The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms*

Ali R. Butt, Chris Gniady, and Y.Charlie Hu
Purdue Univeristy
West Lafayette, IN 47907
{butta, gniady, ychu}@purdue.edu

ABSTRACT

A fundamental challenge in improving the file system performance is to design effective block replacement algorithms to minimize buffer cache misses. Despite the well-known interactions between prefetching and caching, almost all buffer cache replacement algorithms have been proposed and studied comparatively without taking into account file system prefetching which exists in all modern operating systems. This paper shows that such kernel prefetching can have a significant impact on the relative performance in terms of the number of actual disk I/Os of many well-known replacement algorithms; it can not only narrow the performance gap but also change the relative performance benefits of different algorithms. These results demonstrate the importance for buffer caching research to take file system prefetching into consideration.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Measurements, Simulation*

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Buffer caching, prefetching, replacement algorithms

1. INTRODUCTION

A critical problem in improving file system performance is to design an effective block replacement algorithm for the buffer cache. Over the years, developing such algorithms has remained one of the most active research areas in operating systems design. The oldest and yet still widely used replacement algorithm is the Least Recently Used (LRU) replacement policy [9]. The effectiveness of LRU comes from the simple yet powerful principle of locality: recently accessed blocks are likely to be accessed again in the near future. Numerous other replacement algorithms have been developed [2, 11, 12, 13, 14, 18, 19, 21, 25, 27, 30, 33, 34]. However, the vast majority of these replacement algorithm studies used trace-driven simulations and used the cache hit ratio as the sole performance metric in comparing different algorithms.

Prefetching is another highly effective technique for improving the I/O performance. The main motivation for prefetching is to overlap computation with I/O and thus reduce the exposed latency of I/Os. One way to induce prefetching is via user-inserted hints of I/O access patterns which are then used by the file system to perform asynchronous I/Os [7, 8, 32]. Since prefetched disk blocks need to be stored in the buffer cache, prefetching can potentially compete for buffer cache entries. The close interactions between prefetching and caching that exploit user-inserted hints have also been studied [7, 8, 32]. However, such user-inserted hints place a burden on the programmer as the programmer has to accurately identify the access patterns of the application.

File systems in most modern operating systems implement prefetching transparently by detecting sequential patterns and issuing asynchronous I/Os. In addition, file systems perform synchronous read-ahead where requests are clustered to 64KB (typically) to amortize seek costs over larger reads. As in the user-inserted hints scenario, such kernel-driven prefetching also interacts with and potentially affects the performance of the buffer caching algorithm being used. However, despite the well-known potential interactions between prefetching and caching [7], almost all buffer cache replacement algorithms have been proposed and studied comparatively without taking into account the kernel-driven prefetching [2, 13, 18, 19, 25, 27, 30, 33, 34].

In this paper, we perform a detailed simulation study of the impact of kernel prefetching on the performance of a set of representative buffer cache replacement algorithms developed over the last decade. Using a cache simulator that faithfully implements the kernel prefetching of the Linux operating system, we compare different replacement algorithms in terms of the miss ratio, the actual number of aggregated synchronous and asynchronous disk I/O requests issued from the kernel to the disk driver, as well as the ultimate performance measure – the actual running time of applications using an accurate disk simulator, DiskSim [15]. Our study shows that the widely used kernel prefetching can indeed have a significant impact on the relative performance of different replacement algorithms. In particular, the findings and contributions of this paper are:

*This work is supported in part by the U.S. National Science Foundation under CAREER award ACI-0238379.

SIGMETRICS'05, June 6–10, 2005, Banff, Alberta, Canada.

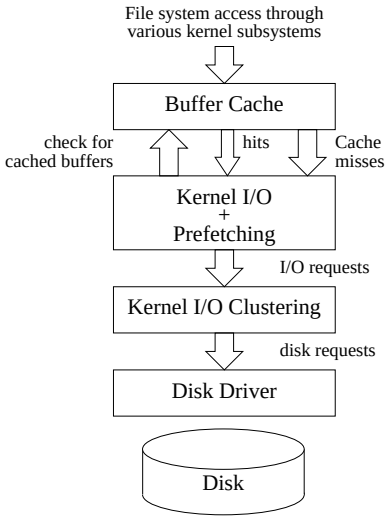

Figure 1: Various kernel components on the path from file system operations to the disk.

- We develop a buffer caching simulator that faithfully implements the Linux kernel prefetching to allow the performance study of different replacement algorithms in a realistic environment;
- We show how to adapt different cache replacement algorithms to exploit kernel prefetching to minimize disk I/Os while preserving the nature of these replacement algorithms;
- We find that kernel prefetching can not only significantly narrow the performance gap of different replacement algorithms, it can even change the relative performance benefits of different algorithms;
- We present results demonstrating that the hit ratio is far from a definitive metric in comparing different replacement algorithms, the number of aggregated disk I/Os gives much more accurate information of disk I/O load, but the actual application running time is the only definitive performance metric in the presence of asynchronous kernel prefetching.

The outline of the paper is as follows. Section 2 describes kernel prefetching in Linux and in 4.4BSD. Section 3 shows the potential impact of kernel prefetching on buffer caching algorithms using Belady's algorithm as an example. Section 4 summarizes the various buffer cache replacement algorithms that are evaluated in this paper. Section 5 presents trace-driven simulation results of performance evaluation and comparison of the studied replacement algorithms. Finally, Section 6 discusses additional related work, and Section 7 concludes the paper.

2. PREFETCHING IN FILE SYSTEM

In this section, we describe the kernel prefetching mechanisms in Linux and 4.4BSD.

2.1 Kernel Prefetching in Linux

The file system accesses from a program are processed by multiple kernel subsystems before any I/O request is actually issued to the disk. Figure 1 shows the various steps that a file system access has to go through before issued as a disk request. The first critical component is the buffer cache, which can significantly reduce the number of on-demand I/O requests that are issued to the components below. For sequentially accessed files, the kernel also attempts to prefetch consecutive blocks from the disk to amortize the cost of on-demand I/Os. Moreover, the kernel has a clustering facility that attempts to increase the size of a disk I/O to the size of a *cluster* – a *cluster* is a set of file system blocks that are stored on the disk contiguously. As the cost of reading a block or the whole *cluster* is comparable, the advantage of clustering is that it provides prefetching at minimal cost.

Prefetching in the Linux kernel is beneficial for *sequential accesses* to a file, i.e., accesses to consecutive blocks of that file. When a file is not accessed sequentially, prefetching can potentially result in extra I/Os by reading data that is not used. For this reason, it is critical for the kernel to make its best guesses of whether future accesses are sequential, and decide whether to perform prefetching.

The Linux kernel decides on prefetching by examining the pattern of accesses to the file, and only considers prefetching for read accesses. To simplify the description, we assume an access is to one block only. Although an access (system call) can be to multiple consecutive blocks, the simplification does not change the behavior of the prefetching algorithm. On the first access (A_1) to a file, the kernel has no information about the access pattern. In this case, the kernel resorts to *conservative prefetching* [1]; it reads the on-demand accessed block and prefetches a minimum number of blocks following the on-demand accessed block. The minimum number of blocks prefetched is at least one, and is typically three. This prefetching is called *synchronous prefetching*, as the prefetched blocks are read along with the on-demand accessed block. The blocks that are prefetched are also referred to as a *read-ahead group*. The kernel remembers the current *read-ahead group* per file and updates it on each access to the file. Note that as A_1 was the first access, no blocks were previously prefetched for this file, and thus the previous *read-ahead group* was empty.

The next access (A_2) may or may not be sequential with respect to A_1. If A_2 accesses a block that the kernel has not already prefetched, i.e., the block is not in A_1's *read-ahead group*, the kernel decides that prefetching was not useful and resorts to conservative prefetching as described above. However, if the block accessed by A_2 is in the previous *read-ahead group*, showing that prefetching was beneficial, the kernel decides that the file is being accessed sequentially and performs more aggressive prefetching. The size of the previous *read-ahead group*, i.e., the number of blocks that were previously prefetched, is doubled to determine the number of blocks (N) to be prefetched on this access. However, N is never increased beyond a pre-specified maximum (usually 32 blocks). The kernel then attempts to prefetch the N contiguous blocks that follow the blocks in the previous *read-ahead group* in the file. This prefetching is called *asynchronous prefetching* as the on-demand block is already prefetched, and the new prefetching requests are issued asynchronously. In any case, the kernel updates the current *read-ahead group* to contain the blocks prefetched on the current access.

