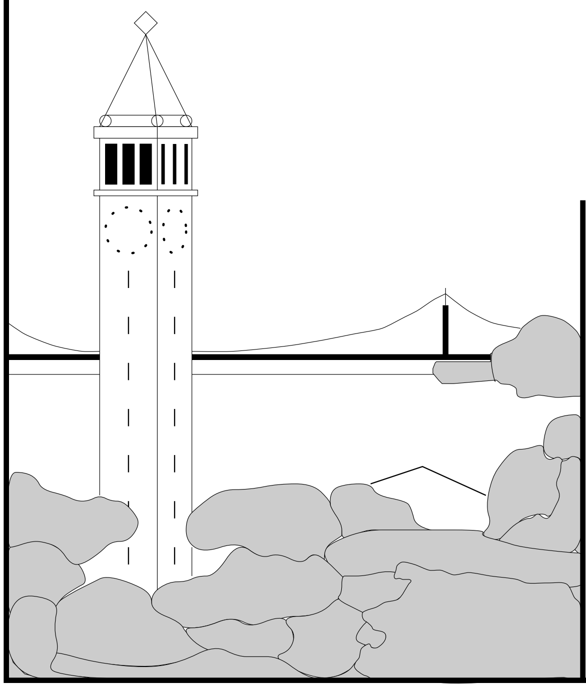
Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance

Windsor W. Hsu



Report No. UCB/CSD-03-1223

January 2003

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance

by

Windsor Wee Sun Hsu

B.S. (University of California at Berkeley) 1994
M.S. (University of California at Berkeley) 1999

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Alan Jay Smith, Chair
Professor David A. Patterson
Professor Dorit S. Hochbaum
Dr. Honesty C. Young

Fall 2002

The dissertation of Windsor Wee Sun Hsu is approved:

Chair Date

Date

Date

Date

University of California at Berkeley

Fall 2002

Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance

Abstract

Dynamic Locality Improvement Techniques for
Increasing Effective Storage Performance

by

Windsor Wee Sun Hsu
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Alan Jay Smith, Chair

As the processor-disk performance gap continues to widen and computer systems become ever more complex, increasingly aggressive and automated methods are needed to improve I/O performance. This work concerns mechanizable ways to effectively increase I/O performance. Our specific contributions are as follows.

First, we introduce our thesis that it is useful and practical for the storage system to be introspective and to automatically optimize disk block layout in response to the dynamic reference behavior. Based on the technology trends, we posit that a promising approach to improving I/O performance is to replicate selected disk blocks and to lay them out to increase the spatial locality of reference.

Next, we characterize how storage is used in real personal computer and server systems. Among the things we analyze are the overall significance of I/O in the workloads, how the I/O load varies over time, and the interaction of reads and writes. We find that there are idle resources that can potentially be used to perform any optimization, and that optimizing the disk block layout is likely to achieve useful benefit.

Based on our workload analysis, we next develop a simulation methodology that more accurately models the timing of I/O arrivals than previous practice. Using this methodology, we study I/O optimization techniques such as read/write caching, sequential prefetching, opportunistic prefetching, request scheduling, striping and short-stroking to understand their actual effectiveness, and to establish an optimized baseline configuration for subsequent experiments. We also examine disk technology trends to quantify their performance effect.

Next, we introduce Automatic Locality-Improving Storage (ALIS), a storage system that automatically replicates and reorganizes selected disk blocks based on their usage patterns. Using trace-driven simulations, we demonstrate that the techniques we develop for ALIS are relatively insensitive to disk technology trends and are extremely effective, improving average read performance by up to 50% for servers and by about 15% for personal computers.

Finally, we consider the additional use of processing power in the storage system to handle application processing such as running decision support queries. Our results suggest that highly scalable parallel software systems are needed for this to be effective.

Professor Alan Jay Smith
Dissertation Committee Chair

Dedicated to my wonderful parents,
Loke and Bee,
and to my lovely wife,
Jaina.

Contents

List of Figures

List of Tables

Acknowledgements

Many people contributed to making this thesis a reality for me. I am much obliged to my advisor, Alan Jay Smith, for giving me the freedom and independence to pursue different areas of research. I began my research with him on aspects of microprocessor design, then moved on to database performance on which I did my Master's thesis, before settling on storage performance. I especially thank him for being so supportive and for giving advice that is always insightful. He is truly a master of many arts from whom I am privileged to learn.

I would also like to express my gratitude to David Patterson, Dorit Hochbaum and Honesty Young for their helpful feedback which added new perspectives to my research. Thanks are also due to Jim Gray and Joseph M. Hellerstein for their very encouraging comments on my work.

During most of my time in graduate school, I was generously supported by IBM, first through an IBM fellowship and subsequently as an employee. I conducted most of this work while a full-time employee at the IBM Almaden Research Center. I would like to express my gratitude to management for their support and continued interest in my research, and for providing the resources for much of this work. I would also like to thank the many colleagues there for the stimulating environment and the practical industry perspective.

In particular, I am much indebted to Honesty Young who has been a great mentor to me since my first summer internship at Almaden as an undergraduate. Besides sharing with me his keen insights into computer performance, especially benchmarking, he has shown a genuine interest in helping me grow in all aspects of life. I would also like to thank Jai Menon, Shauchi Ong, John Palmer and T. Paul Lee for their continuous encouragement. In addition, I would like to thank Peter Haas who assisted me with statistical procedures, William Guthrie and Spencer Ng who helped me in modeling disks, Ed Grochowski who provided me with historical data on disk specifications, and Guy Lohman who made the TPC-D query execution plans available to me. Bruce Lindsay and Ying Chen also helped me greatly by reviewing some of my early work.

I also received a lot of help and encouragement from fellow students at Berkeley. I am most grateful to Jacob Lorch, Jeffrey Rothman and Stephen Von Worley for showing me the ropes, to Kimberly Keeton who reviewed early versions of some of this work, and to Boon Thau Loo who spent a summer at Almaden performing some early trace processing and workload characterization. Thanks are also owed to Jih-Kwon Peir, whom I met at Almaden during a summer internship, for adding breadth to my research activities.

This research would have been seriously hampered if I had to collect all the trace data needed to validate the ideas by myself. I would like to acknowledge Ruth Azevedo, Carlos Fuente, Jacob Lorch, Bruce McNutt, Anjan Sen and John Wilkes for kindly sharing some of their traces with me, which greatly increased the breadth and diversity of the data available to me.

I am grateful to my parents, Loke and Bee, for their unconditional support and their wonderful example, and for leading me down this path by giving me the chance to be one of the very few among my peers to have played with the punch card and paper tape when I was a little boy. I am also thankful for my brother, William, and sisters, Wynne and Jean, who paved the way and set the expectation, making it difficult for me not to do this, not that I would have chosen differently.

In addition, I would like to express heartfelt gratitude to my lovely wife, Jaina, whom God brought into my life three years ago just as I began this work. We shared many happy moments together, although not often alone and away from this work. But she was understanding even when I brought along my laptop on our honeymoon to monitor my simulations. I am especially gratified by her ability to quickly recognize when I am deeply engrossed in an idea and to leave me alone to crunch through the idea, well at least most of the time.

Above all, I would like to thank God who makes all things possible.

Chapter 1

Introduction

I/O certainly has been lagging in the last decade
- Seymour Cray (1976)

I/O's revenge is at hand
- Hennessy and Patterson (1996)

Beating, Breaking, Cheating, Eliminating, Removing, Solving the I/O bottleneck
- various (1988-2002)

1.1 Motivation

Processor performance has been increasing at the phenomenal rate of more than 50% per year [PK98] while disk access time, being limited by mechanical delays, has been improving by less than 10% per year [Gro00, PK98]. As the performance gap between the processor and the disk continues to widen, disk-based storage systems are increasingly the bottleneck in computer systems, even in personal computers where storage performance has been identified as the primary cause of the delays that are highly frustrating to the user [Cor98]. The I/O bottleneck is further compounded by the almost annual doubling in disk capacity [Gro00], which far exceeds the rate of decrease in the access density or the number of I/Os per second per GB of data. For instance, surveys of disk usage in large mainframe installations conducted between 1980 and 1993 [McN95] found that the access density was decreasing by only about 10% per year. In other words, although the disk arm is only slightly faster in each new generation of the disk, each arm is responsible for serving a lot more data. There is therefore an increasingly pressing need to focus on improving I/O performance.

