The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms

Ali R. Butt, *Member*, *IEEE*, Chris Gniady, *Member*, *IEEE*, and Y. Charlie Hu, *Member*, *IEEE*

Abstract—A fundamental challenge in improving 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. Moreover, since prefetching can increase the number of blocks clustered for each disk I/O and, hence, the time to complete the I/O, the reduction in the number of disk I/Os may not translate into proportional reduction in the total I/O time. These results demonstrate the importance of buffer caching research taking file system prefetching into consideration and comparing the actual disk I/Os and the execution time under different replacement algorithms.

Index Terms—Metrics/measurement, operating systems, file systems management, operating systems performance, measurements, simulation.

1 Introduction

A fundamental challenge 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 [10]. 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 [3], [12], [13], [15], [16], [21], [22], [24], [27], [29], [32], [35], [36]. However, although a few of these replacement algorithm studies have been implemented and reported measured results, the vast majority used trace-driven simulations and used the cache hit ratio as the main performance metric in comparing different algorithms.

Prefetching is another highly effective technique for improving the I/O performance. The main motivation of 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 [8], [9], [34]. 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, which exploit user-inserted hints, have also been studied [8], [9], [34]. However, such user-inserted hints place a burden on the programmer as the programmer has to accurately identify the access patterns of the application.

The 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 processes, where requests are clustered to 64 Kbytes (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 [8], almost all buffer cache replacement algorithms have been proposed and studied comparatively without taking into account kernel-driven prefetching [3], [15], [21], [22], [27], [29], [32], [35], [36].

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. By using a cache simulator that faithfully implements the kernel prefetching of the Linux OS, 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, and the ultimate performance measure—the actual running time of applications using an accurate disk simulator, DiskSim [17]. 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:

- We develop a buffer caching simulator, AccuSim, that faithfully implements the Linux kernel prefetching

- *A.R. Butt is with the Department of Computer Science, Virginia Polytechnic Institute and State University, McBryde Hall (0106), Blacksburg, VA 24061. E-mail: butta@cs.vt.edu.*
- *C. Gniady is with the Department of Computer Science, University of Arizona, Gould-Simpson Building, 1040 E. 4th Street, PO Box 210077, Tucson, AZ 85721-0077. E-mail: gniady@cs.arizona.edu.*
- *Y.C. Hu is with the School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Ave., West Lafayette, IN 47907. E-mail: ychu@purdue.edu.*

Manuscript received 4 Nov. 2005; revised 30 Aug. 2006; accepted 9 Nov. 2006; published online 14 Feb. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0396-1105.
Digital Object Identifier no. 10.1109/TC.2007.1029.

and I/O clustering and interfaces with a realistic disk simulator, DiskSim [17], to allow the performance study of different replacement algorithms in a realistic environment.
- We show how we can adapt a set of eight representative 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 but also 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 on disk I/O load, but the actual application running time is the only definitive performance metric in the presence of asynchronous kernel prefetching.

Our experimental results clearly demonstrate the importance of future buffer caching research incorporating file system prefetching. To facilitate this, we are making our simulator publicly available [1]. The simulator implements the kernel prefetching and I/O clustering mechanisms of the Linux 2.4 kernel and provides functionalities that allow easy integration of any cache replacement algorithms into the simulator for accurately studying and comparing the buffer cache hit ratio, the number of disk requests, and the application execution time under different replacement algorithms.

The outline of this paper is listed 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 by using Belady's algorithm as an example. Section 4 summarizes the various buffer cache replacement algorithms that are evaluated in this paper. Section 5 describes the architecture of our buffer cache simulator, AccuSim, and how it interfaces with DiskSim to accurately simulate synchronous and asynchronous prefetching requests. Section 6 presents trace-driven simulation results of performance evaluation and comparison of the studied replacement algorithms. Finally, Section 7 discusses additional related work and Section 8 concludes this paper.

2 Prefetching in File Systems

Our study is motivated by the widespread adoption of transparent prefetching in the file system buffer cache management in modern operating systems. In this section, we describe in detail the kernel prefetching mechanisms in two modern operating systems: 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. Fig. 1 shows the various steps that a file system access has to go through before it is 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 a minimal cost.

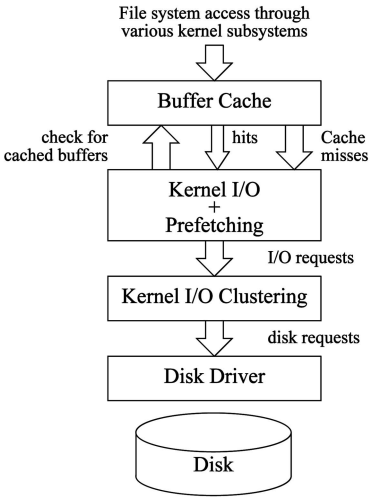


Fig. 1. Various kernel components on the path from file system operations to the disk.

Prefetching in the Linux kernel is beneficial for *sequential accesses* to a file, that is, 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 on 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 that 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, that is, the block is not in A_1's *read-ahead group*, then 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, then the kernel decides that the file is being accessed sequentially and performs more aggressive prefetching. The size of the previous *read-ahead group*, that is, the number of blocks that were previously prefetched, is

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

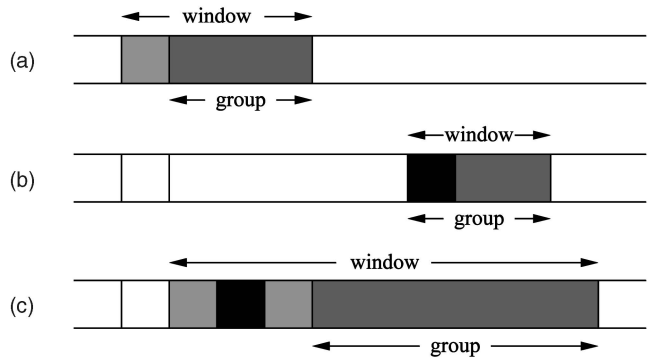


Fig. 2. Prefetching in Linux. The shaded regions show the blocks of the file that are cached and the black region shows the block being accessed. Also shown are the *read-ahead windows* and *read-ahead groups* before and after prefetching. (a) Before prefetching. (b) After synchronous prefetching. (c) After asynchronous prefetching.

doubled to determine the number of blocks (N) to be prefetched on this access. However, N is never increased beyond a prespecified 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 prefetching associated with A_2 is also issued, the next access can benefit from prefetching if it accesses a block either 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* and 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 cases: 1) The block is within the *read-ahead window* and has already been prefetched, justifying further asynchronous prefetching, or 2) the block is outside the *read-ahead window* and synchronous prefetching is invoked.

Fig. 2 (adapted from [5]) shows the adjustment of the *read-ahead group* and the *read-ahead window* under synchronous and asynchronous prefetching. In Fig. 2a, 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*, then synchronous prefetching is employed and the *read-ahead window* is reset to the new *read-ahead group* (Fig. 2b). If the next access is to a block within the *read-ahead window*, then asynchronous prefetching is employed and the *read-ahead group* and *read-ahead window* are updated accordingly (Fig. 2c).

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 that 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 overload the disk [5]. 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*, that is, 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 Fig. 2c is to the light gray area, then 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), then no new disk request is generated. Otherwise, a new disk request is created to fetch all of the consecutive blocks by all of the previous I/O requests accumulated in the queue. A new disk request can also be triggered directly (for example, 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 to a block immediately following the last accessed block, that is, access to a block B is sequential if the 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, then 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, then 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, then 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 the 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

TABLE 1
An Example Scenario where LRU Results in Fewer Disk I/Os Compared to Belady's Replacement Algorithm

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

The cache contents after each access are shown in LRU stack order, from left to right, for both of the algorithms. The blocks read on a cache miss are shown in boldface.

replacement algorithms, we give an example that shows that, given kernel prefetching, Belady's algorithm [4] can be nonoptimal in reducing disk requests.

For simplicity, we assume a prefetching algorithm that is simpler than that used in Linux. We use a cache size of eight 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 three. If the following access is to a block that has been prefetched on the previous access, then the number of blocks to be prefetched on the next miss is increased to eight; otherwise, it remains as three. 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, as prefetched blocks are added to the cache, they do not cause eviction of the current on-demand block or any of the blocks already prefetched along with the on-demand block. If prefetching a block would result in such an eviction, then we do not prefetch any more blocks on the current access. For example, consider a case where block 1 is accessed on demand and blocks 2 to 7 are to be prefetched. If, after blocks 1 to 4 are added to the cache, adding block 5 would result in the eviction of block 2 (which was just prefetched), then we will not prefetch blocks 5 to 7.

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 of the 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 eight by using Belady's algorithm and six by 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 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 algorithms used in our evaluation of the impact of kernel prefetching. For each algorithm, we summarize the original algorithm, followed by the adapted version, which 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, that is, when implemented in the Linux buffer cache.

Since all practical replacement algorithms use the notion of recency/frequency in deciding the victim block for eviction to preserve the original design philosophy in incorporating prefetched blocks into each of the replacement algorithms, we strive to preserve the recency/frequency information associated with the prefetched blocks.

All of the replacement algorithms use the notion of recency in deciding the victim block for eviction. Since the prefetched blocks have not been accessed, they can be added to the cache in two obvious ways: as if they were accessed least recently and, hence, they would be placed in the LRU location, or as if they were accessed most recently and, hence, they would be placed in the most recently used (MRU) location. Since the former approach makes the prefetched blocks immediate candidates for eviction and, since placing prefetched blocks at the MRU location is consistent with the actual Linux implementation, we use this design in all eight algorithms that utilize recency information: optimal replacement algorithm (OPT), LRU, LRU-2, 2Q, Low Interreference Recency Set (LIRS), Least Recently/Frequently Used (LRFU), Multi-Queue buffer management algorithm (MQ), and Adaptive Replacement Cache (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 can be accommodated in the actual cache. For these schemes, if a prefetched block is evicted from the cache before it is accessed, then it is simply discarded, that is, not moved into the ghost cache.

OPT. OPT is based on Belady's cache replacement algorithm [4]. This is an offline algorithm, as it requires an oracle to determine future references to a block. OPT evicts the block that will be referenced furthest 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 that the kernel prefetching is oblivious to future references. However, OPT can immediately determine wrong prefetches, that is, 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 evicted right away if they are accessed further in the future than all of the 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 of 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 algorithm. It is usually implemented as an approximation [10] 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 [32], [33] algorithm tries to avoid the pathological cases of LRU. LRU-K replaces a block based on the *K*th-to-the-last reference. The oldest resident based on this metric is evicted. For simplicity, the authors recommended that $K = 2$. 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 to 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 [22] 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 first-in, first-out (FIFO) order, the addresses of these replaced blocks are temporarily placed in a ghost buffer called $A1out\ queue$. When a block is rereferenced 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 algorithms, that is, 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, then it is simply discarded, as opposed to being moved into the $A1out\ queue$. This is to ensure 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, then it is promoted into Am as if it is accessed on demand.

LIRS. LIRS [21] (and its variant Clockpro [19]) is another recently proposed algorithm which maintains a complexity similar to that of LRU by using the distance between the last and the second-to-the-last references to estimate the likelihood of the block being referenced again. LIRS maintains a variable-sized 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. Similarly to the CRP of LRU-2 or the Kin of 2Q, LIRS allocates a small portion of the cache to store recently seen HIR blocks.

