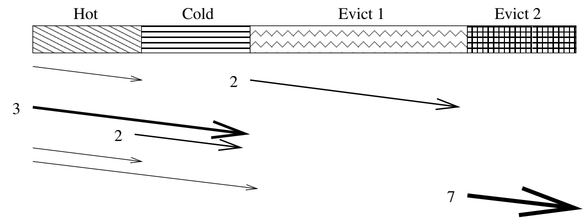


Figure 2.5 **Access Pattern to Fingerprint History.** Four distinct regions of file blocks (*i.e.*, hot, cold, evict1, and evict2) are accessed to set attributes and cause evictions in order to identify whether or not history is being used by the replacement algorithm. Each arrow indicates a region that is being accessed; reads later in time move down the page. The width of each arrow along with a number beside it, indicates the number of times each block is read to set the frequency attributes.

2.2.3 Fingerprinting History

The fingerprinting tool described thus far can identify replacement policies containing a single queue ranking blocks based upon the three attributes. However, the previous step controls only the short-term attributes of blocks and thus cannot identify algorithms that track references to blocks that are no longer in memory (*e.g.*, 2Q [29]) or that track the recency of references other than the last reference to each block (*e.g.*, LRU-K [45]). We describe the LRU-K and 2Q algorithms in Section 2.3.4 when we present the fingerprints for these algorithms. To determine if long-term tracking is performed, *Dust* observes if preference is given to pages that have been referenced and then evicted before.

We now describe how the use of long-term history is identified. As shown in Figure 2.5, there are four regions of file blocks that are now accessed. The test region is divided into two separate regions that are one half the total cache size, a *hot* and a *cold* portion. The algorithm, shown in Figure 2.7, begins by touching all of the hot pages and then evicting them by twice touching the *evict1* region; the *evict1* region contains sufficient blocks to entirely fill the buffer cache. Thus, the hot pages are no longer in the cache, but historical information about them is tracked. *Dust* then touches the *hot* and *cold* regions three times and then touches *cold* two more times. At this point, *evict1* has been evicted entirely and *cold* is preferred whether initial access, recency or frequency attributes are being used by the replacement policy. Then *cold* is touched twice. This causes the *cold* region to be preferred by traditional LRU and LFU. *Hot* is then retouched, this additional

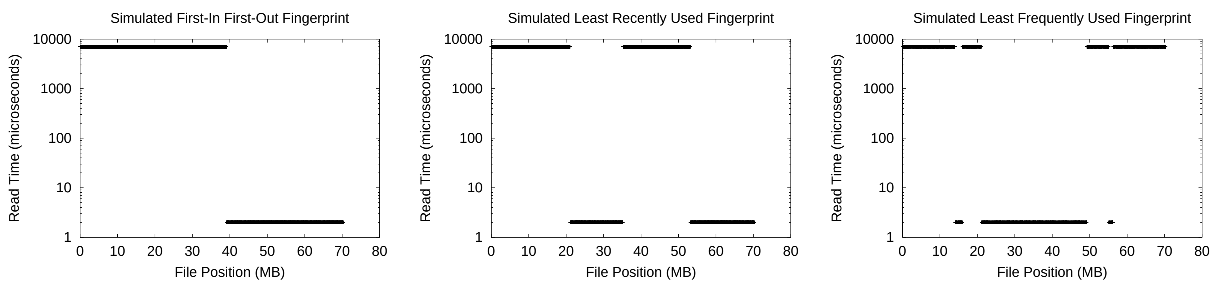


Figure 2.6 **Fingerprints of Basic Replacement Policies (FIFO, LRU, LFU).** The three graphs show the time required to probe blocks within the test region of a file depending upon the buffer cache replacement policy. The x-axis shows the offset of the probed block. The y-axis shows the time required for that probe; where low times ($2\mu s$) indicate the block was in cache, whereas high times ($7ms$) indicate the block was not in cache. From left to right, the graphs simulate FIFO, LRU, and LFU.

```
  /* read hot */
  lseek(fd, 0, SEEK_SET);
  for (i = 0; i < size/2; i += READSIZE)
    read(fd, c, READSIZE);
  /* read evict1 */
  lseek(fd, cache_size, SEEK_SET);
  for (i = 0; i < cache_size; i += page_size) {
    for (j = 0; j < 2; j++) {
      read(fd, c, 1); lseek(fd, -1, SEEK_CUR);
  }
    lseek(fd, page_size, SEEK_CUR);
 }
  /* hot - cold */
  lseek(fd, 0, SEEK_SET);
  for (i = 0; i < size; i += READSIZE) {
    for (j = 0; j < 3; j++) {
      read(fd, c, 1); lseek(fd, -1, SEEK_CUR);
    }
    lseek(fd, page_size, SEEK_CUR);
  }
  /* cold twice */
  lseek(fd, size/2, SEEK_SET);
  for (i = 0; i < size/2; i += READSIZE) {
    for (j = 0; j < 2; j++) {
      read(fd, c, 1); lseek(fd , -1, SEEK_CUR);
    }
    lseek(fd, page_size, SEEK_CUR);
  }
  /* hot */
  lseek(fd, 0, SEEK_SET);
  for (i = 0; i < size/2; i += READSIZE) {
    read(fd, c, READSIZE);
  }
  /* hot - cold */
  lseek(fd, 0, SEEK_SET);
  for (i = 0; i < size; i += READSIZE) {
    read(fd, c, READSIZE);
  }
  /* evict2 times 7 */
  lseek(fd, cache_size*2, SEEK_SET);
  for (i = 0; i < (cache_size/2) + (cache_size * 0.1); i += READSIZE) {
    for (j = 0; j < 7; j++) {
      read(fd, c, 1); lseek(fd, -1, SEEK_CUR);
    }
    lseek(fd, page_size, SEEK_CUR);
  }
}
```

Figure 2.7 **Dust long-term history algorithm.**

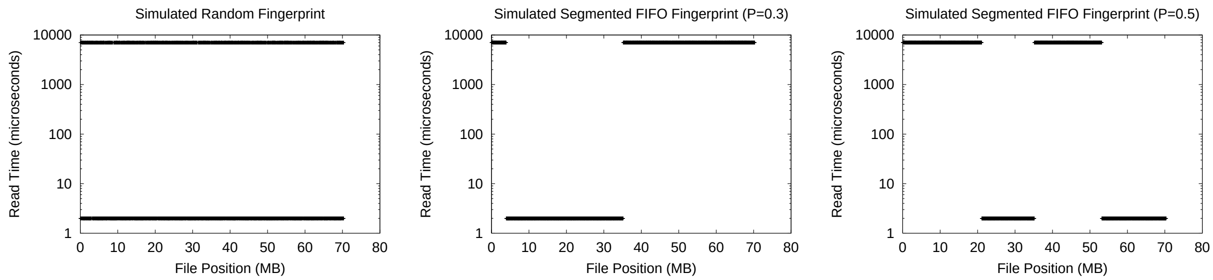


Figure 2.8 **Fingerprints of Random and Segmented FIFO.** The left-most graph shows that a Random page replacement policy has a distinctive fingerprint; that each run of the fingerprint causes different pages to be evicted from the buffer cache. The middle graph shows Segmented FIFO with 30% of the buffer cache devoted to the secondary queue; the resulting fingerprint is a cyclic shift of the FIFO fingerprint. The right-most graph shows Segmented FIFO with at least 50% of the buffer cache devoted to the secondary queue; since this queue is managed with LRU, the fingerprint is identical to LRU.

reference gives the *hot* region preference in policies which use history. The last step prior to eviction is to re-reference both the *hot* and *cold* regions sequentially. Notice that at this point the *hot* region has been touched the same number of times as the *cold* region but, it has been touched in such a way that it will have migrated into the long-term queue of a 2Q or LRU-2 cache, while the *cold* region will have not.

As in the short-term fingerprint, the next phase of *Dust* is to probe the test region to determine which blocks have been kept in the file cache. If the hot region remains in the cache, then we infer that history is being used. If the cold region remains in the cache, then we infer that history is not being used. Given that further identification of history attributes is likely to be specific to each replacement algorithm, we focus on only this simple historical fingerprint.

2.3 Simulation Fingerprints

To illustrate the ability of *Dust* to accurately fingerprint a variety of cache replacement policies, we have implemented a simple buffer cache simulator. In this section, we describe our simulation framework and then present a number of results. Our first simulation results verify the distinctive short-term replacement fingerprints produced for the pure replacement policies of FIFO, LRU, and LFU [53], as well as for other simple replacement policies such as Random and Segmented

FIFO [69]. To explore the impact of internal state within the replacement policy, we investigate Clock [44] and Two-handed Clock [70]. We then demonstrate our ability to identify the use of historical information in the replacement policy, focusing on 2Q [29] and LRU-K [45]. We conclude this section by showing that *Dust* is robust to some inaccuracy in its estimate of buffer-cache size.

2.3.1 Simulation methodology

Given that our simulator is meant only to illustrate the ability of *Dust* to identify different OS buffer cache replacement policies, we keep the rest of the system as simple as possible. Specifically, we assume that the only process running is our fingerprinting software, and thus ignore irregularities due to scheduling interference. We currently model only a buffer cache of a fixed size and do not consider any contention with the virtual memory system. For most of our simulations, we model a buffer cache containing approximately 80 MB (or 20,000 4 KB pages). Finally, we assume that reads that hit in the file cache require a constant time of 2 μs, whereas reads that must go to disk require 7 ms.

2.3.2 Basic Replacement Policies

We begin by showing that the simulation results for strict FIFO, LRU, and LFU replacement policies precisely matches what one can derive from the ordering graphs shown in Figure 2.4. The fingerprints from these three simulations are shown in Figure 2.6. We further show that *Dust* can identify Random replacement and Segmented FIFO [35]. These fingerprints are shown in Figure 2.8. Across all the graphs, one can observe the two levels of probe times, corresponding to blocks that are in cache and those that are not. Also, one can verify that approximately half of the test data remains in cache.

We now examine these basic policies in turn. The FIFO fingerprint shows that the second half of the test region remains in cache; this matches the initial access ordering shown in Figure 2.4 where blocks at the end of the file have priority. The LRU fingerprint shows that roughly the second quarter and the fourth quarter of the test region remains in the buffer cache; once again, this is the expected behavior since those blocks have been accessed the most recently. Finally, the

LFU fingerprint shows that middle half of the file remains resident, as expected, since those blocks have the highest frequency counts. In the LFU fingerprint, one can see two small discontinuous regions that remain in cache to the left and right of the main in-cache area; this behavior is due to the fact that within each stripe, blocks have the same frequency count and these in-cache regions are part of a stripe that was beginning to be evicted.

Fingerprinting a Random replacement policy stresses the importance of running *Dust* multiple times. With a single fingerprint run of twenty probes, there exists some probability that Random replacement behaves identically to FIFO, LRU, or LFU. Therefore, by fingerprinting the system many times, we can definitively see that random pages are selected for replacement. This is illustrated in the first graph of Figure 2.8 with two horizontal lines indicating the "fast" and "slow" access times.

The original VMS system implemented the Segmented FIFO (SFIFO) page replacement policy [35]. SFIFO divides the buffer cache into two queues. The primary queue is managed by FIFO. Non-resident pages are faulted into the primary queue. When a page is evicted from the primary queue, it is moved to the secondary queue. If a page is accessed while in the secondary queue, it moves back into the primary queue. The key parameter in SFIFO is the fraction of the buffer cache devoted to the secondary queue, denoted P (thus, $1 - P$ is the fraction devoted to the primary queue).

A value of $P = 0.3$ is the traditional choice and is fingerprinted in the middle graph of Figure 2.8. The resulting SFIFO fingerprint is a cyclic shift of the pure FIFO fingerprint. The reason for this pattern is as follows. The initial read of the test area sets the contents of the primary and secondary queues such that the first pages accessed (*i.e.*, the left portion of the test area) are shifted down to the secondary queue and the tail of the primary queue; the right portion is at the head of the primary queue. When the pages are touched to set the recency and frequency attributes, the left portion of the test area is moved back to the head of the primary queue while the right portion is shifted down into the secondary queue and end of the primary queue. Thus, as blocks are evicted, the right portion is evicted first, followed by the first blocks of the left portion. Thus, with these

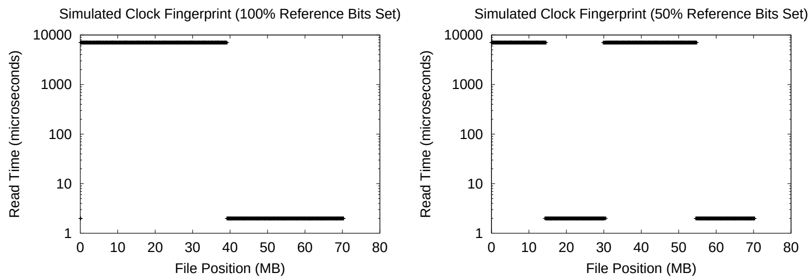


Figure 2.9 **Fingerprints of the Clock Replacement Policy.** To identify Clock, the basic fingerprinting algorithm is run twice. The first time it is run after the use bits have been all set; in this case, Clock behaves identically to FIFO as shown in the graph on the left. The second time it is run after half of the use bits have been set; in this case, Clock has the same fingerprint as LRU, as shown in the graph on the right.

queue sizes, SFIFO produces a distinctive fingerprint which can be used to uniquely identify this policy.

As P increases, SFIFO behaves more like LRU. When $P \geq 0.5$ the fingerprint becomes identical to that of LRU, as shown in Figure 2.8. When the secondary queue is that large, by the time a page is touched for the second time, it has already progressed into the secondary queue. Thus, the fingerprint reveals the LRU behavior of the policy and matches the LRU fingerprint. We feel that since Segmented FIFO is used to approximate LRU (especially with this high value of P), it is acceptable, and even appropriate, that its fingerprint cannot be distinguished from that of LRU.

2.3.3 Replacement Policies with Initial State

The Clock replacement algorithm is a popular approach for managing unified file and virtual memory caches in modern operating systems, given its ability to approximate LRU replacement with a simpler implementation. The Clock algorithm is an interesting policy to fingerprint because it has two pieces of internal initial state: the initial position of the clock hand and whether or not each use bit is set. Thus, we must ensure that Clock can be identified by its fingerprint regardless of its initial state. We now describe small modifications to our methodology to guarantee this behavior.

In the basic implementation of Clock, the buffer cache is viewed as a circular buffer starting from the current position of the *clock hand*; a single *use bit* is associated with each page frame.

Whenever a page is accessed, its use bit is set. When a replacement is needed, the clock hand cycles through page frames, looking for a frame with a cleared use bit and also clearing use bits as it inspects each frame. Thus, Clock approximates LRU by replacing pages that do not have their use bit set and have not been accessed for some time.

Since Clock treats the buffer cache as circular, the initial position of the clock hand does not affect our current fingerprint. The initial position of the clock hand simply determines where the first block of the test region is placed. Since all subsequent actions are relative to this initial position, this position is transparent to *Dust*. Thus, we do not need to modify our fingerprinting methodology to account for hand position.

However, the state of the use bits does impact our fingerprint. Depending upon the fraction of set use bits, U, the Clock fingerprint can look like FIFO or LRU. Specifically, when U is near the two extremes of 0 or 1, the fingerprint looks like FIFO; when U is near 0.5, the fingerprint looks like LRU. We now describe the intuition behind this behavior.

In the simplest case, when $U = 0$, each frame starting with the clock hand is allocated to sequential pages of the test region. As a result, the clock hand wraps back to the beginning of the buffer cache after this allocation and as *Dust* touches each page to set attributes, the use bit of every page is set. During eviction, the first pages of the test region are replaced, matching both the behavior and fingerprint of a FIFO policy. Note that $U = 1$ results in identical behavior, except the clock hand must first sweep through all frames clearing use bits before it allocates the test region sequentially.

When $U = 0.5$, the left and right portions of the test region data are randomly interleaved in memory. This interleaving occurs because pages are allocated in two passes. In the first pass, those frames with cleared use bits are allocated to the left-hand portion of the test region; the use bits of these frames are then set and the use bits of the remaining frames are cleared. In the second pass, the remaining frames are allocated to the right-hand portion of the test region. In the accesses to set the locality and frequency attributes of the pages, the use bits of all frames are again set. Thus, when the eviction phase begins, the first half of pages from both the left and right portions of the test region are replaced. If the frames with set use bits are uniformly distributed, this coincidentally matches the evictions of the LRU policy. If the distribution of use bits were not uniform, the fingerprint would show those blocks whose frames had their use bits initially clear as having been replaced. We consider the case where they are uniformly distributed as this provides a consistent and recognizable fingerprint.

Thus, to identify Clock, *Dust* brings the initial state of the use bits into each of these two configurations and observes the resulting two fingerprints. The following steps can be followed to configure the use bits from outside of the OS. *Dust* sets all of the use bits (*i.e.*, $U = 1$) by allocating a *warm-up* region of pages that fills the entire buffer cache and then touching all pages again (with no intervening allocations) so that their use bits are set.

Setting half of the use bits (*i.e.*, $U = 0.5$) is slightly more complex. The first step is to set all the use bits as in the previous scenario. In the second step, *Dust* allocates a few more pages to the warm-up region; since all of the reference bits are set at this point, the clock hand must pass through the entire buffer cache, clearing all of the reference bits, to find a page to evict. The final step is to randomly touch half of the pages, setting their use bits. In this way, *Dust* can configure the state of the use bits.

In summary, we modify *Dust* slightly to account for internal state. Before running any fingerprint, *Dust* first allocates the warm-up region, which has the effect of setting use bits if the replacement policy implements them. If the resulting fingerprint looks like FIFO, then *Dust* runs again with half the use bits set. If the fingerprint still looks like FIFO, then we conclude that there are no use bits and the underlying policy is FIFO. If the second fingerprint looks like LRU, we conclude that Clock is the underlying policy. The result of running these two steps on the Clock replacement policy is shown in Figure 2.9.

2.3.4 Replacement Policies with History

We now show that *Dust* is able to distinguish those replacement policies that use long-term history from those that do not. We begin by briefly showing that the policies examined above (FIFO, LRU, LFU, Random, Segmented FIFO, and Clock) do not use history. We then discuss in more detail the behavior of those policies (LRU-K and 2Q) that do use history.

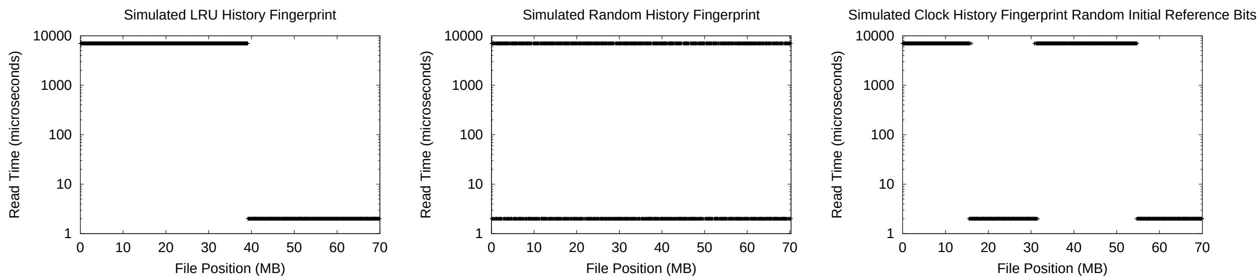


Figure 2.10 **History Fingerprint of Short-term Policies.** Probes are performed on only pages in the hot (*i.e.*, the blocks on the left) and cold (*i.e.*, the blocks on the right) test regions. The graph on the left shows the fingerprint for FIFO, LRU, LFU, and Segmented FIFO. Since the cold test region remains in the buffer cache, these policies do not prefer pages with history. The graph in the middle shows that Random also has no preference for pages with history and thus does not use history. Finally, the graph on the right shows that the historical fingerprint of Clock is ambiguous if the use bits are not set; after the use bits have been properly set, the fingerprint is identical to leftmost graph.

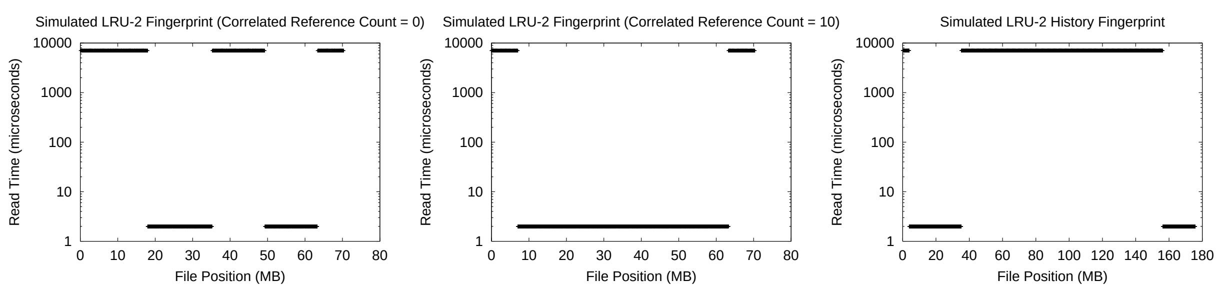


Figure 2.11 **Fingerprints of LRU-2.** The first graph shows the short-term fingerprint of LRU-2 when the correlated reference count is set to zero; in this case, LRU-2 displaces those pages with a frequency count less than 2 and those whose second-to-last reference is the oldest. The second graph shows the short-term fingerprint of LRU-2 when the correlated reference count is increased; here, no pages in the eviction with a frequency count higher than two are evicted. Finally, the last graph shows the history fingerprint of LRU-2, verifying that it prefers the hot pages.

