

## A Survey of I/O Optimization Techniques

Kitsuregawa Laboratory

D1, 48-077409, Sven Groot

## Abstract

While CPU and memory speed advances at a steady pace, performance of disk-based storage systems continues to lag behind. This increasing gap in performance, coupled with rapid growth of the amount of data to be stored and processed, means that it is imperative that we find optimization techniques to minimize the latency induced by the mechanical components of disk drives.

In this paper, we present an overview of approaches for disk I/O optimization, including a high-level evaluation of currently employed approaches throughout the I/O path, and two prefetching systems, *Competitive Prefetching* and *DiskSeen*.

## 1 Introduction

As computers continue to get faster, one part that continues to lag behind are disk drives. While processors and memory continue to get faster, hard disks remain behind because of the relatively slow improvements in their mechanical components such as the platters and disk heads.

In addition, data sizes also continue to grow. Currently data sets often range in the terabytes, while petabyte scale systems are just around the corner. It has also been shown that reliability of systems is not improving, which means that systems that need to process large amounts of data will spend an increasing amount of time writing failure recovery information. I/O bandwidth is also a limiting factor in this situation. [9]

It should be clear that disk drive performance is becoming an increasingly limiting factor on data-heavy applications. Unless solid-state (flash), which may be slower at sequential reads and writes but is significantly better at random access than typical hard disk drives, memory becomes the primary data storage method this trend is unlikely to change.

Hard disks are semi-sequential storage devices. A schematic representation of a hard disk can be seen in Figure 1. A disk consists of a collection of

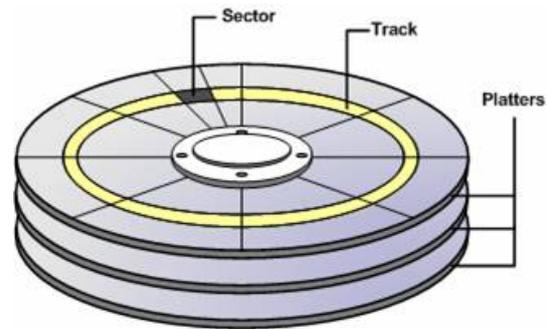


Figure 1: A schematic representation of a hard disk drive

circular platters. Data is stored in sectors, which are laid out along circular tracks on the platters. The disk head is placed on a track and reads the data along the track sequentially. In order to read a specific sector, the head has to move to the track containing that sector. The time this takes is called the *seek delay*. Then it has to wait until disk rotation moves the sector under the head, which is called *rotational latency*.

Seek delay and rotational latency present the largest factors in the performance bottleneck of disk drives; reading or writing data sequentially along a track is considerably faster than random access. For this reasons, I/O optimization approaches aim to reduce the head movement as much as possible, and to read as much data as is feasible during the sequential phase.

In this paper, we will look at several approaches that seek to reduce the I/O performance bottleneck. In section 2, we will look at the I/O path and the optimization opportunities that exists at various levels of this path. We will also explain several disk scheduling methods currently used. In sections 3 and 4 we will look at two prefetching methods, *Competitive Prefetching* and *DiskSeen*.

## 2 The I/O Path

In order to close the gap between disk performance and memory performance, the I/O path has grown

System	Dual Intel Xeon CPU 2.40GHz, 1GB Memory, LSI Fusion MPT SCSI Controller
OS	Gentoo Linux 2.6.16-git11
Application	Postmark / Linux kernel build
File System	Ext2, Ext3, ReiserFS, XFS
IO Scheduler	No-Op, Deadline, CFQ, Anticipatory
Disk Drive	18 GB/15Krpm, 146 GB/15Krpm, 300 GB/10Krpm

Table 1: Measurement environment

Work load	File Size	Work Set	File size	No. of Files	Transactions
SS	Small	Small	9-15 KB	10,000	100,000
SL	Small	Large	9-15 KB	200,000	100,000
LS	Large	Small	0.1-3MB	1,000	20,000
LL	Large	Large	0.1-3MB	4,250	20,000

Table 2: Postmark workloads

long and complex. When looking at the I/O stack in current systems, we can identify several different layers: the filesystem, the disk device driver, the disk (RAID) array and the disk drive itself. Riska et al present an evaluation of I/O optimization throughout the I/O path in [7].