In the presence of kernel prefetching, LIRS is modified so as 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, then 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, then 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. LRFU [27] 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 to how the initial weight for a prefetched block can be assigned 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. MQ [43] was recently proposed as a second-level replacement scheme for storage controllers. The idea is to use m LRU queues (typically, $m = 8$) $Q_0, Q_1, \ldots, Q_{m-1}$, where Q_i contains blocks that have been seen at least

2^i times but not 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, that is, 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 that 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 $currentTime$, then 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 we can 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 record 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 ARC [29] (and its variant CAR [3]). 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, whereas 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, then it is moved not to the second queue but to the first queue. If the block is prefetched correctly, then it will be moved to the second queue upon the subsequent on-demand access. This way, if prefetching brings in blocks that are not accessed again, then 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 AccuSim: Accurate Buffer Cache Simulator

We have discussed the various kernel subsystems that come into play before an I/O access from an application is delivered to the actual disk. The complexity of interactions between such subsystems entails the various buffer caching algorithms being comparatively studied in a realistic setting. However, creating an actual kernel implementation of the algorithm being designed and the algorithms to be compared against can be a laborious task. In addition, even with such a kernel implementation, it is often nontrivial to obtain a controlled environment, for example, a fixed cache size in the presence of a unified buffer cache, in the actual kernel for comparing various algorithms on an even basis. To overcome this challenge faced by cache replacement algorithm designers, we have developed a buffer cache simulator, AccuSim, that realistically models all of the kernel subsystems involved in buffer cache management in modern operating systems: the buffer cache replacement algorithm, kernel prefetcher, and I/O clustering. In addition, the simulator interfaces with a realistic disk simulator, DiskSim [17], to enable the comprehensive study of the hit ratios, the number of disk requests, and the execution times of applications under various replacement algorithms and cache sizes. Such a simulator will greatly simplify the otherwise difficult task of systematic and accurate evaluation of various caching algorithms. In the following, we present the design details of the simulator.

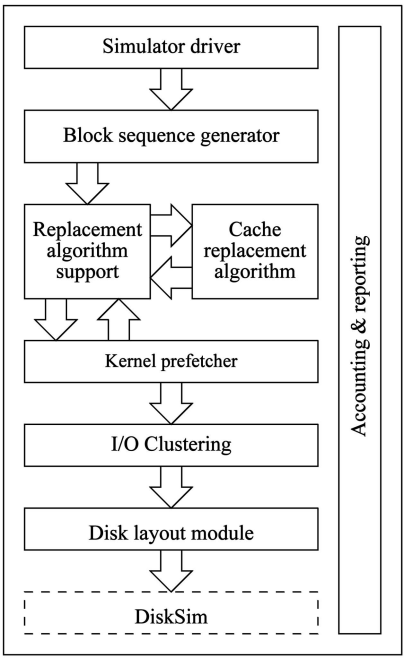


Fig. 3. The architecture of the cache replacement algorithm simulator.

5.1 Simulator Architecture

Fig. 3 shows the main modules in the architecture of the simulator. The simulator is driven by a trace that records the I/O operations during the execution of an application. In particular, for each I/O operation, the trace contains the identifier (`inode`) of the file that is accessed, the offset into the file of the first block that is accessed, the number of blocks being accessed, the I/O issue time, and the I/O completion time. A trace with this information can easily be collected using the standard system call tracing tool `strace`, which can be easily modified to collect only the desired information. The trace is then processed to calculate the time difference between the completion time of an I/O operation and the issue time of the next, which is interpreted as the computation time in between the two I/O operations. Afterward, the I/O issue and completion times are discarded and the processed trace only records the sequence of I/O operations interleaved with the computation time between every two consecutive I/O operations.

A limitation of the above method for trace collection is that it does not distinguish synchronous I/O operations from explicit asynchronous I/O operations issued by the application and treats all applications issued asynchronous I/Os as synchronous. Extending the simulator to handle explicit asynchronous I/O operations from the application is part of our future work.

To drive the simulation, the collected trace is fed to the simulator driver, which processes the I/O information from the trace and uses the information to drive the rest of the

TABLE 2
Parameters for Various Replacement Algorithms

Algorithm	Parameters
OPT	*NA*
LRU	*NA*
LRU-2	CRP = 20, history size = cache size
LRFU	λ = 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

modules of the simulator. The driver also maintains data structures to simulate the system-wide open-file table for keeping track of the different files being accessed, the associated file pointers, and the per-file prefetching information (*read-ahead group*, *read-ahead window*), which the kernel typically stores in the system-wide open-file table. After processing an entry from the trace, the `inode`, the block offset, and the number of blocks to be read are forwarded to the block sequence generator module.

The block sequence generator mimics the Linux kernel in that it converts an I/O request ({`inode`, block offset, number of blocks}) that may be for multiple blocks into multiple individual block requests, each of format ({`inode`, block offset}). These individual block requests are used for ease of cache management, that is, the blocks can be added individually to the cache and the timing information associated with the blocks remains unchanged. In particular, all blocks within an I/O request are issued at the same time as the original I/O request. The disk block requests are, in turn, used to drive the buffer cache.

The replacement algorithm support module provides support functions and data structures, enabling easy integration of any replacement algorithm into the simulator. The module manages the cache on behalf of the replacement algorithm, that is, issuing I/O requests to the disk, evicting blocks, and so forth. This offloads the task of cache management from the replacement algorithm, which can then focus on determining what blocks, if any, need to be evicted. Specifically, the support module simply calls the replacement algorithm with ({`inode`, block offset}) of the block being referenced and the replacement algorithm replies with the ({`inode`, block offset}) of a block that needs to be evicted. The cache replacement algorithm module maintains internal data structures for recording the needed access information of the blocks, such as the recency and frequency.

The cache replacement algorithm module implements the algorithm under study. Our simulator currently includes implementations of the eight cache replacement algorithms discussed in Section 4: OPT, LRU, LRU-2, LRFU, LIRS, MQ, 2Q, and ARC, as well as their corresponding adapted versions, which assume kernel prefetching and explicitly manage prefetched blocks. Table 2 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. Note that the authors of LIRS suggest using 1 percent of the cache size for HIR blocks, but we observed that using this value in the presence of prefetching takes away much of the benefit from prefetching. Therefore, we used 10 percent of the cache size for HIR blocks in LIRS and our selection did not have an adverse effect on the performance of LIRS.

TABLE 3
Seagate Cheetah 9LP Parameters Used for the Simulation

Parameter	Value
Model Number	ST39102LW
RPM	10,045
Single cylinder seek time	0.831 ms
Full strobe seek time	10.627 ms
Zero-latency	no
Number of cylinders	6962
Number of data surfaces	12
Sectors/track	167–254
Number of Sectors	17783240
Bulk sector transfer time	0.105 ms
Capacity	9.1 GB

The kernel prefetcher and I/O clustering modules simulate a buffer cache that faithfully implements the kernel prefetching and I/O clustering of the 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 64 Kbytes.

This disk layout module maps the logical block offsets within a file to unique disk blocks on the simulated disk. In our experiments, we assumed an empty disk at the start of each simulation and utilized a simple first-touch layout scheme, where the files are arranged sequentially on the disk one after another in the order in which they are first accessed. We preprocessed the trace to calculate the maximum size that a file can grow to. However, the users of AccuSim can easily modify the disk layout scheme to best suit their needs. The output of the disk layout module are absolute disk block numbers. This information is then used by DiskSim to simulate the proper location of file blocks on the disk.

Finally, the accounting and reporting module interacts with all other modules of the simulator and provides statistics about the performance of the replacement algorithm being simulated. For each simulation, the simulator measures the 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. For an application with multiple processes, the I/O requests from all of the processes of the application are captured in the trace and are used to drive the simulator as if they were issued from a single process.

5.2 Simulating I/O Time under I/O Clustering and Prefetching

In the presence of I/O clustering and prefetching, it is not possible to obtain an accurate hit ratio without a time-aware simulator. Therefore, we interfaced our buffer cache simulator with DiskSim 3.0 [17], which is an accurate disk simulator. In this paper, we use DiskSim to simulate the Seagate ST39102LW disk. Table 3 shows the various parameters of the disk model.

DiskSim exports an API designed to allow it to be easily integrated into a complete storage system simulator. Our simulator utilizes this API to interface with DiskSim. The `disksim_initialize`, `disksim_shutdown`, and `disksim_dump_stats` APIs are used to start, stop, and extract I/O statistics from DiskSim, respectively. To issue a disk request to the disk, our simulator first creates a tracking entry for the request and initially marks it as "pending." Next, the simulator uses the `disksim_request_arrive` function to issue the request to DiskSim and to register a callback

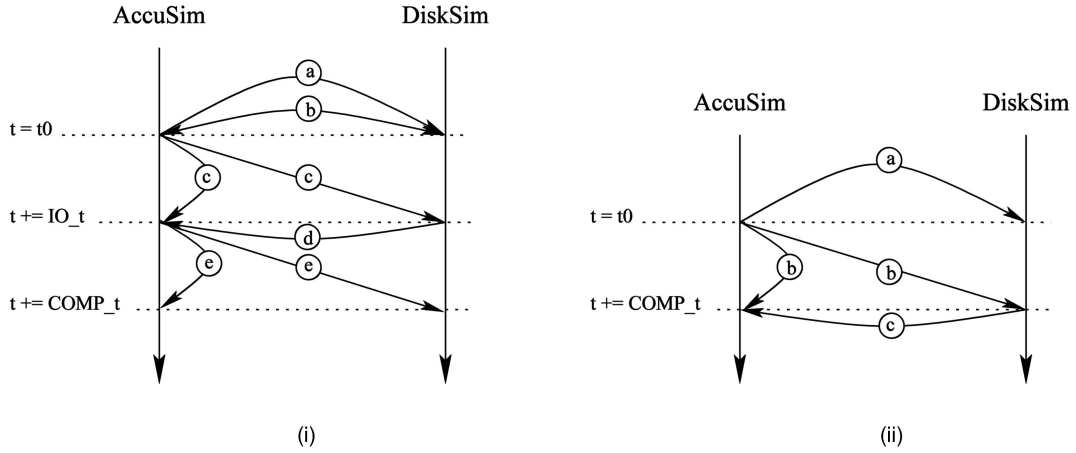


Fig. 4. Examples showing the steps involved in simulating synchronous and asynchronous disk requests. (i) (a) AccuSim issues a synchronous request to DiskSim. (b) AccuSim queries DiskSim to determine the I/O time for the request. (c) AccuSim then increments the time both locally and in DiskSim by the determined I/O time. (d) DiskSim calls the callback function associated with the I/O to mark it as complete. (e) AccuSim increments the time by the next computation time. (ii) (a) AccuSim issues an asynchronous request to DiskSim. (b) AccuSim increments the time by the next computation time. (c) DiskSim determines that the I/O has completed in the interval of the increment and calls the callback function associated with the I/O to mark it as complete.

function that will be automatically triggered by DiskSim upon completion of the request.

To synchronize the time of the buffer cache simulator with DiskSim's internal time reference, the simulator driver uses `disksim_internal_event` to update the internal DiskSim time. The internal DiskSim I/O scheduler uses this time information to generate events for all disk requests that have been completed in that time period. The events then invoke the callback functions associated with any completed requests. Finally, the callback functions mark their associated requests as "ready" in the simulator.

The trace driver updates DiskSim's time differently for synchronous and asynchronous disk requests, that is, from synchronous and asynchronous prefetching. For a synchronous request, after issuing the request, the driver queries DiskSim to determine the time that it would take the request to complete and then updates both the local and DiskSim's time by the request completion time. The steps of the process are shown in Fig. 4i. For an asynchronous request, the driver simply issues the request and continues to process the trace immediately by updating the DiskSim time by the computation time that should follow the I/O request. The steps of the process are shown in Fig. 4ii. This way, the asynchronous request is effectively overlapped with the computation and may be completed at the end of the computation interval, depending on the length of the computation interval. Note that it can happen that the next I/O request (originated by the application) is to a block that was previously prefetched by the kernel by using an asynchronous request and the block is not yet ready. In this case, the simulator will wait for that request to complete by once again querying DiskSim to determine the remaining time required for the request to complete and then updating both the local and DiskSim's time by the remaining request completion time. In summary, the combined cache simulator and DiskSim allow 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 expect that the execution overhead of any reasonably efficient implementations of the replacement algorithms will be insignificant compared to the difference in the execution time from using different algorithms or prefetching.