Figure 2.10 shows the long-term fingerprints of three representative policies that do not use history. The graph on the left is that for LRU; FIFO, LFU, and Segmented FIFO look identical and are not shown. The graph shows the results of probing the hot and cold regions of the test data. As expected, the hot data has been entirely evicted, as shown by its high probe times; although the initial portion of the cold data is also evicted due to the size of the eviction region, the cold data is clearly preferred by these policies. The middle graph shows that Random has no preference for either hot or cold data. Finally, the graph on the right shows that the historical behavior of Clock is difficult to determine when the use bits are not explicitly controlled. In this graph, the use bits are set to $U = 0.5$; as a result, the hot and cold regions are interleaved in the file buffer and then each region is replaced sequentially. To illustrate that Clock does not use history, *Dust* must again ensure that the use bits are all first cleared (or set); with this initialization step, the history fingerprint of Clock is identical to the first graph in the figure. Thus, FIFO, LRU, LFU, Segmented FIFO, Random, and Clock do not use history in making replacements.

The LRU-K replacement policy was introduced by the database community to address the problem that LRU is not able to discriminate between frequently and infrequently accessed pages [45]. The idea behind LRU-K is that it tracks the K-th reference to each page in the past, and replaces the page with the oldest K-th reference (or a page that does not have a K-th reference); thus, traditional LRU is equivalent to LRU-1. Given that $K = 2$ exhibits most of the benefits of the general case, and is the most commonly used value, we only consider LRU-2 further. LRU-2 is sensitive to another parameter as well, the correlated reference period, C; the intuition is that accesses to a page within this period should not be counted as distinct references. Since setting C correctly is a non-trivial task, the default value for C is zero. Given that LRU-2 is complex, we note that our implementation is derived from the version provided by the original authors [46].

We begin by briefly exploring the sensitivity of LRU-2 to the correlated reference period; the short-term fingerprints of LRU-2 are shown in the first two graphs of Figure 2.11. When $C = 0$ (*i.e.*, the default value) the resulting fingerprint is a variation of pure LRU, as shown in the left-most graph. Specifically, the last stripe of the test region is evicted with LRU-2; since this stripe was accessed only twice, its second-to-last reference is very old (*i.e.*, when the page was initially

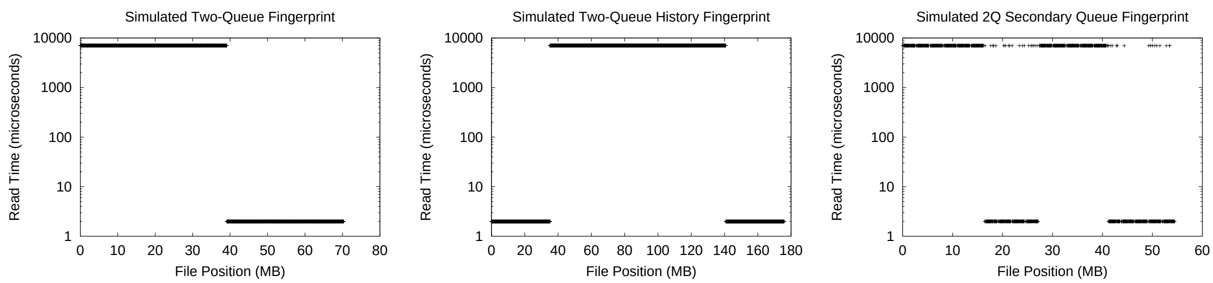


Figure 2.12 **Fingerprints of 2Q.** The first fingerprint of 2Q shows that the short-term replacement policy used is FIFO. The second fingerprint shows that 2Q uses history, preferring pages that have been accessed and then evicted. The third fingerprint shows that the replacement policy used for pages in the main queue is LRU.

referenced). As the correlated reference period is increased such that $C > 0$, the fingerprint looks more similar to LFU, as shown in the middle graph. With this setting, pages in the eviction region are classified as having only correlated references and thus replace mostly themselves; thus, all of those pages that have a frequency count greater than two are kept in memory. Finally, when C is very large, all accesses are treated as correlated and thus no pages have a second-to-last reference; in this case the behavior degenerates to pure LRU (not shown). In summary, LRU-2 produces a distinctive fingerprint that uniquely identifies it and also indicates the approximate setting of the correlated reference period.

Next, we verify that LRU-2 uses history. The last graph in Figure 2.11 shows the historical fingerprint of LRU-2. As desired, the hot region is given preference over data in the cold region; this occurs because the second-to-last reference of pages in the hot region is more recent than the second-to-last reference to those in the cold region. Further, when a replacement must be made within the hot region, those with the oldest second-to-last reference are chosen.

The 2Q algorithm was proposed as a simplification to LRU-2 with less run-time overhead yet similar performance [29]. The basic intuition behind 2Q is that instead of removing cold pages from the main buffer, it only admits hot pages to the main buffer. Thus, the buffer cache is divided into two buffers, a temporary queue for short-term accesses, `A1in` which is managed with FIFO, and the main buffer, `Am`, which is managed with LRU. Pages are initially admitted into the `A1in` queue and only after they have been evicted and reaccessed are they admitted into `Am`. Thus, 2Q

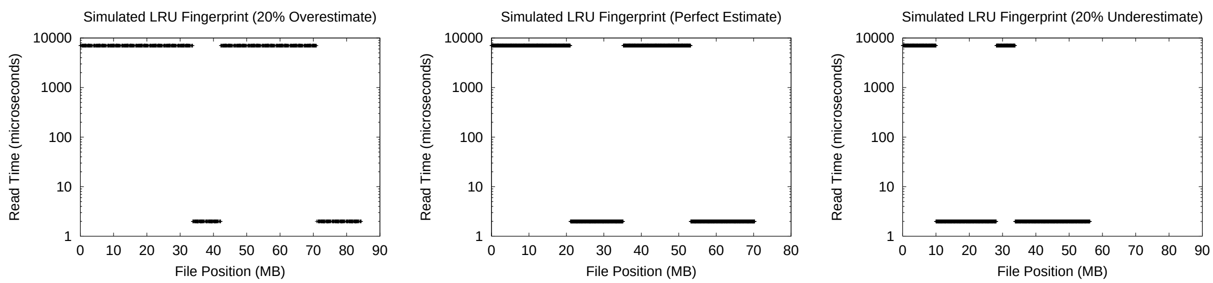


Figure 2.13 **Sensitivity of LRU Fingerprint to Cache Size Estimate.** These graphs show the short-term fingerprints of LRU as the estimate of the size of the buffer cache is varied. In the first graph the estimate is too high by 20%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 20%. However, all fingerprints still uniquely identify LRU.

has another structure to remember the pages that have been accessed but are no longer in the buffer cache, `A1out`. In our experiments, we set `A1in` to use 25% of the buffer cache (with `Am` using the other 75%); `A1out` is able to remember a number of past references equal to 50% of the number of pages in the cache.

We show the fingerprints for 2Q in Figure 2.12. The first graph shows that the short-term fingerprint of 2Q is identical to FIFO. Given that the `A1in` queue is managed with FIFO and the short-term fingerprint does not access pages after they have been evicted, this is the expected result. However, 2Q can be easily distinguished from pure FIFO from observing the history fingerprint shown in the second graph. In the historical fingerprint, we can see that the hot region remains entirely in the buffer cache, since these are the only accesses that are moved to the `Am` buffer. Finally, we are able to identify the replacement policy employed by the long-term buffer, `Am`, by setting the initial access, recency, and frequency attributes of the hot region and then forcing evictions from it. Since this methodology is more specific to the 2Q replacement policy, we do not describe it in more detail. This fingerprint is shown as the last graph of Figure 2.12 and correctly identifies the LRU policy of the `Am` buffer. We note that for LRU-2 or other policies that use history, a similar technique could be used to determine the replacement strategy of the long-term queue. However, explicitly setting the state of the long-term queue requires knowledge of the policy of the short-term queue and the policy for moving a block from one queue to the other.

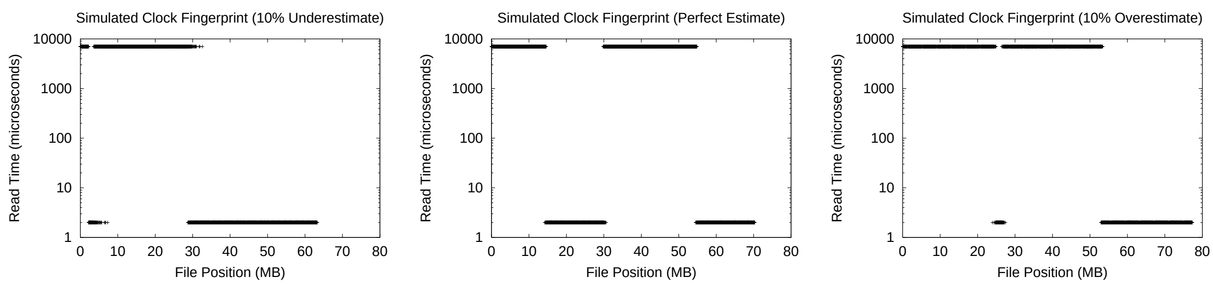


Figure 2.14 **Sensitivity of Clock Fingerprint to Cache Size Estimate.** These graphs show the short-term fingerprints of Clock with half of the use bits set as the estimate of the size of the buffer cache is varied. With $U = 0.5$, Clock is expected to look like LRU. In the first graph the estimate is too high by 10%, in the second graph the estimate is perfect, and in the third graph the estimate is too low by 10%. Thus, the Clock fingerprint is not as robust to inaccuracies in this estimate as the other algorithms.

Hence a fingerprinting technique for the long-term queue is by nature specific to the policy of the short-term queue.

2.3.5 Sensitivity to Buffer Size Estimate

In our last set of experiments we verify the robustness of *Dust* to inaccuracies in its estimate of the size of the buffer cache. If the estimate of the buffer cache size is significantly different than its actual value, then the resulting fingerprints are not identifiable. If the estimate of the cache is much too small, then *Dust* does not touch enough pages to force evictions to occur; if the estimate is much too large, then *Dust* evicts the entire region.

The short-term fingerprint is more sensitive to this estimate than the historical fingerprint: in the short-term fingerprint we must observe the presence or absence of stripes that use only 1/10th of the buffer cache, whereas in the historical fingerprint we must observe a hot or cold region that uses half of the buffer cache. However, as Figure 2.13 shows, the short-term fingerprint of LRU is distinguishable even with estimates that are either 20% under or over the real sizes. The other replacement policies, with the exception of Clock, are robust to a similar degree.

The Clock replacement algorithm is more sensitive to this estimate due to our need to configure the state of the use bits. Specifically, the size of the warm-up region used by *Dust* to fill the buffer cache must be accurate as well. Figure 2.14 shows that *Dust* is still reasonably tolerant to errors in cache-size estimate when identifying Clock but not as robust as when identifying other algorithms.

2.3.6 Summary

Through our simple simulation, we have shown that *Dust* is capable of identifying a wide variety of buffer cache replacement policies. *Dust* differentiates policies based on the attributes of the workload they use to make replacement decisions: initial access order, access recency, access frequency and long-term history. *Dust* also accounts for use bits in Clock-like algorithms. In the next section, we demonstrate *Dust* identifying cache replacement policies of real operating systems.

2.4 Platform Fingerprints

Buffer caching in modern operating systems is often much more complex than the simple replacement policies described in operating systems textbooks. Part of this complexity is due to the fact that the file system buffer cache is integrated with the virtual memory system in many current systems; thus the amount of memory dedicated to the buffer cache can change dynamically based on the current workload. To control this effect, *Dust* minimizes the amount of virtual memory that it uses, and thus tries to maximize the amount of memory devoted to the file buffer cache. Further, we run *Dust* on an otherwise idle system to minimize disturbances from competing processes.

In this section, we describe our experience fingerprinting three Unix-based operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, Solaris 2.7 and HP-UX 11i. As we will see, the fingerprints of real systems contain much more variation than those of our simulations. In addition to fingerprinting the replacement policy of the buffer cache, *Dust* also reveals the cost of a hit versus a miss in the buffer cache, the size of the buffer cache, and whether or not the buffer cache is integrated with the virtual memory system.

Dust takes a considerable amount of time to run on a real system. Generating a sufficient number of data points requires running many iterations of test scan, eviction scan, and probes. In our experiments we always allowed at least 300 iterations. We found that one iteration can take anywhere from 30 seconds to three minutes depending on the system under test. Note that systems with smaller buffer caches can be tested in a shorter period of time since the test region becomes smaller. We feel this relatively long running time is acceptable since, for any given system configuration, *Dust* need only be run once; the results can be stored and made available to applications and programmers.

All of the experiments described in the section were run on systems with dual Pentium III-Xeon processors, 1 GB of physical RAM and a SCSI storage subsystem with Ultra2, 10000 RPM disks.

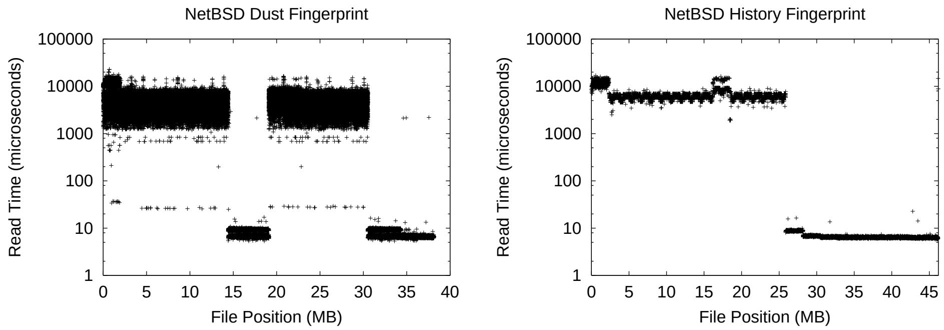


Figure 2.15 **Fingerprints of NetBSD 1.5.** The first graph shows the short-term fingerprint of NetBSD, indicating the LRU replacement policy. The second graph shows the long-term fingerprint, indicating that history is not used.

2.4.1 NetBSD 1.5

Given that NetBSD 1.5 [39] has the most straight-forward replacement policy of the systems we have examined, we begin with its fingerprint, shown in Figure 2.15. As in the simulations, we examine both short-term and long-term fingerprints. The first graph in Figure 2.15 shows the expected pattern for pure LRU replacement; given that *Dust* produces this same fingerprint regardless of whether it attempts to manipulate use bits, we can infer that NetBSD implements strict LRU, and not Clock. This conclusion is further verified by the second graph of Figure 2.15 showing that NetBSD does not use history. Documentation [39] and inspection of the source code [43] confirm our finding.

From the fingerprints we can also infer other parameters. Specifically, we can see that the time for reading a byte from a page in the buffer cache is on the order of 10 μs, whereas the time for going to disk varies between about 1 ms and 10 ms. Further, even on this machine with 1 GB of physical memory, NetBSD devotes only about 50 MB to the buffer cache (most easily shown by the fact that the history fingerprint devotes this much memory to the hot and cold regions); this allows us to infer that the file buffer cache is segregated from the VM system.

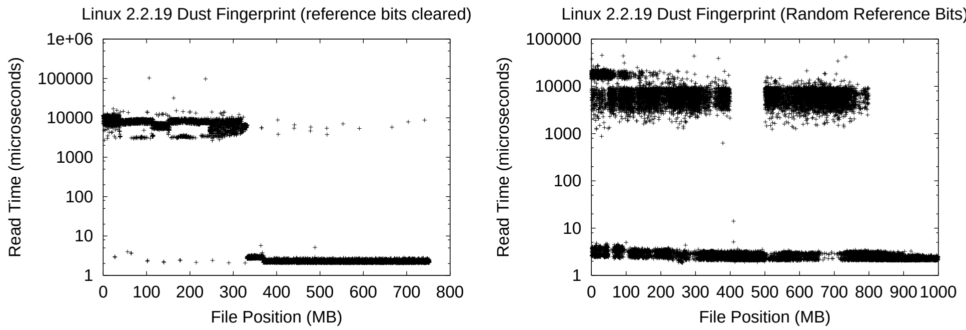


Figure 2.16 **Fingerprints of Linux 2.2.19.** The first graph shows the short-term fingerprint of Linux 2.2.19 when the use bits are all set; the second graph shows the fingerprint when the use bits are untouched.

2.4.2 Linux 2.2.19

Linux 2.2.19 is a very popular version of the Linux kernel in production environments. In Section 2.5 we will run the NeST web server on top of this OS; thus, it is important for us to understand this fingerprint.

The short-term fingerprint of Linux 2.2.19 is shown in Figure 2.16. The graph on the left shows the results when *Dust* attempts to set all of the use bits. Since this graph looks like FIFO, we must investigate further to determine if Clock is actually being used. The graph on the right shows the fingerprint when the use bits are left in a random state. Although this fingerprint is very noisy, one can see that priority is given to pages that are most recently referenced (*i.e.*, pages near the second and fourth quarters); further, after filtering the data, we are able to verify that more pages in the first and third quarters are out of cache than in cache. Thus, this fingerprint is similar to the LRU fingerprint expected for a Clock-based replacement algorithm. Examination of the source code and documentation confirms that the replacement policy is Clock based [36, 72]. Finally, since the buffer cache size is very close to the amount of physical RAM in the system, we conclude a buffer cache that is integrated with the VM.

2.4.3 Linux 2.4.14

The memory management system within Linux underwent a large revision between version 2.2 and 2.4, thus we see a very different fingerprint for Linux 2.4.14, which uses a more complex

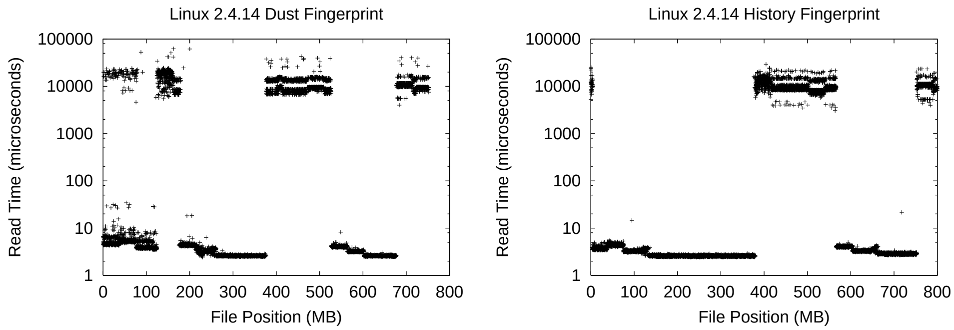


Figure 2.17 **Fingerprints of Linux 2.4.14.** The first graph shows the short-term fingerprint of Linux 2.4.14, indicating that a combination of LRU and LFU is used. The second graph shows the long-term fingerprint, indicating that history is used.

replacement scheme than either Linux 2.2.19 or NetBSD. The short-term fingerprint, shown as the first graph in Figure 2.17, suggests that Linux 2.4 uses both a recency and frequency component, and does not use Clock. Further, the second graph of *Dust* shows that Linux 2.4 does use history in its decision.

Examination of the Linux 2.4.14 source code and existing documentation confirms these results [36, 72]. Linux maintains two separate queues: an active and an inactive list. When memory becomes scarce, Linux shrinks the size of the buffer cache. In doing this, pages that have not been recently referenced (as indicated by their reference bit) are moved from an active list to an inactive list. The inactive list is scanned for replacement victims using a form of page aging, in which an *age* counter is kept for each frame, indicating how desirable it is to keep this page in memory. When scanning for a page to evict, the page age is decreased as it is considered for eviction; when the page age reaches zero, the page is a candidate for eviction. The *age* is incremented whenever the page is referenced.

2.4.4 Solaris 2.7

Solaris presented us with the greatest challenge of the platforms we studied. The VM subsystem of Solaris has not been thoroughly studied; it is believed to use a two-handed, global Clock algorithm [11], but some researchers have noted non-intuitive behavior [4]. In two-handed Clock, one hand clears reference bits while the second hand follows some fixed distance behind, selecting

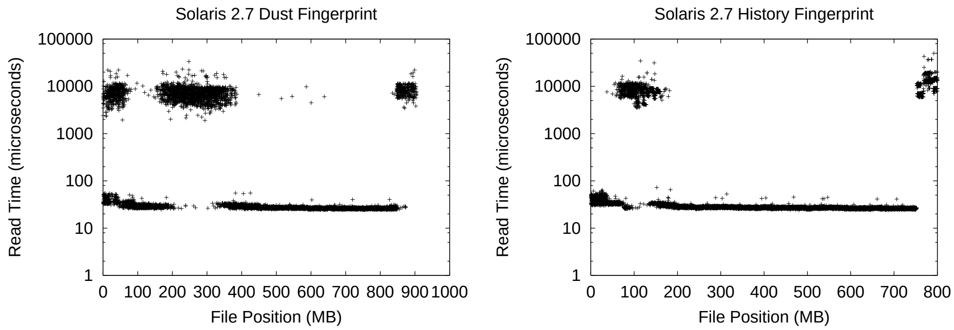


Figure 2.18 **Fingerprint of Solaris 2.7.** The first graph shows the short-term fingerprint of Solaris; the second graph shows the history fingerprint.

a page for replacement if its reference bit is still clear. The hands are advanced in unison such that once the reference bit on a page is cleared, it has some opportunity to be re-referenced before it is a candidate for eviction. When implemented in our simulator, the fingerprint of two-handed Clock looks identical to FIFO (not shown).

The short-term fingerprint of Solaris 2.7 is shown in the first graph of Figure 2.18. The out-of-cache areas on both the far right and left of the fingerprint strongly suggests that Solaris is using a frequency (or aging) component in its eviction decision in addition to Clock. The second graph of Figure 2.18 shows the historical fingerprint for Solaris. Though the data is again noisy, it shows a clear preference for the hot region, again suggesting that history or page aging is also used in Solaris. The fingerprint also shows that the time to service a buffer cache hit is significantly higher in Solaris than in Linux. The fingerprint shows a hit time of over 10 μs, whereas the hit time for Linux 2.4 on the same platform is under 10 μs.