The main methods for I/O optimization discussed here are request scheduling and request merging. In the former approach, requests are reordered to minimize seek time and rotational latency. With the latter approach, sequential requests are merged to take advantage of the high speed of sequential reads or writes on a track.

The measurement environment used is described in table 1. They used two applications to evaluate performance; the first is the Postmark file system benchmark [4], with the workloads described in table 2. The second application is a Linux kernel build.

In the following sections, we will look at each layer of the I/O path individually.

## 2.1 File System Level

At the highest layer is the filesystem. The filesystem is responsible for allocating blocks on the disk and for locating the blocks and associated metadata that belong to files. The following common filesystems were evaluated:

- **Ext2** uses *cylinder groups* for data placement and single, double or triple indirect metadata blocks.
- **Ext3** derives from Ext2, but adds a journal for consistency and reliability.
- **Reiser** uses  $B^+$  trees for metadata and also has a single contiguous journal.

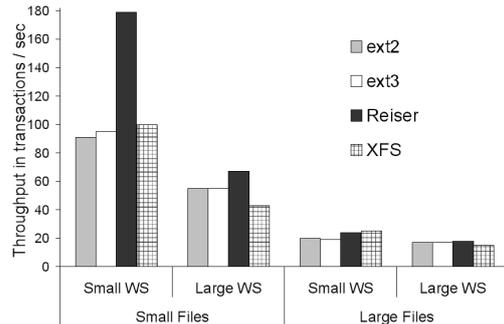


Figure 2: Postmark throughput for the different filesystems

- **XFS** uses a single contiguous journal as well, and allocation groups and extent-based  $B^+$  trees.

Figure 2 shows the Postmark throughput for the filesystems under the different workloads. This is using Anticipatory scheduling on the 18GB disk.

Under the more sequential large files workloads, there is little difference between the different filesystems. This is because there is little room for optimization under these conditions. Under the small files workloads we see that Reiser offers the best throughput.

Measurement has shown that the amount of data read and written is roughly equivalent on all filesystems; which the exception of XFS, none of the filesystems appear to have much overhead from keeping the journals. Because the disk workload is no different, the difference in throughput is mainly attributed to request size. Large sequential requests are more efficient than many small consecutive ones, and Reiser consistently gives larger read requests than the other filesystems. For write requests, Reiser sends constant 4KB write requests to the device driver, as does Ext3, but the outgoing requests from the driver to the disk are consistently larger for Reiser, indicating that Reiser is better at block allocation than the other filesystems.

## 2.2 Device Driver Level

One level lower we find the I/O device driver. This driver receives I/O requests from the filesystem and is responsible for relaying them to the disk or array controller.

At the device driver level, the main method for optimization is request reordering. The driver gets the various requests from the filesystem and tries to order them in an efficient fashion. Other optimizations such as the prefetching techniques mentioned

later in this paper will typically also be at this level.

Disk scheduling algorithms often work in elevator fashion, such as the SCAN or Circular SCAN [13] algorithms. These algorithms scan the disk in a direction, servicing all the requests in its path, then reverse and continue.

Another class of algorithms is Shortest Positioning Time First (SPTF), sometimes also called Shortest Seeking Time First. In this case the next request serviced is the one with the shortest seek distance from the current head position, regardless of the head direction. The problem with this approach is that it can lead to starvation, where requests on distant areas of the disk never get serviced. A variant called Aged-SPTF can be used to correct this. This scheme works like SPTF but it keeps track of the age of requests, servicing them regardless of seek time when their age exceeds a certain limit.

When performing disk scheduling it is possible for a phenomenon known as *deceptive idleness* to occur. It is often the case that computer programs will issue sequential I/O requests synchronously. This means that they will issue a request, wait until it is complete, do some processing and then issue the next request. In a *work-preserving* scheduler, which will immediately schedule a new request if one is available, this means that the second request from the process can come too late. The scheduler will already have picked a different request, making it impossible to exploit the locality of the two synchronous requests.