The AccuSim simulator, which includes the implementations of the eight cache replacement algorithms studied in this paper, has been made publicly available [1].

6 Performance Evaluation

In this section, we evaluate the impact of Linux kernel prefetching on the performance of replacement algorithms by using our simulator described in the previous section.

6.1 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, access issue and completion time, file identifier (`inode`), offset into the file of the first block being accessed, and I/O size.

In this study, we consider a reference to be *consecutive* if it is to a block that is either the same or immediately next (in the same file) to the block that was accessed in the preceding reference. We define the *fraction of consecutive references* as the fraction of total references in a trace that are consecutive. For example, if accesses to a file are blocks 5, 6, 7, and 8, then the fraction of consecutive references is 75 percent. Similarly, the fractions of consecutive references in reference sequences (1, 2, 3, 4, 100, 101, 102, 103), (1, 100, 2, 3, 4, 5, 6, 7), and (1, 2, 3, 100, 101, 102, 103, 104) are 75 percent, 62.5 percent, and 75 percent, respectively.

Tables 4 and 5 show the six applications and three concurrent executions of the mixed applications used in this study. For each application, Table 4 lists the number of I/O references (in 4-Kbyte blocks), the size of the I/O footprint (unique file blocks), the working set size (defined below), the number of unique files accessed, the fraction of consecutive references, and the average and standard deviation of the amount of execution time spent between every two consecutive application-issued I/O system calls. The selected

TABLE 4
Applications and Trace Statistics

Appl.	Num. of references	Data size (MB)	Working set size (MB)	Num. of files	Frac. of cons. refs.	Inter-Appl. I/O time (ms)	
						Average	SD
cscope	1119161	260	148	10635	73.3%	0.13	0.86
glimpse	3102248	669	563	43649	74.8%	0.24	0.31
gcc	965372	163	1	10834	35.0%	2.57	23.10
viewperf	303123	495	31	289	99.2%	1.84	154.98
tpc-h	14275597	1074	59	65	53.0%	0.08	1.25
tpc-r	10986993	1082	65	82	55.5%	0.09	1.07
multi1	1278135	297	146	12246	67.9%	0.26	1.37
multi2	1580871	792	145	12477	73.6%	0.49	56.36
multi3	17383716	1744	587	43737	56.0%	0.15	2.66

applications and workload sizes are comparable to the workloads in recent studies [13], [21], [27], [29], [14], [20] and require cache sizes of up to 1,024 Mbytes. We determined the working set size of the studied applications as follows: First, the maximum hit ratio for an application under Belady's optimal cache replacement algorithm is determined using an infinite-sized cache. The working set size for this application is then determined as the minimum cache size at which the optimal replacement achieves a hit ratio equal to 90 percent of the maximum hit ratio. We classify the applications into three groups: *sequential-access applications*, which read entire files mostly consecutively (the fraction of consecutive references is high), *random-access applications*, which perform small accesses to different parts of the file (the fraction of consecutive references is low), and *mixed-access applications*, which represent a mix of concurrently running sequential-access and random-access applications. Finally, the intersystem call times in Table 4 give an indication of how much opportunity is available in each application to overlap computation with asynchronous disk requests.

6.1.1 Sequential-Access Applications

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

Cscope [37] performs source code examination. The examined source code is Linux kernel 2.4.20. *Glimpse* [28] is an indexing and query system and is used to search for text strings in 669 Mbytes 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 4 shows that *cscope* and *glimpse* are good candidates for sequential prefetching since 73.3 percent and 74.8 percent of references, respectively, occur to the consecutive blocks. The remaining fraction of references are not consecutive because they reference the beginning of new files.

Gcc builds Linux kernel 2.6.10 and is one of the commonly used benchmarks. It has a very small working set: 4 Mbytes 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 35 percent of the references occur to consecutive blocks, that is, the fraction of consecutive references is 35 percent. The reason for the low fraction of consecutive references is that, in *gcc*, 40 percent of the references are to files that are only one block long. Since accesses to such files are always to the first block of the files, the kernel always attempts prefetching (of two more blocks), but the one-block length implies that there are no blocks to be prefetched. If we ignore the references to the one-block-long files, then we find that, of the remaining references, 36 percent occur to first blocks of files and 56 percent occur to consecutive blocks, as those files are scanned. Since *gcc* almost always scans through the files that it processes, we have classified it as a sequential-access application.

Viewperf is a *System Performance Evaluation Cooperative* (*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 percent of the references are to consecutive blocks within a few large files, resulting in a perfect opportunity for prefetching.

6.1.2 Random-Access Applications

The *MySQL* [31] database system is used to run *TPC-H* (*tpc-h*) and *TPC-R* (*tpc-r*) benchmarks [41] against a database of size 10 Gbytes. *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.[2] However, 55 percent of the references occur to consecutive blocks for the following reason: These two benchmarks perform a separate I/O operation to access a database record and consecutive I/O operations tend to access records that reside on the same disk block. In this case, although only one block is accessed several times, the I/O operations are counted as consecutive and contribute to the overall high percentage of consecutive references. Note, however, that such kinds of consecutive references provide no opportunity for prefetching and, hence, we classify these applications as random-access applications. In fact, prefetching will not only increase the disk bandwidth demand but also pollute the cache.

6.1.3 Concurrent Applications

Multi1 consists of concurrent executions of *cscope* and *gcc*; it represents the workload in a code development environment.

TABLE 5
Concurrent Applications

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

2. We note that the predominant access pattern seen in database workloads depends on the size of the database. For large databases used in industry database workloads, which can be 10 to 100 times larger than ours, the entries accessed by the queries will be significantly larger and the workload tends to show a predominantly sequential pattern.

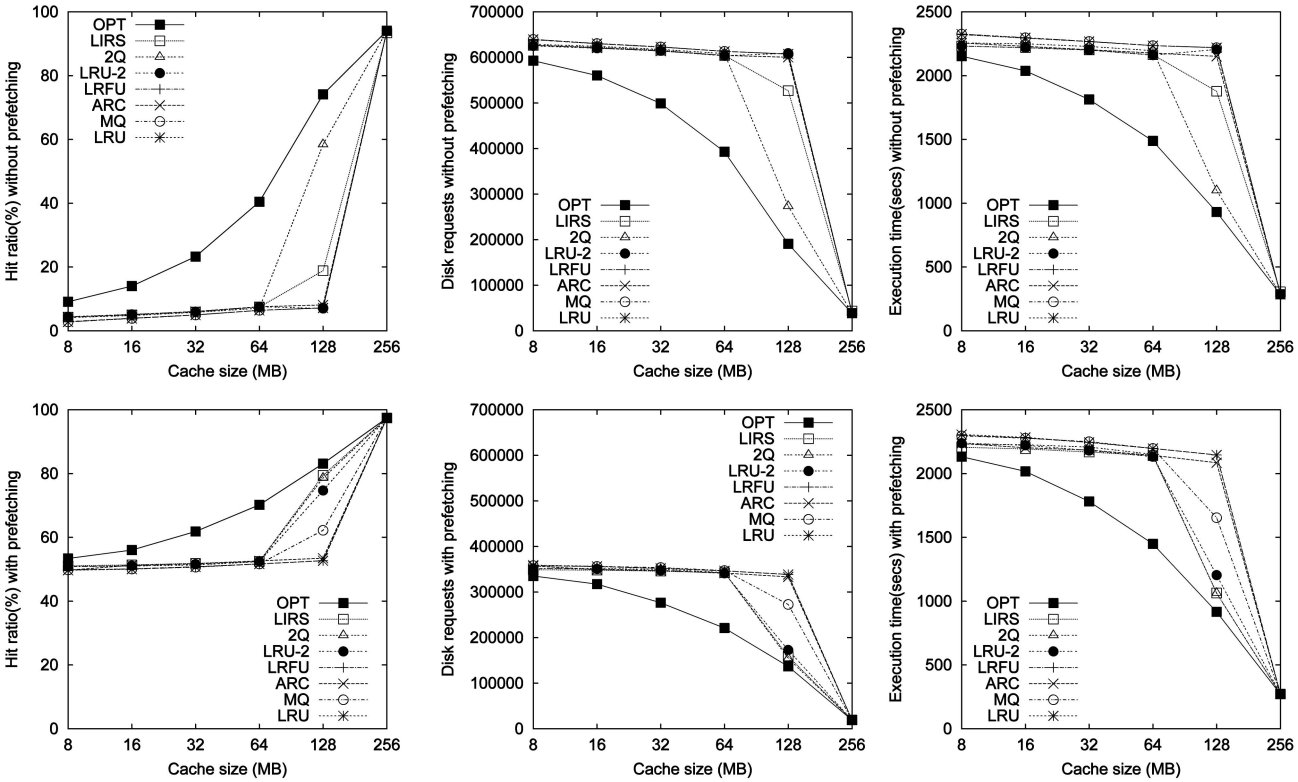


Fig. 5. The hit ratio, number of clustered disk requests, and execution time for *cscope* under various algorithms.

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.

6.2 Results

In this section, we examine the impact of kernel prefetching on each group of the applications.

6.2.1 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 in others.

Cscope. Fig. 5 shows the results for *cscope*. The following observations can be made: First, kernel prefetching has a significant impact on the hit ratio, but the improvements for different algorithms differ. At 128 Mbytes cache size, although the hit ratios for LRU, LRFU, and ARC increase by 45.5 percent, 45.5 percent, and 45.4 percent, respectively, the hit ratios for LRU-2, LIRS, and 2Q increase by 67.7 percent, 60.6 percent, and 20.2 percent, respectively. This suggests that it is important to consider kernel prefetching when comparing the hit ratios of different replacement schemes.

Second, kernel prefetching can significantly affect the effectiveness of clustering of I/O requests and, hence, the reduction in the number of disk requests. For instance, at 64 Mbytes cache size, all schemes besides OPT perform about 609,000 disk requests without prefetching but only about 344,000 requests with prefetching. This is because, without prefetching, *cscope* offers little opportunity for clustering as it reads files in small chunks (mostly, two 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, then it may not result in a decrease in the number of disk requests. This is observed for random-access applications in Section 6.2.2.

Fourth, 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 128 Mbytes. For example, without prefetching, the hit ratios of 2Q and OPT differ by 16 percent. With prefetching, the gap is reduced to less than 5 percent. As another example, without prefetching, LRU-2 and LIRS achieve 51 percent and 40 percent lower hit ratios than 2Q, respectively, but, with prefetching, they achieve 4 percent lower and 1 percent higher hit ratios than 2Q, respectively.

