A Prefetching Scheme Exploiting both Data Layout and Access History on Disk

SONG JIANG, Wayne State University
XIAONING DING, New Jersey Institute of Technology
YUEHAI XU, Wayne State University
KEI DAVIS, Los Alamos National Laboratory

Prefetching is an important technique for improving effective hard disk performance. A prefetcher seeks to accurately predict which data will be requested and load it ahead of the arrival of the corresponding requests. Current disk prefetch policies in major operating systems track access patterns at the level of file abstraction. While this is useful for exploiting application-level access patterns, for two reasons file-level prefetching cannot realize the full performance improvements achievable by prefetching. 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 performance penalty for mis-prefetching a randomly-located block, relative to that of a sequential block, is correspondingly greater.

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, any present file-level prefetch policies. 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. It also implements a mechanism to minimize mis-prefetching, on a per-application basis, to mitigate the corresponding performance penalty.

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%-60% for micro-benchmarks and real applications such as *grep*, *CVS*, and *TPC-H*. Even for workloads specifically designed to expose its weaknesses DiskSeen incurs only minor performance loss.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Design Studies*; D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*

General Terms: Algorithms, Performance, Design, Experimentation

Additional Key Words and Phrases: Prefetching, spatial locality, hard disk, buffer cache

ACM Reference Format:
Jiang, S., Ding, X., Xu, Y., Davis, K.. 2013. A Prefetching Scheme Exploiting both Data Layout and Access History on the Disk. ACM Trans. Storage 9, 4, Article 39 (March 2010), 21 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

This research was supported in part by National Science Foundation grants CCF-0702500, CCF-0845711, CAREER CCF 0845711, CNS 1117772, and CNS 1217948. A preliminary version has been published in the Proceedings of 2007 USENIX Annual Technical Conference, Santa Clara, CA, June 2007.
Author's addresses: Song Jiang and Yuehai Xu, Electrical and Computer Engineering Department, Wayne State University, Detroit, MI 48202, USA. Email:{sjiang, yhxu}@wayne.edu; Xiaoning Ding, Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA. Email: xiaoning.ding@njit.edu; and Kei Davis, CCS Division, Los Alamos National Laboratory, Los Alamos, NM 87545, USA. Email: kei.davis@lanl.gov.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

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 becoming 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 effective 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 the actual time costs of individual accesses. First, a hard disk is a non-uniform-access device for which accessing sequential locations without disk head movement is at least an order of magnitude faster than random access. Second, as an application load becomes increasingly I/O-bound, such that disk accesses can be decreasingly hidden behind computation, the importance of carrying out sequential prefetching increases relative to the importance of performing accurate but random prefetching. 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 expected benefits, or too likely to hurt performance in some common scenarios. To be more relevant to common practice the following discussion is specific to prefetch policies used in general-purpose operating systems.

Most existing prefetch policies 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 this structure. However, because these policies are oblivious to disk data layout, they do not have the knowledge of where the next prefetched block would be relative to the current fetched block to estimate prefetching cost. Thus their measure of prefetching effectiveness, which is usually used as feedback to adjust prefetching behavior, is in terms of the number of mis-prefetched blocks rather than a more relevant metric, the penalty for 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 dispatch 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. Our 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.

First, 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. Microsoft's NTFS) or becomes full (e.g. Linux Ext2) this correspondence may deteriorate. This worsens the penalty for mis-prediction.

Second, the file abstraction is not a convenient level for recording deep access history information. This is exacerbated by the complications 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 [Papathanasiou and Scott 2005]. A further consequence is that sequential access of a short file will not trigger the prefetch mechanism.

Third, inter-file sequentiality is not exploited. In a general-purpose OS, file-level prefetching usually takes place within individual files because of the complexities and overhead of inter-file prefetching. This precludes practical detection of sequential accesses 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 working set on the disk. 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 in-disk prefetching.

Our presentation proceeds as follows. In Section 2 we first describe an efficient method for tracking disk block accesses and analyzing associations between blocks. We then show how to efficiently detect sequences of accesses of disk blocks and to appropriately initiate prefetching. In Section 3 we show how to use access history information to detect and exploit complex pseudo-sequences with high accuracy. In Section 4 we show that an implementation of these algorithms—collectively DiskSeen—in the current Linux kernel can yield significant performance improvements on representative applications. Section 5 discusses related work, and Section 6 concludes.

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.

2.1. Exposing Disk Layout Information

Generally, the more specific the information available for a particular disk, the more accurate an estimation 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 [J. Schindler and Ganger 2002]. As another example, knowing that a set of non-contiguous blocks has 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 [Schlosser et al. 2005]. However, detailed disk performance characterization requires knowledge of physical disk geometry that is not disclosed by disk manufacturers, and its extraction, either interrogative or empirical, is a challenging task [Schindler and Ganger 2000]. Different extraction approaches may have different accuracy and work only with certain types of disk interfaces (e.g. SCSI).

An interface abstraction that disk devices commonly provide is logical disk geometry: a linearized data layout and represented by a sequence $[0, 1, 2, \ldots, n]$ of *logical block numbers* (LBNs). Disk manufacturers generally make great 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 [Schlosser et al. 2005]. 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. We show that exposing 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 entries of their neighboring blocks via LBNs, and efficient addition and removal of block entries.

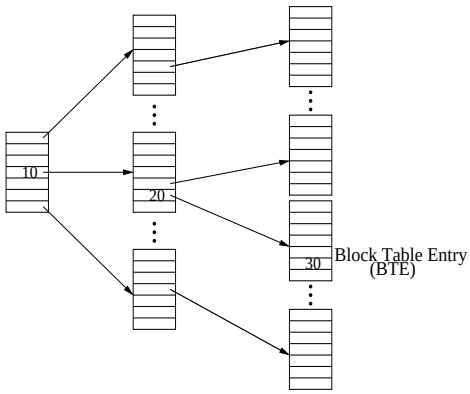

Fig. 1. **Block table**. There are three levels in this example block table: two directory levels and one leaf level. The table entries at differing levels are fit into separate memory pages. An entry at the leaf level is called a block table entry (BTE). If one page can hold 512 entries, the access 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, where 10 and 20 are indices in their respective pages and each entry contains the address of corresponding page in its next level.

