DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch

Xiaoning Ding[1], Song Jiang[2], Feng Chen[1], Kei Davis[3], and Xiaodong Zhang[1]

[1] *CSE Department*
Ohio State University
Columbus, OH 43210, USA
{dingxn,fchen,zhang}@cse.ohio-state.edu

[2] *ECE Department*
Wayne State University
Detroit, MI 48202, USA
sjiang@eng.wayne.edu

[3] *CCS-3 Division*
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
kei.davis@lanl.gov

Abstract

Current disk prefetch policies in major operating systems track access patterns at the level of the file abstraction. While this is useful for exploiting application-level access patterns, file-level prefetching cannot realize the full performance improvements achievable by prefetching. There are two reasons for this. First, certain prefetch opportunities can only be detected by knowing the data layout on disk, such as the contiguous layout of file metadata or data from multiple files. Second, non-sequential access of disk data (requiring disk head movement) is much slower than sequential access, and the penalty for mis-prefetching a 'random' block, relative to that of a sequential block, is correspondingly more costly.

To overcome the inherent limitations of prefetching at the logical file level, we propose to perform prefetching directly at the level of disk layout, and in a portable way. Our technique, called *DiskSeen*, is intended to be supplementary to, and to work synergistically with, file-level prefetch policies, if present. *DiskSeen* tracks the locations and access times of disk blocks, and based on analysis of their temporal and spatial relationships, seeks to improve the sequentiality of disk accesses and overall prefetching performance.

Our implementation of the DiskSeen scheme in the Linux 2.6 kernel shows that it can significantly improve the effectiveness of prefetching, reducing execution times by 20%-53% for micro-benchmarks and real applications such as *grep*, *CVS*, and *TPC-H*.

1 Introduction

As the speed differential between processor and disk continues to widen, the effect of disk performance on the performance of data-intensive applications is increasingly great. Prefetching—speculative reading from disk based on some prediction of future requests—is a fundamental technique for improving effective disk performance. Prefetch policies attempt to predict, based on analysis of disk requests, the optimal stream of blocks to prefetch to minimize disk service time as seen by the application workload. Prefetching improves disk performance by accurately predicting disk requests in advance of the actual requests and exploiting hardware concurrency to hide disk access time behind useful computation.

Two factors demand that prefetch policies be concerned with not just accuracy of prediction, but also actual time cost of individual accesses. First, a hard disk is a non-uniform-access device for which accessing sequential positions without disk head movement is at least an order of magnitude faster than random access. Second, an important observation is that as an application load becomes increasingly I/O bound, such that disk accesses can be decreasingly hidden behind computation, the importance of sequential prefetching increases relative to the importance of prefetching random (randomly located) blocks. This is a consequence of the speculative nature of prefetching and the relative penalties for incorrectly prefetching a sequential block versus a random block. This may explain why, despite considerable work on sophisticated prefetch algorithms (Section 5), general-purpose operating systems still provide only sequential prefetching or straightforward variants thereof. Another possible reason is that other proposed schemes have been deemed either too difficult to implement relative to their probable benefits, or too likely to hurt performance in some common scenarios. To be more relevant to current practice, the following discussion is specific to prefetch policies used in general-purpose operating systems.

Existing prefetch policies usually detect access patterns and issue prefetch requests at the logical file level. This fits with the fact that applications make I/O requests based on logical file structure, so their discernible access patterns will be directly in terms of logical file structure. However, because disk data layout information is not exploited by these policies, they do not have the knowledge of where the next prefetched block would be relative to the currently fetched block to estimate prefetching cost. Thus, their measure of prefetching effectiveness, which is usually used as a feedback to adjust prefetching behavior, is in terms of the number of mis-prefetched blocks rather than a more relevant metric, the penalty of mis-prefetching. Disk layout information is not used until the requests are processed by the lower-level disk scheduler where requests are sorted and merged, based on disk placement, into a dispatching queue using algorithms such as SSTF or C-SCAN to maximize disk throughput.

We contend that file-level prefetching has both practical and inherent limitations, and that I/O performance can be significantly improved by prefetching based on disk data layout information. This disk-level prefetching is intended to be supplementary to, and synergistic with, any file-level prefetching. Following we summarize the limitations of file-level prefetching.

Sequentiality at the file abstraction may not translate to sequentiality on disk. While file systems typically seek to dynamically maintain a correspondence between logical file sequentiality and disk sequentiality, as the file system ages (e.g. in the case of Microsoft's NTFS) or becomes full (e.g. Linux Ext2) this correspondence may deteriorate. This worsens the penalty for mis-prediction.

The file abstraction is not a convenient level for recording deep access history information. This is exacerbated by the issue of maintaining history information across file closing and re-opening and other operations by the operating system. As a consequence, current prefetch schemes maintain shallow history information and so must prefetch conservatively [21].[1] A further consequence is that sequential access of a short file will not trigger the prefetch mechanism.

Inter-file sequentiality is not exploited. In a general-purpose OS, file-level prefetching usually takes place within individual files, which precludes detection of sequential access across contiguous files.

Finally, blocks containing file system metadata cannot be prefetched. Metadata blocks, such as inodes, are not in files, and so cannot be prefetched. Metadata blocks may need to be visited frequently when a large number of small files are accessed.

In response, we propose a disk-level prefetching scheme, *DiskSeen*, in which current and historical information is used to achieve *efficient* and *accurate* prefetching. While caches in hard drives are used for prefetching blocks directly ahead of the block being requested, this prefetching is usually carried out on each individual track and does not take into account the relatively long-term temporal and spatial locality of blocks across the entire disk working set. The performance potential of the disk's prefetching is further constrained because it cannot communicate with the operating system to determine which blocks are already cached there; this is intrinsic to the disk interface. The performance improvements we demonstrate are in addition to those provided by existing file-level and disk-level prefetching.

We first describe an efficient method for tracking disk block accesses and analyzing associations between blocks (Section 2). We then show how to efficiently detect sequences of accesses of disk blocks and to appropriately initiate prefetching at the disk level. Further aided by access history information, we show how to detect complicated pseudo-sequences with high accuracy (Section 3). We show that an implementation of these algorithms—collectively DiskSeen—in the current Linux kernel can yield significant performance improvements on representative applications (Section 4).

2 Tracking Disk Accesses

There are two questions to answer before describing DiskSeen. The first is what information about disk locations and access times should be used by the prefetch policy. Because the disk-specific information is exposed using the unit of disk blocks, the second question is how to efficiently manage the potentially large amount of information. In this section, we answer these two questions.

2.1 Exposing Disk Layout Information

Generally, the more specific the information available for a particular disk, the more accurate an estimate a disk-aware policy can make about access costs. For example, knowing that blocks span a track boundary informs that access would incur the track crossing penalty [24]. As another example, knowing that a set of non-contiguous blocks have some spatial locality, the scheduler could infer that access of these blocks would incur the cost of semi-sequential access, intermediate between sequential and random access [26]. However, detailed disk performance characterization requires knowledge of physical disk geometry, which is not disclosed by disk manufacturers, and its extraction, either interrogative or empirical, is a challenging task [30, 23]. Different extraction approaches may have different accuracy and work only with certain types of disk drives (such as SCSI disks).

An interface abstraction that disk devices commonly provide is logical disk geometry, which is a linearized data layout and represented by a sequence $[0, 1, 2, ..., n]$ of *logical block numbers* (LBNs). Disk manufacturers usually make every effort to ensure that accessing blocks with consecutive LBNs has performance close to that of accessing contiguous blocks on disk by carefully mapping logical blocks to physical locations with minimal disk head positioning cost [26]. Though the LBN does not disclose precise disk-specific information, we use it to represent disk layout for designing a disk-level prefetch policy because of its standardized availability and portability across various computing platforms. In this paper, we will show that *exposing*[2] this logical disk layout is sufficient to demonstrate that incorporating disk-side information with application-side information into prefetch policies can yield significant performance benefits worthy of implementation.

2.2 The Block Table for Managing LBNs

Currently LBNs are only used to identify locations of disk blocks for transfer between memory and disk. Here we track the access times of recently touched disk blocks via their LBNs and analyze the associations of access times among adjacent LBNs. The data structure holding this information must support efficient access of block entries and their neighboring blocks via LBNs, and efficient addition and removal of block entries.

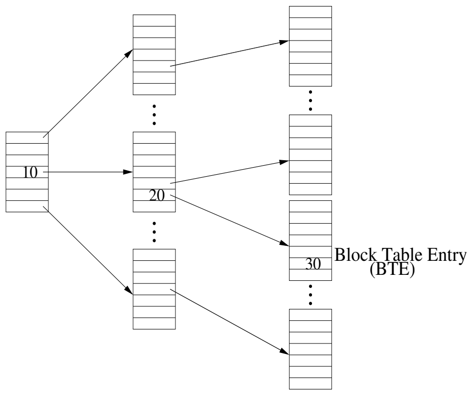

Figure 1: **Block table**. There are three levels in the example block table: two directory levels and one leaf level. The table entries at differing levels are fit into separate memory pages. An entry at the leaf level is called a block table entry (BTE). If one page can hold 512 entries, the access time of a block with LBN 2,631,710 ($10\times 512^2 + 20\times 512 + 30$) is recorded at the BTE entry labeled 30, which can be efficiently reached via directory-level entries labeled 10 and 20.

The block table, which has been used in the DULO scheme for identifying block sequences [12], is inspired by the multi-level page table used for a process's memory address translation, which is used in almost all operating systems. As shown in Figure 1, an LBN is broken into multiple segments, each of which is used as an offset in the corresponding level of the table. In the DULO scheme, bank clock time, or block sequencing time, is recorded at the leaf level (i.e., block table entry (BTE)) to approximate block access time. In DiskSeen, a finer block access timing mechanism is used. We refer to the entire sequence of accessed disk blocks as the *block access stream*. The n^{th} block in the stream has *access index* n. In DiskSeen, an access counter is incremented with each block reference; its value is the access index for that block and is recorded in the corresponding block table entry to represent the access time.

To facilitate efficient removal of old BTEs, each directory entry records the largest access index of all of the blocks under that entry. Purging the table of old blocks involves removing all blocks with access indices smaller than some given index. The execution of this operation entails traversing the table, top level first, identifying access indices smaller than the given index, removing the corresponding subtrees, and reclaiming the memory.

3 The Design of *DiskSeen*

In essence, DiskSeen is a sequence-based history-aware prefetch scheme. We leave file-level prefetching enabled; DiskSeen concurrently performs prefetching at a lower level to mitigate the inadequacies of file-level prefetching. DiskSeen seeks to detect sequences of block accesses based on LBN. At the same time, it maintains block access history and uses the history information to further improve the effectiveness of prefetching when recorded access patterns are observed to be repeated. There are four objectives in the design of DiskSeen.

1. *Efficiency.* We ensure that prefetched blocks are in a localized disk area and are accessed in the ascending order of their LBNs for optimal disk performance.
2. *Eagerness.* Prefetching is initiated immediately when a prefetching opportunity emerges.
3. *Accuracy.* Only the blocks that are highly likely to be requested are prefetched.
4. *Aggressiveness.* Prefetching is made more aggressive if it helps to reduce request service times.

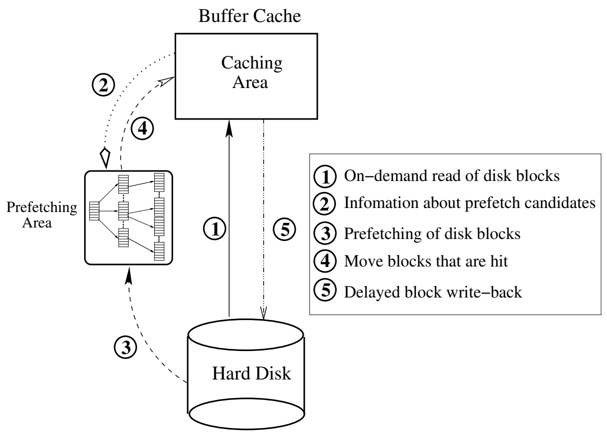
Buffer Cache
Caching Area
Prefetching Area
Hard Disk
1
2
3
4
5
1 On−demand read of disk blocks
2 Infomation about prefetch candidates
3 Prefetching of disk blocks
4 Move blocks that are hit
5 Delayed block write−back

Figure 2: **DiskSeen system diagram**. Buffer cache is divided into two areas, prefetching and caching areas, according to their roles in the scheme. A block could be prefetched into the prefetching area based on either current or historical access information—both are recorded in the disk block table, or as directed by file-level prefetching. The caching area corresponds to the traditional buffer cache and is managed by the existing OS kernel policies except that prefetched but not-yet-requested blocks are no longer stored in the cache. A block is read into the caching area either from the prefetching area, if it is hit there, or directly from disk, all in an on-demand fashion.

As shown in Figure 2, the buffer cache managed by DiskSeen consists of two areas: prefetching and caching areas. The caching area is managed by the existing OS kernel policies, to which we make little change for the sake of generality. We do, however, reduce the size of the caching area and use that space for the prefetching area to make the performance comparison fair.

DiskSeen distinguishes on-demand requests from file-level prefetch requests, basing disk-level prefetch decisions only on on-demand requests, which reflect applications' actual access patterns. While DiskSeen generally respects the decisions made by a file-level prefetcher, it also attempts to identify and screen out inaccurate predictions by the prefetcher using its knowledge of deep access history. To this end, we treat the blocks contained in file-level prefetch requests as prefetch candidates and pass them to DiskSeen, rather than passing the requests directly to disk. DiskSeen forwards on-demand requests from existing request mechanisms directly to disk. We refer to disk requests from 'above' DiskSeen (e.g., application or file-level prefetchers) as *high-level requests*.

3.1 Recording Access Indices

Block access indices are read from a counter that increments whenever a block is transferred into the caching area on demand. When the servicing of a block request is completed, either via a hit in the prefetching area or via the completion of a disk access, the current reading of the counter, an access index, is used as an access time to be recorded in the corresponding BTE in the block table. Each BTE holds the most recent access indices, to a maximum of four. In our prototype implementation, the size of a BTE is 128 bits. Each access index takes 31 bits and the other 4 bits are used to indicate block status information such as whether a block is resident in memory. With a block size of 4K Bytes, the 31-bit access index can distinguish accesses to 8 TBytes of disk data. When the counter approaches its maximum value, specifically the range for used access index exceeds 7/8 of the maximum index range, we remove the indices whose values are in the first half of the used range in the block table. In practice this progressive *index clearing* takes place very infrequently and its impact is minimal. In addition, a block table that consumes 4MB of memory can record history for about 1GB file access working set.

3.2 Coordinating Disk Accesses

We monitor the effectiveness of high-level prefetchers by tracking the use of prefetch candidates by applications. When a prefetch candidate block is read into the prefetching area, we mark the status of the block as *prefetched* in its BTE. This status can only be removed when an on-demand access of the block occurs. When the high-level prefetcher requests a prefetch candidate that is not yet resident in memory and has the *prefetched* status, DiskSeen ignores this candidate. This is because a previous prefetching of the block has not been followed by any on-demand request for it, which suggests an inaccurate prediction on the block made by the high-level prefetcher. This ability to track history prefetching events allows DiskSeen to identify and correct some of the misprefetchings generated by file-level prefetch policies.

For some access patterns, especially sequential accesses, the set of blocks prefetched by a disk-level prefetcher may also be prefetch candidates of file-level prefetchers or may be on-demand requested by applications. So we need to handle potentially concurrent requests for the same block. We coordinate these requests in the following way. Before a request is sent to the disk scheduler to be serviced by disk, we check the block(s) contained in the request against corresponding BTEs to determine whether the blocks are already in the prefetching area. For this purpose, we designate a *resident* bit in each BTE, which is set to 1 when a block enters buffer cache, and is reset to 0 when it leaves the cache. There is also a *busy* bit in each BTE that serves as a lock to coordinate simultaneous requests for a particular block. A set busy bit indicates that a disk service on the corresponding block is under way, and succeeding requests for the block must wait on the lock. DiskSeen ignores prefetch candidates whose resident or busy bits are set.

3.3 Sequence-based Prefetching

The access of each block from a high-level request is recorded in the block table. Unlike maintaining access state per file, per process, in file-level prefetching, DiskSeen treats the disk as a one-dimensional block array that is represented by leaf-level entries in the block table. Its method of sequence detection and access prediction is similar in principle to that used for the file-level prefetchers in some popular operating systems such as Linux and FreeBSD [2, 20].

3.3.1 Sequence Detection

Prefetching is activated when accesses of K contiguous blocks are detected, where K is chosen to be 8 to heighten confidence of sequentiality. Detection is carried out in the block table. For a block in a high-level request we examine the most recent access indices of blocks physically preceding the block to see whether it is the Kth block in a sequence. This back-tracking operation on the block table is an efficient operation compared to disk service time. Because access of a sequence can be interleaved with accesses in other disk regions, the most recent access indices of the blocks in the sequence are not necessarily consecutive. We only require that access indices of the blocks be monotonically decreasing. However, too large a gap between the access indices of two contiguous blocks indicates that one of the two blocks might not be accessed before being evicted from the prefetching area (i.e., from memory) if they were prefetched together as a sequence. Thus these two blocks should not be included in the same sequence. We set an access index gap threshold, T, as $1/64$ of the size of the total system memory, measured in blocks.

3.3.2 Sequence-based Prefetching

When a sequence is detected we create two 8-block windows, called the current window and the readahead window. We prefetch 8 blocks immediately ahead of the sequence into the current window, and the following 8 blocks into the readahead window. We then monitor the number f of blocks that are hit in the current window by high-level requests. When the blocks in the readahead window start to be requested, we create a new readahead window whose size is $2f$, and the existing readahead window becomes the new current window, up to a maximum window size. Specifically, we set minimal and maximum window sizes, *min* and *max*, respectively. If $2f < min$, the prefetching is canceled. This is because requesting a small number of blocks cannot amortize a disk head repositioning cost and so is inefficient. If $2f > max$, the prefetching size is max. This is because prefetching too aggressively imposes a high risk of mis-prefetching and increases pressure on the prefetching area. In our prototype, min is 8 blocks and max is 32 blocks (with block size of 4KB). We note that the actual number of blocks that are read into memory can be less than the prefetch size just specified because resident blocks in the prefetch scope are excluded from prefetching. That is, the window size becomes smaller when more blocks in the prefetch scope are resident. Accordingly, prefetching is slowed down, or even stopped, when many blocks to be prefetched are already in memory.

3.3.3 Data Structure for Managing Prefetched Blocks

In the DiskSeen scheme, each on-going prefetch is represented using a data structure called the prefetch stream. The prefetch stream is a pseudo-FIFO queue where prefetched blocks in the two windows are placed in the order of their LBNs. A block in the stream that is hit moves immediately to the caching area. For one or multiple running programs concurrently accessing different disk regions, there would exist multiple streams. To facilitate the replacement of blocks in the prefetching area, we have a global FIFO queue called the reclamation queue. All prefetched blocks are placed at the queue tail in the order of their arrival. Thus, blocks in the prefetch windows appear in both prefetch streams and the reclamation queue.[3] A block leaves the queue either because it is hit by a high-level request or it reaches the head of the queue. In the former case the block enters the caching area, in the latter case it is evicted from memory.

3.4 History-aware Prefetching

In the sequence-based prefetching, we only use the block accesses of current requests, or recently detected access sequences, to initiate sequential prefetching. Much richer history access information is available in the block table, which can be used to further improve prefetching.

3.4.1 Access Trails

To describe access history, we introduce the term *trail* to describe a sequence of blocks that have been accessed with a small time gap between each pair of adjacent blocks in the sequence and are located in a pre-determined region. Suppose blocks $(B_1, B_2, ..., B_n)$ are a trail, where $0 < access\_index(B_i) - access\_index(B_{i-1}) < T$, and $|LBN(B_i) - LBN(B_1)| < S$, $(i = 2, 3, ..., n)$, where T is the same access index gap threshold as the one used in the sequence detection for the sequence-based prefetching. A block can have up to four access indices, any one of which can be used to satisfy the given condition. If B_1 is the start block of the trail, all of the following blocks must be on either side of B_1 within distance S. We refer to the window of $2S$ blocks, centered at the start block, as the *trail extent*. The sequence detected in sequence-based prefetching is a special trail in which all blocks are on the same side of start block and have contiguous LBNs. By using a window of limited size (in our implementation S is 128), we allow a trail to capture only localized accesses so that prefetching such a trail is efficient and the penalty for a mis-prefetching is small. For an access pattern with accesses over a large area, multiple trails would be formed to track each set of proximate accesses rather than forming an extended trail that could lead to expensive disk head movements. Trail detection is of low cost because, when the access index of one block in a trail is specified, at most one access index of its following block is likely to be within T. This is because the gap between two consecutive access indices of a block is usually very large (because they represent access, eviction, and re-access). Figure 3 illustrates.

3.4.2 Matching Trails

While the sequence-based prefetching only relies on the current on-going trail to detect a pure sequence for activating prefetching, we now can take advantage of history information, if available, to carry out prefetching even if a pure sequence cannot be detected, or to prefetch more accurately and at the right time. *The general idea is to use the current trail to match history trails and then use matched history trails to identify prefetchable blocks.* Note that history trails are detected in real-time and that there is no need to explicitly record them.

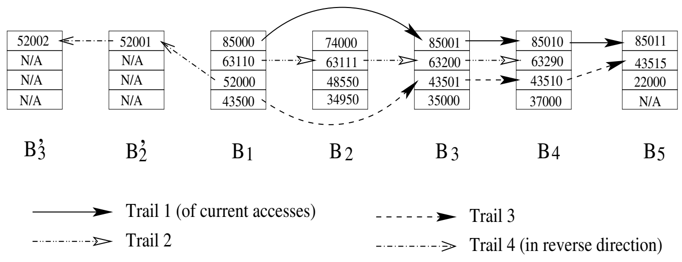

Figure 3: **Access trails.** Access index threshold T is assumed to be 256. There are four trails starting from block B_1 in a segment of the block table: one *current trail* and three *history trails*. Trail 1 (B_1, B_3, B_4, B_5) corresponds to the on-going continuous block accesses. This trail cannot lead to a sequence-based prefetch because B_2 is missing. It is echoed by two history trails: Trails 2 and 3, though Trail 1 only overlaps with part of Trail 2. A trail may run in the reverse direction, such as Trail 4.

When there is an on-demand access of a disk block that is not in any current trail's extent, we start tracking a new trail from that block. Meanwhile, we identify history trails consisting of blocks visited by the current trail in the same order. Referring to Figure 3, when the current trail extends from $B_1(85000)$ to $B_3(85001)$, two history trails are identified: Trail 2 ($B_1(63110)$, $B_3(63200)$) and Trail 3 ($B_1(43500)$, $B_3(43501)$). When the current trail advances to block B_4, both Trail 2 and Trail 3 successfully extend to it. However, only Trail 3 can match the current trail to B_5 while Trail 2 is broken at the block.

3.4.3 History-aware Prefetching

Because of the strict matching requirement, we initiate history-aware prefetching right after we find a history trail that matches the current trail for a small number of blocks (4 blocks in the prototype). To use the matched history trails to find prefetchable blocks, we set up a trail extent centered at the last matched block, say block B. Then we run the history trails from B in the extent to obtain a set of blocks that the matched history trails will probably visit. Suppose ts is an access index of block B that is used in forming a matched history trail, and T is access index gap threshold. We then search the extent for the blocks that contain an access index between ts and $ts + T$. We obtain the extension of the history trail in the extent by sorting the blocks in the ascending order of their corresponding access indices. We then prefetch the non-resident ones in the order of their LBNs and place them in the current window, similarly to the sequence-based two-window prefetching. Starting from the last prefetched block, we similarly prefetch blocks into a readahead window. The initial window sizes, or the number of blocks to be prefetched, of these two windows are 8. When the window size is less than min(=8), prefetching aborts. When the window size is larger than max(=64), only the first max blocks are prefetched. If there are multiple matched history trails, we prefetch the intersection of these trails. The two history-aware windows are shifted forward much in the same way as in the sequence-based prefetching. To keep history-aware prefetching enabled, there must be at least one matched history trail. If the history-aware prefetching aborts, sequence-based prefetching is attempted.

3.5 Balancing Memory Allocation between the Prefetching and Caching Areas

In DiskSeen, memory is adaptively allocated between the prefetching area and caching area to maximize system performance, as follows. We extend the reclamation queue with a segment of 2048 blocks which receive the metadata of blocks evicted from the queue. We also set up a FIFO queue, of the same size as the segment for the prefetching area, that receives the metadata of blocks evicted from the caching area. We divide the runtime into epochs, whose size is the period when N_{p-area} disk blocks are requested, where N_{p-area} is a sample of current sizes of the prefetching area in blocks. In each epoch we monitor the numbers of hits to these two segments (actually they are misses in the memory), $H_{prefetch}$ and H_{cache}, respectively. If $|(H_{prefetch} - H_{cache})|/N_{p-area}$ is larger than 10%, we move 128 blocks of memory from the area with fewer hits to the other area to balance the misses between the two.

4 Experimental Evaluation

To evaluate the performance of the DiskSeen scheme in a mainstream operating system, we implemented a prototype in the Linux 2.6.11 kernel. In the following sections we first describe some implementation-related issues, then the experimental results of micro-benchmarks and real-life applications.

4.1 Implementation Issues

Unlike the existing prefetch policies that rely on high-level abstractions (i.e., file ID and offset) that map to disk blocks, the prefetch policy of DiskSeen directly accesses blocks via their disk IDs (i.e., LBNs) without the knowledge of higher-level abstractions. By doing so, in addition to being able to extract disk-specific performance when accessing file contents, the policy can also prefetch metadata, such as inode and directory blocks, that cannot be seen via high-level abstractions, in LBN-ascending order to save disk rotation time. To make the LBN-based prefetched blocks usable by high-level I/O routines, it would be cumbersome to proactively back-translate LBNs to file/offset representations. Instead, we treat a disk partition as a raw device file to read blocks in a prefetch operation and place them in the prefetching area. When a high-level I/O request is issued, we check the LBNs of requested blocks against those of prefetched blocks. A match causes a prefetched block to move into the caching area to satisfy the I/O request.

To implement the prototype, we added to the stock Linux kernel about 1100 lines of code in 15 existing files concerned with memory management and the file system, and another about 3700 lines in new files to implement the main algorithms of DiskSeen.

4.2 Experimental Setup

The experiments were conducted on a machine with a 3.0GHz Intel Pentium 4 processor, 512MB memory, Western Digital WD1600JB 160GB 7200rpm hard drive. The hard drive has an 8MB cache. The OS is Redhat Linux WS4 with the Linux 2.6.11 kernel using the Ext3 file system. Regarding the parameters for DiskSeen, T, the access index gap threshold, is set as 2048, and S, which is used to determine the trail extent, is set as 128.

4.3 Performance of One-run Benchmarks

We selected six benchmarks to measure their individual run times in varying scenarios. These benchmarks represent various common disk access patterns of interest. Among the six benchmarks, which are briefly described following, *strided* and *reversed* are synthetic and the other four are real-life applications.

1. *strided* is a program that reads a 1GB file in a strided fashion—it reads every other 4KB of data from the beginning to the end of the file. There is a small amount of compute time after each read.

2. *reversed* is a program that sequentially reads one 1GB file from its end to its beginning.

3. *CVS* is a version control utility commonly used in software development environment. We ran *cvs -q diff*, which compares a user's working directory to a central repository, over two identical data sets stored with 50GB space between them.

4. *diff* is a tool that compares two files for character-by-character differences. This was run on two data sets. Its general access pattern is similar to that of CVS. We use their subtle differences to illustrate performance differences DiskSeen can make.

5. *grep* is a tool to search a collection of files for lines containing a match to a given regular expression. It was run to search for a keyword in a large data set.

6. *TPC-H* is a decision support benchmark that processes business-oriented queries against a database system. In our experiment we use PostgreSQL 7.3.18 as the database server. We choose the scale factor 1 to generate the database and run a query against it. We use queries 4 and 17 in the experiment.

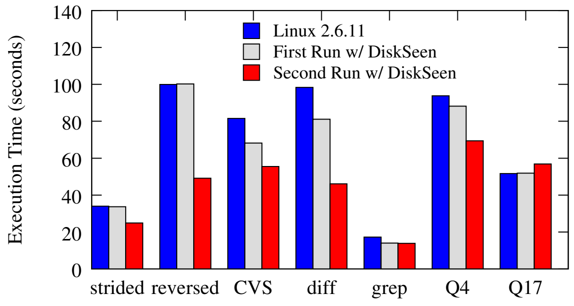

Figure 4: Execution times of the six benchmarks, including two TPC-H queries, Q4 and Q17.

To facilitate the analysis of experiment results across different benchmarks, we use the source code tree of Linux kernel 2.6.11 as the data set, whose size is about 236MB, in benchmarks *CVS*, *diff*, and *grep*. Figure 4 shows the execution times of the benchmarks on the stock Linux kernel, and the times for their first and second runs on the kernel with the DiskSeen enhancement. Between any two consecutive runs, the buffer cache is emptied to ensure all blocks are accessed from disk in the second run. For most of the benchmarks, the first runs with DiskSeen achieve substantial performance improvements due to DiskSeen's sequence-based prefetching, while the second runs enjoy further improvement because of the history information from the first runs. The improved performance for the second runs is meaningful in practice because users often run a program multiple times with only part of the input changed, leaving the on-disk data set accessed as well as access patterns over them largely unchanged across runs. For example, a user may run *grep* many times to search different patterns over the same set of files, or *CVS* or *diff* again with some minor changes to several files. Following we analyze the performance results in detail for each benchmark.

Strided, reversed. With its strided access patterns no sequential access patterns can be detected for *stride* either at the file level or at disk level. The first run with DiskSeen does not reduce its execution time. Neither does it increase its execution time, which shows that the overhead of DiskSeen is minimal. We have a similar observation with *reversed*. With the history information, the second runs of the two benchmarks with DiskSeen show significant execution reductions: 27% for *stride* and 51% for *reversed*, because history trails lead us to find the prefetchable blocks. It is not surprising to see a big improvement with *reversed*. Without prefetching, reversed accesses can cause a full disk rotation time to service each request. DiskSeen prefetches blocks in large aggregates and requests them in ascending order of their LBNs, and all these blocks can be prefetched in one disk rotation. Note that the disk scheduler has little chance to reverse the continuously arriving requests and service them without waiting for a disk rotation, because it usually works in a work-conserving fashion and requests are always dispatched to disk at the earliest possible time. This is true at least for synchronous requests from the same process. Recognizing that reverse sequential and forward/backward strided accesses are common and performance-critical access patterns in high-performance computing, the GPFS file system from IBM [25] and the MPI-IO standard [19] provide special treatment for identifying and prefetching these blocks. If history access information is available, DiskSeen can handle these access patterns as well as more complicated patterns without making file systems themselves increasingly complex.

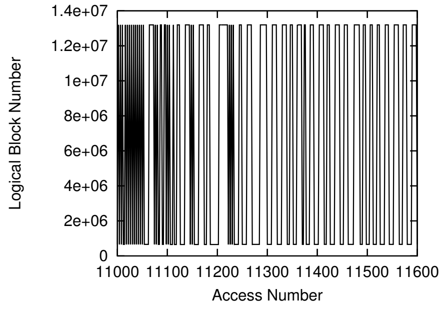

Figure 5: A sample of CVS execution without DiskSeen.

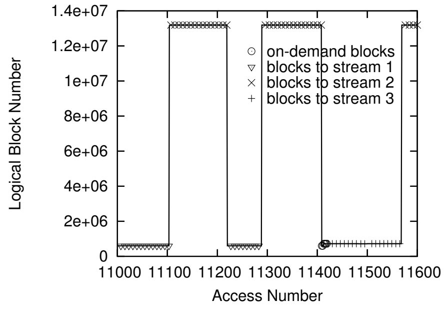

Figure 6: A sample of CVS execution with DiskSeen.

CVS, diff. As shown in Figure 4, DiskSeen significantly improves the performance of both *CVS* and *diff* on the first run and further on the second run. This is because the Linux source code tree mostly consists of small files, and at the file level, sequences across these files cannot be detected, so prefetching is only occasionally activated in the stock Linux kernel. However, many sequences can be detected at the disk level even without history information. Figure 5 shows a segment of *CVS* execution with the stock kernel. The X axis shows the sequence of accesses to disk blocks, and the Y axis shows the LBNs of these blocks. Lines connect points representing consecutive accesses to indicate disk head movements. In comparison, Figure 6 shows the same segment of the second run of *CVS* with DiskSeen. Most of disk head movements between the working directory and the *CVS* repository are eliminated by the disk-level prefetching. The figure also marks accesses of blocks that are on-demand fetched or prefetched into different prefetch streams. It can be seen that there are multiple concurrent prefetch streams, and most accesses are prefetches.

Certainly the radial distance between the directories also plays a role in the *CVS* executions because the disk head must travel for a longer time to read data in the other directory as the distance increases. Figure 7 shows how the execution times of *CVS* with the stock kernel and its runs with DiskSeen would change with the increase in distance. We use disk capacity between the two directories to represent their distance. Although all execution times increase with the increase of the distance, the time for the stock kernel is affected more severely because of the number of head movements involved. For example, when the distance increases from 10GB to 90GB, the time for the original kernel increases by 70%, while the times for first run and second run with DiskSeen increases by only 51% and 36%, respectively.

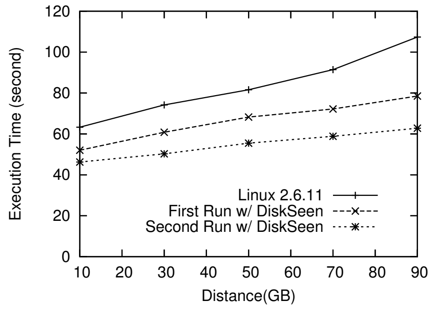

Figure 7: CVS execution times with different directory distances.

While the first runs of *CVS* and *diff* with DiskSeen reduce execution times by 16% and 18%, respectively, the second of them can further reduce the times by another 16% and 36%. For *CVS*, each directory in a *CVS*-managed source tree (i.e., working directory) contains a directory, named as *CVS*, to store versioning information. When *CVS* processes each directory, it first checks the *CVS* subdirectory, then comes back to examine other files/directories in their order in the directory. This visit to the *CVS* subdirectory disrupts the sequential accesses of regular files in the source code tree, and causes a disruption in the sequence-based prefetching. In the second run, new prefetch sequences including the out-of-order blocks (that might not be purely sequential) can be formed by observing history trails. Thus the performance gets further improvement. There are also many non-sequentialities in the execution of *diff* that prevents its first run from exploiting the full performance potential.

When we extract a kernel tar ball, the files/directories in a parent directory are not necessarily laid out in the alphabetical order of their names. However, *diff* accesses these files/directories in strict alphabetical order. So even though these files/directories have been well placed sequentially on disk, these mismatched orders would break many disk sequences, even making accesses in some directories close to random. This is why *diff* has worse performance than *CVS*. Again during the second run, history trails help to find the blocks that are proximate and have been accessed within a relatively short period of time. DiskSeen then sends prefetch requests for these blocks in the ascending order of their LBNs. In this way, the mismatch can be largely corrected and the performance is significantly improved.

Grep: While it is easy to understand the significant performance improvements of *CVS* and *diff* due to their alternate accesses of two remote disk regions, we must examine why *grep*, which only searches a local directory, also has good performance improvement, a 20% reduction in its execution time.

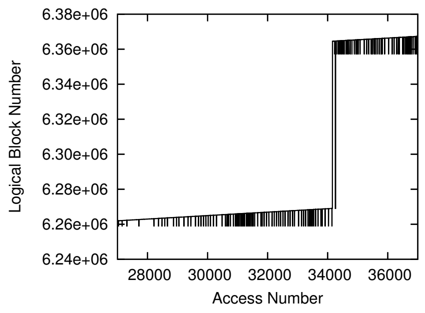

Figure 8: A sample of grep execution without DiskSeen.

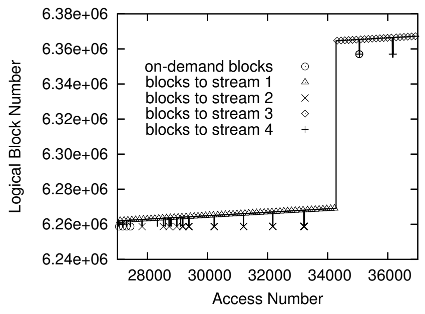

Figure 9: A sample of grep execution with DiskSeen.

Figure 8 shows a segment of execution of *grep* in the stock Linux kernel. This collection of stair-like accesses corresponds to two cylinder groups. In each cylinder group, *inode* blocks are located in the beginning, followed by file data blocks. Before a file is accessed, its *inode* must be inspected, so we see many lines dropping down from file data blocks to *inode* blocks in a cylinder group. Figure 9 shows the corresponding segment of execution of first run of *grep* with DiskSeen. By prefetching *inode* blocks in DiskSeen, most of the disk head movements disappear. The figure also shows that accesses to *inode* blocks and data blocks from different prefetch streams. This is a consequence of the decision to only attempt to prefetch in each localized area.

TPC-H: In this experiment, Query 4 performs a merge-join against table *orders* and table *lineitem*. It sequentially searches table *orders* for records representing orders placed in a specific time frame, and for each such record the query searches for the matched records in table *lineitem* by referring to an index file. Because table *lineitem* was created by adding records generally according to the order time, DiskSeen can identify sequences in each small disk area for prefetching. In addition, history-aware prefetching can exploit history trails for further prefetching opportunities (e.g., reading the index file), and achieve a 26% reduction of execution time compared to the time for the stock kernel.

However, the second run of Q17 with DiskSeen shows performance degradation (a 10% execution increase over the time for the run on the stock kernel). We carefully examined its access pattern in the query and found that table *lineitem* was read in a close-to-random fashion with insignificant spatial locality in many small disk areas. While we used a relatively large access index gap ($T = 2048$) in the experiment, this locality would make history-aware DiskSeen form many prefetch streams, each for a disk area, and prefetch a large number of blocks that will not be used soon. This causes thrashing that even the extended metadata segment of the reclamation queue cannot detect it due to its relative small size. To confirm this observation, we reduced T to 256 and re-ran the query with DiskSeen to which history access information is available. With the reduced T, the execution time is increased by only 2.6%.

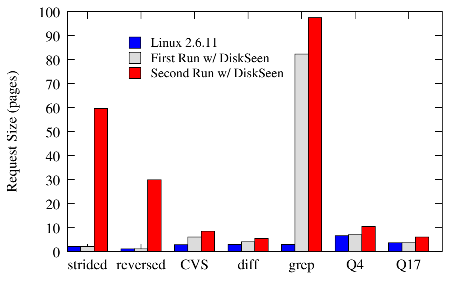

Figure 10: The sizes of requests serviced by disk.

Disk request sizes: Disk performance is directly affected by the sizes of requests a disk receives. To obtain the sizes, we instrument the Linux kernel to monitor READ/WRITE commands issued to the IDE disk controller and record the sizes of corresponding requests. We report the average size of all the requests during the executions of the benchmarks in Figure 10. From the figure we can see that in most cases DiskSeen significantly increases the average request sizes, which corresponds to their respective execution reductions shown in Figure 4. These increases are not proportional to their respective reductions in execution time because of factors such as the proportion of I/O time in the total execution time and differences in the seek times incurred.

4.4 Performance of Continuously Running Application

For applications that are continuously running against the same set of disk data, previous disk accesses could serve as the history access information to improve the I/O performance of current disk accesses. To test this we installed a Web server running the general hypertext cross-referencing tool Linux Cross-Reference (LXR) [15]. This tool is widely used by Linux developers for searching Linux source code.

We use the LXR 0.3 search engine on the Apache 2.0.50 HTTP Server, and use Glimpse 4.17.3 as the free-text search engine. The file set searched is three versions of the Linux kernel source code: 2.4.20, 2.6.11, and 2.6.15. Glimpse divides the files in each kernel into 256 partitions, indexes the file set based on partitions, and generates an index file showing the keyword locations in terms of partitions. The total size of the three kernels and the index files is 896MB. To service a search query, glimpse searches the index file first, then accesses the files included in the partitions matched in the index files. On the client side, we used WebStone 2.5 [29] to generate 25 clients concurrently submitting freetext search queries. Each client randomly picks a keyword from a pool of 50 keywords and sends it to the server. It sends its next query request once it receives the results of its previous query. We randomly select 25 Linux symbols from file */boot/System.map* and another 25 popular OS terms such as *"lru", "scheduling", "page"* as the pool of candidate query keywords. Each keyword is searched in all three kernels. The metric we use is throughput of the query system represented by MBit/sec, which means the number of Mega bits of query results returned by the server per second. This metric is also used for reporting WebStone benchmark results.

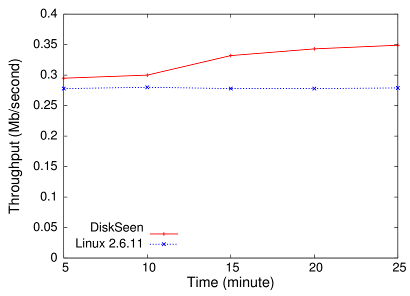

Figure 11: LXR throughputs with and without DiskSeen.

Figure 11 shows the LXR throughputs on the kernels with and without DiskSeen at different times during its execution. We have two observations. First, DiskSeen improves LXR's throughput. This is achieved by prefetching contiguous small files at disk level. Second, from the tenth minute to twentieth minute of the execution, the throughput of LXR with DiskSeen keeps increasing, while the throughput of LXR without DiskSeen does not improve. This demonstrates that DiskSeen can help the application self-improve its performance by using its own accumulated history access information.

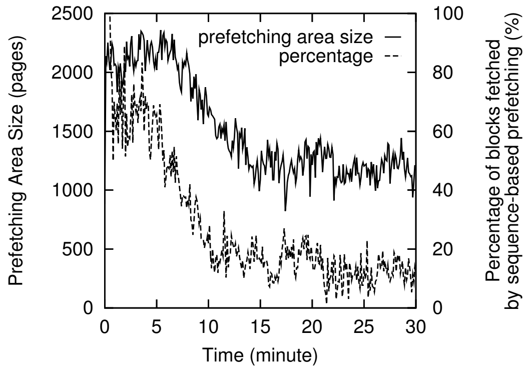

Figure 12: Prefetching area allocation and percentage of blocks fetched by sequence-based prefetching

Figure 12 shows that the size of the prefetching area changes dynamically during execution, and the percentages of blocks that are prefetched through sequence-based prefetching, including the prefetch candidates that are loaded, over all prefetched blocks. We can see that smaller percentages of blocks are loaded through sequence-based prefetching as the application proceeds, i.e., a larger percentage of blocks are loaded through history-aware prefetching, because of the availability of history information. This trend corresponds to the reduction of the prefetching area size. History-aware prefetching has higher accuracy than sequence-based prefetching (the miss ratios of history-aware prefetching and sequence-based prefetching are 5.2% and 11%, respectively), and most blocks fetched by history prefetching are hits and are moved to the caching area shortly after they enter the prefetching area. Thus, there are fewer hits to the metadata segment extended from the reclamation queue in the prefetching area. Accordingly, DiskSeen adaptively re-allocates some buffer space used by the prefetching area to the caching area.

4.5 Interference of Noisy History

While well matched history access information left by prior run of applications is expected to provide accurate hints and improve performance, a reasonable speculation is that a misleading history could confuse DiskSeen and even direct DiskSeen to prefetch wrong blocks so as to cause DiskSeen to actually degrade application performance. To investigate the interference effect caused by *noisy* history on DiskSeen's performance, we designed experiments in which two applications access the same set of data with different access patterns. We use *grep* and *diff* as test applications. *Grep* searches a keyword in a Linux source code tree, which is also used by *diff* to compare against another Linux source code tree. We know that *grep* scans files basically in the order of their disk layout, but *diff* visits files in the alphabetic order of directory/file names.

In the first two experiments, we run the applications alternatively, specifically in sequence (*diff*, *grep*, *diff*, *grep*) in experiment I and sequence (*grep*, *diff*, *grep*, *diff*) in experiment II. Between any two consecutive runs, the buffer cache is emptied to ensure the second run does not benefit from cached data while history access information in the block table is passed across a sequence of runs in an experiment. The execution times compared to the stock kernel are shown in Table 1.

Experiment	Execution times (seconds)			
Linux	*diff*	98.4	*grep*	17.2
I	*diff*	*grep*	*diff*	*grep*
	81.1	16.3	46.4	14.0
II	*grep*	*diff*	*grep*	*diff*
	14.0	67.7	13.9	46.1
Linux	*grep*/*diff*		20.9/55.8	
III	*grep*/*diff*		15.2/44.9	
			17.0/34.6	
			17.9/34.8	
			18.2/34.5	
			18.3/35.1	

Table 1: Execution times for *diff* and *grep* when they are alternately executed in different orders or concurrently, with DiskSeen, compared to the times for the stock kernel. The times reported are wall clock times.