A non-work-preserving scheduler will in certain circumstances wait for a short time before scheduling a new request, even if there are outstanding requests available, in the hope a better candidate will be presented in the mean time. Anticipatory scheduling [3] is such a technique, and can work complimentary to most other disk scheduling algorithms.

The Linux kernel currently comes with four scheduling algorithms [10, 6], listed below.

- **No-Op** is essentially a first-come first-serve algorithm that performs no optimizations. Requests are serviced in the order they arrive.
- **Deadline** is an Aged-SPTF-like algorithm. Requests are ordered based on seek time, unless a read has been waiting for more than 600ms or a write has been waiting for more than 5 seconds.
- **Anticipatory** is based on the Deadline scheduler, but will wait for up to 6ms for a better request. This is currently the default scheduler in Linux.

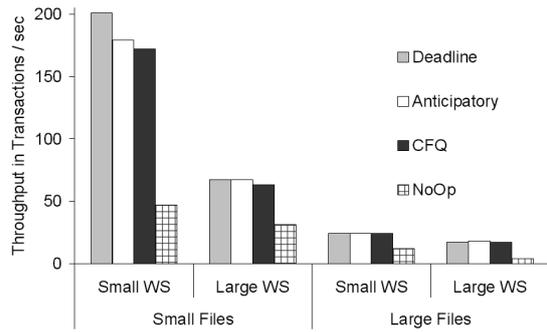


Figure 3: Postmark throughput for the different schedulers

- **CFQ** is an elevator-based scheduler that attempts to give each process equal scheduling time in a round-robin fashion.

In figure 3 you can see the scheduler throughputs for the Postmark workloads from table 2. This experiment used the Reiser file system and the 18GB disk, fixing all parameters except the scheduler.

It should come as no surprise that all the algorithms outperform No-Op, proving that adaptive scheduling is better than FCFS. We can also see that again most opportunity for optimization exists in the small files workload, rather than the more sequential large files workload. The Deadline scheduler performs the best in this case, indicating that deceptive idleness was not a large factor in this particular configuration.

The maximum relative benefit that can be gained from changing only the scheduler is only 20%, whereas the results of figure 2 indicate that the filesystems can yield as much as 80% benefit in the case of Reiser.

All of the schedulers perform request merging. Even No-Op is capable of doing request merging, although it requires that sequential requests arrive consecutively for them to be merged. Anticipatory is the most aggressive when it comes to merging, sometimes even idling the disk while waiting for a sequential request. The effects of this can be seen in figure 4, which depicts the average request size. As can be seen, the size of the requests going to the disk is larger than those received from the file system in all cases.

## 2.3 Disk Drive Level

The last level evaluated is the disk drive itself. Although the higher levels are the most effective place for optimizations such as request merging, the best place to do request reordering is at the disk drive

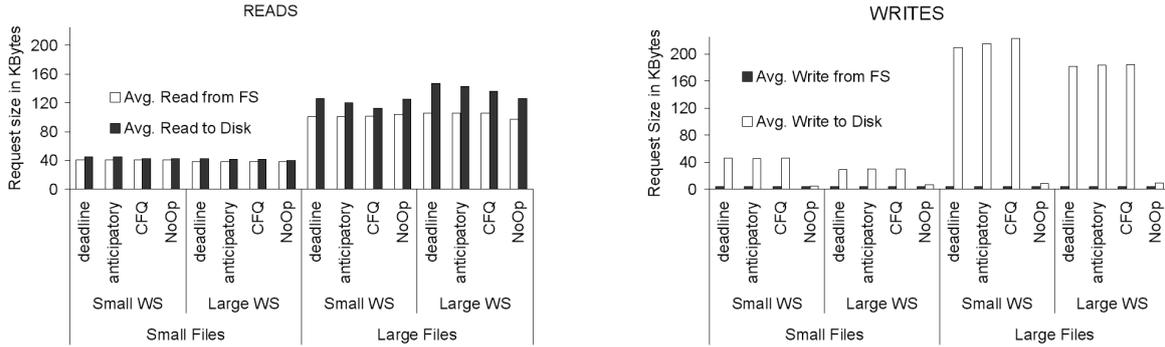


Figure 4: Request size for the scheduling algorithms for reads (left) and writes (right).