The block table, which has been used in the DULO scheme for identifying block sequences [Jiang et al. 2005], is inspired by the multi-level page table used in almost all operating systems for memory address translation. 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 DiskSeen one or multiple timestamps are recorded at the leaf level entry (i.e., block table entry (BTE)) of a block to represent its most recent access times. An access counter is incremented with each block reference; its value is the timestamp for that block and is recorded in the corresponding BTE to represent the access time.

To facilitate efficient removal of old BTEs, each directory (non-leaf) entry records the largest timestamp of all of the blocks under that entry. To purge the table—remove all blocks with timestamps smaller than some given timestamp—entails traversing the table, top level first, identifying timestamps smaller than the given timestamp, 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 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. Significant mis-prefetching automatically suppresses prefetching.
(4) *Aggressiveness.* Prefetching is made more aggressive if it helps to reduce request service times.

As shown in Figure 2, the buffer cache managed by DiskSeen consists of a prefetching area and a caching area. The caching area is managed by the existing OS kernel policies, to which we make

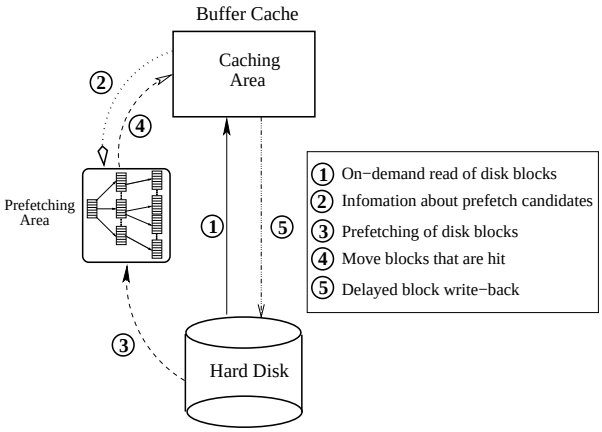

Fig. 2. **DiskSeen system diagram**. Buffer cache is divided into prefetching and caching areas according to their roles in the scheme. A block could be prefetched into the prefetching area based on either current or historical access information—both are recorded in the disk block table, or as directed 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.

minimal changes for the sake of portability. We do, however, reduce the size of the caching area by the size of 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 referenced by file-level prefetch requests as *prefetch candidates* and pass them to DiskSeen rather than passing them 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., applications or file-level prefetchers) as *high-level requests*.

3.1. Recording Block Access Times

The access times of a block are represented by timestamps that are read from a counter that is incremented 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 value of the counter is used as an access time to be recorded in the corresponding BTE in the block table. Each BTE holds the most recent timestamps, to a maximum of four. In our prototype implementation the size of a BTE is 128 bits. Each timestamp is 31 bits and the remaining 4 bits are used to indicate block status information such as whether a block is resident in memory. With a block size of 4KB the 31-bit timestamp can distinguish accesses to 8TB of disk data. When the counter approaches its maximum value, specifically when the range for used timestamps exceeds $7/8$ of the maximum timestamp range, we remove the timestamps whose values are in the first half of the current range of the block table. In practice this progressive timestamp clearing occurs very infrequently and its effect is minimal. A block table using 4MB of memory can record history for a working set of about 1GB. The space overhead for recording the timestamps is modest when using the table purging mechanism. If needed, old history information could be saved on disk for future use. By having four timestamps per block, the history-aware prefetching has relatively deep history information with which to make predictions of blocks to be requested. Furthermore, this usually makes prefetching more immune to access *interference*, as described and shown in Section 4.5.

3.2. Coordinating Disk Accesses

We monitor the effectiveness of high-level prefetchers by tracking the use of prefetch candidates. When a prefetch candidate is read into the prefetching area we mark the block's BTE as *prefetched*. This status is only cleared if an on-demand access of the block occurs. When the high-level prefetcher requests a block that is not resident in memory and has *prefetched* status, DiskSeen ignores the request. This is because a previous prefetching of the block was not followed by an on-demand request for it before it was evicted, suggesting an inaccurate prediction on the block previously made by the high-level prefetcher. This ability to track prefetching history allows DiskSeen to identify and correct some of the mis-prefetchings 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 requested by file-level prefetchers or may be on-demand requests by applications. To coordinate concurrent requests for the same block, before a request is sent to the disk scheduler to be serviced by disk we check the blocks 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 requests whose resident or busy bits are set. Thus only requests for blocks whose resident and busy bits are not set are sent to the disk scheduler.

3.3. Sequence-based Prefetching

The access of each block by 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 [Butt et al. 2005; R. Pai and Cao 2004], but because DiskSeen operates directly on disk mappings it is not constrained by file boundaries.

3.3.1. Sequence Detection. Prefetching is activated when accesses of K contiguous blocks are detected, where K is chosen to be 8 to give confidence of sequentiality. Selection of this K value is based on empirical knowledge obtained in the system evaluation. DiskSeen's performance advantage is not sensitive to this parameter between values of 6 and 10. Detection is carried out in the block table. For a block in a high-level request we examine the most recent timestamps 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 efficient compared to disk service time. Because access of a sequence can be interleaved with accesses to other disk regions, the most recent timestamps of the blocks in the sequence might not be consecutive, so we only require that the timestamps be monotonically decreasing. However, too large a gap between the timestamps 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, in which case these two blocks should not be included in the same sequence. We use a *timestamp gap threshold*, T, equal to $1/64$ of the size of the total system buffer memory, measured in blocks, which is also the default minimum prefetching area size (Section 3.6). As an example, for memory size of 512MB and block size of 4KB, the threshold is 2048 ($= 512\text{MB}/(64*4\text{KB})$). Our empirical data show that in many cases the prefetch area holds about 1/64 of the total system buffer size. The performance of DiskSeen is not sensitive to the value of T within a large range. If the threshold is extremely small (e.g., smaller than 50) a prefetch sequence may not be formed; if it is very large it may lead to wasteful prefetching and consequent suspension of prefetching by DiskSeen's quality control mechanism (Section 3.5).

3.3.2. Sequence-based Prefetching. When a sequence is detected we create two 8-block windows, 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 $2f$ (up to a maximum window size), and the existing readahead window becomes the new current window. In detail, we set minimum and maximum window sizes, *min* and *max*, respectively. If $2f < min$, prefetching is canceled 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* 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 4KB block size). We note that the actual number of blocks that are read into memory can be less than the prefetch size so 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 memory. Thus potentially inefficient prefetching, such as requests for non-contiguous and/or a small number of blocks, can be avoided.