So far, we used the previous *read-ahead group* to determine whether a file is being accessed sequentially or not. After the

[1]An exception is that if A_1 is to the first block in the file, the kernel assumes the following accesses to be sequential.

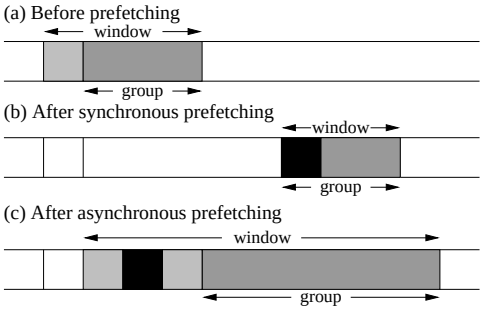

Figure 2: Prefetching in Linux. Shaded regions show blocks of the file that are cached and the black region shows the block being accessed. Also shown are the *read-ahead window* and *read-ahead group*.

prefetching associated with A_2 is also issued, the next access can benefit from prefetching if it either accesses a block in A_2's *read-ahead group* or in A_1's *read-ahead group*. For this reason, the kernel also defines a *read-ahead window* which contains the current *read-ahead group* as well as the previous *read-ahead group*. The *read-ahead window* is updated at the completion of an access and its associated prefetching, and is used for the access that immediately follows.

Any further (on-demand) access to the file falls into one of the following two cases: (i) the block is within the *read-ahead window* and has already been prefetched, justifying further asynchronous prefetching; or (ii) the block is outside the *read-ahead window* and synchronous prefetching is invoked.

Figure 2 (adapted from [4]) shows the adjustment of the *read-ahead group* and the *read-ahead window* under synchronous and asynchronous prefetching. In Figure 2(a), the light-gray area represents the previous *read-ahead group*, the dark-gray area represents the current *read-ahead group*, and the *read-ahead window* is also shown at the completion of an access. If the next access (shown as black) is to a block outside the *read-ahead window*, synchronous prefetching is employed, and the *read-ahead window* is reset to the new *read-ahead group* (Figure 2(b)). If the next access is to a block within the *read-ahead window*, asynchronous prefetching is employed, and the *read-ahead group* and *read-ahead window* are updated accordingly (Figure 2(c)).

Note that in both synchronous and asynchronous prefetching, it is possible that some blocks in the coverage of the *read-ahead group* already exist in the cache, in which case they are not prefetched, but are included in the current *read-ahead group*.

A subtlety with kernel prefetching is that prefetching of multiple blocks is atomic, but the creation of ready entries in the cache is done one by one. Linux first sequentially creates entries in the cache, marking all entries *locked*, implying I/O has not completed on them. It then issues prefetching which results in these blocks being read either at once for synchronous prefetching or sequentially for asynchronous prefetching. It marks the entries *unlocked* as the blocks arrive from the disk. In any case, a newly created entry will never be evicted before it is marked as unlocked.

Two special situations can occur in asynchronous prefetching. First, before the actual I/O operation on a prefetched block is completed, an on-demand access to the block may arrive. When this happens, the kernel does not perform any further prefetching. This is because the disk is already busy serving the prefetched block, and thus starting additional prefetching may over-load the disk [4]. Second, it is possible that an on-demand accessed block is found in the *read-ahead window* but not in the current *read-ahead group*, i.e., it is found in the previous *read-ahead group*. Further prefetching is not performed in this case either. For example, if the next access after the situation in Figure 2(c) is to the light-gray area, no further prefetching will be performed.

After the kernel has issued I/O requests for blocks that it intends to prefetch, I/O clustering is invoked to minimize the number of disk requests that are issued by clustering multiple I/O requests into a single disk request. When an I/O is requested on a block, it is passed to the clustering function, which puts the I/O request in the I/O request queue. If this I/O request is for a block following the block accessed by the previous I/O request (which is still in the queue), no new disk request is generated. Otherwise, a new disk request is created to fetch all the consecutive blocks by all the previous I/O requests accumulated in the queue. A new disk request can also be triggered directly (e.g., via timing mechanisms) to ensure that I/O requests are not left in the queue indefinitely.

2.2 Kernel Prefetching in 4.4BSD

The file system prefetching employed in 4.4BSD is similar to that used in Linux. An access is considered sequential if it is either to the last accessed block, or a block immediately following the last accessed block, i.e., access to a block B is sequential if last access was to block B or B-1. For sequential accesses, the file system prefetches one or more additional blocks. The number of blocks that are prefetched is doubled on each disk read. However, it does not exceed the maximum cluster size (usually 32) or the remaining blocks of a file. If the file only occupies a fraction of the last block allocated to it, this final fragmented block is not prefetched.

It may happen that a prefetched block is evicted before it is used even once. If the accesses remain sequential, the block will be subsequently accessed but will not be found in the cache (as it was prefetched and evicted), indicating that the prefetching process is too aggressive. This is addressed by halving the number of blocks to be prefetched on each access whenever such an event occurs. Finally, if the access is not sequential, no prefetching is used. This is in contrast to Linux, where one extra block is always read from the disk even if the access falls outside the *read-ahead window*.

3. MOTIVATION

The goal of buffer replacement algorithm is to minimize the number of disk I/O operations and ultimately reduce the running time of the applications. To show the potential performance impact of kernel prefetching on buffer caching replacement algorithms, we give an example that shows given kernel prefetching, Belady's algorithm [3] can be non-optimal in reducing disk requests.

For simplicity, we assume a prefetching algorithm simpler than that used in Linux. We use a cache size of 8 blocks, and assume that prefetching is only done on a miss so that the I/Os for the prefetched blocks can be clustered with the I/O for the on-demand block. On the first access, the minimum number of prefetched blocks is set to 3. If the following access is to a block that has been prefetched on the previous access, the number of blocks to be prefetched on the next miss is increased to 8, otherwise it remains as

Acc. Num.	Blk	Belady's algorithm		LRU	
		cache content	I/O	cache content	I/O
1	a	[**a b c** - - - - -]	y	[**a b c** - - - - -]	y
2	c	[a b c - - - - -]	n	[a b c - - - - -]	n
3	e	[a b c **e f g h i**]	y	[**e f g h i j k l**]	y
4	g	[a b c e f g h i]	n	[e f g h i j k l]	n
5	i	[a b c e f g h i]	n	[e f g h i j k l]	n
6	k	[a b c e f g **k l**]	y	[e f g h i j k l]	n
7	m	[a b c e f g **m n**]	y	[g i k l **m n o p**]	y
8	o	[a b c e f g **o p**]	y	[g i k l m n o p]	n
9	a	[a b c e f g o p]	n	[k m n o p **a b c**]	y
10	b	[a b c e f g o p]	n	[k m n o p a b c]	n
11	c	[a b c e f g o p]	n	[k m n o p a b c]	n
12	d	[b c **d** e f g o p]	y	[**d e f g h i j k**]	y
13	e	[b c d e f g o p]	n	[d e f g h i j k]	n
14	f	[b c d e f g o p]	n	[d e f g h i j k]	n
15	g	[b c d e f g o p]	n	[d e f g h i j k]	n
16	h	[**h i j k l m n** o]	y	[d e f g h i j k]	n
17	i	[h i j k l m n o]	n	[d e f g h i j k]	n
18	j	[h i j k l m n o]	n	[d e f g h i j k]	n
19	k	[h i j k l m n o]	n	[d e f g h i j k]	n
20	l	[h i j k l m n o]	n	[i j k **l m n o p**]	y
21	m	[h i j k l m n o]	n	[i j k l m n o p]	n
22	n	[h i j k l m n o]	n	[i j k l m n o p]	n
23	o	[h i j k l m n o]	n	[i j k l m n o p]	n
24	p	[i j k l m n o **p**]	y	[i j k l m n o p]	n
I/Os			8		6

Table 1: An example scenario where LRU results in fewer disk I/Os compared to Belady's replacement algorithm. The cache content after each access are shown. The blocks read on a cache miss are shown in bold.

3. Furthermore, if a block to be prefetched is already in the cache, only the blocks between the on-demand accessed block and the cached block are prefetched. The reason for this is that prefetching beyond the cached block will prevent some prefetched requests from being clustered with the on-demand access. We also assume that if a block would cause the eviction of the on-demand accessed block or any other associated prefetched block, we do not prefetch it or the blocks after it, as they would not be clustered with the on-demand access.

