ON-DISK SEQUENCE CACHE(ODSC): USING EXCESS DISK CAPACITY TO INCREASE PERFORMANCE

by

Christopher R. Slade

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2005

Copyright © 2005 Christopher R. Slade

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Christopher R. Slade

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	J. Kelly Flanagan, Chair
Date	Eric G. Mercer
Date	Christophe Giraud-Carrier

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Christopher R. Slade in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

J. Kelly Flanagan
Chair, Graduate Committee

Accepted for the Department

Paris K. Egbert
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

ON-DISK SEQUENCE CACHE(ODSC): USING EXCESS DISK CAPACITY TO INCREASE PERFORMANCE

Christopher R. Slade

Department of Computer Science

Master of Science

We present an on-disk sequence cache (ODSC), which improves disk drive performance. An ODSC uses a separate disk partition to store disk data in the order that the operating system requests it. Storing data in this order reduces the amount of seeking that the disk drive must do. As a result, the average disk access time is reduced. Reducing the disk access time improves the performance of the system, especially when booting the operating system, loading applications, and when main memory is limited.

Experiments show that our ODSC speeds up application loads by as much as 413%. Our ODSC also reduces the disk access time of the Linux boot by 396%, and speeds up a Linux kernel make by 28%. We also show that an ODSC improves performance when main memory is limited.

ACKNOWLEDGMENTS

I would like to thank Kelly Flanagan for all of the support, guidance and advice through the entire process. I could not have done it without him. I would also like to thank the members of the lab, Myles Watson, Elizabeth Sorenson, and Hyrum Carroll, for all of the help along the way.

I also appreciate my family and all of their support. I would not have made it here without them.

Contents

Abstract ix

List of Tables xv

List of Figures xvii

1 Introduction 1

1.1 Motivation 1

1.2 Thesis Statement and Layout 2

2 Foundational Material 5

2.1 Disk Drive Fundamentals 5

2.1.1 Mechanical Components 5

2.1.2 Disk Drive Operations 6

2.1.3 Operating System Interactions 8

2.2 Dynamic Sequence Detection 8

2.3 Prior Work 9

2.4 Disk Caches 10

2.5 Summary 11

3 On-Disk Sequence Cache (ODSC) 13

3.1 Overview 13

3.2 Trace Collection 14
3.3 Disk Request Translation 16
3.4 Kernel Modifications 16
3.5 Disk Caching Module (DCM) 18
3.6 Disk Caching Daemon (DCD) 19
3.7 Summary 20

4 Experiments and Results **21**

4.1 Overview 21
4.2 Application Loading 22
4.2.1 Application Loading Results 22
4.3 Linux Booting 24
4.3.1 Linux Booting Results 24
4.4 Limited Memory Experiment 28
4.4.1 Limited Memory Experiment Results 29
4.5 Summary 29

5 Conclusion and Future Work **33**

5.1 Conclusion 33
5.2 Future Work 33

A Additional Graphs **35**

Bibliography **49**

List of Tables

3.1 DCM IOCTL Functions 19

4.1 Application Loading Results 23

4.2 Linux Booting Results 28

List of Figures

1.1 Access Time and Cost of SRAM, DRAM, and Disks 2

2.1 Mechanical Components of a Disk Drive 6

3.1 Design of the On-Disk Sequence Cache 14

3.2 Trace collection 15

3.3 Disk Request Translation 17

4.1 Seek Distance of Vi with and without the ODSC 25

4.2 Disk Access Time of Vi 26

4.3 Seek Distance vs. Time of Vi 27

4.4 Miss Rate of the Memory Disk Cache 30

4.5 Average Disk Access Times 31

A.1 Seek Distance of Gimp without the ODSC 36

A.2 Seek Distance of Gimp with the ODSC 36

A.3 Disk access times of Gimp without the ODSC 37

A.4 Disk access times of Gimp with the ODSC 37

A.5 Seek Distance vs. Time of Gimp without the ODSC 38

A.6 Seek Distance vs. Time of Gimp with the ODSC 38

A.7 Seek Distance of Emacs without the ODSC 39

A.8 Seek Distance of Emacs with the ODSC 39

A.9 Disk Access Times of Emacs without the ODSC 40
A.10 Disk Access Times of Emacs with the ODSC 40
A.11 Seek Distance vs. Time of Emacs without the ODSC 41
A.12 Seek Distance vs. Time of Emacs with the ODSC 41
A.13 Seek Distance of the Linux Boot without the ODSC 42
A.14 Seek Distance of the Linux Boot with the ODSC 42
A.15 Disk Access Times of the Linux Boot without the ODSC 43
A.16 Disk Access Times of the Linux Boot with the ODSC 43
A.17 Seek Distance vs. Time of the Linux Boot without the ODSC 44
A.18 Seek Distance vs. Time of the Linux boot with the ODSC 44
A.19 Seek Distance of a Kernel Make without the ODSC 45
A.20 Seek Distance of a Kernel Make with the ODSC 45
A.21 Disk Access Times of a Kernel Make without the ODSC 46
A.22 Disk access times of a Kernel Make with the ODSC 46
A.23 Seek Distance vs. Time of a Kernel Make without the ODSC 47
A.24 Seek Distance vs. Time of a Kernel Make with the ODSC 47

Chapter 1

Introduction

Researchers have sought to increase the modern computer's performance since its advent. Improving computer performance is a complex task that focuses on many different components. These components include hardware components such as processors, memory, and disk drives, and software components like operating systems, drivers, and application programs. By improving a component's performance, the researcher hopes that the overall performance of the computer increases so that people can accomplish computational tasks in less time. The process of improving system performance includes many trade-offs.

One trade-off made by researchers is trading space for speed. In software, a programmer may decide to use more space in memory by using a tree structure instead of an array in order to achieve better performance on operations like searches and sorts. In hardware, carry-look-ahead adders use more circuitry to decrease the amount of time each operation takes. Trading space for speed can also be applied to disk drives, by trading disk capacity for increased performance.

1.1 Motivation

The disk drive is a critical component in a computer system. Not only is the disk drive used for permanent storage, it is also used for temporary storage when system

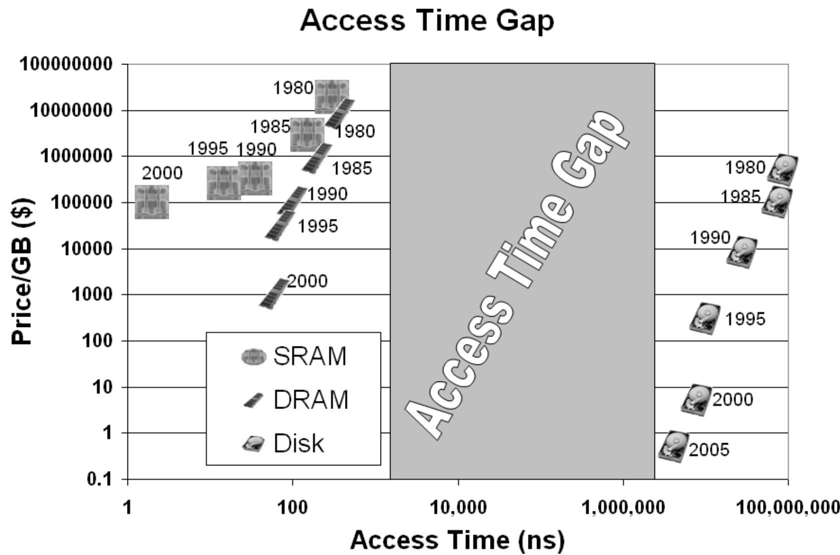


Figure 1.1: Access time and cost of SRAM, DRAM, and Disks [2].

---

memory is low. The disk drive is involved in almost every task performed by the user, at one point or another. Because of the heavy reliance on disk drives, improving the performance of the disk drive can be beneficial to the performance of a computer system.

Over the past few decades, disk drive performance has improved at a slower rate than other computer components. Processors have roughly doubled in performance every 1.6 years while hard drives have taken about eight to ten years to double in performance[1]. However, the capacity of hard drives have increased dramatically over the past few decades. These trends are shown in Figure 1.1.

One way to use extra disk capacity is to trade it for increased performance. Extra disk capacity can be used as a sequence cache that stores data in the order that the operating system requests it. By storing data in the order the operating system requests it, disk access time is decreased, thus improving the overall system performance.

1.2 Thesis Statement and Layout

We submit a new idea that trades disk space for an On-Disk Sequence Cache (ODSC) that caches sectors in the order they are accessed. An ODSC decreases the average disk

access time, resulting in faster application load times. The reduced disk access time also increases system performance and improves system performance when physical memory is limited.

Chapter 2 describes disk drive fundamentals and how they relate to system performance. Chapter 2 also outlines prior work done in this area. Chapter 3 describes our implementation of an ODSC. Chapter 4 describes the experiments and results that test our thesis. Chapter 5 summarizes the results, draws conclusions, and describes areas of future work.

Chapter 2

Foundational Material

This chapter discusses foundational material to help the reader understand concepts related to disk drive performance and our ODSC. The *disk drive fundamentals* section overviews the different components of a disk drive and discuses how these components affect performance. Then dynamic sequence detection, a concept upon which the ODSC relies, is described. This chapter also discusses related work and disk caches in current computer systems.

2.1 Disk Drive Fundamentals

It is important to understand the fundamentals of a disk drive because the ODSC rearranges and copies data on the disk to achieve better performance. This section covers the mechanical components and operations of a disk drive as well as operating system interactions. The mechanical components and operations of a disk drive show the impact data layout has on disk drive performance and how rearrangement can improve performance. Operating system interactions show why data can be rearranged.

2.1.1 Mechanical Components

Ruemmler and Wilkes [3] describe the mechanical components of a disk drive as follows. A disk drive contains one or more platters that rotate on a central spindle, as shown in Figure 2.1. Each platter surface (each platter has two surfaces) has an

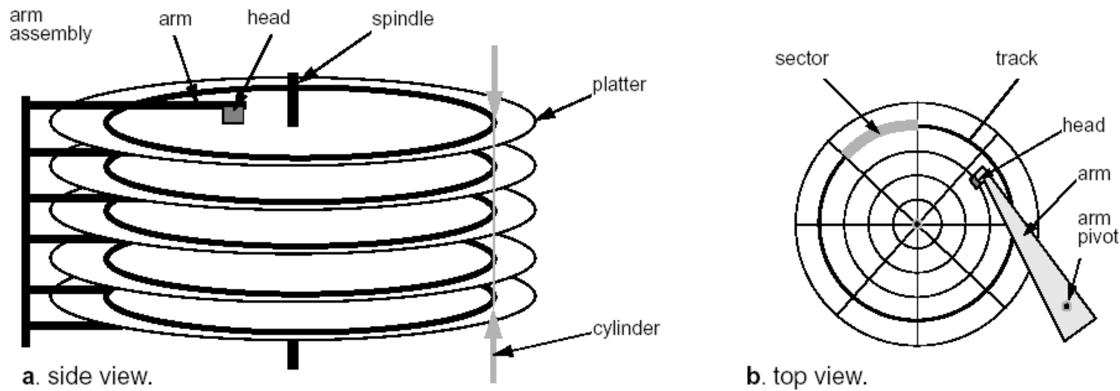


Figure 2.1: Mechanical components of a disk drive [3].

associated disk head that reads and writes the magnetic flux variations on the platter surface. The disk heads are connected to a single read/write data channel that not only switches between the various heads, but also encodes and decodes the data stream into and from the magnetic phase changes stored on the disk.