3.3.3. Data Structure for Managing Prefetched Blocks. In DiskSeen each on-going prefetch is represented using a *prefetch stream*, 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 is immediately moved 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 tail of the reclamation queue in the order of their arrival. Thus blocks in the prefetch windows appear in both prefetch streams and the reclamation queue.[1] A block leaves the reclamation 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, and in the latter case it is evicted from memory.

3.4. History-aware Prefetching

In sequence-based prefetching we only use the block accesses of current requests, or recently detected access sequences, to initiate sequential prefetching. A key observation is that the block table in fact contains much richer access information that 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 has been accessed with a small time interval between each consecutive pair of blocks in the sequence and is located in a spatially bounded region. Suppose blocks $(B_1, B_2, \ldots, B_n)$ are a trail, where $0 < timestamp(B_i) - timestamp(B_{i-1}) < T$, and $|LBN(B_i) - LBN(B_1)| < S$, $(2 \leq i \leq n)$, where T is the same timestamp gap threshold used for sequence detection in sequence-based prefetching. A block can have up to four timestamps, 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*. Thus a sequence detected in sequence-based prefetching is the special case of a 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 that spans a large area multiple trails would be formed to track each set of proximate accesses rather than forming an extended trail that could result in expensive disk head movements. Trail detection is of low cost because, when the timestamp of one block in a trail is specified, at most one timestamp of its following block is likely to be within T, in turn because the gap between

[1] In the implementation the prefetch streams are data structures embedded in the reclamation queue—in general blocks are never duplicated.

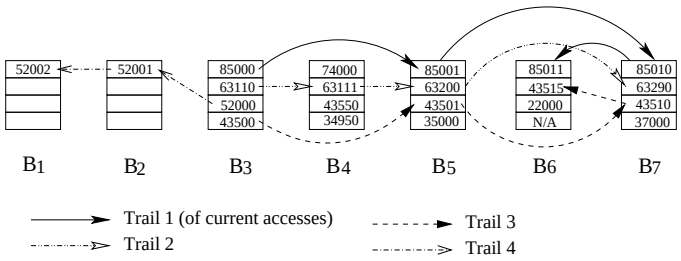

Fig. 3. **Access trails.** B_1 through B_7 are consecutive contiguous blocks in the block table. There are four trails starting from block B_3: one *current trail* and three *history trails*. Trail 1 (B_3, B_5, B_7, B_6) corresponds to the on-going continuous block accesses. This trail cannot lead to a sequence-based prefetch because B_4 is missing. It is echoed by two history trails: Trails 2 and 3, though Trail 1 only overlaps with part of Trail 2. In this example the timestamp threshold T is 256.

two consecutive timestamps of a block is usually very large (because they represent access, eviction, and re-access). Figure 3 illustrates four example trails on a segment of a block table.

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 can now take advantage of history information, if available, to carry out prefetching even if a pure sequence is not 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 so there is no need to explicitly record them.

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. At the same time we identify history trails consisting of blocks visited by the current trail in the same order. Referring to Figure 3, when the current trail (Trail 1) extends from $B_3(85000)$ to $B_5(85001)$, two history trails are identified: Trail 2 ($B_3(63110)$, $B_5(63200)$) and Trail 3 ($B_3(43500)$, $B_5(43501)$). When the current trail advances to block B_7 both Trail 2 and Trail 3 extend to it. However, only Trail 3 can match the current trail to B_6 while Trail 2 is broken at that block. The CPU time used for matching the trails is negligible relative to the disk-data prefetching time. For I/O-intensive workloads this prefetching time significantly overshadows the overhead of running DiskSeen.

3.4.3. History-aware Prefetching. Because of the strict matching requirement we initiate history-aware prefetching when 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. We then follow the history trails from B in the extent to obtain a set of blocks that the matched history trails are expected to visit. Suppose ts is a timestamp of block B that is used in forming a matched history trail, and T is the timestamp gap threshold. We then search the extent for blocks that contain a timestamp between ts and $ts + T$. Note that this criterion is stricter than the requirement that the gap between the timestamps of two adjacent blocks be less than T, which makes the search efficient and perform well in practice. We obtain the extension of the history trail in the extent by sorting the blocks in ascending order of their corresponding timestamps. We then prefetch the non-resident ones in the order of their LBNs and place them in the current window, similarly to 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, are each of size min (8 in our implementation). If the window size becomes less than min prefetching aborts. When the window size becomes larger than max (64 in our implementation), only the first max blocks are prefetched. If there are multiple matched history trails, we prefetch the intersection of these trails. By prefetching the intersection, rather than union, of multiple matched trails, DiskSeen reduces the risk of retrieving useless blocks, which would waste I/O bandwidth and potentially cause the prefetching to be disabled. 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 history-aware prefetching aborts, sequence-based prefetching is attempted.

3.5. Reducing the Penalty of Mis-prefetching

As described in Section 3.4.1, history-aware prefetching searches for and prefetches blocks in a small disk region. This both improves the efficiency of prefetching and limits the penalty of mis-prefetching. To further limit the number of mis-prefetching occurrences DiskSeen monitors the accuracy of prefetching *for each application* and stops prefetching for that application when the fraction of blocks that are mis-prefetched exceeds a predefined limit.

Mis-prefetching by history-aware prefetching is caused by changes in access patterns on specific disk regions. This may lead to a scenario where previously recorded access history only partially matches current accesses in a disk region so that history-prefetching is triggered but the predictions are mostly incorrect. For example, suppose that in a small disk region, application P_1 has accessed blocks (B_1, B_2, ..., B_{20}) in order. Then another application P_2 accesses blocks (B_1, B_2, B_3, B_4), but does not access blocks (B_5, B_6, ..., B_{20}). The matching of the first four blocks of these two trails activates history-aware prefetching on behalf of application P_2 according to the history trail left by P_1, and blocks (B_5, B_6, ..., B_{20}) are mis-prefetched. While occasional mis-prefetching only mildly degrades the benefit of prefetching, frequent mis-prefetching may cause substantial performance degradation as has been reported on the earlier design of DiskSeen [Ding et al. 2007]. Mis-prefetching can also occur with sequence-based prefetching, particularly when when a large value is selected for the timestamp gap threshold T. To address this issue DiskSeen provides a prefetching quality control mechanism.