The I/O bottleneck has long been recognized as a serious problem and a plethora of optimization techniques have been invented to try to overcome it [Chapter 3]. However, many of these techniques have to be configured and tuned to work well for different workloads.

In a bid to wring more performance out of the system, many of the tuning parameters or knobs have been exposed so that the user or administrator is now faced with a multitude of knobs to adjust. It is, however, difficult or even impossible for the human mind to fully understand the dynamics of even a relatively simple operation, especially how it relates to the increasingly complicated storage hierarchy with its various levels of abstraction or virtualization. Even in the simple case of a storage system with only a single disk, its performance depends not just on where the data is placed and how the data is accessed but also on its dynamic state, specifically, what was accessed previously. Therefore, although raw storage is inexpensive, managing its performance is complex and costly. Thus the pressing need is not merely to improve I/O performance but to improve it transparently or autonomically [IBM01a], without requiring low-level user involvement.

Our thesis is that it is useful and practical for the storage system to be introspective and to have the intelligence to automatically optimize disk block layout based on the actual reference behavior and its own performance characteristics. In particular, we observe that although disk access time has been improving by less than 10% per year, disk transfer rate has been increasing by as much as 40% per year [Gro00, PK98]. Furthermore, with disk capacities growing quickly, increasing amounts of disk space are available for storage optimization. Therefore, we posit that a promising approach to improving I/O performance is to replicate selected disk blocks and to lay them out so as to increase the spatial locality of reference and thereby leverage the high and rapidly growing disk transfer rate. We contend that as more computing resources become available following Moore's Law [HP96] or can be added relatively easily to the storage system [Chapter 5], sophisticated techniques for optimizing storage performance transparently, without human intervention, are increasingly possible.

In this dissertation, we explore how storage systems tend to be used to show that the idea of automatically optimizing disk block layout is feasible. In addition, we propose various techniques for performing the optimization and demonstrate that they achieve useful and dramatic results.

1.2 Background

The time taken by the disk to service an I/O request is composed of two parts – access time and transfer time. Access time refers to the time required to position the disk head over the correct sector and comprises the delay in moving the disk arm to the correct track and the time spent waiting for the requested sector to rotate under the head. Transfer time is the actual time needed to read the requested data after the head is in position. In other words, transfer time is the only time during which data is being read or written, and access time is the overhead for the transfer.

Therefore, to effectively utilize the bandwidth of the disk and achieve good I/O performance, data has to be requested in large units so that the transfer time dominates the access time. As the disparity between random and sequential disk performance grows, we have to resort to increasingly larger transfer units, either by using bigger blocks or pages [GG97], or by more aggressive sequential prefetch. The effectiveness of transferring more data at a time is, however, constrained by the amount of spatial locality in the reference stream. For example, if the data that are located physically close together on the disk are not related in their usage patterns, using a larger block size or prefetch amount will result only in the transfer of extra data that are not likely to be used, thereby wasting resources like I/O bandwidth and cache space.

Two general approaches have been used to increase spatial locality in the reference stream. The first is to rely on the application developer or the administrator to ensure that accesses are localized. Since the I/O reference behavior of a program is a direct consequence of the kinds of operations it performs and the order in which it performs them, the application developer has some control over the locality present in the reference stream. But applications are increasingly complicated as is the storage hierarchy, and the I/O reference behavior is likely to be dynamic and to vary with the data. Thus, we cannot generally expect programmers to be very successful at developing programs that exhibit good spatial locality, especially since the goal is locality system-wide and the system is likely to be multiprogrammed. Similarly system administrators, who have even less knowledge and control, cannot be expected to be very effective at ensuring that related data are clustered together.

The second approach is for the system to use various heuristics to lay out data on disk so that data that are expected to be used contemporaneously are located close to one another (*e.g.,* [GK97, MJLF84, MK91, Pea88]. The shortcoming of these *a priori* techniques is that they are based on static information such as the name space relationships of files, which may not reflect the actual reference behavior. Therefore, we believe that the reference stream is often not as local as it can be, and that there may be considerable opportunities for mechanizable techniques that utilize information about the dynamic reference behavior to improve spatial locality.

1.3 Outline of Dissertation

This dissertation has six chapters and six appendices. The first chapter introduces and motivates this work. The last chapter concludes the dissertation and discusses some avenues for future work. The four chapters in the middle address the following.

Chapter 2 characterizes how storage is actually used in real environments. We examine multi-week traces of the I/O activity of a wide variety of both personal computer and server systems to explore whether it is feasible and useful to automatically optimize disk block layout. Among the things we analyze are the I/O intensity of the workloads, the overall significance of I/O in the workloads, how the I/O load varies over time, and the interaction of reads and writes. Our results indicate that improving I/O performance is important, and that there are idle resources that can potentially be used to perform any optimization. We also discover that only a small fraction of the data stored is in active use, suggesting that it will be useful to identify the blocks that are in use and to optimize their layout.

Based on the results of our workload analysis, we develop, in Chapter 3, a simulation methodology that more accurately models the timing of request arrivals in real workloads than previous practice. Using this methodology, we systematically study the many I/O optimization techniques (*e.g.,* read caching, sequential prefetching, opportunistic prefetching, write buffering, request scheduling, striping and short-stroking) that have been invented over the years to determine their actual effectiveness at improving I/O performance for our various workloads, and the optimal parameters for each technique. The optimal parameters establish a baseline configuration for our experiments in the subsequent chapter. We also examine disk technology trends to quantify the actual performance effect of the evolution in disk technology.

In Chapter 4, we introduce Automatic Locality-Improving Storage (ALIS), a storage system that automatically replicates and reorganizes selected disk blocks to take advantage of technology trends to improve the effective performance of disk-based storage. Using extensive trace-driven simulations, we present and motivate the various algorithms that we develop to select blocks for replication and reorganization, and to lay these blocks out. We demonstrate that these techniques are extremely effective, improving the average read response and service times for server workloads by as much as 50% and those for personal computer workloads by about 15%. As part of our analysis, we also examine how improvement in disk technology will impact the effectiveness of ALIS and confirm that the benefit of ALIS is relatively insensitive to disk technology trends.

As processing power becomes increasingly available in the storage system, it enables sophisticated optimizations such as those performed by ALIS. The processing capability can also be used to offload application processing, parts of file system functionality (*e.g.,* object-based storage), *etc.* from the host system. In Chapter 5, we project the effectiveness of offloading application processing to the storage system by performing some meta-analysis of published benchmark results. Our findings suggest that for the storage system to effectively perform application processing such as running decision support queries, we have to develop parallel software systems that are scalable and that can efficiently utilize the large number of processing units that will likely be in such a storage system.

Chapter 2

Characteristics of I/O Traffic in Personal Computer and Server Workloads

2.1 Synopsis

Understanding the characteristics of I/O traffic is increasingly important as the performance gap between the processor and disk-based storage continues to widen. Recent advances in technology, coupled with market demands, have, moreover, led to several new and exciting developments in storage, including network storage, storage utilities, and intelligent self-optimizing storage. In this chapter, we empirically examine the physical I/O traffic of a wide range of real server and personal computer (PC) workloads, focusing on how these workloads will be affected by the new storage developments. As part of our analysis, we compare our results with historical data and reexamine rules of thumb (*e.g.,* bits per second per MIPS, I/Os per second per megabyte) that have been widely used for designing computer systems. We find that the I/O traffic is bursty and appears to exhibit self-similar characteristics. Our analysis also indicates that there is little cross-correlation in traffic volume among the server workloads, which suggests that aggregating these workloads will likely help to smooth out the traffic and enable more efficient utilization of resources. We discover that there is a lot of potential for harnessing "free" system resources for purposes such as automatic optimization of disk block layout. In general, we observe that the characteristics of the I/O traffic are relatively insensitive to the amount of caching upstream and that our qualitative results still apply when the upstream cache is increased in size.

2.2 Introduction

Processor performance has been increasing at the rate of 60% per year [HP96] while disk access time, being limited by mechanical delays, has been improving by less than 10% per year [Gro00]. Compounding this widening performance gap between processor and disk storage is the fact that disk capacity has been growing by more than 60% per year recently [Gro00] so that each disk is responsible for the storage and retrieval of rapidly increasing amounts of data. The overall result of these technology trends, which show no signs of easing, is that computer systems are increasingly bottlenecked by disk-based storage systems. The key step in overcoming this bottleneck is to understand how storage is actually used so that new optimization techniques and algorithms can be designed.