Table 1 shows the behavior of Belady's replacement algorithm and LRU for a sequence of file system requests under the above simple prefetching. In a system without prefetching as used in almost all previous studies of cache replacement algorithms, the Belady algorithm will result in 16 cache misses, which translate into 16 I/O requests, whereas LRU will result in 23 cache misses, or 23 I/O requests. However, with prefetching, the number of I/O requests is reduced to 8 using Belady's algorithm and 6 using LRU. The reason for this is that Belady's algorithm has knowledge of the blocks that will be accessed in the nearest future and keeps them in the cache without any regard to how retaining these blocks will affect prefetching. Since Belady's algorithm results in more I/O requests than LRU, it is not optimal in minimizing the number of I/O operations.

4. REPLACEMENT ALGORITHMS

In this section, we discuss the eight representative recency/frequency-based buffer cache replacement algorithms used in our evaluation of the impact of kernel prefetching. For each replacement algorithm, we summarize the original algorithm followed by the adapted version that manages the blocks brought in by kernel prefetching. We emphasize that all algorithms assume an unmodified kernel prefetching underneath. The reason is simply to compare the different algorithms in a realistic scenario, i.e., when implemented in the Linux buffer cache.

All practical replacement algorithms use the notion of *recency* in deciding the victim block for eviction, and how to assign the recency to prefetch blocks is an important issue. Since placing prefetched blocks at the most recently used (MRU) location is consistent with the actual Linux implementation, we use this design in LRU and all other algorithms that utilize recency information, including LRU-2, 2Q, LIRS, LRFU, MQ, and ARC.

With the exception of LRU, the above algorithms also use the notion of *frequency* in deciding the victim block for eviction, and thus it is important to assign appropriate frequency information to prefetched blocks to preserve the original behavior of each replacement algorithm. Fortunately, for most algorithms, this can be achieved by recording each prefetched block as "not accessed yet". When the block is accessed, its frequency is adjusted accordingly. Some of these algorithms also use a ghost cache to record the history of a larger set of blocks than that can be accommodated in the actual cache. For these schemes, if a prefetched block is evicted from the cache before it is ever accessed, it is simply discarded, i.e., not moved into the ghost cache.

OPT

The optimal or OPT scheme is based on Belady's cache replacement algorithm [3]. This is an off-line algorithm as it requires an oracle to determine future references to a block. OPT evicts the block that will be referenced farthest in the future. As a result it maximizes the hit rate.

In the presence of the Linux kernel prefetching, prefetched blocks are assumed to be accessed most recently, one after another, and inserted into the cache according to the original OPT algorithm. Note the kernel prefetching is oblivious of future references. But OPT can immediately determine wrong prefetches, i.e., prefetched blocks that will not be accessed on-demand at all. Such blocks become immediate candidates for removal, and can be replaced right after they are fetched. Similarly, prefetched blocks can also be right away evicted if they are accessed further in future than all other blocks in the cache. However, blocks prefetched together do not evict each other. This is because such blocks are most likely prefetched in a single disk request, and all the blocks should stay resident in the cache till the I/O operation associated with them completes.

LRU

LRU is the most widely used replacement policy. It is usually implemented as an approximation [9] without significant impact on the performance. LRU is simple and does not require tuning of parameters to adapt to changing workload. However, LRU can suffer from its pathological case when the working set size is larger than the cache and the application has a looping access pattern. In this case, LRU will replace all blocks before they are used again, resulting in every reference incurring a miss.

Kernel prefetching is incorporated into LRU in a straightforward manner. On each access, the kernel determines the number of blocks that need to be prefetched based on the algorithm explained in Section 2.1. The prefetched blocks are inserted in the MRU location just like regular blocks.

LRU-2

The LRU-K [30, 31] scheme tries to avoid the pathological cases of LRU. LRU-K replaces a block based on the Kth-to-the-last reference. The oldest resident based on this metric is evicted. For simplicity, the authors recommended K=2. By taking the time of the penultimate reference to a block as the basis for comparisons, LRU-2 can quickly remove cold blocks from the cache. However, for blocks without significant differences of reference frequencies, LRU-2 performs similarly as LRU. In addition, LRU-2 is costly; each block access requires $log(N)$ operations to manipulate a priority queue, where N is the number of blocks in the cache.

In the presence of kernel prefetching, LRU-2 is adapted as follows. First, when a block is prefetched, it is marked as without any access history, so that when it is accessed on-demand for the first time, its prefetching time will not be mistaken as its penultimate reference time. Second, to implement the *Correlated Reference Period* (CRP), after a block is accessed and before it becomes eligible for replacement, it is put in a list for recording ineligible blocks. Only eligible blocks are added to the replacement priority queue. With prefetching, all prefetched blocks are initially ineligible for replacement as they are considered to be last accessed (together) less than the CRP.

2Q

2Q [19] was proposed to perform as well as LRU-2 yet with a constant overhead. It uses a special buffer, called the *A1in queue*, in which all missed blocks are initially placed. When the blocks are replaced from the *A1in queue* in the FIFO order, the addresses of these replaced blocks are temporarily placed in a ghost buffer called *A1out queue*. When a block is re-referenced and its address is in the *A1out queue*, it is promoted to a main buffer called *Am*, which stores frequently accessed blocks. Thus this approach filters temporarily high frequency accesses. By setting the relative sizes of *A1in* and *Am*, 2Q picks a victim block from either *A1in* or *Am*, whichever grows beyond the preset boundary.

In the presence of kernel prefetching, 2Q is adapted similarly as in previous schemes, i.e., prefetched blocks are treated as on-demand blocks. When a block is prefetched before any on-demand access, it is placed into the *A1in queue*. On the subsequent on-demand access, the block stays in the *A1in queue*, as if it is being accessed for the first time. If the block is evicted from the *A1in queue* before any on-demand access, it is simply discarded, as opposed to being moved into the *A1out queue*. This is to ensure that that on its actual on-demand access, the block will not be incorrectly promoted to *Am*. If a block currently in the *A1out queue* is prefetched, it is promoted into *Am* as if it is accessed on-demand.

LIRS

Low Inter-reference Recency Set (LIRS) [18] (and its variant Clockpro [17]) is another recently proposed algorithm that maintains a complexity similar to that of LRU by using the distance between the last and second-to-the-last references to estimate the likelihood of the block being referenced again. LIRS maintains a variable-size LRU stack of blocks that have been seen recently. LIRS classifies each block into an LIR block if it has been accessed again since it was inserted on the LRU stack, or an HIR block if the block was not on the LRU stack. HIR blocks are referenced less frequently. The stack variability is caused by removal of blocks below the least recently seen LIR block on the stack. Similar to CRP of LRU-2 or Kin of 2Q, LIRS allocates a small portion of the cache, 1% as suggested by authors, to store recently seen HIR blocks.

In the presence of kernel prefetching, LIRS is modified not to insert any prefetched blocks into the LRU stack to prevent distortion of the history stored in the LRU stack. Instead, a prefetched block is inserted into the portion of the cache that maintains HIR blocks, since an HIR block does not have to appear in the LRU stack. If a prefetched block did not have an existing entry on the stack, the first on-demand access to the block will cause it to be inserted onto the stack as an HIR block. If an entry for the block was present in the stack, the first on-demand access that follows will result in the block being treated as an LIR block. In both cases, the outcome is consistent with the behavior of the original LIRS.

LRFU

Least Recently/Frequently Used (LRFU) [25] is a recently proposed algorithm that provides a continuous range of policies between LRU and LFU. A weight $C(x)$ is associated with every block x, and at every time t, $C(x)$ is updated as

$$C(x) = \begin{cases} 1 + 2^{-\lambda}C(x) & \text{if } x \text{ is referenced at time } t \\ 2^{-\lambda}C(x) & \text{otherwise} \end{cases}$$

where λ is a tunable parameter. The LRFU algorithm replaces the block with the smallest $C(x)$ value. The performance of LRFU critically depends on the choice of λ.

Prefetching is employed in LRFU similarly as in LRU; prefetched blocks are treated as the most recently accessed. One problem arises as how to assign the initial weight for a prefetched block, as the single weight combines both recency and frequency information. Our solution is to set a prefetched flag to indicate that a block is prefetched and not yet accessed on-demand. When the block is accessed on-demand and the prefetched flag is set, we reset the value of $C(x)$ to the default initial value instead of applying the above function. This ensures that the algorithm counts an on-demand accessed block as once-seen and not twice-seen.