Last, 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, at 128 Mbytes cache size, prefetching reduces the numbers of disk requests for LRU, LRFU, and ARC by 44 percent, 44 percent, and 45 percent, respectively, whereas the corresponding execution times only decrease by 3.3 percent,

TABLE 6
Number and Size of Synchronous and Asynchronous Disk I/Os in *cscope* at 128 Mbytes Cache Size

	No prefetching			Prefetching							Improvement			
				Synchronous			Asynchronous			Avg. Time	Hit	Num.	I/O	Exec.
	Num. I/Os	Size/ I/O	Time/ I/O	Num. I/Os	Size/ I/O	Time/ I/O	Num. I/Os	Size/ I/O	Time/ per I/O	per I/O (ratio over no p.)	ratio (%)	I/Os (%)	time (%)	time (%)
LRU2	609333	1.71	3.37	29965	2.16	4.27	142910	3.71	7.34	6.81 (2.0)	67	72	49	45
LIRS	526926	1.72	3.28	136381	1.83	3.75	22601	6.76	12.98	5.06 (1.5)	61	70	47	43
MQ	607421	1.71	3.40	264251	1.62	3.46	8463	8.07	17.26	3.89 (1.2)	55	55	27	25
ARC	600312	1.71	3.33	259244	2.18	4.07	74042	6.34	11.86	5.80 (1.7)	45	45	4	3
LRFU	607421	1.71	3.40	264485	2.17	4.14	74138	6.35	12.14	5.89 (1.7)	45	44	4	3
LRU	607421	1.71	3.40	264485	2.17	4.14	74138	6.35	12.14	5.89 (1.7)	45	44	4	3
2Q	273568	1.70	3.47	121259	2.12	4.11	33226	6.43	12.46	5.91 (1.7)	20	44	4	3
OPT	190860	1.52	4.09	117984	1.69	4.47	18818	5.06	13.36	5.69 (1.4)	9	28	2	2

All times are in milliseconds.

3.3 percent, and 3.1 percent, respectively. In contrast, prefetching reduces the numbers of disk requests for LIRS, LRU-2, and MQ by 70 percent, 72 percent, and 55 percent, respectively, which causes the execution time to decrease by 43.4 percent, 45.4 percent, and 25.2 percent, respectively.

To understand the above disparate improvement in the execution time for different algorithms, we examined *cscope* at 128 Mbytes for each of the studied algorithm: the behavior of (clustered) synchronous disk requests that are issued without prefetching and synchronous and asynchronous disk requests that are issued when prefetching is enabled. Table 6 shows the numbers for the eight algorithms sorted by the decreasing amount of improvement from prefetching. For each kind of request, we measured the number of requests issued, the average number of blocks accessed per request, and the average time that it took to service the request. The case without prefetching represents the actual on-demand accesses issued by the application. These blocks, if not present in the cache, are always read synchronously. For prefetching, we also show the average time to service a request over both synchronous and asynchronous requests and the ratio of this time to that without prefetching. Additionally, we list the percentage improvement in the hit ratio, the number of I/Os, the I/O time, and the overall execution time.

Table 6 shows that the total number of disk requests (synchronous and asynchronous) decreases by 44 percent to 72 percent, with prefetching for all schemes except OPT. The number of synchronous disk requests is reduced and, instead, a large number of asynchronous disk requests, each for a larger number of blocks, are issued. As expected, with prefetching, the disk requests (synchronous or asynchronous) are for a larger number of blocks than without prefetching. However, with prefetching, the average time to service a disk request also increases by a factor between 1.2 and 2.0 for all schemes. The reason for this increase is explained as follows: The time to service a disk request, in general, is spent on moving the disk arm to the desired location on disk (seek time and rotational delay) and reading the data from that location (transfer time). Since *cscope* has predominantly sequential accesses, the I/O time is mainly the transfer time, with or without prefetching. Since more blocks are read per request with prefetching than without prefetching, the time to service a prefetching request is proportionally longer.

Since *cscope* is an I/O-bound application with an average interval between successive I/Os of only 0.13 ms (Table 4), there is little opportunity for overlapping the asynchronous prefetching request time of over 5 ms with computation. Hence, the total execution time with and without prefetching is dominated by the total I/O times, whether they are synchronous or asynchronous I/O requests. For ARC, LRFU, LRU, and 2Q, the number of disk requests reduced by about 44 percent, but the time to service a request increased by a factor of 1.7. As a result, the overall execution time only improves a little. In contrast, for LRU2, LIRS, and MQ, the disk requests reduced by 72 percent, 70 percent, and 55 percent, respectively, whereas the average time to service a request increased by a factor of 2.0, 1.5, and 1.2, respectively. As a result, the execution time is reduced by 45 percent, 43 percent, and 25 percent, respectively. In summary, with prefetching, although the increase in average time to service a request for these algorithms is comparable, the numbers of disk requests are quite different, resulting in different execution times for different algorithms.

Finally, the above significant difference in the average number of disk requests for the studied algorithms with prefetching can be explained by the difference in the hit ratios of the caching algorithms. *Cscope* is a benchmark with a strong looping reference pattern. Prefetching only reduces the total number of requests by enlarging the size of each request, but the average hit ratio is still correlated with the request number. This is because *cscope* is I/O bound and, hence, asynchronous prefetching cannot be overlapped with computation. Further, each prefetching request is for consecutive blocks and, hence, the subsequent (looping) accesses after a miss, which are to the prefetched blocks, will be hits.

Glimpse. *Glimpse* (Fig. 6) also benefits from prefetching. The curves for the hit ratio shift up by an average of 10 percent for small cache sizes. In contrast, the hit ratios in *cscope* increase by over 40 percent. The smaller increase can be explained as follows: About 72 percent of the files accessed by *glimpse* are one or two blocks long. For these file accesses, there is little benefit from prefetching. In addition, 5 percent 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 *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 2Q and ARC at 128 Mbytes cache size. Without prefetching, 2Q and ARC have hit ratios of 7.2 percent and 11.4 percent, respectively. With prefetching, the hit ratios change to 19.4 percent and 11.5 percent, respectively. Similarly, there is a flip between 2Q and LIRS at cache sizes of 32 and 64 Mbytes.

The improved hit ratio due to prefetching does not translate into a proportionally reduced number of disk requests, as shown in Fig. 6. The hit ratio for LRU-2 at 64 Mbytes cache size increases by 10 percent with prefetching,

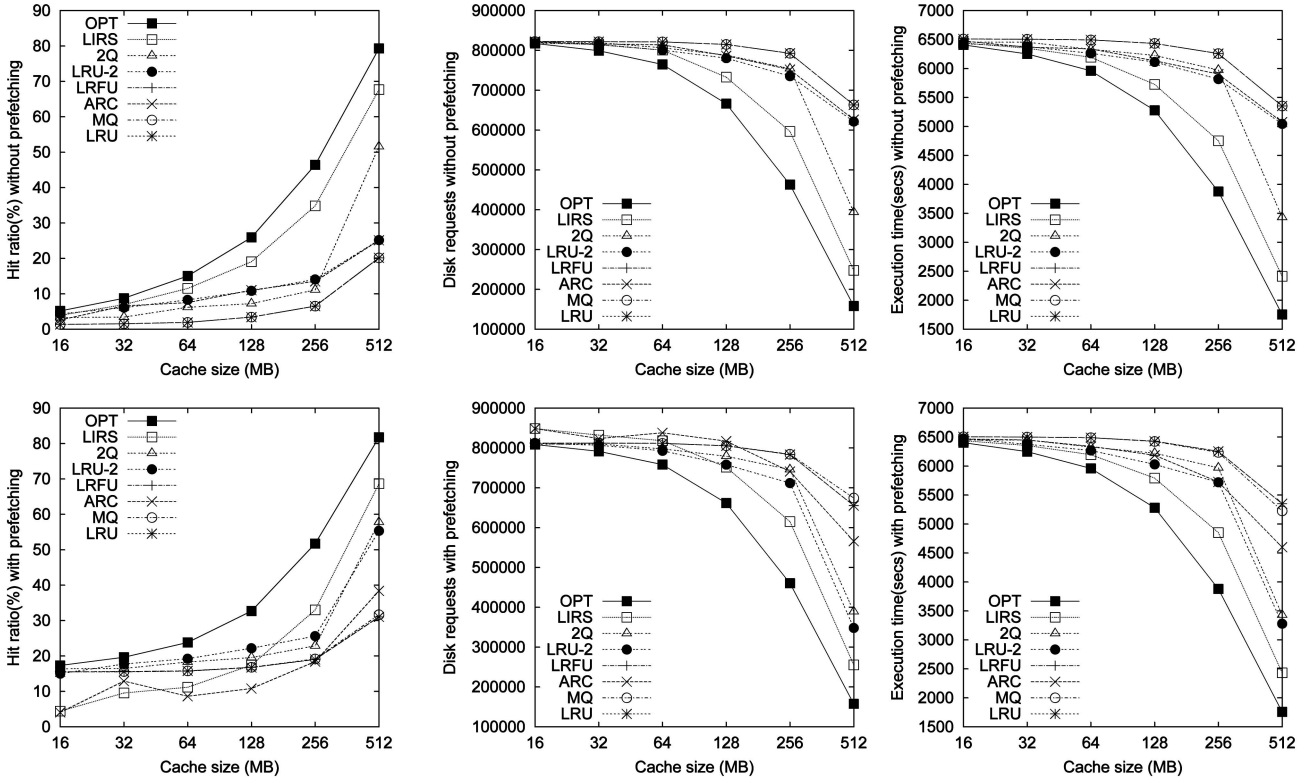


Fig. 6. The hit ratio, number of clustered disk requests, and execution time for *glimpse* under various algorithms.

but the corresponding number of disk requests decreases by less than 2 percent. This can be explained as follows: As discussed earlier, *glimpse* reads either small files for which clustering has little advantage or large files in large chunks where clustering is able to minimize disk requests even without prefetching. Although 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 32 Mbytes cache size with prefetching, the hit ratio for 2Q, LRU-2, LRFU, and MQ increase by 13 percent, 12 percent, 14 percent, and 14 percent, respectively. However, the execution times for these schemes show virtually no improvement (a mere 0.5 percent decrease in time), as observed in the graph. Examining the number of disk requests shows that, for the four schemes, the number of disk requests decreases by less than 1 percent with prefetching, which gives a much better indication of the effect of prefetching on the execution time. As another example, at 128 Mbytes cache size, the hit ratios for 2Q, LRU-2, LRFU, and MQ increase by about 12 percent. The corresponding numbers of disk requests decrease by less than 1 percent, except for LRU-2, for which the number decreases by 2 percent. Hence, although the hit ratio suggests improvement in execution time for all schemes, the number of disk requests suggests a slight improvement in execution time for LRU-2 only. The actual execution time is virtually unchanged, except for LRU-2, where it is 1 percent lower than without prefetching. A similar correspondence between the number of disk requests and the execution time can be observed for other cache sizes.

Gcc. In *gcc* (Fig. 7), the benefit from prefetching is not as pronounced as in *cscope* and *glimpse*. Here, even for OPT, the average improvement in hit ratios is under 0.8 percent, which has a negligible effect on the number of disk requests and the resulting execution time. The main reason for such a small impact on prefetching in *gcc* is that many accesses are to small files, for which there is little opportunity for prefetching. Hence, prefetching is unable to create longer disk requests that can be issued asynchronously and overlapped with the execution time. As a result, all three performance metrics—the hit ratio, the number of disk requests, and the execution time—are almost identical, with or without prefetching for all replacement algorithms.

An exception to this is ARC at 1 Mbyte cache size, where prefetching improves the hit ratio by 16.6 percent. This improvement allows ARC to perform better I/O clustering, resulting in a decrease of 41.1 percent in disk requests, consequently decreasing the execution time by 21.2 percent.

Also notice the flip in the performance of ARC at 1 Mbyte with respect to both LRFU and 2Q. Without prefetching, LRFU and 2Q achieve a hit ratio of 71.8 percent and 73.0 percent, respectively, whereas ARC achieves only 62.0 percent. However, prefetching improves the hit ratios of LRFU and 2Q by mere 1.2 percent and 0.9 percent, respectively, but that of ARC by 16.6 percent (to 78.6 percent), which makes ARC outperform both LRFU and 2Q.