A focused examination of the I/O characteristics of real workloads is also needed to determine the actual effect of the new paradigms and developments that have recently emerged in the storage industry. First, storage is increasingly placed on some form of general network so that it can be shared and accessed directly by several computers at the same time (*e.g.,* Network Attached Storage (NAS) for file storage and Storage Area Networks (SANs) for block storage). The performance of such network storage hinges on knowing the I/O traffic patterns, and optimizing the network for such patterns. Second, consolidating the storage now distributed throughout an organization, for instance to storage utilities or Storage Service Providers (SSPs), is expected to become increasingly popular. Whether such an approach leads to more efficient pooling of resources among different groups of users depends on the characteristics of their workloads, specifically on whether the workloads are independent. In practice, we will need rules of thumb that describe the storage and performance requirements of each group of users, as well as realistic traffic models. Third, the rapid growth in available processing power in the storage system (*e.g.,* [Chapter 5] and [Gra98]) makes it possible to build intelligent storage systems that can dynamically optimize themselves for the workload [Chapter 4]. The design of these systems requires a good understanding of how real workloads behave.

In this research, therefore, we empirically examine how storage is used by a variety of real users and servers from the perspective of evaluating these new storage opportunities. A total of 18 traces gathered from a wide range of environments are examined. We focus in this chapter on analyzing the physical I/O traffic, specifically, (1) the I/O intensity of the workloads and the overall significance of I/O in the workloads, (2) how the I/O load varies over time and how it will behave when aggregated, and (3) the interaction of reads and writes and how it affects performance. We compare our results with historical data to note any trends and to revalidate rules of thumb that are useful for systems design and sizing. To make our results more broadly applicable, we also study the effect of increased upstream caching on our analysis. In the next chapter, we examine how these real workloads are affected by disk improvements and I/O optimizations such as caching and prefetching. The insights gained from this research are instrumental to the block reorganization technique presented in Chapter 4.

The rest of this chapter is organized as follows. Section 2.3 contains a brief overview of previous work in characterizing I/O behavior. Section 2.4 discusses our methodology and describes the traces that we use. In Sections 2.5 to 2.7, we analyze the I/O traffic of our various workloads in detail. Concluding remarks appear in Section 2.8. Because of the huge amount of data that is involved in this study, we present only a characteristic cross-section in the main text. More detailed graphs and data are presented in Appendix A. Some of the more involved mathematical material appears in Appendix B.

2.3 Related Work

The behavior of I/O at the file system level has been characterized in some detail (*e.g.,* [ODH+85, BHK+91, BGW91, Vog99, RLA00]). There have also been several studies of the logical I/O characteristics of large database and scientific systems; see [HSY01a, HSY01b] for a brief bibliography. Compared to the analysis of I/O behavior at the logical level, physical I/O characterization has received much less attention. Part of the reason is that storage level characteristics are sensitive to the file system and buffer pool design and implementation so that the results of any analysis are less broadly applicable. But this is precisely the reason to analyze the physical I/O characteristics of different systems.

Traces of the physical I/Os in large IBM mainframe installations [Smi85] and production VAX/VMS systems [BRT93, KLW94] have been used to study design issues in disk caches. There has also been some analysis of the physical I/O characteristics of Unix systems [RW93] and Novel NetWare file servers [HH95] in academic/research environments. Even though personal computers (PCs) running various flavors of Microsoft Windows are now an integral part of many office activities, there has, to the best of our knowledge, been no published systematic analysis of how physical storage is used in such systems.

2.4 Methodology

Trace data can generally be gathered at different levels in the system depending on the reason for collecting the data. For instance, to evaluate cache policies for the file system buffer, I/O references have to be recorded at the logical level, before they are filtered by the file system buffer. In general, collecting trace data at the logical level reduces dependencies on the system being traced and allows the trace to be used in a wider variety of studies, including simulations of systems somewhat different from the original system. For example, to study physical storage systems, we could filter a logical trace through models of the file system layer to obtain a trace of the physical I/Os. A commonly used method for obtaining such a logical trace is to insert a filter driver that intercepts all requests to an existing file system device, and records information about the requests before passing them on to the real file system device.

However, this approach does not account for I/Os that bypass the file system interface (*e.g.,* raw I/O, virtual memory paging and memory-mapped I/O). Recent results [RLA00] show that 15% of reads and nearly 30% of writes in Windows NT workloads can be attributed to paging by running programs. In addition, 85% of processes now memory-map files compared with 36% that read files and 22% that write them. From a practical perspective, the approach of starting with a logical trace to evaluate physical storage systems requires that a lot of data be collected, which adds disturbance to the systems being traced, and then painstakingly filtered away by simulating not only the buffer cache and prefetcher but also how the data is laid out and how the metadata is referenced. For today's well-tuned systems, each of these components is complicated and the details of their operation are seldom publicly available. For instance, the file system buffer on many systems (*e.g.,* Windows NT) is integrated with the memory manager and dynamically sized based on perceived workload characteristics. Therefore, the net result of taking a logical trace and filtering it through models of the file system components is not likely to reflect the workload seen by any real storage system. Since file systems today are relatively stable and rarely undergo radical changes, we believe that for the purpose of studying physical storage systems, analyzing traces collected at the physical level is generally more practical and realistic. This is the method we use in this thesis.

In order to make our characterization more useful for subsequent mathematical analyses and modeling by others, we fitted our data to various functional forms through non-linear regression, which we solved by using the Levenberg-Marquardt method [PFTV90]. When appropriate, we also fitted standard probability distributions to our data by using the method of maximum likelihood to obtain parameter estimates and then optimizing these estimates by the Levenberg-Marquardt algorithm [PFTV90].

2.4.1 Trace Collection

The traces analyzed in this study were collected on three different platforms – Windows NT, IBM AIX and HP-UX. A different trace facility was used on each platform. The Windows NT traces were collected by using VTrace [LS00], a software tracing tool for Intel x86 PCs running Windows NT and Windows 2000. VTrace was primarily developed to collect data for energy management studies for portable computers. In this study, we focus mainly on the disk activities, which are collected by VTrace through the use of device filters. We have verified the disk activity collected by VTrace with the raw traffic observed by a bus (SCSI) analyzer.

After VTrace is installed on a system, each disk request generates a trace record consisting of the time (based on the Intel Pentium cycle counter), sequence number, file object pointer, disk and partition numbers, start address, transfer size, and flags describing the request (*e.g.,* read, write, synchronous). After the disk request has been serviced, a completion record is written. In a post processing step, we match up the sequence number recorded in the request and completion records to obtain the service times.

To better understand the I/O behavior of the system, it is useful to be able to associate each disk request with the name of the corresponding file and process. Because VTrace also collects data on file system activities, in most cases, we are able to match up the file object pointer with a file open record to obtain the filename. When the match fails, we try to determine the filename by looking up the block address in a reverse allocation map that is constructed from the daily snapshots that VTrace takes of the Windows NT file system (NTFS) metadata. Since VTrace was designed to collect data for energy management studies, it also gathers data about process and thread creations and deletions as well as thread switches. By using the thread create and thread switch trace records, we are able to match up I/O requests with the names of the requesting processes. In addition, the thread switch records enable us to determine the overall significance of I/O in these workloads. We will look at this in Section 2.5.1.

To keep the amount of data collected manageable, process and thread trace records are gathered only for a span of one and a half hours every three and a half hours. In addition, all trace collection is turned off ten minutes after the cessation of user mouse and keyboard activity. Newer versions of VTrace collect some trace data all the time but in order to have a consistent set of data, we have processed the traces used in this study to delete trace records that occur more than ten minutes after the last user keyboard or mouse activity. In other words, we consider the system to be idle from ten minutes after the last user action until the next user action, and we assume that there is no I/O activity during the idle periods. This means that the traces contain only the I/Os that occur when the user is actively interacting with the system, and which are therefore likely to be noticed. We believe that this is a reasonable approximation in the PC environment, although it is possible that we are ignoring some level of activity due to periodic system tasks such as daemons. This latter type of activity should have a negligible effect on the I/O load, although it might be important for other types of studies, such as power usage.