In DiskSeen, blocks in the same prefetching window are close to each other on the disk and are prefetched together. For a prefetching window with mis-prefetched blocks, if any blocks in the window are eventually used by any application before they are evicted, the performance penalty associated with the mis-prefetched blocks can be regarded as small. This is because the additional cost to prefetch unneeded blocks in the window is small compared to the seek time to read the needed blocks in the window. However, if none of the prefetched blocks in a window are used, the cost for prefetching of the entire window of blocks becomes a performance penalty. For this reason DiskSeen monitors the number of windows containing only mis-prefetched blocks, called *mis-prefetched windows*, to determine if an on-going history-aware prefetching should continue. Specifically, for each application (process) DiskSeen periodically checks the ratio of mis-prefetched windows among total prefetch windows, and disables history-aware prefetching for the application if the ratio is greater than 50% in the most recent 512 prefetch windows. While prefetching is disabled, DiskSeen continues to create and maintain prefetch streams but does not perform actual disk accesses. When the ratio of mis-prefetched windows drops below 30% prefetching for the application is resumed.

3.6. 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 that 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 duration is the period when N_{pre_area} disk blocks are requested, where N_{pre_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 $|(H_{prefetch} - H_{cache})|/N_{pre\_area}$ is greater 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. To keep prefetching from being fully neutralized simply because caching is very effective with a large number of hits on the caching area, we set a minimum prefetching area size—by default 1/64 of the total system buffer memory.

3.7. DiskSeen for the Disk Array

To leverage I/O parallelism for higher disk throughput, a disk array, or RAID, is frequently adopted to replace a single disk as the storage device of its host computing system. Because the design of DiskSeen is based on the abstraction of a logical linear index in the form of LBN addresses, without reference to actual disk configuration, and a disk array exposes one consistent LBN space to the system, DiskSeen as designed for the single disk can perform its prefetching functionality on a disk array without changes.

The significant consequence of this underlying change in configuration is that the trails formed by DiskSeen may now span multiple disks because consecutive logical addresses are striped across the member disks of an array using a fixed striping unit size. Spanning trails across an array of disks can nullify DiskSeen's use of prefetching to improve I/O performance. When there exist concurrent requests that belong to different trails on a disk, it is critical for the disk to serve only one trail's requests at a time to avoid thrashing the disk head among trails on different disk regions. When DiskSeen carries out prefetching on each of its trails, it generates prefetch requests for each trail using its two-window-based policy. If the trails are on a single disk, the disk's scheduler should be able to serve requests of the same trail in a batch to exploit access spatial locality for good disk efficiency. However, if the trails are spread over multiple disks because of data striping, any given disk is unlikely to serve consecutive requests from the same trail. In a disk array, each disk has its own scheduler and its requests are independently scheduled. The subsequent requests may not be immediately scheduled at their respective disks as the disks can be serving requests of other trails. Because a prefetch window is shifted forward only when requests in its previous window have been served and the pre-loaded data start to be used, it can be a long time for the next request on the same trail to reach a disk. Consequently the scheduler must move the disk head to serve a request on a different trail. Such a scenario could occur on every disk in the array and for every trail, leading to disk head thrashing.

To address this problem we need to coordinate the scheduling of the disks so that (1) all prefetch requests generated by DiskSeen on a trail can be served together at each disk; and, (2) each disk can continuously serve a number of local requests that belong to the same trail before it moves its disk head to serve another trail's requests. To achieve this coordination we assign requests on the same trail a distinct common identifier and embed it in the requests. All requests that do not belong to any trail use a reserved identifier. Thus if there are N trails at a given time, the coordinated disk array has $N+1$ scheduling groups, each consisting of requests with a common identifier. We alternate the disk array's service among the scheduling groups, each for a time slice. In each time slice only the requests from the same scheduled group are served. To ensure that this coordination does not compromise the entire disk array's efficiency we make two adjustments. First, we monitor the system's I/O throughput during each group's time slice, and make the lengths of the groups' time slices proportional to their respective throughputs. Second, because a trail is formed by requests of strong locality, the throughput corresponding to serving a trail should be higher than that for requests not on trails. For this reason, if we observe that the former throughput is actually lower than the latter—for example, when requests in a prefetch window are served but requests in the next prefetch window have not yet been issued—we end the current time slice for the trail.

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 considerations, then the experimental results of micro-benchmarks and real-world applications.

4.1. Implementation Considerations

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 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 blocks, that cannot be seen via high-level abstractions, in LBN-ascending order. 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 from which 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.

DiskSeen for the disk array has an additional component built into the Linux software RAID md module. The md module is responsible for splitting a request into sub-requests and sending them to respective disks, where the regular disk scheduler does the local scheduling. By knowing to which trail requests belong, and which requests are not on any trails, this component schedules them to the individual disks during their respectively assigned time slices.

The prototype implementation of DiskSeen consists of approximately 1100 lines of code added to 15 existing files concerned with memory management, the file system, and block devices in the Linux kernel, and approximately 3800 lines of code in new files to implement the main algorithms.

4.2. Experimental Setup

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

4.3. Performance of One-run Benchmarks

We selected six benchmarks for measuring 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-world applications.

(1) *strided* is a program that reads a 1GB file in a strided fashion, reading every other 4KB of data from the beginning to the end of the file with a small amount of compute time after each read.
(2) *reversed* is a program that sequentially reads a 1GB file from its end to its beginning.
(3) *CVS* is a version control utility commonly used in software development environments. We ran *cvs -q diff*, which compares a user's working directory to a central repository, over two identical data sets with a 50GB disk space gap 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 *Q4* and *Q17* (described later) for the experiment.

For the analysis of experimental results across different benchmarks we use the source code tree of Linux kernel 2.6.11, of size approximately 236MB, as the data set for 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. Before every run the buffer cache is emptied to ensure that all blocks are accessed from disk. For most of the benchmarks the first runs with DiskSeen achieve substantial performance improvements because of

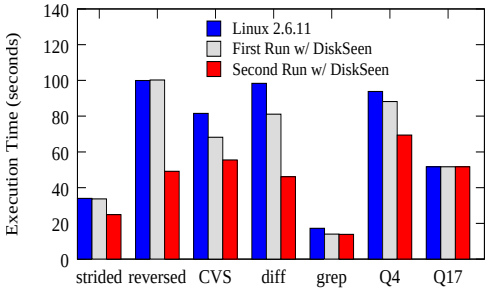

Fig. 4. Execution times of the six benchmarks, including two *TPC-H* queries, *Q4* and *Q17*.