Table 7 shows the breakdown of synchronous and asynchronous disk requests issued with and without prefetching at 1 Mbyte cache size for *gcc*. Unlike in Table 6 for *cscope*, the total number of disk requests does not decrease much with prefetching. Similarly as in *cscope*, the number of synchronous disk requests is reduced and a large number of asynchronous disk requests, each for a larger

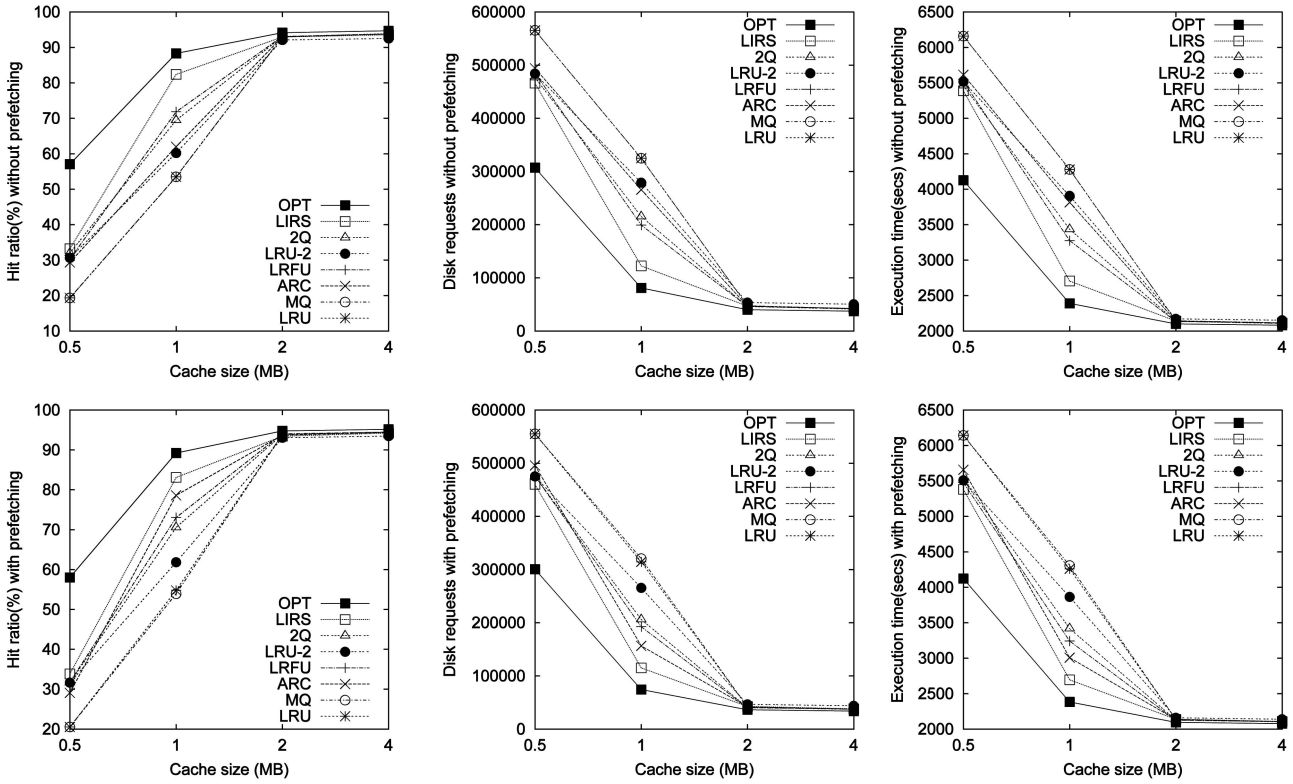


Fig. 7. The hit ratio, number of clustered disk requests, and execution time for *gcc* under various algorithms.

TABLE 7
Number and Size of Synchronous and Asynchronous Disk I/Os in *gcc* at 1 Mbyte Cache Size

	No prefetching			Prefetching							Improvement			
				Synchronous			Asynchronous							
	Num. I/Os	Size/ I/O	Time/ I/O	Num. I/Os	Size/ I/O	Time/ I/O	Num. I/Os	Size/ I/O	Time/ per I/O	Avg. Time per I/O (ratio over no p.)	Hit ratio (%)	Num. I/Os (%)	I/O time (%)	Exec. time (%)
LRU-2	279080	1.38	2.78	263948	1.39	2.80	4670	4.21	8.48	2.90 (1.0)	1	4	2	1
LIRS	128083	1.39	2.68	118429	1.44	2.78	3700	4.20	8.10	2.94 (1.1)	1	5	1	<1
MQ	325025	1.38	2.84	317776	1.39	2.86	3608	4.70	9.67	2.94 (1.0)	<1	1	<1	<1
ARC	265867	1.38	2.77	223262	1.38	2.77	4241	4.19	8.41	2.88 (1.0)	6	14	14	7
LRFU	199279	1.36	2.72	187939	1.39	2.78	4902	3.89	7.78	2.91 (1.1)	1	3	3	1
LRU	325025	1.38	2.83	312709	1.39	2.85	3643	4.70	9.64	2.93 (1.0)	1	3	1	<1
2Q	215509	1.37	2.73	202897	1.38	2.75	3532	4.72	9.41	2.86 (1.0)	1	4	2	1
OPT	81077	1.39	3.72	72144	1.42	3.80	3533	4.19	11.21	4.15 (1.1)	1	7	1	<1

All times are in milliseconds.

number of blocks, are issued. As *gcc* accesses small files, the average size of individual disk requests does not increase by a large factor, as was the case for *cscope*. As a result, the average time to service a disk request only increases by at most a factor of 1.1. Once again, we observe that the time to service a request to more blocks is proportionally longer. Finally, with prefetching, with the exception of ARC, which posts a 7 percent improvement in execution time, other algorithms experience less than 1 percent improvement in execution time.

Viewperf. The behaviors of the different cache replacement algorithms in *viewperf*, as shown in Fig. 8, are similar to those observed in *cscope*. As *viewperf* accesses large files in small chunks, it is able to see the maximum benefit from prefetching. For instance, when prefetching is turned on, the hit ratio improves from 37 percent to 89 percent and the number of disk requests is reduced by a factor of 12.4, on average, across different cache sizes for all algorithms except OPT.

Although all algorithms post a huge gain with prefetching for *viewperf*, we can still observe the different effect prefetching has on different algorithms. At 32 Mbytes cache size without prefetching, the hit ratios of LRU-2, ARC, and 2Q, 33.4 percent, 38.5 percent, and 43.3 percent, are about 5 percent apart. Prefetching reduces this difference to about a mere 1.4 percent, with LRU-2, ARC, and 2Q achieving 87.3 percent, 88.7 percent, and 90.3 percent hit ratios, respectively. This shows that prefetching can take away the benefit that one cache replacement algorithm may have over another algorithm without prefetching.

These observations stress the importance of taking prefetching into account while comparing different algorithms, even if the benefit of prefetching is as small as in *gcc* or as large as in *viewperf*.

Finally, since *viewperf* is a CPU-bound application, the improvement in the hit ratio and the number of disk requests does not translate into any significant reduction in the execution time. For example, for all algorithms, although the hit ratios and the number of disk requests are improved significantly, as seen in the hit ratio and disk request graphs, the execution time is reduced, on average, over all cache sizes across all algorithms except OPT and

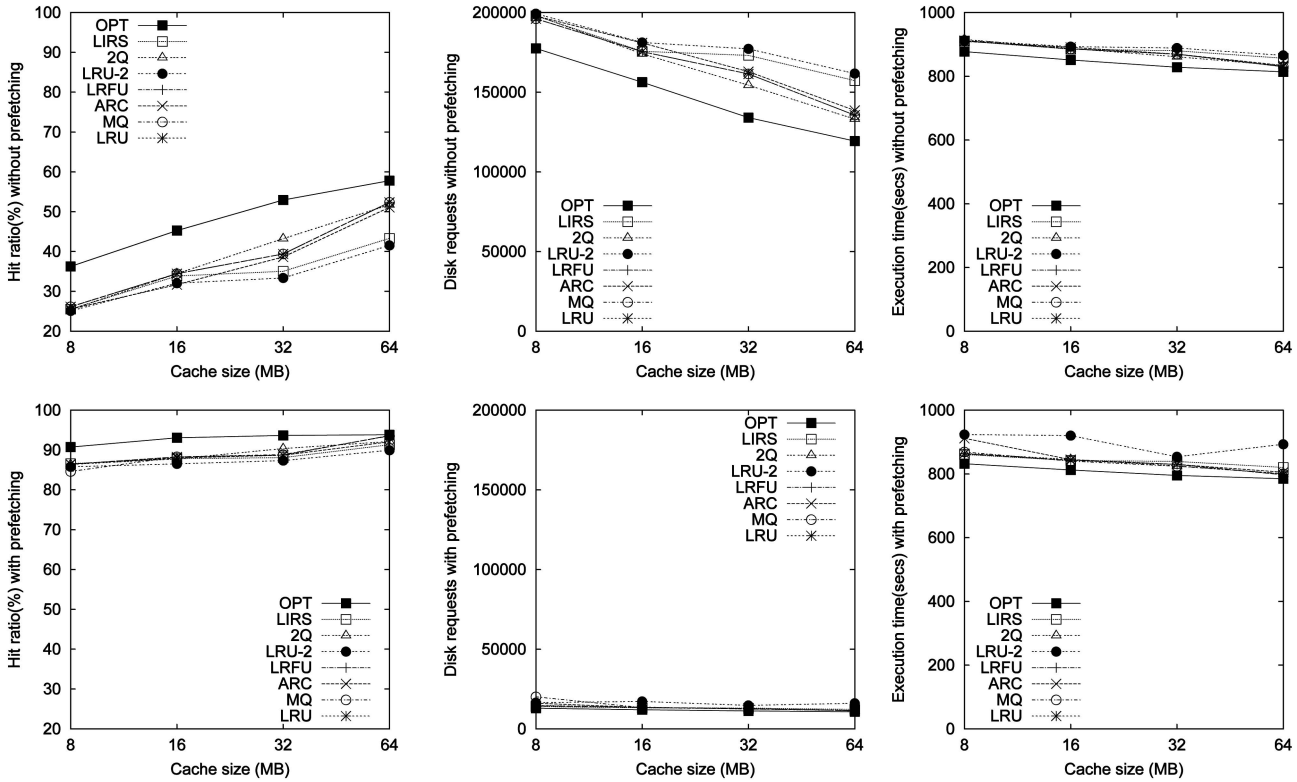


Fig. 8. The hit ratio, number of clustered disk requests, and execution time for *viewperf* under various algorithms.

LRU-2 from 876 to 837 sec, that is, by 4.7 percent only. Although small, this improvement in execution time is due to the large time interval between successive I/O requests, which allows asynchronously issued disk requests to be overlapped with the computation time.

In summary, the trend is not predictable from a simple investigation of effects of prefetching on the hit ratio or the number of disk requests, stressing that it is important to take execution time into account when comparing different algorithms.

6.2.2 Random-Access Applications

The two applications in this group, *tpc-h* and *tpc-r*, exhibit predominantly random accesses. As expected (shown in Figs. 9 and 10), 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.

We investigated *tpc-h* further by examining the breakdown of the total number of disk requests into synchronous and asynchronous disk requests with prefetching and by comparing them to the number of synchronous requests without prefetching. We found that the number of synchronous disk requests stays almost the same with and without prefetching. However, with prefetching, the synchronous disk requests also read an extra prefetched block. Furthermore, an almost-equal number of asynchronous disk requests as the synchronous requests are issued. Each asynchronous disk request was found to prefetch two to four blocks. Hence, although the disk requests double, the number of blocks accessed from the disk increases by a factor of 5. This, in turn, results in the execution time of *tpc-h* to be almost doubled with prefetching.

Although, with prefetching, both *tpc-h* and *tpc-r* 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 and the number of disk requests gives a much better indication of the relative performance of different replacement algorithms.

The above results show 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 by 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 Linux.