Both the IBM AIX and HP-UX traces were collected using kernel-level trace facilities built into the respective operating systems. These trace facilities are completely transparent to the user and adds no noticeable processor load. Among the information collected for each physical I/O are: timing information, disk and partition numbers, start address, transfer size and flags describing the request. More details about the IBM AIX trace facility can be found in [IBM96]. The HP-UX trace facility is described in [RW93].

2.4.2 Trace Description

In this study, we use traces collected on both server and PC systems. Table 2.1 summarizes the characteristics of the traces. The *footprint* of a trace is defined as the amount of data referenced at least once in the trace. Figure 2.1 plots the trace footprint as a function of the number of references, which is a measure of the trace length. Similar plots for the read footprint and the write footprint are in Figure A.1 in Appendix A.

The PC traces are denoted as P1, P2, ..., P14. The term "P-Avg." represents the arithmetic mean of the results for all the PC traces. These traces were collected on Windows NT PCs over a period ranging from about a month to well over nine months. In this thesis, we utilize only the first 45 days of the traces. In addition to engineers and graduate students, the users of these systems include a secretary and several people in senior managerial positions. By having a wide variety of users in our sample, we believe that our traces are illustrative of the PC workloads in many offices, especially those involved in research and development. Note, however, that the traces should not be taken as typical or representative of any other system. Despite this disclaimer, the fact that many of our results turn out to correspond to those obtained previously, albeit in somewhat different environments, suggest that our findings are to a large extent generalizable.

Design-ation	User Type	System Configuration					Trace Characteristics			
		System	Memory (MB)	File Systems	Storage Used[i] (GB)	# Disks	Duration	Footprint[ii] (GB)	Traffic (GB)	Requests (10^6)
P1	Engineer	333MHz P6	64	1GB FAT[i] 5GB NTFS[i]	6	1	45 days (7/26/99 - 9/8/99)	0.945	17.1	1.88
P2	Engineer	200MHz P6	64	1.2, 2.4, 1.2GB FAT	4.8	2	39 days (7/26/99 - 9/2/99)	0.509	9.45	1.15
P3	Engineer	450MHz P6	128	4, 2GB NTFS	6	1	45 days (7/26/99 - 9/8/99)	0.708	5.01	0.679
P4	Engineer	450MHz P6	128	3, 3GB NTFS	6	1	29 days (7/27/99 - 8/24/99)	4.72	26.6	2.56
P5	Engineer	450MHz P6	128	3.9, 2.1GB NTFS	6	1	45 days (7/26/99 - 9/8/99)	2.66	31.5	4.04
P6	Manager	166MHz P6	128	3, 2GB NTFS	5	2	45 days (7/23/99 - 9/5/99)	0.513	2.43	0.324
P7	Engineer	266MHz P6	192	4GB NTFS	4	1	45 days (7/26/99 - 9/8/99)	1.84	20.1	2.27
P8	Secretary	300MHz P5	64	1, 3GB NTFS	4	1	45 days (7/27/99 - 9/9/99)	0.519	9.52	1.15
P9	Engineer	166MHz P5	80	1.5, 1.5GB NTFS	3	2	32 days (7/23/99 - 8/23/99)	0.848	9.93	1.42
P10	CTO	266MHz P6	96	4.2GB NTFS	4.2	1	45 days (1/20/00 – 3/4/00)	2.58	16.3	1.75
P11	Director	350MHz P6	64	2, 2GB NTFS	4	1	45 days (8/25/99 – 10/8/99)	0.73	11.4	1.58
P12	Director	400MHz P6	128	2, 4GB NTFS	6	1	45 days (9/10/99 – 10/24/99)	1.36	6.2	0.514
P13	Grad. Student	200MHz P6	128	1, 1, 2GB NTFS	4	2	45 days (10/22/99 – 12/5/99)	0.442	6.62	1.13
P14	Grad. Student	450MHz P6	128	2, 2, 2, 2GB NTFS	8	3	45 days (8/30/99 – 10/13/99)	3.92	22.3	2.9
P-Avg.	-	318MHz	109	-	5.07	1.43	41.2 days	1.59	13.9	1.67

(a) Personal Systems.

Design-ation	Primary Function	System Configuration					Trace Characteristics			
		System	Memory (MB)	File Systems	Storage Used[i] (GB)	# Disks	Duration	Footprint[ii] (GB)	Traffic (GB)	Requests (10^6)
FS1	File Server (NFS[iii])	HP 9000/720 (50MHz)	32	3 BSD[iii] FFS[iii] (3 GB)	3	3	45 days (4/25/92 - 6/8/92)	1.39	63	9.78
FS2[iv]	File Server (AFS[iii])	IBM RS/6000	-	23 AIX[iii] JFS[iii] (99.1GB)	99.1	17	8am – 6pm (11/6/2000)	-	1.70	-
TS1	Time-Sharing System	HP 9000/877 (64MHz)	96	12 BSD FFS (10.4GB)	10.4	8	45 days (4/18/92 - 6/1/92)	4.75	123	20
DS1	Database Server (ERP[iii])	IBM RS/6000 R30 SMP[iii] (4X 75MHz)	768	8 AIX JFS (9GB), 3 paging (1.4GB), 30 raw database partitions (42GB)	52.4	13	7 days (8/13/96 – 8/19/96)	6.52	37.7	6.64
S-Avg.[v]	-	-	299	-	18.5	8	32.3 days	4.22	74.6	12.1

[i] Sum of all the file systems and allocated volumes.
[ii] Amount of data referenced at least once (using block size of 512 bytes)
[iii] AFS – Andrew Filesystem, AIX – Advanced Interactive Executive (IBM's flavor of UNIX), BSD – Berkeley System Development Unix, ERP – Enterprise Resource Planning, FFS – Fast Filesystem, JFS – Journal Filesystem, NFS – Network Filesystem, NTFS – NT Filesystem, SMP – Symmetric Multiprocessor
[iv] Only per second I/O statistics were collected.
[v] Excluding FS2.

(b) Servers.

Table 2.1: Trace Description.

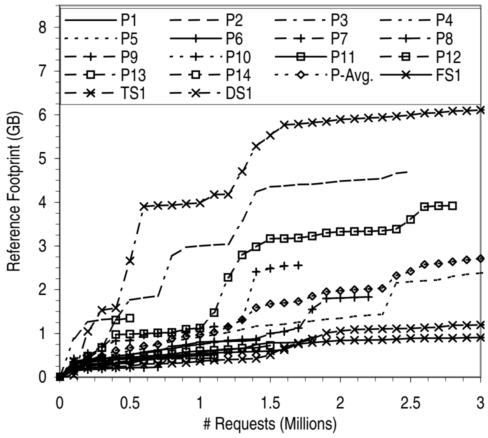


Figure 2.1: Footprint vs. Number of References.

The servers examined include two file servers, a time-sharing system and a database server. The first file server workload (FS1) was taken off a file server for nine clients at the University of California, Berkeley. This system was primarily used for compilation and editing. It is referred to as "Snake" in [RW93]. The second file server workload (FS2) was taken off an Andrew File System (AFS) server at one of the major development sites of a leading computer vendor. The system was the primary server used to support the development effort. For this system, only per-second aggregate statistics of the I/O traffic were gathered; addresses for individual I/Os were not collected. The FS2 data was therefore used for only a few of our analyses. The trace denoted TS1 was gathered on a time-sharing system at an industrial research laboratory. It was mainly used for news, mail, text editing, simulation and compilation. It is referred to as "cello" in [RW93]. The database server trace (DS1) was collected at one of the largest health insurers nationwide. The system traced was running an Enterprise Resource Planning (ERP) application on top of a commercial database. Throughout this thesis, we use the term "S-Avg." to denote the arithmetic mean of the results for FS1, TS1 and DS1.