MQ

The Multi-Queue buffer management scheme (MQ) [41] was recently proposed as a second-level replacement scheme for storage controllers. The idea is to use m LRU queues (typically $m = 8$), Q_0, Q_1, ..., Q_{m-1}, where Q_i contains blocks that have been seen at least 2^i times but no more than $2^{i+1} - 1$ times recently. The algorithm also maintains a history buffer Q_{out}. Within a given queue, blocks are ranked by the recency of access, i.e., according to LRU. On a cache hit, the block frequency is incremented, and the block is placed at the MRU position of the appropriate queue, and its $expireTime$ is set to $currentTime + lifeTime$. The $lifeTime$ is a tunable parameter indicating the amount of time a block can reside in a particular queue without an access. On each access, $expireTime$ for the LRU block in each queue is checked, and if it is less than $currentTime$, the block is demoted to the MRU position of the next queue.

In the presence of prefetching, MQ is adapted similarly as in LRFU; the only issue is how to correctly count block access frequency in the presence of prefetching. This is solved by not incrementing the reference counter when a block is prefetched. MQ also maintains a ghost cache equal to the size of the cache for remembering information about blocks

that were evicted. We modified the behavior of the ghost cache to not recording any prefetched blocks that have not been accessed upon eviction from the cache.

ARC

The most recent addition to the recency/frequency-based policies is Adaptive Replacement Cache (ARC) [27] (and its variant CAR [2]). The basic idea of ARC/CAR is to partition the cache into two queues, each managed using either LRU (ARC) or CLOCK (CAR): the first contains pages accessed only once, while the second contains pages accessed more than once. A hit to the first queue moves the accessed block to the second queue. Moreover, a hit to a block whose history information is retained in the ghost cache also causes the block to move to the second queue. Like LRU, ARC/CAR has a constant complexity per request.

Kernel prefetching is exploited in ARC similarly as in 2Q. A prefetched block is put into the first queue with a special flag, so that upon the subsequent on-demand access, it will stay in the first queue. An important difference from 2Q is that if a prefetched block is already in the ghost cache, it is not moved to the second queue, but to the first queue. If the block is prefetched correctly, it will be moved to the second queue upon the subsequent on-demand access. In this way, if prefetching brings in blocks that are not accessed again, they do not pollute the second queue. Finally, ARC also implements a ghost cache. As in MQ and 2Q, prefetched blocks that have not been accessed upon eviction will not be put into the ghost cache.

5. PERFORMANCE EVALUATION

In this section we evaluate the impact of Linux kernel prefetching on the performance of replacement algorithms.

5.1 Traces

The detailed traces of the applications were obtained by modifying the *strace* Linux utility. *Strace* intercepts the system calls of the traced process and is modified to record the following information about the I/O operations: access type, time, file identifier (`inode`), and I/O size.

Tables 2 and 3 show the six applications and three concurrent executions of the mixed applications used in this study. For each application, Table 2 lists the number of I/O references, the size of the I/O reference stream, the number of unique files accessed, and the fraction of references to consecutive file blocks. The selected applications and workload sizes are comparable to the workloads in recent studies [12, 18, 25, 27] and require cache sizes of up to 1024MB. We classify applications in three groups, *sequential applications* that read entire files mostly sequentially, *random access applications* that perform small accesses to different parts of the file, and a third group which represents applications containing a mix of sequential and random accesses.

5.1.1 Sequential access applications

Cscope, glimpse, gcc and *viewperf* read entire files sequentially and thus prefetching will benefit these applications.

Cscope [35] performs source code examination. The examined source code is Linux kernel 2.4.20. *Glimpse* [26] is an indexing and query system and is used to search for text strings in 550MB of text files under the `/usr` directory. In both *cscope* and *glimpse*, an index is built first, and single word queries are then issued. Only I/O operations during the query phases are used in the experiments. Table 2 shows that *cscope* and *glimpse* are good candidates for sequential prefetching since 76% and 74% of references, respectively, occur to the consecutive blocks. The remaining fraction of references are not consecutive because they reference the beginning of new files.

Appl.	Num. of references	Data size [MB]	Num. of files	Seq. refs.
cscope	1119161	260	10635	76%
glimpse	3102248	669	43649	74%
gcc	158667	41	2098	27%
viewperf	303123	495	289	99%
tpc-h	13468995	1187	49	3%
tpc-r	9415527	1087	49	3%
multi1	1278135	297	12246	70%
multi2	1580908	792	12514	75%
multi3	16571229	1855	43696	16%

Table 2: Applications and trace statistics

Appl.	Applications executed concurrently
multi1	*cscope, gcc*
multi2	*cscope, gcc, viewperf*
multi3	*glimpse, tpc-h*

Table 3: Concurrent applications

Gcc builds Linux kernel 2.4.20 and is one of the commonly used benchmarks. It has a very small working set; 4MB of buffer cache is enough to contain the header files. *Gcc* reads entire files sequentially. However, the benefit of prefetching will be limited since only 27% of references occur to consecutive blocks. Most of 73% of references are to the beginning of new files, since in *gcc*, 50% of references are to files that are at most two blocks long.

Viewperf is a *SPEC* benchmark that measures the performance of a graphics workstation. The benchmark executes multiple tests to stress different capabilities of the system. The patterns are mostly regular loops as *viewperf* reads entire files to render images. Over 99% of references are to consecutive blocks within a few large files, resulting in a perfect opportunity for prefetching.

5.1.2 Random access applications

The *MySQL* [29] database system is used to run *TPC-H* (*tpc-h*) and *TPC-R* (*tpc-r*) benchmarks [39]. *Tpc-h* and *tpc-r* access a few large data files, some of which have multiple concurrent access patterns. *Tpc-h* and *tpc-r* perform random accesses to the database files. As a result, only 3% of references occur to consecutive blocks. We can predict that neither of these benchmarks will benefit from prefetching. Prefetching will not only increase the disk bandwidth demand, but also pollute the cache.

5.1.3 Concurrent applications

Multi1 consists of concurrent executions of *cscope* and *gcc*. It represents the workload in a code development environment. *Multi2* consists of concurrent executions of *cscope*, *gcc*, and *viewperf*. It represents the workload in a workstation environment used to develop graphical applications and simulations. *Multi3* consists of concurrent executions of *glimpse* and *tpc-h*. It represents the workload in a server environment running a database server and a web index server.

Scheme	Parameters
OPT	*NA*
LRU	*NA*
LRU-2	*CRP = 20, history size = cache size*
LRFU	$\lambda = 0.001$
LIRS	*HIR = 10% of cache, LRU stack = 2 * cache size*
MQ	*4 queues, ghost cache = cache size*
2Q	*A1in = 25% of cache, ghost cache = cache size*
ARC	*ghost cache = cache size*

Table 4: Parameters for various replacement algorithms

5.2 Simulation environment

We implemented a buffer cache simulator that faithfully implements the kernel prefetching and I/O clustering of Linux 2.4 kernel as described in Section 2.1. The I/O clustering mechanism attempts to cluster I/Os to consecutive blocks into disk requests of up to 64KB. In addition, our simulator implements the eight cache replacement algorithm discussed in Section 4, OPT, LRU, LRU-2, LRFU, LIRS, MQ, 2Q, and ARC, as well as their corresponding adapted versions that assume kernel prefetching and manage prefetched blocks. Tables 4 lists the default parameters for each algorithm. To evaluate each algorithm with comparable parameters, we provide the additional history (ghost cache) equal to the cache size for each algorithm that uses history information.

For each simulation, we measure the respective hit ratio, the number of resulting (clustered) disk requests, and the execution time. With prefetching, access to a prefetched block is counted as a hit if the prefetching is completed, and as a miss otherwise. Since obtaining an accurate hit ratio in the presence of prefetching is not possible without a time-aware simulator, we interfaced our buffer cache simulator with DiskSim 3.0, an accurate disk simulator [15] to simulate the Seagate ST39102LW disk model. The combined simulator allows us to simulate the I/O time of an application and measure the reduction of the execution time under each replacement algorithm.

In comparing the execution times of each application under different replacement algorithms, we do not consider the execution overhead of implementing the different replacement algorithms. We believe that the effects of prefetching on the execution time that we will observe in the following overshadows the execution overhead of any reasonably efficient implementations of the studied algorithms.