	ST318453LC	ST3146854LC	ST3300007LC
Capacity	18 GB	146 GB	300 GB
RPM	15,000	15,000	10,000
Platters	1	4	4
Linear density	64K TPI	85K TPI	105K TPI
Avg seek time	3.6/4 ms	3.4/4 ms	4.7/5.3 ms
Cache	8 MB	8 MB	8MB

Table 3: Disk drive specifications

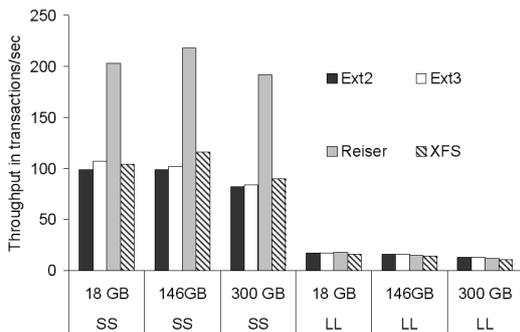


Figure 5: Postmark throughput for the three disks.

itself [11], because this is the only location where truly accurate information about disk geometry and current head position is available. Disks can also perform caching and other optimizations.

The specifications of the three Seagate disks used for the evaluation are listed in table 3. Although one would expect the seek time to drop for the 300GB disk because of its higher density (the tracks are closer together), the platters of the 300GB disk are bigger which means the disk head arm is bigger and therefore heavier.

The throughput for the disks with varying filesystems and anticipatory scheduling is shown in figure 5. The 146GB gets the highest throughput in all conditions. The 300GB disk loses due to its higher seek time, despite the higher linear density. Similar results are found when keeping the file system constant and instead varying the scheduler.

Computing the optimal request schedule is NP-complete and uses the head positioning time rather than the seek time [1], which is only available at the disk drive and not at the device driver. Modern disk drives which have more on-board CPU and memory can use advanced heuristics to find a good schedule, and can handle deeper requests queues.

Another factor at the disk level is write cache. A disk can use either write-back cache, where the application is notified immediately of request completion when the request is placed in the write cache, or write-through cache, where the request is not completed until it has actually been written to disk. The first method offers performance benefits at the cost of reliability since data not yet written to disk is lost in case of power failure, unless non-volatile memory is used for the write cache. Experimentation has shown however that the I/O optimization opportunities throughout the path combined with disk-level queuing can effectively close the gap between write-back and write-through schemes.

### 3 Competitive prefetching

Now that we have seen an overview of several disk optimization techniques, let us continue to look at a specific type of optimization, prefetching. Prefetching is the practice of attempting to predict future I/O requests from an application and performing those ahead of time. This is done particularly with sequential I/O requests, because as we have seen these are the fastest.

One of the problems prefetching aims to solve is when requests for a sequential I/O stream are interleaved with other I/O requests. This can cause the deceptive idleness problem described earlier. Switching between the I/O requests of two streams is called an I/O switch from here on.

Prefetching can serve as an alternative method

to Anticipatory Scheduling to solve the deceptive idleness problem. Both methods have their advantages and disadvantages. Prefetching is difficult because predicting future workload is difficult, especially without help from the application, and too aggressive prefetching increases the chance of doing unnecessary work which can decrease effective I/O bandwidth. Anticipatory Scheduling works on the process level so is less effective if two streams are used by the same process (e.g. the *diff* utility which works on two files simultaneously) or if it cannot see the context of the I/O requests, for instance with remote file servers or if the requests come from a virtual machine.

Li et al propose a method for dealing with concurrent sequential stream in [5] which they call Competitive Prefetching. We will describe this method in the following sections.

### 3.1 2-Competitive Prefetching

There are two competing metrics involved when doing prefetching: too conservative prefetching leads to a high I/O switch overhead, while too aggressive prefetching may lead to wasted I/O bandwidth due to fetching of unnecessary data. Competitive Prefetching is called so because it attempts to balance this tradeoff.