2.4.5 HP-UX 11i

The last system we fingerprint is HP-UX 11i, running on an Intel Itanium system. For this experiment we configured HP-UX to limit the size of the buffer cache to 40,000 pages. This is entirely for convenience as *Dust* completes faster on smaller caches.

Figure 2.19 shows the fingerprint for HP-UX 11i. The graphs show the fingerprint both attempting to set all of the use bits to be uniform, and leaving them set randomly. Since the graphs

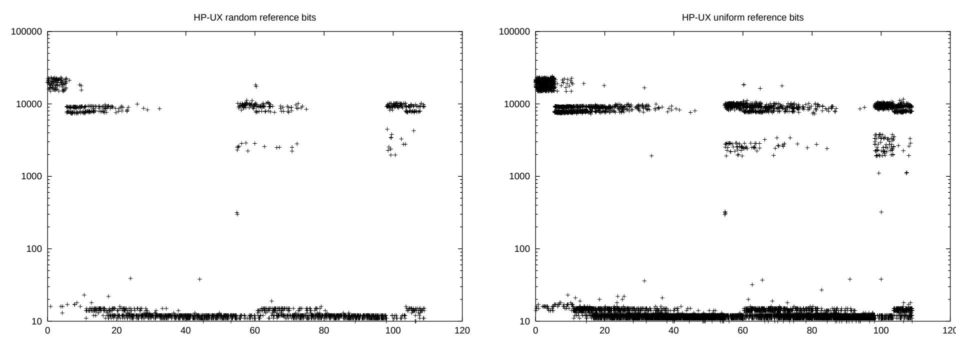


Figure 2.19 **Fingerprints of HP-UX 11i (Itanium).** The first graph shows the short-term fingerprint of HP-UX 11i when the use bits are randomized; the second graph shows the fingerprint when the use bits are uniform. The fingerprint indicates that a combination of recency and frequency is being used. The fingerprint doesn't significantly change based on use bit manipulation, so use bits are being ignored by the policy.

are virtually the same, we conclude that no use bits exist, or they are being ignored by the replacement policy. Similar to the short-term fingerprint of Linux 2.4.14, HP-UX appears to use both recency and frequency in determining replacement decisions. Source code for HP-UX 11i is not available to us so we are unable to verify our conclusion in this case.

2.4.6 Summary

This section describes *Dust* fingerprints for several popular operating systems: NetBSD 1.5, Linux 2.2.19 and 2.4.14, Solaris 2.7 and HP-UX 11i. Fingerprinting real systems is more challenging for a variety of reasons. Real operating systems often use replacement policies that are more sophisticated than simple LRU or LFU, and thus are more difficult to identify. Fingerprints from real systems are far noisier than simulated fingerprints due to variations in access times in the storage stack and other processes running on the system. Despite these difficulties, *Dust* is able to identify the replacement policies of these systems.

In some cases, such as NetBSD it is easy to identify the precise replacement algorithm from the *Dust* fingerprint. For systems with more sophisticated replacement policies, it may not be possible to pinpoint the replacement policy, but the *Dust* fingerprint still reveals what workload attributes are being used to make replacement decisions. We believe that even this more limited information is still useful. In the next section, we discuss a cache-aware webserver and show that somewhat inaccurate information is still quite valuable.

2.5 Cache-Aware Web Server

In this section, we describe how knowledge of the buffer cache replacement algorithm can be exploited to improve the performance of a real application. We do so by modifying a web server to re-order its accesses to first serve requests that are likely to hit in the file system cache, and only then serve those that are likely to miss. This idea of handling requests in a non-FIFO service order is similar to that introduced in connection scheduling web servers [17]; however, whereas that work scheduled requests based upon the size of the request, we schedule based upon predicted cache content. As we will see, re-ordering based on cache content both lowers average response time (by emulating a shortest-job first scheduling discipline) and improves throughput (by reducing total disk traffic).

2.5.1 Approach

The key challenge in implementing the cache-aware server is to use our knowledge of the file caching algorithm to determine which files are in the cache. By keeping track of the file access stream being presented to the kernel, the web server can simulate the operating system's buffer cache and thus predict at any given time what data is in cache. We term this *algorithmic mirroring*, and believe that it is a general and powerful manner in which to exploit gray-box knowledge.

One important assumption of algorithmic mirroring is that the application induces most or all of the traffic to the file system, and thus the mirror cache is likely to accurately represent the state of the real OS cache. Although this assumption may not hold in the general case within a multi-application environment, we believe it is feasible when a single application dominates all file-system activity. Server applications such as a web server or database management system are thus a perfect match for such mirroring methods.

The NeST storage appliance [9] supports HTTP as one of its many access protocols. NeST allows a configurable number of requests to be serviced simultaneously. Any requests received beyond that number are queued until one of the pending requests completes. By default, NeST services queued requests in FIFO order. We term this default behavior as *cache-oblivious NeST*.

We have modified the NeST request scheduler to keep a model of the current state of the OS buffer cache. The model is updated each time a request is scheduled. NeST bases its model of the underlying file cache on the algorithm exposed by *Dust*. NeST uses this model to reorder requests such that those requests for files believed to be in cache are serviced first. Note that NeST does not perform caching of files itself, but relies strictly upon the OS buffer cache.

For the cache mirror to accurately reflect the internal state of the OS, NeST must have a reasonable estimate of the cache size. In our current approach, NeST uses the static estimate produced by *Dust*; the disadvantage of this approach is that this estimate is produced without contention with the virtual memory system, and thus may be larger than the amount available when the web server is actually running. To increase the robustness of our estimate, it would be possible to modify NeST to dynamically estimate the size of the buffer cache by measuring the time for each file access. If the time is “low”, the file must have been in the cache, and if it is “high”, the file was likely on disk. By comparing these timings with the prediction provided by the mirror cache, NeST can adjust the size of the mirror cache.

2.5.2 Performance

To evaluate the performance benefits of cache-aware scheduling, we compare the performance of cache-aware NeST to cache-oblivious NeST for two different workloads. In all tests, the web server is run on a dual Pentium III-Xeon machine with 128 MB of main memory and Ultra II disks. For clients, we use four machines (identical to the server, except containing 1 GB of main memory) each running 36 client threads. The clients are connected to the server with Gigabit Ethernet.

The server and clients are running Linux 2.2.19, which was shown in Section 2.4.2 to use the Clock replacement algorithm; therefore, cache-aware NeST is configured to model the Clock algorithm as well. In our configuration, the server has approximately 80 MB of memory dedicated to the buffer cache. In our experiments, we explore the performance of cache-aware NeST as we vary its estimate of the size of the buffer cache. As discussed previously, the Clock algorithm has initial state in the form of use bits, which effect replacement. We ignore this small complication in cache-aware NeST. This may result in some inaccuracy in the model initially, but since NeST dominates the systems workload, the previous state is quickly flushed and the model kept by NeST becomes accurate.

In our first experiment, we consider a workload in which each client thread repeatedly requests a uniformly distributed random file from a set of 200 1 MB files. Figures 2.20 and 2.21 show the average response time and throughput, respectively for three different web servers: the Apache web server [1], cache-oblivious NeST, and cache-aware NeST as a function of its estimate of cache-size. We begin by comparing the response time and the throughput of NeST and Apache; from the two figures, we see that although NeST incurs some overhead for its flexible structure (*e.g.*, NeST can handle multiple transfer protocols, such as FTP and NFS), it achieves respectable performance as a web server and is a reasonable platform for studying cache-aware scheduling. Second, and most importantly, adding cache-aware scheduling significantly improves both the response time and the throughput of NeST. By first servicing requests that hit in the cache, cache-aware scheduling improves average response time by servicing short requests first. More dramatically, cache-aware scheduling improves throughput by reducing the number of disk reads (verified through the `/proc` interface): in-cache requests are handled before their data is evicted from the cache. Finally, the performance of cache-aware NeST improves when its estimate of the cache size is closer to the real value, but is robust to a large range of cache size estimates.

In our second experiment, we consider a workload created by the SURGE HTTP workload generator [7]. The SURGE workload uses approximately 12,000 distinct files with sizes taken from a Zipf distribution with a mean of approximately 21 KB. SURGE is thus a more representative web workload than is presented above.

With the SURGE workload, we measure qualitatively similar results to those above, except for two main differences. First, the performance of cache-oblivious NeST relative to Apache degrades slightly more; for example, the average response time for cache-oblivious NeST is 0.80 seconds and for Apache is 0.65 seconds. This result is expected, given that NeST is designed for staging data in the Grid, and is thus optimized for large files and not the small files more typical in web workloads. Second, the performance of cache-aware NeST is not as sensitive to its estimate of the cache size; for example, performance improves from 4.27 MB/s to 4.69 MB/s (approximately 10%) as the cache size estimate is improved from 10 MB to 80 MB. Apache achieves 4.91 MB/s.

2.6 Conclusions

We have shown that various buffer cache replacement algorithms can be uniquely identified with a simple fingerprint. Our fingerprinting tool, *Dust*, classifies algorithms based upon whether they consider initial access, locality, frequency, and/or history when choosing a block to replace. With a simple simulator, we have shown that FIFO, LRU, LFU, Clock, Random, Segmented FIFO, 2Q, and LRU-K all produce distinctive fingerprints, allowing them to be uniquely identified. We

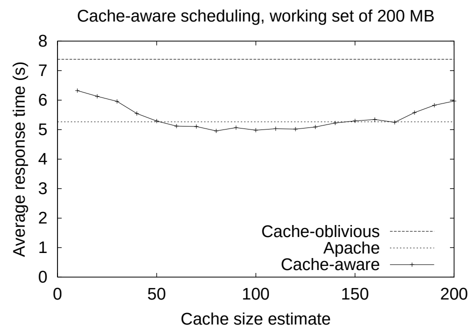


Figure 2.20 **Response Time as a Function of Cache Size Estimate.** Response time in cache-aware NeST is lowest when the estimate of cache size is closest to the true size of the cache.

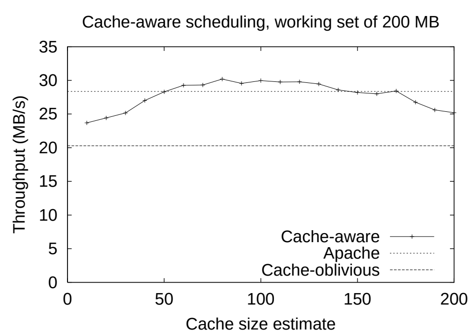


Figure 2.21 **Sensitivity to Cache Estimate Accuracy.** The performance of cache aware NeST improves as the estimate of cache size approaches the true size of the buffer cache. The buffer cache is approximately 80 MB. Cache-oblivious NeST and Apache are shown for comparison.

have also begun to address the more challenging problem of fingerprinting real systems. By running *Dust* on NetBSD, Linux, and Solaris, we have shown that we can determine which attributes are considered by each page replacement algorithm.

Further, we have shown that the algorithmic knowledge revealed by *Dust* is useful for predicting the contents of the file cache. Specifically, we have implemented a cache-aware web server that services first those requests that are predicted to hit in the file cache, improving both response time and bandwidth. Thus it is possible for applications to discover and utilize knowledge of the contents of the buffer cache, despite the operating system being designed to hide such information. In this way, we regain some of the knowledge lost due to implementing caching at the operating system level, while at the same time retaining the benefits of a centralized caching infrastructure. We also show that even though this information might be have a degree of inaccuracy, it can still be used to gain significant performance improvements.

The cache information discovered by *Dust* is obtained and used by cache-aware NeST without any alterations to the existing operating system. Leaving the OS unmodified yields several advantages. First, porting to a new OS is made simple; it only requires running *Dust* to discover the new system's cache management strategy. Second, any risk of adding bugs to the OS is eliminated; any new bugs will be confined to the user-level application. Finally, techniques based on implicit information can be used and deployed without requiring the consent of OS developers; techniques based on implicit information can be used on a per-application basis, rather than on a per-OS basis.

Implicit information is sometimes inaccurate. While the optimization described here can tolerate some inaccuracy, one might imagine more aggressive optimizations that require accurate information. Here if cache-aware NeST mispredicts the cache state, it merely results in a bad scheduling decision. As long as its predictions are correct most of the time, performance will still be better than it would be if cache state were ignored. If the application is attempting an optimization where a misprediction of the cache state could, for example, result in extra disk accesses, performance might suffer severely if the information being used is inaccurate. For optimizations in this class, implicit information may be insufficient.

Chapter 3

Exposing Buffer Cache State with Explicit Information

3.1 Introduction

Implicit information is, by its nature, imperfect. In the case of our cache-aware web server, imperfect information is acceptable. Having an imperfect algorithmic mirror is unlikely to lower performance below the level of being cache-oblivious. As we discovered, the cache size estimate can be quite inaccurate before performance drops to be near that of the cache-oblivious connection scheduler. The reason for this is twofold. First, as long as the predictions are right most of the time, performance will be better than the cache-oblivious web server. Second, since the policy being used was recency based, for any cache size estimate, the most recent items in the cache will be predicted correctly since these items will be in the intersection of the sets of contents of both the cache mirror and the actual cache. That said, we would like to be able to perform more aggressive optimizations where it is possible that inaccurate information could actually degrade performance.

There are other potential sources of inaccuracy as well. If the algorithmic mirror is simulating the wrong policy, the resulting cache predictions may be incorrect. The level of error in the prediction will be a function of the difference between the policy being simulated and the actual policy. For example, if the algorithmic mirror is simulating LRU and the actual policy is Clock, the degradation is likely to be very small since these policies are very similar in what they keep cached. However, if the actual policy is Clock and the algorithmic mirror is simulating MRU, depending on the workload the predictions are likely to be wrong most of the time.

Another possible source of error is interference from other processes. Consider a web site with a database back-end. At a small site, the web server might run on the same machine as the

DBMS. If either of these applications depends on an algorithmic mirror, that mirror is likely to be inaccurate. The only way to prevent this inaccuracy is to have both applications aware of each other and collaborate to keep a common algorithmic mirror up to date.

As we saw in the previous chapter, implicit information can also be costly to acquire. *Dust* for example, requires that the system be quiescent and even then may take hours to run to completion. If a production system is being upgraded to a new version of the operating system, the amount of downtime required to run *Dust* may not be acceptable and another system with an identical OS version may not be available.

To address these limitations, we explore *explicit information*. We modify the operating system to explicitly expose useful information to applications. This gives applications timely, accurate information that is easy to access at the cost of having to modify the operating system kernel.

Exposing information only, rather than adding new mechanisms, makes our kernel modifications very simple. This helps to mitigate the usual difficulties associated with kernel development. There is very little risk that our modifications will interfere with the existing mechanisms and policies of the system, since we only read kernel data structures, we never modify them. Since our modifications are relatively safe in this respect, we believe it will be easier to have them integrated into popular systems.

We also believe that information interfaces are often more versatile than implementing new mechanisms directly in the operating system kernel. For example, in this chapter we show that by exposing cache state, it is possibly to transform the kernel's buffer cache replacement policy into nearly any other policy. There are two ways in which this information-based approach is more versatile than a direct implementation. First, using the approach this chapter describes, an application can implement *any* cache policy that is desired. This is much more flexible than having the kernel provide a different cache policy. Even if the kernel provided a selection of policies to choose from, the application would still have to choose from that finite set. By providing information only, we let the application implement whatever policy suits it. Second, the interface we describe could be used for purposes other than policy transformation. For example, we could use it to make our algorithmic mirror from the previous chapter perfectly accurate and eliminate the need for *Dust*.

Extensible operating systems have been studied for many years [10, 58]. One approach to extensible systems has been to allow applications to upload code into the kernel. The problem then becomes one of protecting the rest of the system from misbehaving kernel extensions. Exporting information provides a way to extend kernel functionality without the risks and costs of allowing application code to be loaded into the kernel.

In this chapter, we show that by exposing two key pieces of internal operating system state to user-level applications, we enable those applications to efficiently transform the kernel provided buffer cache policy into the replacement policy of the applications choice. For applications with a high degree of knowledge about the IO workload they present, the ability to alter the cache replacement policy can yield substantial benefits. Database management systems are particularly well suited to this sort of optimization since, once a query plan is chosen, a DBMS has a great deal of knowledge about it's near-term future IO patterns [16].

We now discuss some issues and trade-offs with exposing kernel information in general. We then describe and evaluate our new interface for exposing file system buffer cache state, focusing on our Linux implementation with a brief discussion of our experience porting that interface to NetBSD.

3.2 Information Exposure Issues

In this section, we discuss the general issues of exposing internal kernel state. We begin by presenting the benefits of exposing more information about the policies and state of the OS to higher-level services and applications. We then discuss some of the fundamental tensions concerning how much information should be exposed. This discussion is applicable not only to exposing buffer cache information, but to exposing any kernel state to the user-level [5].

3.2.1 Tensions in Design

When exposing kernel state, a number of design decisions must be made. We now discuss some of the issues pertaining to the amount of information that is exposed, exposing information across process boundaries, and exposing information instead of adding new mechanisms.

3.2.1.1 Amount of information

One tension when designing an kernel to expose information is to decide what information should be exported. On the one hand, exporting as much information as possible is beneficial since one cannot always know *a priori* what information is useful to higher-level services. On the other hand, exposing information from the OS greatly expands the API presented to applications and destroys encapsulation.

There are unfortunate implications for both the application and the OS when the API is expanded in this way [19]. For example, consider a new user-level service that wants to control the page replacement algorithm and must know the next page to be evicted by the OS. If the service is developed on a system that uses Clock replacement, then the application examines the clock hand position and the reference bits. However, if this service is moved to a system with pure LRU replacement, the service must instead examine the position of the page in the LRU list. From the perspective of the user-level service, a new API implies that either the service no longer operates correctly or that it must be significantly rewritten. From the perspective of the OS, a fixed API discourages developers from implementing new algorithms in the OS and thus constrains its evolution.

Therefore, for application portability, information exposing interfaces must keep some information hidden and instead provide abstractions. However, for the sake of OS innovation, these abstractions must be at a sufficiently high level that an operating system can easily convert new internal representations to these abstractions.

Exposing too much information might also lead to performance issues. Ones first instinct might be to expose the contents of the *entire* buffer cache to applications. Exposing everything would ensure that the application had all the of the information it could possibly need (provided such information exists), however it would be difficult to design an interface to efficiently move that much data from the kernel into a user-level process. Further, since the cache state is constantly changing it would be challenging to keep the information seen by the application up to date fast enough if the entire cache were exposed.

We believe that precisely determining the correct abstractions for exposing kernel state in this way requires experience with a large number of case studies and operating systems. In this dissertation, we take a first step in defining these abstractions in the case of the file system buffer cache. More importantly, we demonstrate the ease with which information can be exposed and the potential application improvements that can be realized by leveraging such information.

To represent the file cache replacement algorithm, we find that a prioritized list of resident pages allows user-level services to efficiently determine which pages will be evicted next. Our implementation illustrates that implementing these abstractions for an existing OS is relatively simple and involves few lines of code.

3.2.1.2 Process boundaries

Another tension when designing interfaces to expose information information is to determine the information about competing processes that should be exposed to others. On the one hand, the more information about other processes that is exposed, the more one process can optimize its behavior relative to that of the entire system. On the other hand, more information about other processes could allow one process to learn secrets or to harm the performance of another process.

Clearly, some information about other processes must be hidden for security and privacy (*e.g.*, the data read and written and the contents of memory pages). Although other information about the resource usage of other processes may increase the prevalence of covert channels, this information was likely to be already available, but at a higher cost. For example, with a resident page list, a curious process may infer that another process is accessing a specific file; however, by timing the open system call for that file, the curious process can already infer from a fast time that the inode for the file was in the file cache. If exposure of certain information proves to be a risk, it can be hidden by doing more work; with the resident page list example, the corresponding file block number can be removed for those pages that do not belong to the calling process.

This issue also addresses the suitability of competing applications performing information based optimizations. One concern is that services are encouraged to "game" the OS to get the control they want, which may harm others. With more information, a greedy process can acquire more than its fair share of resources; for example, a greedy service that keeps its pages in memory by touching them before they are evicted is able to steal frames from other processes. However, given that an we not providing any new mechanisms, this behavior was possible in the original OS, albeit more costly to achieve. For example, without explicit cache state information, a greedy process can continually touch its pages blindly, imposing additional overhead on the entire system. As was shown in the Chapter 2 even without explicit information, applications can already determine cache state to a fair degree of accuracy. In summary, applications using explicit information stresses the role of the OS to arbitrate resources across competing applications (*i.e.*, to define limits in its existing policies), but does not impart any new responsibilities.

3.3 Cache Information Interfaces

Different applications benefit from different file cache replacement algorithms [14, 45, 60], and modifying the replacement policy of the OS has been used to demonstrate the flexibility of extensible systems [58]. We can emulate similar functionality with only minimal changes to the operating system. We implement a user-level library, InfoReplace, that demonstrates a variety of replacement algorithms (*e.g.*, FIFO, LRU, MRU, and LFU) can be implemented on top of the unmodified Linux replacement algorithm.

We begin by describing the intuition for how the file cache replacement policy can be treated as a mechanism, giving replacement control to applications. Consider the case where an application wishes to keep a hot list of pages resident in memory (*i.e.*, the target policy), but the OS supports only a simple LRU-replacement policy (*i.e.*, the source policy). To ensure that this hot list remains resident, the user process must know when one of these pages is about to be evicted; then the user process accesses this page some number of times, according to the source replacement policy, to increase the priority of that page. More generally, one replacement policy can be converted to another by accessing pages that are about to be evicted given the source policy, but should not be evicted according to the target policy.

3.3.1 Abstractions

To support the InfoReplace user-level library, the operating must export enough information such that applications can determine the next victim pages and the operations to move those pages up in priority.