Our traces capture the actual workloads that are presented to the storage system and are therefore likely to be sensitive to the amount of filtering by the file system cache and/or the database buffer pool. However, we believe that changing the amount of caching upstream will only affect our characterization quantitatively and that the qualitative results still apply. To show that our characterization is relatively insensitive to the amount of caching upstream, we filtered our traces through a Least-Recently-Used (LRU) write-back cache to obtain another set of traces on which to run our analysis. Following the design of most file systems, we allow a dirty block to remain in the cache for up to 30 seconds. When a block is written back, we write out, in the same operation, all the dirty blocks that are physically contiguous, up to a maximum of 512 blocks. The size of the cache is chosen to be the size of the entire main memory in the original systems (Table 2.1). These filtered traces are denoted by adding an "f" to the original designation. For instance, the trace obtained by fil-

	Number of MBs			Number of Requests		
	Read	Write	Overall	Read	Write	Overall
P1	0.575	0.176	0.441	0.618	0.575	0.605
P2	0.503	0.173	0.385	0.547	0.495	0.525
P3	0.583	0.163	0.291	0.632	0.498	0.537
P4	0.301	0.175	0.219	0.358	0.630	0.527
P5	0.369	0.232	0.275	0.438	0.620	0.574
P6	0.831	0.190	0.436	0.821	0.548	0.617
P7	0.546	0.143	0.246	0.551	0.548	0.549
P8	0.592	0.239	0.426	0.629	0.657	0.642
P9	0.484	0.146	0.317	0.488	0.471	0.479
P10	0.216	0.162	0.192	0.316	0.537	0.436
P11	0.515	0.245	0.409	0.520	0.641	0.577
P12	0.416	0.179	0.290	0.450	0.721	0.601
P13	0.557	0.257	0.391	0.585	0.615	0.603
P14	0.356	0.221	0.282	0.415	0.683	0.596
P-Avg.	0.489	0.193	0.329	0.526	0.589	0.562
FS1	0.594	0.573	0.582	0.570	0.681	0.633
TS1	0.583	0.394	0.474	0.546	0.454	0.495
DS1	0.057	0.203	0.122	0.133	0.702	0.488
S-Avg.	0.412	0.390	0.393	0.416	0.612	0.539

Table 2.2: Fraction of I/O Activity that is Filtered.

tering P1 is denoted as P1f. We further denote the arithmetic mean of the results for all the filtered PC workloads as "Pf-Avg" and that for the filtered FS1, TS1 and DS1 workloads as "Sf-Avg".

In Table 2.2, we present the faction of I/O activity that is satisfied by such an additional cache. On average, over 50% of the I/O requests are removed by the cache, which shows that the amount of caching has been significantly increased over what was in the original traced systems. Observe further that the traffic volume is reduced less significantly than the number of operations. This is because the smaller requests tend to have a higher chance of hitting in the cache. Furthermore, by delaying the writes, we are able to consolidate them into fewer but larger sequential writes. In Table 2.3 and Figure 2.2, we present the request size distribution for both the original and the filtered traces. *The average request size for the original workloads is about 7-9 KB.* The filtered traces have larger writes on average but their request size distributions track those of the original traces remarkably well. That the filtered traces maintain the qualitative behavior of the original traces is a result that we will see repeated for different characteristics in the rest of the chapter.

2.5 Intensity of I/O

We begin our characterization by focusing on the I/O intensity of the various workloads. This is akin to understanding the size of a problem so that we can better approach it. The

	All Requests				Read Requests				Write Requests			
	Avg.	Std. Dev.	Min.	Max.	Avg.	Std. Dev.	Min.	Max.	Avg.	Std. Dev.	Min.	Max.
P1	19.1	26.6	1	128	17.7	22	1	128	22.4	35.4	1	128
P2	17.2	27.4	1	1538	19.1	24.4	1	128	14.6	30.9	1	1538
P3	15.5	24.8	1	128	15.5	19.4	1	128	15.5	26.8	1	128
P4	21.7	33.8	1	128	20.4	30.3	1	128	22.5	35.8	1	128
P5	16.3	25	1	298	20.8	28.3	1	129	14.8	23.6	1	298
P6	15.7	23.7	1	128	23.1	25.5	1	128	14.7	23.2	1	128
P7	18.5	30.3	1	128	19.1	23.9	1	128	18.4	31.9	1	128
P8	17.4	25.8	1	128	16.8	20.9	1	128	18.2	30.9	1	128
P9	14.7	21.1	1	128	15.4	20.2	1	128	13.9	21.8	1	128
P10	19.6	30.7	1	128	23.7	32.8	1	128	15.7	28	1	128
P11	15.2	23.1	1	128	19.4	24.7	1	128	11.7	21.1	1	128
P12	25.3	58.6	1	512	27.5	54.6	1	512	23.6	61.4	1	512
P13	12.3	18.2	1	180	14.5	18.8	1	128	11	17.7	1	180
P14	16.1	28.1	1	1539	20.6	31.2	1	128	14	26.2	1	1539
P-Avg.	17.5	28.4	1	373	19.5	26.9	1	156	16.5	29.6	1	373
Pf-Avg.	27.4	64.3	1	512	21.3	29.3	1	155	34.1	84.2	1	512
FS1	13.5	5.08	2	512	12.5	5.47	2	64	14.2	4.65	2	512
TS1	12.9	7.77	2	512	12.4	6.52	2	224	13.3	8.62	2	512
DS1	11.9	21.9	1	512	17.4	27.1	1	512	8.55	17.3	1	256
S-Avg.	12.8	11.6	1.67	512	14.1	13.0	1.67	267	12.0	10.2	1.67	427
Sf-Avg.	16.4	29.8	1.67	512	14.0	13.5	1.67	222	18.9	41.0	1.67	512

Table 2.3: Request Size (Number of 512-Byte Blocks).

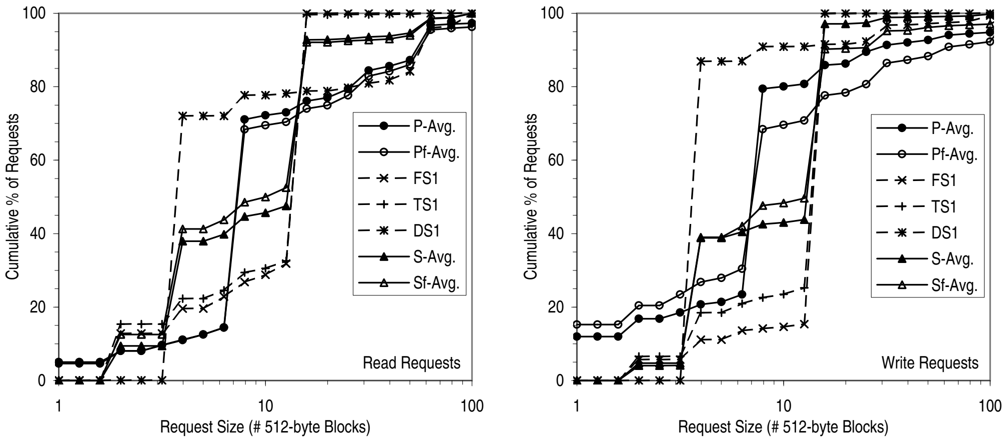


Figure 2.2: Distribution of Request Size.

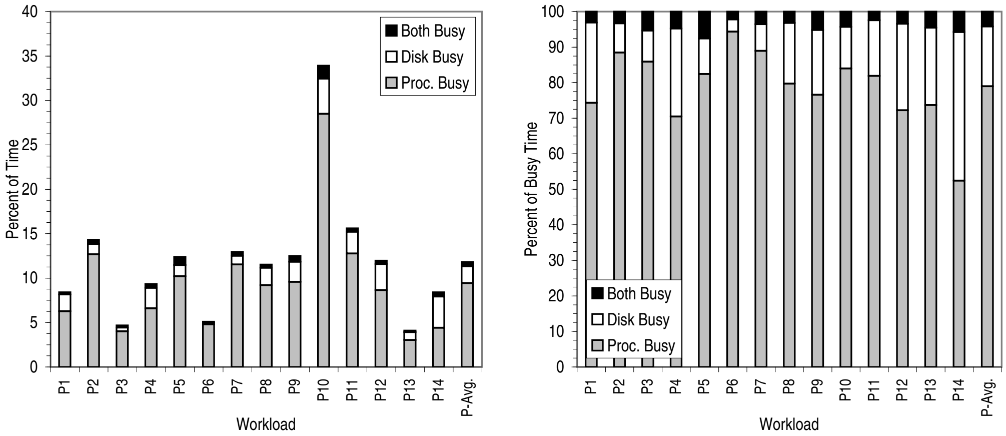


Figure 2.3: Disk and Processor Busy Time.

questions we seek to address in this section include how significant is the I/O component in the overall workload, how many I/Os are generated, and how fast do the requests arrive.