The platters store data in a series of concentric circles, called tracks. “A single stack of tracks at a common distance from the spindle is called a cylinder” [3]. Each track is divided up into sectors, which is the base storage unit on a disk and usually stores 512 bytes. Sectors are also grouped into blocks by the operating system. Therefore, the data stored on the disk is referred to in sectors, but once it is read off the disk it is referred to in blocks.

In order to read or write the data from or to the disk, the heads must be positioned over the correct track. All the heads are attached to a single arm assembly that moves the heads together. Once the head is positioned over the correct track, the data is read or written as the platter rotates past the head.

2.1.2 Disk Drive Operations

Ng [4] divides the time it takes to read to/write from a disk drive (disk access time) into four categories:

- command overhead - the time it takes for the disk drive's microprocessor to process the request,
- seek time - the time it takes to move from the current cylinder to the target cylinder,
- rotational latency - the time it takes for the target sector to rotate under the head,
- data transfer time - the time it takes to transfer the data from/to the disk drive.

Of these four categories, seek time and rotational latency, the two mechanical operations, comprise 60% and 30% of the total disk access time, respectively [4].

Seeks comprise most of the disk access time because of the mechanical limitations of a disk drive. Ruemmler and Wilkes [3] describe a seek as being composed of:

- "a *speed up*, where the arm is accelerated until it reaches half of the seek distance or a fixed maximum velocity,"
- "a *coast* for long seeks, where the arm moves at its maximum velocity,"
- "a *slowdown*, where the arm is brought to a rest close to the desired track," and
- "a *settle*, where the disk controller adjusts the head to access the desired location."

Since shorter seeks take less time than longer seeks [3], arranging sequential disk accesses close together will reduce seek time. Rotational latency may also be reduced by storing sectors in the order they are accessed on the disk, such that the next sector to be read will be under the reading head. (The disk drive also reads the next few sectors of the current request and stores them on a cache RAM. Therefore, when the operating system requests the next sector, it will get the sector from the cache instead of reading it off the disk.) Reducing seek time and rotational latency reduces disk access time and improves system performance, as shown by Akyurek [5].

2.1.3 Operating System Interactions

In order to understand why data can be rearranged to improve disk drive performance, one must understand how the operating system interacts with the disk drive. Modern operating systems improve disk performance by reducing the number of times they access the disk. Demand paging is one of the techniques operating systems have implemented to reduce disk accesses. Demand paging only reads a section, called a page, of a file as it is needed instead of reading the entire file off the disk. Although demand paging causes the hard drive to seek more, it can increase overall performance because pages that are not needed are never read off the disk [6].

Another technique that operating systems use to reduce the number of disk accesses is disk caching. Disk caches store blocks of disk data in main memory so the blocks do not have to be re-read off the disk. The disk cache improves system performance in two different ways. First, the number of disk reads is reduced because blocks are cached in memory. Second, it makes writes non-critical to system performance because the changes are stored in memory until the system has time to write them out to disk [7].

Although demand paging and disk caching have improved disk performance, they have also increased the average seek time. Demand paging increases seek time because accesses to multiple files are interleaved. Because accesses to multiple files are interleaved, a disk defragmenter, or a tool that combines file fragments sequentially on the disk, does not greatly reduce seek time. Disk caching also creates greater seek times because some pages in a file can be removed from the disk cache without removing all the pages in the file. We can rearrange sectors on the disk to further increase performance, since demand paging and disk caching have increased seek time [8].

2.2 Dynamic Sequence Detection

Dynamic sequence detection was introduced by Peacock [9] and is a technique upon which the ODSC depends. By tracing disk requests real time, dynamic sequence de-

tection is able to find common sequences as a computer is running. Sequences are determined by obtaining and analyzing the sector and time of each disk request the operating system makes. The ODSC uses Peacock's work to find sequences that can be copied into the on-disk cache.

2.3 Prior Work

There is a large body of work on improving disk drive performance. Some of the first work studied disk drives and modeled their performance in order to evaluate different methods for improving performance. Based upon these models, researchers then sought to improve performance by improving filesystems. Even though filesystems improved, researchers noticed that many disk accesses were still non-sequential. Several researchers developed new ideas to increase the number of sequential accesses. Some of these ideas included trading capacity for performance.

Some of the first work done in disk drive performance started with studying and modeling disk drives. Ng [10] studied the causes of disk latency and different methods that reduce latency. Most of the methods studied involved adding physical components to disk drives or copying data on the disk. Ruemmler and Wilkes [3] developed a model of a disk drive and applied that model to evaluate different methods that seek to improve disk performance.

Some of the methods that Ruemmler and Wilkers studied included work done to improve filesystems. Filesystem improvements started with Ousterhout's work [11] that traced the UNIX 4.2 BSD Filesystem and analyzed its performance. Then McKusick [12] designed and developed a fast Filesystem for UNIX, and Card [13] further developed the EXT2 Filesystem, also described by Oxman [14]. These faster file systems were designed to put files sequentially on the disk.

Even though filesystems were becoming more and more efficient at storing files sequentially on the disk, Ruemmler and Wilkes [1] noticed that 25-50% of disk accesses were non-sequential. Zhou [6] also noticed the non-sequential accesses when she devel-

oped a software model for simulating a disk drive. She used this model to show that rearranging disk data in the order the operating system accesses it greatly improves disk performance.

Because of Ruemmler and Wilkes [1], and Zhou's [6] work, researchers sought to either increase the percentage of sequential accesses or reduce the penalty for non-sequential accesses. Yin [8] rearranged disk data to increase the number of sequential accesses during application loading. Lin [15] showed that by rearranging disk data, power consumption can be reduced. Lei and Duchamp [16] used file access traces to predict which file would be requested next, and then used that information to develop a file prefetching strategy. Also, both Peacock [9] and Li [17] developed ways to analyze disk traces in real time, as described above.

Other researchers focused on copying disk data to improve performance. Akyurek and Salem [5] [18] used real time traces to determine "hot blocks", or blocks that are frequently accessed, and copied these "hot blocks" to the center of the disk. Hu and Yang [19] used an extra partition to cache disk writes, and Yu [20] showed that copying disk data improves performance in a disk array.

By copying disk data and using real-time traces, our work builds off of the work done by Peacock [9] and Akyurek [5]. We use Peacock's work to trace file accesses in real time to find sequences that can be cached. Like Akyurek, we copy data to a separate part of the disk, however instead of just copying blocks we copy sequences.

2.4 Disk Caches

In a computer system there are at least two different disk caches. The first disk cache stores recently accessed sectors in a memory chip on the disk drive. Disk blocks stored in a computer's main memory (usually by the operating system) comprise the second disk cache. In order to alleviate confusion between the different disk caches, we will refer to the first cache as the disk cache, the second cache as the memory disk cache, and our ODSC as either on-disk cache or sequence cache.

2.5 Summary

This chapter introduced disk drive fundamentals including both the mechanical components, operations, and operating system interactions. It also reviewed dynamic sequence detection, prior work, and identified the different disk caches. These concepts describe why a sequence cache, which is described in the next chapter, is beneficial to system performance, and gives a basic understanding of the concepts upon which a sequence cache is built.

Chapter 3

On-Disk Sequence Cache (ODSC)

In the previous chapter we discussed foundational material to help the reader understand concepts related to the ODSC. This chapter overviews the design and implementation of our ODSC. After an overview of the ODSC, we cover trace collection and disk request translation – the two main functions of the ODSC. We also cover the three separate parts of the ODSC and describe how these parts accomplish the trace collection and disk request translation.

3.1 Overview

In order to efficiently copy commonly used sequences into the on-disk cache, the ODSC must first identify the commonly used sequences. In order to identify the sequences, the ODSC needs to collect a disk trace, or list of disk accesses, in real time. We use the same method that Peacock [9] uses. After collecting the trace, some processing is done to discover the sequences, and then the sequences are copied into the sequence cache. The next function that the ODSC accomplishes is translating disk requests so that they are read from the sequence cache instead of their original location.

After describing both of the these functions, we describe the three separate parts of the ODSC. The first part includes modifications to the 2.4.26 Linux kernel, the latest stable kernel when this work began. The second part is a kernel module based on Rubini

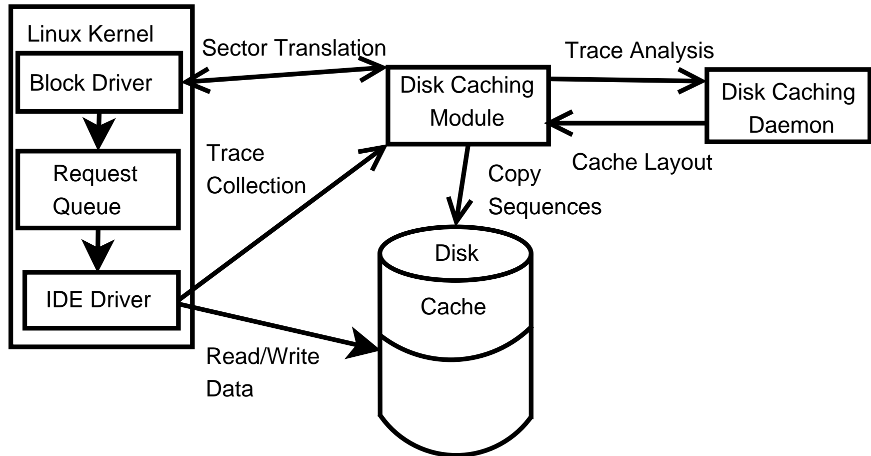


Figure 3.1: Design of the On-Disk Sequence Cache(ODSC). The diagram shows the three parts of our ODSC and the interactions between them.

---

and Corbet's [21] example module with added functionality to collect traces and translate disk requests. A daemon that is responsible for analyzing the disk trace, determining the sequences, and organizing the sequence cache comprises the last part. Figure 3.1 diagrams these three parts and their interactions.