5.3 Results

In this section, we examine how prefetching affects each group of the applications.

5.3.1 Sequential access applications

The accesses in the applications in this group exhibit looping reference patterns where blocks are referenced repeatedly with regular intervals. As expected for applications in this group, prefetching improves the hit ratios. However, the improvement from prefetching is more pronounced in some applications than others.

Cscope

Figure 3 shows the results for *cscope*. The following observations can be made. First, kernel prefetching has a significant impact on the hit ratio, but the improvement for different algorithms differ. At 128MB cache size, while the hit ratios for LRU, LRFU and ARC increase by 39%, 39%, and 38%, respectively, the hit ratios for LRU-2, LIRS, and 2Q increase from 7.0%, 18.8%, and 58.4% to 78.1%, 77.1%, and 75.8%, respectively. This suggests that it is critical to consider kernel prefetching when comparing different replacement schemes.

Second, the clustering of I/O requests in the presence of prefetching also results in a significant reduction in the number of disk requests compared to without prefetching. For instance at 64MB cache size, all schemes besides OPT perform about 604,000 disk requests without prefetching, but only about 422,000 requests with prefetching. This is because without prefetching, *cscope* offers little opportunity for clustering as it reads files in small chunks (mostly 2 blocks at a time). When prefetching is enabled, the sequential nature of the accesses allows a larger number of contiguous blocks to be read asynchronously. These blocks are clustered together, resulting in more blocks read per disk request. As a result, the number of disk requests decreases.

Third, the effect of prefetching on disk requests cannot always be predicted based on the effect of prefetching on the hit ratio. The relationship between prefetching and disk requests can be complex and is closely tied to the application file access patterns. *Cscope* gives an example where prefetching increases the opportunity for clustering, which in turn reduces the number of disk requests. However, if prefetched blocks are not accessed, it may not result in a decrease in the number of disk requests. This is observed for random assess applications in Section 5.3.2.

Fourth, the reduction in the number of disk requests due to kernel prefetching does not necessarily translate into a proportional reduction in the execution time. For example with prefetching and at 64MB cache size, the numbers of disk requests for LRU, LRFU and ARC decrease by 31%, 31%, and 30%, while the corresponding running times only decrease by 1.6%, 1.2%, and 1.0%, respectively. In contrast, at 128MB cache size, the number of disk requests for LIRS and LRU-2 reduce by 64.4% and 75.4%, respectively, which are significant changes and cause the execution time to decrease by 43.2% and 52.5%, respectively.

Lastly, prefetching can result in significant changes in the relative performance of replacement algorithms. Some interesting effects are seen as the cache size is increased to 128MB. For example, without prefetching the hit ratios of 2Q and OPT differ by 18%. With prefetching, the gap is reduced to under 6%. As another example, without prefetching LRU-2 and LIRS achieve 51% and 40% lower hit ratios than 2Q, respectively, but with prefetching, they achieve 2% and 1% higher hit ratios than 2Q, respectively.

Glimpse

Glimpse (Figure 4) also benefits from prefetching. The curves for the hit ratio shift up by an average of 10% for small cache sizes. In contrast, the hit ratios in *cscope* increase by over 35%. The smaller increase can be explained as follows. About 72% of the files accessed by *glimpse* are one or two blocks long, where there is little benefit from prefetching. In addition, 5% of the accesses to the rest of the files read chunks of 25 or more blocks, which limits the additional benefit that can be derived from clustering with prefetching compared to without prefetching. As a result, the benefit from prefetching is small compared to in *cscope*.

The changes in the relative behavior of different algorithms observed in *cscope* with prefetching are also observed in *glimpse*. In fact, there is a flip between the performance of

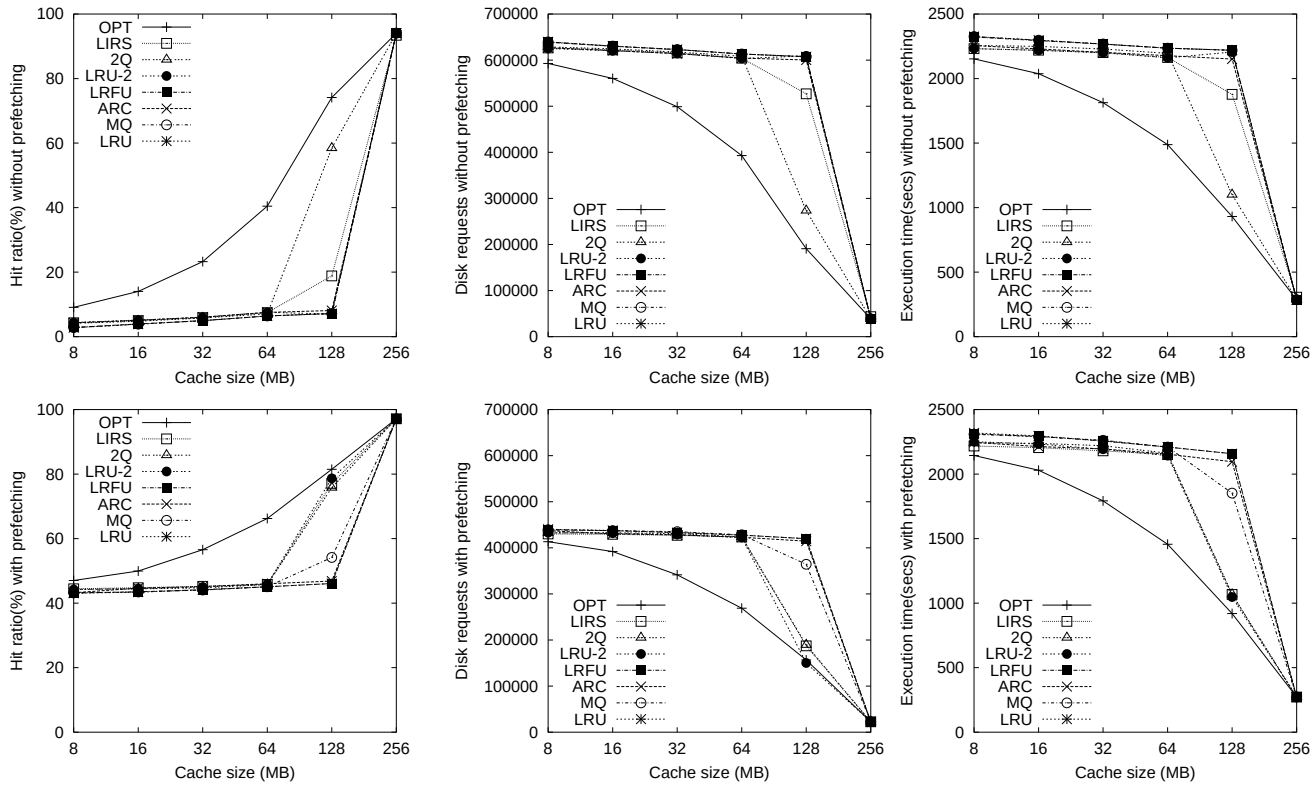

Figure 3: The hit ratio, number of clustered disk requests, and execution time for *cscope* under various schemes.

2Q and ARC at 128MB cache size. Without prefetching, 2Q and ARC have hit ratios of 7.2% and 11.4%, respectively; with prefetching, the hit ratios change to 19.4% and 11.5%, respectively. Similarly, there is a flip between 2Q and LIRS at cache sizes 64MB and 128MB.

The improved hit ratio due to prefetching does not translate into proportionally reduced disk requests, as shown in Figure 4. The hit ratio for LRU-2 at 64MB cache size increases by 10% with prefetching, but the corresponding number of disk requests decreases by less than 2%. This can be explained as follows. As discussed earlier, *glimpse* either reads small files where clustering has little advantage, or reads big files in big chunks where clustering is able to minimize disk requests even without prefetching. While prefetching provides improvements in the hit ratio by bringing in blocks before they are accessed, it does not provide any additional benefit of clustering I/Os together into chunks as it would for small-sized accesses to large files. Hence, there is only a small reduction in the number of disk requests.