2.5.1 Overall Significance of I/O

In Figure 2.3, we present the percent of time the disk and processor are busy for the PC workloads. Similar results for the server workloads would be interesting but unfortunately, this analysis relies on information that is available only in the PC traces. Specifically, we calculate the processor busy time by looking at the thread switch records to determine when the processor is not in the idle loop. The disk busy time is taken to be the duration during which one or more of the disks in the system are servicing requests. Recall that for the PC workloads, we only have trace data for the periods during which user input activity occurs at least once every ten minutes. The results in Figure 2.3 therefore cover only the periods during which the user is actively interacting with the system.

From the figure, the processor is, on average, busy for only about 10% of the time while the disk is busy for only about 2.5% of the time. This low level of busy time is misleading, however, because the user is interested in response time; CPU idle generally represents user think time, and would occur in any case in a single user environment. We cannot, therefore, conclude that the processor and I/O system are ”fast enough”. What the results do suggest is that *there is a lot of idle time for performing background tasks, even without having to deliberately leave the computer on when the user is away*. In other words, significant resources are available even when the system is being actively used. The challenge is to harness these idle resources without affecting the foreground work. If this can be done unobtrusively, it will pave the way for sharing idle resources in collaborative computing, a paradigm commonly referred to as peer-to-peer (P2P) computing. In addition, the idle resources can be used to optimize the system so that it will perform better in future for the

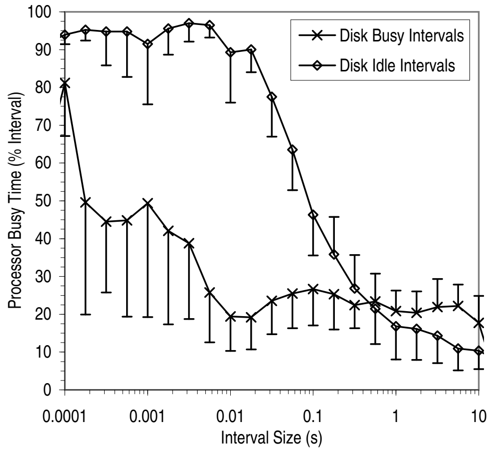


Figure 2.4: Processor Busy Time during Disk Busy/Idle Intervals. Bars indicate standard deviation (To reduce clutter, we show only the deviation in one direction).

foreground task (*e.g.,* see Chapter 4). We will characterize the disk idle periods in detail in Section 2.6.3.

I/O is known to be a major component of server workloads (*e.g.,* [RBH+95]). But if processors continue to increase in performance at the historical rate of 60% per year [HP96], as many believe they are likely to for the near future, the results in Figure 2.3 suggest that *I/O may also become the dominant component of personal computer workloads in the next few years.* More memory will of course be available in the future for caching but the PC systems in our study are already well-endowed with memory. A common way of hiding I/O latency is to overlap it with some computation either through multiprogramming or by performing I/O asynchronously. From Figure 2.3, this technique appears to be relatively ineffective for the PC workloads since *only a small fraction (20% on average) of the disk busy time is overlapped with computation*. In Figure 2.4, we compare the processor busy time during the disk idle intervals with that during the disk busy intervals. A disk idle interval refers to the time interval during which all the disks are idle. A disk busy interval is simply the period of time between two consecutive disk idle intervals. Reflecting the low average processor utilization of the workloads, the processor is busy less than 20% of the time for the long intervals ($> 0.1s$), regardless of whether any of the disks are busy. During the short intervals ($< 0.1s$), the processor is busy almost all the time when all the disks are idle but the processor utilization drops to less than 50% when one or more of the disks are busy. Such results imply that *little processing can be overlapped with I/O and that I/O response time is important for these kinds of workloads.*

That only a small amount of processing is overlapped with I/O suggests that there is effectively little multiprocessing in the PC workloads. Furthermore, as shown in Table 2.4, *I/Os, especially those in the PC workloads, tend to be synchronous.* This means that the system generally has to wait for I/Os to be completed before it can continue with subsequent processing. Observe from the table that although Windows NT provides a common

	Read	Write	Overall
P1	0.974	0.667	0.887
P2	0.970	0.627	0.825
P3	0.931	0.701	0.770
P4	0.829	0.731	0.768
P5	0.927	0.776	0.814
P6	0.967	0.849	0.864
P7	0.878	0.723	0.758
P8	0.968	0.835	0.909
P9	0.800	0.605	0.699
P10	0.763	0.749	0.756
P11	0.926	0.705	0.805
P12	0.961	0.566	0.736
P13	0.610	0.695	0.664
P14	0.733	0.714	0.720
P-Avg.	0.874	0.710	0.784
FS1	0.854	0.254	0.505
TS1	0.835	0.671	0.744
DS1	-	-	-
S-Avg.	0.845	0.462	0.624

Table 2.4: Fraction of I/O Requests that are Synchronous.

convenient interface for performing both synchronous and asynchronous I/O, on average nearly 80% of the I/O requests are flagged as synchronous. Metadata updates account for most, but not all, of the synchronous writes. Excluding metadata writes, about half of the writes are synchronous. In the FS1 and TS1 traces, some I/O requests are not explicitly flagged as synchronous or asynchronous. For these traces, we assume that I/Os are synchronous unless they are explicitly flagged otherwise. The DS1 trace does not contain information about whether the I/Os are synchronous.

A common difficulty in using trace-driven simulations to study I/O systems is to realistically account for events that occur faster or slower in the simulated system than in the original system. Since the PC workloads have little multiprocessing and most of the I/Os are synchronous, these workloads can be modeled by assuming that after completing an I/O, the system has to do some processing and the user, some "thinking", before the next set of I/Os can be issued. For instance, in the timeline in Figure 2.5, after request R0 is completed, there are delays during which the system is processing and the user is thinking before requests R1, R2 and R3 are issued. Because R1, R2 and R3 are issued after R0 has been completed, we consider them to be dependent on R0. Similarly, R4 and R5 are deemed to be dependent on R1. Presumably, if R0 is completed earlier, R1, R2 and R3 will be dragged forward and issued earlier. If this in turn causes R1 to be finished earlier, R4 and R5 will be similarly moved forward in time.

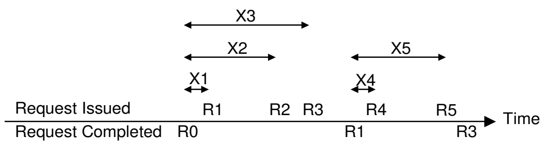


Figure 2.5: Intervals between Issuance of I/O Requests and Most Recent Request Completion.

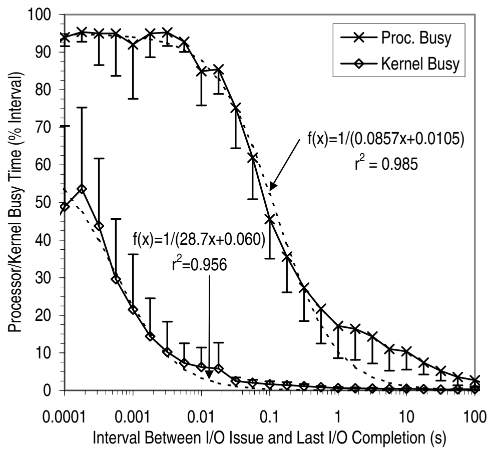


Figure 2.6: Processor/Kernel Busy Time during Intervals between Issuance of I/Os and Most Recent Request Completion. Bars indicate standard deviation (To reduce clutter, we show only the deviation in one direction).

In Figure 2.6, we plot the percent of time the processor is busy during the interval between when an I/O request is issued and the most recent completion of an I/O request (the X's in Figure 2.5). We are interested in the processor busy time during such intervals to model what happens when the processing time is reduced through faster processors. From Figure 2.6, we find that for the PC workloads, the processor utilization during the intervals between I/O issuance and the last I/O completion is related to the length of the interval by a reciprocal function of the form $f(x) = 1/(ax + b)$ where $a = 0.0857$ and $b = 0.0105$. The reciprocal function suggests that there is a fixed amount of processing per I/O. To model a processor that is n times faster than was in the traced system, we would scale only the system processing time by n, leaving the user think time unchanged. Specifically, we would replace an interval of length x by one of $x[1 - f(x) + f(x)/n]$. We believe that for the PC workloads, this is considerably more realistic than simply scaling the inter-arrival time between I/O requests by n, as is commonly done.