Linux 2.4.18uses a 2Q-like replacement policy. This policy divides the buffer cache into two queues. One list of *hot* pages, managed in an LRU fashion and a list of non-hot pages, managed using FIFO. To provide the general representation of a prioritized list of all physical pages, `pageList`, the kernel exports the concatenation of these two queues through a system call. With this information, InfoReplace can examine the end of the queue for the pages of interest. The drawback of the `pageList` abstraction is that its large number of elements imposes significant overhead when copying the queue to user space; therefore, the call can be made only infrequently. However, if the queue is checked only infrequently, then pages can be evicted before the user-level library notices. Therefore, the kernel provides a `victimList` abstraction, containing only the last N pages of the full queue, as well as a mechanism to quickly determine when new pages are added to this list.

The operating system already provides a mechanism for increasing the priority of a pages in the buffer cache. For most replacement policies, `read()` increases the priority of a page as a side-effect. If the kernel provided policy is FIFO, or another policy that doesn't increase a page's priority when they are touched, then we cannot efficiently transform the replacement policy using the techniques described here. However, we have not encountered any modern operating systems that have this problem.

3.3.2 Implementation

The state within Linux can be converted into this form with low overhead as follows. Linux 2.4.18 has a unified file and page cache with a 2Q-like replacement policy [29]: when first referenced, a page is placed on the *active queue*, managed with a two-handed clock; when evicted from there, the page is placed upon the *inactive queue*, managed with FIFO.

Our modified Linux exports an estimate of how rapidly the queues are changing by reporting how many times items are moved out of the inactive queue; this is done efficiently by counting the

Kernel Task	**C Statements**
Memory-map counter setup	64
Track page movement	1
Reset counter	14
Export `victimList`	30
Total for `victimList` abstraction	**109**

Table 3.1 **Code size for kernel portion of InfoReplace** The number of C statements (counted with the number of semicolons) needed to implement both the `victimList` abstraction and memory-mapped counter within Linux.

User-Level Task	**C Statements**
Setup	4
Simulation framework	720
Target policies	
FIFO	86
LRU	115
MRU	75
LFU	110
Check `victimList` and refresh	251
Total for InfoReplace library	**1361**

Table 3.2 **Code size for user level portion of InfoReplace** The number of C statements (counted with the number of semicolons) needed to implement the InfoReplace library at user-level library.

number of times key procedures are called.[1] This counter is activated only when a service registers interest and is fast to access from user-space because it is mapped into the address space of the user process. Once this counter is approximately equal to N, the process performs the more expensive call to get the state of the last N pages on the inactive queue. As shown Table 3.1, the `victimList` abstraction can be implemented in only 109 C statements; in fact, more than half of the code is needed to setup the memory-mapped counter.

With the `victimList` abstraction, the user-level InfoReplace library can frequently poll the OS and when new pages are near eviction, obtain the list of those pages; if any of these pages should not be evicted according to the target policy, InfoReplace accesses them to move them to the active list. Thus, one of the roles of InfoReplace is to track the pages that would be resident given the target policy. For simplicity, the InfoReplace library currently exports a set of wrappers, which applications call instead of the `open()`, `read()`, `write()`, `lseek()`, and `close()` system calls. Hence, the library only tracks file pages accessed with these explicit calls; however, our interface could be expanded to return access information about each page in the process address space. Thus, on each read and write, the InfoReplace library first performs a simulation of the target replacement algorithm to determine where the specified page belongs in the page queue; InfoReplace then uses `victimList` to see if any of the pages that should have high priority are near eviction and accesses them accordingly. Since InfoReplace does not know the specific replacement policy that the kernel is using, only that it increases the priority of a page when it is accessed, InfoReplace rechecks the `victimList` after accessing all of the high-priority pages that are near eviction. If some of these pages are still on the `victimList` InfoReplace accesses them again. It repeats this procedure until either all of the pages on the `victimList` are non-resident in the simulated target policy, or a fixed maximum number of iterations is exceeded. Thus InfoReplace is capable of running on recency and frequency based policies. In our implementation, this maximum is set to 1000, and during none of our experiments was it exceeded. Finally, the library wrapper performs the requested read or write and returns.

[1] In Linux 2.4.18, these procedures are `shrink_cache` and the macro `del_page_from_inactive_list`.

Following these basic steps, we have implemented FIFO, LRU, MRU, and LFU on top of the Linux 2Q-based replacement algorithm. The bottom half of Table 3.2 shows the amount of C code needed to implement InfoReplace. Although more than one thousand statements are required, most of the code is straightforward, with the bulk for simulation of different replacement policies.

3.4 Evaluation

We evaluate the usefulness of our approach in two ways. First, we measure the overhead incurred and accuracy achieved when using InfoReplace to transform the Linux 2.4.18 2Q-like policy into a variety of other replacement policies. Second, we evaluate the performance improvement achieved on a synthetic workload devised to simulate lookups in a tree-based on-disk index.

3.4.1 Experimental Configuration

Our experimental platform consists of a 2.4 GHz Pentium 4 processor with 512 MB of main memory and two 120 GB 7200 RPM Western Digital WD1200BB ATA/100 hard drives. One drive serves as the system disk, we run our experiments on the other disk. We set N, the size of the `victimList` that InfoReplace requests, to 100. While we did not thoroughly study the effects of varying this value, 100 worked well in practice.

3.4.2 Overhead and Accuracy

To evaluate the overhead and accuracy of our approach, we run the *Dust* workload described in Chapter 2. Recall that this workload accesses a large file (1.5 times the size of memory), touching blocks such that the initial access order, recency, and frequency attributes of each block differ; thus, which blocks are evicted depends upon which attributes the replacement policy considers. We measure the accuracy of the target policy at the end of the run, by comparing the actual contents of memory with the expected contents.

Figure 3.1 shows both the accuracy and overhead of implementing these algorithms in InfoReplace. The graph on the left shows the inaccuracy of InfoReplace, defined as the percentage of

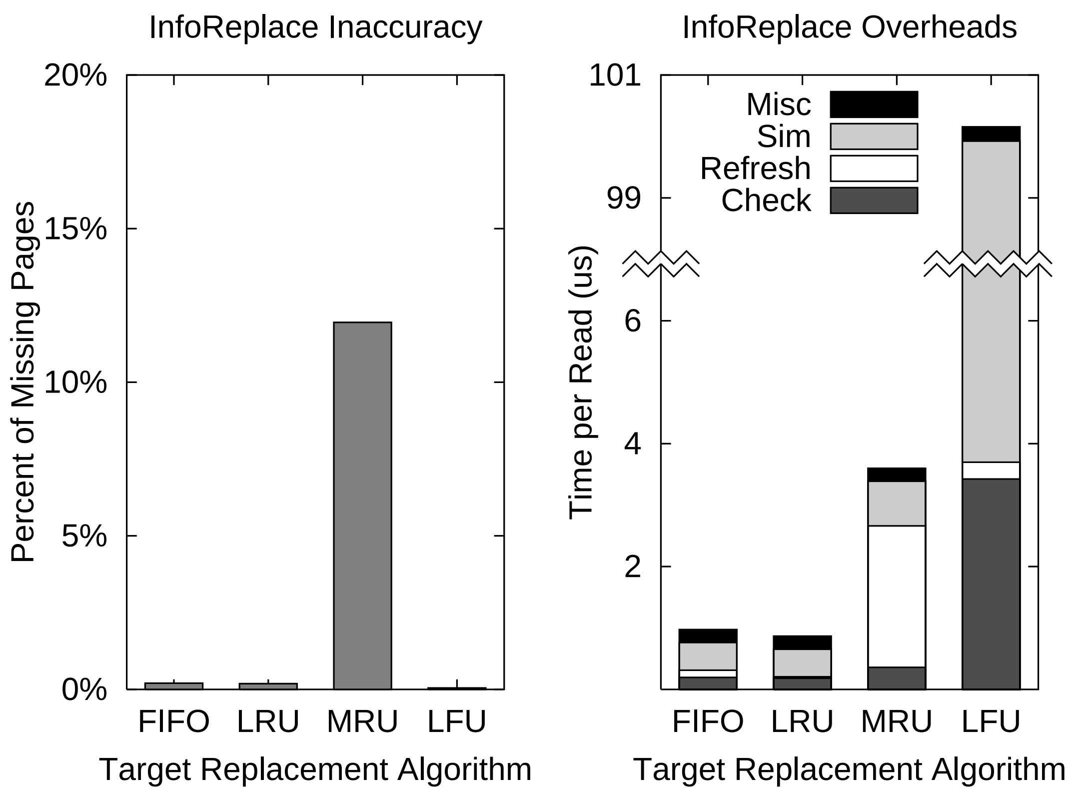


Figure 3.1 **Accuracy and overhead of InfoReplace.** FIFO, LRU, MRU, and LFU have been implemented on top of the 2Q-based replacement algorithm in Linux 2.4. The bar graph on the left shows the inaccuracy of InfoReplace, where inaccuracy is the percentage of pages that are not in memory (but should be) when the workload ends. The bar graph on the right shows the average overhead incurred on each `read()` or `write()`; this time is divided into the time to check the `victimList` abstraction, to refresh the pages that should not be evicted, to simulate the target replacement algorithm, and to perform miscellaneous setup.

pages that are not resident in memory but should be, given a particular target replacement algorithm. By this metric, if four pages A, B, C, and D should be in memory for a given target policy, but instead pages A, B, C, and X are resident, inaccuracy is 25%. In general, the inaccuracy of InfoReplace is low. The inaccuracy of MRU is the highest, at roughly 12% of resident pages, because the preferences of MRU highly conflict with those of 2Q; therefore, when emulating MRU, InfoReplace must constantly probe pages to keep them in memory.

The graph on the right of Figure 3.1 shows the overhead of implementing each policy, in terms of the increase in time per `read()` or `write()` operation; this time is broken down into the time to check the `victimList` abstraction, to probe the pages that should not be evicted, to simulate the target replacement algorithm, and to perform miscellaneous setup. The overhead of InfoReplace is generally low, usually below 4 μs per read or write call. The exception is pure LFU, which incurs a high simulation overhead (roughly 100 μs per call) due to the logarithmic number of operations required per read or write to order pages by frequency. However, assuming that the cost of missing in the file cache is about 10 ms, even the relatively high overhead of emulating LFU pays off if the miss rate is reduced by just 1%.

3.4.3 Workload Benefits

Database researchers have observed that policies provided by general-purpose operating systems deliver suboptimal performance to database management systems [63]. To demonstrate the utility of InfoReplace, we provide a file cache replacement policy inspired by DBMIN [16] that is better suited for database index lookups.

Given that indices in DBMS systems are typically organized as trees, the replacement policy should keep nodes that are near the root of the tree in memory since these pages have a higher probability of being accessed. For simplicity, our policy, PinRange, assumes that the index is allocated with the root at the head of a file and the leaves near the end; therefore, PinRange gives pages preference based on their file offset. Pages in the first N bytes of the file are placed in one large LRU queue, while the remaining pages are placed in another much smaller queue. PinRange is also simple to implement, requiring roughly 120 C statements in the InfoReplace library.

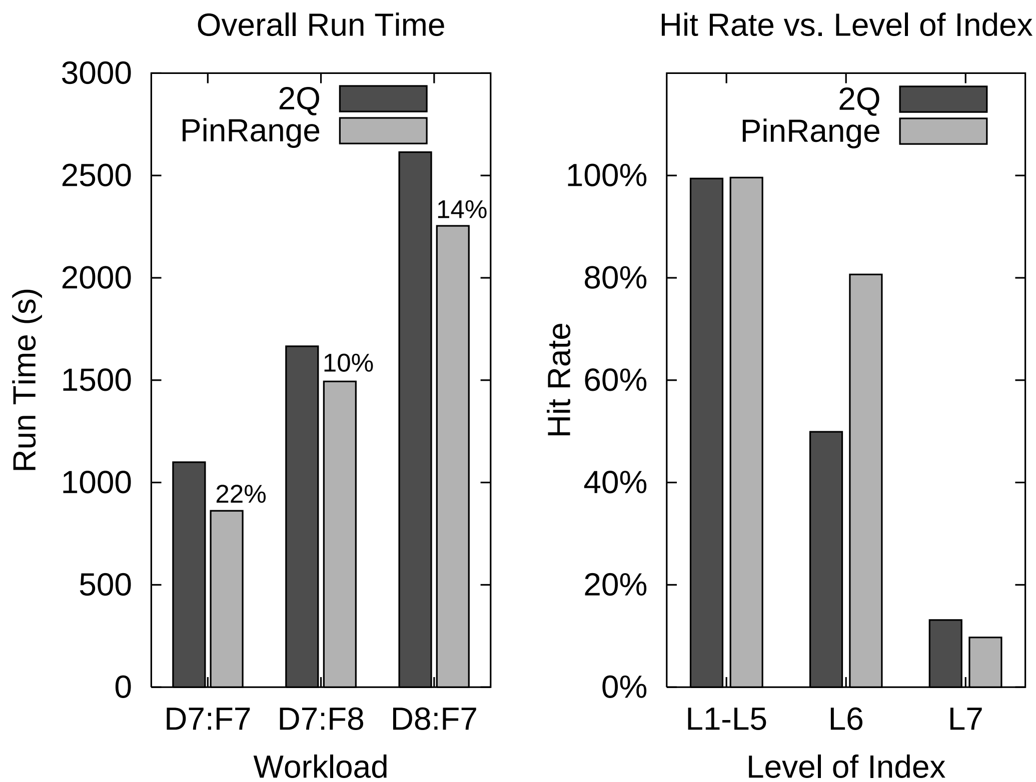


Figure 3.2 **Workload benefits of InfoReplace.** The graph on the left depicts the run-time of three synthetic database index lookup workloads on two systems. The bars labeled 2Q show run time for 100,000 index lookups on the stock Linux 2.4 kernel, whereas the bars labeled PinRange show run time for the specialized PinRange policy on infoLinux. The x-axis varies the workload, specifically the depth and fan-out of the index (*e.g.*, Dx:Fy implies an index of depth x and fan-out of y). The graph on the right shows details of why PinRange speeds up performance for the D7:F7 workload, by showing the hit rate for different levels of the 7-level index.

To demonstrate the benefits of InfoReplace for repeated index lookups, we compare workload run-time using PinRange versus the default Linux 2Q replacement policy. We note that 2Q is already a fairly sophisticated policy, introduced by the database community specifically to handle these types of access patterns [29]; as a result, 2Q gives some preference to pages at the top of the tree.

For our experiments, we run synthetic workloads emulating 100,000 lookups in index trees with seven or eight levels and a fan-out of seven or eight. On a machine with 128 MB of memory, PinRange is configured to prefer the first 90 MB of the file, since 90 MB fits well within main memory. The graph on the left of Figure 3.2 shows that PinRange improves run-time between 10% and 22% for three different trees.

To illustrate why PinRange improves performance over 2Q, the graph on the right of Figure 3.2 plots hit rate as a function of the index level, for a tree with seven levels and a fan-out of seven. The graph shows that PinRange noticeably improves the hit rate for the sixth level of the tree while only slightly reducing the hit rate in the lowest (seventh) level of the tree. This improvement in total hit rate results in a 22% decrease in run-time, which includes approximately 3 seconds of overhead from the InfoReplace library.

3.5 Experience porting to NetBSD

In this section, we describe our initial experience porting our cache information interfaces to NetBSD 1.5. In our discussion, we focus on the main differences between the NetBSD and Linux implementations.

The `pageList` we use in NetBSD is quite similar to that which we used in Linux. Since NetBSD has a fixed-sized file cache, the primary difference between the two implementations is that for Linux, `pageList` contains every page of memory, whereas for NetBSD, it contains only those pages in the file cache. Given that the NetBSD file cache is managed with pure LRU replacement, the NetBSD implementation simply exports this LRU list for `pageList` and the last N elements for `victimList`. To enable processes to quickly determine how the elements are moving in the lists, NetBSD tracks the number of evictions that have occurred from the LRU list. Only

40 C statements are needed to export these abstractions in NetBSD; the primary savings compared to the Linux implementation, which requires 109 statements, is that we have not implemented the memory-mapped interface to the eviction count.

Through this exercise, we have shown that the abstractions we defined for out Linux implementation are straight-forward to implement in NetBSD as well. Thus, we are hopeful that these list-based abstractions are sufficiently general to capture the behavior of other UNIX-based operating systems and that porting this interface to other operating systems will not be difficult. We note that operating systems that export these same interfaces will be able to directly leverage the user-level libraries created for other operating systems, which is where the majority of code resides.

3.6 Conclusions

This case study shows that new replacement policies can be implemented when information is exposed from the OS: the `victimList` abstraction is sufficiently flexible to build a variety of classic replacement algorithms. We believe this compares favorably to direct in-kernel implementations; for example, in Cao *et al.*'s work [14], applications can easily invoke policies that are some combination of LRU and MRU strategies; however, their system has difficulty emulating the behavior of a wider range of policies (*e.g.*, LFU). This case study also illustrates that care must be taken to efficiently perform the conversion from internal state to the general `victimList` abstraction. Furthermore, InfoReplace demonstrates that target replacement algorithms that are most similar to the source algorithm can be implemented with the most accuracy and least overhead.

Modifying the operating system to expose *explicit information* is one way to overcome the inherent limitations of implicit techniques. Exposing information explicitly gives applications easy access to information about the operating system's internal state. The information provided is always accurate and never out of date. By only exposing information, we avoid many of the complications usually associated with modifying operating systems. Since the interfaces presented here are purely read-only, there is no risk that OS internal data structures will become corrupted, thus making it far easier to have confidence in the safety of the new code. All of the code with any significant complexity is kept safely in user-space, where it will only corrupt a single application if it is buggy.

We have presented an interface by which applications can access information concerning the current state of the file system buffer cache. Applications can leverage this information to manipulate the ultimate behavior of the operating system. This is done by observing the current state of the buffer cache and issuing additional reads to bring the OS into the state desired by the application. With these techniques we have demonstrated that the default cache replacement policy used by Linux 2.4 can be transformed into a variety of other replacement policies. We have also ported our interface to NetBSD. We see no reason why the same techniques could not be used on any operating system that provides file system caching.

Information-based approaches as described in this and the preceding chapter provide applications with a great deal of power to modifies their behavior to better fit the policies of the operating system, and to manipulate the behavior of the OS to better fit the needs of the application. Both the implicit and explicit information techniques we have described leverage *implicit control*. That is, they use the existing interfaces provided by the OS to exercise control. This naturally limits the OS behavior modifications that are possible. For example, if Linux used strict a strict FIFO policy to manage the buffer cache, there would be no way to efficiently alter the cache policy since FIFO doesn't increase the priority of pages when they are accessed.

Chapter 4

Controlling Write Ordering

The approaches based on implicit information and explicit information, discussed in Chapters 2 and 3 have in common that control over the operating system is applied using implicit techniques. Even in the case where information is exposed explicitly, no interfaces were added for altering the behavior of the operating system. While this provides a larger degree of control than was previously available, in some cases implicit techniques are not powerful enough to exercise the needed control, or are not suited to exercising that control in an efficient manner. This chapter describes one such instance, controlling the order in which data is committed to stable storage, and two interfaces for allowing the application the control it needs. We show that the interface exposed to the application can make a tremendous difference in the performance and usability of a feature.

4.1 Write Ordering

At every level in the storage stack, write requests may be reordered to optimize performance. In fact, the order in which the application submits writes to the operating system is typically not even recorded. Many factors determine the order in which data is finally written to the storage device. When an application sends a write request to the kernel, the update is applied to a copy of the data in the file system buffer cache, then the write call returns. At this point the operating system is free to write the data to disk at its own convenience. Assuming the system doesn't crash, the new data will eventually be written to disk. How and when that happens depends on a number of factors. If the application calls `fsync()` or `sync()`, the data will be flushed to disk immediately. If there is memory pressure, the data may be flushed as a part of the cache replacement algorithm.

Also, most systems will assure dirty data is written within some defined period of time (30 seconds in many systems). Once requests are sent to the disk scheduler, they will be reordered to maximize the performance of the disk. Depending on the policies implemented in the operating system, data sent to the kernel to be written to disk can end up being committed to disk in nearly any conceivable order.

4.1.1 Motivation

For most applications, this situation is acceptable. The performance benefit gained by allowing reordering far outweighs any benefits of restricting reordering. However, for certain classes of applications the order in which data is committed to disk is critical.

Any application that is concerned with the consistency of the data it writes to disk will be concerned with the order in which data is committed to the media. Complex on-disk data is likely to contain references to other parts of the data set. Ordering is necessary to ensure that, in the event of a crash, these references do not point to data that was never committed to disk. For example, FFS-like file systems ensure that data blocks are written to disk before inodes and indirect blocks that point to them [38]. In the application space, CAD/CAM systems often use large datasets with ordering requirements similar to database systems [8].

Perhaps the most common example of the need for ordering is write-ahead logging [24], used by journalling file systems [12, 51, 64, 68] and database management systems [40]. These systems maintain the write-ahead invariant: the log entry for a transaction will be written to stable storage strictly before the data updates for that transaction. This guarantees that no there will be no data updates applied which aren't described in the write-ahead log; allowing the system to recover to a consistent state in the event of a crash. This is the example we focus on here.

4.1.2 Current Ordering Strategies

Currently applications have several choices on how to deal with consistency when running on a modern OS. Each of these options sacrifices at least one of performance, manageability or reliability.

Perhaps the simplest option that maintains ordering is to use `fsync()` to force data to disk when necessary to preserve ordering. While this does accomplish the goal of maintaining the correct order, it also slows down the application since these calls are synchronous. It isn't necessary to use a synchronous call in this case since we don't care *when* the data reaches the disk, only that it gets there in the correct *order*.