6.2.3 Concurrent Applications

The applications in this group contain accesses that are a mix of sequential and random accesses. *Multi1* and *multi2* contain more sequential accesses than *multi3* and are dominated by *cscope*. The hit ratios and disk requests for *multi1* (Fig. 11) with or without prefetching exhibit similar behavior as that for *cscope*. Observe the different effects of prefetching at 128 Mbytes cache size on the hit ratios of 2Q, LIRS, LRU-2, MQ, and ARC, which increase by 18.1 percent, 53.8 percent, 59.6 percent, 48.1 percent, and 39.9 percent, respectively. The different effects on hit ratios cause LIRS to perform 1 percent better than 2Q, whereas it did 35 percent worse than 2Q without prefetching. Similarly, LRU-2 at 128 Mbytes achieves 17.7 percent higher hit ratio than ARC with prefetching but 2.0 percent lower than ARC without prefetching.

The improvement in hit ratios causes the number of disk requests with prefetching to be almost half that without prefetching (average reduction of 47.8 percent). However, this does not translate into similar savings in the execution time. Only for MQ, LIRS, and LRU-2 at 128 Mbytes cache size does the improvement in hit ratio and number of disk requests cause the execution time to decrease by 22.4 percent,

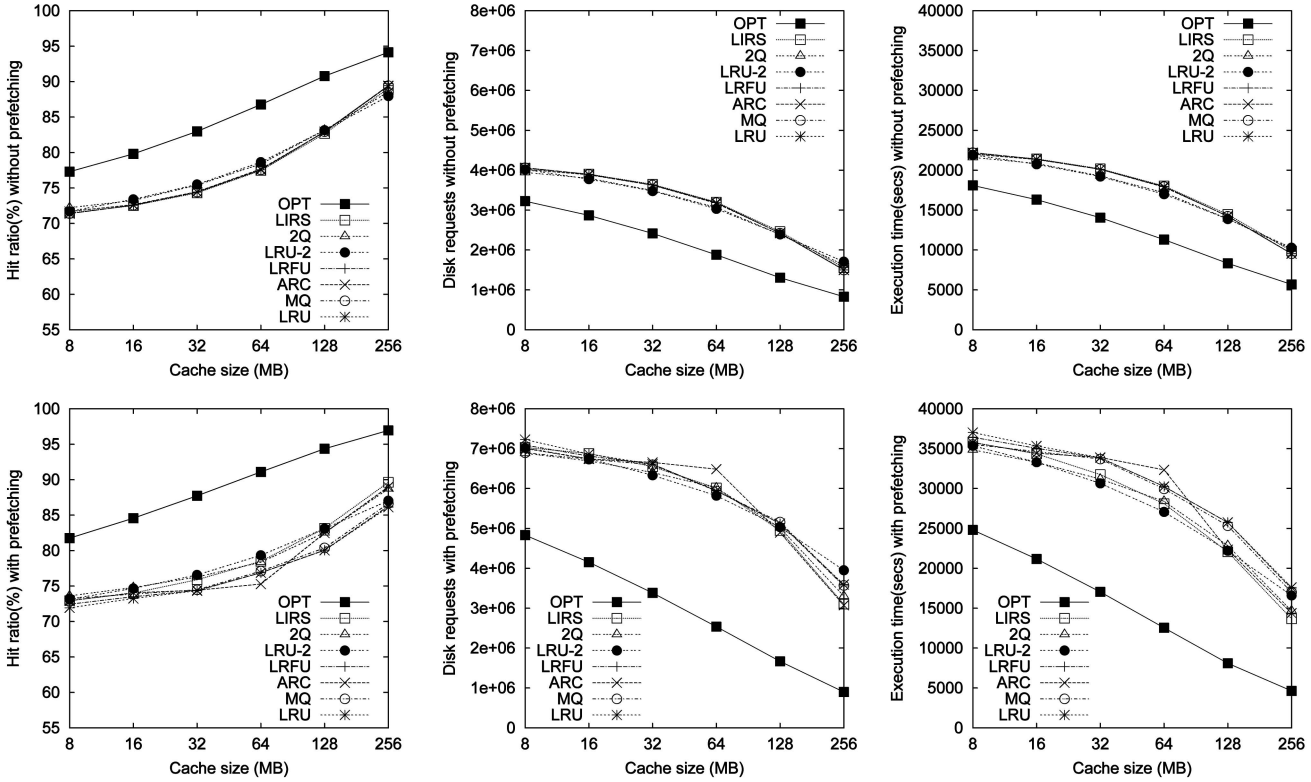


Fig. 9. The hit ratio, number of clustered disk requests, and execution time for *tpc-h* under various algorithms.

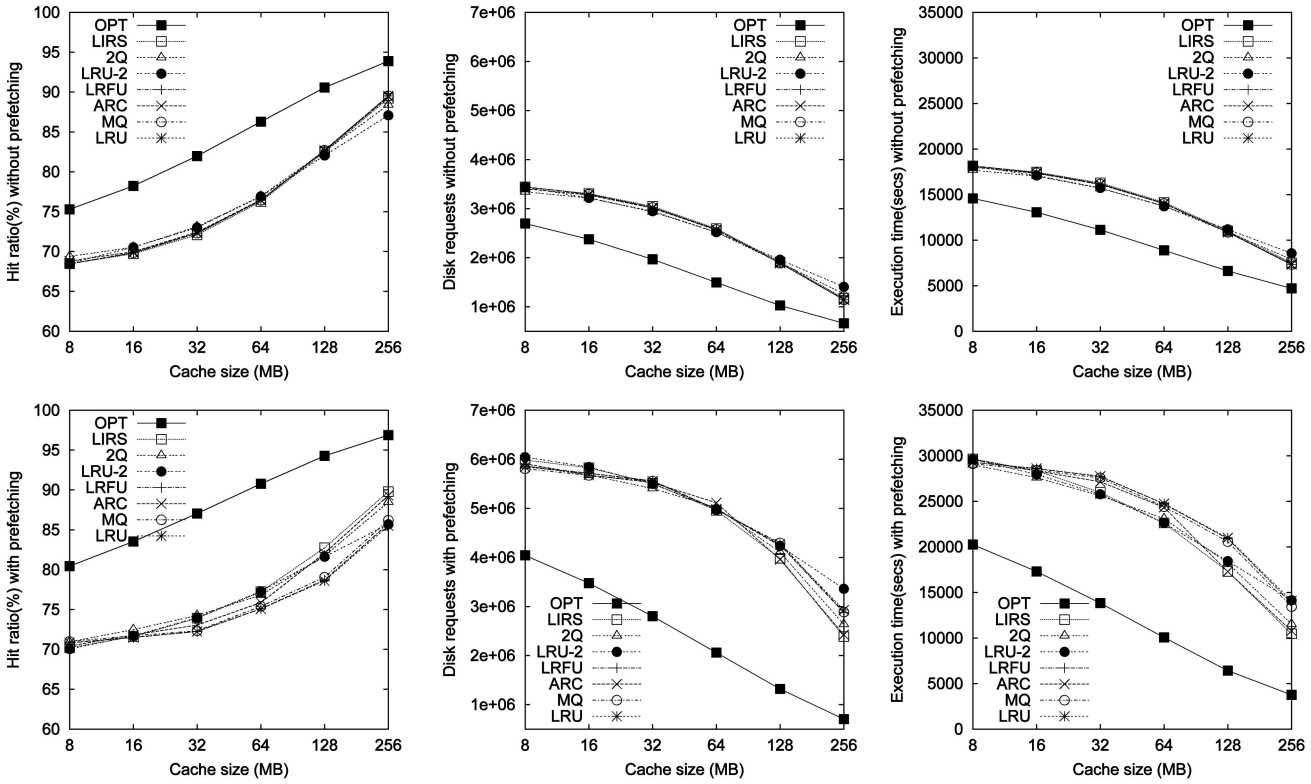


Fig. 10. The hit ratio, number of clustered disk requests, and execution time for *tpc-r* under various algorithms.

38.6 percent, and 41.0 percent, respectively. For all other algorithms at 128 Mbytes, the average reduction in the number of disk requests by 39.3 percent translates into a mere average improvement of 2.6 percent in the execution time.

Multi2 (Fig. 12) behaves similarly to *multi1* due to the predominant sequential accesses of *cscope*. However, due to the presence of sequential accesses from *viewperf*, prefetching

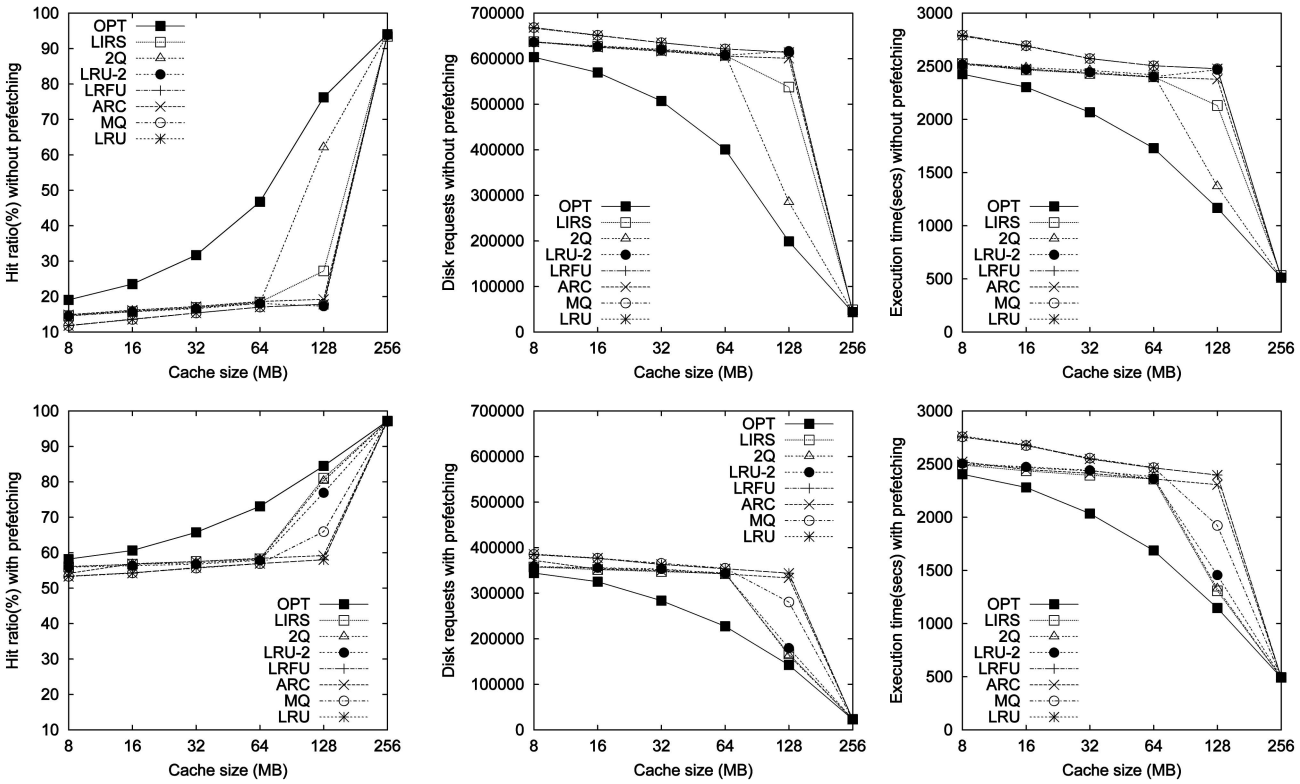


Fig. 11. The hit ratio, number of clustered disk requests, and execution time for *multi1* under various algorithms.