In Figure 2.6, we also plot the percent of time that the kernel is busy during the intervals between when an I/O request is issued and the previous I/O completion. We consider the kernel to be busy if the kernel process (process ID = 2 in Windows NT) is allocated the CPU. As shown in the figure, the kernel busy time is also related to the length of the

			P-Avg.	Pf-Avg.	FS1	TS1	DS1	S-Avg.	Sf-Avg.
# I/O Requests (10^3)	Average	Read	25	12	92.7	190	344	209	137
		Write	37	15	129	246	564	313	113
		Total	62	27	222	436	908	522	251
	Max.	Read	82	48	286	577	725	530	446
		Write	102	30	355	393	833	527	162
		Total	183	78	641	970	1558	1056	609
I/O Traffic (MB)	Average	Read	234	131	568	1152	3017	1579	1161
		Write	295	236	895	1604	2407	1635	1090
		Total	529	368	1462	2756	5425	3214	2250
	Max.	Read	973	701	1677	3613	4508	3266	2731
		Write	1084	856	2446	2573	5159	3393	2403
		Total	2057	1557	4124	6186	9667	6659	5134

Table 2.5: Daily Volume of I/O Activity in Thousands of Requests and Megabytes of Traffic. Pf-Avg. and Sf-Avg. denote the arithmetic mean of the results for the filtered PC and server workloads respectively.

interval by a reciprocal function, as we would expect when there is some fixed kernel cost per I/O.

2.5.2 Amdahl's Factor and Access Density

Table 2.5 presents the average and maximum amount of I/O traffic generated per day by the various workloads. Note that the average is taken over the days when there is some I/O activity recorded in the traces. This means that for the PC workloads, the weekends are, for the most part, ignored. We find that the maximum daily I/O traffic is about two to four times higher than the average. The server workloads are clearly more I/O intensive than the PC workloads and we expect that servers today will have even higher rates of I/O activity. Nevertheless, it should still be the case that *collecting a daily trace of the disk blocks referenced for later analysis and optimization (*e.g., *to optimize disk block placement as in Chapter 4) is very feasible*. For instance, for the database server workload, logging eight bytes of information per request will create just over 12 MB of data on the busiest day.

When designing the IBM System/360, Amdahl observed that the amount of I/O generated per instruction tends to be relatively constant [Amd70]. More specifically, Amdahls' rule of thumb states that a typical data processing system requires approximately 1 Mb/s of I/O bandwidth for every million instructions per second (MIPS) of processing power. This rule of thumb dates back to the sixties before buffering and caching techniques were widely used. It was recently revalidated for the logical I/O of database workloads in the production environments of some of the world's largest corporations [HSY01a]. Due to the advent of caching, however, Amdahl's factor for the ratio of physical I/O bandwidth to

	Avg. Number of Mbs of I/O				Avg. Number of I/Os			
	Per Second	/s /MHz	/s /MIPS	/s /GB	Per Second	/s /MHz	/s /MIPS	/s /GB
P1	0.588	0.00177	0.00177	0.0980	5.80	0.0174	0.0174	0.967
P2	0.557	0.00278	0.00278	0.116	7.25	0.0363	0.0363	1.51
P3	0.811	0.00180	0.00180	0.135	6.42	0.0143	0.0143	1.07
P4	6.84	0.0152	0.01520	1.14	61.0	0.135	0.135	10.2
P5	3.50	0.00778	0.00778	0.583	14.7	0.0326	0.0326	2.45
P6	0.106	0.000639	0.000639	0.0212	1.44	0.00866	0.00866	0.287
P7	2.84	0.0107	0.0107	0.711	28.5	0.107	0.107	7.13
P8	1.08	0.00361	0.00361	0.270	8.65	0.0288	0.0288	2.16
P9	1.11	0.00671	0.00671	0.371	15.4	0.0929	0.0929	5.14
P10	5.71	0.0215	0.0215	1.36	44.8	0.168	0.168	10.7
P11	0.852	0.00243	0.00243	0.213	10.9	0.0310	0.0310	2.72
P12	4.63	0.0116	0.0116	0.771	22.8	0.0570	0.0570	3.80
P13	0.385	0.00193	0.00193	0.0963	8.03	0.0401	0.0401	2.01
P14	4.14	0.00919	0.00919	0.517	51.8	0.115	0.115	6.47
P-Avg.	2.37	0.00697	0.00697	0.457	20.5	0.0632	0.0632	4.04
Pf-Avg.	1.92	0.00569	0.00569	0.372	9.24	0.0312	0.0312	1.94
FS1	1.26	0.0252	0.0503	0.419	26.8	0.536	1.07	8.94
TS1	1.99	0.0311	0.0621	0.191	39.0	0.610	1.22	3.75
DS1	6.11	0.0204	0.0407	0.117	72.4	0.241	0.482	1.38
S-Avg.	3.12	0.0255	0.0511	0.242	46.1	0.462	0.925	4.69
Sf-Avg.	2.98	0.0234	0.0467	0.217	29.5	0.375	0.750	3.99

Table 2.6: Intensity of I/O during the Busiest One-Hour Period.

MIPS was found to be on the order of 0.05 [HSY01a]. Since the value of Amdahl's factor determines what constitutes a balanced system, it would be useful to see if the same figure applies to the current set of workloads.

To this end, we calculated the ratio of I/O intensity, *i.e.,* the rate of I/O activity, to processor speed for our workloads. Unlike the traces used in [HSY01a] which cover only the peak periods of the workloads as identified by the system administrator, the traces in the current study span periods of days and weeks, and includes periods of light activity as well as those of heavy activity. Therefore, in calculating the I/O intensity normalized by processor speed in Table 2.6, we consider the busiest one-hour interval, which we define as the one hour interval with the highest I/O bandwidth requirement. The I/O intensity averaged over various time intervals ranging from 100 milliseconds to the trace length is presented in Table 2.7. Notice from Table 2.6 that the filtered traces have significantly fewer I/O operations during the busiest one-hour interval. However, because the request sizes for the filtered traces are much larger during this period (see Table 2.8), the bandwidth figures for the filtered traces are just slightly lower than those for the original workloads. In this section, our focus is on establishing general rules of thumb with regards to the I/O intensity of our various workloads. It turns out that the effect of filtering the workloads is not large enough to significantly affect any of our findings.

	0.1s	1s	10s	1min	10min	1hr	Trace Len.
P1	115	55.3	27.3	8.63	2.13	0.588	0.0366
P2	64.8	26.8	20.9	7.54	2.35	0.557	0.0234
P3	50.3	27.2	15.9	13.1	4.19	0.811	0.0129
P4	121	99.5	80.0	56.1	34.6	6.84	0.0893
P5	40.2	28.0	26.3	17.6	13.2	3.50	0.0674
P6	45.0	23.3	8.51	2.81	0.44	0.106	0.00549
P7	61.3	47.1	18.5	10.4	4.78	2.84	0.0463
P8	51.9	36.4	19.8	11.8	3.60	1.08	0.0204
P9	50.0	27.0	11.1	5.99	3.71	1.11	0.0306
P10	85.0	75.0	48.5	34.9	17.1	5.71	0.0358
P11	133	46.4	29.0	12.7	2.06	0.852	0.0266
P12	90.0	48.7	26.2	20.1	10.7	4.63	0.0139
P13	45.0	21.5	7.77	4.39	1.26	0.385	0.0148
P14	71.6	51.5	32.5	29.0	12.4	4.14	0.0476
P-Avg.	73.1	43.8	26.6	16.8	8.04	2.37	0.0337
Pf-Avg.	109	45	24.0	15.0	6.66	1.92	0.0237
FS1	382	41.1	26.1	11.9	2.05	1.26	0.133
TS1	264	96.3	14.9	10.8	4.88	1.99	0.260
DS1	156	108	91.9	85.1	19.7	6.11	0.515
S-Avg.	267	81.7	44.3	35.9	8.87	3.12	0.302
Sf-Avg.	262	76	42.9	32.0	7.71	2.98	0.213

Table 2.7: I/O Intensity (Mb/s) Averaged over Various Time Intervals, showing the peak or maximum value observed for each interval size.