The optimal situation can be achieved with *a-priori* information about the streams; in this case the amount of prefetched data is precisely length of each stream, and there is only one switch. Assuming a total stream size of  $S_{tot}[i]$  for stream  $i$  and a sequential data transfer rate of  $R_{tr}[i]$  (modern disks often have different transfer rates for different sections of the disk, so we cannot assume the same transfer rate for all streams) and an I/O switch cost of  $C_{switch}[i]$ , the optimal total cost of accessing  $N$  streams is:

$$C_{total}^{opt} = \sum_{1 \leq i \leq N} \frac{S_{tot}[i]}{R_{tr}[i]} + C_{switch}[i]$$

In practice we do not know the total stream size. Assuming a constant prefetching depth of  $S_p[i]$  and a total stream length of  $S_{tot}[i]$  for stream  $i$ , the number of prefetches required for the stream is  $\lceil \frac{S_{tot}[i]}{S_p[i]} \rceil$ . Let the cost for switch number  $j$  while reading stream  $i$  be  $C_{switch}[i, j]$ , then the total switch cost for stream  $i$  is:

$$C_{tot\_switch}[i] = \sum_{q \leq j \leq \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil} C_{switch}[i, j]$$

The time wasted reading unnecessary data is bounded by the cost of the last prefetch:

$$C_{waste}[i] \leq \frac{S_p[i]}{R_{tr}[i]}$$

Therefore the total disk resource (in time) con-

sumed by the access of all streams is constrained by:

$$\begin{aligned} C_{total} &= \sum_{1 \leq i \leq N} \frac{S_{tot}[i]}{R_{tr}[i]} + C_{waste}[i] + C_{tot\_switch}[i] \\ &\leq \sum_{1 \leq i \leq N} \left( \frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} + \sum_{q \leq j \leq \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil} C_{switch}[i, j] \right) \end{aligned}$$

The average switch time is dependent on two factors, the seek time and the rotational delay. It has been found that the expected average I/O switch time does not depend on the chosen prefetch scheme, which allows us to simplify the equations for the optimal total cost and the optimal cost:

$$C_{total}^{opt} = \sum_{1 \leq i \leq N} \frac{S_{tot}[i]}{R_{tr}[i]} + C_{switch}^{avg}$$

$$\begin{aligned} C_{total} &= \\ &\leq \sum_{1 \leq i \leq N} \left( \frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \\ &\quad \sum_{1 \leq i \leq N} \lceil \frac{S_{tot}[i]}{S_p[i]} \rceil \cdot C_{switch}^{avg} \\ &< \sum_{1 \leq i \leq N} \left( \frac{S_{tot}[i]}{R_{tr}[i]} + \frac{S_p[i]}{R_{tr}[i]} \right) + \\ &\quad \sum_{1 \leq i \leq N} \left( \frac{S_{tot}[i]}{S_p[i]} + 1 \right) \cdot C_{switch}^{avg} \end{aligned}$$

Using these equations, we can deduce the following:

$$\text{if } S_p[i] = C_{switch}^{avg} \cdot R_{tr}[i], \text{ then } C_{total} < 2 \cdot C_{total}^{opt}$$

In other words, we can guarantee that a prefetching scheme using this prefetching depth will have a disk resource consumption at most twice that of the optimal offline solution. It can be shown that this result is optimal; it is not possible to have a scheme with better performance guarantees.

In order to prevent prefetching from degrading performance in random-access scenarios, competitive prefetching uses a *slow start*. Prefetching begins with a low depth, and as a sequential stream is detected the depth is increased gradually until the desired competitive depth.