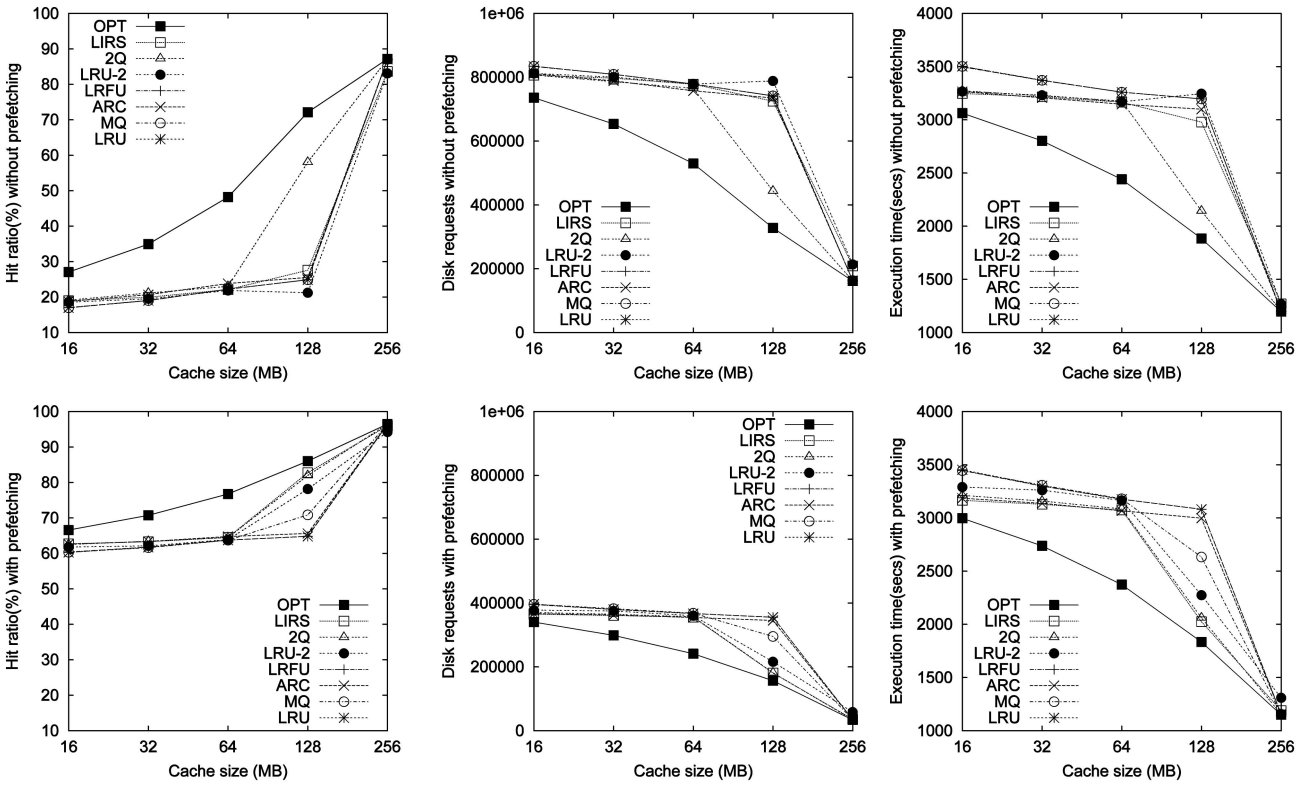


Fig. 12. The hit ratio, number of clustered disk requests, and execution time for *multi2* under various algorithms.

causes the average hit ratios for the different algorithms to increase from about 20 percent to 60 percent.

The varying effect of prefetching on the different algorithms can also be observed. For instance, at 128 Mbytes cache size, without prefetching, LIRS achieved 30 percent lower hit ratio than 2Q. Prefetching changes this behavior and causes LIRS to achieve 1 percent higher hit ratio than 2Q.

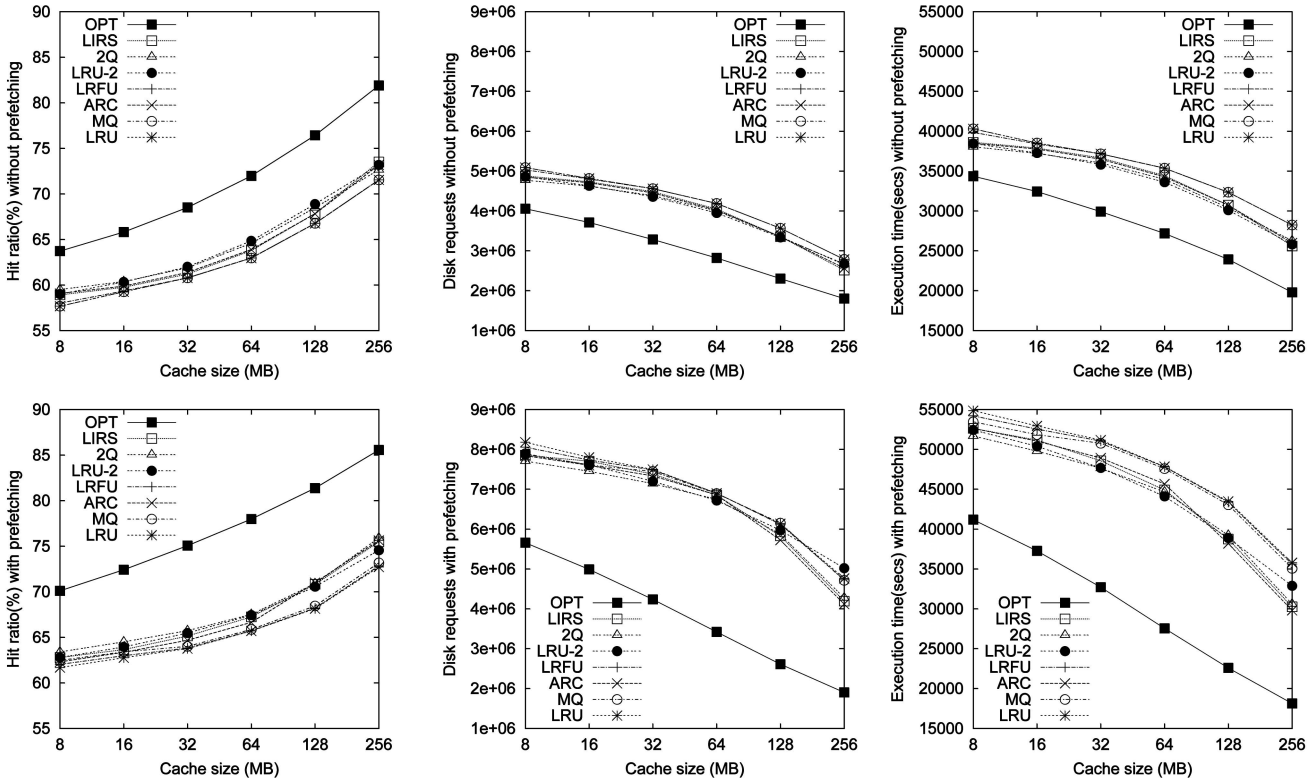


Fig. 13. The hit ratio, number of clustered disk requests, and execution time for *multi3* under various algorithms.

Another interesting effect can be noticed for LRU-2 at 128 Mbytes cache size. Without prefetching, LRU-2 achieves 4.3 percent and 3.7 percent lower hit ratios than ARC and MQ, respectively, but, with prefetching, LRU-2 achieves 12.5 percent and 7.3 percent higher hit ratios than ARC and MQ, respectively. Interestingly, ARC and MQ also flip behavior with respect to each other at 128 Mbytes cache size, with ARC doing 0.7 percent better than MQ without prefetching and MQ doing 5.2 percent better than ARC with prefetching.

The trend in hit ratios for *multi2* is also observed in the disk requests as prefetching provides a better opportunity for clustering predominantly sequential accesses in the mix of applications. However, except LIRS, LRU-2, and MQ, the improvement in disk requests is not mirrored in the execution time. This is because 1) the presence of the CPU-bound *viewperf* in the mix lowers the improvement on the execution time for all algorithms and 2) a significant reduction in disk requests is needed to achieve reduced execution time for *cscope*, as discussed in Section 6.2.1.

Multi3 (Fig. 13) has a large number of random accesses due to *tpc-h* and, therefore, its performance curves look similar to those of *tpc-h*. Similar observations as for *tpc-h* can be made from the graphs for *multi3*. For instance, the improvement in hit ratios is very small, but prefetching causes data that are not used by the application to be read from the disk, resulting in the number of disk requests increasing, on average, by a factor of 1.7. Consequently, the execution time, on average, increases by a factor of 1.3.

The observations made for the concurrent applications stress the same inferences drawn from the study of individual applications: Prefetching affects the hit ratio, number of disk requests, and execution time for different algorithms differently and, moreover, kernel prefetching degrades the performance of random-access applications.

6.2.4 Summary

Table 8 categorizes the kinds of impact of prefetching on the hit ratio, number of disk requests, and execution time of different algorithms, along with example scenarios where these impacts can be observed.

First, prefetching has an impact on the hit ratio for all applications and for all algorithms studied. However, this impact is different for different algorithms, benefiting some algorithms more than others. As a result, the relative performance of two algorithms can flip with respect to each other, that is, an algorithm that achieves a higher hit ratio than another algorithm without prefetching can end up with a lower hit ratio than the other algorithm with prefetching. Therefore, it is critical to take kernel prefetching into account while comparing the hit ratios of different cache replacement algorithms.

Second, the effect of prefetching on the hit ratio may not provide an indication of how the disk requests will be affected. Prefetching increases the opportunity for I/O request clustering for sequential accesses, which results in a reduced number of disk requests. However, for random accesses, prefetching can cause wrong data to be read, resulting in an increased number of disk requests. Therefore, the number of disk requests should be carefully studied when comparing different cache replacement algorithms.

Finally, the effect of prefetching on disk requests may not provide a good indication of the execution time. In addition to the reduction of the number of disk requests, the improvement on the final execution time of an application is also affected by the amount of the computation time between successive I/O requests and, hence, the opportunities of overlapping asynchronous disk requests (from asynchronous prefetching) with computation. For example, *cscope*, which is I/O bound, does not provide such an opportunity,

TABLE 8
Summary of Observations Made during Evaluation of Various Schemes

Scenario	Observation	Examples
1	Prefetching improves hit ratio for all applications	*all applications* – all algorithms
2a	Hit ratio improvements differ for different algorithms	*cscope* – LRU and LRU-2; *glimpse* – LIRS and ARC; *multi2* – LIRS and LRU-2
2b	Hit ratios may flip	*cscope* – LIRS and 2Q at 128 MB; *glimpse* – ARC and 2Q at 128 MB, LIRS and LRU-2 at 128 MB; *gcc* – ARC and LRFU at 1 MB, ARC and 2Q at 1 MB; *multi2* – MQ and LRU-2 at 128 MB, LIRS and 2Q at 128 MB
3a	Prefetching reduces the number of disk requests	*cscope, viewperf, multi1, multi2* – all algorithms
3b	Prefetching increases the number of disk requests	*tpc-h, tpc-r* – all algorithms
4a	Disk requests improve with hit ratios	*cscope, viewperf, multi3* – all algorithms; *glimpse* – LIRS; *gcc* – ARC
4b	Disk requests do not improve with hit ratios	*glimpse* – MQ, LRU; *gcc* – 2Q, LRU-2, LIRS
5a	Prefetching improves execution time	*cscope* – LIRS, MQ, LRU-2 at 128MB; *glimpse* – LRU-2; *viewperf, tpc-h, tpc-r* – all algorithms;
5b	Prefetching does not improve execution time	*cscope* – 2Q, LRU, ARC; *gcc* – all except ARC
6a	Reduction in disk requests improves execution time	*cscope* – LIRS, MQ, LRU-2 at 128MB; *glimpse* – LRU-2; *gcc* – ARC; *viewperf* – all algorithms
6b	Reduction in disk requests does not improve execution time	*cscope* – 2Q, LRU, ARC; *tpc-h, tpc-r* – all algorithms

whereas *viewperf*, which is CPU bound, does. Hence, the final execution time is the only definitive measure of application performance and it should be studied while comparing different cache replacement algorithms.

7 Related Work

7.1 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. The Sequential algorithm (SEQ) [15] detects sequential page fault patterns and applies the MRU policy to those pages. For other pages, the LRU replacement is applied. However, SEQ does not distinguish sequential and looping references. Early Eviction LRU (EELRU) [36] detects looping references by examining aggregate recency distribution of referenced pages and changes the eviction point by using a simple cost-benefit analysis. DEtection-based Adaptive Replacement (DEAR) [12], [13], Unified Buffer Management (UBM) [24], and Program-Counter-based Classification (PCC) [16] 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 limitations, we did not evaluate pattern-based algorithms in this paper.

Hint-based algorithms. In application-controlled cache management [9], [34], the programmer is responsible for inserting hints into the application that indicate to the 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. To eliminate the burden on the programmers, compiler-inserted hints are proposed [6]. These methods provide the benefits of user-inserted hints for existing applications by simply recompiling the applications with the proposed compiler. However, more complicated access patterns or input-dependent patterns may be difficult for the compiler to characterize.

7.2 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 [30] or even explicitly prescribed by a binary rewriter [11] 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 [18], [42]. Another approach uses time series modeling [40] to predict temporal access patterns and issue prefetches during computation intervals. Prefetch algorithms tailored for parallel I/O systems have also been studied [2], [23], [25].

7.3 Integrated Prefetching and Caching

In [8], 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, [2], [9], [23], [25], [26], [34], and [39], which are either offline or based on hints of I/O access patterns.

Although an integrated prefetching and caching design is not supported in any modern OS, all modern operating systems implement some form of kernel prefetching in their file systems in 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 [3], [15], [21], [22], [27], [29], [32], [35], [36].

In [38], Belady's algorithm [4] 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.

Most recently, DUal LOcality (DULO) [20] and Wise Ordering of Writes (WOW) [14] study the combined effect

of temporal and spatial locality on buffer caching. DULO improves the buffer cache performance by maintaining a separate buffer for prefetched blocks, whereas WOW exploits similar principles specifically for write caches. Both works are good examples that reaffirm the importance of taking into account the effects of kernel prefetching in designing better cache replacement algorithms, as first pointed out by our work [7].

8 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 representative replacement algorithms.

Our study shows that such kernel prefetching can have a profound 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*, and *viewperf*) or may not (*glimpse*, *tpc-h*, and *tpc-r*) translate into a similar difference in the number of disk requests. The difference in the number of disk requests may (*gcc*, *glimpse*, *tpc-h*, and *tpc-r*) or may not (*cscope* and *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*, and *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 significantly 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 a standard file system such as in Linux. These results clearly demonstrate the importance for buffer caching research to take file system prefetching into consideration.

Our study also raises several questions on the buffer cache replacement algorithm design. How can we adapt an existing buffer cache replacement algorithm to exploit kernel prefetching? How can we 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? How should the Linux or BSD kernel prefetching be changed to avoid counterproductive prefetches for random-access applications? We are studying these questions as part of our future work.

Acknowledgments

This work is supported in part by US National Science Foundation CAREER Award ACI-0238379. The authors thank the anonymous reviewers, whose comments have greatly improved the content and presentation of this paper. They thank Zheng Zhang for his help with distributing and maintaining the AccuSim simulator.

References

[1] AccuSim, http://shay.ecn.purdue.edu/~dsnl/accusim/, Nov. 2005.

[2] S. Albers and M. Büttner, "Integrated Prefetching and Caching in Single and Parallel Disk Systems," *Proc. 15th Ann. ACM Symp. Parallel Algorithms and Architectures,* June 2003.

[3] S. Bansal and D.S. Modha, "CAR: Clock with Adaptive Replacement," *Proc. USENIX Conf. File and Storage Technologies,* Mar. 2004.

[4] L.A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems J.,* vol. 5, no. 2, pp. 78-101, 1966.

[5] D.P. Bovet and M. Cesati, *Understanding the Linux Kernel,* second ed. O'Reilly and Associates, 2003.

[6] A.D. Brown, T.C. Mowry, and O. Krieger, "Compiler-Based I/O Prefetching for Out-of-Core Applications," *ACM Trans. Computer Systems,* vol. 19, no. 2, pp. 111-170, 2001.

[7] A.R. Butt, C. Gniady, and Y.C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '05),* June 2005.

[8] P. Cao, E. Felten, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '95),* May 1995.

[9] 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 Trans. Computer Systems,* vol. 14, no. 4, pp. 311-343, 1996.

[10] R.W. Carr and J.L. Hennessy, "WSCLOCK—A Simple and Effective Algorithm for Virtual Memory Management," *Proc. Eighth Symp. Operating Systems Principles,* Dec. 1981.

[11] F.W. Chang and G.A. Gibson, "Automatic I/O Hint Generation through Speculative Execution," *Proc. Third Usenix Symp. Operating Systems Design and Implementation,* Feb. 1999.

[12] J. Choi, S.H. Noh, S.L. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," *Proc. 1999 USENIX Ann. Technical Conf.,* June 1999.

[13] 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," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '00),* June 2000.

[14] B.S. Gill and D.S. Modha, "WOW: Wise Ordering for Writes—Combining Spatial and Temporal Locality in Non-Volatile Caches," *Proc. USENIX Conf. File and Storage Technologies,* Dec. 2005.

[15] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '97),* June 1997.

[16] C. Gniady, A.R. Butt, and Y.C. Hu, "Program Counter Based Pattern Classification in Buffer Caching," *Proc. Sixth Symp. Operating Systems Design and Implementation,* Dec. 2004.

[17] B.L.W. Gregory, R. Ganger, and Y.N. Patt, "The Disksim Simulation Environment," Technical Report CSE-TR-358-98, Dept. of Electrical Eng. and Computer Science, Univ. of Michigan, Feb. 1998.

[18] J. Griffioen and R. Appleton, "Performance Measurements of Automatic Prefetching," *Proc. Parallel and Distributed Computing Systems,* Sept. 1995.

[19] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement," *Proc. USENIX Ann. Technical Conf.,* Apr. 2005.

[20] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality," *Proc. USENIX Conf. File and Storage Technologies,* Dec. 2005.

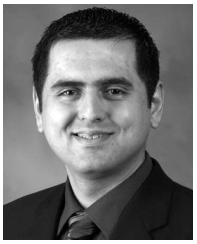
[21] S. Jiang and X. Zhang, "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '02),* June 2002.

[22] T. Johnson and D. Shasha, "2Q: A Low-Overhead High-Performance Buffer Management Replacement Algorithm," *Proc. 20th Int'l Conf. Very Large Data Bases,* Jan. 1994.

[23] M. Kallahalla and P.J. Varman, "Optimal Prefetching and Caching for Parallel I/O Sytems," *Proc. 13th Ann. ACM Symp. Parallel Algorithms and Architectures,* July 2001.

[24] 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," *Proc. Fourth USENIX Symp. Operating System Design and Implementation,* Oct. 2000.

[25] T. Kimbrel and A.R. Karlin, "Near-Optimal Parallel Prefetching and Caching," *SIAM J. Computing,* vol. 29, no. 4, pp. 1051-1082, 2000.
[26] T. Kimbrel, A. Tomkins, 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," *Proc. Symp. Operating Systems Design and Implementation,* Oct. 1996.
[27] 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 Trans. Computers,* vol. 50, no. 12, pp. 1352-1360, Dec. 2001.
[28] U. Manber and S. Wu, "GLIMPSE: A Tool to Search through Entire File Systems," *Proc. USENIX Winter 1994 Technical Conf.,* Jan. 1994.
[29] N. Megiddo and D.S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," *Proc. Second USENIX Conf. File and Storage Technologies,* Mar. 2003.
[30] T.C. Mowry, A.K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," *Proc. Second USENIX Symp. Operating Systems Design and Implementation,* 1996.
[31] MySQL, http://www.mysql.com/, 2005.
[32] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM Int'l Conf. Management of Data (Sigmod '93),* May 1993.
[33] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "An Optimality Proof of the LRU-K Page Replacement Algorithm," *J. ACM,* vol. 46, no. 1, pp. 92-112, 1999.
[34] R.H. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," *Proc. 15th ACM Symp. Operating Systems Principles,* Dec. 1995.
[35] J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," *Proc. ACM Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '90),* May 1990.
[36] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '99),* May 1999.
[37] J. Steffen, "Interactive Examination of a C Program with Cscope," *Proc. USENIX Winter 1985 Technical Conf.,* Jan. 1985.
[38] O. Temam, "An Algorithm for Optimally Exploiting Spatial and Temporal Locality in Upper Memory Levels," *IEEE Trans. Computers,* vol. 48, no. 2, pp. 150-158, Feb. 1999.
[39] A. Tomkins, R.H. Patterson, and G.A. Gibson, "Informed Multi-Process Prefetching and Caching," *Proc. ACM Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '97),* June 1997.
[40] N. Tran and D.A. Reed, "Arima Time Series Modeling and Forecasting for Adaptive I/O Prefetching," *Proc. 15th Int'l Conf. Supercomputing,* June 2001.
[41] Transaction Processing Council, http://www.tpc.org, 2005.
[42] V. Vellanki and A.L. Chervenak, "A Cost-Benefit Scheme for High-Performance Predictive Prefetching," *Proc. 1999 ACM/IEEE Conf. Supercomputing,* Nov. 1999.
[43] Y. Zhou, P.M. Chen, and K. Li, "The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches," *Proc. 2001 USENIX Ann. Technical Conf.,* June 2001.



Ali R. Butt received the BSc (Hons.) degree in electrical engineering from the University of Engineering and Technology Lahore, Pakistan, in 2000 and the PhD degree in electrical and computer engineering from Purdue University in 2006. At Purdue, he also served as the president of the Electrical and Computer Engineering Graduate Student Association for 2003 and 2004. He is an assistant professor of computer science at Virginia Polytechnic Institute and State University. His research interests lie broadly in distributed systems and operating systems. In particular, he has explored distributed resource sharing systems spanning multiple administrative domains, applications of peer-to-peer overlay networking to resource discovery and self-organization, and techniques for ensuring fairness in the sharing of such resources. His research in operating systems focuses on techniques for improving the efficiency of modern file systems via innovative buffer cache management. He is a member of USENIX, the ACM, and the IEEE.

Chris Gniady received the BS degree in electrical and computer engineering from Purdue University in 1997 and the PhD degree from the School of Electrical and Computer Engineering, Purdue University, in 2005. He is an assistant professor of computer science at the University of Arizona. His research focuses on providing personalized computing by adapting software and hardware to the user's behavior. He uses instruction-based prediction and user-behavior monitoring to optimize energy and performance in computer systems. He is a member of the IEEE.

Y. Charlie Hu received the MS and MPhil degrees from Yale University in 1992 and the PhD degree in computer science from Harvard University in 1997. He is an assistant professor of electrical and computer engineering and computer science at Purdue University. From 1997 to 2001, he was a research scientist at Rice University. His research interests include operating systems, distributed systems, networking, and parallel computing. He has published more than 85 papers in these areas. He received the Honda Initiation Grant Award in 2002 and the US National Science Foundation CAREER Award in 2003. He served as a technical program committee (TPC) vice chair for the 2004 International Conference on Parallel Processing and the IEEE International Conference on Distributed Computing Systems (ICDCS) 2007. He was also a cofounder and TPC cochair for the International Workshop on Mobile Peer-to-Peer Computing. He is a member of Usenix, the ACM, and the IEEE.