In contrast to the hit ratio, the number of disk requests provides a much better indication of the relative execution time among different replacement algorithms. For instance, at 32MB cache size with prefetching, the hit ratio for 2Q, LRU-2, LRFU, and MQ increase by 13%, 12%, 14%, and 14%, respectively. But the execution times for these schemes show virtually no improvement (a mere 0.5% decrease in time) as observed from the graph. Examining the number of disk requests shows that for the four schemes, the number of disk requests decreases by under 1% with prefetching, which gives a much better indication of the effect of prefetching on the execution time. As another example, at 128MB cache size, the hit ratios for 2Q, LRU-2, LRFU, and MQ increase by about 12%. The corresponding numbers of disk requests decrease by less than 1% except for LRU-2, for which the number decreases by 2%. Hence, while the hit ratio suggests improvement in execution time for all schemes, the number of disk requests suggests a slight improvement for LRU-2 only. The actual execution time is virtually unchanged except for LRU-2 where it is 1% lower than without prefetching. Similar correspondence between the number of disk requests and the execution time can be observed for other cache sizes.

Gcc

In *gcc* (the results for *gcc* can be found in [6]), the benefit from prefetching is not as pronounced as in *cscope* and *glimpse*. This is because in *gcc* many accesses are to small files, for which there is little opportunity for prefetching. As a result, all three performance metrics, the hit ratio, the number of disk requests, and the execution time, are almost identical with and without prefetching for each replacement algorithm.

Viewperf

The behavior of the different cache replacement algorithms in *viewperf* (the results for *viewperf* can be found in [6]) is similar to that observed in *cscope*. As *viewperf* accesses large files in small chunks, it is able to see maximum benefit from prefetching. For instance, the hit ratio on average improves from 35% to 87%, and the number of disk requests is reduced by a factor of 11 when prefetching is turned on.

Finally, since *viewperf* is a CPU-bound application, the improvement in the hit ratio and number of disk requests do not translate into any significant reduction in the execu-

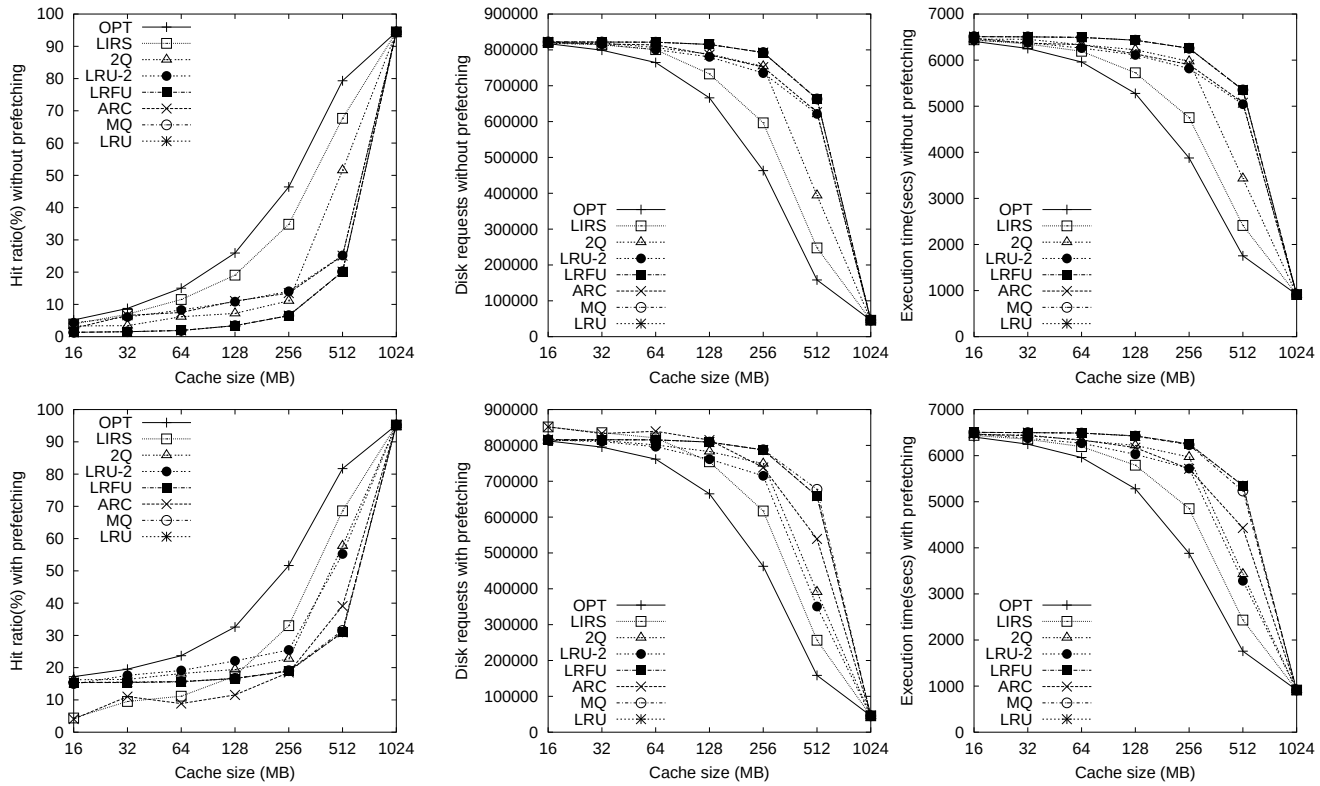

Figure 4: The hit ratio, number of clustered disk requests, and execution time for *glimpse* under various schemes.

tion time. For example, while the hit ratio is improved and the number of disk requests is reduced as mentioned above, the execution time is reduced from 880 seconds to only 850 seconds, i.e., by 3.41%.

5.3.2 Random access applications

The two applications in this group, *tpc-h* and *tpc-r*, exhibit predominantly random accesses. As expected (shown in Figures 5 and 6), prefetching provides little improvement in the hit ratio for these applications. Furthermore, most of the prefetched blocks are not accessed and as a result both the number of disk requests and the execution time are doubled. For example, for *tpc-h*, with prefetching, the number of synchronous disk requests, each of which prefetches an extra block, is almost the same as the number of disk requests that only access on-demand blocks without prefetching. Furthermore, an almost equal number of asynchronous disk requests are issued, each of which prefetches between 2 to 4 blocks. As a result, the total number of disk blocks under prefetching is about 5 times that without prefetching. This in turn results in the running time of *tpc-h* to be doubled with prefetching.

Although with prefetching both *tpc-h* and *tpc-h* show small improvement in hit ratio, the significant increase in the number of I/Os translates into a significant increase in the execution time. This is a clear example where the relative hit ratio is not indicative of the relative execution time while the number of disk requests gives a much better indication of the relative performance of different replacement algorithms.

We observe that even the elaborate prefetching scheme of the Linux kernel is unable to stop useless prefetching and prevent performance degradation. This implies that applications dominated with random I/O accesses such as database applications should disable prefetching or use alternative file access methods when running on standard file systems such as in Linux.

5.3.3 Concurrent applications

The applications in this group contain accesses that are a mix of sequential and random accesses. *Multi1* contains more sequential accesses as compared to *multi2* and *multi3*, and are dominated by *cscope*. Therefore, the hit ratios and disk requests for *multi1* with or without prefetching exhibit similar behavior as that for *cscope*. *Multi2* behaves similarly as *multi1*. However, prefetching does not improve the execution time, since this mix contains CPU-bound *viewperf*. *Multi3* has a large number of random accesses due to *tpc-h*, and therefore its performance curves look similar to those of *tpc-h*. The detailed results for these applications can be found in [6].

5.3.4 Synchronous vs. asynchronous prefetching

We illustrate the breakdown of disk requests into synchronous and asynchronous requests for *cscope* only at 128MB cache size due to page limitation. Other sequential access applications follow a similar trend. Table 5 shows the number of (clustered) disk requests that are issued without prefetching, along with synchronous and asynchronous disk requests when prefetching is enabled. For each case, the average number of blocks accessed per disk request is also reported. The case without prefetching represents the actual on-demand accesses issued by the programs. These blocks, if not present in the cache, are always read synchronously.