The application can access the raw disk device, bypassing the file system entirely. This has a small performance advantage and allows the application to explicitly control the ordering in which blocks are sent to the disk. In exchange for this additional performance, the user sacrifices some convenience in management. By running on top of the raw disk, the administrator loses the use of file system and data management utilities which depend on the file system. Users are now dependent on the application authors to provide reimplementations of backup utilities, space management utilities, and so forth. In general, applications are easier to set up and administer when run on top of a file system. At least one commercial DBMS manufacturer recommends using the file system for smaller databases for this reason [28]. Further, interfaces to access raw storage devices are not well standardized. The users choice of operating system is limited to those the application vendor supports the raw device interface on.

The final option is to run on top of the file system and not use any ordering control. Without guaranteed ordering, the users and administrator have no guarantee that the application data will be consistent, or even recoverable, when the system comes back after a crash. The popular open source DBMS, PostgreSQL, runs on top of the file system and allows the administrator to choose whether to use `fsync()`or no ordering control [50].

The choice is between manageability, performance, and reliability. The file system provides certain management conveniences which we would like to preserve. Raw storage access performs well but can be difficult to manage. Using no ordering control is fast and easy to manage, but unreliable in the face of failures. We would like a system that is fast, reliable and allows the file system to be used for storage management.

4.1.3 New Ordering Strategies

One solution is to provide a way for applications to express ordering constraints to the operating system. To this end we propose *file system barriers*. File system barriers provide an additional `barrier()` system call to applications. When `barrier()` is called, the operating system guarantees that all file system writes issued before the call to `barrier()` will be committed to stable storage strictly before any writes issued subsequently. File system barriers have the advantage that a call to `barrier()` is a direct, asynchronous, replacement for a call to `fsync()`. Thus it is very easy to convert an application that uses `fsync()` for ordering to use file system barriers. However, file system barriers require keeping most logical writes physically separate. That is, if the same piece of data is written to twice, with a barrier between the two writes, two physical disk writes are required to maintain the correct semantics. This results in more disk traffic when file system barriers are used when compared to other ordering mechanisms. To address this limitation, we propose *asynchronous graphs*. Asynchronous graphs is an interface that allows an application to specify ordering constraints among its write requests in a very fine-grained manner. Ordering is enforced only where necessary, leaving the operating system free to combine many logical writes to a piece of data into one physical write in most cases. The operating system also has a greater degree of freedom to reorder write requests with asynchronous graphs since the ordering constraints are specified at a very fine granularity.

Since our primary target application is database management systems, it is necessary to examine the effect of using file system barriers or asynchronous graphs on traditional database semantics [8, 25]. By allowing ordering to be controlled despite all of the IO being done asynchronously, we sacrifice one aspect of traditional ACID semantics, durability. ACID semantics guarantee that each transaction is executed atomically, that is, either all of the transactions updates are applied, or none of them are. An application which provides atomicity when using `fsync()` will still provide it correctly when using file system barriers or asynchronous graphs. Whether or not a transaction's updates are considered to be effective is determined by the presence or absence of that transaction's commit record in the log. If the commit record is present, all of the transaction's updates, which are also recorded in the log, apply. If the commit record is absent, none of those updates apply. Similarly, consistency, the assurance that if the database is in a consistent state when a transaction starts, it will be in a consistent state when the transaction ends, is not effected by using asynchronous ordering control. Isolation guarantees that no transaction will see another transaction's updates until that transaction commits. In a DBMS, this guarantee is provided by two-phase locking internally. The only time using asynchronous ordering control may affect isolation occurs in the event of a crash. Some transactions that had committed prior to the crash, and whose results thus became visible to other transactions, may disappear after recovery if their log records hadn't been written to disk yet. With true ACID semantics, the application is assured that once the DBMS reports that the transaction has successfully committed, that data will not be lost. Since both file system barriers and asynchronous graphs allow IO calls to return before any data is written to disk, it is possible that data will be lost in the event of a system crash. What these two new interfaces provide is that, if properly used, the database will be recoverable to a consistent state in the event of a crash. Though some transactions might be lost, they will be lost or recovered atomically. Our results show that by making this sacrifice, it is possible to achieve performance very near that achieved using no ordering control at all. When no ordering control is use, the ability to recover the on-disk data to a consistent state is not guaranteed and so reliability in the face of system failures is sacrificed.

With the current techniques used to control ordering, users are forced to choose between manageability, performance and reliability. By introducing asynchronous mechanisms to control ordering, we offer a new point in this space that the user might choose. Applications can run on the file system, maintaining ease of administration. As we will show in the following sections, the performance of asynchronous graphs is competitive with the performance when using no ordering control. As discussed above, some reliability, namely durability, is sacrificed, but consistency is preserved. While it is not possible to implement strict ACID semantics using the mechanisms described in this chapter, it is possible to provide a great deal more reliability than is possible without any ordering control. With these mechanisms, asynchronous graphs in particular, it is now possible to sacrifice a small amount of reliability for a significant improvement in performance.

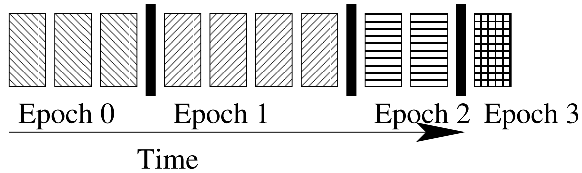


Figure 4.1 A number of writes broken into epochs by calls to `barrier()`.

4.2 File System Barriers

Like architectural memory barriers, a file system barrier ensures that all writes requested before the barrier are committed to disk before any of the writes requested subsequent to the barrier. Unlike architectural barriers, we needn't worry about read requests; since `read()` doesn't return until the data is read into memory, reads are always ordered as issued. The buffer cache will ensure that all processes and threads see the same data at the same time. File system barriers allow an application to specify write ordering without using any synchronous system calls and while still using the file system. Thus the user can have a fast and reliable system without sacrificing manageability.

4.2.1 Semantics of File System Barriers

File system barriers can be thought of as an asynchronous version of `sync()`. All writes issued by the application before a call to `barrier()` will be committed to stable storage strictly before subsequent writes. The interface is simple: a single additional system call, `barrier()`. Existing applications can be converted to use file system barriers for ordering by simply replacing calls to `fsync()` with calls to `barrier()`.

A typical application will call `barrier()` many times as it updates its on-disk data. These repeated calls to `barrier()` group writes into *epochs*. An epoch is defined as the set of all writes issued between one call to `barrier()` and the next call to `barrier()`. Writes can be reordered *within* their epoch, but cannot be reordered *across* epochs. There are two epochs that are of particular importance. The *safe epoch* is the earliest epoch for which there are still dirty buffers in the system. Writes in this epoch, and this one only, can be safely flushed to disk. The collection of writes that were issued after the most recent call to `barrier()` constitute the *open epoch*. The open epoch is the only epoch to which writes can be added. All epochs that are not the open epoch are said to be *closed*. Figure 4.1 illustrates a number of write operations, broken into epochs by three calls to `barrier()`. In the figure, Epoch 0 is the safe epoch, Epochs 1 and 2 are closed and Epoch 3 is the open epoch.

Consider an application that performs write-ahead logging and uses `fsync()` to maintain ordering between the log and the data. An application might use code similar to that shown in Figure 4.2 to implement write-ahead logging. First the log is updated and that update is forced to disk, then the data is updated. Note that the updates to the data are applied in the buffer cache only at this point, the operating system will commit the new data to disk at its leisure. The log data, however, is safely on disk when the call to `fsync()` returns.

Figure 4.3 shows the same transaction implemented using file system barriers. When this code completes execution, there is no assurance that any of the data, or the log updates, are on disk. But it is guaranteed that the write to the log will be flushed disk before the the writes to the data. Thus, in the event of a system crash, this transaction may or may not be lost, but it will be possible to use whatever portion of the log is on disk to bring the entire data set into a consistent state.

4.2.2 Implementing File System Barriers

The `barrier()` system call implementation is fairly straightforward. The entire `barrier()` call takes place under a global lock. It isn't clear what the proper semantics would be for two barriers executing at the same time, so that situation isn't permitted. For each call to `barrier()`, the kernel creates an `epoch_list` structure and assigns it a unique, monotonically increasing `epoch_id`. It then traverses the lists of dirty buffers and adds each buffer to the `epoch_list` and puts the `epoch_id` in a special field in the buffer header. If that buffer is listed on a previous `epoch_list` it is skipped. The new `epoch_list` is then added to a global list of all the `epoch_list` structures currently in the system. By default, each new buffer is marked as not being a part of any barrier epoch.

```
/* first transaction */
/* write an entry to the log */
write(log, logbuf, 128);
fsync(log);
/* write some data pages */
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf, datasize);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf2, datasize2);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf3, datasize3);

/* second transaction */
/* write an entry to the log */
write(log, logbuf, 128);
fsync(log);
/* write some data pages */
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf, datasize);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf2, datasize2);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf3, datasize3);
```

Figure 4.2 Code to execute two transactions with default workload parameters using `fsync()`.

```
/* first transaction */
/* write an entry to the log */
write(log, logbuf, 128);
barrier();
/* write some data pages */
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf, datasize);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf2, datasize2);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf3, datasize3);

/* second transaction */
/* write an entry to the log */
write(log, logbuf, 128);
barrier();
/* write some data pages */
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf, datasize);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf2, datasize2);
lseek(datafd, random_offset(), SEEK_SET);
write(datafd, databuf3, datasize3);
```

Figure 4.3 Code to execute two transactions with default workload parameters using file system barriers.

To flush a buffer to disk, the OS first must flush all of the buffers that come before it in the barrier ordering. It is important to observe that in most cases when the operating system needs to write the contents of a buffer to disk, it doesn't need to write any specific buffer. If buffers are being laundered to relieve memory pressure or keep the number of dirty buffers below some threshold, the OS can choose which buffers to send to the disk. In these cases, it is wise to choose buffers from the safe epoch.

A complication arises if there is a write to a buffer, a barrier, then another write to the same buffer, without an intervening flush of the buffer. In the implementation as described so far, the writes will be combined, either before or after the barrier operation depending on whether the buffers `epoch_id` is updated or not. This is not the correct behavior: write operations should never logically move between epochs.

To prevent this situation, separate logical writes to the same buffer must remain physically separate in memory if there is a barrier operation between them. When the second write is issued, the kernel notices that that buffer is already behind a barrier. To make sure the writes stay separate, a new buffer is allocated and the old version of the data is copied into it and placed in the appropriate `epoch_list`. The write is then allowed to proceed normally. Now two versions of the data exist in the buffer cache, each in a different epoch. The normal procedure for maintaining barrier ordering will ensure that the old version is written first.

There are three disadvantages to doing this. First, the `write()` call will take longer whenever it has to copy a buffer. Second, memory usage increases due to potentially having many versions of one disk block in memory. Finally, disk traffic increases due to the additional writes.

The only alternative to duplicating buffers is to synchronously flush the old version of the data as part of the `write()` system call. We choose copying the buffer rather than flushing it since one of our goals is to remove excess synchrony from the system. Since the old version of the buffer will never be written to or read from again, an asynchronous write to disk is initiated on it as soon as the copy is created.

The increased memory usage can be mitigated by issuing an asynchronous write on the buffer containing the old version of the data at the time the copy is made. Flushing the old version immediately both lets the application continue running at memory speed and ensures that the additional buffer space will only be in use a short time. Once the write is complete, the buffer is released. The evaluation will show that this is an effective way to reduce the impact of copying buffers.

Disk traffic will be increased because the operating system must keep logical writes separate where in a conventional system they would be combined into a single physical write. This problem is due to the semantics of the barrier operation.

These three issues can be alleviated by using a more fine-grained interface to express ordering, at the expense of ease of programming.

4.3 Asynchronous Graphs

In order to address some of the limitations of file system barriers, we introduce a new interface for controlling write ordering, *asynchronous graphs*. Asynchronous graphs allow the application to specify ordering constraints at the level of individual calls to `write()`. Writes are not grouped into epochs and the application is able to express ordering precisely where it is needed. This is similar to the way soft-updates tracks dependencies between various types of file system metadata blocks [22]. Soft-updates, however, exploit semantic knowledge of the data it is managing that is not available to the operating system in the general case addressed here. This more fine-grained interface reduces the number of extra writes necessary to maintain correct ordering (to zero in some cases) and reduces the need for buffer copying (to zero in most cases). Improved performance comes at the expense of programming ease. Ordering dependencies between writes must be tracked by the application and passed along to the operating system.

4.3.1 Semantics of Asynchronous Graphs

Asynchronous graphs introduces a modified `write()` call, `graphwrite()`. This new system call returns an integer to identify the write operation each time it is called. These identifiers can then be passed into subsequent calls to `graphwrite()` to specify ordering constraints. The C

```
typedef int write_id;
write_id graphwrite(int fd, void * buf, ssize_t size,
                    int ndeps, write_id *dependencies)
```

Figure 4.4 **C language signature for** `graphwrite()`.

language function signature for `graphwrite()` is shown in Figure 4.4. The first three arguments are the same as the traditional `write()` system call: a file descriptor, a buffer containing the data to be written, and the size of the data to be written. The two additional arguments, `ndeps` and `dependencies`, allow the application to specify a set of previous writes that must be committed to disk before the current one. The `dependencies` parameter is an integer array which lists the relevant write identifiers. The `ndeps` parameter gives the length of `dependencies`.

Figure 4.5 shows two simple WAL transactions using asynchronous graphs. The first call to `graphwrite()` updates the write-ahead log and records the identifier for that write operation. No write identifiers are passed in since the writes to the log have no ordering dependencies. The subsequent calls perform the updates to the data itself. These calls each pass in the identifier for the write to the log. This additional argument specifies that the update to the log must be applied before any of the data updates. Notice that there are no dependencies specified between the data updates, so the operating system is free to reorder them amongst themselves. The code then performs another write to the log for a second transaction. The data updates for the second transaction are then applied.

Figure 4.6 shows the dependency graph generated by the code in Figure 4.5. Arrows show the "must be written to disk before" relationship. The two transactions are completely independent of each other. Depending on the policies of the kernel, the write operations involved may be reordered in a variety of ways.

The dependency graph as perceived by the user-level application and the dependency graph as implemented within the operating system may be different. For example, if the two log updates from Figure 4.5 happened to be to the same buffer internally, the nodes representing the log updates will be internally combined. The resulting graph is shown in Figure 4.7. Similarly, if some of the

```
int logid;

/* first transaction */
/* write an entry to the log */
logid = graphwrite(logfd, logbuf, 128, 0, NULL)
/* update the data */
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf, datasize, 1, &logid);
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf2, datasize2, 1, &logid);
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf3, datasize3, 1, &logid);

/* second transaction */
/* write an entry to the log */
logid = graphwrite(logfd, logbuf, logsize, 0, NULL);
/* update the data */
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf, datasize, 1, &logid);
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf2, datasize2, 1, &logid);
lseek(datafd, random_offset(), SEEK_SET);
graphwrite(datafd, databuf3, datasize3, 1, &logid);
```

Figure 4.5 **Two simple transactions using asynchronous graphs.** Each log write has no ordering dependencies. The data writes are required to be written after the log writes for the transaction that touches them.

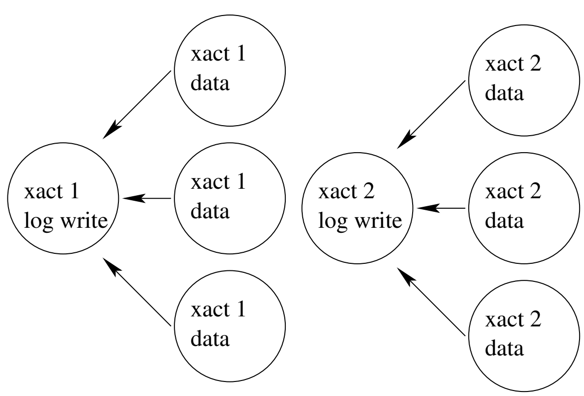


Figure 4.6 **The graph generated by two simple transactions.** Data writes are required to be committed to disk after writes to the log, other reorderings are permitted.

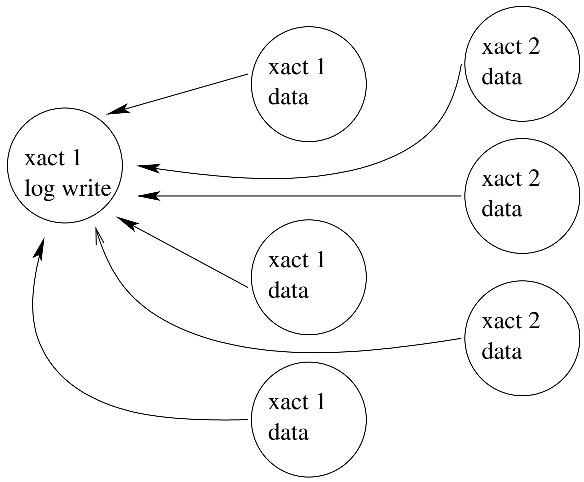


Figure 4.7 **The graph generated by two simple transactions if the log writes are to a common buffer.** Since there are no writes required to be written to disk in between the two writes to the log, the log writes can be safely combined.

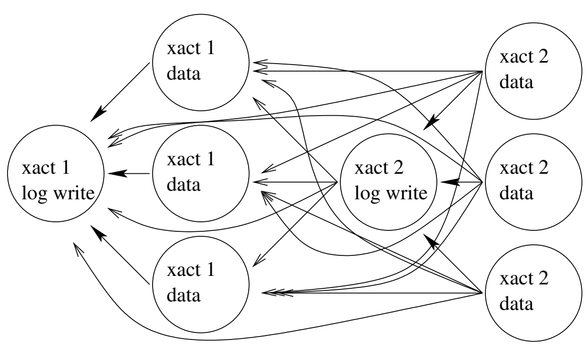


Figure 4.8 **A graph representation of two transactions with file system barrier ordering semantics.** File system barriers impose numerous unnecessary ordering dependencies.

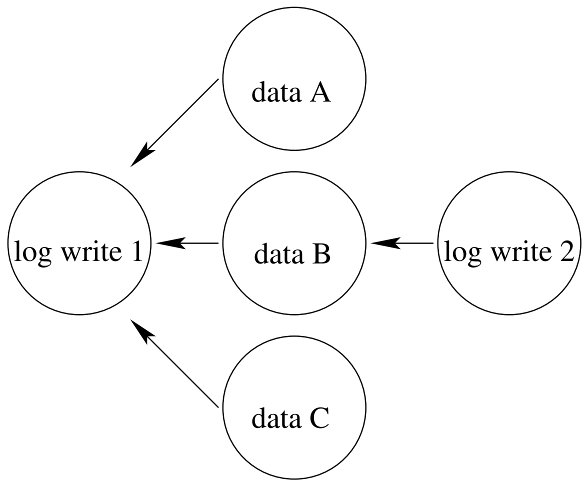


Figure 4.9 **A graph with writes that cannot be combined.** The two writes to the log cannot be combined without incorrectly ordering one of them relative to the write of data B.

data writes are to common file system buffers, those nodes in the graph can be combined as well. Writes are combined in this way in order to reduce the needs for duplication of buffers.

There are situations where two writes to the same buffer cannot be combined. If there is at least one write operation required to be performed between them, they cannot be combined. If both writes are part of the same connected subgraph and they are not adjacent, then they cannot be combined. Figure 4.9 shows a graph for such a situation. This is the same as the example transactions previous described with the addition of a second log write ordered *after* one of the data writes. While we don't expect this situation to occur often in practice, it serves for illustration. The two writes to the log cannot be safely combined. If they were, and the write to disk occurred before the the write of data B, then the ordering would be incorrect since log write 2 would be on disk before data B. Similarly, if the two log writes were combined and the write to disk occurred after data B, the ordering would also be incorrect since log write 1 must be on disk before data B. Therefore, the two log writes must be executed as separate physical writes to disk. In this case, the same techniques that are used to keep writes separated in file system barriers can be used. Either the old version can be flushed, or the buffer can be duplicated. In the workloads presented in this chapter, this situation never occurs.

Notice that file system barrier semantics are expressible in terms of the asynchronous graphs interface. When issuing the writes for epoch N using `graphwrite()` every write is specified as being after every write from epoch $N-1$. Figure 4.8 shows the dependency graph for two transactions where the ordering is controlled using file system barrier semantics. This illustrates the extra dependencies imposed by file system barrier semantics. One could imagine other ordering interfaces being implemented on top of asynchronous graphs through a user-level library.

4.3.2 Implementation of Asynchronous Graphs

4.3.2.1 Data Structures

The data structures necessary to implement asynchronous graphs are considerably more complex than was necessary to implement file system barriers. This additional complexity is due to the more fine-grained level of control and the additional freedom that the operating system has to combine and reorder write requests. The operating system must have a representation of a node in the dependency graph. We term this structure a `write_node`. Each `write_node` includes a unique integer to identify it, a list of the dirty buffers associated with that node, and a list of write operations that must be performed before it (*i.e.* outbound edges). There is also a table which maps `write_node` identification numbers to pointers to the actual `write_node` structures, the `writemap`. This table becomes important when it comes time to enforce correct ordering.

When a new write operation is requested by an application with `graphwrite()` the operating system performs a number of steps to correctly add the new write to the existing dependency graph. In the simplest case, all of the buffers being written are clean. In this case, a new `write_node` is allocated, the buffers being dirtied are added to its buffer list and the list of outbound edges is populated from the `graphwrite()` arguments. Then the requested updates are applied to the relevant buffers as in a conventional OS.

If the buffers being updated are already dirty, the operating system needs to determine if any of the buffers need to be duplicated. Recall that a buffer needs to be duplicated if it is already present in the connected subgraph containing the new write operation and is not adjacent to the new write operation. Thus, the operating system traverses the graphs rooted at each of the write operations the new write depends on. If the buffer that the current write updates is found in any of those subgraphs and is not the root, then the buffer is duplicated. The original buffer remains in its current position in the graph and the copy is updated by the pending write. The buffers are then updated and `graphwrite()` returns to the application. In any of these cases, the value returned to the application by `graphwrite()` is the identifier for the `write_node` created or affected by the call.