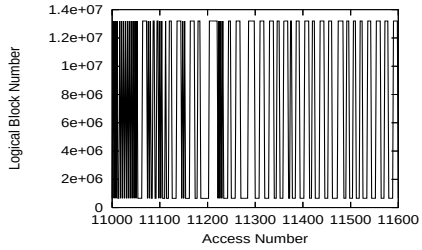

Fig. 5. A sample of CVS execution without DiskSeen.

DiskSeen's sequence-based prefetching, while the second runs enjoy further improvements because of the history information from the first run. 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 but with the on-disk data set and the 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* with some minor changes to several files. Following we analyze the performance results in detail for each benchmark.

Strided, reversed. With its strided access pattern no sequential access patterns can be detected for *strided* 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, showing that the overhead of DiskSeen is minimal. We have similar observations with *reversed*. With the history information, the second runs of the two benchmarks with DiskSeen show significant execution reductions: 27% for *strided* and 51% for *reversed*, because the histories correctly indicate the prefetchable blocks. It is not surprising to see a large improvement with *reversed*: without prefetching, reversed accesses can incur the time for a full disk rotation to service each request. DiskSeen prefetches blocks in large aggregates and requests them in ascending order of their LBNs, and each aggregate can be prefetched in one disk rotation. Note that the disk scheduler has little opportunity 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, 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 [Schmuck and Haskin 2002] and the MPI-IO standard [MPI-IO] provide special treatment for identifying and prefetching such sequences. If history access information is available, DiskSeen can handle these access patterns as well as more complex patterns without making the file systems or I/O interfaces more complex.

CVS, diff. As shown in Figure 4, DiskSeen significantly improves the performance of both *CVS* and *diff* on the first run, and more so on the second run. This is because the Linux source code

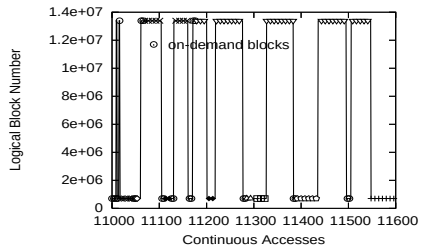

Fig. 6. A sample of CVS execution with DiskSeen, first run

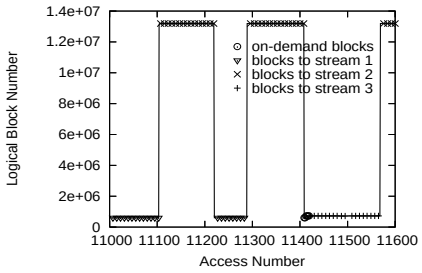

Fig. 7. A sample of CVS execution with DiskSeen, second run.

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 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 logical distance between accessed blocks, and thus (beyond a threshold) the necessity for disk head movement. Figures 6 and 7 show the same segment of the first and second runs of *CVS* with DiskSeen, respectively. While both sequence-based prefetching in the first run and history-aware prefetching in the second run significantly reduce disk head movement, history-aware prefetching is more effective than sequence-based prefetching. We observe some on-demand requests in the first run between prefetching requests, e.g., the on-demand requests near accesses 11015 and 11160. These requests are to read CVS versioning information and cannot be prefetched with sequence-based prefetching as will be explained later. In the second run these requests are serviced by history-aware prefetching. The figures also show that each sequence-based prefetching usually prefetches fewer blocks than a history-aware prefetch and thus incurs correspondingly more disk head movement. The figures also differentiate accesses of blocks that are fetched on demand or prefetched into different prefetch streams. It is evident that there are multiple concurrent prefetch streams, and most accesses are prefetches.

While the first runs of *CVS* and *diff* with DiskSeen reduce execution times by 16% and 18%, respectively, the second runs further reduce the times by 16% and 36%. For *CVS*, each directory in a *CVS*-managed source tree (i.e., working directory) contains a directory, named *CVS*, to store versioning information. When *CVS* processes each directory, it first checks the *CVS* subdirectory, and then examines 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 correspondingly disrupts sequence-based prefetching. In the second run, new prefetch sequences including the out-of-order blocks (that might not be purely sequential) are formed by examining history trails, yielding a performance inprovement. There are also many non-sequentialities in the execution of *diff* that prevent

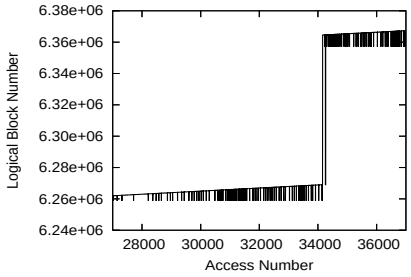

Fig. 8. A sample of *grep* execution without DiskSeen.

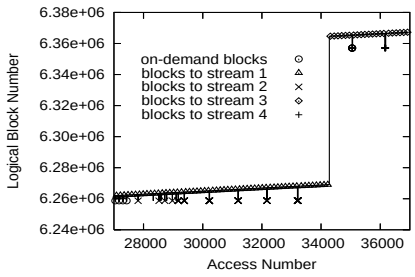

Fig. 9. A sample of *grep* execution with DiskSeen.

its first run from achieving the full performance potential. When we extract a kernel tar file, 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 break physical sequentiality to the point of making accesses in some directories close to random, resulting in *diff* having 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, and DiskSeen sends prefetch requests for these blocks in ascending order of their LBNs. In this way the mismatch is largely corrected for and performance is significantly improved.

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

Figure 8 shows a segment of execution of *grep* with the stock kernel. The two distinct regions correspond to two cylinder groups. In each cylinder group inode blocks are located at 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 the first run of *grep* with DiskSeen. By prefetching inode blocks most of the disk head movement is eliminated. The figure also shows that accesses to inode blocks and data blocks are from different prefetch streams. This is a consequence of independent prefetching in each localized area.

TPC-H: In this experiment query *Q4* performs a merge-join against tables *orders* and *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 matching records in table *lineitem* by referring to an index file. Because table *lineitem* was created by adding records generally corresponding 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 achieves a 26% reduction in execution time compared to the stock kernel.

DiskSeen does not show any performance improvements for query *Q17*. We carefully examined its access pattern and found that the query carried out index scans repeatedly on table *lineitem* and accessed it in a nearly random fashion, and with weak spatial locality, in many small disk areas. Sequence-based prefetching is not triggered because of the absence of long access sequences. Weak spatial locality also prevents history-aware prefetching from being triggered in most disk regions. When history-aware prefetching is activated, because data blocks of table *lineitem* are accessed multiple times with different access patterns, prefeching is misled to load data that are not immediately used and are evicted before they are requested. Instead of allowing the mis-prefetching to degrade I/O performance, as exhibited by the earlier version of DiskSeen (approximately 10% increase in execution time for *Q17*) [Ding et al. 2007], DiskSeen disables history-aware prefetching for *Q17* when mis-prefetching is detected.