Table 5 shows that the total number of disk requests (syn-

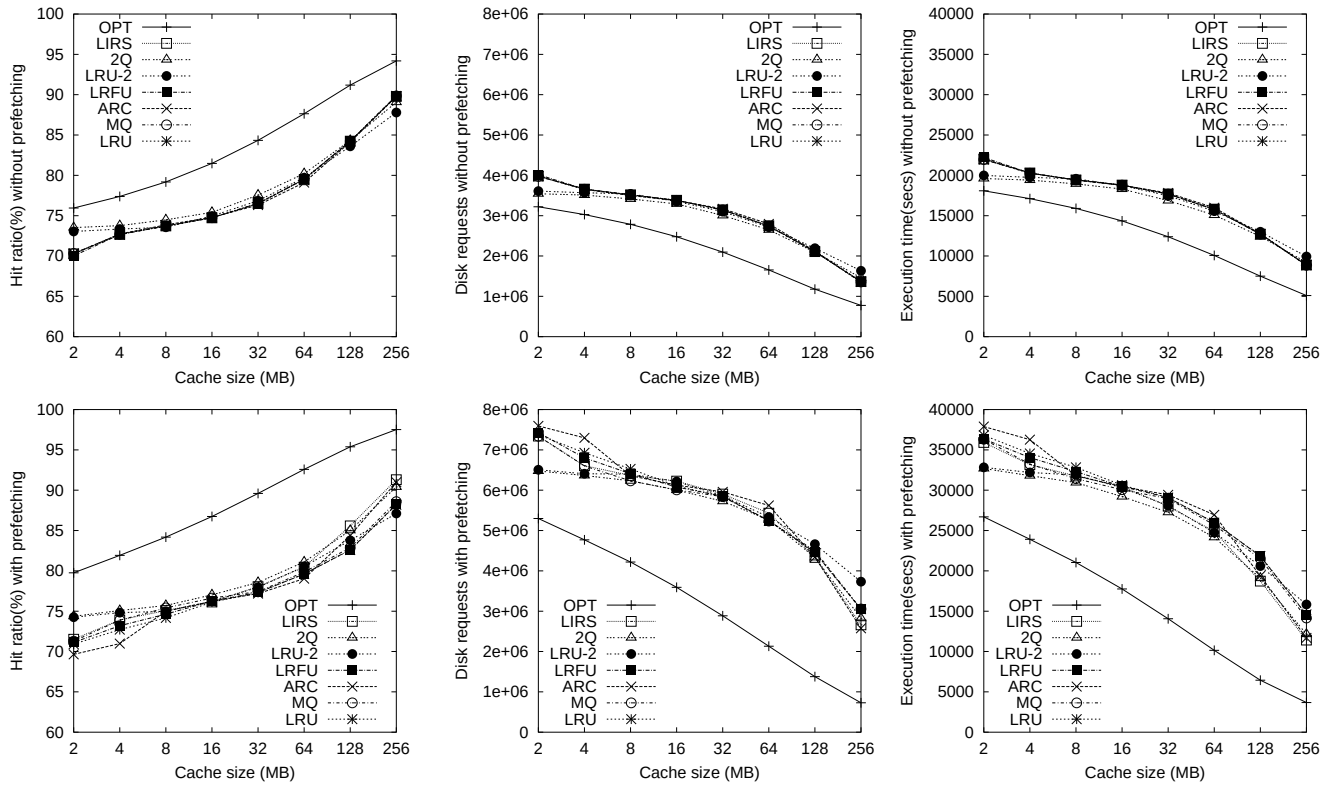

Figure 5: The hit ratio, number of clustered disk requests, and execution time for *tpc-h* under various schemes.

	No prefetching		Prefetching			
			Synchronous		Asynchronous	
	I/Os	Size	I/Os	Size	I/Os	Size
OPT	190860	1.5	141128	1.5	15573	5.4
LIRS	486799	1.7	153078	1.5	14811	7.3
2Q	273568	1.7	161746	1.7	27718	6.9
LRU-2	609333	1.7	38542	1.8	111544	4.1
LRFU	607421	1.7	358085	1.7	61778	6.8
ARC	600312	1.7	352821	1.7	61682	6.8
MQ	607421	1.7	358064	1.4	6254	9.3
LRU	607421	1.7	358085	1.7	61778	6.8

Table 5: Number and size of synchronous and asynchronous disk I/Os in *cscope* at 128MB cache size.

chronous and asynchronous) with prefetching is at least 30% lower than without prefetching for all schemes except OPT, and most of the reduction in disk requests comes from reducing synchronous disk requests and instead issuing asynchronous disk requests which can be overlapped with the CPU time. The asynchronous disk requests prefetch between 5 to 9 blocks per access, efficiently utilizing the disk bandwidth.

6. RELATED WORK

6.1 Other Replacement Algorithms

In addition to recency/frequency-based cache replacement algorithms, many of which are described in Section 4, there are two other classes of cache replacement algorithms: hint-based and pattern-based.

Pattern-based algorithms SEQ [13] detects sequential page fault patterns and applies the Most Recently Used (MRU) policy to those pages. For other pages, the LRU replacement is applied. However, SEQ does not distinguish sequential and looping references. EELRU [34] detects looping references by examining aggregate recency distribution of referenced pages and changes the eviction point using a simple cost/benefit analysis. DEAR [11, 12], UBM [21], and PCC [14] are three closely related pattern-based buffer cache replacement schemes that explicitly separate and manage blocks that belong to different reference patterns. The patterns are classified into three categories: sequential, looping, and other (random). The three schemes differ in the granularity of classification; classification is on a per-application basis in DEAR, a per-file basis in UBM, and a per-call-site basis in PCC. Due to page limitation, we did not evaluate pattern-based algorithms in this paper.

Hint-based algorithms In application-controlled cache management [8, 32], the programmer is responsible for inserting hints into the application which indicate to OS what data will or will not be accessed in the future and when. The OS then takes advantage of these hints to decide what cached data to discard and when. This can be a difficult task as the programmer has to accurately identify the access patterns of the application so that the resulting hints do not degrade the performance. To eliminate the burden on the programmers, compiler inserted hints are proposed [5]. These methods provide the benefits of user inserted hints for existing applications that can be simply recompiled with the proposed compiler. However, more complicated access patterns or input dependent patterns may be difficult for the compiler to characterize.

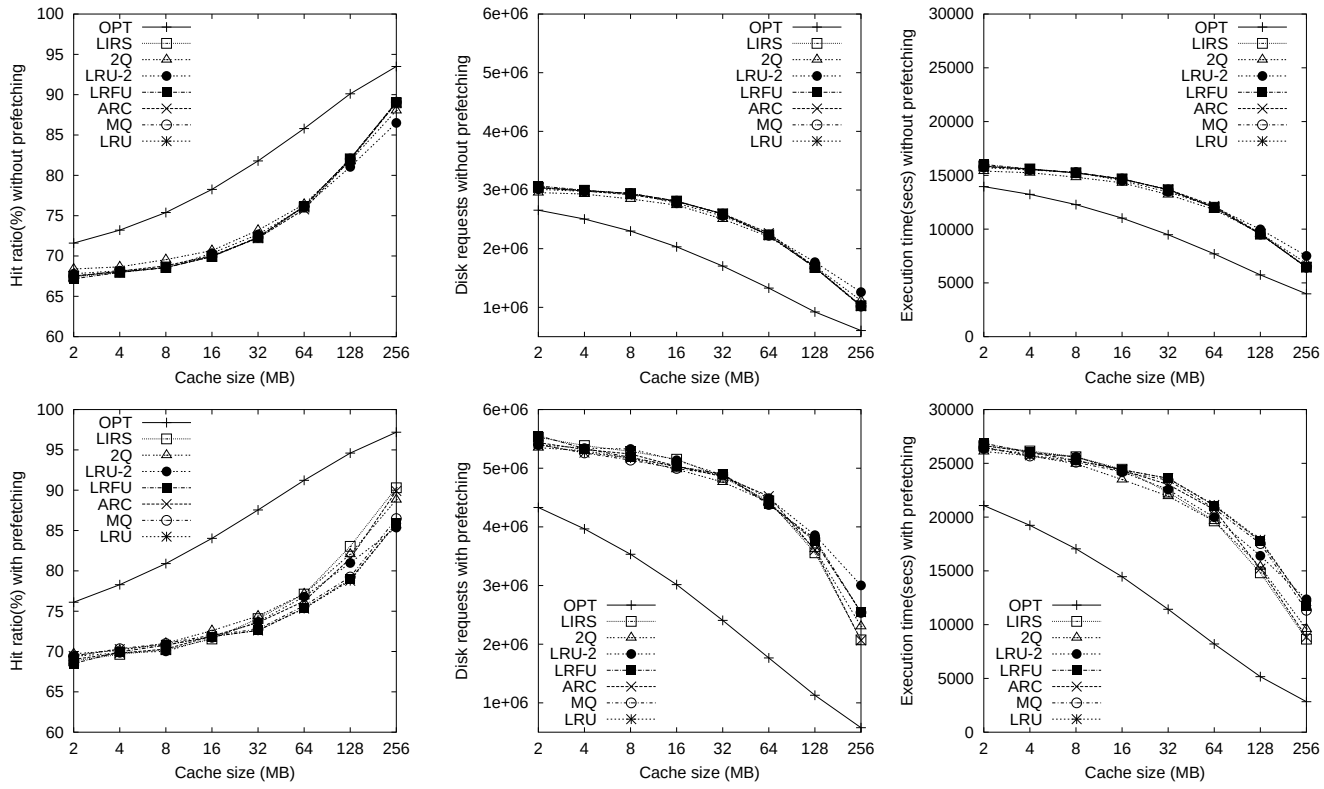

Figure 6: The hit ratio, number of clustered disk requests, and execution time for *tpc-r* under various schemes.