Further complications arise if the write affects multiple buffers, those buffers are already dirty and some of them belong to different `write_node` structures. Suppose a call to `graphwrite()` will dirty buffers A and B. Suppose that A is already part of `write_node` a and B is part of `write_node` b. If there is no path between a and b then it is safe to combine them into a single `write_node` since there are no ordering requirements between them. The `writemap` is then updated to map the identifiers for a and b to the common `write_node`. This mapping must be maintained since the application is not aware that the two `write_node` structures have been combined and may still use either identifier to refer to it. The `write_node` also includes a field that lists all of the `write_node`'s aliases. This list is used when all of the buffers in a given `write_node` are finally written to disk so the additional mappings in the `writemap` can be garbage collected.

4.3.2.2 Enforcing Ordering

In most cases, no additional disk traffic is necessary to maintain a correct write ordering. Since ordering is only specified where it is absolutely necessary, it is often the case that any given buffer has very few or no buffers that need to be written prior to it. Also, when the operating system needs to launder buffers, it is most often the case that any buffer will do, so the OS chooses a buffer that can be written without any extra work. Lastly, a buffer flushing daemon, whose behavior we modify slightly, tends to keep dependency graphs from getting very deep. A typical buffer flushing daemon traverses the entire buffer cache periodically, flushing every buffer that has been dirty for more than a fixed period of time (often 30 seconds). We modify this behavior so the daemon skips buffers with unsatisfied ordering constraints. That is, it only flushes buffers that are part of leaf nodes in the dependency graph. This simplifies the implementation of the daemon since the buffers it skips in one pass will be written during a subsequent pass.

When it is necessary to flush a buffer to disk, the operating system examines the `write_node` with which the buffer is associated. If the list of dependencies in the `write_node` is empty, then the buffer is safe to launder. If the list is non-empty, then it is possible that it is not safe to launder the buffer. If all of the pages in all of the `write_node`'s listed have been flushed, then the buffer is, in fact, safe to write. Since a `write_node` is deallocated when all of its pages have been flushed, the `writemap` is used to indicate whether the specified `write_node` still exists. When the last buffer listed in a `write_node` is flushed to disk, the `write_node` is deallocated, and all mappings to it are removed from the `writemap`. This level of indirection allows the memory used by `write_nodes` to be safely deallocated.

If the operating system finds that it is safe to flush the buffer to disk, the write is initiated and proceeds normally. If the OS finds that there are still dirty buffers in memory that must be written before the current one, then there is a choice to be made. If it doesn't matter which buffer gets laundered, as is the case when buffers are laundered due to memory pressure, the OS simply selects another, safe, buffer to launder.

If a particular buffer needs to be flushed, as is the case during a call to `fsync()` then all of the buffers ordering dependencies must first be satisfied. In this case, the OS begins traversing the dependency graph rooted at the `write_node` the given buffer is associated with. It will issue writes on all of the pages in the leaf nodes and as these writes complete, working its way back up to the required `write_node`. Having to do this will be relatively rare in practice; much of the need for using `fsync()` will be eliminated by the availability of an ordering interface. In the case of a call to `sync()`, the operating system makes repeated passes through the buffer cache, on each pass writing whatever buffers have no ordering constraints. After N passes, where N is the maximum depth of any connected subgraph, all of the buffers will have been written in a safe order.

4.4 Evaluation

We evaluate file system barriers and asynchronous graphs in trace-driven simulation. This allows exploration of a broader range of potential system configurations than a real implementation would. It also allows broad parameter studies to be conducted relatively quickly. We want to determine what aspects of the workload effect the performance of each of the ordering mechanisms we study.

4.4.1 Simulator

The simulator uses an event driven architecture. There are 23 event types representing system calls, memory operations, disk IO, buffer duplication and a buffer flushing daemon. Reads and writes to memory take a constant $2\mu s$, disk access times increase linearly with block distance, with a minimum access time of $1ms$ and a maximum of $10ms$. This simple disk model is adequate for our purposes since most of the differences in performance are due to the number of writes performed, not the specific locations on the disk. Simulated files are laid out back to back starting at offset zero. Results are presented using FIFO and C-LOOK disk scheduling. C-LOOK scheduling begins at block zero of the disk and sweeps across the disk servicing requests as it goes. When the end of the disk is reached, the algorithm returns to block zero and begins again. No requests are serviced when the disk head is moving from the end of the disk, back to block zero. Results using LOOK and Shortest Seek Time First (SSTF) are substantially similar to C-LOOK and aren't discussed further.

When controlling ordering asynchronously, the operating systems policy for flushing dirty buffers is critical. Current systems usually assume that all buffers cost roughly the same to flush. When ordering constraints are present, this is no longer the case. If there are two dirty buffers in the system and one must be written before the other, the cost of flushing one of these is double the cost of flush the other. So, in our simulator, when it is necessary to flush a buffer to disk, we always choose one with no ordering constraints (*i.e.*, there are no buffers that must be flushed before it). Note that such a buffer always exists. Our simulation also includes a buffer flushing daemon. Every 10 simulated seconds, the buffer flushing daemon scans the buffer cache and flushes any buffers that have been dirty for at least 30 seconds. The buffer daemon skips any buffers that cannot be immediately flushed due to ordering constraints. We do this to keep the buffer daemon simple, noting that if no additional ordering dependencies are added, every buffer will eventually become free of ordering dependencies.

In the default configuration, we simulate cache of 312 MB, or, 20000 pages of 16 KB each. Variations of these parameters are noted where appropriate.

4.4.2 Controlled Workload

The first set of experiments were conducted using synthetically generated traces. Synthetic traces allow easy modification of various aspects of the workload and allow the effects of those changes to be easily observed without being convoluted by the complexities of a more realistic workload.

4.4.2.1 Workload

The workload is based loosely on TPC-B [66]. Using the default parameters each transaction writes 128 bytes to a write-ahead-log, then reads one byte from each of three randomly selected offsets, then writes one byte each to each of three randomly selected offsets. The random offset range from zero to 615 MB. A write to an uncached page incurs the cost of a disk read. To keep this synthetic workload simple and easy to understand, the *History* relation normally present in TPC-B is omitted.

The size of the log write, the number of pages read and the number of pages written are varied in the experiments where specified. Table 4.1 shows the default values for each workload parameter and the range over which each parameter is varied.

Depending on the mechanism being tested, an `fsync()` or `barrier()` may be issued after the write to the log. If asynchronous graphs is being tested, then each write is specified as being

```
/* write an entry to the log */
write(log, logbuf, 128);
fsync(log); /* or barrier() or nothing */
/* read some data pages */
for (i = 0; i < 3; i ++) {
  lseek(data, random_offset(), SEEK_SET);
  read(data, buf, 1);
}
/* write some data pages */
for (i = 0; i < 3; i ++) {
  lseek(data, random_offset(), SEEK_SET);
  write(data, buf, 1);
}
```

Figure 4.10 **Code to execute one transaction with default workload parameters using `fsync()` or file system barriers.** The call to `barrier()` directly replaces the call to `fsync()`. When using no ordering control, both `fsync()` and `barrier()` are omitted.

dependent on the preceding log write. In these experiments, the workload always consists of 10000 transactions except where specified and concludes with a call to `sync()`.

In the case where `fsync()`, barriers, or no ordering is being used, the workload is described by the pseudocode in Figure 4.10. The situation is slightly more complicated in the case of asynchronous graphs. In this case, we use a new system call `graphwrite()` described in Section 4.3. Figure 4.11 shows C-like pseudocode for one transaction using asynchronous graphs.

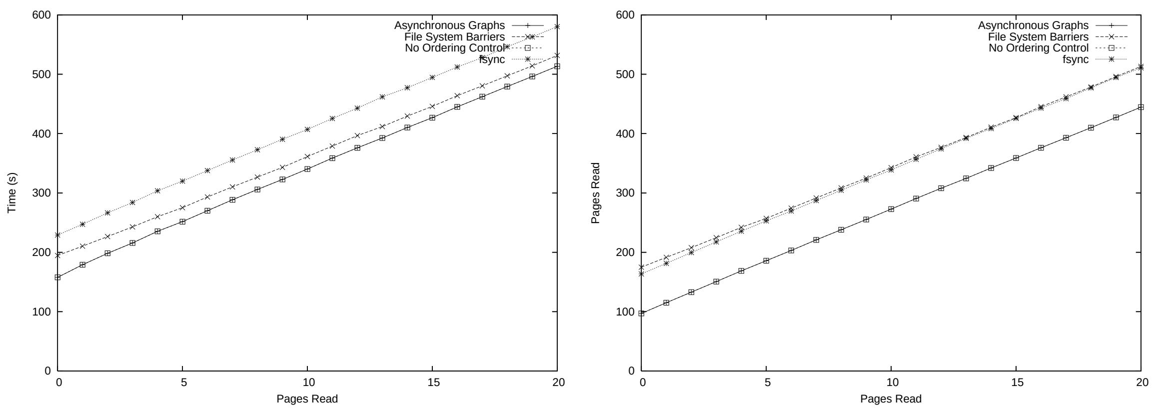
We conducted five groups of experiments. In each group, one of the following was varied: number of pages read in each transaction, number of pages dirtied in each transaction, the size of the write to the log, the number of transactions executed and the total size of the buffer cache. Table 4.1 summarizes the parameters varied and the range over which they were varied. Parameters are always varied one at a time, holding the others constant at their default values.

```
write_id log_write;
/* write an entry to the log */
log_write = graphwrite(log, logbuf, 128, 0, NULL);
/* read some data pages */
for (i = 0; i < 3; i ++) {
  lseek(data, random_offset(), SEEK_SET);
  read(data, buf, 1);
}
/* write some data pages */
for (i = 0; i < 3; i ++) {
  lseek(data, random_offset(), SEEK_SET);
  graphwrite(data, buf, 1, 1, &log_write);
}
```

Figure 4.11 **Code to execute one transaction with default workload parameters using asynchronous graphs.** The updates to the data are made dependent on the writes to the log. The data updates are unordered among themselves.

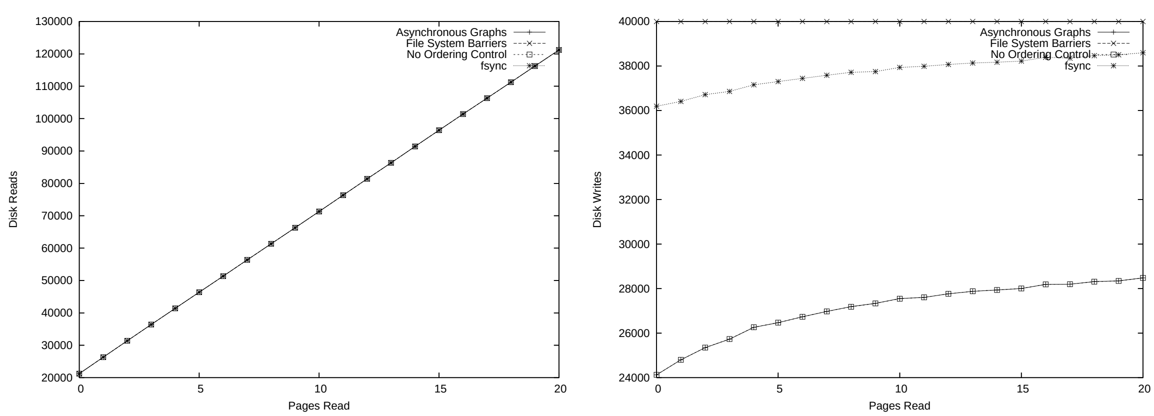
Parameter	Default	Range	Step
Pages Read	3	0-20	1
Pages Written	3	1-20	1
Log Write Size	128 B	64 B - 16 KB	64 B
Transactions	10000	5000-1000000	5,000
Cache Size	312 MB	16 MB - 781 MB	16 MB

Table 4.1 **Experimental Parameters** Parameters of the synthetic workload and the range over which they are varied.



(a) FIFO Disk Scheduling (b) C-LOOK Disk Scheduling

Figure 4.12 **Simulated execution time as the number of pages read per transaction increases.** the line for asynchronous graphs is collinear with the line for no ordering control.



(a) Disk Reads (b) Disk Writes

Figure 4.13 **The number of disk reads and writes as a function of the number of pages read in each transaction.** In (a) all four lines are collinear. In (b) the line for asynchronous graphs is collinear with the line for no ordering control.

4.4.2.2 Pages Read

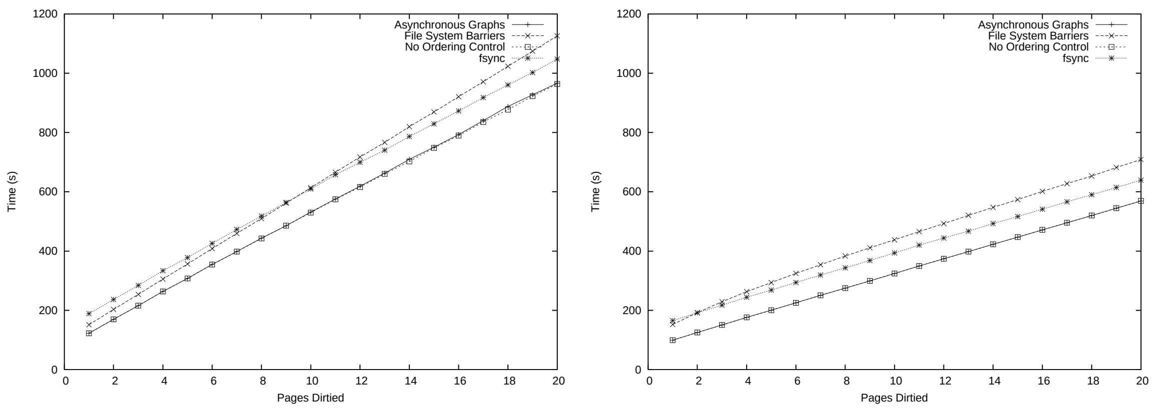
Adjusting the number of pages read essentially adjusts the pressure on the buffer cache. This group of experiments is presented first since it is the simplest and illustrates some trends that will be present in many of the subsequent experiments.

As the number of pages read increases, the execution time increases, as does the total disk IO. Figure 4.12 shows the simulated execution time as a function of the number of pages read during each transaction. The graph on the left shows the execution time with a FIFO disk scheduler and the graph on the write shows the execution time with a C-LOOK scheduler. The performance of all of the ordering mechanisms improves with the better disk scheduler. The performance of file system barriers improves less than the other methods. File system barriers groups writes into epochs, and in order to maintain the correct ordering, only one epochs worth of writes can be sent to the disk scheduler at a time. In this workload, each epoch contains only four disk writes. As a result, there is only a small amount of improvement that can be gained by rescheduling such a small number of disk writes.

Naturally, the increase in execution time is mostly a function of the number of disk reads, which increases as the number of pages read per transaction increases. This is shown in Figure 4.13 in the left-hand graph. The right graph shows that the number of disk writes incurred is dependent on the ordering mechanism being used.

The barrier mechanism incurs the most disk writes and is constant at 40000. File system barriers force writes to the same buffer in different transactions to be executed as separate writes to the disk. The number 40000 comes from three page writes and one log write in each of the 10000 transactions. Using `fsync()` incurs the second greatest number of disk writes. `fsync()`, like barriers, keeps writes to the log separate, but allows multiple writes to the same data page to be combined into one disk write. Thus, the number of writes increases as cache pressure increases since when a buffer is written twice, it more often happens that the page is evicted between the first and second writes.

The lowest line shows the number of writes when either no ordering control is used or asynchronous graphs is used. These incur the same number of disk writes since they both allow logical



(a) FIFO Disk Scheduling

(b) C-LOOK Disk Scheduling

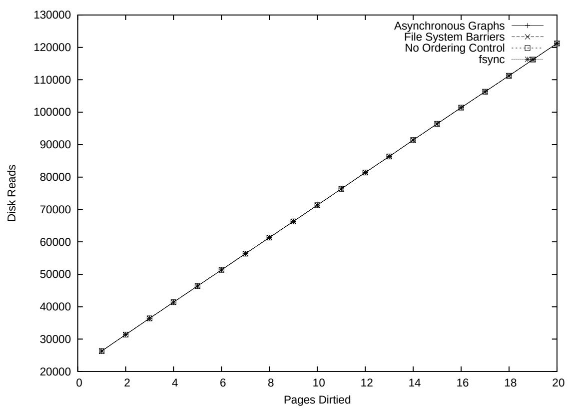
Figure 4.14 **Simulated execution time as the number of pages dirtied per transaction increases.** The line for asynchronous graphs is collinear with the line for no ordering control.

writes to both the data and the log pages to be combined into a fewer number of physical disk writes. Asynchronous graphs achieves this by carefully maintaining the ordering between logical writes and allowing the writes from each transaction to remain independent of each other. Asynchronous graphs does not incur a greater number of disk writes than using no ordering control, but the writes are executed in a different order than when no control is used.

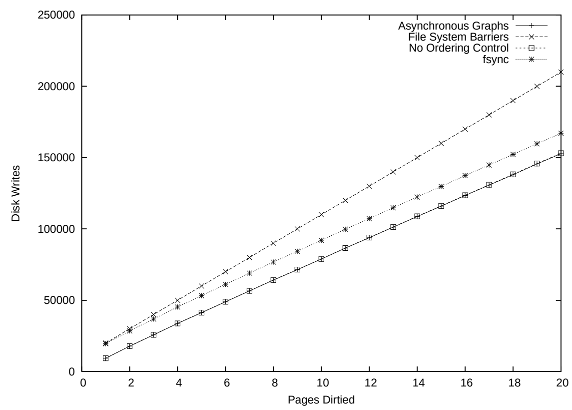
4.4.2.3 Pages Dirtied

For all of the ordering mechanisms other than file system barriers, increasing the number of pages dirtied has a similar effect to increasing the number of pages read. There is more cache pressure and an increase in the number of disk IOs. For file system barriers, the effect is to increase the number of write operations in each epoch.

Figure 4.14 shows the simulated execution time versus the number of pages dirtied in each transaction with both FIFO and C-LOOK disk scheduling. In both cases, the performance of file system barriers suffers more than the other mechanisms as the number of dirtied pages increases. This shows the effect of having to keep logical writes physically separate. Whereas the three non-barrier mechanisms allow multiple writes to the same data page to be combined into a single

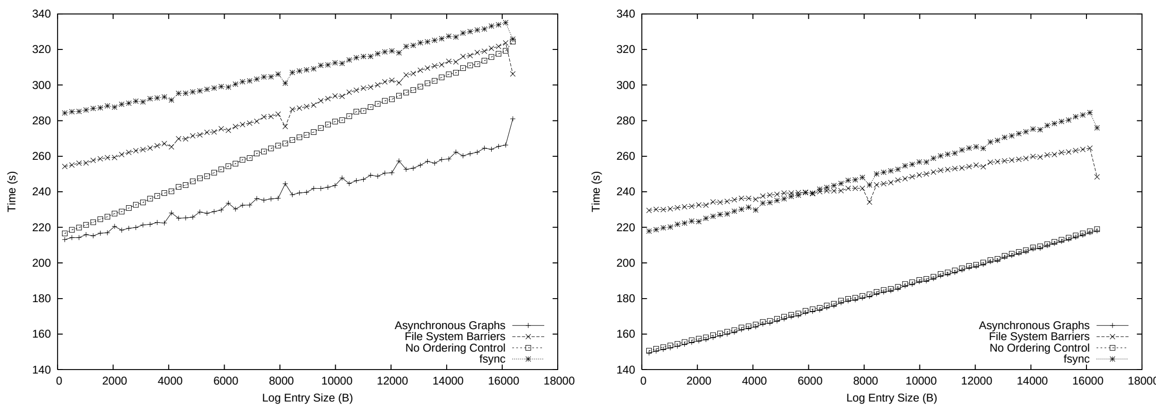


(a) Disk Reads



(b) Disk Writes

Figure 4.15 **The number of disk reads and writes as a function of the number of pages written in each transaction.** In (a) all four lines are collinear. In (b) the line for asynchronous graphs is collinear with the line for no ordering control.



(a) FIFO Disk Scheduling (b) C-LOOK Disk Scheduling

Figure 4.16 **Simulated execution time as the number of bytes written to the log per transaction increases.**

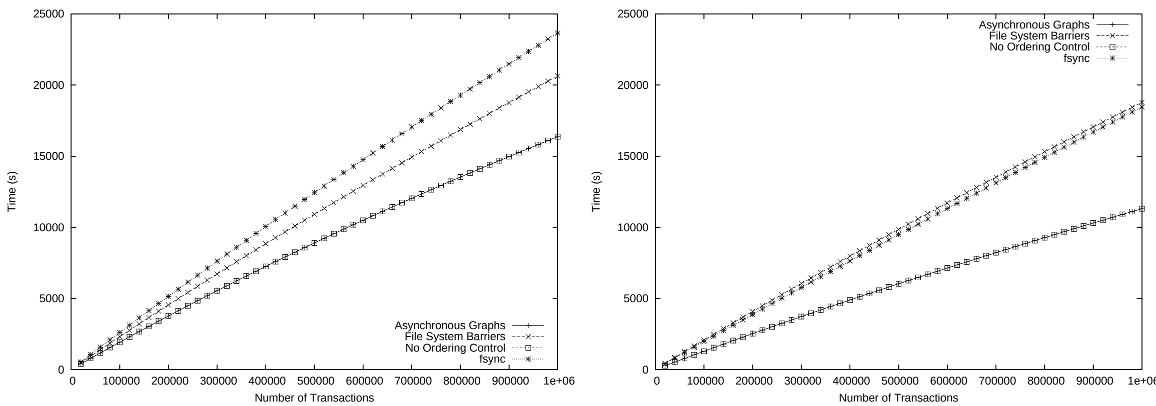
physical write, file system barriers doesn't allow this. As there are more and more pages being dirtied, the situation where a page is dirtied more than once happens more frequently.