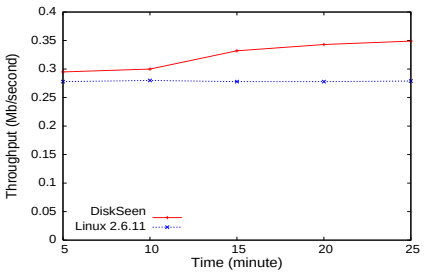

Fig. 10. LXR throughputs with and without DiskSeen.

4.4. Performance of Continuously Running Applications

For applications that are continuously running against the same set of disk data, previous disk accesses should serve as history access information to improve the I/O performance of current disk accesses. To test this hypothesis we installed a Web server running the general hypertext cross-referencing tool Linux Cross-Reference (LXR) [LXR], a tool 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 freetext 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' files 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 [WebStone] to generate 25 clients concurrently submitting freetext search queries. Each client randomly chooses a keyword from a pool of 50 keywords and sends it to the server, and sends its next query request once it receives the results of its previous query. We randomly select 25 Linux symbols from the file */boot/System.map* and another 25 frequently used OS terms such as "lru", "scheduling", and "page" as the pool of candidate query keywords. Each keyword is searched for in all three kernels. The performance metric we use is the throughput of the query system in MBit/sec, i.e., the bit rate of raw query results returned by the server. This metric is also used for reporting the WebStone benchmark results.

Figure 10 shows the LXR throughputs on the kernels with and without DiskSeen during an interval of execution. We make two observations. First, DiskSeen improves LXR's throughput by prefetching contiguous small files at disk level. Second, from the tenth minute to twenty-fifth minute of the execution, the throughput of LXR with DiskSeen monotonically increases while the throughput without DiskSeen remains flat. This demonstrates that DiskSeen can progressively improve performance as it accumulates history access information.

4.5. Interference by Noisy History

While well-matched history access information left by prior runs of applications is expected to provide accurate hints and improve performance, a reasonable speculation is that a misleading history could cause DiskSeen to prefetch unneeded blocks and degrade application performance. To investigate the interference effect caused by such *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 for a keyword in a Linux source code tree that is also used by *diff* to compare against another Linux source code tree. We know that *grep* scans files essentially in the order of their disk layout, while *diff* visits files in alphabetical order of directory/file names.

Table I. Execution times and hit ratios

Experiments	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	

Experiments	Hit Ratios (%)			
Linux	*diff*	100	*grep*	100
I	*diff*	*grep*	*diff*	*grep*
	84	92	97	98
II	*grep*	*diff*	*grep*	*diff*
	87	89	99	97
Linux	*grep*/*diff*		100	
III	*grep*/*diff*		90	
			94	
			96	
			98	
			97	

Execution times and hit ratios for *diff* and *grep* when they are alternately executed in different orders or concurrently, with DiskSeen, compared to the times and hit ratios for the stock kernel. The times reported are wall clock times. The hit ratio describes the percentage of prefetched data that have been actually requested by the programs before they are evicted.

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

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 a 16% degradation in performance in the first run of *grep* (16.3s vs. 14.0s) in experiment I, while it serendipitously improves the performance in the first run of *diff* by 17% (67.7s vs. 81.1s) in experiment II. The degradation in experiment I is due to the misleading history access information left by *diff* while running *grep*, wherein a matched history trail is found and history-based prefetching is activated. However, the matched history trail is broken when *diff* visits files in a different order. This causes DiskSeen to fall back to 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 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 I, along with their counterparts with 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 nondeterminism in process scheduling, access sequences cannot be exactly repeated between different runs. Each run of the two applications leaves different access trails over the shared blocks, which are noisy history that interferes with the current prefetching. The more runs there have been, and the more history is recorded, the easier it is to trigger an incorrect history-aware prefetching. This explains why the execution time of *grep* keeps increasing until the fifth run (we keep at most four timestamps for each block). Unlike *grep*, the execution time of *diff* in the second run is decreased by

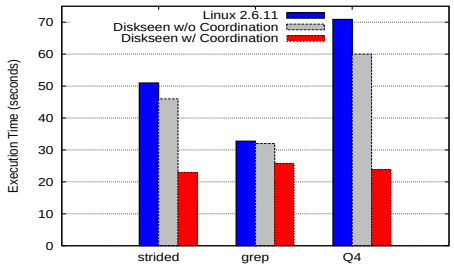

Fig. 11. Benchmark execution times on different Linux kernels.

23% (34.6s over 44.9s). This is because history-aware prefetching of the second source code tree, which is not touched by *grep*, is not affected by the interference.

In the experiments, DiskSeen does not disable an activated history-aware prefetching for either *grep* or *diff* as mis-prefetching is not sufficiently severe and the prefetching is still beneficial to both applications in spite of the presence of noisy history.

Note that the presented performance improvements for DiskSeen are not due to improved prefetch accuracy, which is quantified as prefetch hit ratio, or the percentage of prefetched data that are actually requested by the programs before they are evicted. In fact, in the experiments the stock kernel has comparable or even higher hit ratios than DiskSeen, as shown in Table I. In addition, for the experiments described in Section 4.3, hit ratios are larger than 97% for DiskSeen and almost 100% for the stock kernel's prefetching. For LXR in Section 4.4, the ratio varies between 72% and 77% for DiskSeen, and around 90% for the stock kernel. DiskSeen's performance advantages mainly come from two sources: (1) exploitation of more prefetching opportunities; and, (2) more efficient disk access when prefetching. Though the stock kernel can have almost 100% of its prefetched data used by programs, there are many requests whose data have not been prefetched and must be retrieved from the disk on-demand.