6.2 I/O Prefetching

A number of work have considered I/O prefetching based on hints (reference patterns) about an application's I/O behavior. Such hints can be explicitly inserted by the programmer, or derived and inserted by a compiler [28], or even explicitly prescribed by a binary rewriter [10] in cases where recompilation is not possible.

Alternatively, dynamic prefetching has been proposed to detect reference patterns in the application and predict future block references. Future references to the file can be predicted by probability graphs [16, 40]. Another approach uses time series modeling [38] to predict temporal access patterns and issue prefetches during computation intervals. Prefetch algorithms tailored for parallel I/O systems have also been studied [1, 20, 22].

6.3 Integrated Prefetching and Caching

In [7], Cao et al. point out the interaction between integrated prefetching and caching and derive an aggressive prefetching policy with excellent competitive performance in the context of complete knowledge of future accesses. The work is followed by many integrated approaches, for example, [1, 8, 20, 22, 23, 32, 37] which are either offline or based on hints of I/O access patterns.

While an integrated prefetching and caching design is not supported in any modern operating system, all modern operating systems implement some form of kernel prefetching in their file systems, on which buffer caching is layered on top. Though not integrated, kernel prefetching is expected to affect the buffer caching behavior as in integrated approaches. However, most recent caching algorithm studies did not consider the performance impact of kernel prefetching [2, 13, 18, 19, 25, 27, 30, 33, 34].

Finally, in [36], Belady's algorithm [3] is extended to simultaneously perform caching and read-ahead and the extended algorithm minimizes the cache miss ratio, or the number of disk I/Os. The offline algorithm is effectively a layered approach where caching is layered on top of disk read-ahead.

7. CONCLUSION

Despite the well-known interactions between prefetching and caching, almost all buffer cache replacement schemes proposed over the last decade were studied without taking into account the file system prefetching which exists in all modern operating systems. In this paper, we performed a detailed simulation study of the impact of the Linux kernel prefetching on the performance of a set of eight replacement algorithms.

Our study shows such kernel prefetching can have a significant impact on the relative performance of different replacement algorithms. In particular, prefetching can significantly improve the hit ratios of some sequential access applications (*cscope*) but not other sequential access applications (*gcc*) or random access applications (*tpc-h and tpc-r*); the difference in hit ratios may (*cscope, gcc, viewperf*) or may not (*glimpse, tpc-h, tpc-r*) translate into similar difference in the number of disk requests; the difference in the number of disk requests may (*gcc, glimpse, tpc-h, tpc-r*) or may not (*cscope, viewperf*) translate into differences in the execution time; as a result, the relative hit ratios and numbers of disk requests may (*gcc*) or may not (*cscope, glimpse, viewperf*) be a good indication of the relative execution times; for random access

applications (*tpc-h and tpc-r*), prefetching has little impact on the relative hit ratios of different algorithms, but can have a significant adverse effect on the number of disk requests and the execution time compared to without prefetching. This implies that ,if possible, prefetching should be disabled when running random access applications such as database systems on standard file systems such as in Linux. These results clearly demonstrate the importance for buffer caching research to take file system prefetching into consideration.

Our study also raise several questions on the buffer cache replacement algorithm design. How to modify an existing buffer cache replacement algorithm (or design a new one) to leverage the knowledge about access history to explicitly perform prefetching, as opposed to passively using kernel prefetching? What are the potential benefits of such a change? How should the Linux or BSD kernel prefetching be changed to avoid counter-productive prefetches for random access applications? We are studying these questions as part of our future work.

8. REFERENCES

[1] S. Albers and M. Bttner. Integrated prefetching and caching in single and parallel disk systems. In *Proc. 15th ACM SPAA*, June 2003.

[2] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proc. 3rd USENIX FAST*, March 2004.

[3] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel (2nd Edition)*. O'Reilly and Associates, Inc., 2003.

[5] A. D. Brown, T. C. Mowry, and O. Krieger. Compiler-based I/O prefetching for out-of-core applications. *ACM TOCS*, 19(2):111–170, 2001.

[6] A. R. Butt, C. Gniady, and Y. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Technical Report TR-ECE-05-04*, Purdue Univeristy, March 2005.

[7] P. Cao, E. Felten, and K. Li. A study of integrated prefetching and caching strategies. In *Proc. ACM SIGMETRICS*, May 1995.

[8] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM TOCS*, 14(4):311–343, 1996.

[9] R. W. Carr and J. L. Hennessy. WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc. ACM SOSP-08*, Dec. 1981.

[10] F. W. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. 3rd USENIX OSDI*, Feb. 1999.

[11] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme. In *Proc. USENIX ATC*, June 1999.

[12] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: a case for fine-grained buffer management. In *Proc. ACM SIGMETRICS*, June 2000.

[13] G. Glass and P. Cao. Adaptive page replacement based on memory reference behavior. In *Proc. ACM SIGMETRICS*, June 1997.

[14] C. Gniady, A. R. Butt, and Y. C. Hu. Program counter based pattern classification in buffer caching. In *Proc. 6th USENIX OSDI*, Dec. 2004.

[15] B. L. W. Gregory R. Ganger and Y. N. Patt. The disksim simulation environment. In *University of Michigan, EECS, Technical Report CSE-TR-358-98*, Feb. 1998.

[16] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proc. 8th ICPDCS*, Sep. 1995.

[17] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proc. USENIX ATC*, Apr. 2005.

[18] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proc. ACM SIGMETRICS*, June 2002.

[19] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. 20th VLDB*, Jan. 1994.

[20] M. Kallahalla and P. J. Varman. Optimal prefetching and caching for parallel I/O sytems. In *Proc. 13th ACM SPAA*, July 2001.

[21] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A Low-Overhead, High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. 4th USENIX OSDI*, Oct. 2000.

[22] T. Kimbrel and A. R. Karlin. Near-optimal parallel prefetching and caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.

[23] T. Kimbrel, A. T. R. H. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. 2nd USENIX OSDI*, Oct. 1996.

[24] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proc. ACM SIGMETRICS*, May 1999.

[25] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1360, 2001.

[26] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Winter 1994 Technical Conference*, Jan. 1994.

[27] N. Megiddo and D. S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. 2nd USENIX FAST*, Mar. 2003.

[28] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proc. 2nd USENIX OSDI*, Oct. 1996.

[29] MySQL: The world's most popular open source database. `http://www.mysql.com/`, 2005.

[30] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Conference*, May 1993.

[31] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *J. ACM*, 46(1):92–112, 1999.

[32] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proc. ACM SOSP-15*, Dec. 1995.

[33] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. ACM SIGMETRICS*, May 1990.

[34] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: simple and effective adaptive page replacement. In *Proc. ACM SIGMETRICS*, May 1999.

[35] J. Steffen. Interactive examination of a c program with cscope. In *Proc. USENIX Winter Technical Conference*, Jan. 1985.

[36] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, 1999.

[37] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed multi-process prefetching and caching. In *Proc. ACM SIGMETRICS*, June 1997.

[38] N. Tran and D. A. Reed. Arima time series modeling and forecasting for adaptive i/o prefetching. In *Proc. 15th ICS*, June 2001.

[39] Transaction Processing Performance Council. `http://www.tpc.org/`, 2005.

[40] V. Vellanki and A. L. Chervenak. A cost-benefit scheme for high performance predictive prefetching. In *Proc. ACM/IEEE SC'99*, Nov. 1999.

[41] Y. Zhou, P. M. Chen, , and K. Li. The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches. In *Proc. USENIX ATC*, June 2001.