Figure 4.15 shows the number of disk reads on the left and disk writes on the right for each ordering mechanisms. The number of reads increases as each page that is dirtied must be read off of the disk before it can be updated in memory. The number of writes for file system barriers increases linearly with the number of pages dirtied since it keeps nearly all of these writes physically separate. Disk writes for the other three mechanisms increase sub-linearly since a cache hit on a write results in two or more logical writes being combined into a single physical write.

Comparing the performance of FIFO and C-LOOK illustrates how file system barriers prevents the operating system from optimizing the performance of disk write traffic. While all four mechanisms improve with better disk scheduling, file system barriers improves less than the other three mechanisms because writing buffers an epoch at a time keep the disk queue very shallow.

4.4.2.4 Size of Log Entries

The most interesting effect of enlarging the amount of per-transaction log data is when file system barriers are in use, it enlarges the amount of data in each epoch that would otherwise be subject to `fsync()`. Instead of forcing the application to block while this data is written, file system



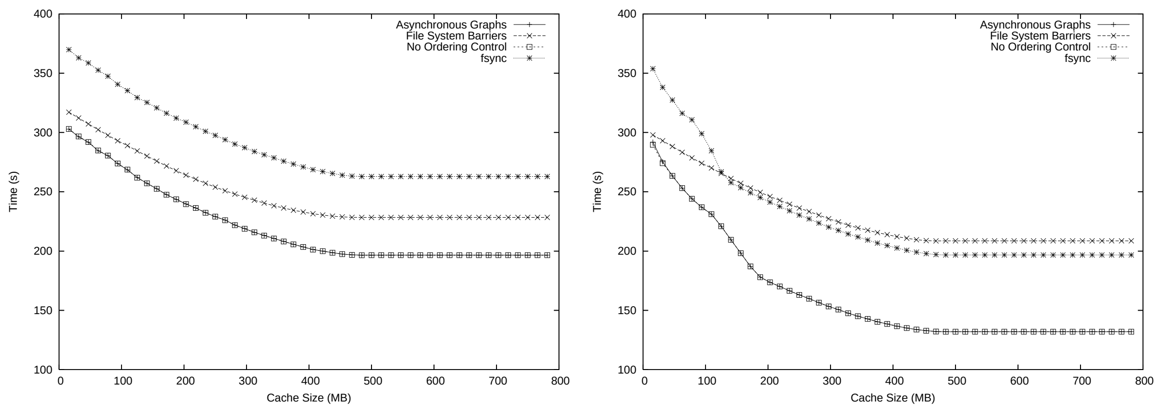
(a) FIFO Disk Scheduling (b) C-LOOK Disk Scheduling

Figure 4.17 **Simulated execution time as a function of the number of transactions executed.** The line for asynchronous graphs is collinear with the line for no ordering control.

barriers performs those writes in the background. Also as the size of the log entries increases, the fraction of buffer containing log data that need to be duplicated decreases. Only a buffer that contains log data for two or more transactions will ever require duplication. This accounts for the dips in runtime at 8 KB and 16 KB for file system barriers. As a result, we see the first result where barriers performs significantly better than using `fsync()` as Figure 4.16 shows. With C-LOOK scheduling the crossover point between barriers and `fsync()` is around 6 KB. This is a reasonable size for a log entry in some cases. PostgreSQL, for example, writes an entire data page to the write-ahead log the first time that page is dirtied [50]. This results in many log entries being at least as large as the default internal page size, which in PostgreSQL is 8 KB.

4.4.2.5 Number of Transactions

When using `fsync()` to control ordering, a DBMS will typically execute one `fsync()` call for each transaction, this flushes the log entries for that transaction to disk. As a result, the more transactions a workload executes, the greater the benefit of asynchronous graphs over `fsync()`. Figure 4.17 shows the execution time as a function of the number of transactions increases. As expected, the relative benefit increases with the number of transactions.



(a) FIFO Disk Scheduling

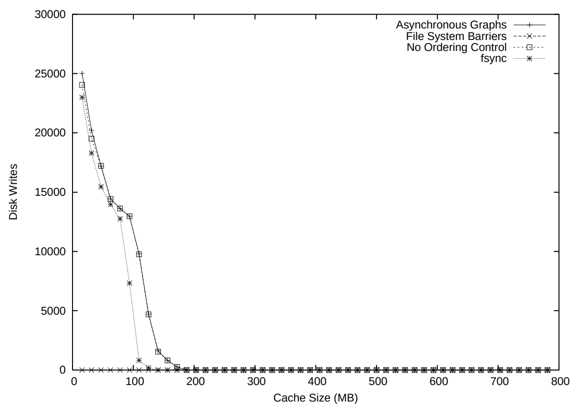
(b) C-LOOK Disk Scheduling

Figure 4.18 **Simulated execution time as a function of cache size.** The line for asynchronous graphs is collinear with the line for no ordering control.

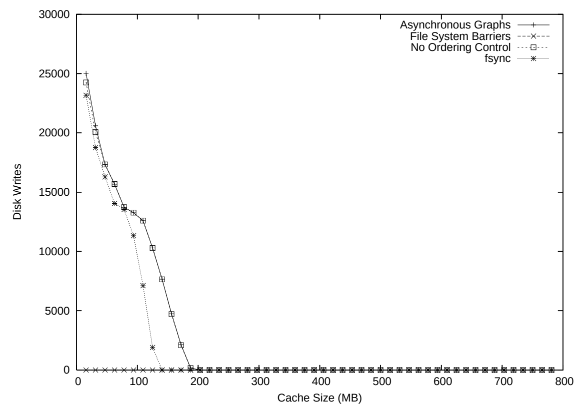
4.4.2.6 Cache Size

Figure 4.18 shows the execution time against cache size for both FIFO and C-LOOK scheduling as the size of the buffer cache is varied. Performance steadily improves until the entire working set fits in the cache at around 490 MB.

For very small cache sizes with C-LOOK scheduling, file system barriers performs better than using `fsync()`. When `fsync()` is being using, a number of pages are left dirty by each transaction, only writes to the log are forced to disk. The application blocks later while those dirty pages are written out so they can be replaced according to the replacement policy. Whereas file system barriers issues a write as soon as each log page is duplicated, this causes all of the pages in each preceding epoch to also be written in the background, keeping the total number of dirty pages in the cache to a minimum. Figure 4.19 shows the number of disk writes due to cache eviction. This effect is seen under C-LOOK scheduling because file system barriers sends whole epochs to the disk scheduler at once, whereas `fsync()` is sending only a single buffer at a time. This anomalous behavior for small cache sizes could be alleviated with better buffer flushing policies. However we believe caches this small are rare so we don't explore these possibilities any further.

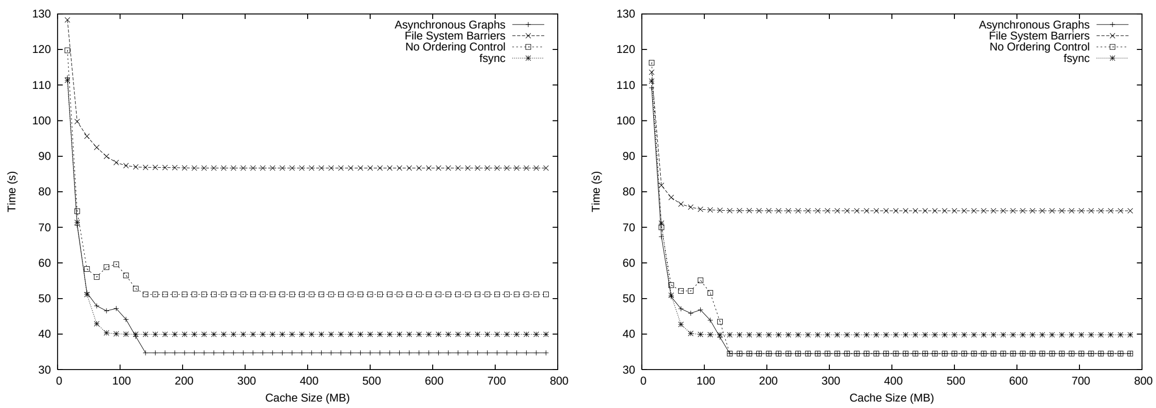


(a) FIFO Disk Scheduling



(b) C-LOOK Disk Scheduling

Figure 4.19 **Number of writes due to cache replacement as a function of cache size.** For most cache sizes the line for asynchronous graphs is collinear with the line for no ordering control.



(a) FIFO Disk Scheduling

(b) C-LOOK Disk Scheduling

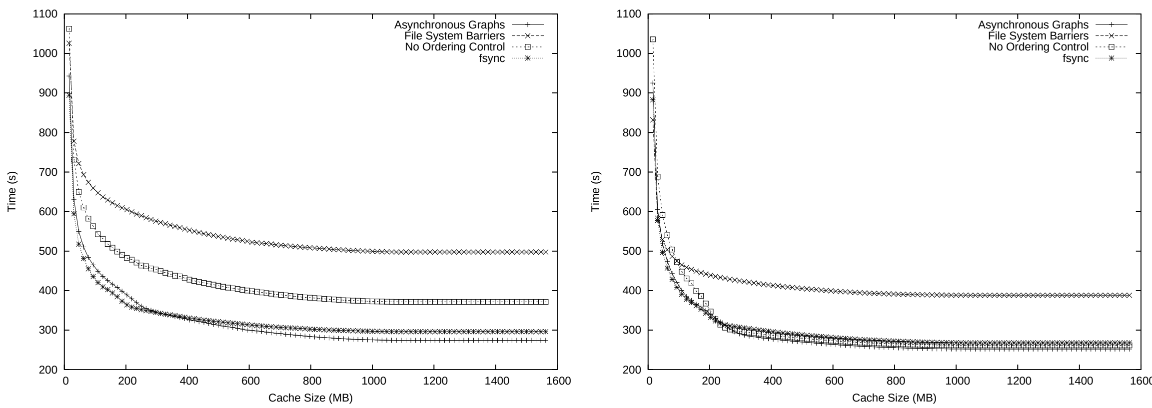
Figure 4.20 **Simulated execution time of TPC-B as a function of cache size.** In (b), for most cache sizes the line for asynchronous graphs is collinear with the line for no ordering control.

4.4.3 System Traces

4.4.3.1 TPC-B

In order to evaluate file system barriers and asynchronous graphs under more realistic workloads, this section presents further evaluation using an actual TPC-B system trace. This trace was collected by modifying PostgreSQL 8.1.3 to log the IO related system calls it issues, whenever it dirties a page in its buffer pool and the transaction number associated with each of these operations. Knowing when a buffer pool page is dirtied and the transaction numbers is necessary for generating the dependency information for asynchronous graphs. The TPC-B workload is generated by the `pgbench` tool, a TPC-B implementation included in the PostgreSQL distribution. In these experiments, pgbench is configured to execute 10000 transactions.

There are a number of differences between this workload and the synthetic workload used in the previous section. First, PostgreSQL is performing buffering in userspace. This means that by the time a data page is sent to the operating system, it may have been modified by many transactions. Thus, it may have a “must be written after” relationship with dozens or hundreds of log entries. This does not have a large impact on the performance since, even though all of the dependencies must be tracked for correctness, the number of actual dependencies remaining for a given data page when it is written is still small since the buffer flushing daemon tends to keep the number



(a) FIFO Disk Scheduling (b) C-LOOK Disk Scheduling

Figure 4.21 **Simulated execution time of TPC-C as a function of cache size.**

of dirty pages in the system low. The buffering effect reduces the write traffic when file system barriers are in use, since data pages that are dirtied as part of multiple transactions are duplicated less often. The log entries in this workload are larger than that of the synthetic workload, and they vary in size. The effect of this is to make file system barriers slightly more competitive than on a workload with small log entries for the reasons stated in Section 4.4.2.4. Lastly, the working set used by pgbench is much smaller than that used by the synthetic workload.

Figure 4.20 shows the simulated execution time of the pgbench workload as a function of the simulated cache size. We see results similar to the results for our synthetic benchmark. Performance levels off at a smaller cache size since the working set size is significantly smaller. For very small cache sizes, `fsync()` performs better since it keeps the number of dirty buffers containing log data low (*i.e.*, there is between one and zero at any time), and with extremely small cache sizes, these buffers make up a significant portion of the contents of the cache if they're not flushed. File system barriers perform poorly for the reasons discussed previously.

4.4.3.2 TPC-C

TPC-B is a relatively simple workload, with small, fixed sized transactions. In order to evaluate file system barriers and asynchronous graphs on a more complicated workload, we collected traces of a TPC-C [67] workload. This trace was collected in the same way as the TPC-B trace in the previous section. The workload was generated by BenchmarkSQL 2.3.2, a Java TPC-C implementation which runs on PostgreSQL and other DBMSs. BenchmarkSQL is configured to run with one terminal, 10,000 transactions and 10 warehouses. In TPC-C the number of warehouses is the parameter used to scale the size of the dataset. 10 warehouses yields a dataset of approximately 1 GB.

The results of our simulation are shown in Figure 4.21. As a fraction of the total execution time, asynchronous graphs provides a smaller performance improvement for this workload than for TPC-B. However, the improvement in *absolute* performance is similar. This is to be expected since, like our TPC-B workload, this workload consisted of 10,000 transactions. As we saw in 4.4.2.5, the benefit of asynchronous graphs over `fsync()` is a function of the number of transactions. When using `fsync()` for ordering, each transaction results in one call to `fsync()`. Thus both the TPC-B workload from the previous section and our TPC-C trace have the same number of `fsync()` calls, and so have the same degradation in performance due to those calls.

4.5 Conclusions

Buffer caching at the file system level provides a significant performance benefit to applications. However, in using a cache that is controlled by the operating system, the application is forced to cede to the OS control over the order in which its writes are committed to disk. While this is acceptable for a large number of applications, there is a significant class of applications for which it impedes having a system that is both fast and correct in the face of failures. Database management systems, for example, fall into this class, as would any application that performs incremental updates to complex on-disk data structures. When running on conventional operating systems, these applications are forced to either use direct access to the storage system to gain the control they need, which is inconvenient, or use OS facilities such as `fsync()`, which are slow.

This chapter proposes two new operating system interfaces that allow applications to express write-ordering constraints to the operating system. The first, file system barriers, appears at first to be beneficial, being a direct, asynchronous, replacement for `fsync()`. However, the semantics of the `barrier()` operation impede efficient execution of IO in many cases, mainly by forcing nearly all logical writes to be executed as separate writes to the disk resulting in increased disk traffic. In most cases, this makes file system barriers a worse choice than `fsync()`. This shows the importance of specifying ordering at a sufficiently fine granularity, and more generally, the importance of carefully defining OS interfaces. The second interface, asynchronous graphs, provides a fine-grained level of control to the application. This allows the operating system maximum freedom to optimize the IO path while still obeying write-ordering constraints laid down by the application. However, using asynchronous graphs requires more intrusive modifications to existing applications. Applications are required to carefully track ordering dependencies between write operations and pass those along to the operating system.

The simulation study presented shows for a limited set of workloads, file system barriers represent an improvement over using `fsync()` for ordering. The results show that asynchronous graphs, however, performs far better than either file system barriers or `fsync()`. In fact, in nearly all of the workload variations studied, asynchronous graphs is competitive with using no ordering control at all.

Chapter 5

Related Work

Storage caches have been in use at least since the early versions of UNIX [52]. As a result, the body of work concerning storage caching is substantial. This chapter first presents a brief overview of more recent work on buffer cache management, then presents work more closely related to the specific techniques described in this thesis.

5.1 General Caching Management

There is an enormous amount of literature on cache management. We summarize some of it here. In general, this body of work complements the work we have presented in this thesis. At the very least, any of these policies could be emulated on top of an existing operating system using InfoReplace or detected using (a possibly extended) *Dust*. In fact, we believe the large variety of policies that have been proposed for various purposes further motivates having application-selected replacement policies.

Cao, *et al.* proposed application controlled cache replacement policies where the kernel determined how much of the cache each process was permitted to occupy, but the application itself made individual replacement decisions [13, 14]. This is similar to InfoReplace in its goal of giving the application control over replacement decisions. The primary difference is that in Cao's work, the mechanisms within the kernel are modified to enable this OS/application collaboration. Specifically, when the kernel needs to free a page from the buffer cache, it chooses a process to take the page from, then the process decides which page to relinquish, and explicitly tells the kernel this decision. In InfoReplace the kernel's cache management mechanisms are unmodified, the only modification to the kernel is to expose portions of the already extant cache management data structures.

Forney, *et al.*, propose storage-aware caching, which bases the buffer cache replacement decisions performance of the underlying storage [21]. That is, taking into account the cost rereading a page from disk when making replacement decisions. It would be an interesting challenge to detect such a policy with the techniques that *Dust* uses. *Dust* assumes that it can influence all of the factors that the operating system is using to make replacement decisions. To detect a storage aware policy, *Dust* would need to be aware of the storage configuration, its performance characteristics and be able to control the layout of the test data on that storage. Only with that knowledge could *Dust* test whether replacement decisions were being made based on storage performance.

Various policies have been proposed in the context of database management systems. LRU-k, by O'Neil, *et al.*, remembers access information about pages that have already been evicted from the cache [45]. Specifically, replacement decisions are based on the k-th most recent access to each page. Thus, regular LRU is LRU-1. The authors found that most of the benefit of LRU-k was realized using LRU-2. That is basing replacement decision not on the most recent access to each page, but on the access previous to that. Johnson and Sasha developed a more efficient algorithm that has similar benefits to LRU-2, TwoQueue [29].

The Generalized Clock algorithm [44], proposed by Nicola, *et al.*, proposes a clock algorithm where instead of a use bit, each page has a weight associated with it, which is assigned when the page is read into memory. When the clock hand passes a given pages, instead of just resetting the use bit, the weight is decremented. The page becomes a replacement candidate when the weight value reaches zero. These weights are set using semantic knowledge about the database management system which is using the cache. For example, weights might be based on data type.

Lee, *et al.*and Smaragdakis, *et al.*, propose adaptive algorithms. Lee's algorithm, LRFU [34] takes into account both the recency of access to a block and the frequency of accesses, changing the weight given to recency and frequency according to the workload. LRFU thus subsumes LRU and LFU. Smaragdakis' algorithm, Early-Eviction LRU (EELRU) [60], uses LRU replacement in the common case, but when repeated sequential reads larger than the cache are detected, it adapts by evicting pages that are part of these runs early. In this way the worst-case scenario for LRU is avoided.

R.H. Patterson, *et al.*, suggest allowing applications to provide hints to the operating system to better manage prefetching and caching [49]. The goals of informed prefetching and caching are similar to the goals of InfoReplace. InfoReplace relies on the application using implicit techniques to alter the behavior of the operating system whereas informed prefetching and caching provides an explicit interface by which the application can pass hints to the OS.

Several techniques for cooperatively managing server and client caches have been proposed. Some of these are similar in spirit to our own work as they involve examining information flow between client and server. Wong and Wilkes propose extending distributed storage protocols to allow server and client caches to be made exclusive [75]. In contrast Zhou and Philbin, created a new replacement policy for second-level (*e.g.*server) caches that takes into account the expected behavior of the client caches [79]. In a similar vein, Chen, *et al.*, place data in a second-level cache only when it has been evicted from the first level cache, using implicit techniques to detect first-level evictions [15].

5.2 Implicit Information and Covert Channels

Fingerprinting system components to determine their behavior is not new and has been used successfully in other contexts, notably in networking and storage. Specifically, fingerprinting has been used to uncover key parameters within the TCP protocol and to identify the likely OS of a remote host [23, 47]. The primary difference between fingerprinting within TCP and in our context is that we are trying to identify policies that can have arbitrary behavior, rather than implementations that are expected to adhere to given specifications. In [56, 76] techniques similar to those used in *Dust* were used to determine various characteristics of disks, such as size of the prefetch window, prefetching algorithm and caching policy.

Fingerprinting also shares much in common with microbenchmarking. Specifically, both perform requests of the underlying system in order to characterize its behavior. For example, with simple probes in microbenchmarks, one can determine parameters of the the memory hierarchy [3, 55], processor cycle time [61], and characteristics of disk geometry [57, 65]. In our view, the key difference between fingerprinting and microbenchmarking is that a fingerprint is used to discover the policy or algorithm employed by the underlying layer, whereas a microbenchmark is typically used to uncover specific system parameters.

The idea of discovering characteristics of lower layers of a system and using that knowledge in higher layers to improve performance is not new. In traxtents [57] the file system layer of the operating system was modified to avoid crossing disk track boundaries so as to minimize the cost incurred due to head switching and exploit "zero-latency" access. Yu, *et al.* developed a method of predicting the position of the disk head without hardware support and used that information to determine which of several rotational replicas to use to service a given request [77], thus giving software expanded knowledge of hardware state.

Our approach involves informing the application of the buffer cache replacement policy in use by the operating system. SLEDs [71] and dynamic sets [62] seek to increase the knowledge that the application and operating system have of each other. Both take the approach of embellishing the interface between the OS and the application to allow the explicit exchange of certain types of information. In the case of dynamic sets, the application has the ability to provide more knowledge about its future access patterns. This allows the OS to reorder the fetching of data to improve cache performance. SLEDs allows the OS to export performance data to the application, enabling the application to modify its workload based on the performance characteristics of the underlying system.

The idea of servicing requests within a web server in a particular order was explored in connection-scheduling web servers [17]. The main thesis of that research is that better performance can be obtained by controlling the scheduling of requests within the web server, rather than with the OS. While their approach used static file size to schedule requests, cache-aware NeST uses a dynamic estimate of the contents of the buffer cache.

Our cache-aware web server has similarities to locality-aware request distribution (LARD) cluster-based web servers [48]. In LARD, the front-end node directs page requests to a specific back-end node based upon which back-end has most recently served this page (modulo load-balancing constraints); thus, the front-end has a simple model of the cache contents of each back-end and tries to improve their cache hit rates. Our approaches are complementary, as LARD partitions requests across different nodes, whereas we use cache content to service requests in a different order on a single node.

5.3 Explicit Information and Information Interfaces

The work on explicit information discussed in Chapter 3 was part of a larger project known as infoKernel [5] wherein the general issues in exposing operating system information to applications were explored. We now discuss some of the work related to this approach as it applies both to information exposure in general and to exposing cache information specifically.

An infoKernel, like other extensible systems, has the goal of tailoring an operating system to new workloads and services with user-specified policies. The primary difference is that an infoKernel strives to be *evolutionary* in its design. We believe that it is not realistic to discard the great body of code contained in current operating systems; an infoKernel instead transforms an existing operating system into a more suitable building block.

The infoKernel approach has the further difference from other extensible systems in that application-specific code is not run in the protected environment of the OS, which has both disadvantages and advantages. The disadvantages are that an infoKernel will probably not be as flexible in the range of policies that it can provide, there may be a higher overhead to indirectly controlling policies, and the new user-level policies must be used voluntarily by processes. However, there is an advantage to this approach as well: an infoKernel does not require advanced techniques for dealing with the safety of downloaded code, such as software-fault isolation [73], type-safe languages [10], or in-kernel transactions [58]. The open question that we address is whether the simple control provided by an infoKernel is sufficient to implement a range of useful new policies.

The idea of exposing more information has been explored for specific components of the OS. For instance, the benefits of knowing the cost of accessing different pages [71] and the state of network connections [54] has been demonstrated. An infoKernel further generalizes these concepts.

We now compare the infoKernel philosophy to three related philosophies in more detail: exokernel, Open Implementation, and gray-box systems. The goal of exposing OS information has been stated for exokernels [20, 30]. An exokernel takes the strong position that all fixed, high-level abstractions should be avoided and that all information (*e.g.*, page numbers, free lists, and cached TLB entries) should be exposed directly. An exokernel thus sacrifices the portability of applications across different exokernels for more information; however, standard interfaces can be supplied with library operating systems. Alternately, an infoKernel emphasizes the importance of allowing operating systems to evolve while maintaining application portability, and thus exposes internal state with abstractions to which many systems can map their data structures.

The philosophy behind the Open Implementation (OI) project [31, 32] is also similar to that of an infoKernel. The OI philosophy states, in part, that not all implementation details can be hidden behind an interface because not all are mere details; some details bias the performance of the resulting implementation. The OI authors propose several ways for changing the interface between clients and modules, such as allowing clients to specify anticipated usage, to outline their requirements, or to download code into the module. Clients may also choose a particular module implementation (*e.g.*, BTree, LinkedList, or HashTable); this approach exposes the algorithm employed, as in an infoKernel, but does not address the importance of exposing state.

Finally, there is a relationship between infoKernels and work on gray-box systems [4]. The philosophy of gray-box systems also acknowledges that information in the OS is useful to applications and that existing operating systems should be leveraged; however, the gray-box approach takes the more extreme position that the OS cannot be modified and thus applications must either assume or infer all information. There are a number of limitations when implementing user-level services with a gray-box system that are removed with an infoKernel. First, with a gray-box system, user-level services make key assumptions about the OS which may be incorrect or ignore important parameters. Second, the operations performed by the service to infer internal state may impose significant overhead (*e.g.*, a web server may need to simulate the file cache replacement algorithm on-line to infer the current contents of memory as with *Dust* and cache-aware NeST). Finally, it may not be possible to make the correct inference in all circumstances (*e.g.*, a service may not be able to observe all necessary inputs or outputs). Therefore, an infoKernel still retains most of the advantages of leveraging a commodity operating system, but user-level services built on an infoKernel are more robust to OS changes and more powerful than those on a gray-box system.

5.4 Explicit Control of File Systems and Caching

Explicit control over ordering of operations is not unprecedented. Processor architectures that allow multiple processors also need to allow the programmer to ensure that ordering constraints among loads and stores are honored. To do this, most modern architectures include a memory barrier instruction in some form [59] [74]. This instruction allows a programmer to place a fence in the instruction stream; no memory instruction can be reordered across such a fence. On the Alpha, the `MB` instruction is defined as: "Guarantee that all subsequent loads or stores will not access memory until after all previous loads and stores have accessed memory, as observed by other processors.". It is desirable that the operating system provide an interface to applications for specifying what orderings for writes are permitted. This dissertation explores two such interfaces, file system barriers and asynchronous graphs.

The Hewlett-Packard MPE XL operating system [33], provides dependency queues, which allow an application to issue any number of writes and be assured that ordering will be maintained. The dependency queues, however, impose a strict linear ordering on writes. This would be similar to an application which used file system barriers and issued a barrier after every write. Both of the ordering mechanisms described herein allow the operating system a greater opportunity to reorder writes for performance.

Chapter 6

Conclusions

6.1 Summary

This dissertation has presented three approaches for improving cooperation between operating system's caching subsystems and the applications that run on them. The key to this new level of cooperation is *information*. The more the operating system and applications know about one another, the more they are able to adapt to each others behavior to enhance the overall performance of the system.

6.1.1 Discovering Cache Information

Chapter 2 presents *Dust*, a tool for fingerprinting buffer cache replacement policies. *Dust* first executes a controlled synthetic workload on the file system buffer cache. This workload is specifically designed to distinguish cache replacement policies in that the initial access order, access recency, access frequency and use bit of each page is different and known. It then evicts half of the data from this workload and measures the amount of time required to read a random selection of the test data. By measuring the time to read a page, it is easy to determine whether that page was cached or not at the time of the read. Knowing which portions of the test data were still cached after half of the data has been evicted tells us the bases on which the cache policy makes replacement decisions. *Dust* runs entirely in user space and requires no kernel extensions.

In many cases, the information revealed by *Dust* is enough to produce a simulation of the operating system's caching policy. Such a model, together with knowledge of most of the accesses to the buffer cache, allows an application to predict the contents of the cache. We modified NeST, a software storage appliance, to use a cache simulation to schedule web requests on an in-cache-first basis. This resulted in a significant improvement in throughput. Further, we found that even imprecise cache predictions still enable a performance improvement.

6.1.2 Exposing Cache Information

InfoReplace depends on having accurate information about the state of the buffer cache. Unlike cache-aware NeST, if InfoReplace acts on inaccurate cache information, completely useless disk IO is the result (with cache-aware NeST, the data is at least still used to service a request). Because of this need for precise knowledge, implicit information is inadequate. To address this, we extended Linux to provide two key pieces of information explicitly. First, a new system call allows applications to obtain a list of the next N buffer cache pages that will be evicted, the victim list. Second, an eviction counter, residing in kernel memory but mapped into the user process, lets the application efficiently check how much the victim list has changed since the last time it was retrieved from the kernel. Since these interfaces simply expose information, that is, they do not provide any new mechanisms, they are very simple to implement. Most importantly, they only read, they never modify any already existing kernel data structures. We believe this minimality of intrusiveness is an advantage if these kinds of interfaces are to obtain widespread adoption.

In Chapter 3 we show that these additional interfaces enable the application to transform the kernel-provided buffer cache replacement policy into the policy of the application's choice. The application observes the victim list and when a page that the application wants to remain cached is in danger of being evicted, the application issues a read on that page. The only caveat being that the kernel must implement a policy that increases a page's priority when that page is read. Every modern operating system we have encountered has this property.

6.1.3 Controlling the Cache

When leveraging information, be it explicit or implicit, we relied entirely on implicit techniques to exercise control over the operating system. That is, we only used the existing interfaces to control the OS. This poses a limitation on the sorts of extensions that can be implemented.

For example, it is impossible to implement zero-copy networking on a kernel that doesn't otherwise support it using implicit techniques. No amount of coercion from user-space will reduce the number of data copies it takes to move data from a user-level buffer to the network adapter.

Chapter 4 presents two new interfaces which applications can use to express write ordering constraints to the operating system. Applications with complex on-disk data structures, such as those that employ write-ahead-logging, depend on having this control to ensure recoverability in the event of a crash. File system barriers introduces the `barrier()` system call. If an application issues some writes, calls `barrier()`, then issues additional writes, the writes will not be reordered across the barrier. Asynchronous graphs provides a more fine-grained interface whereby applications can specify a "must be written to disk after" relationship between individual write operations. We found that for two transactional workloads, asynchronous graphs has the best performance in more circumstances, though it is more difficult for an application programmer to use. In fact, in most cases, the performance of asynchronous graphs was comparable to the performance of using no ordering control at all. Surprisingly, file system barriers performed consistently poorly. This is due in large part to file system barriers not allowing two writes to the same page to be combined into a single physical write. As a result, using file system barriers results in a significantly higher number of disk writes than any of the other ordering methods examined.

6.2 Discussion

Since it is relatively easy to implement and deploy, we see cache-aware task scheduling as being a simple way to increase the performance of existing systems, such as web and storage servers. Though we didn't examine issues such as starvation, or combining predicted cache-state with other factors that might be a basis for a scheduling decision, gaining cache knowledge by algorithmic mirroring is an easy way to gain more performance-critical knowledge. We also point out that *Dust* only needs to be run once per version of an operating system. Once the replacement policy of a given version of a given OS is known, it could be provided as part of a cache-awareness library that applications could then use to determine cache state. This would make it easy to deploy such server applications. Most importantly, we have shown that useful internal information can be extracted from the operating system through the existing interface, and that that information doesn't have to be perfect to be useful.

Information exposing interfaces provide a safe, flexible way for application developers to extend operating systems to better suit application workloads. We have shown here that in the case of the buffer cache, exposing relatively simple information provides the application with a great deal of power. We believe the simplicity of the kernel modifications involved will enable them to gain adoption more readily than proposals to introduce new, complex mechanisms into the operating system. The kernel modifications involved are simple enough that even if they were not adopted into mainline OS releases, it would not be unreasonable to maintain such interfaces as separate patchsets.

Write ordering in commodity operating systems is a problem that we believe has not heretofore been adequately addressed. It is not reasonable to ask small sites to run applications on the raw disk partition. Not only do the open-source applications these sites use not support direct access to storage, but operating in this mode increases management overhead. Likewise, these sites shouldn't have to choose between performance and consistency. We've shown in this dissertation that is it possible for them to have both. We hope to see write-ordering interfaces in commodity operating systems in the future. While adding ordering interfaces represents a substantial modification to the internals of the operating system, it represents only a small, backward compatible change to the operating system interface. The ability to control ordering might also encourage application developers to design on-disk data structures that are recoverable after a crash.

6.3 Future Directions

It is an open question as to how bad the information can be and still be useful. For instance, what if the application is modeling the wrong policy? Reasoning about the difference between a model of a buffer cache and the cache itself would be best served by having a formal metric for the difference between two policies. For instance, LRU and Clock would have a very small difference, whereas LRU and MRU would have a large difference. A starting point point for such a metric might be the difference in final cache state after running a canonical workload, such as

Dust. However, a metric based on a canonical workload will be limited by the decision criteria (*e.g.* recency, frequency, initial order) that the workload tests for.

Explicit information interfaces are powerful but must be designed carefully. They must convey enough information to be useful but not so much that private process information is revealed. They must be efficient to be useful. A great deal more work will be required to make extensions like InfoReplace more common. To facilitate application portability, it will be necessary to standardize the information-revealing interfaces. Long before any standardization occurs, further research is necessary to determine what information is useful and how to expose it efficiently and safely. Further, rather than extend POSIX to provide information interfaces, we feel it would be better to design an operating system from the beginning to expose information. Such a system would have process-private information clearly separated from information that is safe to expose. Designing a system this way answers the question of what to expose: expose everything that is part of the "safe" data. Since the OS would have been designed from the beginning with information exposure in mind, will be easier to avoid opening security holes with information exposing interfaces. We believe that exposing information has some advantages over implementing new mechanisms in the OS kernel. It is impossible for OS developers to anticipate the needs of future applications. Exposing information provides a way to allow applications to safely extend the system on their own, relieving OS developers of some of the burden of being all things to all applications. The work presented here and as part of infoKernel is only the first step down this road.

We have shown that introducing write ordering control into the kernel is useful when fine grained control is required. It remains to be seen how large the class of workloads that can benefit from such interfaces is. Also in need of further study is the interface itself. File system barriers has a simple interface that is easy to use, but performs poorly. Asynchronous graphs performs well but requires more work on the part of the application programmer. Further study will be required to find out if there is a middle ground. Can there be an write-ordering interface that is fine-grained enough to perform well, but simple enough that application developers won't be hesitant to use it?

6.4 Broad Conclusions

The boundary between user-level applications and the operating system kernel traditionally allows only limited information to move across it. The operating system has knowledge of the global state of the system, and a number of abilities that applications do not possess. The OS does not have detailed knowledge of the needs of individual applications, has limited knowledge that it can use to predict their future needs and doesn't know when the application can change its workload (as it can in scheduling web connections) or when it cannot (as when performing IO to maintain transactional semantics). The result of this division of knowledge and division of responsibility is that in some instances, interesting information and the ability to act on that information are on opposite sides of the OS/application divide.

We have argued in this dissertation that it is useful to increase the amount of information moving across this interface, at least in the realm of cache management. The goal in moving information across the OS/application interface is to bring the information needed for making good decisions together with the mechanisms necessary to act on them. In Chapter 2, *Dust* brought approximate cache state knowledge into the application, where the ability to schedule web connections resides. In Chapter 3 we brought precise knowledge of the cache state into the application, the application then took that knowledge, along with knowledge of its own workload, to coerce the buffer cache policy into behaving in a more advantageous way. In this case, the application used knowledge of data semantics (*e.g.*the level of a page in an on-disk index), knowledge the OS does not possess, to improve cache management. Finally, file system barriers and asynchronous graphs in Chapter 4 moves knowledge about write ordering dependencies from the application into the operating system, where it can be acted on. The application knows what the safe orders of its writes are, but the operating system is actually responsible for performing the physical disk writes. We brought the knowledge needed to make smart decisions together with the the mechanisms necessary to act on those decisions.

Technologies such as virtual machine monitors, large-scale clusters and distributed storage add more and more layers to already complex systems. Each of these layers, like the operating systems and applications discussed herein, has certain knowledge of their part of the system, and certain decision making ability. The more layers a system has, the more likely it will become that the ability to act on a decision and the knowledge necessary to make a good decision will be in different parts of the system. Thus, we believe that information-based techniques such that those we described here will become more important in the future.

The decision on how to implement a new feature is based largely on how difficult it is to change the interface in question and how difficult it is to change the code on either side of that interface. When the interface between layers is entrenched, as POSIX is, our first attempt should be to use implicit techniques to move information across the system. In many cases, this will allow system designers to bring together the information needed for good decision making, and the mechanisms that can act on that information, without the upheaval required to change a popular, standardized interface. When an interface can be changed, but perhaps the code on the other side of that interface can't be radically altered, exposing explicit information gives an evolutionary path to extending the system. In some cases, information based techniques won't be powerful enough to implement the needed functionality. If this is the case and the interface and system behind it can be changed easily, or the new functionality is compelling enough to justify the pain, implementing an explicit mechanism may be the right answer.

LIST OF REFERENCES

[1] Apache Foundation. Apache web server. http://www.apache.org.

[2] Martin F. Arlitt and Carey L. Williamson. Internet web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, 1997.

[3] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve Steinberg, and Kathy Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[4] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[5] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[6] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and Dave Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.

[7] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of the SIGMETRICS '98 Conference*, June 1998.

[8] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[9] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *To appear in HPDC-11*, 2002.

[10] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[11] Jonathan L. Bertoni. Understanding solaris filesystems and paging. Technical Report TR-98-55, Sun Microsystems, 1998.

[12] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[13] Pei Cao, Edward W. Felten, and Kai Li. Application-Controlled File Caching Policies. In *Proceedings of the USENIX Summer Technical Conference (USENIX Summer '94)*, Boston, Massachusetts, June 1994.

[14] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 165–177, Monterey, California, November 1994.

[15] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Placement for Storage Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 269–282, San Antonio, Texas, June 2003.

[16] Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB 11)*, pages 127–41, Stockholm, Sweden, August 1985.

[17] Mark Crovella, Robert Frangioso, and Mor Harchol-Balter. Connection Scheduling in Web Servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[18] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[19] Peter Druschel, Vivek Pai, and Willy Zwaenepoel. Extensible Kernels are Leading OS Research Astray. In *Proceedings of the 6th Workshop on Workstation Operating Systems (WWOS-VI)*, pages 38–42, Cape Codd, Massachusetts, May 1997.

[20] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[21] Brian Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-Aware Caching: Revisiting Caching For Heterogeneous Storage Systems. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 61–74, Monterey, California, January 2002.

[22] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[23] Thomas Glaser. TCP/IP Stack Fingerprinting Principles. http://www.sans.org/newlook/resources/IDFAQ/ TCP_fingerprinting.htm, October 2000.

[24] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[25] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[26] John L. Hennessy and David A. Patterson, editors. *Computer Architecture: A Quantitative Approach, 3rd edition*. Morgan-Kaufmann, 2002.

[27] Yiming Hu, Qing Yang, and Tycho Nightingale. RAPID-Cache – A Reliable and Inexpensive Write Cache for Disk I/O Systems. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, Orlando, Florida, January 1999.

[28] *IBM DB2 Universal Database Administration Guide: Planning Version 8.2*. IBM Corp., 2004.

[29] Theodore Johnson and Dennis Shasha. 2Q: A Low-Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 439–450, Santiago, Chile, September 1994.

[30] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malo, France, October 1997.

[31] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail C. Murphy. Open Implementation Design Guidelines. In *International Conference on Software Engineering (ICSE '97)*, pages 481–490, Boston, Massachusetts, May 1997.

[32] Gregor Kiczales, John Lamping, Chris Maeda, David Keppel, and Dylan McNamee. The Need for Customizable Operating Systems. In *Proceedings of the 4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 165–169, Napa, California, October 1993.

[33] Alan J. Kondoff. The MPE XL Data Management System Exploiting the HP Precision Architectures for HP's Next Generation Commercial Computer Systems. In *IEEE Compcon Proceedings*, San Francisco, CA, 1988.

[34] Donghee Lee, Jongmoo Choi, Jun-Hum Kim, Sam H. Noh, Sang Lyul Min, Yookum Cho, and Chong Sang Kim. On The Existence Of A Spectrum Of Policies That Subsumes The Least Recently Used (LRU) And Least Frequently Used (LFU) Policies. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, Georgia, May 1999.

[35] H. Levy and P. Lipman. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, 15(3):35–41, March 1982.

[36] Linux Kernel Archives. Linux source code. http://www.kernel.org/.

[37] David Mazieres. A toolkit for user-level file systems. In *USENIX Technical Conference*, Boston, MA, June 2001.

[38] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[39] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[40] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions of Database Systems*, 17(1):94–162, 1992.

[41] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File systems – or – Your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 305–314, San Francisco, California, January 1992.

[42] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions of Computer Systems*, 6(1), February 1988.

[43] NetBSD Kernel Archives. NetBSD 1.5 Source Code. http://www.netbsd.org/.

[44] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *SIGMETRICS and PERFORMANCE*, 1992.

[45] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 297–306, Washington, DC, May 1993.

[46] Patrick O'Neil. Lru-2 source code. ftp://ftp.cs.umb.edu/pub/lru-k/lru-k.tar.Z.

[47] Jitendra Padhye and Sally Floyd. Identifying the TCP Behavior of Web Servers. In *SIGCOMM*, June 2001.

[48] Vivek Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers . In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, 1998.

[49] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, Colorado, December 1995.

[50] *PostreSQL 8.0 Documentation*. PostgreSQL Global Development Group, 2005.

[51] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[52] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[53] John T. Robinson and Murthy V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, 1990.

[54] Marcel-Catalin Rosu and Daniela Rosu. Kernel Support for Faster Web Proxies. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 225–238, San Antonio, Texas, June 2003.

[55] Rafael H. Saavedra and Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[56] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, 1999.

[57] Jiri Schindler, John L. Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, 2002.

[58] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, Washington, October 1996.

[59] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[60] Yannis Smaragdakis, Scott F. Kaplan, and Paul R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *Proceedings of the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, pages 122–133, Atlanta, Georgia, May 1999.

[61] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 155–166, New Orleans, Louisiana, June 1998.

[62] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proceedings of the the 15th ACM Symposium on Operating Systems Principles (SOSP)*, volume 27;5, pages 252–263, 1997.

[63] Michael Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[64] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[65] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[66] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[67] Transaction Processing Council. TPC Benchmark C Standard Specification, Revision 5.2. Technical Report, 1992.

[68] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[69] R. Turner and H. Levy. Segmented FIFO Page Replacement. In *1981 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1981.

[70] Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.

[71] Rodney Van Meter and Minxi Gao. Latency Management in Storage Systems. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 103–117, San Diego, California, October 2000.

[72] Rik van Riel. Page replacement in linux 2.4 memory management. http://www.surriel.com/lectures/linux24-vm.html, June 2001.

[73] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 203–216, Asheville, North Carolina, December 1993.

[74] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual*. PTR Prentice Hall, 1994.

[75] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, San Diego, California, June 2000.

[76] Bruce L. Worthington, Greg R. Ganger, Yale N. Patt, and John Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 146–156, Ottawa, Canada, May 1995.

[77] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[78] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamically Tracking Miss-Ratio-Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Boston, Massachusetts, October 2004.

[79] Yuanyuan Zhou, James F. Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 91–104, Boston, Massachusetts, June 2001.