4.6. Performance of DiskSeen on the Disk Array

To evaluate the effect of coordinated disk scheduling on DiskSeen's performance on a disk array we use four identical disks to form a RAID 0 with a 64KB striping unit, and the three benchmarks *strided*, *grep*, and *Q4* from *TPC-H*. For each of the benchmarks we simultaneously run three instances, each accessing its own file. Figure 11 shows the execution times for each of the benchmarks with the stock kernel, and DiskSeen without and with coordinated scheduling. The results for DiskSeen are for the second runs of the benchmarks. With the use of disk array we expected that the effectiveness of prefetching enabled by DiskSeen would be amplified because the data preloading operation can be carried out by multiple disks in parallel. However, as Figure 11 shows, the improvements without scheduling coordination are relatively small, even compared to their respective one-disk counterparts (Figure 4). With coordination the potential of I/O parallelism with a disk array is unleashed and produces much greater improvements. The execution times of *strided*, *grep*, and *Q4* are reduced by 51%, 19%, and 60%, respectively, compared to DiskSeen without scheduling coordination. Because *strided* and *Q4* have more regular access patterns than *grep* from which DiskSeen can form longer prefetch sequences, they achieve higher improvement ratios. Figures 12 and 13 show disk accesses in terms of LBNs in the executions of *Q4* for DiskSeen with and without scheduling coordination. As shown in the graphs, without scheduling coordination at different disks each disk serves two trails of requests at almost the same time, indicating frequent disk head movement and low disk efficiency. In contrast, scheduling coordination allows a segment of each trail to be served efficiently before switching to another trail. We conclude that synchronizing the service of prefetch requests by DiskSeen is essential for its effectiveness in the disk array environment.

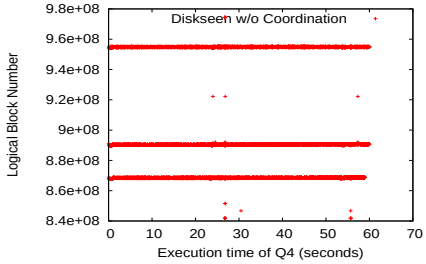

Fig. 12. A sample of *TPC-H Q4* execution using DiskSeen without disk coordination.

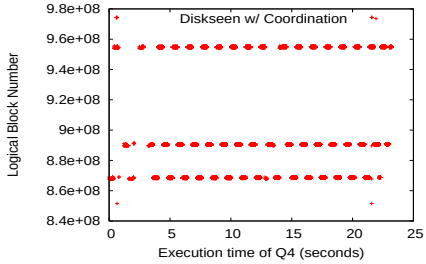

Fig. 13. A sample of *TPC-H Q4* execution using DiskSeen with disk coordination.

5. RELATED WORK

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

Intelligent prefetching algorithms: Prefetching is an active research area for improving I/O performance. Operating systems typically employ sophisticated heuristics to detect sequential block accesses to activate prefetching, as well as to adaptively adjust the number of blocks to be prefetched within the scope of a single file [R. Pai and Cao 2004; Smith 1978]. By working at the file abstraction and lacking mechanisms for recording historically detected sequential access patterns, these prefetch policies usually make conservative predictions and so may miss many prefetching opportunities [Papathanasiou and Scott 2005], including those that span files.

There are sequence-based prefetching schemes that are not limited to the file abstraction, and so can be used in data storage systems such as SAN systems. While predictions are made on the detected access sequences, the major issues of interest in these works are when and how much to prefetch, rather than what to prefetch. Specifically, the AMP scheme [Gill and Bathen 2007] is concerned with cache pollution and prefetch wastage. Using a formal analysis of the criteria necessary for optimal throughput, AMP adaptively changes prefetch timing and aggressiveness for the highest I/O efficiency. The STEP system groups blocks into different sequence streams or prefetch contexts [Liang et al. 2007]. Additionally, a cost-benefit model is built to determine the prefetch length by considering prefetching cost and hit ratio. Interestingly, the method of grouping accesses into different streams for prefetching has also been applied in the prefetching of data from memory [Diaz and Cintra 2009]. On the aforementioned issues, DiskSeen follows a more straightforward method usually adopted by existing general-purpose operating systems. These sophisticated designs complement DiskSeen's design. In contrast, DiskSeen's history-aware prefetching, which can capture additional prefetching opportunities and turn random accesses into sequential ones, still holds a unique advantage.

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 that track sequences of file access events and evaluate the probability of file occurrences in the sequences [Griffioen and Appleton 1994; Kroeger and Long 2001]. 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, making the approaches more suitable to Web proxy/server file prefetching than for general-purpose operating systems [Chen and Zhang 2003]. 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, or file metadata.

In a more recent work on a prefetching scheme for disk arrays, the ASP scheme is proposed for using more aggressive prefetching to leverage I/O parallelism in a disk array [Baek and Park 2008]. ASP seeks to avoid so-called independency loss in which a request must be served by multiple

disks. The scheduling coordination mechanism in DiskSeen does not require such independency for disk efficiency, which allows the prefetching scheme to use larger size and number of requests for greater I/O efficiency. The problem of lost disk efficiency in the use of the disk array has also been identified in design of the disk scheduler [Xu and Jiang 2011] and in the use of data servers managed by parallel file systems [Zhang et al. 2010]. In this paper we use a similar scheduling coordination design specifically for prefetch requests.

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 [Patterson et al. 1995; Tomkins et al. 1997]. 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 [Cao et al. 1996; Cao et al. 1994].

Compared with the method used in DiskSeen, application-hinted prefetching has limitations: (1) The requirements for generating detailed hints may be too burdensome for 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 [Vogels 1999]. Another study conducted at Microsoft Research shows a consistent result [Douceur and Bolosky 1999]. It can be challenging and burdensome for programmers to provide detailed hints, sometimes requiring restructuring of programs, as described in the context of TIP [Patterson et al. 1995; Tomkins et al. 1997]. The DiskSeen scheme, in contrast, is transparent to applications. (2) Sequentiality across files and disk data disk locations cannot be known by applications, but are important for prefetching small files. In our work this sequentiality can be easily detected and exploited.

Prefetching hints can also be automatically abstracted by compilers [Mowry et al. 1996] or generated by OS-supported speculative execution [Chang and Gibson 1999; Faser and Chang 2003]. Another interesting work is a tool called *C-Miner* [Li et al. 2004], 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 I/O layers for better utilization of hard disk is an active research topic. Most of the work focuses on using disk-specific knowledge for improving data placement on disk to 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 [McKusick et al. 1984; Ganger and Kaashoek 1997]. Track-extent-aware file systems exclude track boundary blocks from being allocated for better disk sequential access performance [J. Schindler and Ganger 2002]. 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 [Vogels 1999]), prefetching based on individual files cannot take full advantages of these techniques. 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 observed disk access patterns [Huang et al. 2005]. These replicas can be used to provide faster accesses of disk data. FS2 dynamically adjusts disk data layout to make it friendly to changing data request patterns, 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 to looking for free disk space to make replicas consistent with the access patterns in FS2, DiskSeen can be more flexible and responsive to changing access patterns.