3.2 Trace Collection

As mentioned above, the ODSC needs to collect a trace in order to find the most frequently used sequences, and copy those sequences into the sequence cache. The trace is collected from the IDE driver in the Linux kernel, right before the driver sends a request to the IDE controller. By collecting the trace in the IDE driver, we are able to trace the requests in the exact order they are sent to the disk drive. The trace is stored temporarily in kernel memory allocated by the Disk Caching Module(DCM) (see bellow). The trace is then read and analyzed by the Disk Caching Daemon(DCD). The daemon is responsible for finding the sequences in the trace.

For each disk request there is one trace record with the following information:

- the operation to be performed, either a read or a write,

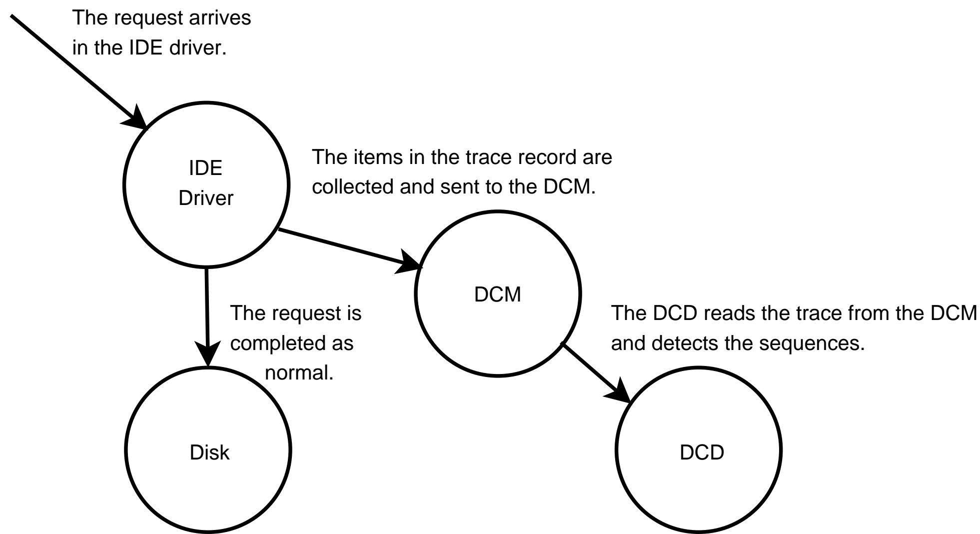


Figure 3.2: The diagram of the trace collection process.

- the major and minor number of the device. These numbers are unique to every I/O device and help the Linux kernel identify and communicate with the I/O devices,
- the sector that is to be read/written,
- the number of sectors to read/write, and
- a time stamp collected from the clock tick counter on the CPU.

All of this information is passed to the module and used by the daemon to find sequences.

Although not important to the operation of the ODSC, another trace record is collected at the end of each request in order to determine disk access times. This trace record is collected in the IDE interrupt handler that gets called as soon as the IDE controller finishes the request and has the same information described above. We can calculate a disk access time for the request by combining the information from the trace record collected before the request and the trace record collected after the request. We use this service time to compare the ODSC's performance to normal disk performance. The trace collection process is diagramed in Figure 3.2.

3.3 Disk Request Translation

The second operation that the ODSC must perform is disk request translation. The ODSC translates requests for sectors that are stored in the sequence cache. These requests are translated in the block driver portion of the Linux kernel. Just before the request is sent from the block driver to the IDE driver, the kernel calls a function within the DCM to see if that sector is in the sequence cache. If it is cached, it translates that request to read it from the sequence cache instead of the original location.

The disk caching daemon has nothing to do with the disk request translation other than deciding what sectors should be cached and where they should be located. It is also important to note that a sector can be copied into the cache multiple times – one time for each sequence in which it appears. The disk caching module keeps track of the disk head location so it can translate the request to the closest copy.

In order to make sure that all the copies of a sector contain the same data, we modify the write procedure. When a write request is sent to the DCM for translation, the DCM checks to see if there are any copies of the requested sector in the cache. If so, the DCM copies the new data to every copy in the sequence cache. It then returns an unmodified request so that the kernel will write the sector to the original location. The extra writes are not critical to system performance due to the disk cache, that allows writes to be delayed until the system is idle [7]. The translation process is diagramed in Figure 3.3.

3.4 Kernel Modifications

There are two places in which the kernel is modified. The first place is in the IDE driver. The IDE driver is modified to send the trace to the Disk Caching Module (DCM). In the IDE driver there is a function that is called *do_read_write_request* which is replaced by our own function. Our function first checks to see if the DCM is loaded in the kernel, since it can be loaded or unloaded at any time. Next, it asks the DCM if the request's major and minor number should be traced. If so, it collects the items in the trace record mentioned above and calls a function in the DCM that records the trace record. Then,

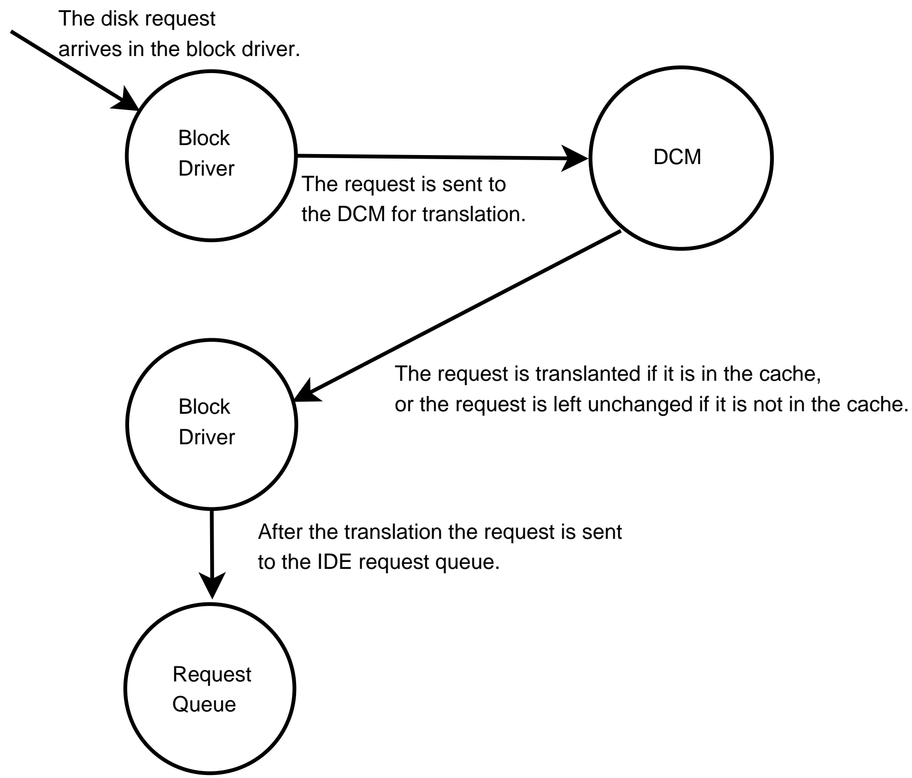


Figure 3.3: The diagram of the disk request translation process.

---

the function continues to send the request to the IDE controller. The same procedure is done to collect the trace at the end of the request in the IDE interrupt handler routine.

The second place the kernel is modified is in the block driver. As mentioned above the block driver is modified to translate requests for cached sectors. This is done in the function that is called whenever a block read or write is requested. We modify this function to first check to make sure the DCM is loaded and that the major and minor number of the requested block is currently being cached. If so, it asks the DCM to translate the block. If the block is not in the cache, the request is left unchanged. If it is in the cache, the request is translated to the copy closest to the disk head. After the translation, the request is then sent to the IDE driver's request queue.

3.5 Disk Caching Module (DCM)

As shown in Figure 3.1, the DCM communicates with the kernel and the Disk Caching Daemon (DCD). The DCM communicates with the kernel by collecting the trace and translating the requests. The DCM collects the trace through a function called by the kernel, as explained above. It stores the trace in a flexible-sized buffer. This buffer can grow as the trace is collected and shrink as the trace is read by the DCD. The buffer grows by allocating another buffer and linking it to the previous buffer in a manner similar to a linked list. When a buffer is completely read, the buffer is deallocated and the read pointer is directed to the next buffer.

In addition to collecting the trace from the kernel, the DCM also translates disk requests. The DCM uses a hash table, stored in memory, to keep track of which sectors have been cached, and where those sectors are located in the cache. Whenever the kernel asks the DCM to translate a request, it looks up the sector in the hash table. It then returns either the cached location of the sector, the closest cached location to the disk head if that sector is cached multiple times, or the original sector if that sector is not in the cache. The kernel uses this information to translate the request so that it is read from the cache instead of the original location. If the request is a write, the DCM copies the data in the sector to all of the cached locations, and the kernel continues to write the data in the sector to the original location. As explained above, the extra writes are not critical to system performance.

In order to organize the cache, the DCM communicates with the DCD. The DCD reads the trace through a device file created when the module is loaded (/dev/dcm). The DCD instructs the DCM to cache items through I/O Control (IOCTL) calls. These calls can be made by any process that has permission to write to the device file. The DCM has the IOCTL functions listed in Table 3.1. Through these IOCTL calls, the DCD can add sectors to the cache, remove sectors from the cache, clear the cache, and

Name	Purpose	Parameters	Return Value
Start	Start tracing/caching a disk partition.	Major and minor numbers of the partition	None.
Stop	Stop tracing/caching a disk partition.	Major and minor numbers of the partition	None.
Add	Add a sector to the cache.	Sector and cache location.	None.
Remove	Remove a sector from the cache.	Sector.	None.
Clear	Clear the cache.	None.	None.
Save	Write the hash table to Disk.	None.	None.
Translate	Translates a sector (for testing).	Sector.	Location.

Table 3.1: DCM IOCTL Functions with their purpose, parameters, and return values.

---

save the cache. The DCD can also tell the DCM to start tracing and caching a particular disk partition.

The DCM is also responsible for copying sectors into the cache. When the DCM is loaded into the kernel, it is given a disk partition to use as a cache. After being loaded, the DCM checks the cache partition for a special code at the beginning of the cache to indicate that a valid hash table has been saved to the partition. If there is a valid hash table, it is read into memory; if not, it creates an empty hash table and saves that to the cache partition. When the DCD instructs the DCM to add a sector into the cache, the DCM reads that sector off the disk, copies it to the indicated location and then adds the sector to the hash table. The hash table is saved only when the *save* IOCTL function is called. The DCD must keep track of the size of the hash table and save the appropriate amount of space at the beginning of the cache so that the hash table can be written to the cache partition.

3.6 Disk Caching Daemon (DCD)

The Disk Caching Daemon (DCD) is responsible for reading and analyzing the trace, determining the sequences, and deciding what sequences should be cached. As mentioned

above, the DCD reads the trace from the DCM by opening and reading the DCM's device file (/dev/dcm) created when the module is loaded into the kernel. When there is nothing to be read, the read blocks, putting the DCD to sleep. As the DCD reads the trace, it analyzes the trace and builds sequences.

Since this work only proves that a sequence cache improves performance, our DCD collects all of the reads and builds one big sequence. It does not worry about the size limitation of the cache because the cache partition is sufficiently larger than all of the reads executed during the benchmark. Therefore, the DCD can just collect all the reads and cache them in the correct order. We leave cache replacement and sequence detection experiments for future work. The DCD can be modified to employ a variety of sequence detection and caching schemes.

After the benchmark is run, we can signal the DCD to copy the sequence to the cache by giving the DCD process a *USR1* signal. Upon receiving the signal, the DCD wakes up and instructs the DCM to clear the current cache and to insert the new sequence into the cache by the IOCTL calls listed in Table 3.1. After the DCM is finished caching the new sequence, the DCD tells the DCM to save the hash table, and returns to reading the trace.

3.7 Summary

This chapter described our implementation of a ODSC, by first covering the functions of the ODSC – trace collection and disk request translation. After describing the functions it reviewed the three parts of the ODSC – the kernel modifications, the DCM, and the DCD. Now that the reader understands how our ODSC functions, the next chapter will describe the experiments performed and their results.

Chapter 4

Experiments and Results

The previous chapter described the design and operation of our ODSC. This chapter discusses the experiments performed with the ODSC and associated results. First, we discuss application loading experiments. Next, we review Linux booting experiments, and finish with limited memory experiments. These experiments and results show that our ODSC improves disk performance.

4.1 Overview

The ODSC improves the performance of the memory disk cache by reducing the miss penalty, or the time it takes to access disk blocks that are not in the memory disk cache. Two types of cache misses are compulsory misses and capacity misses. Compulsory misses are cache misses of blocks that have not been read off the disk and cached in memory since the last reboot. Capacity misses are cache misses of blocks that have been read off the disk, but have later been evicted from the memory disk cache. The first two experiments test the ODSC's performance on compulsory misses, and the last experiment tests the ODSC's performance on capacity misses.

All experiments are performed on a Dell Optiplex GX260. This machine uses an Intel Pentium 4 Processor running at 2.532 GHz and has 512 MB of RAM. We also use a Western Digital 40 GB hard drive with a rotational speed of 7200RPM.

4.2 Application Loading

We use the loading of three different applications (Emacs, Gimp and Gvim) and a kernel make as benchmarks to test the effectiveness of our ODSC. We are able to time the loading of the three applications by having them quit as soon as they are loaded. For Emacs and Gimp, we use batch commands that are put on the command line. For Gvim, we use a script file that tells the application to quit. The kernel make is a make of the kernel after "touching" the IDE header. "Touching" the header file updates the time stamp, which tells *make* to recompile portions of the kernel. All four benchmarks are timed with the UNIX *time* function.

We gather three different times for each benchmark. The first time is the time it takes to run the benchmark on the original Linux 2.4.26 kernel with no modifications. We call this time the original time. The second time we collect is the tracing time, which is the time it takes to load the application while the DCD is collecting a trace in order to find the sequences that should be cached. After the DCD loads the sequences into the sequence cache, we collect a third time, which is the time it takes to load the application from the ODSC. Before we run each benchmark, we reboot the machine to clear the memory disk cache. Rebooting assures us that the memory disk cache is empty and that the benchmark will be read from the disk instead of memory. The three times for each benchmark are collected multiple times to show that the results are statistically significant.

4.2.1 Application Loading Results

The results of the application loading experiments are in Table 4.1. The ODSC speeds up the application loading by at least 88.2% (and for Emacs by 413%). The results also show that the tracing, although slightly slower, does not greatly impact performance. In all cases besides Emacs, the difference in the original time and the tracing time is not statistically significant. These results show that the ODSC improves

Application	Original Time (s)	Tracing Time (s)	ODSC Time (s)	Speedup
Emacs	1.355	1.469	0.2638*	413%
Gimp	3.095	3.104	1.644	88.2%
Make	48.549	48.123	37.728	28.7%
VI (gvim)	1.323	1.367	0.5190	154%

Table 4.1: Application loading results in seconds. The mean varies less than two percent with 99.9% confidence, except * varies from 0.191 to 0.336 seconds with the same confidence.

---

disk performance when applications are read out of the cache and that the tracing does not significantly slow down the system.

We are only able to speed up the kernel make by 28.7%. One reason why we cannot improve the kernel make performance as much as the other benchmarks could be because there is a higher percentage of processing. Another reason could be because the object files might be written to a different place than where they were originally.

Figure 4.1 through Figure 4.3 show the sources of the speedup. The data for these graphs comes from traces of the Gvim benchmark both with and without the ODSC. The graphs of the other benchmarks are in Appendix A. Figure 4.1 shows the seek distance for each request with and without the ODSC. These figures clearly show that a large amount of seeking is done without the ODSC, but with the ODSC, there is almost no seeking. Figure 4.2 graphs the disk access time of each request with and without the ODSC. These graphs show that by reducing the seek distance, disk access time is reduced. The final figure, Figure 4.3 further show the effect of seeking on disk access time by graphing access time as a function of seek distance.

As an interesting side note, Figure 4.3(a), shows many properties of disk accesses. The slope of the upper "band" of points steadily increases showing that the further the

seek distance, the longer it takes. The width of the upper "band", about 8 milliseconds, shows the rotational latency for a 7200 RPM drive. The "line" of points along the x-axis is due to the disk cache.

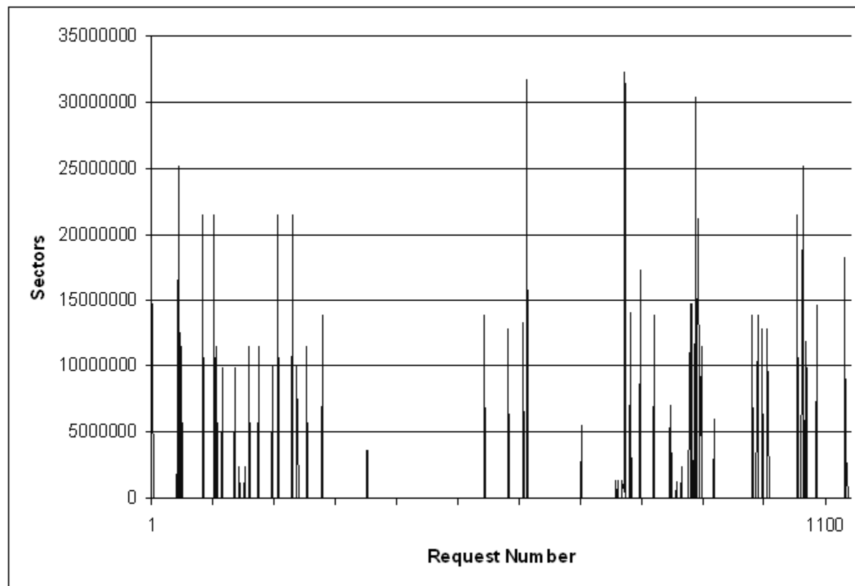
4.3 Linux Booting

Besides using the loading of applications, we also use the Linux boot as a benchmark to evaluate our ODSC. Since we cannot use the UNIX *time* function to time the Linux boot, we use a trace in order to evaluate the results. As mentioned in Chapter 3, we collect a trace record right before each request is sent to the IDE controller and right after the request completes. With these records, we calculate the disk access time for each request. We use the sum of these times to compare our ODSC to normal disk drive operation.

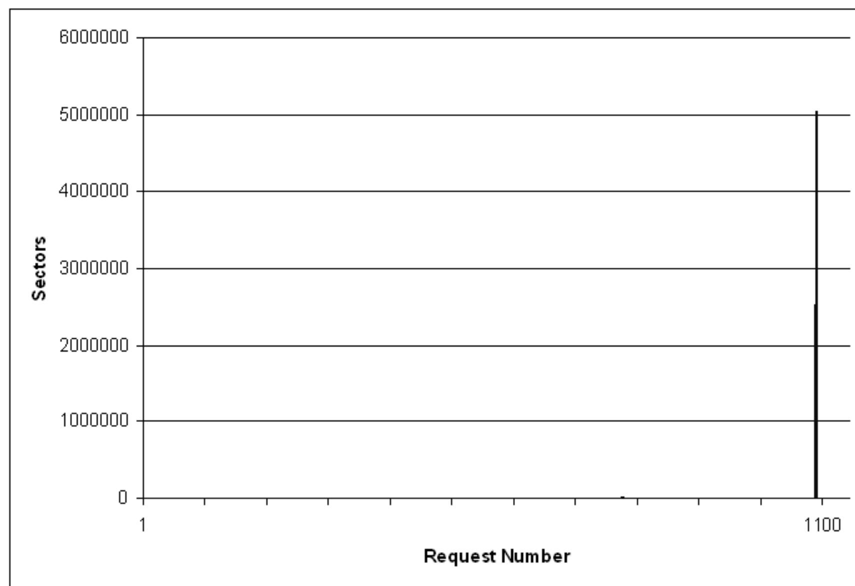
Another limitation to using the Linux boot as a benchmark, besides not being able to use the UNIX *time* function, is that we cannot trace and cache the entire boot. The DCM cannot be loaded into the kernel until after the kernel is fully loaded. The earliest time we can load the DCM is right after the IDE driver is loaded. Therefore, we start our trace and sequence cache right after the IDE driver and the DCM is loaded. We stop collecting results after we have logged in and Gnome, the desktop environment, is loaded. As with the application loading benchmarks, we perform the experiment multiple times to show statistical significance.

4.3.1 Linux Booting Results

Table 4.2 contains the results of the Linux boot benchmark. As shown, the ODSC reduces the total disk access time of the Linux boot by a little more than ten seconds, representing a speedup of 396%. The results correspond to the speedup of the Linux boot while timing it with a stopwatch. From these results, we show that the ODSC is effective in reducing the time it takes to boot Linux.

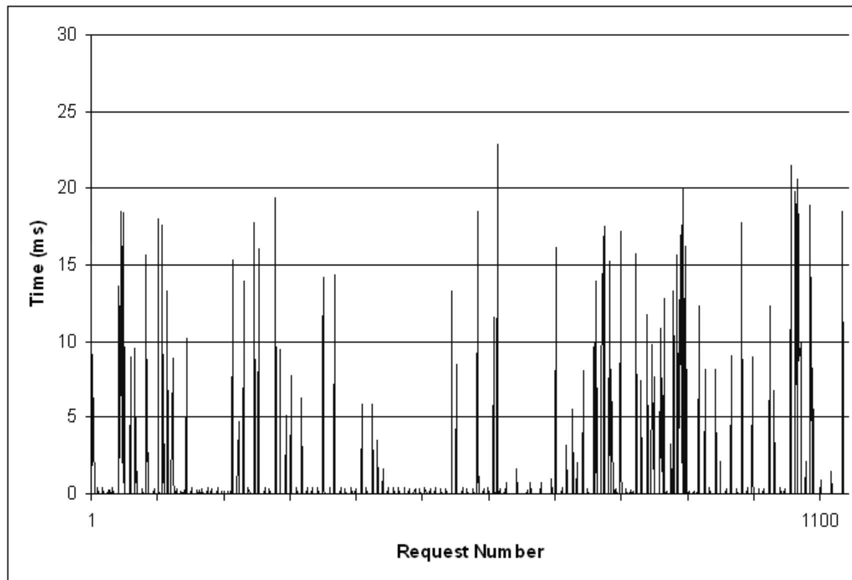


(a) Without the ODSC.

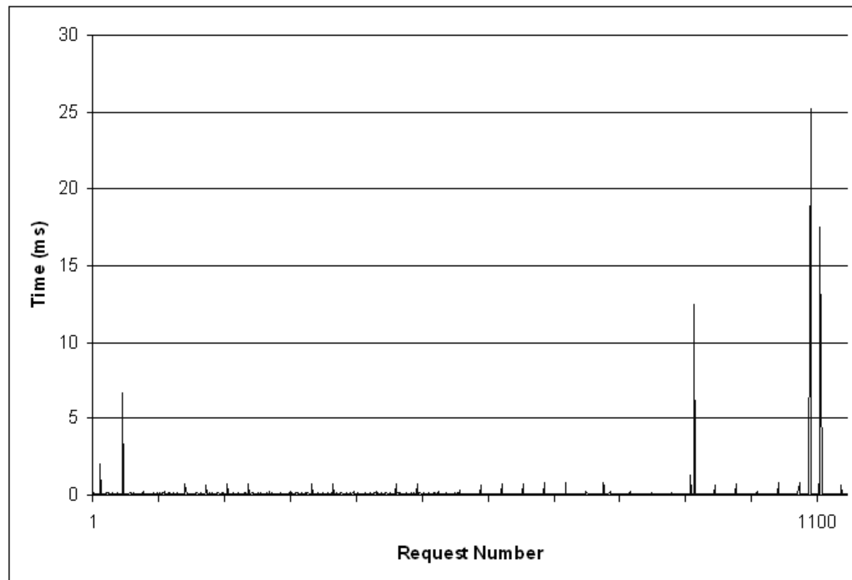


(b) With the ODSC.

Figure 4.1: Seek distance of Vi (Gvim) (a) without the ODSC and (b) with the ODSC. The total seek distance is 814,833,144 sectors without the ODSC and 10,210,904 sectors with the ODSC. The ODSC greatly reduces the seek distance.

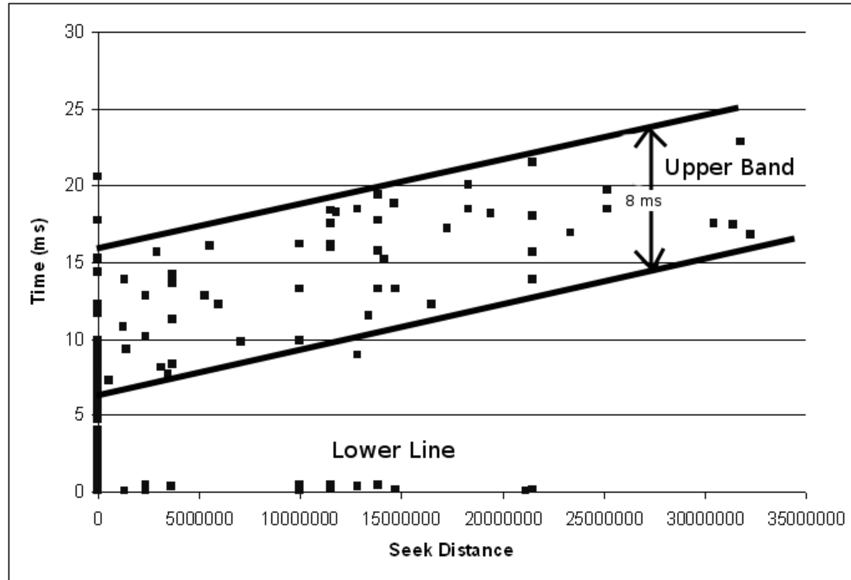


(a) Without the ODSC.

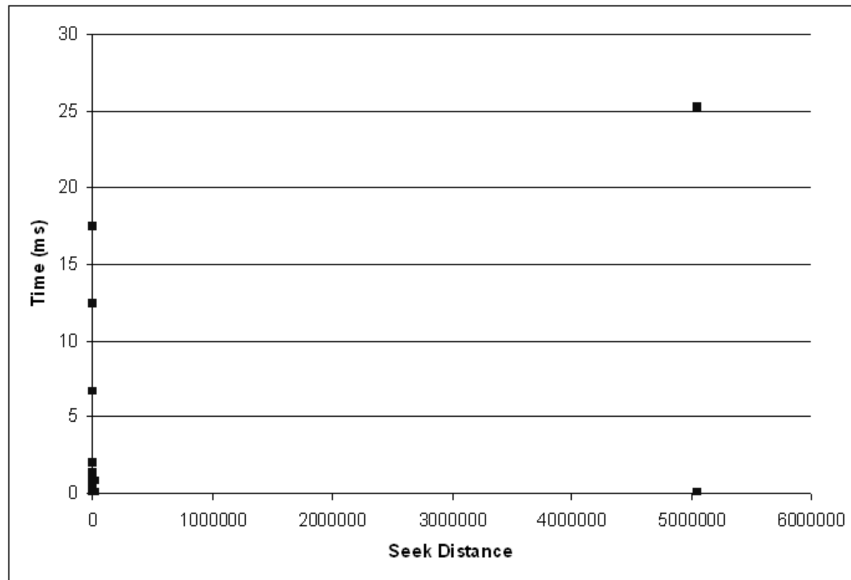


(b) With the ODSC.

Figure 4.2: Disk access time of Vi (Gvim) (a) without the ODSC and (b) with the ODSC. The total disk access time is 1,215.14 ms without the ODSC and 202.67 ms with the ODSC. The ODSC reduces the disk access times because of the reduced seek distances.



(a) Without the ODSC.



(b) With the ODSC.

Figure 4.3: Seek distance vs. time of Vi (Gvim) (a) without the ODSC and (b) with the ODSC. (a) shows many properties of disk accesses that are explained in the text.

	Sum of Access Times (s)	**Average Access Time (ms)**
Without Cache	12.65	3.315
With Cache	2.555	0.2626

Table 4.2: The total disk access time and the average disk access time of the Linux boot in seconds and milliseconds, respectively. The mean varies less than five percent with 99.9% confidence. This represents a 396% speedup.

4.4 Limited Memory Experiment

As stated above, this experiment tests the ODSC's performance on capacity misses. In order to test the ODSC's performance on capacity misses we need to reduce the amount of available memory. Reducing the amount of available memory causes more blocks to be evicted from the memory disk cache, which, in turn, creates capacity misses when the benchmark is run again.

We would perform a direct experiment in order to show that the ODSC improves performance when physical memory is limited, but our implementation of an ODSC does not work well in limited memory situations. For each disk request that is translated to the sequence cache, the memory disk cache stores two copies – one for the original block and one for the block in the sequence cache. This double caching prevents our ODSC from working well under limited memory conditions.

Despite this limitation, we can analyze the effect of an ODSC when memory is limited by measuring the miss rate of the memory disk cache. According to Hennessy and Patterson, "*average memory access time* = *hit time* + *miss rate* ∗ *miss penalty*" [22]. By applying this formula to the memory disk cache, we get *average disk access time* = *memory disk cache hit time* + *memory disk cache miss rate* ∗ *disk access time*. We can ignore the memory disk cache hit time, since it is the same with and without the ODSC. For the disk access time we can average the time of the four application loading

benchmarks (0.249 *ms* with the ODSC and 4.018 *ms* without the ODSC). Now we only need to obtain the miss rate in order to compute an average disk access time (minus the memory disk cache hit time).

To obtain the miss rate we use a shell script that runs the four application loading benchmarks. We trace the disk the first time the shell script is run without anything in the memory disk cache to count the total number of disk accesses. Next we reduce the memory, by allocating a large amount of memory in the DCM, and run the shell script twice. The first time makes sure that every application has been loaded once. The second time we trace the disk to count the number of disk accesses, which is the number of times it misses the memory disk cache. We can calculate the miss rate by dividing the number of memory disk cache misses by the total number of accesses obtained in the first step.

4.4.1 Limited Memory Experiment Results

Figure 4.4 contains the miss rates of the memory disk cache at various levels of available physical memory. Once the memory disk cache becomes full, which happens between 192 MB and 160 MB, the number of disk accesses increases dramatically. This shows that in limited memory situations, there is an increase in the number of disk accesses. Figure 4.5 is made by applying the average of the times found in the application loading experiments to the miss rates in Figure 4.4. This graph shows that an ODSC would improve performance when physical memory is limited.

4.5 Summary

In this chapter we discussed the experiments that we performed to show that the ODSC decreases the average disk access time. We also showed that the decreased average disk access time results is faster application load times and, through an analysis, improves system performance when physical memory is limited. In the next chapter we draw conclusions and discuss future work.

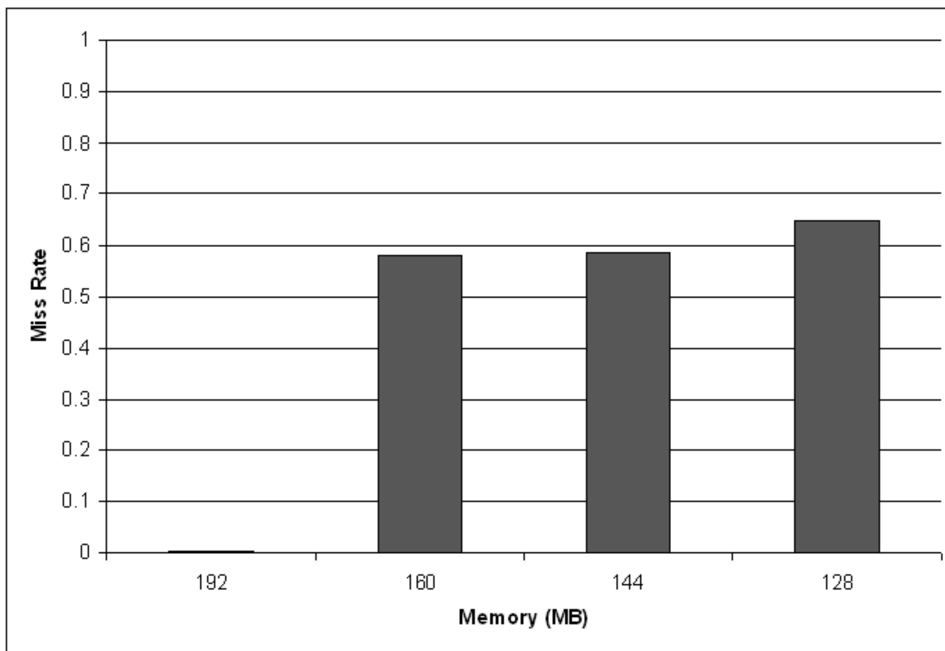


Figure 4.4: Miss rate of the memory disk cache as a function of the amount of physical memory. The mean miss rate at each data point varies less than five percent with 99% confidence.

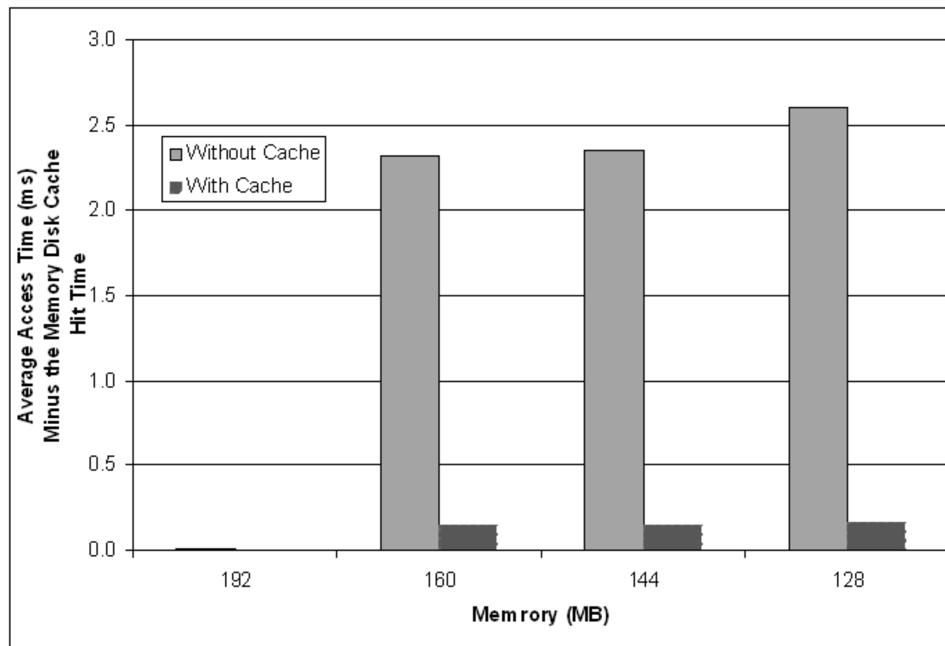


Figure 4.5: Computed average access times with and without an ODSC, minus the memory disk cache hit time. This graph uses the average of the times found in the application loading experiments(0.249 *ms* with the ODSC and 4.018 *ms* without the ODSC) to compute analytical times.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, we present a new idea to improve disk drive performance. Our idea, an on-disk sequence cache (ODSC), uses a separate disk partition as a cache that stores disk data in the order that the operating system requests it. By storing disk data in the order that the operating system requests it, we reduce the amount of seeking that the disk drive head must do. The average disk access time is reduced because the disk drive head seeks less. Reducing disk access time improves the performance of the system, especially when booting the operating system, loading applications, and when main memory is limited.

Our experiments show that an ODSC improves the performance of the Linux boot, application loads, kernel makes, and when physical memory is limited. The ODSC speeds up the Linux boot by 396% and application loading by as much as 413%, and as little as 88.2%. We also show that an ODSC improves performance during a kernel make and that an ODSC improves system performance when main memory is limited.

5.2 Future Work

We can build upon this work by improving our implementation of the ODSC. As mentioned in the previous chapter, we use the block driver to read the sequence cache,

which doubles the size of the memory disk cache. By further integrating our ODSC with the Linux kernel, we can avoid this problem and improve the efficiency of the sequence cache. Other implementations can be explored by building the ODSC completely within the IDE driver, the IDE controller, or the disk drive itself.

Other future work includes actual memory reduction experiments to verify the analytical memory reduction results in this work. We can also explore different caching schemes and sequence detection schemes. It would also be interesting to study the ODSC's effect on different types of workloads, and find out which caching scheme works best for each workload.

Appendix A

Additional Graphs

The following graphs are seek distance, access time, and seek distance vs. time graphs of Gimp, Emacs, the Linux Boot, and a kernel make.

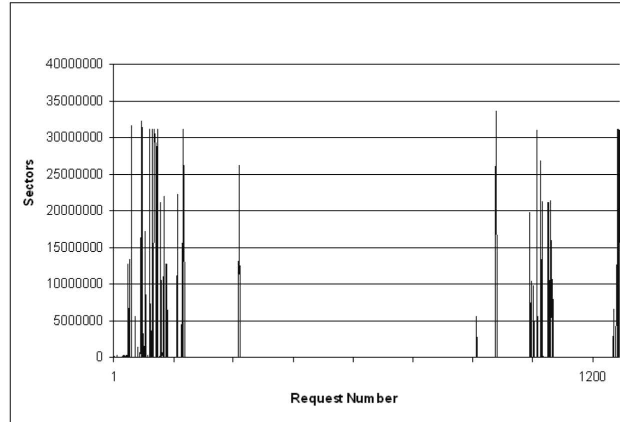


Figure A.1: Seek distance of Gimp without the ODSC for each request. Each request may read multiple blocks.

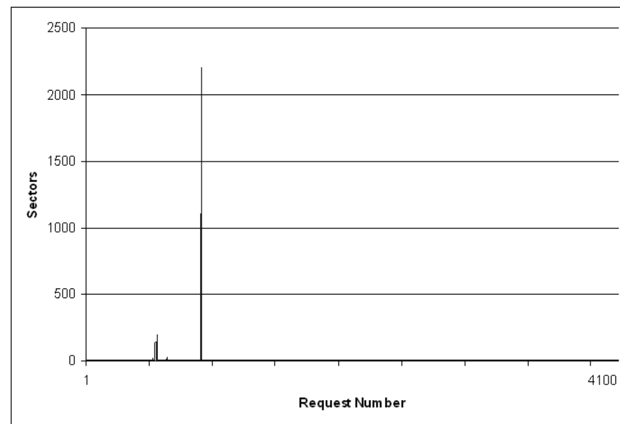


Figure A.2: Seek distance of Gimp with the ODSC for each request. There are more requests than Figure A.1 because each request reads one block.

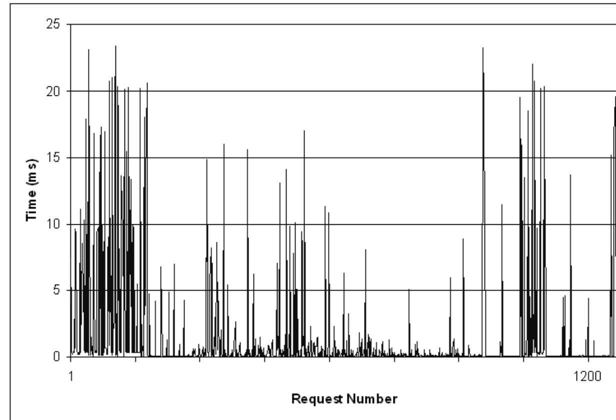


Figure A.3: Disk access times of Gimp without the ODSC for each request. Each request may read multiple blocks.

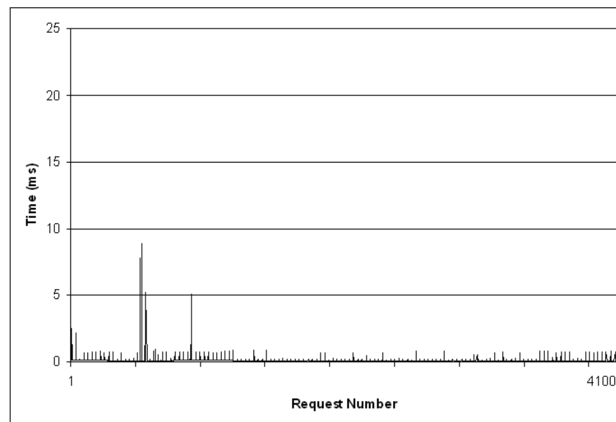


Figure A.4: Disk access times of Gimp with the ODSC for each request. There are more requests than Figure A.3 because each request reads one block.

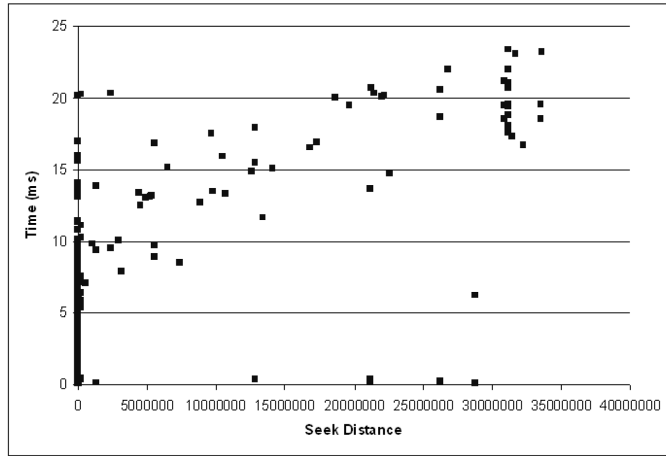


Figure A.5: Seek distance vs. time of Gimp without the ODSC.