### 3.2 Results

Competitive benchmarking was tested using several microbenchmarks as well as real applications. Some of the results are shown in figure 6, which shows the performance of the default

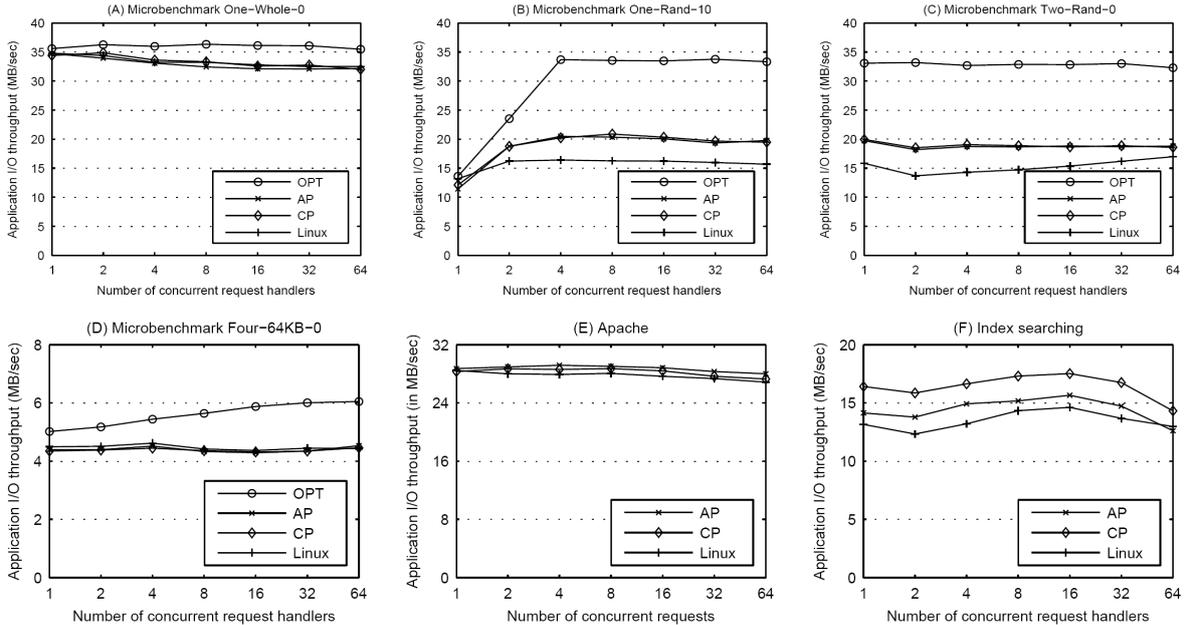


Figure 6: Performance of competitive scheduling for several microbenchmarks and two real applications.

Linux settings, aggressive prefetching (AP), competitive prefetching (CP) and, for the microbenchmarks only, the clairvoyant optimal prefetching (OP) strategy.

It can be seen that competitive prefetching is no worse than aggressive prefetching, and in some cases even better. It outperforms the original Linux kernel by as much as 53% and is at most 42% worse than the optimal strategy in the tested cases. It has also been found that competitive prefetching offers significant benefits over Anticipatory scheduling in the aforementioned cases where Anticipatory scheduling falls short (those results are not shown here to preserve space).

## 4 DiskSeen

Current prefetching algorithms usually work on the logical file level, which makes sense because application requests are done at that level. This approach does however have several limitations: sequential file blocks may not be laid out sequentially on disk; recording access history information at the file level may be inconvenient, which causes many prefetch schemes to have only a shallow history so they must prefetch conservatively [12]; inter-file sequentiality cannot be exploited; and file-system metadata blocks cannot be prefetched.

Ding et al [2] propose a prefetching scheme that works on the level of disk blocks instead of the filesystem level, called *DiskSeen*. *DiskSeen* can

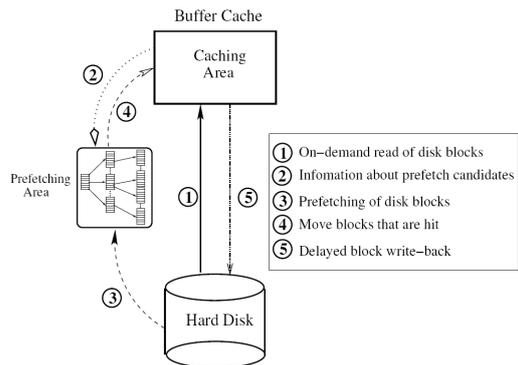


Figure 7: The design of *DiskSeen*.

work next to existing file-level prefetching schemes.

### 4.1 *DiskSeen* Design

Because more detailed physical geometry information is often not available, *DiskSeen* uses Logical Block Numbers (LBNs) to track disk accesses. Disk manufacturers attempt to ensure that consecutive LBNs maps to consecutive physical disk sectors [8].

*DiskSeen* keeps a counter that is incremented every time a disk block is accessed. The current counter value when a block is requested is stored in the block table.

*DiskSeen* distinguishes between I/O requests from applications (on-demand requests) and those from file-level prefetchers. Only on-demand re-

quests are used when making prefetching decisions.

DiskSeen coordinates with file-level prefetchers in one more way. When a prefetching request comes in from a high-level prefetcher, this block is marked *prefetched* in the block table. Only when this block is later used by an on-demand request is this status removed. If a prefetch request for a block comes in and that block already has the *prefetched* status, DiskSeen ignores the request because a previous prefetch of that block was not followed by a use of that block proving the prefetch prediction incorrect. In this manner, DiskSeen can correct some of the mistakes made by higher-level prefetchers.

The access of each block from an on-demand or file-level request is recorded in the block table. Prefetching is activated when a sequence of  $K$  consecutive blocks is detected, where  $K$  is 8 in the prototype setup. To detect this, the access indices of consecutive blocks are examined; if the blocks are sequentially accessed, the access indexes should be uniformly ascending (they need not be strictly consecutive if access of this stream is interleaved with other I/O accesses).

When a sequence is detected, two windows are created. Eight blocks immediately ahead of the sequence are prefetched into the current window, and the following 8 blocks into the readahead window. The number of  $f$  blocks in the current window that are hit by high-level requests is monitored. If the blocks in the readahead window are requested, a new readahead window of size  $2f$  is created, and the existing readahead window becomes the current window up to a predefined maximum size. Additionally, if the value of  $2f$  falls below a certain threshold prefetching is cancelled.

In addition to using only current access information, DiskSeen can also use history information. To do this we describe the concept of a trail of blocks which is a set of blocks  $(B_1, B_2, \dots, B_n)$  such that  $0 < access\_index(B_i) - access\_index(B_{i-1}) < T$  and  $|LBN(B_i) - LBN(B_1)| < S$ . In other words, a trail is a sequence of blocks that have been accessed at an interval of at most  $T$  and that are at most  $S$  blocks apart.

When a block is accessed, DiskSeen looks at previous access indices for this block and surrounding blocks and tries to find old trails leading from that block that match the current trail (which need not be a strict sequence). By following matching trails we can identify blocks that are likely candidates for prefetching. History trails can also run backwards.

## 4.2 Evaluation

Figure 8 shows the result of experimental evaluation of DiskSeen against several benchmarks.

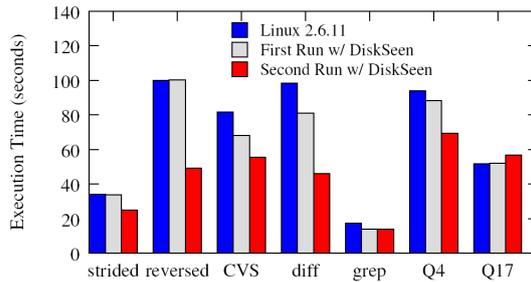


Figure 8: Execution time of several benchmarks with DiskSeen, including two TPC-H queries, Q4 and Q17.

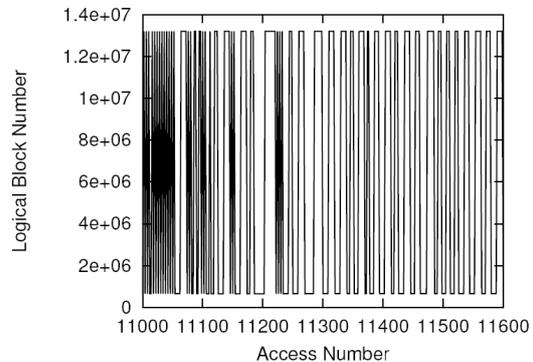


Figure 9: Disk head movements for CVS without diskseen.

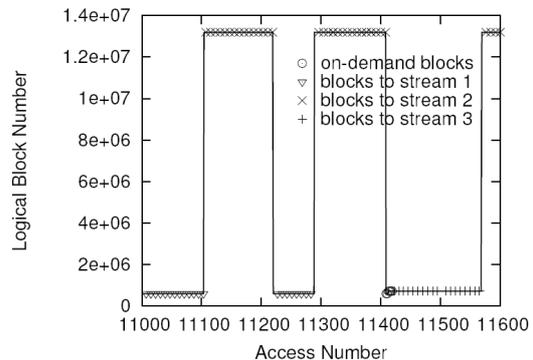


Figure 10: Disk head movements for CVS with diskseen.

In the **strided** benchmark, a 1GB file is read in strided fashion, i.e. every other 4KB is read. This means no sequences are found so the first run with DiskSeen does not improve performance. The fact that it doesn't reduce performance either proves DiskSeen has low overhead. During the second run, history streams are used to find the blocks to prefetch, increasing performance. The **reverse** benchmark reads the 1GB file backwards leading to much the same results.

Large improvements can be seen in the **CVS** and **diff** benchmark, which use the well-known CVS and diff utilities against the Linux kernel. This is because the Linux kernel consists of many small files making file-level prefetching ineffective. In addition, DiskSeen drastically reduces the amount of I/O context switches (and thus head movements) needed, as can be seen from figures 9 and 10.

In the original run of the **grep** benchmark, which operates on a single directory rather than separate disk regions like CVS and diff, most of the disk head movements are caused by reading the inode blocks. With DiskSeen, these are prefetched and most of the disk head movements disappear.

The **TPC-H** benchmarks run business oriented queries against a database. Query 4 performs a merge-join against two tables using an index, allowing DiskSeen to identify sequences in each small disk section.

The performance reduction in the second run of query 17 is caused by the near-random access of one of the tables in that query. Because of the large access index gap  $T$  used for identifying history trails, DiskSeen wastes I/O doing unnecessary prefetches. Reducing the access index gap also reduced the performance degradation, confirming this assertion.

## 5 Conclusion

We have looked at several approaches to I/O optimization, including a general overview of optimizations throughout the I/O path and two specific prefetching schemes. We have seen that there are many different ways in which I/O performance can be optimized and in fact, we have seen only the tip of the iceberg here, as a large variety of different approaches exists.

As the gap between the performance of disk and other components such as CPU and memory continues to grow, it should be clear that the importance of effective disk scheduling and other optimizations will grow equally.

## References

- [1] Andrews, Bender, and Zhang. New algorithms for the disk scheduling problem. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.
- [2] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance i/o prefetch. In *USENIX Annual Technical Conference*, 2007.
- [3] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.
- [4] Katcher J. Postmark: A new file system benchmark. Technical Report 3022, Network Appliances, Oct. 1997.
- [5] Chuanpeng Li, Kai Shen, and Athanasios E. Papatheasiou. Competitive prefetching for concurrent sequential i/o. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 189–202, New York, NY, USA, 2007. ACM Press.
- [6] S. Pratt and D. Heger. Workload dependent performance evaluation of the linux 2.6 i/o schedulers. In *Proceedings of the Ottawa Linux Symposium 2004*, July 2004.
- [7] Alma Riska, James Larkby-Lahet, and Erik Riedel. Evaluating block-level optimization through the io path. In *USENIX Annual Technical Conference*, 2007.
- [8] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 259–274, Berkeley, CA, USA, 2002. USENIX Association.
- [9] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *J. Phys.: Conf. Ser.*, 78(1):012022+, 2007.
- [10] Patricia Teller Seetharami Seelam, Rodrigo Romero. Enhancements to linux i/o scheduling. In *Proceedings of the Linux Symposium, Volume Two*, July 2005.
- [11] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.
- [12] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3):223–247, 1978.
- [13] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 241–251, Nashville, TN, USA, 16–20 1994.