6. CONCLUSIONS

DiskSeen addresses a pressing issue in prefetch techniques—how to exploit disk-level information so that effective disk performance is improved. By efficiently tracking disk accesses at the level of logical block number, 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 than file-level prefetching. DiskSeen overcomes limitations intrinsic to the file-level abstraction such as the difficulties in relating accesses across file boundaries or across lifetimes of open files, and the invisibility of file metadata. At the same time, DiskSeen complements rather than supplants high-level prefetching schemes, and is effective for both individual disks and disk arrays. 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%-60% for micro-benchmarks and real applications such as *grep*, *CVS*, *TPC-H*.

REFERENCES

BAEK, S. H. AND PARK, K. H. 2008. Prefetching with adaptive cache culling for striped disk arrays. In *ATC'08: Proceedings of USENIX 2008 Annual Technical Conference*. ATC'08.

BUTT, A. R., GNIADY, C., AND HU, Y. C. 2005. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *SIGMETRICS'05: Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems*. 157–168.

CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1996. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst. 14,* 4, 311–343.

CAO, P., FELTEN, E. W., AND LI, K. 1994. Application-controlled file caching policies. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference*.

CHANG, F. AND GIBSON, G. A. 1999. Automatic i/o hint generation through speculative execution. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*.

CHEN, X. AND ZHANG, X. 2003. A popularity-based prediction model for web prefetching. *IEEE Computer 36,* 3, 63–70.

DIAZ, P. AND CINTRA, M. 2009. Stream chaining: exploiting multiple levels of correlation in data prefetching. In *ISCA'09: Proceedings of the 36th annual international symposium on Computer architecture*.

DING, X., JIANG, S., CHEN, F., DAVIS, K., AND ZHANG, X. 2007. DiskSeen: exploiting disk layout and access history to enhance i/o prefetch. In *USENIX'07: Proceedings of the 2007 USENIX Annual Technical Conference*.

DOUCEUR, J. R. AND BOLOSKY, W. J. 1999. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM international conference on Measurement and modeling of computer systems*. 59–70.

FASER, K. AND CHANG, F. 2003. Operating system i/o speculation: How two invocations are faster than one. In *USENIX'03: Proceedings of the 2003 USENIX Annual Technical Conference*. 325–338.

GANGER, G. R. AND KAASHOEK, M. F. 1997. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. In *USENIX'97: Proceedings of 1997 USENIX Annual Technical Conference*.

GILL, B. S. AND BATHEN, L. A. D. 2007. AMP: Adaptive multi-stream prefetching in a shared cache. In *FAST'07: Proceedings of the 5th USENIX Conference on File and Storage Technologies*.

GRIFFIOEN, J. AND APPLETON, R. 1994. Reducing file system latency using a predictive approach. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference*.

HUANG, H., HUNG, W., AND SHIN, K. G. 2005. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *SOSP'05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*. 263–276.

J. SCHINDLER, J. L. GRIFFIN, C. R. L. AND GANGER, G. R. 2002. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*.

JIANG, S., DING, X., CHEN, F., TAN, E., AND ZHANG, X. 2005. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*.

KROEGER, T. M. AND LONG, D. D. E. 2001. Design and implementation of a predictive file prefetching algorithm. In *USENIX'01, Proceedings of 2001 USENIX Annual Technical Conference*. 105–118.

LI, Z., CHEN, Z., SRINIVASAN, S. M., AND ZHOU, Y. 2004. C-Miner: Mining block correlations in storage systems. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. 173–186.

LIANG, S., JIANG, S., AND ZHANG, X. 2007. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *ICDCS'07: Proceedings of 27th IEEE International Conference on Distributed Computing Systems*.

LXR. Linux cross-reference. *URL : http://lxr.linux.no/.*

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. *1984. A fast file system for unix.* ACM Trans. Comput. Syst. 2, *3, 181–197.*

MOWRY, T. C., DEMKE, A. K., AND KRIEGER, O. *1996. Automatic compiler-inserted i/o prefetching for out-of-core applications. In* OSDI '96: Proceedings of the second USENIX symposium on Operating systems design and implementation.

MPI-IO. *MPI-2: Extensions to the message-passing interface. URL:http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.*

PAPATHANASIOU, A. E. AND SCOTT, M. L. *2005. Aggressive prefetching: An idea whose time has come. In* Proceedings of the 10th Workshop on Hot Topics in Operating Systems.

PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. *1995. Informed prefetching and caching. In* SOSP '95: Proceedings of the 15th ACM symposium on Operating systems principles. *79–95.*

R. PAI, B. P. AND CAO, M. *2004. Linux 2.6 performance improvement through readahead optimization. In* Proceedings of the Linux Symposium.

SCHINDLER, J. AND GANGER, G. R. *2000. Automated disk drive characterization. In* SIGMETRICS '00: Proceedings of the 2000 ACM international conference on Measurement and modeling of computer systems. *112–113.*

SCHLOSSER, S. W., SCHINDLER, J., PAPADOMANOLAKIS, S., SHAO, M., AILAMAKI, A., FALOUTSOS, C., AND GANGER, G. R. *2005. On multidimensional data and modern disks. In* FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies.

SCHMUCK, F. AND HASKIN, R. *2002. GPFS: A shared-disk file system for large computing clusters. In* FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies.

SMITH, A. J. *1978. Sequentiality and prefetching in database systems.* ACM Trans. on Database Systems 3, *3, 223–247.*

TOMKINS, A., PATTERSON, R. H., AND GIBSON, G. *1997. Informed multi-process prefetching and caching. In* SIGMETRICS '97: Proceedings of the 1997 ACM international conference on Measurement and modeling of computer systems. *100–114.*

VOGELS, W. *1999. File system usage in windows NT 4.0. In* SOSP '99: Proceedings of the 17th ACM symposium on Operating systems principles. *93–109.*

WEBSTONE. *WebStone — the benchmark for web servers. URL: http://www.mindcraft.com/benchmarks/webstone/.*

XU, Y. AND JIANG, S. *2011. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In* FAST '11: Proceedings of the 9th USENIX Conference on File and Storage Technologies.

ZHANG, X., DAVIS, K., AND JIANG, S. *2010. IOrchestrator: improving the performance of multi-node i/o systems via inter-server coordination. In* SC '10: Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis.