A Study of
Data Partitioning and Prefetching
for Hybrid Storage Systems

by

Maliha Sultana

M.Sc., Bangladesh University of Engineering and Technology, 2007
B.Sc., Bangladesh University of Engineering and Technology, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate Studies
(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
September 2011
© Maliha Sultana 2011
Abstract

Storage system performance has received much attention since the early days of computing systems because of the mismatch between computing power and I/O access times. The invention of new technologies have increased storage system performance, but due to the cost-performance trade off no one type of storage media is capable to meet both performance and capacity requirements. This motivated us to study the impact of data management techniques such as data partitioning and correlated prefetching on I/O performance when two different non-volatile storage media are integrated into a computing system. First, we consider partitioning data blocks between two devices, where one device is significantly faster than the other. We assume that significantly faster performance also implies a significantly smaller capacity. Clearly not all data can be stored or cached in the faster device. Second, to improve performance of the slower device, we investigate if correlation-directed prefetching (CDP) may offer significant benefits. Although CDP has been studied previously, we look into some special aspects of it. We analyze how different block correlation analysis heuristics affect the performance of CDP.

We developed a simulator to study the effect of the different techniques when using devices with differing characteristics. Our results show that data partitioning can significantly improve storage system performance. For a hard disk and solid-state drive based system, we achieved 2–92% improvement for different traces. We also show that data partitioning based on application long-range block access patterns performs significantly better than caching temporal locality of references.

To evaluate the hybrid system in real world settings, we present a case study, a prototype data block manager for Linux-based systems that permits
Abstract

data to be partitioned across an SSD and an HDD. This partitioning is transparent to the file system and the block manager can also trigger data prefetches when there is high correlation between data block accesses.

Maliha Sultana. malihas@ece.ubc.ca
# Table of Contents

Abstract .................................................................................. ii

Table of Contents ................................................................. iv

List of Tables ............................................................... vii

List of Figures ............................................................... viii

Acknowledgments ............................................................... x

1 Introduction ................................................................. 1
  1.1 The storage hierarchy and data management techniques ... 3
  1.2 Motivation ................................................................. 5
  1.3 Overview of our work .................................................. 7
  1.4 Scope ......................................................................... 9
  1.5 Outline ................................................................. 10

2 Overview of Storage Management Techniques ................. 11
  2.1 Data partitioning ........................................................ 11
    2.1.1 Design choices .................................................. 13
  2.2 Correlation-directed prefetching .................................. 14
    2.2.1 Design choices .................................................. 15
  2.3 Integration of partitioning and correlated prefetching ....... 17

3 Workload Characteristics .................................................. 19
  3.1 Repetition ................................................................. 20
  3.2 Long-range overlap .................................................... 21
# Table of Contents

3.3 Non-uniformity of access ................................................. 22  
3.4 Think time .............................................................. 22  
3.5 Block correlation ......................................................... 23  

4 Simulation Study ............................................................ 25  
4.1 Data access pattern analysis ............................................. 26  
  4.1.1 The BORG heuristic .................................................. 27  
  4.1.2 Think time in block correlation analysis .......................... 28  
  4.1.3 The C-Miner heuristic ............................................... 29  
  4.1.4 The simulator .......................................................... 31  
4.2 Simulation results ......................................................... 33  
  4.2.1 Data partitioning ..................................................... 34  
  4.2.2 Correlation-directed prefetching ................................... 40  

5 A Case Study: HDD+SSD Storage for a Linux Based System .............................................. 48  
5.1 Solid-state drives .......................................................... 49  
5.2 HDD vs SSD: performance and price ................................... 50  
5.3 Implementation of data partitioning .................................... 51  
  5.3.1 I/O redirection approach ............................................ 53  
  5.3.2 Inode mapping approach .......................................... 55  
5.4 Limitations of partitioning ............................................... 57  
5.5 Implementation: correlation-directed prefetching ..................... 59  
5.6 Evaluation of prefetching ............................................... 60  
5.7 Limitations of prefetching ............................................... 63  

6 Literature Review .......................................................... 64  
6.1 Hybrid storage ........................................................... 65  
6.2 Re-organizing data layout .............................................. 67  
6.3 Disk I/O scheduling ...................................................... 68  
6.4 Prefetching ............................................................... 69  
  6.4.1 Sequential prefetching ............................................... 69  
  6.4.2 Application-directed prefetching .................................. 70  
  6.4.3 Heuristic-based prefetching ....................................... 70
Table of Contents

6.4.4 File prefetching ........................................... 71

7 Conclusion ....................................................... 73
  7.1 Future work ................................................. 75
    7.1.1 Write optimization .................................... 75
    7.1.2 Impact on energy consumption ....................... 76
    7.1.3 Minimization of fragmentation ....................... 76
    7.1.4 Start blocks in faster device ....................... 76
    7.1.5 Incorporation of correlated prefetching with caching in Linux ........................................... 77

Bibliography ...................................................... 78
List of Tables

3.1 Traces .................................................. 20
4.1 Disk specification ...................................... 34
5.1 Price comparison of SSDs and HDDs ................. 52
# List of Figures

1.1 Storage hierarchy ............................................. 5  
2.1 Data partitioning ............................................. 15  
3.1 Repetition in block accesses ................................. 20  
3.2 Overlap in block accesses ................................. 21  
3.3 Think time in traces ........................................... 23  
3.4 Non-sequentiality in block accesses ...................... 24  
4.1 Creating partition and prefetch lists from graph .......... 28  
4.2 Stages of the simulator ........................................ 32  
4.3 Impact of latency difference on partitioning ............. 35  
4.4 Percentage of blocks read from devices ................... 36  
4.5 Impact of bandwidth difference on partitioning .......... 37  
4.6 Data partitioning vs. caching ................................ 38  
4.7 With increasing SSD sizes:  
   (a) Improvement in response times,  
   (b) Percentage of blocks served from SSD ................. 40  
4.8 (a) Response times for different optimization techniques,  
     (b) Percentage of blocks served from devices ......... 42  
4.9 Performance of CDP-Slack .................................... 44  
4.10 Effect of prefetch cache size .............................. 45  
4.11 Chain of correlation:  
    (a) Hybrid system response times,  
    (b) Prefetched blocks as a percentage of total blocks ... 46  
5.1 SSD vs. HDD:  
    (a) Comparison of access times,  
    (b) Sequential read/write transfer performance ......... 51  
5.2 $/MB: SSD vs. HDD ........................................... 52
List of Figures

5.3 Data partitioning using I/O redirection ......................... 54
5.4 Inode remapping: (a) Linux file system inode structure, (b) Changing Inode pointers ............................... 56
5.5 Inode remapping: (a) Improvement in response times, (b) Percentage of data blocks read from each device ............... 58
5.6 Correlation-directed prefetching: (a) response times, (b) percentage of blocks read from devices .......................... 61
5.7 Prefetched pages: distribution of access ......................... 62
Acknowledgments

All praise to Allah, the most benevolent and the Almighty, for His boundless grace in successful completion of this thesis.

I would like to take the opportunity of thanking all who have been of great help and support during the period of my graduate studies.

First of all, I would like to express my sincere gratitude for Dr. Sathish Gopalakrishnan, who has been an excellent advisor — providing his thoughtful suggestions and constant guidance, and at the same time letting me work independently. I wish Sathish, his wife Harini and their son Vedant all the best in their lives. I would also like to thank Sathish for letting me use his MacBook which was of great help over the past two years.

The courses that I took at UBC not only enriched my theoretical background, but also introduced me to the evolving world of emerging technologies through various interesting course projects. I would like to thank all of the professors who put their effort in making the courses so interesting: Dr. Sathish Gopalakrishnan, Dr. Matei Ripeanu, Dr. Tor M. Aamodt, Dr. Charles “Buck” Krasic and Dr. Laks V.S. Lakshmanan.

I would like to thank Jorge Guerra, PhD Student, Florida International University (whom I met in SOSP’09) and other authors of the block reorganization project (BORG) for sharing their original code with me. Their source code helped me to get a better understanding of their algorithm which I studied in my thesis work.

I would like to thank my husband Nasa, who was there for me always — in my happy and tough times. My parents and parents-in-law were always the source of inspiration although they were half way around the world during my study period. I would like to thank my 5-year old nephew, Zayan, who helped me feel less homesick, although he never liked me working in my
laptop, and was always a source of much welcome and happy distraction.

Thanks to all of my labmates, Debojit, Sriram, Theepan, Bader, Layali, Karim and Joe, who always extended their help and support whenever needed, not only to me, but also to all others in the lab.
Chapter 1

Introduction

The storage system is a crucial component within a larger computing system, primarily because reading and writing data to some storage device is essential to computing. With the increase in computing power and the capability for highly parallel execution of programs, it is likely that performance of computing systems be limited by the latency and bandwidth bottlenecks involved in accessing data rather than the actual manipulation of data through computation. Storage system performance has received much attention since the early days of computing systems because of the mismatch between processing speeds and data retrieval/storage speeds but what is interesting is that input/output operations may become the bottleneck even for applications that have so far been considered compute-bound [17].

Storage system performance has been influenced, positively, by two areas of work: (i) the engineering of new materials and electronic devices for storing data and (ii) techniques for managing data usage at either the computer architecture or software system level.

Tape devices being replaced by magnetic hard drives, and hard drives themselves being superseded recently by solid-state disks utilizing non-volatile random access memory is one example of the first area of work influencing non-volatile memory. This area of work has also led to the developments that have improved the performance of volatile memory. Higher packing density is achieved in the the Dynamic Random-Access Memory (DRAM) requiring single transistor per cell as compared to the Static Random-Access Memory (SRAM) that requires six transistors per cell. The pipeline-based Synchronous DRAMs enable higher speed than the traditional asynchronous DRAMs and are now widely used in all types of computers.
Chapter 1. Introduction

Historically, volatile memory has always been faster, and more expensive in financial terms, than non-volatile memory and this distinction influenced the architecture of storage systems and the design of data management techniques. Much attention has been paid to data caching, and in particular non-exclusive caching, wherein data on slower, non-volatile storage could be read into faster memories when needed and with some level of prediction. The boundary between the performance of volatile and non-volatile memories has been blurred with the invention of flash storage and phase-change memories because these devices are non-volatile and yet support high read/write rates, sometimes to the magnitude of an order or two faster than other non-volatile storage devices and sometimes comparable to volatile storage technologies. This trend may require a new approach to data management.

The improvements in non-volatile storage technology have resulted in faster devices but the storage cost per gigabyte of data still favours traditional devices and therefore there is a move towards using a combination of devices to provide a balance between performance and cost of a system. In a manner in which the disparity in performance of on-chip cache memory and off-chip memory resulted in widespread adoption of data caching, similar approaches are becoming necessary for managing multiple types of non-volatile storage devices. With the emergence of anytime and anywhere computing thanks to ubiquitous data network access, non-volatile data storage can extend to network-based storage or cloud storage.

We have acknowledged a blurring of the divide between volatile and non-volatile memory and there have in fact been proposals for constructing entire storage systems using only non-volatile memory [29]. To explore the entire space of design choices for data storage systems spanning volatile and non-volatile memory is a challenging task and the work that will be discussed in greater detail is not an attempt at such a broad study. The focus of the work to be presented is restricted to non-volatile storage and we consider storage systems with two types of devices, with the devices differing in characteristics such as read/write latency and bandwidth. In this setting, of primarily a two-level non-volatile storage system, we seek some
answers to questions about the efficiency of software-level data management techniques such as caching, partitioning and prefetching. We restrict our attention to improving I/O latencies experienced by applications. Other system considerations such as energy consumption are important but we will only mention these issues in brief.

1.1 The storage hierarchy and data management techniques

In a computing system it is not uncommon to organize storage devices, characterized by their I/O latency, throughput, capacity and volatility, in a hierarchy with high-cost, high-performance and typically low capacity devices at the top (Figure 1.1). There is a cost-performance-capacity tradeoff that is made when deciding upon the storage hierarchy and on the capacity of the different devices. As an example, in a typical volatile memory organization, a small capacity, fast, on-chip cache is used to hold recently used data items under the assumption (that does hold true for many workloads) that recently used data is likely to be reused. Caching thus hides the lower performance of the off-chip RAM.

When we consider the storage hierarchy for non-volatile storage with different devices we are confronted with choices regarding the techniques to use to manage data across the devices. The primary choices are non-exclusive caching (which is the traditional approach to caching) and exclusive caching or data partitioning. In addition one can apply data prefetching. We shall now elaborate on these ideas in some more detail.

Non-exclusive caching is the common approach to managing data across two layers in the storage hierarchy. The faster device, which is also of lower capacity than the slower device, retains some of the recently read or written data items. When the caching device is full and new data needs to be cached then a data eviction policy is used to determine the data objects that will be replaced by the new data objects. Data that is written to the cache is also transferred to the slower device so that there is no loss of data when
1.1. The storage hierarchy and data management techniques

objects are evicted from the cache. Because the same data object resides in
the cache and in the slower storage device, this approach is *non-exclusive*.

**Exclusive caching** also relies on the faster storage device to retain
recent or frequently used data objects. Unlike non-exclusive caching, a data
object is only on one device at any point in time. Exclusive caching is
difficult to implement correctly in with volatile memories because of the
possibility of data loss (e.g., if power supply to the devices is affected). When
both devices are non-volatile then exclusive caching can be implemented and
implementation requires a *data partitioning* policy that determines where a
data object should be located. Data partitioning is performed when a system
is offline or occasionally because the overhead to moving data objects across
two storage devices is significant.

We will use the word *caching* to mean non-exclusive caching unless quali-
fied appropriately. Caching typically takes into account temporal and spatial
locality of reference. Data objects that are allocated to successive memory
addresses are retrieved when one memory access is made with the assump-
tion that data accesses are either repeated or involve consecutive memory ad-
dresses. A typically-used cache eviction policy is least-recently-used (LRU)
although a random replacement policy is also used when the overhead of
implementing LRU is considered high.

Caching does not take into account long-term dependencies between
memory accesses. Better predictions of future memory accesses can be
made by logging memory accesses to find access patterns or one can use
application-level hints. Such methods may be better suited to data parti-
tioning because the process of finding patterns is time-consuming and is,
therefore, difficult to implement at runtime with low overheads.

**Prefetching** In addition to caching or data partitioning, *data prefetching*
is a mechanism that can be implemented in conjunction with both schemes.
Data prefetching involves making predictions about future memory accesses
and moving data objects that are likely to be used in the near-future from
the slower device to the faster device thereby masking the slower device.
Prefetching can be effective if the prediction accuracy is high and if it is
1.2. Motivation

Our work is motivated by studies concerning disk access patterns that establish non-uniform access frequencies across data blocks, temporal and spatial locality in block accesses, and the presence of correlation even in random access patterns [10, 25]. By analyzing 7-day traces collected from a web...
1.2. Motivation

server, an SVN server, a developer and an office workload, Bhadkamkar et al. [10] have shown that for all of these workloads only 4.5–22.2% of the total storage was accessed during the trace collection period. These researchers have also found substantial overlap in block accesses over the 7-day period. Among the non-sequential accesses 15.55–65.42% accesses are repeated more than once. The non-uniform access of data blocks as well as overlap in block accesses certainly makes it beneficial to partition data blocks between two devices with different performance characteristics (for example, a low latency and a high latency device) and optimizing the file system so that we can extract the benefits of both as well as overcome the shortcomings of the same. On the other hand, the presence of block correlation among non-sequential accesses motivated us to study correlated prefetching in more detail and utilize application think time for prefetching. We integrate these observations with the existing trend of integrating different types of storage in computing systems today. Some systems include an HDD and an SSD, some include an HDD that uses a small flash memory-based cache. Others permit users to map data on network-based storage services.

Although caching has been extensively used in many computing systems, partitioning data blocks based on access patterns is only studied by a few and all those studies are limited to re-organizing data blocks on the same hard disk [7, 15, 22, 33]. Data partitioning between two devices were not studied mainly because we did not have the right devices before. For example, partitioning data blocks between a hard disk and a DRAM is not a good idea because of a few reasons. First of all, DRAM is volatile. However, we can always do non-exclusive partitioning, but then arises the second issue — DRAM is very expensive. A separate DRAM for data partitioning increases the overall cost of the computing system significantly. On the other hand, using the same DRAM for both partitioning and as the main memory results in space conflicts as we need main memory for heaps, executables etc. With the invention of new technologies, for example, flash memory, emerges the notion of data partitioning. As flash memory based SSD devices are non-volatile, have significantly better access times than hard disks, and are also significantly less expensive than DRAMs, it is now time to study partitioning
data blocks between a hard disk and an SSD. In a system with an HDD and an SSD, how should one allocate data to the SSD? How much benefit can a small flash memory-based cache offer when compared to a system with a separate HDD and SSD? These are a few of the system design considerations that our study will have some bearing on.

1.3 Overview of our work

We study the impact of different data management techniques (caching, partitioning and prefetching) on the I/O performance observed by applications when at least two different non-volatile storage media are integrated into a computing system. We consider the benefits of reasoning about long-range dependencies between data accesses in an application when partitioning and prefetching data. For this type of analysis, we use traces of data accesses from different applications.

First, we consider partitioning data blocks between two secondary storage devices, where one device is significantly faster than the other. We assume, based on pricing data \([4]\), that significantly faster performance also implies a significantly smaller capacity. Clearly not all data can be stored or cached in the faster device. Second, to improve performance of the slower device, we investigate if correlation-directed prefetching may offer significant benefits. With this technique, we read correlated blocks ahead of time from the slower device when a related block is read.

We have built an event-driven simulator to help us study the effect of the different techniques when using devices with differing characteristics. We have also implemented, as a case study, a prototype data block manager for Linux-based systems that permits data to be partitioned across an SSD and an HDD. This partitioning is transparent to the file system and the block manager can also trigger data prefetches when there is high correlation between data block accesses. In our simulation study and the prototype case study, partitioning and prefetching are based on application-level traces or history information concerning data accesses.

The objective of this work is to study different aspects of partitioning
1.3. Overview of our work

and correlated prefetching as well as workload characteristics so as to pro-
vide some guidelines on what is beneficial and why, for different types of
workloads and storage systems. The broader questions that we want to
answer in this work are organized in three groups and are presented below:

Workload characteristics

1. How much repetition, overlap and random block correlation do we see
   in the long term working set of workloads?

2. How can we utilize these access patterns for improving file system
   performance?

Partitioning

1. How much performance improvement can we get by partitioning data
   blocks between two devices with varying latency and bandwidth?

2. What could be the limitations of partitioning data blocks?

3. How does partitioning perform as compared to caching?

4. By how much does the difference between latency and bandwidth of
   the two media affect partitioning?

Correlated prefetching

1. Can prefetching based on block correlation improve performance over
   partitioning?

2. Analysis of different block correlation mining techniques present in
   literature.

3. How can we utilize the think time between two accesses in prefething
   correlated blocks?

4. Instead of prefetching the immediate correlated block, will it be ben-
   eficial to prefetch the next block in the chain of correlations?
5. How does data partitioning affect the performance of prefetching?

We will briefly discuss some of our findings here. The rest are answered through our simulation-based study (Chapter 4) and Linux data block manager implementation (Chapter 5). We found, both in our simulation and in the Linux system, that partitioning data blocks between two devices with varying latency significantly improves performance over the slower device-only system. Our evaluation shows that partitioning data blocks between an SSD and an HDD improves the performance of the Linux file system by up to 79%. We also found that, although, caching does a very good job of reducing disk-only read response times, data partitioning performs even better. This is because data partitioning utilizes application long-range block access patterns, such as non-uniformity of access, repetition and long-range overlap, whereas caching utilizes only short-range patterns such as temporal locality of references. We show that data partitioning improves file system read performance over LRU-based caching by up to 64%. Data partitioning is a more suitable choice than caching for SSD-based systems. This is because caching involves continuous updating of blocks in the SSD which may wear out the device more quickly. The poor performance of SSD small writes may also affect the performance of caching.

When implemented together, correlation-directed prefetching and data partitioning improves read response times slightly over a system that employs data partitioning only. We found that the difference between block correlation analysis heuristics have negligible impact on prefetching performance. We analyze some other aspects of prefetching that we describe in more detail in Section 4.2.2.

1.4 Scope

We only consider data read performance offered by secondary storage devices. We have not attempted to evaluate write performance.
1.5. Outline

We present an overview of the data management techniques that we study in this work in Chapter 2. We discuss our workloads and their characteristics in chapter 3. We study the effect of data partitioning and prefetching through an event driven simulator in Chapter 4. The case study, an HDD combined with an SSD in the Linux kernel is presented in Chapter 5. Related work is described in Chapter 6. Chapter 7 concludes the thesis with some guidelines for future work in this area.
Chapter 2

Overview of Storage Management Techniques

Today’s increasing need for storage capacity and I/O performance has led to the development of new storage technologies and data management solutions. However, we are yet to see a single device that excels in all areas of storage requirements, such as cost, performance and capacity. This motivated the move towards using a combination of devices to provide a balance between performance and cost of a system. In this work, we study such a storage system combining two different types of non-volatile storage devices with different capacities and performance characteristics. We study the impact of two storage management solutions in the hybrid storage system: (i) *data partitioning* partitions the file system data blocks between the two devices while providing the illusion of having a single device (Section 2.1) and (ii) *correlation-directed prefetching* improves the performance of the slower device by prefetching correlated blocks when a related block is read (Section 2.2). Although a number of studies focused on predicting file correlations and prefetching correlated files ahead of time, the finer granularity of block-level prefetching based on non-sequential correlation is studied by a few [25]. In this work we investigate different aspects of correlation-directed prefetching and investigate how data partitioning provides some design options for improving the performance of the same.

2.1 Data partitioning

Data partitioning represents the notion of splitting file system data blocks between two non-volatile storage media with significant differences in perfor-
2.1. Data partitioning

Partitioning can be advantageous when one storage device is significantly faster than the other. From the cost-performance-capacity trade off discussed in Chapter 1, we know that high performance storage devices are usually more expensive and hence are usually available in smaller capacities only. The basic idea of data partitioning is to split file system data blocks between two different storage devices in such a way that a relatively smaller high performance device (lower latency, higher throughput, higher cost) gets most of the read requests and a significantly larger low performance device (higher latency, lower throughput, lower cost) meets the capacity requirements. Such a combination of a relatively small high performance device (HPD) and a large low performance device (LPD) provides the illusion of having a HPD with data capacity as high as the underlying LPD.

Partitioning is related to caching in the way that caching also tend to hide the slower performance of a storage device by predicting future references and storing those data blocks in a faster storage device. Caching usually takes into account the short term working set, more specifically only temporal and spatial locality of references. However, applications’ long-range block access pattern show significant data repetition, overlap in block accesses and block correlation [10]. These characteristics are not exploited by traditional caching policies such as least-recently-used. The goal of data partitioning is to utilize these long-range access patterns to improve file system performance.

An example of data partitioning could be a desktop or a server storage system that partitions its data blocks between a solid-state drive (SSD) and a hard disk drive (HDD). Cloud storage is another example where partitioning would be beneficial. Bandwidth is the most important bottleneck for utilizing cloud storage for day to day use. But an effective partitioning of data blocks between a cloud storage and a local storage (for example, an HDD/SSD) can beat that bottleneck. Mobile and hand held devices usually have smaller on device storage. These devices can provide the illusion of having more storage than installed on the device by partitioning their data between a server and the built-in storage. The server would be used for
2.1. Data partitioning

storing maximum data, whereas the built-in storage would hold the long
term working set as well as would prefetch data blocks from the server when
needed.

2.1.1 Design choices

A number of issues need to be addressed for an efficient implementation of
data partitioning.

Partitioning granularity  Data partitioning can be done in a file-level
or in a block-level granularity. Depending on the application charac-
teristics, certain data blocks within a file may be accessed more often than
others. It is, therefore, advantageous to partition data blocks inside files
between the two devices.

Exclusive vs non-exclusive partitioning  Because of the possibility of
data loss (e.g., if power supply to the devices is affected), the volatile stor-
age devices, used in most of the caching technologies, prefer non-exclusive
caching. Data partitioning, however, employs two non-volatile storage de-
vices and hence exclusive partitioning would result in more efficient use of
the storage available. Exclusive partitioning also simplifies updating of data
blocks as a data object is only on one device at any point in time.

Partitioning interval  Data partitioning involves analyzing of application
block access patterns and moving of large amount of data between the two
participating storage devices. The partitioning interval should be sufficiently
long so that we can capture the access pattern as well as minimize the
overhead associated with data movement. In this work, we used week long
traces and partitioned data blocks based on the I/O activity of the first half
of the traces, which represents 3/4 days of data access pattern.

Partitioning strategies  The most important question that needs to be
answered regarding partitioning is: based on what criteria should we split
the data blocks between the two devices? We certainly want most of the
read requests to be served from the high performance device which leads to the choice of splitting data blocks based on block access frequency. The most frequently accessed blocks should reside in the faster device and the rest should be in the slower device.

Now arises the question as to how to monitor the frequently accessed blocks. One option could be to monitor the block accesses online and keep counters for each block that would be updated whenever a block is accessed [33]. The problem with online monitoring is that the updating of block counters introduce an additional time and space overhead with each block access. The other approach is to collect block access streams for a sufficiently long period and analyze the access streams to find the most frequently accessed blocks. Once the access stream is collected, we can simply count the access frequencies of each block in the access stream. We can also represent the access stream as a weighted graph and find the most connected blocks from that graph [10]. Both of these approaches find the blocks that are accessed more often than others. In this work, we moved the most connected blocks to the faster device.

In addition to moving the frequently accessed blocks, one can also move the first few blocks of each application/file to the faster device and prefetch the rest of the blocks as soon as these start blocks are read. This way the files or applications would have a quick launch time as well as quick response times if the rest of the data blocks can be prefetched in time. We, however, do not study the impact of this approach in this work.

2.2 Correlation-directed prefetching

Block correlations are commonly found in storage systems. The very basic form of block correlation is the sequential correlation representing spatial locality of reference. Storage systems usually employ some read ahead techniques that utilize this type of correlation by prefetching a range of contiguous blocks when a block is read. Sequential prefetching can improve performance when data are accessed sequentially by applications, such as media applications.
2.2. Correlation-directed prefetching

Block correlation can also be noticed in non-sequential access patterns [10, 25]. In databases represented as trees, the tree nodes are correlated to their parent nodes, in file systems a data block is correlated to its inode block and a file inode block is correlated to its directory inode blocks. In a web server, a web page is correlated to its sub-pages, images and scripts. In an integrated project development environment, certain files are accessed in some pre-deterministic order that also reveals correlation. The correlated blocks tend to be accessed together; a property that has been utilized to re-organize disk layout so that correlated blocks are placed together and can be fetched together by only one disk access [10, 25]. The non-sequential block correlation can also be utilized by the correlation-directed prefetching technique, where a range of correlated blocks are prefetched when a related block is read [25]. In this work, we investigate different aspects of correlation-directed prefetching.

2.2.1 Design choices

Correlation-directed prefetching (CDP), if done efficiently, has the potential to improve the storage system performance over sequential prefetching. It, however, depends on a few factors such as how much correlation is present in certain workloads and how accurately we can predict them. CDP also interferes with the demand requests when accessing the underlying stor-
2.2. Correlation-directed prefetching

age devices and may even degrade performance, if not configured carefully. An efficient implementation of CDP technique requires investigation of the following issues.

**Mining block correlation** Although block correlations commonly exist in storage systems, finding them is not an easy task. Depending on accuracy, transparency and generality, block correlation mining techniques can be divided into two broad categories: (i) *application-directed*, and (ii) *heuristic-based* (Section 6.4). The first technique uses hints from the applications themselves to find correlated blocks [31, 38]. This approach results in more accurate prediction, but it requires modification of the applications. The fact that correlated blocks tend to be accessed together is utilized by the *heuristic-based* techniques where information about past block references are collected for a reasonably long time and future block correlations are predicted based on that [10, 25]. In this work, we investigate the latter approach because it is transparent to the applications and to the file system, although it may not be as accurate as the application-directed one.

Different heuristics are available for mining block correlation from access history. One can represent the block access stream as a weighted graph where vertices represent blocks or block ranges and edges represent block correlations [10]. Another approach is to represent access streams as database sequences and apply frequent sequence mining algorithms to find block correlations [25]. Although both heuristics work towards finding block correlations in access streams, their perspectives are different. We seek to analyze how this difference in perspectives affect CDP and hence we investigated both of these heuristics in this work (detailed description of these heuristics in Section 4.1.1 and Section 4.1.3).

**Effect of think time** Another aspect that needs to be studied in mining block correlation is the effect of application think time. Think time is the time an application needs to process received data before submitting a subsequent request for more data. Correlated blocks tend to be accessed together not only in space but also in time. This motivates us to include think time
in block correlation analysis. We modified the graph-based heuristic [10] to include think time, in addition to access frequency, as a guiding parameter for analyzing block correlation (Section 4.1.2).

Prefetching from chain of correlation Instead of prefetching the immediately correlated block, in some cases, it might be beneficial to look ahead in the chain of correlation and prefetch the secondarily correlated blocks. For example, for the correlation chain $A \rightarrow B \rightarrow C \rightarrow D$, it might be beneficial to prefetch block $C$ when block $A$ is read and to prefetch block $D$ when block $B$ is read and so on. This way both block $C$ and block $D$ will be in memory when needed even if the think time between $B \rightarrow C$ or $C \rightarrow D$ is low. Prefetching from the chain of correlation essentially gives us more time to prefetch, whereas prefetches the same sequence of blocks.

2.3 Integration of partitioning and correlated prefetching

Both data partitioning and correlation-directed prefetching techniques have potentials for improving storage system performance. Furthermore, data partitioning changes the way storage systems behave and introduces new design options for correlation-directed prefetching.

One option is to have a hybrid storage system that has both data partitioning and correlation-directed prefetching installed – the only coordination between the two is that if a block is present in the faster device then it would not be prefetched by the CDP system. This is because serving the block from the faster device is already low overhead.

Another option is to utilize the slack that data partitioning might introduce for correlation-directed prefetching. With data partitioning installed, we can expect that some of the requests will completely be served from the high performance device (or from memory if prefetching is installed too). Since the latency of the high performance device is expected to be far far lower than the low performance device, serving a request completely from the high performance device results in much lower response time. This gives
us some *slack* that we can utilize for issuing prefetching requests. In other words, we can issue prefetching requests only when a request do not access the low performance device.

We can also partition data blocks in such a way that some of the correlated blocks of the frequently accessed blocks reside in the high performance device. We can use think time as a guidance for both partitioning and prefetching. For example, if two correlated blocks have very small think time then it may not be wise to issue prefetch request of the correlated block as the request may not be complete in time. On the other hand, CDP would be beneficial when the think time between two accesses are large enough so that prefetching can be completed in time. To efficiently utilize such cases, in addition to moving the frequently accessed blocks, we can also move some of their correlated blocks with small think times to the faster device and issue CDP requests for those with large think times.

In this work, we implemented CDP on top of data partitioning and investigated all of these options.
Chapter 3

Workload Characteristics

Researchers have pointed out many interesting characteristics that applications’ block access patterns exhibit, for example, (i) locality of reference: in particular, temporal and spatial locality, (ii) long-range overlap in block accesses: some blocks remain significantly popular for relatively longer periods of time, (iii) non-uniform access frequency distribution: only a small portion of the total file system is accessed frequently and (iii) block correlation: even non-sequential accesses are sometimes related and are found to be repeated [10, 25, 33, 40]. All of these characteristics affect the performance of the hybrid storage system that we investigate in this work. We want to see by what extent these characteristics are present in real world workloads and to determine how these characteristics can be utilized to improve the performance of the data partitioning and correlation-directed prefetching techniques. We are particularly interested in the long-range access patterns, namely repetition (Section 3.1), overlap with previous accesses (Section 3.2), non-uniformity of access (Section 3.3), think time (Section 3.4) and block correlation (Section 3.5).

In our analysis of workloads, we used real traces collected from Microsoft Research enterprise servers at Cambridge (Table 3.1). These traces are collected by Narayanan et al. [28] and are available in the SNIA IOTTA repository [6]. The Web trace in Table 3.1 is collected by Bhadkamkar et al. as part of their block re-organization project, BORG [10]. These traces are also used in all of our experiments.
### 3.1 Repetition

The partitioning of data blocks between a faster and a slower device is based on the observation that all blocks are not accessed with equal frequency. Some blocks are accessed more frequently than others and we can move these blocks to a device with lower latency for better performance. Interestingly, all the traces that we analyze here showed significant repetition rate (Figure 3.1). Other than the media server trace from MSR, all traces show 41–73% repeated accesses which clearly indicates that the data partitioning strategy would be beneficial for these types of workloads.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Description</th>
<th>Size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web</td>
<td>CS web server</td>
<td>13.70</td>
</tr>
<tr>
<td>Usr</td>
<td>User home directories, MSR enterprise Server</td>
<td>35.30</td>
</tr>
<tr>
<td>Proj0</td>
<td>Project directories, MSR enterprise Server</td>
<td>8.97</td>
</tr>
<tr>
<td>Proj3</td>
<td>Project directories, MSR enterprise Server</td>
<td>18.23</td>
</tr>
<tr>
<td>Hm</td>
<td>Hardware monitoring, MSR enterprise Server</td>
<td>9.96</td>
</tr>
<tr>
<td>Src</td>
<td>Source control, MSR enterprise Server</td>
<td>8.82</td>
</tr>
<tr>
<td>Mds</td>
<td>Media server, MSR enterprise Server</td>
<td>3.25</td>
</tr>
</tbody>
</table>

Table 3.1: Traces

![Figure 3.1](image.png)

Figure 3.1: Repetition in block accesses. More than 40% of the blocks are accessed at least twice for all traces other than the media server trace.
3.2 Long-range overlap

![Chart showing overlap of the second half of the traces with the first half and with the top 20% accesses of the first half of the traces.](image)

**Figure 3.2:** Overlap of the second half of the traces with the first half and with the top 20% accesses of the first half of the traces. This shows us that some blocks remain significantly popular over the entire week of the trace collection period.

Bhadkamkar et al. [10] analyzed data overlap in week long traces representing a web server, a version control server, an office and a developer workload. They observed significant overlapped accesses across successive days with day 1. Their analysis shows that some blocks remain popular for significantly longer periods of time. Data partitioning depend on this long-range block popularity as we partition based on previous accesses.

To analyze overlap in the MSR traces, we divided the traces into two halves and computed overlap of the second half with the first half of the trace. This is because, we capture access patterns from the first half of the traces and run experiments with the second half of the traces. We observed that for most of the traces, more than 60% of the accesses in the second half overlaps with the first half of the trace (Figure 3.2). The exceptions are the Mds trace and the Proj3 trace where the overlap with the first half of the trace is 9.27% and 0.71% respectively. The limited capacity of the high performance device allows for only a small portion of the frequently accessed blocks. Therefore, we computed whether there is any overlap of the second half of the trace with the top 20% accesses of the first half of the trace. We found significant overlap, 34%–99%, with the top 20% accesses of the first
3.3 Non-uniformity of access

Block accesses are distributed non-uniformly over the entire file system [10, 33, 40]. Bhadkamkar et al. [10] found that only 4.5–22.3% of the file system data were accessed by different workloads during the one week trace collection period. Unfortunately we do not have enough data to compute the non-uniformity of access for the MSR traces. However, we observed that the trace sizes representing read requests are significantly small — the smallest one being the Mds trace representing a media server (3.25GB) and the largest one being the Usr trace representing a user directory server (35.3GB) (Table 3.1). These trace sizes are much smaller than a typical file system size in enterprise servers which indicate that a very small portion of the file system was actually accessed during the one week trace collection period.

3.4 Think time

Think time plays an important role in correlation-directed prefetching. If the think time between two correlated accesses are low, then prefetching might not be completed in time. Also, since prefetching interferes with the demand requests, low think times between demand requests introduce additional overhead from prefetching.

We studied the think times between two consecutive accesses in the traces. As the traces have millions of requests, it is difficult to plot all the think times. We thus divided the think times into a few categories, for example less than or equal to 1 ms, greater than 1 ms but less than or equal to 5 ms and so on (Figure 3.3). Our analysis of the MSR traces show that more than 50% of the accesses in all traces have 1 ms or lower think times. Whether this 1 ms think time is ‘too small’ or ‘large enough’ for prefetching depends on the type of storage media and data management solutions used.
For example, enterprise-level magnetic disk drives usually require around 5 ms access time. On the other hand, solid-state drives require only 0.1 ms.

### 3.5 Block correlation

Sequentiality is a special type of block correlation that represents spatial locality of reference. Typical storage systems usually employ some sort of sequential prefetching technique that can make sequential workloads’ performance significantly better than non-sequential ones. In this work, we analyzed sequentiality separately from block correlation because we need to know the degree of sequentiality the workloads have. If the workloads show very high degree of sequentiality then only sequential read ahead would be enough to improve storage system performance. On the other hand, mostly random accesses in the access streams would not be as benefited from sequential read ahead and motivate us in digging other types of block correlation so that we can minimize disk access time by prefetching correlated blocks ahead of time.

Two accesses are considered sequential if they are at most $k$ blocks apart. $k = 0$ indicates strict sequentiality where the access gap is zero blocks be-
3.5. Block correlation

![Graph showing % Randomness for different values of k]

Figure 3.4: Non-sequentiality in block accesses. \( k \) defines the maximum access gap up to which two accesses are considered sequential. More than 50% of all the accesses in all the traces are non-sequential even if we consider \( k = 256 \).

We measured the sequentiality of traces for different values of \( k \), for up to \( k = 256 \). We found that almost all of the traces show a very high degree of randomness (Figure 3.4); even for \( k = 256 \), 51–84% of the accesses are non-sequential in different traces. For all of the traces, more than 80% of the accesses are non-sequential when \( k = 0 \). A more accurate analysis would be to analyze sequentiality on a per process basis instead of considering the whole trace as a single stream. Since the MSR traces do not have process information we were not able to analyze this way.

As discussed in Section 2.2 even non-sequential accesses can show significant block correlation that can be utilized to reduce disk access latency by prefetching correlated blocks when a related block is read. We analyzed block correlations in the workloads using two heuristics: (i) BORG, a greedy heuristic based on maximal weight, presented by Bhadkamkar et al. [10] and (ii) C-Miner, a heuristic based on frequent sequence mining algorithm, presented by Li et al. [25]. The heuristics are described in more detail in Section 4.1 and their effectiveness in correlation-directed prefetching is studied through our simulation experiments.
Chapter 4

Simulation Study

Our analysis of workload characteristics (Chapter 3) shows that long-range block access patterns have interesting characteristics that can benefit both data partitioning and correlation-directed prefetching techniques. To understand the behavior of the hybrid storage system for different types of devices with different types of system settings we need a simulation-based study as it gives us flexibility and control over the environment. We seek to understand how the difference in device characteristics affect the performance of data partitioning, how data partitioning performs as compared to caching, how much space we need in the faster device to achieve a certain performance target and how data partitioning can be integrated with correlation-directed prefetching. To answer these questions, we developed a discrete event simulator (Section 4.1.4) that simulates two storage devices with varying latency and bandwidth, an OS cache for storing temporal and spatial locality of references and a prefetch cache for storing correlated blocks that are read ahead of time. We used manufacturer specified characteristics for the devices we simulate (Table 4.1). The traces we used represent a real web server [10] and Microsoft Research server workloads [6] (Section 3.1).

Inputs to the simulator are a partition list specifying which blocks should be moved to the faster device and a correlation list describing which blocks to prefetch when a related block is read. We generated the partition list using a greedy heuristic, BORG [10] that finds the most connected blocks from access history (Section 4.1.1). We seek to analyze how different block correlation analysis techniques affect the performance of CDP and hence implemented CDP with two heuristics: (i) BORG [10], a greedy heuristic based on maximal weight, and (ii) C-Miner [25], a correlation analysis heuristic based on a frequent sequence mining algorithm (Section 4.1.1 and
Correlated blocks tend to be accessed together within a short distance and also within a short time. While most heuristics are based on access distance only, we want to see whether access time has any effect on block correlations and hence we modified the BORG heuristic to include access time in block correlation analysis (Section 4.1.2).

4.1 Data access pattern analysis

Both data partitioning and correlation-directed prefetching depend on an application’s long-range block access patterns. For data partitioning we need to generate a partition list that specifies the location of each block. In other words, the partition list specifies which blocks should be moved to the high performance device. For correlation-directed prefetching, we need to generate a prefetch list that specifies which blocks are correlated so that when a block is read we can prefetch its correlated blocks ahead of time.

A number of different techniques are available to analyze access patterns and to find block correlations (Section 6.4). Some of these techniques use hints from application front-end, some use access history to generate correlation. Although hints from applications themselves can result in more accurate prediction, these approaches are not transparent to the applications. Therefore, in this work, we investigated only the access history-based block correlation analysis techniques. We used two different heuristics in our simulations to see how different block correlation analysis techniques affect correlation-directed prefetching. In particular, we used, BORG, a greedy heuristic based on maximal weight [10], and C-Miner a heuristic based on a frequent sequence mining algorithm, [25] described in more detail below. These two heuristics differ in a number of ways, for example, the BORG heuristic takes process information into account while C-Miner does not. Unfortunately we could not see the impact of this difference in our experiments as the traces we used do not include process information.
4.1. Data access pattern analysis

4.1.1 The BORG heuristic

Bhadkamkar et al. [10] have analyzed block correlation to create a self-organizing correlation-directed data layout on disk. Block accesses from a request stream are used to build a directed process access graph \( G_i(V_i, E_i) \) for each process \( i \). Each edge \((u, v)\) in the access graph represents two consecutive requests by the same process where \( u \) and \( v \) denote LBA ranges and the edge weight represents frequency of access. Once all the process graphs are ready, a master access graph \( G(V, E) \) is created from the process access graphs that captures all correlation. The master access graph splits partially overlapped LBA ranges whenever necessary so that each LBA range is represented by at most one vertex (Figure 4.1).

A greedy heuristic generates the correlation-directed layout from the master access graph. For each connected component in the graph, it first chooses the most connected vertex \( u \) with the maximum sum of incoming and outgoing edges. Next, it choses the most connected vertex \( v \) to \( u \) in incoming or outgoing direction and places \( v \) before or after \( u \) respectively. Then the vertex that is most connected to the group \( u \cup v \) is placed and so on.

Note that this process actually generates a placement of blocks instead of block correlation rules. We modified the algorithm slightly to generate the block correlations that can be used for the data partitioning and prefetching stages of the hybrid storage system. In particular, we need a partition list representing the blocks to be moved to the high performance device and a prefetch list representing block correlation rules in the form of \( u \rightarrow v \) so that when block \( u \) is accessed we can prefetch block \( v \). Generating the partition list is straightforward, we move the most connected vertices to the list in decreasing order of connectedness as described in the BORG heuristics. For the prefetch list, for each vertex \( u \), we generated the list of vertices \( v \) that share an outgoing edge with vertex \( u \) — these vertices represent the list of probable future accesses after \( u \) is accessed. The prefetch list contains these correlation rules in order of decreasing access frequency (Figure 4.1).
4.1. Data access pattern analysis

Figure 4.1: Creating partition and prefetch lists from the master access graph. Vertices represent (start LBA, length of request) and edges represent correlation. For data partitioning, we list the vertices in decreasing order of connectedness (the original BORG heuristic creates a disk layout from the graph). Here, vertex B is most connected having the maximum sum of incoming and outgoing edges. The complete partition list, in decreasing order of connectedness is: B, E, D, C, E and F. In the correlation list, we generate all correlations (represented by outgoing edges) in decreasing order of frequency. Here, the correlations are: A → B, B → E, B → D, C → B, D → E, D → A, E → C, E → F, F → E.

4.1.2 Think time in block correlation analysis

Most heuristics analyze block correlation based on access distance only, whereas think time also plays an important role in block correlation. Think time is the elapsed time between two subsequent requests — it represents the time an application needs to process received data before submitting a subsequent request for more data. Our observation is that smaller think times indicate stronger correlation as correlated blocks tend to be accessed together within a short period of time. On the other hand, a very large think time between two accesses may indicate a coincidence rather than a correlation. In this work, we study the effect of application think time in block correlation analysis by slightly modifying the BORG [10] heuristic.

In BORG, two LBA ranges are considered as correlated whenever they are accessed together by the same process — the higher the access frequency is, the more correlated they are. This correlation is represented by annotating the edge weights with access frequency. We re-defined edge weights to
4.1. Data access pattern analysis

be a function of both access frequency and think time. In our experiments we have used the following simple function:

\[
\text{Edge weight} = \frac{\text{Access Frequency}}{\text{Average Think Time}}. \tag{4.1}
\]

This representation of edge weight prefers smaller think times. Our motivation is that if two blocks are accessed together with high frequency but large average think time then it may not be wise to prefetch the correlated block when one of them is read because the prefetched block may get prematurely evicted. On the other hand, smaller think time indicates that the correlated block is going to be accessed soon and it would improve storage system response time if that block is already present in memory. We used this modified BORG heuristic in prefetching only, it does not affect our partitioning approach.

4.1.3 The C-Miner heuristic

Li et al. proposed C-Miner [25], a block correlation analysis technique that is based on frequent sequence mining algorithm and utilized it for correlation-directed prefetching and correlation-directed disk layout generation. Their simulation studies on real workloads demonstrated a reduction of average I/O response time by 7–20% compared to sequential prefetching technique.

C-Miner is based on CloSpan, a closed sequence mining algorithm [41]. Sequence mining identifies frequent subsequences, i.e., the subsequences that occur in at least a pre-specified number (called min_supp) of sequences in a given sequence database. A closed sequence is a subsequence whose support is different from that of its super-sequences. Sequence mining is analogous to block correlation mining as a block can be mapped to an item and an access sequence to a item sequence in the database. A frequent subsequence indicates that the set of blocks are found to be accessed together a given number of times in the access stream and hence signals correlation. C-Miner adds a restriction: a set of blocks are considered correlated only if they appear in the original sequence within a pre-specified access distance, called gap in the algorithm. C-Miner has the following phases:
Create a database  Since request access stream is a single sequence instead of a database, C-Miner first divides the stream into a set of smaller chunks so that they represent a database of streams. The subsequences that fall within the boundaries of the chunks are not considered sequential. For example, for the access stream \{a b d c a b f d e\}, we can create the sequence database as follows:

\{a b d\}, \{c a b\}, \{f d e\}.

Generate candidate set  A candidate set of frequent subsequences of length one is generated using a depth-first search procedure. For the above database, the candidate set is \{a, b, d\} when \textit{min-support} is 2.

Generate the frequent subsequences  The main observation here is that if a sequence (i.e., \textit{abc}) is frequent, then all its subsequences (i.e., \{a, b, c, ab, ac, bc\}) are frequent. C-Miner thus recursively generates the longer frequent subsequences by concatenating every frequent item to a shorter frequent subsequence that is already produced in the previous steps.

Association rules  The frequent subsequences are finally broken down into association rules that define correlations. For example, the following correlations can be derived from the frequent subsequence \textit{abc}:

\{a \rightarrow b, a \rightarrow c, b \rightarrow c, ab \rightarrow c\}

Confidence of rules  For a rule \textit{a \rightarrow b}, if \textit{a} occurs \textit{x} times, but \textit{ab} occurs \textit{y} times within the specified gap, then the confidence of the rule \textit{a \rightarrow b} is defined as \((y/x \times 100)\). That is, if \textit{a} occurs 5 times, but \textit{ab} occurs 4 times in the access stream, then the probability that \textit{b} will be accessed after \textit{a} in near future is 80%. The higher the confidence is, the more accurate the prediction of correlation is.

Once the correlation rules are generated, they can be used to generate placement configuration needed for correlation directed disk layout or can be used for correlation-directed prefetching. We used the \textit{C-Miner} algorithm.
to implement correlation-directed prefetching in our simulator. We used the correlation rules having more than 50% confidence and implemented single correlations only, i.e., correlations of the form $u \rightarrow v$. The effectiveness of the algorithm is compared with BORG heuristic by running simulation experiments.

### 4.1.4 The simulator

We developed a discrete event simulator that maintains a priority queue of events based on event time. We simulated a high performance storage device, a low performance storage device, a prefetch cache and a OS cache. The OS cache, 256 MB in size, stores recently accessed blocks and implements an LRU eviction policy. We also implemented a simple sequential read ahead policy that prefetches 8 consecutive pages whenever a request is redirected to the underlying low performance device; these pages are also stored in the OS cache. The correlated pages that are prefetched ahead of time are stored in the prefetch cache which is also 256 MB in size. We kept the prefetch cache separate so that correlation-directed prefetched pages do not pollute the operating system cache representing the temporal and spatial locality of references. We implemented a FIFO eviction policy for the prefetch cache. The observation behind this is that the prefetched blocks are supposed to be read in the order of prefetching. So, the block that was prefetched a long time ago, probably has already been requested and is safer to evict than a block that is recently prefetched.

The simulator operates through the following stages:

**processRequest**  In this stage, we pop a demand request from the queue, find it’s correlated blocks and send it to the next stage sendRequest. A small overhead (2 nanosecond in our experiments) is added in this stage that represents processing time of the request. We also schedule a prefetch request that prefetches the correlated blocks of the current demand request. Once the current request is processed in this stage, we push the next demand request to the event queue. The demand requests use the same time stamp as
in the original trace. However the prefetch requests are scheduled at current simulation time. This is because demand requests should be processed in the order they arrive, but prefetch requests should have a lower priority.

The prefetch requests are handled differently when compared to the demand requests. For example, prefetch requests do not trigger any more correlated prefetching, i.e., there is no recursive prefetching. Also, there is no read ahead for the prefetch requests. Before scheduling a prefetch request we first check if the requested block is present in the high performance device in which case the request is cancelled. The reason is straightforward — we can save a low performance device access by not prefetching and serving the block from the HPD which already has much lower access time.

![Diagram](image)

**Figure 4.2:** Stages of the simulator. ProcessRequest processes a request from the event queue and issues prefetch requests, SendRequest sends the request, divided into multiple one page requests, to appropriate devices, ServeRequest represents the actual time required by the physical devices to serve the request, and CompleteRequest updates the statistics and the caches.

**sendRequest** This stage redirects the current request to the corresponding device. The request is first divided into multiple one page units and each one of them is served separately depending on where the page is present. We have an *OS cache*, a prefetch cache, a *high performance device* (HPD) and a *low performance device* (LPD) — the demand pages are searched in these devices in this order.

The blocks that are read from the low performance device are scheduled to be stored in the *OS cache*. We do not waste cache space for storing recently accessed blocks from the HPD as they can be read again with low
4.2 Simulation results

overhead. The same reasoning is applied for read ahead strategy. We enable read ahead only for the LPD.

**serveRequest** This stage simulates the underlying device that actually serves the request. The processing time at this stage represents the request serving time of the underlying device. It depends on the latency and bandwidth of the device and is computed as follows:

\[
\text{request serving time} = \text{device latency} + \frac{\text{request size}}{\text{device bandwidth}} \tag{4.2}
\]

Once the processing time is computed, the request is scheduled for completion.

**completeRequest** When a request is completed we add the accessed pages to the *OS cache* or to the *prefetch cache* if the request was a prefetch request. This stage also updates the statistics, such as response times, total blocks served from each of the storage devices, namely the *OS cache*, the *prefetch cache*, the HPD and the LPD.

4.2 Simulation results

We ran the simulations using real traces collected from MSR enterprise servers (Table 3.1). We also used a web server trace collected by Bhadkamkar et al. [10]. These traces represent week-long disk activity. In our experiments, we simulated the read requests only and discard the write requests. We divided the traces in two halves, and used the first half to train the correlation analyzer that generates the partition and prefetch lists. The second half of the traces are used to test the hybrid storage system.

For all experiments we simulated real enterprise-level storage devices unless otherwise mentioned. For data partitioning, we used a combination of a magnetic hard disk drive (HDD) and a solid-state drive (SSD). Solid-state drives do not have any mechanical parts (detailed description of SSD in Section 5.1) and hence are much faster than the magnetic hard disk
drives which introduce seek and rotational latency due to the movement of read/write heads and platters (comparison of SSD and HDD based on performance and cost in Section 5.2). We used only the latency and throughput parameters of these storage devices and noted these parameters from their manufacturer specified disk specifications (Table 4.1). These disks cannot cope with the requests in the traces, and hence we increased the think time between requests by 10 ms. Unless otherwise stated, all of our experiments are run with the increased think time.

<table>
<thead>
<tr>
<th>Device</th>
<th>Model</th>
<th>Read throughput (MB/sec)</th>
<th>Read latency (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDD</td>
<td>WDC WD2500AAKS-22B3A0</td>
<td>163</td>
<td>5.4</td>
</tr>
<tr>
<td>SSD</td>
<td>INTEL SSDSA2MH080G1GN</td>
<td>250</td>
<td>0.075</td>
</tr>
</tbody>
</table>

Table 4.1: Disk specification

4.2.1 Data partitioning

This section presents our simulation results for data partitioning where the file system data blocks are split between two devices having different capacities and performance characteristics. We study through experiments the impact of data partitioning for different types of storage system settings. In particular we answer the following questions: (i) How the difference between performance characteristics (i.e., latency and bandwidth) of the two devices affect performance? (ii) How does partitioning perform as compared to caching? (iii) How much capacity of the HPD is needed to achieve a certain performance target?

We keep the most connected blocks, as described in Section 4.1.1, in the low latency device. For the simulations, we do not move the physical blocks but keep a mapping of where the blocks are.
4.2. Simulation results

Figure 4.3: Impact of latency difference between the HPD and the LPD. x-axis is represented in logarithmic scale and represents the ratio between HPD latency and LPD latency. y-axis represents the improvement in response time, normalized to the LPD-only system.

4.2.1.1 Impact of varying latency and bandwidth on partitioning

We observed that the latency difference between two devices significantly affects the performance of data partitioning (Figure 4.3). Up to a certain threshold (about 16x), the higher the latency difference is the better the performance we get. For this experiment, the bandwidth is kept constant at 250MB/sec.

We used 10% of the total trace size as the capacity of the high performance device. The percentage of blocks read from the high performance device (Figure 4.4) depends on trace characteristics and is different for each of the traces. This explains the variation in performance improvement for different traces. For example, data partitioning, in the case of trace Proj3, does not have any performance improvement over the LPD-only system. This is because the Proj3 trace has very low data overlap, specially with the top 20% accesses (Figure 3.2). This is again verified in this experiment as we can see that only 0.29% of the blocks are read from the high performance device (Figure 4.4). This is also the case for the Mds trace. On the other hand, most improvement is achieved in the case of the Usr trace.
where the second half of the trace shows more than 90% overlap with the top 20% accesses of the first half of the trace.

Figure 4.4: Percentage of blocks read from each device in data partitioning. The high performance device capacity is maintained at only 10% of the total trace size.

The performance impact of the difference between the LPD and the HPD bandwidth is not very significant for most of the traces (Figure 4.5). Only the Usr trace is affected significantly which is because ∼99% of the blocks are served from the high performance device (Figure 4.4). For this experiment, the latency difference between the two devices is kept constant at 100 ms. The high performance device capacity is again maintained at 10% of the total trace size.

4.2.1.2 Data partitioning vs. caching

Data partitioning utilizes application long-range block access patterns whereas caching utilizes temporal locality of reference only. We evaluated, through simulations, how partitioning performs as compared to caching. For data partitioning, we moved the most frequently accessed blocks to a solid-state drive and kept the rest in a magnetic hard disk drive. For caching, we stored the temporal locality of references in an SSD of same capacity and characteristics as used in partitioning. We simulated data partitioning and caching for both 1GB and 2GB SSD sizes (Figure 4.6). As mentioned before, data
4.2. Simulation results

![Graph showing simulation results]

**Figure 4.5:** Impact of bandwidth difference between HPD and LPD. x-axis is represented in logarithmic scale and represents the ratio between HPD bandwidth and LPD bandwidth. y-axis represents the improvement in response time from the variation in bandwidth, normalized to the response time where bandwidth is same for the HPD and the LPD.

partitioning uses the first half of the traces for training and the second half for simulation.

Other than the case for Proj3 trace we observed significant performance benefit of data partitioning over caching for different traces and with different SSD sizes. Data partitioning improved performance over caching by up to 58% and 64% for 1GB and 2GB SSDs respectively. Although Proj3 trace shows significant repetition of blocks (Figure 3.1), the overlap of the second half of the trace with the frequently accessed blocks of the first half of the trace is very low (Figure 3.2). This implies that the repetition of blocks occur within short access distances which benefit the SSD cache over the data partitioning technique. In the case of Proj3 trace, more than 60% of the blocks are read from the SSD cache for even 1GB SSD size.

When using the flash memory based solid-state drives, data partitioning is better choice than caching. The fact that SSD blocks need to be erased before re-written and that each block permits a limited number of erase cycles makes SSD writes complicated. The device controller optimizes the life time of each block by distributing writes uniformly over the entire device, a technique known as wear leveling. Sometimes write-amplification occurs
4.2. Simulation results

Figure 4.6: Data partitioning vs. caching. Simulations are run with enterprise-level hard disk drives and solid-state drives. No other type of cache is simulated for these experiments. In data partitioning, the frequently accessed blocks are moved to the SSD. Here, Cache represents a simple LRU cache that stores recently accessed blocks in an SSD. We compared the performance of data partitioning vs. caching for both 1GB and 2GB SSD sizes.

as a result of wear leveling which leads to poor performance of SSD writes as compared to disk writes. When implemented as a cache to store recently accessed blocks, SSD blocks are continuously written which might lead to earlier wear out of the blocks. Special mechanisms can be implemented in software to coalesce multiple writes in memory before sending to the device which complicates the entire process [27]. On the other hand, data partitioning updates the blocks that reside in the SSD once a pre-specified partitioning interval which can be as large as one week. This minimizes the adverse effect of SSD writes on the life of the device.

Data partitioning also minimizes the data movement overhead. Given the limited capacity of the SSD, we may have to move some of the previously resided blocks back to the disk before sending more blocks to SSD. When SSD is used as a cache, this data movement is done continuously with the change in short-term working set. However, data partitioning can minimize this overhead by performing data movement offline, only once in a few days.
and when the disk is otherwise idle.

4.2.1.3 Capacity of the high performance device

We need to understand how much of the high performance storage is needed to have significant performance improvement by data partitioning. This measure depends on the file system usage as well as the actual file system size. Since we do not have the knowledge about the complete file system size for the MSR traces, we report high performance device capacity as a percentage of the total trace size. For this experiment, we have partitioned data blocks between a hard disk drive and a solid-state drive, both having enterprise-level characteristics (Table 4.1).

Most of the traces show significant data repetition and overlap (Figure 3.1 and Figure 3.2 respectively) which lead to more than 20% performance improvement by using an SSD of only 5% of the total trace size (Figure 4.7(a)). Most improvement (up to 93%) is observed for the Usr trace, where ~99% of the second half overlaps with the first half of the trace.

The percentage of total blocks that are read from the SSD increases as we increase SSD size but only up to a certain threshold, about 20% of the trace size (Figure 4.7(b)). This is because, we have simulated an OS cache with read ahead enabled for this experiment. The OS cache has higher preference of serving requested data blocks than the SSD and for all traces, a certain percentage of data blocks are always read from the OS cache. For the Proj3 trace, more than 50% of the requests are served from the OS cache (note the low response time in Figure 4.7(a)).

Overall, we can see that data partitioning with an SSD of capacity as low as 5% of the total trace size can improve the storage system performance by more than 20%. For most of the traces, more than 50% (up to 93%) performance improvement is possible if the SSD capacity is increased to 20% of the total trace size.

We have the file system information available only for the Web trace. This trace is 13.7GB and accesses only 8% of the file system (file system
4.2. Simulation results

![Graphs showing simulation results](image)

**Figure 4.7:**
(a) Improvement in response times with increasing SSD sizes.
(b) Percentage of blocks served from SSD with increasing SSD sizes.

size is 169.54 [10]). For this trace, data partitioning with an SSD capacity of only about 1.6% of the complete file system size results in more than 50% performance improvement.

### 4.2.2 Correlation-directed prefetching

We studied different aspects of correlation-directed prefetching (CDP). We are mainly interested in answering the following questions: (i) How different...
4.2 Simulation results

ent block correlation analysis techniques affect the performance of CDP?
(ii) What is the impact of application think time in the analysis of block
correlation? (iii) Instead of prefetching the immediately correlated block,
will it be beneficial to prefetch the next block in the chain of correlations?
(iv) Does data partitioning have any positive impact on the performance of
CDP?

4.2.2.1 Data partitioning vs. CDP

We studied the impact of data partitioning and CDP separately on a disk-
based system (Figure 4.8(a)). Here, disk represents the base-line case, a
disk-only system with 256MB of OS cache and 8-page sequential read ahead.
CDP-BORG represents the same system with correlated prefetching and
uses a separate 256MB prefetch cache for correlated pages. Data Parti-
tioning represents a combination of an HDD and an SSD that implements
the OS caching but no correlated prefetching. We observed that prefetch-
ing correlated blocks can improve the read I/O performance over sequential
prefetching. But the most improvement is achieved by data partitioning;
except for the Mds and Proj3 trace, data partitioning has 32–92% better
average response times than CDP.

4.2.2.2 Impact of different correlation analysis heuristics

Interestingly, the impact of the difference in block correlation analysis is
negligible in correlation-directed prefetching (Figure 4.8(a)). For these ex-
periments, we implemented both data partitioning and correlated prefetch-
ing. For CDP, we analyzed block correlations using two heuristics BORG
and C-Miner (Section 4.1) represented by CDP-BORG and CDP-Cminer
respectively. We observed that the effect of these heuristics in prefetching is
almost same — CDP-Cminer is slightly better or worse for different traces.
Although BORG takes process information into account while analyzing
block correlation, we could not use this feature as process information was
not available for any of the MSR traces. Another difference between BORG
and C-Miner is that BORG considers only neighboring correlation. For
4.2. Simulation results

![Simulation results chart](image)

**Figure 4.8:** (a) Hybrid system response times for different optimization techniques. Disk is the base-line case showing the disk-only response times; CDP-BORG represents correlated prefetching only in a disk-based system (no data partitioning); Data partitioning represents the response times of a storage system combining an HDD and an SSD (no correlated prefetching). Rest of the experiments implement both data partitioning and prefetching but uses different heuristics for correlation analysis. We computed the standard deviation of the response times and found that, in most experiments, the standard deviation lies between 2 to 3 ms. (b) Percentage of the blocks served from the devices for the case of CDP-thinktime optimization technique. We have observed similar distribution of blocks in other CDP techniques.
4.2. Simulation results

example, for the access sequence abcd, BORG would output the following correlation rules: \( a \rightarrow b, b \rightarrow c \) and \( c \rightarrow d \). On the other hand C-Miner would generate the correlation rules as \( a \rightarrow b, a \rightarrow c, a \rightarrow d, b \rightarrow c, b \rightarrow d \) and \( c \rightarrow d \). Although C-Miner generates more correlation rules, but a lot of them are filtered out by the confidence threshold (at least 50% in our experiments). We also restricted the issuing of prefetch requests so that at most 2 prefetch requests (having maximum support) are issued for each demand request. This restriction is needed because these prefetch requests result in additional disk accesses. The more prefetch requests we issue, the more disk access would be necessary which would increase prefetch overhead as these accesses interfere with the demand requests. With these limitations, both BORG and C-Miner prefetch similar data and show similar performance.

We modified the BORG heuristic to include think time in the correlation analysis (Section 4.1.2), represented by the CDP-thinktime notation in Figure 4.8(a). We observed that the inclusion of think time in the analysis does not have any impact on correlation-directed prefetching.

4.2.2.3 Slack-based integration of data partitioning and CDP

Data partitioning changes the way storage systems behave and introduces new design options for correlation-directed prefetching. So far for the CDP-BORG, CDP-thinktime or CDP-Cminer techniques (Figure 4.8(a)) we only skipped prefetching the correlated blocks that are available in the faster device. Serving a request from the faster device requires less access time as compared to the slower device. This gives us a slack between two successive demand requests, and we can use this slack for issuing prefetch requests without causing delay to any demand request. In this slack-based integration of data partitioning and CDP, we issue prefetch request only if a request is completely served from the faster device or from the prefetch cache. In other words, we restrain from prefetching correlated blocks if reading the related block requires accessing the slower device. We found that CDP-Slack performs better when think time is low. This is reflected in Figure 4.9, where we can see that for most of the traces 20-90% improvement is possible.
with the \textit{CDP-Slack} technique over \textit{CDP-thinktime}. We compared \textit{CDP-Slack} with \textit{CDP-thinktime} only, as other CDP techniques perform similar (Figure 4.8(a)). With moderate think times, both \textit{CDP-Slack} and \textit{CDP-thinktime} techniques have similar performances. But when the think time between accesses are fairly high, \textit{CDP-Slack} suffers from not prefetching more correlated blocks. We ran this experiment with constant think times and since the traces have different characteristics regarding repetition and overlap, we used different low, medium and high think times for each of the traces.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig4.9.png}
\caption{Percentage improvements in response time by \textit{CDP-Slack} with respect to \textit{CDP-thinktime}. Due to the variation in trace characteristics, we used different low, medium and high think times for different traces. The low, medium and high think times are (10, 15, 20)ms for the Web and Proj3 traces, (6, 10, 15)ms for the Src and Mds traces and (5, 10, 15)ms for the rest. \textit{CDP-Slack} does not have any performance impact on the Usr trace because less than 1% requests are served from the slower device for this trace. Thus both \textit{CDP-Slack} and \textit{CDP-thinktime} prefetches same amount of data.}
\end{figure}

\subsection*{4.2.2.4 Effect of prefetch cache size}

There is almost no effect of prefetch cache size on performance (Figure 4.10). There are two reasons behind this: (i) we prefetch conservatively, at most 2 prefetch requests can be issued per demand request. This restriction is needed since prefetch requests interfere with demand requests when accessing the disk. (ii) Prefetch cache holds specific pages that are going to be
4.2. Simulation results

Figure 4.10: Response time improvements of the hybrid storage system with increasing prefetch cache size.

accessed in near future. Once these pages are accessed, they can be safely evicted. This experiment also justifies our FIFO eviction policy for the prefetch cache.

4.2.2.5 Prefetching from the chain of correlation

Most CDP techniques prefetch only the immediately correlated blocks. However, if think time is small, then prefetching may not complete in time. To allow us more time to complete the requests, we consider a different approach here — we skip the immediately correlated block and prefetch the secondarily correlated blocks. For example, consider the following chain of correlation:

\[ A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \] (4.3)

Here, block \( A \) is correlated to block \( B \), \( B \) is correlated to \( C \) and so on. In usual prefetching techniques, when \( A \) is accessed, the storage system starts prefetching \( B \). If the think time between \( A \rightarrow B \) is small, then the prefetch request might not complete when the request for \( B \) comes in and we might have to serve \( B \) from disk. However, once requested, we cannot cancel the prefetch request of \( B \), which only adds overhead by interfering with other requests. One solution could be to maintain the list of prefetch requests on flight and hold the demand requests until prefetching is completed. This technique has both time and space overheads and complicates the process.
4.2. Simulation results

Figure 4.11: (a) Hybrid system response times for different chain of correlation-directed prefetching optimization techniques. CCDP-0 represents immediate prefetching. CCDP-1, CCDP-2 and CCDP-3 represent skipping 1, 2 and 3 correlations from the chain and prefetching the next one respectively. (b) Prefetched blocks as a percentage of total blocks accessed in the traces.

In this work, we considered an alternative by looking ahead in the chain of correlation. For example, when A is accessed, instead of prefetching B, we can start prefetching C; this way C would be in memory when needed even if B → C think time is small. We call this technique CCDP-1 as it skips one correlation and prefetches the next. This approach essentially gives us more time to complete the prefetch requests, whereas prefetches the same sequence of blocks. Similarly, to allow us even more time to complete the prefetch requests, we can look even further in the chain of correlation,
e.g., we can prefetch block $D$ or even block $E$ when $A$ is accessed (CCDP-2 and CCDP-3 respectively). With reasonably large think times all these techniques perform similarly. However, when think time is low, prefetching from the chain of correlation has better response times than immediate prefetching (CCDP-0) (Figure 4.11(a)). However, due to some possible breaks in the chains, the CCDP techniques prefetch slightly less amount of blocks than CDP (Figure 4.11(b)) — this reduces prefetch overhead and may also have some positive effect on performance.

### 4.2.2.6 Moving correlated blocks to the HPD

We also investigated another technique for integrating data partitioning and CDP where we move some of the correlated blocks of the frequently accessed blocks to the high performance device. The basic idea here is to move correlated blocks with small think times to the HPD and prefetch those with relatively large think times. The rationale behind this approach is that correlated blocks having very small think times cannot be prefetched on time and only adds to the overhead of prefetching. On the other hand, correlated blocks with reasonably large think times can safely be prefetched. This technique degraded the performance of the hybrid storage system by a large amount (we do not report the experimental results here). This is because moving some of the correlated blocks to the HPD replaces some of the frequently accessed blocks as the HPD capacity is limited. This degrades the performance of data partitioning increasing overall read response time.
Chapter 5

A Case Study: HDD+SSD Storage for a Linux Based System

This chapter describes a case study: a hybrid storage system, consisting of a Western Digital hard disk drive (HDD) and an Intel solid state drive (SSD), implemented in the Linux kernel.

HDDs are the most common medium of secondary storage of data since the Sixties. They have improved a lot in cost, size and capacity - a 3.5” HDD can now hold up to terabytes of data. Unfortunately, since the HDDs involve movement of mechanical arms, their performance has not scaled with that of today’s increasing CPU performance. The recently commercialized solid state drives use flash memory technology that do not require movement of mechanical arms for retrieving or storing data and have thus shown significantly better performance than HDDs (Section 5.1 and Section 5.2). However, these new technologies are still very expensive considering the cost per Gigabyte of storage, and cannot yet replace the typical HDDs. This motivates us in combining an SSD and an HDD and then partitioning file system data blocks between the two media based on application block access patterns.

We explored two different strategies for implementing data partitioning in the Linux kernel (Section 5.3). We first implemented an I/O redirection approach where we developed a Linux kernel module that performs data transfer between the HDD and the SSD and also redirects I/O requests from the disk request queue to appropriate devices as necessary (Section 5.3.1).
5.1. Solid-state drives

However, the problem with this approach is that the redirection of requests from the HDD request queue to the SSD request queue causes a double queueing overhead and the observed improvement in response times is not as good as expected. Our second approach is to implement partitioning by remapping inode pointers (Section 5.3.2). This approach gave us the desired performance improvement which is evident in our system evaluation.

We also implemented correlation-directed prefetching (CDP) in the Linux kernel (Section 5.5). Although our simulation results showed performance improvement by CDP, the evaluation in the Linux kernel, has dictated otherwise. Correlation-directed prefetching along with data partitioning did not improve performance more than data partitioning alone did (Section 5.6).

5.1 Solid-state drives

Solid-State Drives are becoming more and more prevalent these days. Made from non-volatile flash memory, usually NAND-based flash memory, SSDs provide persistent storage while requiring no moving parts. HDDs, on the other hand use rotating disks and movable read/write heads for accessing data which result in a higher access time and latency. Having no moving parts has not only resulted in better read/write performance for SSDs, but also has enabled them to withstand extreme physical shock, vibration and temperature ranges. SSDs also generate less noise and are much more power efficient compared to HDDs.

Due to their increased reliability, the flash-based SSDs have gained much popularity in mission-critical applications, such as military and aerospace industries, since their first introduction by M-Systems in 1995. However, the much higher cost per gigabyte of storage of SSDs as compared to the hard disk drives has still prevented them from replacing the disk drives for personal computer clients. The other limiting factor for adapting SSDs as the secondary storage is their limited capacity. While current HDD technology allows storage of terabytes of data for personal computers, SSDs are still capable of storing in the range of hundreds of gigabytes.

The SSD performance and capacity comes as a trade-off: the multiple-
level cell (MLC) SSDs provide better packing density, resulting in lower costs than their counterpart, single-level cell (SLC) SSDs, which provide better access times. Although the read performance of both of these types of SSDs is significantly better than that of HDDs, it is the write performance, specially small writes, that makes them sometimes undesirable. The problem with writes arises from the fact that SSD blocks cannot be updated — they need to be erased first and then re-written like EEPROMs (Electrically Erasable Programmable Read-Only Memory). This erasing puts a limit on the lifetime of each SSD block — typically one million erase cycles are possible for most SSDs. To increase overall SSD lifetime, manufacturers implement a technique called wear leveling in the flash controller. Wear leveling reduces over-utilization by distributing erasures and re-writes evenly across the medium, and hence prevents premature failure of disk blocks.

5.2 HDD vs SSD: performance and price

http://www.tomshardware.com provides a large number of charts showing performance evaluations for different SSDs and HDDs. We summarize a few of them here to compare access times and read/write throughputs of some SSDs and HDDs.

Figure 5.1(a) compares random access times of an Intel X25 MLC SSD and a Seagate Momentus 7200rpm HDD. Since SSDs have no mechanical arm/platter movement, they have neither any seek time nor any rotational latency – any block in an SSD has the same access time. On the other hand, seek time and rotational latency are the two major drawbacks of HDDs. From figure 5.1(a) we can see that an average SSD offers 156x better access times than an average HDD.

Even for sequential accesses SSDs are much faster than HDDs. Figure 5.1(b) compares the average sequential read and write transfer performance of the same SSD and HDD as in Figure 5.1(a). From the figure we can see that even for sequential reads, SSDs are more than three times faster for example
5.3. Implementation of data partitioning

Both of our partitioning and prefetching mechanisms require block level information. In order to determine frequently accessed and correlated blocks, we first need to collect information about the I/O requests submitted to

than HDDs. Although the write performance of SSDs is still better than that of HDDs, the difference is not significant. This is because SSD writes are internally re-organized and sometimes amplified by the flash controller so as to provide a uniform writing to all blocks available. Otherwise some blocks that are heavily written can wear out much earlier than the rest.

HDDs, on the other hand, are very attractive because of the much lower cost per gigabyte of storage. The historical price per Megabyte trend of SSD and HDD (Figure 5.2) shows the large gap that existed between SSD and HDD prices for years [1]. Recently, this gap has been reduced a lot, but still HDDs are at least one order of magnitude less expensive for the same amount of data storage on an SSD. Table 5.1 shows the most recent prices of some desktop and enterprise class SSDs and HDDs taken from http://www.amazon.com. From the table, we can see that even a 2TB HDD is cheaper than an 80GB SSD. Thus a storage system that cleverly partitions data blocks between a small SSD and a large HDD can benefit by serving most of the requests from the SSD while the HDD would serve as the storage for most file system data blocks.

5.3 Implementation of data partitioning
5.3. Implementation of data partitioning

![Historical price ($) per Megabyte trend of SSD and HDD \[1\].](image)

<table>
<thead>
<tr>
<th>Device</th>
<th>Capacity</th>
<th>Price($)</th>
<th>Price($/MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X25M SSD</td>
<td>80GB</td>
<td>220</td>
<td>0.0027</td>
</tr>
<tr>
<td>Intel X25M SSD</td>
<td>250GB</td>
<td>550</td>
<td>0.0021</td>
</tr>
<tr>
<td>Seagate Barracuda HDD</td>
<td>500GB</td>
<td>50</td>
<td>0.000097</td>
</tr>
<tr>
<td>Seagate Barracuda HDD</td>
<td>2TB</td>
<td>150</td>
<td>0.000071</td>
</tr>
</tbody>
</table>

Table 5.1: Price comparison of SSDs and HDDs based on 2011 prices (Source: \[http://www.amazon.com\](http://www.amazon.com))

The trace collection period should be reasonably long so that the requests represent actual access patterns. For monitoring I/O requests any block level tracing tool can be used, blktrace \[9\] being one example. Blktrace is a block layer I/O tracing mechanism that collects detailed information about the request queue operations on a per I/O basis. Blktrace stores the data collected in files that can later be used to extract and format only fields of interest. For our access pattern analysis, we only need the following information:

- Request submission time,
- Process ID,
- Logical block address,
- Request length,
- Event type, and
- Request type (read or write).
5.3. Implementation of data partitioning

We are interested only in the read requests submitted to the queue (Q event type in the case of blktrace). Once a trace is collected, it can be used to analyze the access patterns. In our experiments, however, we do not collect traces. Instead, we have used real traces representing Microsoft Research Server workloads that are collected from the Storage Networking Industry Association’s Input/Output Traces, Tools, and Analysis (SNIA IOTTA) Repository [6] (Section 3.1).

As already mentioned, we implemented partitioning in two different ways: (i) redirecting I/O in the kernel and (ii) remapping inode pointers from user space.

5.3.1 I/O redirection approach

We developed a linux kernel module that sits in the block layer and intercepts requests in the request queue. The advantage of using a kernel module is that installation and un-installation of a module is very simple; it does not require re-compilation or re-installation of the whole kernel. We used the commands \texttt{insmod} and \texttt{rmmod} for installing and un-installing the kernel module respectively.

Data partitioning has three major steps: generating the partition list consisting of which blocks to move to SSD, moving the data blocks to and from SSD, and redirecting I/O requests correspondingly. Generating the partition list has been described already in Section 4.1. Rest of the steps, implemented in the kernel module, are described in detail below.

Data Movement In each partition interval, a partition list (Section 4.1) is generated in user space and is communicated with the kernel module through a character device. The user space program writes to the character device the source and destination addresses of a block, and the transfer direction – HDD to SSD or the other way. Movement of a block from one device to another is done in two steps: first, the block is read from the source device, and then in the second step it is written to the destination device. Whenever the character device is written, a new bio (Linux data structure
5.3. Implementation of data partitioning

representing a block I/O request) is created that reads the block to be moved from its original location. For example, if a block is to be moved from HDD to SSD then the \textit{bi\_sector} field of the \textit{bio} denotes the sector number of this block and \textit{bi\_bdev} denotes the HDD device pointer. When this I/O request completes, our customized end I/O handler is called that creates another \textit{bio} and initiates another disk request which writes the block that is just read to its destination. We use the Linux implementation of the Radix Tree data structure to keep track of which block is where. Once the data transfer is complete, the radix tree is updated to reflect the new location of the block.

![Figure 5.3: Improvement in response times for the Web trace using the I/O redirection approach.](image)

**Redirection** Our Linux kernel module replaces the request queue handler function, \textit{make\_request\_fn()}, with a customized function \textit{hybrid\_layer()}. This new request handler first splits the block I/O requests (denoted by the \textit{bio} data structures) into smaller requests consisting of one page each. Then it searches the radix tree to find out the mapped location of each of the requested pages. If a page is found to be in the SSD, the \textit{bi\_sector} field of this \textit{bio} is changed so that it points to the SSD device block, and the \textit{bi\_bdev} field is changed to SSD device pointer. The request is then submitted again using \textit{generic\_make\_request()}. The problem with this approach is that this re-submission of requests causes the \textit{bio} to wait in both the HDD device request
5.3. Implementation of data partitioning

queue and the SSD device request queue, and this adds to the overhead of redirections.

**Evaluation of the data partitioning technique using I/O redirection**  We evaluated data partitioning using I/O redirection approach for the Web trace only. We used 25% of the total trace size as available SSD capacity for partitioning. In order to clearly understand the overhead associated with the I/O redirection approach, we computed SSD-only response times twice: (i) **SSD-only** (Figure [5.3]) denotes the average response time when all requests are initially submitted to the SSD queue and (ii) **SSD-only (redirected)** denotes the same when all requests are initially submitted to the disk queue and then redirected to the SSD queue through our kernel module. We can clearly see that a large overhead is associated with the I/O redirection approach – the **SSD-only (redirected)** system is $\sim 7.8x$ slower than the **SSD-only** system. This is why the data partitioning approach using I/O redirection improves the **Disk-only** system by only 17% whereas our simulation results (Figure [4.7(a)]) show that more than 50% improvement is possible.

### 5.3.2 Inode mapping approach

The I/O redirection approach did not work as expected. Each block is a part of an inode in the file system, therefore our new approach is to move the data block of a corresponding inode and change the inode pointer accordingly. We have to create a single file system comprising of both the SSD and the HDD. We used the Linux tool `mdadm` for this purpose. `mdadm` is usually used for creating software RAID devices, but it also supports a LINEAR mode, where two given devices are concatenated and presented to the underlying OS as a single device. We used the HDD as the first device and the SSD as the second in the concatenated device.

**Data Movement**  Our partitioner scans all the direct, indirect, double indirect and triple indirect pointers of each inode of a file system (Figure [5.5(a)]). If any data block pointed to by those pointers are supposed to
5.3. Implementation of data partitioning

Figure 5.4: (a) Linux file system inode structure. (b) Data partitioning: After moving the frequently accessed blocks (dashed blocks in HDD) to the SSD, inode pointers are changed to point to the new location of the data blocks.
be in the SSD, then these data blocks are moved from the disk to the SSD and corresponding pointer of the inode is changed (Figure 5.5(b)). This process is continued until the SSD is full.

**Redirection**  No redirection is necessary in this approach. Since the inode pointers are changed along with data movement, we do not need to maintain any other mapping of data blocks which saves us from additional memory overhead. Also, no time is spent in redirection. The limitation of this approach (particularly in our implementation) is that the inode pointer blocks reside in the HDD, and are not moved to the SSD. To access a data block, the inode is first accessed from the HDD, and then the data is read from the SSD. Accessing the inode from the HDD may introduce some overhead for each data block that is read from the SSD. One solution of this problem could be creating the file system in the SSD, so that all inode blocks and pointers reside in SSD. We have not evaluated this approach in this work.

**Evaluation of the data partitioning technique using inode re-mapping**

The inode re-mapping approach improves the performance of data partitioning as expected. For these experiments we kept the SSD device size to 10% of the total trace size. These results conform to our simulation results (Figure 4.7(a)) for the case of 10% SSD size. For the Web trace, the I/O re-direction approach improved read performance of the storage system by 17% whereas the inode re-mapping approach improves the same by more than 40%. Except for the media server trace, more than 40% (up to 79%) improvement is achieved for most of the traces.

### 5.4 Limitations of partitioning

The limitations we describe here are specific to the SSD+HDD implementation of the hybrid storage system.

**Fragmentation**  Since SSD size is limited, during each partitioning interval we may have to move back some of the previously stored SSD blocks to
5.4. Limitations of partitioning

the HDD if the on disk long term working set is significantly changed. This operation may cause fragmentation in the HDD and in the SSD. Fragmentation in SSD does not matter, as all the blocks in SSD have the same access time. However, fragmentation in HDD may significantly increase response time. We have not evaluated performance after fragmentation, but we hope that data partitioning will be able to overcome the impact in performance, if any. There are many off-the-shelf de-fragmenters available for Linux that can be run periodically. The best option would be to incorporate a auto de-fragmenter in the partitioner itself.

Figure 5.5: (a) Improvement in response times by data partitioning using the inode re-mapping approach. (b) Percentage of data blocks read from each device.
SSD small write performance  Due to the property of the NAND sub-
strate of SSD devices, the SSD blocks have a limited lifetime. In order to
improve block efficiency and reduce premature failure of blocks, wear level-
ing is performed on SSD devices that distributes the writes evenly to SSD
blocks. Write-amplification is a side effect of wear leveling which negatively
impacts SSD write performance, specially for small random writes. To ac-
count for this writes can be coalesced in memory [27] or even a disk based
cache [36] can be used.

Data movement overhead  During the partitioning interval, a large num-
ber of blocks will be moved from one media to another. It might cause tem-
porary increase in response time for some requests involving those blocks in
flight. Partitioning can be scheduled offline when the system is otherwise
idle or in a very low usage state.

5.5 Implementation: correlation-directed prefetching

CDP is implemented in the kernel module. The prefetch list, represent-
ing block correlations, is generated once in every partitioning interval as
described in Section 4.1.1 and is communicated with the kernel through a
character device. We use the Linux radix tree implementation to store cor-
relation data, we call it the CORRELATION_RADIX_TREE. If block A is
correlated to block B, then we call block A the source of the correlation and
block B the destination of the correlation. In our implementation, we break
LBA ranges representing the source of the correlation so that the source is
always a single page. The source of the correlation is the key in the radix
tree and the destination LBA range is the value. Since multiple LBA ranges
can be correlated with a single source LBA range, the value field in the radix
tree is implemented as a linked list ordered by frequency of the correlation
divided by the think time between the two accesses (i.e., we implemented
CDP-thinktime in the Linux kernel, more in Section 4.1.2).
As in the I/O redirection approach (Section 5.3.1), the request queue handler is modified to implement prefetching. The modified handler intercepts each request (submitted through a bio data structure), splits it into multiple one-page bios. Splitting is necessary because each bio represents a disk I/O request consisting of multiple pages of which not all might be present in the prefetch memory. Our modified request queue handler maintains a radix tree mapping of which pages are present in the prefetch memory, we call it the PREFETCH_RADIX_TREE. Thus if a page is found to be present in prefetch memory, it is served immediately and the corresponding bio is forcefully ended. On the other hand if the page is not present in memory then the corresponding bio is sent to the underlying device driver which then handles the request.

When all the pages of a bio are processed, either served from prefetch memory or submitted to disk, our handler initiates a set of prefetch requests (at most two in our experiments) for this bio. It searches the CORRELATION_RADIX_TREE to find out which blocks to prefetch. Any block that is present in the SSD or in the prefetched memory is not prefetched again. The modified handler creates a bio for each page to be prefetched and submits it to disk. The priority of these prefetch requests is set to BIO_IDLE_PRIORITY. When any prefetch request is completed, our customized bio_end_io handler is called which then adds the prefetched page to the PREFETCH_RADIX_TREE.

5.6 Evaluation of prefetching

We evaluated correlation-directed prefetching (CDP) along with data partitioning implemented in the Linux kernel. Since, from the simulations (Figure 4.8(a)), we have seen that the impact of different block correlation analysis heuristics in CDP is negligible, we used only one of them in the Linux data block manager. In particular, we implemented the CDP-thinktime heuristic-based correlation-directed prefetching in the Linux kernel. Our experimental results show (Figure 5.6(a)) that, for the Web, Hm and Mds traces, data partitioning alone can improve performance more than data partitioning and
5.6. Evaluation of prefetching

CDP together can do. This is contrast to simulation results (Figure 4.8(a)), because in the simulations we carefully controlled prefetching (i.e. blocks present in the OS cache were not prefetched). Also, no prefetching was done for the requests representing OS read ahead. However, in the Linux kernel module implementation, we could not control prefetching at this level because of excessive overhead. Linux employs an effective sequential prefetching technique that, depending on the access sequence, prefetches more or less blocks. Thus many requests were actually served from the OS cache and did not even appear in the disk request queue where we intercepted requests.

---

![Graph](image)

**Figure 5.6:** (a) Performance of data partitioning and correlation-directed prefetching technique implemented in the Linux kernel. Our HDD is not able to cope with the requests in the traces, and hence we used 20 ms constant think time for this experiment. (b) Percentage of data blocks read from each device for the CDP technique.
5.6. Evaluation of prefetching

To better understand why CDP performs poorly we computed the number of evictions from the prefetch cache and the percentage of prefetched pages that are actually read from the prefetch cache. The number of evictions gives us some indication about whether the prefetched pages are being evicted prematurely. In our experiments, we have seen that each page in the prefetch cache is evicted 0 or 1 times for most of the traces during the whole trace replay period which is more than a few hours for each of the traces. This number is not significant enough to indicate premature eviction.

To see whether prefetching results in more overhead than benefit, we computed the percentage of prefetched pages that were actually accessed by the workloads. We observed that for most of the traces a large number of prefetched pages remained unused (Figure 5.7) which explains some of the overhead associated with prefetching.

![Figure 5.7: Percentage of the prefetched pages that are never accessed, accessed once and accessed at least twice.](image)

We also computed the percentage of prefetch requests that are not completed on time. These requests only add to the overhead of prefetching and do not have any impact on performance improvement. We observed that only less than 1% of the total blocks read were not prefetched on time.
5.7 Limitations of prefetching

Interference with Application Reads  Prefetching requests interfere with demand requests as they require access to the disk. To account for this, the prefetch requests I/O priority should be set to idle.

Not Synched with Linux Read Ahead  Prefetching should be synchronized with the operating system caching and read ahead. Otherwise a page present in OS cache can be prefetched which only adds overhead. In our implementation, we haven’t incorporated prefetching with OS caching or read ahead yet.
Chapter 6

Literature Review

Magnetic hard disk drives are currently the most common medium of secondary storage since the sixties. These devices show significant differences in sequential vs. random accesses — sequential access is much faster than random access as the latter involves movements of mechanical arms (seek latency) and rotational delay. Many schemes have been developed and a lot of them are employed in different types of storage systems to optimize disk accesses. Implementations of such techniques are found in the operating system, for example, buffers can hold temporal and spatial locality of references so that some of requests can be served from memory thus reducing disk access. Writes can also be buffered in memory and coalesced to reduce write latency. In device driver level, request queues and request scheduling algorithms such as SCAN, FSCAN, C-LOOK and other such algorithms help merging multiple disk requests so that disk arm movements are minimized.

In the disk controller level, buffers are also used to cache readahead data so that not all requests that arrive at the controller result in actual disk access. These are very commonly used techniques and all modern computer systems have some or all of them installed. Recently commercialized flash memory based devices, such as solid-state drives do not involve any mechanical movement and are much faster than the disk drives. Many researchers now focus on combining flash memory and disk drives to improve disk performance (Section 6.1). Utilizing I/O access pattern is another important area of research for improving storage system performance. Re-organizing disk layout (Section 6.2) and prefetching correlated data (Section 6.4) are two interesting optimization techniques that utilize I/O access patterns.
6.1 Hybrid storage

The price differential that exists between solid-state drives (SSD) and magnetic hard disk drives (HDD) has motivated storage systems design that use NAND-based flash memory (the substrate for SSDs) as a cache on hard disks as in Hybrid Hard Disk Drive [3]; the guiding principle is that a small amount of expensive storage may be sufficient for reasonable performance improvements if used carefully.

In order to make the best use of the internal flash-based cache, the Seagate Momentus® XT Solid State Hybrid Drives, use the flash memory for clever caching by storing frequently accessed data. Using flash memory as a cache on a HDD improves read access performance for frequently accessed blocks or for sequential accesses where successive blocks are read ahead of time when a particular block is accessed. This leverages well established ideas from memory caches but even greater improvements are possible by understanding data access patterns of applications and customizing storage systems to meet application requirements. This is where data partitioning differs from the hybrid hard disk drives. Also, the hybrid drives are hardware solution and available as a complete device — they cannot be customized depending on different types of devices or different workload characteristics. On the other hand, data partitioning based on application long-range block access pattern offers a more general solution that can be used by any type of storage systems and also can be customized according to the needs of users.

Another approach to integrating SSDs in the storage system hierarchy is to treat the HDD and the SSD as a Combo Drive and to partition files between the SSD and HDD to improve performance [32]. In a Combo Drive, three different optimization strategies guide file movements between the two devices:

(i) Organization based on file type: Program libraries and executable files are moved to the SSD and the remaining files are moved to the HDD.

(ii) Organization based on file access: Files that are randomly accessed are moved to the SSD and files that are contiguously accessed are moved to
6.1. Hybrid storage

the HDD. Also, files that are both randomly and contiguously accessed are moved to the HDD.

(iii) Organization based on file access frequency: Most frequently accessed files are moved to the SSD and all other files are moved to the HDD.

Combo Drive has been evaluated with a Microsoft Word 2003 benchmark and a Windows XP startup benchmark. In both cases, the Combo Drive resulted in ten times improvement in launch times over a HDD-only system. The authors also measured read/write throughput by HDTunePro [2] and were able to get closer to SSD throughput. HDTunePro, however, reads/writes data from the beginning to the end of a device and does not reflect any real workload. The Combo Drive provides coarse partitioning whereas we advocate finer partitioning granularity. Depending on the application characteristics, certain data blocks within a file may be accessed more often than others. It is, therefore, advantageous to partition data blocks inside files between the two devices.

Makatos et al. [27] introduced SSD as a compressed cache. They proposed an I/O system, FlaZ, that performs transparent and online I/O compression to increase the capacity of the SSD-based cache. Although FlaZ increases SSD efficiency as a cache by up to 99%, it all comes as a trade off of CPU cycles needed for the compression and decompression of each block as it goes to the cache or is retrieved from the cache respectively. Another problem of using SSD as a cache is that multiple smaller writes may wear out the device more quickly. Although FlaZ buffers writes in memory and only writes to SSD in large chunks, but it does not solve the problem completely.

Although caching in RAM significantly reduces read latency, write latency is not improved significantly because of the non-volatility characteristics of RAM as applications often require that writes be persistent. To reduce hard disk write latency Bisson and Brandt [11] presented a flash backed I/O scheduling algorithm that leverages a small on-disk flash memory. Instead of writing immediately to the disk, the I/O scheduling algorithm queues up write requests in main memory backed up by the flash memory whenever it results in lower access time than writing to disk. This write re-
direction has the benefit of reducing seek and rotational latency even more as more requests can now be kept in main memory and coalesced together which results in reduction in write latency by up to 70%. This scheme also reduces disk power consumption.

Kundeti et al. [24] proposed a flash backed storage to increase reliability. Assuming a daily backup service, the authors log only per day changes of data to a small flash memory. If the disk fails, all the data can be recovered from the flash and from the backup storage. The read throughput overhead is negligible and for only about 29% overhead on the write throughput, the flash backed system can achieve RAID1 reliability.

6.2 Re-organizing data layout

Workload characteristics have long been used as a guidance to improve disk performance. Bhadkamkar et al. [10] and Li et al. [25] both proposed techniques to put correlated data together in the disk. Their observation is that placing correlated blocks contiguously in a smaller area of a disk reduces seek and rotational latency and hence improves disk utilization as only one disk access is required to fetch the correlated data. Bhadkamkar et al. [10] used a separate disk partition to cache correlated data blocks and to buffer writes to disk. Their approach reduced disk busy times by 5% to 50% depending on different types of workloads. The authors used semantic graph based methods to analyze block correlation (details in Section 4.1.1) and implemented their system in the Linux kernel. The authors, however, do not measure the impact of re-organizing data layout on response time. Li et al. [25] used a frequent sequence mining algorithm to mine block correlations and utilized that for re-organizing disk layout and for prefetching. They were able to improve I/O response times by 12–25% over no prefetching and 7–20% over commonly used sequential prefetching techniques by using both correlation directed disk layout and prefetching. They used simulation based experiments and noticed that correlation directed prefetching has more impact on reducing I/O response times than correlation directed disk layout has.

Akyurek and Salem [7] proposed to monitor stream of requests online
and to copy the frequently accessed blocks to a smaller place near the center of the disk. The authors used block level counters to keep track of access frequencies and found that only 12–13% of all the blocks are referenced in a day level trace. Using trace driven simulations the authors show that seek times can be reduced by half by re-arranging only 1–2% of the data. In the new arrangement, only 5–10 reserved cylinders near the center of the disk absorb 70–80% of the total requests.

In general, disk shuffling refers to the types of data re-arrangements where groups of frequently accessed data are moved towards the center of the disk to improve disk performance. Disk shuffling can be performed by disk controller or by the device driver. Shuffling quanta may also vary for different techniques, for example, both block or cylinder level shuffling is possible. Ruemmler and Wilkes [33] studied, extended and partially validated different disk shuffling techniques. Their observation is that, although these techniques can achieve significant reduction in seek times, the overall improvement in disk performance is not as significant. The best results can be obtained by doing relatively infrequent shuffling (once a week) and with small granularity of data, such as blocks or track sized data.

Hsu et al. [22] improved over the common disk shuffling techniques by taking into account not only the frequently accessed blocks but also their whole read sequences. They analyzed the request streams to identify long repeated read sequences and placed these sequences physically closer in disk. Trace driven simulations on real workloads demonstrate that this technique outperforms disk shuffling techniques to improve read performance by 50%.

Other studies concerning disk arrangement include cylinder shuffling by Vongsathorn and Carson [15] and file based re-organization by Staelin and Garcia-Molina [37].

6.3 Disk I/O scheduling

Researchers also focused on finding efficient I/O scheduling techniques instead of disk re-organization to improve disk performance [26,34,35]. Lumb et al. [26] proposed a freeblock scheduling technique that anticipates the
amount of rotational delay for the foreground processes and uses this free bandwidth opportunity for serving background requests such as backup, virus scanning and prefetching.

6.4 Prefetching

Prefetching, also known as read-ahead, is a way to hide higher latencies of the lower level devices in the memory hierarchy. Different types of prefetching are described in more detail in the following sections.

6.4.1 Sequential prefetching

Predicting future disk accesses and reading ahead of time have long been employed by different types of operating systems to improve sequential read performance. A perfect example application where sequential read-ahead guarantees improved performance is accessing media files. However, if not done carefully, read-ahead may even degrade performance as it could pollute the OS cache, cause read-ahead thrashing or interfere with demand requests. Also, not all disk requests become benefitted by sequential read-ahead.

Linux implements a generic read-ahead and read-around framework for buffered reads and memory mapped file accesses respectively [18]. For buffered reads, Linux tries to predict patterns and prefetch data ahead of time. However, this is done in a conservative way — there is small limit on how many pages will be read ahead with each page fault, and this number is increased only if a strict sequential pattern is detected in the request stream. These prefetched pages reside in the page cache together with the cached pages. Read-around framework is designed mainly for executables and binaries, where for each cache miss, a number of pre-faults are auto generated around that faulty page. There are also a number of system calls available in Linux that enables applications to alter the basic prefetching or implement their own prefetching, for example madvise, posix_fadvise and readaheadread.

The sequential read-ahead strategy has been studied and extended by
6.4. Prefetching

a number of researchers; Fengguang et al. [18] proposed a number of optimizations implemented as a new read-ahead framework for Linux that performs both sequential and semi-sequential prefetching. Papathanasiou and Scott [30] argued on behalf of aggressive prefetching to reduce the gap between today’s increased processor speeds and relatively slower disks. Integrated caching and prefetching, their optimization and resource allocation are studied in [12–14, 19].

6.4.2 Application-directed prefetching

Not all files are sequentially accessed and hence cannot be benefitted by sequential prefetching. Random accesses also show deterministic patterns that can be utilized to prefetch correlated data ahead of time. Applications’ knowledge of their access pattern can be utilized in identifying block correlation. In application-directed prefetching systems, applications inform the storage system about their future accesses. Patterson et al. [31] showed that their informed prefetching technique can reduce execution times by 20–83%. Vandebogart et al. [38] developed a library, libprefetch, that implements application-directed prefetching to reduce seek times for non-sequential accesses. libprefetch achieved 4.9x and 20x speed ups for image processing and database workloads respectively. Application-directed prefetching can utilize most accurate block correlations and hence can result in higher speed ups; but there are three disadvantages with this approach: (i) it is not transparent to applications, i.e., applications need to be modified to communicate their future accesses with the storage system, (ii) it is assumed that applications know about their future accesses long enough before sending the actual requests, and (iii) each application communicates their future requests individually, i.e., not knowing about other concurrent applications.

6.4.3 Heuristic-based prefetching

The heuristic-based prefetching techniques tend to analyze workload characteristics or access streams to predict future references without any help from the application front-ends. Correlated accesses tend to be closer together,
in space and in time and these accesses also exhibit repetition. Therefore, if two blocks are almost always accessed together within a short distance and time, they might be correlated. Researchers proposed semantic distance-based methods \cite{10} or frequent sequence mining algorithms \cite{25} to compute correlation between two data blocks. Vitter and Krishnan \cite{39} established the theoretical basis for using data compressors as prefetchers for large scale databases and hypertext systems and Curewitz et al. \cite{16} analyzed the practical aspects of the idea. The intuition behind the idea is that compressors best perform by predicting future data references and thus assign shorter codes to them — thus compression algorithms can be adopted for predicting future I/O accesses as well.

6.4.4 File prefetching

File level prefetching shows a coarser granularity of prefetching than block level prefetching. Griffioen and Appleton \cite{20} describe automatic prefetching that masks file system latency by predicting future file requests from past history of file accesses and prefetching as necessary. The authors implemented a probability graph based analyzer that, given a look-ahead period determines which files are correlated. Two files are considered correlated if they are opened consecutively within the specific look-ahead period. The authors also studied the performance and stability of automatic prefetching under various file systems \cite{21}. Automatic prefetching reduced cache miss rates by up to 70\% for certain cache sizes and reduced average read access time by up to 42\%.

Kuenning \cite{23} implemented file prefetching for portable platforms. Semantic distance based methods are used to predict future file accesses and those files are cached before the portable platform gets disconnected from the server. For file prefetching in mobile devices, Andraus et al. \cite{8} demonstrated the importance of combining multiple predictors (such as sequence of previous access, semantic distance or directory membership) and dynamically changing their weightage based on previous learning.

Yeh et al. \cite{42, 43} used both user and program level information to in-
6.4. Prefetching

crease the accuracy of file prediction for prefetching. The *Last Successor* (LS) model predicts the next file access based on the last accesses. For example, in the file trace, if file B was accessed after last access of file A, then LS predicts B as the future access whenever file A is accessed. Yeh et al. [42, 43] extended the LS model by taking into account the users and programs that were involved in the previous accesses and showed that their approach results in at least 20% fewer incorrect predictions than the LS model.
Chapter 7

Conclusion

In this work, we have investigated a hybrid storage system consisting of two storage devices with varying latency and bandwidth. The hybrid system utilizes application long-range block access patterns by partitioning file system data blocks between the two devices and prefetching correlated blocks when a related block is read. Whereas correlation-directed prefetching has been investigated previously, we looked into some special aspects of CDP and also investigated how data partitioning can improve the performance of CDP.

We evaluated the hybrid storage system through an event-driven simulator and used real enterprise-level traces collected from Microsoft Research servers.

We presented a case study — a data block manager implemented in the Linux kernel. We implemented data partitioning by re-mapping inode pointers along with data movement. We developed a Linux kernel module that implements correlation-directed prefetching by intercepting requests from the disk request queue and issuing prefetch requests as necessary. Our evaluation shows that data partitioning can improve the performance of the Linux file system by up to 79%. Correlation-directed prefetching, however, require more investigation as we observed that the improvement from prefetching was canceled out by the overhead of the same.

We mainly answer the following broad questions:

(i) What characteristics do we usually see in application long-range block access patterns and how can we utilize these to improve file system performance?

(ii) How can we combine two storage media with different characteristics
to improve file system read performance?

(iii) How can we integrate data partitioning and correlation-directed prefetching?

(iv) How different block correlation analysis heuristics affect the performance of CDP?

Application long-range block access patterns show interesting characteristics. Our analysis of some real traces collected from Microsoft Research servers show high degree of non-sequentiality, significant repetition and long-range overlap in block access. The presence of block correlation even in non-sequential accesses and non-uniformity of access are the other two characteristics that gained much attention from other researchers [10, 25].

The non-uniformity of access, repetition and long-range overlap motivated us to investigate a data partitioning technique where file system data blocks are split between two non-volatile storage devices with different performance characteristics. This data partitioning technique moves frequently accessed blocks to a small high performance device having lower latency so that most of the read requests can be served with low overhead. A large low performance device provides for the capacity requirement. Our evaluation of the system shows that data partitioning improves file system read performance by 2–92%.

Data partitioning is different from caching in two aspects: (i) instead of utilizing only spatial and temporal locality, partitioning takes advantage of the applications’ long-range block access patterns and (ii) instead of copying, data blocks are actually moved across the two media. Moving data blocks has the advantage of better utilization of the capacity of both devices and also solves the write-back or write-through debate found in all cache involved systems. We show that data partitioning based on application long-range block access patterns improves file system read performance over LRU-based caching by up to 64%. Data partitioning is a more suitable choice than caching for SSD-based systems. This is because caching involves continuous updating of blocks in the SSD which may wear out the device more quickly.
7.1 Future work

The poor performance of SSD small writes may also affect the performance of caching.

We show that the overhead of correlation-directed prefetching can be reduced by utilizing the slack that results from data partitioning. In the slack-based prefetching technique, we issue prefetch requests only when its related blocks are completely served from the faster device, i.e., when a request do not require access to the slower device. We show that when think time is small, the slack-based integration of data partitioning and CDP improves overall read performance.

We implemented correlation-directed prefetching using two different block correlation analysis heuristics, BORG [10] and C-Miner [25]. Both heuristics aim at mining block correlation, but their perspectives are different. Our implementation shows that the difference in mining block correlations do not have much effect on prefetching performance.

Motivated by the observation that correlated blocks tend to be accessed together not only within a short distance but also within a short period of time, we modified a block correlation analysis heuristic to include application think time in the analysis. Our evaluation, however, shows that including time in the analysis neither improves nor degrades performance.

We investigated prefetching from the chain of correlation — instead of prefetching the immediately correlated block we look ahead in the chain of correlation for prefetching. This approach essentially gives us more time to complete the prefetch requests before they are demanded and improves performance when think time is low.

7.1.1 Write optimization

The direct extension of our work would be to analyze how data partitioning affects file system write performance. In this work, we partitioned file system data blocks based on read access frequency only. An effective partitioning strategy needs to be developed when considering both read and
write workloads.

Write optimization is specially needed with SSD-based hybrid systems. Due to the fact that SSD blocks need to be erased before they can be updated, SSD small-writes are sometimes magnified by the controller when performing wear leveling. SSD small writes can be avoided by using a disk-based write cache [36].

7.1.2 Impact on energy consumption

The flash memory based devices do not require movement of mechanical parts for storing/retrieving data and hence consume very low power as compared to the hard disk drives. The impact of data partitioning on energy consumption would be an interesting topic of study when two such devices are combined.

7.1.3 Minimization of fragmentation

Data partitioning results in fragmentation in the underlying storage devices. For some devices, fragmentation does not matter (for example, devices based on flash memory have the same access time for every block). Currently, many storage systems employ magnetic hard disk drives whose performance degrades greatly with fragmentation. Many off-the-shelf hard disk defragmenter tools are available that can be easily incorporated with data partitioning.

7.1.4 Start blocks in faster device

In addition to moving the frequently accessed blocks, we can also move the first few blocks of each application/file to the faster device and prefetch the rest of the blocks as soon as these start blocks are read. This way the files or applications would have a quick launch time as well as quick response times if the future data blocks can be prefetched in time.
7.1.5 Incorporation of correlated prefetching with caching in Linux

We did not see performance improvement from correlated prefetching in the Linux kernel. One reason could be because we implemented correlated prefetching separately without synchronizing with Linux file system caching. It would be interesting to see if CDP results in performance improvement when integrated with Linux caching.
Bibliography

[1] Flash memory vs. hard disk drives - which will win? Website. 
   http://www.storagesearch.com/semico-art1.html

   http://www.hdtune.com/

   http://en.wikipedia.org/wiki/Hybrid_drive/

   http://www.amazon.com/s/ref=nb_sb_noss?url=search-alias%3Daps&field-keywords=intel+ssd&x=0&y=0

   http://en.wikipedia.org/wiki/Memory_hierarchy

   http://www.snia.org/home/


   prefetching for mobile devices using on-line learning.


[10] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, 
    and V. Hristidis. Borg: Block-reorganization for self-optimizing storage systems. 
    FAST, 2009.


[41] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In In SDM, pages 166–177, 2003.

[42] Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt. Increasing predictive accuracy by prefetching multiple program and user specific files.