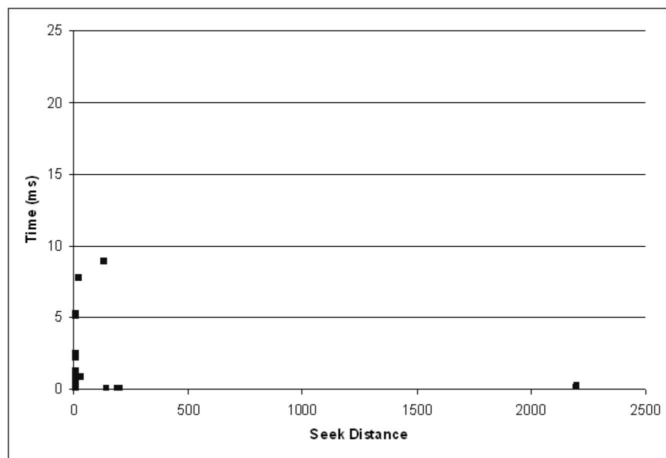


Figure A.6: Seek distance vs. time of Gimp with the ODSC.

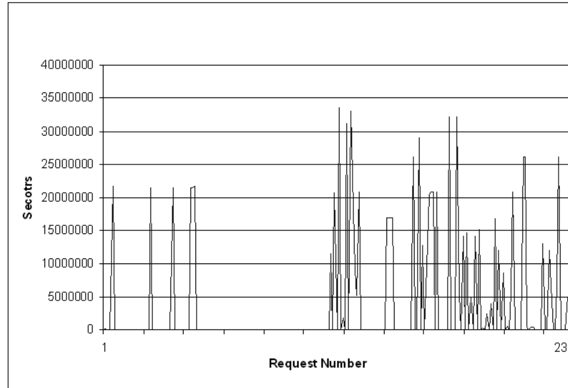


Figure A.7: Seek distance of Emacs without the ODSC for each request. Each request may read multiple blocks.

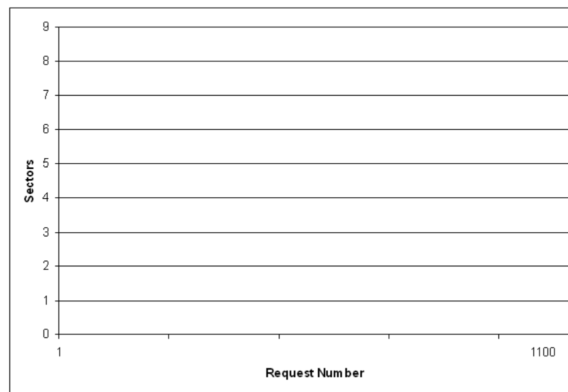


Figure A.8: Seek distance of Emacs with the ODSC for each request. There are more requests than Figure A.7 because each request reads one block. There is no line because every request's seek distance is zero.

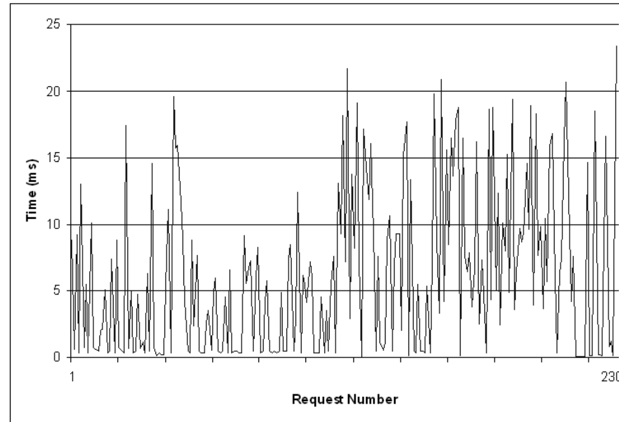


Figure A.9: Disk access times of Emacs without the ODSC for each request. Each request may read multiple blocks.

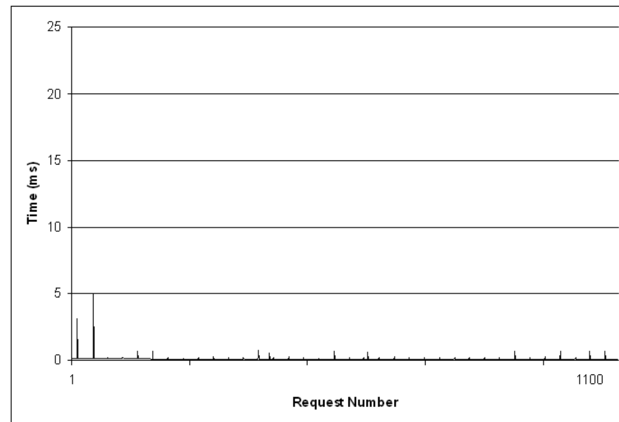


Figure A.10: Disk access times of Emacs with the ODSC for each request. There are more requests than Figure A.9 because each request reads one block.

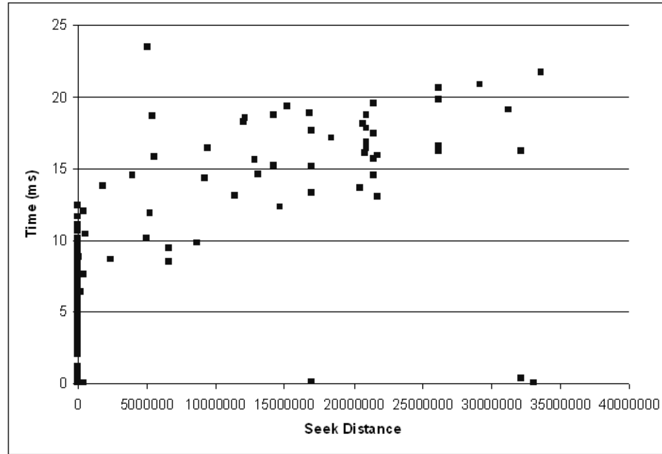


Figure A.11: Seek distance vs. time of Emacs without the ODSC.

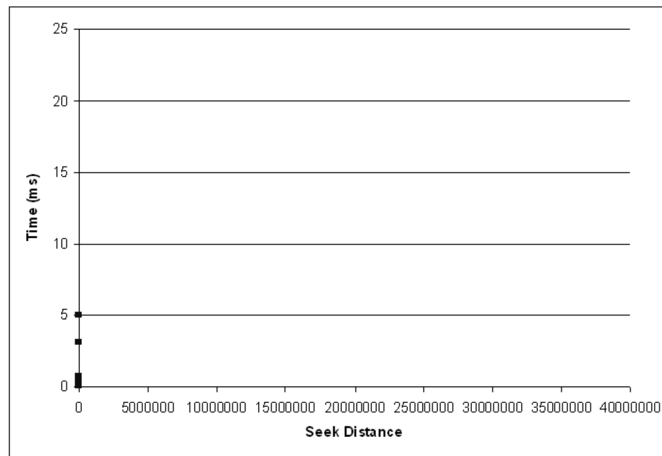


Figure A.12: Seek distance vs. time of Emacs with the ODSC.

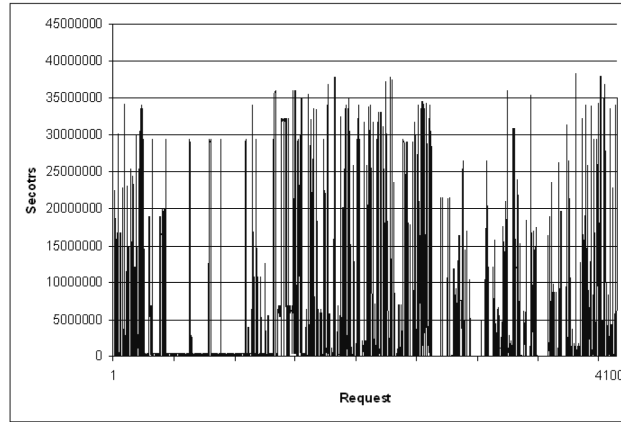


Figure A.13: Seek distance of the Linux boot without the ODSC for each request. Each request may read multiple blocks.

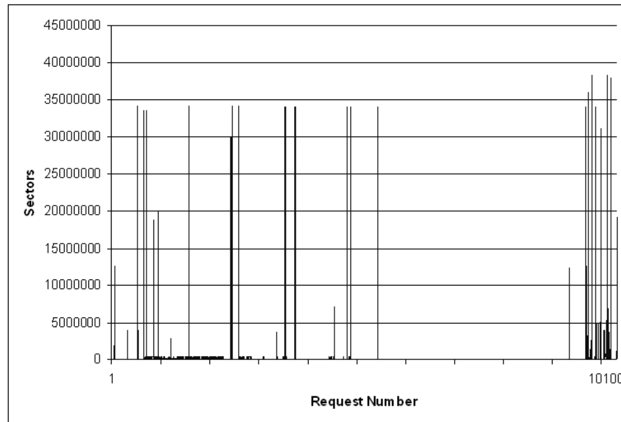


Figure A.14: Seek distance of the Linux boot with the ODSC for each request. There are more requests than Figure A.13 because each request reads one block.

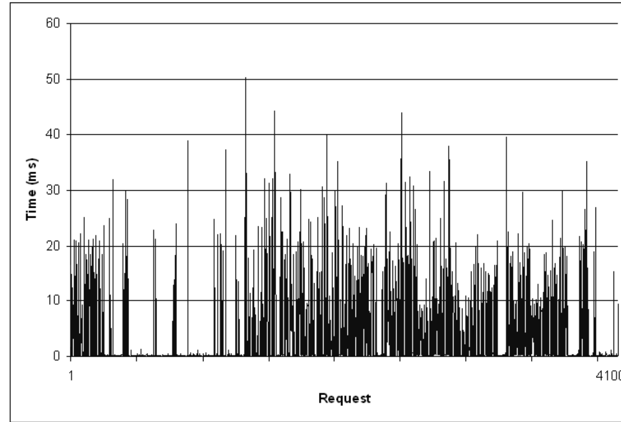


Figure A.15: Disk access times of the Linux boot without the ODSC for each request. Each request may read multiple blocks.

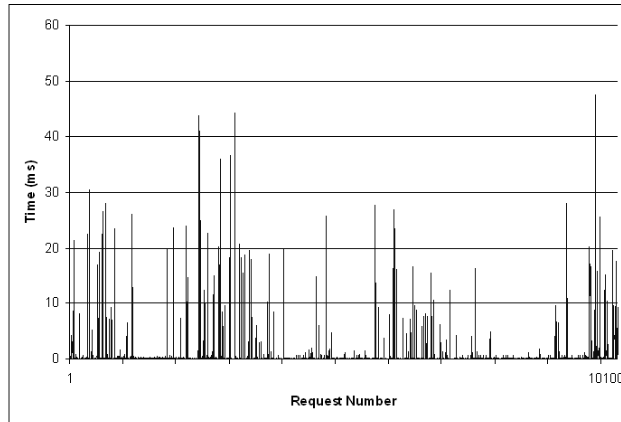


Figure A.16: Disk access times of the Linux boot with the ODSC for each request. There are more requests than Figure A.15 because each request reads one block.

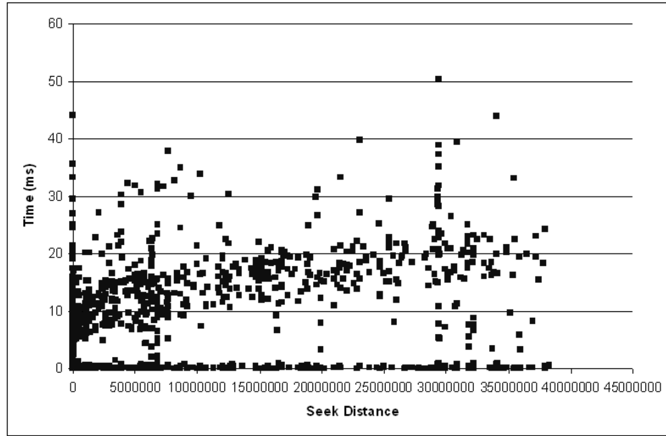


Figure A.17: Seek distance vs. time of the Linux boot without the ODSC.

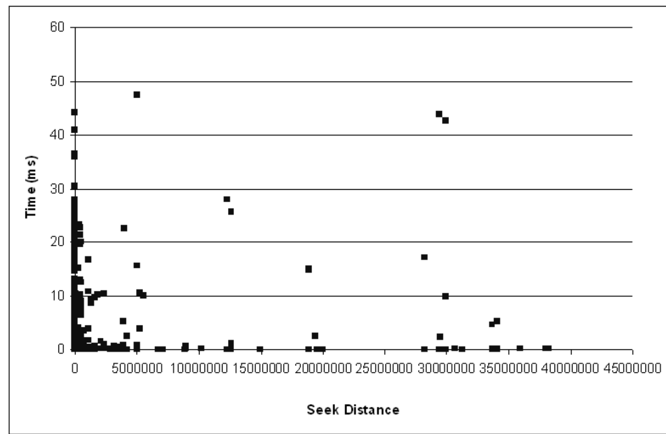


Figure A.18: Seek distance vs. time of the Linux boot with the ODSC.

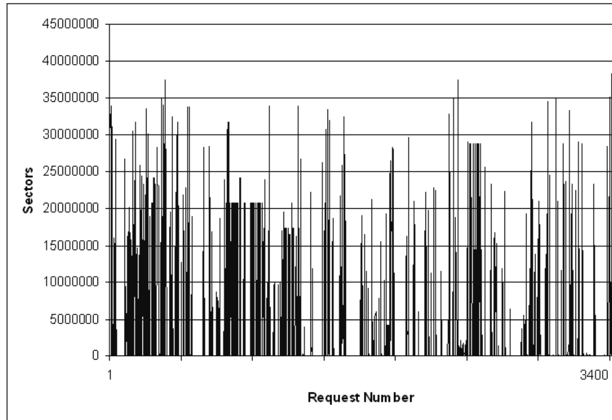


Figure A.19: Seek distance of a kernel make without the ODSC for each request. Each request may read multiple blocks.

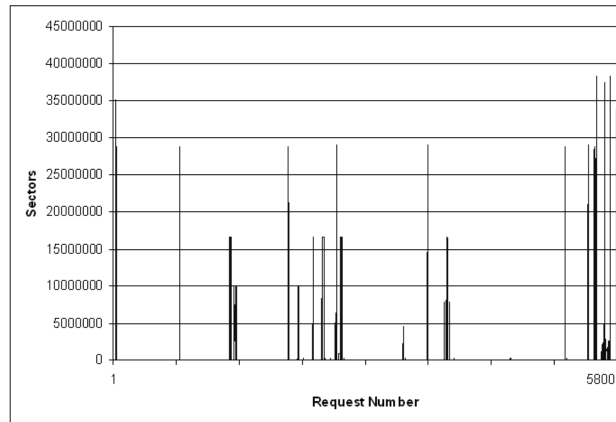


Figure A.20: Seek distance of a kernel make with the ODSC for each request. There are more requests than Figure A.19 because each request reads one block.

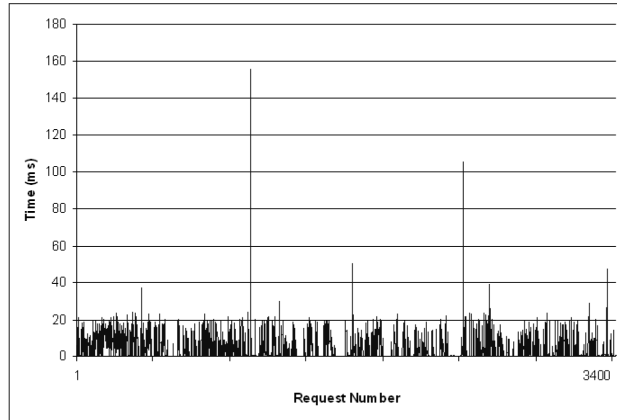


Figure A.21: Disk access times of a kernel make without the ODSC for each request. Each request may read multiple blocks.

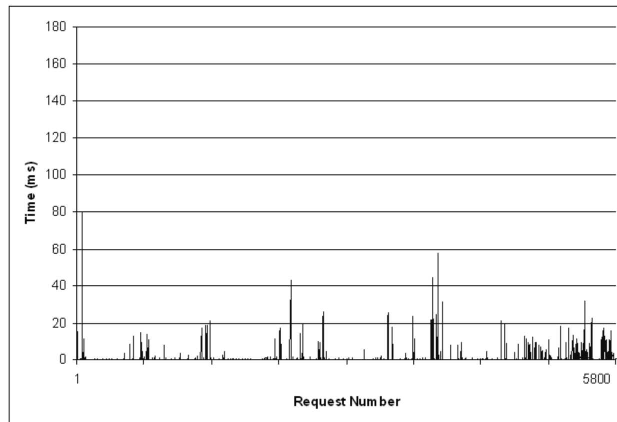


Figure A.22: Disk access times of a kernel make with the ODSC for each request. There are more requests than Figure A.21 because each request reads one block.

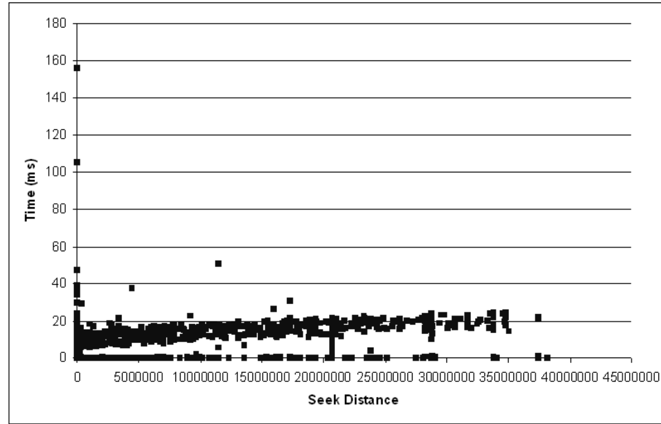


Figure A.23: Seek distance vs. time of a kernel make without the ODSC.

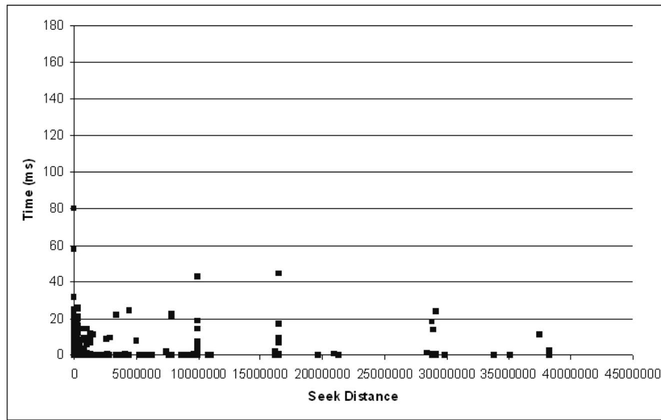


Figure A.24: Seek distance vs. time of a kernel make with the ODSC.

Bibliography

[1] C. Ruemmler and J. Wilkes, "Unix disk access patterns," in *USENIX Technical Conference*, January 1993, pp. 405–420.

[2] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 1st ed. Prentice Hall, 2003.

[3] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *IEEE Computer*, pp. 17–29, March 1994.

[4] S. W. Ng, "Advances in disk technology: performance issues," *IEEE Computer*, pp. 75–81, May 1998.

[5] S. Akyurek and K. Salem, "Adaptive block rearrangement," *ACM Transactions on Computer Science*, vol. 13, pp. 89–121, May 1995.

[6] H. Zhou, "A system-assisted disk [I/O] simulation technique," Master's thesis, Brigham Young University, December 1998.

[7] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. O'Reilly, 2002.

[8] N. Yin, "Reducing application load time by rearranging disk data," Master's thesis, Brigham Young University, August 1998.

[9] A. Peacock, "Dynamic detection of deterministic disk access patterns," Master's thesis, Brigham Young University, April 2001.

[10] S. W. Ng, "Improving disk performance via latency reduction," *IEEE Transactions on Computers*, pp. 22–30, January 1991.

[11] J. Ousterhout, "A trace-driven analysis of the unix 4.2 bsd file system," in *The Tenth ACM Symposium on Operating System Principles*, Orcas Island, Washington, USA, December 1985, pp. 15–24.

[12] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.

[13] S. T. R. Card, T. Ts'o, "Design and implementation of the second extended filesystem," in *The First Dutch International Symposium on Linux*, 1995.

[14] G. Oxman, "The extended-2 filesystem overview," http://nondot.org/sabre/os/S3FileSystems/Ext2fs-overview-0.1.pdf, August 1995.

[15] D. Lin, "Reducing consumption using disk data reorganization," Master's thesis, Brigham Young University, June 2000.

[16] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *1997 USENIX Annual Technical Conference*, Anaheim, California, USA, 1997.

[17] Z. Li, Z. Chen, S. Srinivasan, and Y. Zhou, "C-miner: Mining block correlations in storage systems," 2004.

[18] S. Akyurek and K. Salem, "Adaptive block rearrangement under UNIX," *Software - Practice and Experience*, vol. 27, no. 1, pp. 1–23, 1997.

[19] Y. Hu and Q. Yang, "DCD - disk caching disk: A new approach for boosting I/O performance," in *ISCA*, 1996, pp. 169–178.

[20] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson, "Trading capacity for performance in a disk array," in *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation*. San Diego: USENIX Association, 2000, pp. 243–258.

[21] A. Rubini and J. Corbet, *Linux Device Drivers*, 2nd ed. O'Reilly, 2001.

[22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kauffman, 2003.