If we use the execution times without any history as reference points (the first runs in experiments I and II), where only sequence-based prefetching occurs, noisy history causes the degradation of performance in the first run of *grep* by 16% (14.0s vs. 16.3s) in experiment I, while it accidentally helps improve the performance in the first run of *diff* by 17% (81.1s vs. 67.7s) in experiment II. The degradation in experiment I is due to the history access information left by *diff* that misleads DiskSeen, which is running *grep*, to infer that a matched history trail has been found and initiate a history-based prefetching. However, the matched history trail is broken when *diff* takes a different order to visit files. This causes DiskSeen to fall back to its sequence-based prefetching, which takes some time to be activated (accesses of 8 contiguous blocks). Thus, history-aware prefetching attempts triggered by noisy history keep sequence-based prefetching from achieving its performance potential. It is interesting to see that a trail left by *grep* improves the performance of *diff*, which has a different access pattern, in Experiment II. This is because the trails left by *grep* are also sequences on disk. Using these trails for history-aware prefetching essentially does not change the behavior of sequence-based prefetching, except that the prefetching becomes more aggressive, which helps reduce *diff*'s execution time. For the second runs of *grep* or *diff* in either experiment, the execution times are very close to those of the second runs shown in Figure 4. This demonstrates that noisy history only very slightly interferes with history-aware prefetching if there also exists a well-matched history in the block table (e.g., the ones left by the first runs of *grep* or *diff*, respectively).

In the third experiment, we concurrently ran these two applications five times, with the times of each run reported in Table 1, along with their counterparts for the stock kernel. The data shared by *diff* and *grep* are fetched from disk by whichever application first issues requests for them, and requests for the same blocks from the other application are satisfied in memory. The history of the accesses of the shared blocks is the result of mixed requests from both applications. Because of the uncertainty in process scheduling, access sequences cannot be exactly repeated between different runs. Each run of the two applications leaves different access trails on the shared blocks, which are noisy history that interferes with the current DiskSeen prefetching. The more runs there have been, the more history is recorded, the easier it is to trigger an incorrect history-aware prefetching. This is why the execution time of *grep* keeps increasing until the fifth run (we keep at most four access indices for each block). Unlike *grep*, the execution time of *diff* in the second run is decreased by 23% (34.6s over 44.9s). This is because history-aware prefetching of the other source code tree, which is not touched by *grep*, is not affected by the interference.

4.6 DiskSeen with a Contrived Adverse Workload

To demonstrate the extent to which DiskSeen could be ill-behaved, we designed an arguably worst-case scenario in which all predictions made by history-aware prefetching are wrong. In the experiment, a 4GB file was divided into chunks of 20 4KB blocks. Initially we sequentially read the file from its beginning to create a corresponding sequential trail. After removing buffered blocks of the file from memory, we read four blocks at the beginning of each chunk, chunk by chunk from the beginning to the end of the file. The access of four blocks in a chunk triggers a history-aware prefetching, which prefetches two windows, each of 8 blocks, in the same chunk. These 16 blocks in each chunk are all mis-predicted. The experimental result shows that for the second file read DiskSeen increased the execution time by 3.4% (from 68.0 seconds in the stock kernel to 70.3 seconds with DiskSeen). The small increase is due to the sequential access of chunks, in which the disk head will move over the prefetched blocks whether or not prefetch requests are issued. To eliminate this favorable scenario, we randomly accessed the chunks in the second read, still with only four blocks requested from each chunk. This time DiskSeen increased the execution time by 19% (from 317 seconds in the stock kernel to 378 seconds with DiskSeen), which represents a substantial performance loss. However, this scenario of a slowdown of more than fourfold (for either scheme) could often be avoided at the application level by optimizing large-scope random accesses into sequential accesses or small-region random accesses.

4.7 Discussion and Future Work

From the benchmarking we have conducted, DiskSeen is most effective in transforming random or semi-random accesses that take place on one or more limited disk areas into (semi-)sequential accesses in each disk locality. It is also effective in discovery and exploitation of sequential on-disk access that is difficult to detect at the file level.

We have not implemented a prefetch throttling mechanism in DiskSeen. This makes our system incapable of responding to overly-aggressive prefetching that leads to thrashing (e.g., in the case of Q17 of TPC-H) and miss-prefetching (e.g., in the case described in Section 4.6). An apparent fix to the issue would be a policy that adaptively adjusts the access index gap (T) based on the effectiveness of recent prefetchings (i.e., the percentage of blocks prefetched by the history-aware approach that were subsequently used). However, in a system where applications of various access patterns run concurrently, the adjustment may have to be made differently for different applications, or different access index gaps need to be used. While our experiments suggest that a fixed T works well for most access patterns and its negative impact is limited, we leave a comprehensive investigation of the issue as our future work.

There are several limitations in our work to be addressed in the future. First, our implementation and performance evaluations are currently based on one disk drive. Most enterprise-level storage systems are composed of RAIDs and their associated controllers. While we expect that DiskSeen can retain most of its performance merits because the mappings between logical blocks and the physical blocks on multiple disks still maintain high performance for contiguous LBN accesses, some new issues have to be addressed, such as the conditions on which prefetching should cross the disk boundary and the relationship between prefetching aggressiveness and parallelism of RAID. Second, we have evaluated the prototype only in a controlled experimental setting. It would be worthwhile to evaluate the system in a real-world environment with mixed workloads running for extended periods, such as using it on a file server that supports programming projects of a class of students or an E-business service. Third, the block table could become excessively large. For example, streaming of data from an entire 500GB disk drive can cause the table grow to 2GB. In this case, we need to page out the table to the disk. Other solutions would be compression of the table or avoidance of recording streaming access in the table.

5 Related Work

There are several areas of effort related to this work, spanning applications, operating systems, and file systems.

Intelligent prefetching algorithms: Prefetching is an actively research area for improving I/O performance. Operating systems usually employ sophisticated heuristics to detect sequential block accesses to activate prefetching, as well as adaptively adjust the number of blocks to be prefetched within the scope of a single file [20, 22]. By working at the file abstraction and lacking mechanism for recording historically detected sequential access patterns, the prefetch policies usually make conservative predictions, and so may miss many prefetching opportunities [21]. Moreover, their predictions cannot span files.

There do exist approaches that allow prefetching across files. In these approaches, system-wide file access history has been used in probability-based prediction algorithms, which track sequences of file access events and evaluate the probability of file occurrences in the sequences [9, 13]. These approaches may achieve a high prediction accuracy via their use of historical information. However, the prediction and prefetching are built on the unit of files rather than file blocks, which makes the approaches more suitable to web proxy/server file prefetching than to the prefetching in general-purpose operating systems [6]. The complexity and space costs have also thus far prevented them from being deployed in general-purpose operating systems. Moreover, these approaches are not applicable to prefetching for disk paging in virtual memory and file metadata.

Hints from applications: Prefetching can be made more effective with hints given by applications. In the TIP project, applications disclose their knowledge of future I/O accesses to enable informed caching and prefetching [18, 27]. The requirements on hints are usually high—they are expected to be detailed and to be given early enough to be useful. There are some other buffer cache management schemes using hints from applications [3, 5].

Compared with the method used in DiskSeen, application-hinted prefetching has limitations: (1) The requirements for generating detailed hints may put too much burden on application programmers, and could be infeasible. As an example, a file system usage study for Windows NT shows that only 5% of file-opens with sequential reads actually take advantage of the option for indicating their sequential access pattern to improve I/O performance [28]. Another study conducted at Microsoft Research shows a consistent result [7]. It would be a big challenge to require programmers to provide detailed hints sometimes by even restructuring the programs, as described in the papers on TIP [18, 27]. The DiskSeen scheme, in contrast, is transparent to applications. (2) The sequentiality across files and the sequentiality of data disk locations still cannot be disclosed by applications, which are important for prefetching of small files. In our work this sequentiality can be easily detected and exploited.

Prefetching hints can also be automatically abstracted by compilers [16] or generated by OS-supported speculative executions [4, 8]. Another interesting work is a tool called *C-Miner* [14], which uses a data mining technique to infer block correlations by monitoring disk block access sequences. The discovered correlations can be used to determine prefetchable blocks. Though the performance benefits of these approaches can be significant, they do not cover the benefits gained from simultaneously exploiting temporal and spatial correlations among on-disk blocks. In a sense, our work is complementary.

Improving data placement: Exposing information from the lower layers up for better utilization of hard disk is an active research topic. Most of the work focuses on using disk-specific knowledge for improving data placements on disk that facilitate the efficient servicing of future requests. For example, Fast File System (FFS) and its variants allocate related data and metadata into the same cylinder group to minimize seeks [17, 10]. Traxtent-aware file system excludes track boundary block from being allocated for better disk sequential access performance [24]. However, these optimized block placements cannot be seen at the file abstraction. Because most files are of small sizes (e.g., a study on Windows NT file system usage shows that 40% of operations are to files shorter than 2KB [28]), prefetching based on individual file abstractions cannot take full advantages of these efforts. In contrast, DiskSeen can directly benefit from these techniques by being able to more easily find sequences that can be efficiently accessed based on optimized disk layout.

Recently, the FS2 file system was proposed to dynamically create block replicas in free spaces on disk according to the observed disk access patterns [11]. These replicas can be used to provide faster accesses of disk data. FS2 dynamically adjusts disk data layout to make it friendly to the changing data request pattern, while DiskSeen leverages buffer cache management to create disk data request patterns that exploit current disk layout for high bandwidth. These two approaches are complementary. Compared with looking for free disk space to make replicas consistent to the access patterns in FS2, DiskSeen can be more flexible and responsive to the changing access pattern.

6 Conclusions

DiskSeen addresses a pressing issue in prefetch techniques—how to exploit disk-specific information so that effective disk performance is improved. By efficiently tracking disk accesses both in the live request stream and recorded prior requests, DiskSeen performs more accurate block prefetching and achieves more continuous streaming of data from disk by following the block number layout on the disk. DiskSeen overcomes barriers imposed by file-level prefetching such as the difficulties in relating accesses across file boundaries or across lifetimes of open files. At the same time, DiskSeen complements rather than supplants high-level prefetching schemes. Our implementation of the DiskSeen scheme in the Linux 2.6 kernel shows that it can significantly improve the effectiveness of prefetching, reducing execution times by 20%-53% for micro-benchmarks and real applications such as *grep*, *CVS*, *TPC-H*, and LXR.

7 Acknowledgements

We are grateful to Dr. Fay Chang for her detailed comments and suggestions on the final version of the paper. We thank the anonymous reviewers for their constructive comments. This research was supported in part by National Science Foundation grants CNS-0405909 and CCF-0602152.

References

[1] Journaling-Filesystem Fragmentation Project, *URL: http://www.informatik.uni-frankfurt.de/ loizides/reiserfs/ agesystem.html*

[2] A. R. Butt, C. Gniady, and Y. C. Hu, “The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms”, *in Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS’05)*, June 2005.

[3] P. Cao, E. W. Felten, A. Karlin and K. Li, “Implementation and Performance of Integrated Application-Controlled Caching, Prefetching and Disk Scheduling”,

ACM Transaction on Computer Systems, November 1996.

[4] F.W. Chang and G.A. Gibson, "Automatic I/O Hint Generation through Speculative Execution", *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, February 1999.

[5] P. Cao, E. W. Felten and K. Li, "Application-Controlled File Caching Policies", *Proceedings of the USENIX Summer 1994 Technical Conference*, 1994.

[6] X. Chen and X. Zhang, "A popularity-based prediction model for Web prefetching", *IEEE Computer*, Vol. 36, No. 3, March 2003.

[7] J. R. Douceur and W. J. Bolosky, "A Large-Scale Study of File-System Contents", *Proceedings of the 1999 ACM SIGMETRICS conference*, May 1999.

[8] K. Fraser and F. Chang, "Operating system I/O Speculation: How two invocations are faster than one", *Proceedings of the USENIX Annual Technical Conference* June 2003.

[9] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach", *Proceedings of the Usenix Summer Conference*, June 1994, pp. 197-208.

[10] G. Ganger and F. Kaashoek, "Embedded Inodes and Explicit Groups: Exploiting Disk Bandwidth for Small Files", *Proceedings of the 1997 USENIX Annual Technical Conference*, January 1997.

[11] H. Huang, W. Hung, and K. G. Shin, "FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption", *Proceedings of 20th ACM Symposium on Operating Systems Principles*, October 2005.

[12] 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", *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST'05)*, December 2005.

[13] T. M. Kroeger and D.D.E. Long, "Design and implementation of a predictive file prefetching algorithm", *Proceedings of the 2001 USENIX Annual Technical Conference*, January 2001.

[14] Z. Li, Z. Chen, S. Srinivasan and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems", *Proceedings of 3rd USENIX Conference on File and Storage Technologies (FAST04)*, March 2004.

[15] Linux Cross-Reference, *URL : http://lxr.linux.no/*.

[16] T. C. Mowry, A. K. Demke and O. Krieger. "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications", *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*,, October 1996.

[17] M. K. Mckusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX", *Transactions on Computer Systems*, 2(3), 1984.

[18] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, "Informed Prefetching and Caching", *Proceedings of the 15th Symposium on Operating System Principles*, 1995, pp. 1-16.

[19] MPI-2: Extensions to the Message-Passing Interface, *URL : http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html*

[20] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 Performance Improvement through Readahead Optimization", *Proceedings of the Linux Symposium*, July 2004.

[21] A. E. Papathanasiou and M. L. Scott, "Aggressive Prefetching: An Idea Whose Time Has Come", *Proceedings of the Tenth Workshop on Hot Topics in Operating Systems*, June 2005.

[22] A. J. Smith, "Sequentiality and Prefetching in Database Systems", *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978, pp. 223-247.

[23] J. Schindler and G. R. Ganger, "Automated Disk Drive Characterization", *Proceeding of 2000 ACM SIGMETRICS Conference*, June 2000.

[24] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, "Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics", *USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[25] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", *USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[26] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger, "On Multidimensional Data and Modern Disks", *Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST'05)*, December 2005.

[27] A. Tomkins, R. H. Patterson and G. Gibson, "Informed Multi-Process Prefetching and Caching", *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997.

[28] W. Vogels, "File system usage in Windows NT 4.0", *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, December 1999.

[29] WebStone — The Benchmark for Web Servers, *URL : http://www.mindcraft.com/benchmarks/webstone/*

[30] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes, "On-line extraction of SCSI disk drive parameters" *In Proceeding of 1995 ACM SIGMETRICS Conference*, May 1995.

Notes

[1] We make this statement for generic OS kernels. Some operating systems adopt aggressive prefetch policies which rely on high-level knowledge about user/application behaviors. An example is the SuperFetch technique in Windows Vista, which performs prefetching according to particular applications, users, usage times of day or even usage days of week.

[2] Specifically we do not expose information about logical disk layout, which actually has been available for prefetch operations in operating systems. We use 'expose' to indicate a general approach utilizing low-level disk-specific knowledge, which could include hidden disk geometry information below the LBN abstraction in future work.

[3] In the implementation the prefetch streams are only a conceptual data structure—they are embedded in the reclamation queue and blocks appear only once.