From Table 2.6, the server workloads are fairly consistent, generating about 0.02-0.03 Mb/s of I/O for every MHz of processing power. The PC workloads are less I/O intensive, generating about 0.007 Mb/s/MHz on average. In order to determine an order of magnitude figure for the ratio of I/O bandwidth to MIPS, we need a rough estimate of the Cycles Per Instruction (CPI) for the various workloads. We use a value of one for the PC workloads because the CPI for the SPEC95 benchmark on the Intel Pentium Pro processor has been found to be between 0.5 and 1.5 [BD97]. For the server workloads, we use a CPI value of two in view of results in [ADHW99, KPH$^+$98a]. Based on these estimates of the CPI, we find that the *server workloads generate around 0.05 bits of real I/O per instruction*, which is consistent with the estimated Amdahl's factor for the production database workloads in [HSY01a]. *The figure for the PC workloads is seven times lower at about 0.007 bits of I/O per instruction.*

Interestingly, surveys conducted between 1980 and 1993 of large data processing mainframe installations found that the number of physical I/Os per second per MIPS was decreasing by just over 10% per year to 9.0 in 1993 [McN95]. This figure is about ten times higher than what we are seeing for our server workloads. A possible explanation for this large discrepancy is that the mainframe workloads issue many small I/Os but data reported in [McN95] show that the average I/O request size for the surveyed mainframe installations was about 9 KB, which is slightly larger than the 8 KB for our server workloads (Table 2.8).

	All Requests				Read Requests				Write Requests			
	Avg.	Std. Dev.	Min.	Max.	Avg.	Std. Dev.	Min.	Max.	Avg.	Std. Dev.	Min.	Max.
P1	26	32.9	1	128	20.4	22.9	1	128	42.3	48.5	1	128
P2	19.7	30	1	1536	16.5	20.7	1	128	44.3	63.1	1	1536
P3	32.3	43.5	1	128	18.6	28.5	1	128	38.9	47.7	1	128
P4	28.7	40.2	1	128	15.5	21.1	1	128	29.8	41.2	1	128
P5	61	58.9	1	129	96.4	53.1	1	129	13	18.7	1	128
P6	18.9	29.4	1	128	27.6	29	1	128	16.8	29.1	1	128
P7	25.5	36.1	1	128	21.9	25	1	128	27.4	40.5	1	128
P8	32	42.6	1	128	21.2	31.5	1	128	55.3	52.9	1	128
P9	18.5	27.7	1	128	19.3	27.9	1	128	16	26.7	1	128
P10	32.6	44.4	1	128	37	46.4	1	128	23.1	37.8	1	128
P11	20.1	29	1	128	21.6	27.3	1	128	17.4	31.5	1	128
P12	51.9	120	1	512	96.4	144	1	512	38.2	107	1	512
P13	12.3	18.8	1	128	14.6	20.9	1	128	10.5	16.7	1	128
P14	20.5	38.6	1	128	13.7	29.1	1	128	72	58.3	1	128
P-Avg.	28.6	42.3	1	256	31.5	37.7	1	156	31.8	44.3	1	256
Pf-Avg.	55.5	93.3	1	512	34.2	38.2	1	155	91	141	1	512
FS1	12	5.52	2	18	11.6	5.61	2	18	14.4	4.15	2	16
TS1	13	9.87	2	512	12.6	5.52	2	64	14.9	19.9	2	512
DS1	21.6	35.3	1	128	25.4	38.6	1	108	19	32.5	1	128
S-Avg.	15.5	16.9	1.67	219	16.5	16.6	1.67	63.3	16.1	18.9	1.67	219
Sf-Avg.	25.8	12.7	2.00	213	27	8.73	2.00	51.3	13.1	13.4	2.67	213

Table 2.8: Request Size (Number of 512-Byte Blocks) during the Busiest One-Hour Period.

Of course, mainframe MIPS and Reduced Instruction Set Computer (RISC) MIPS are not directly comparable and this difference could account for some of the disparity, as could intrinsic differences between the workloads. In addition, our calculations are based on the MIPS rating of the system, which is what we have available to us. The mainframe surveys, on the other hand, used utilized MIPS [Maj81] or the processing power actually consumed by the workload. To make our calculations consistent with the survey results, we could factor in the processor utilization when the workload is running. For instance, if the processor utilization is 10%, as suggested by our earlier results for the PC workloads, we would multiple our figures by 10. With this adjustment, the PC workloads still generate less than one I/O per second per MIPS. The server traces, unfortunately, do not contain information from which we can derive the processor utilization for these workloads. But we expect the processor in these workloads to be also less than fully utilized so that the number of I/Os generated per second per MIPS by these workloads is actually closer to the survey results than the raw numbers suggest.

Another useful way of looking at I/O intensity is with respect to the storage used (Table 2.1). In this thesis, the storage used by each of the workloads is estimated to be the combined size of all the file systems and logical volumes defined in that workload. This makes our calculations comparable to historical data and is a reasonable assumption unless storage can be allocated only when written to, for instance by using storage virtualization

	Bandwidth (Mb/s)	Processing Power (GHz)				Storage (GB)			
		P-Avg.	Pf-Avg.	S-Avg.	Sf-Avg.	P-Avg.	Pf-Avg.	S-Avg.	Sf-Avg.
Ethernet	10	0.718	0.879	0.196	0.214	10.9	13.4	20.6	23.0
Fast Ethernet	100	7.18	8.79	1.96	2.14	109	134	206	230
Gigabit Ethernet	1000	71.8	87.9	19.6	21.4	1093	1344	2063	2302
Ultra ATA-100	800	57.4	70.3	15.7	17.1	875	1075	1650	1842
Serial ATA	1200	86.1	105	23.5	25.7	1312	1613	2475	2763
UltraSCSI 320	2560	184	225	50.1	54.8	2799	3441	5281	5894
Fiber Channel	1000	71.8	87.9	19.6	21.4	1093	1344	2063	2302
Infiniband	2500	179	220	49.0	53.5	2733	3360	5157	5756

Table 2.9: Projected Processing Power and Storage Needed to Drive Various Types of I/O Interconnect to 50% Utilization.

software that separates the system view of storage from the actual physical storage. Table 2.6 summarizes, for our various workloads, the number of I/Os per second per GB of storage used. This metric is commonly referred to as access density and is widely used in commercial data processing environments [McN95]. The survey of large data processing mainframe installations cited above found the access density to be decreasing by about 10% per year to 2.1 I/Os per second per GB of storage in 1993. Notice from Table 2.6 that the access density for DS1 appears to be consistent with the mainframe survey results. However, the access density for FS1 and TS1 is about two to four times higher. The PC workloads have, on average, an access density of 4 I/Os per second per GB of storage, which is on the order of the figure for the server workloads even though the server workloads are several years older. Such results suggest that *PC workloads may be comparable to server workloads in terms of access density. Note, however, that as disks become a lot bigger and PCs have at least one disk, the density of access with respect to the available storage is likely to be much lower for PC workloads.*

Table 2.6 also contains results for the number of bits of I/O per second per GB of storage used. The PC workloads have, on average, 0.46 Mb of I/O per GB of storage. By this measure, the server workloads are less I/O intense with an average of only 0.24 Mb of I/O per GB of storage. Based on these results, we project the amount of processing power and storage space that will be needed to drive various types of I/O interconnect to 50% utilization. The results are summarized in Table 2.9. Note that all the modern I/O interconnects offer Gb/s bandwidth. Some of them, specifically Ethernet and Fiber Channel, have newer versions with even higher data rates. For the kinds of workloads that we have, the I/O interconnect is not expected to be a bottleneck any time soon. However, we would expect to see much higher bandwidth requirements for workloads that are dominated by large sequential I/Os (*e.g.,* scientific and decision support workloads). In such environments, and especially when many workloads are consolidated into a large server and many disks are consolidated into a sizeable outboard controller, the bandwidth requirements have to be carefully evaluated to ensure that the network or connection between the disks and the host does not become the bottleneck.

Inter-Arrival Time (s)	P-Avg.	Pf-Avg.	FS1	TS1	DS1	S-Avg.	Sf-Avg.
1st Moment	3.25	7.23	0.398	0.194	0.0903	0.227	0.561
2nd Moment	7.79E+05	1.86E+06	368	23.1	2.00	131	363
3rd Moment	6.46E+11	1.60E+12	2.02E+07	8.09E+03	67.4	6.74E+06	1.88E+07

Table 2.10: First, Second and Third Moments of the I/O Inter-Arrival Time.

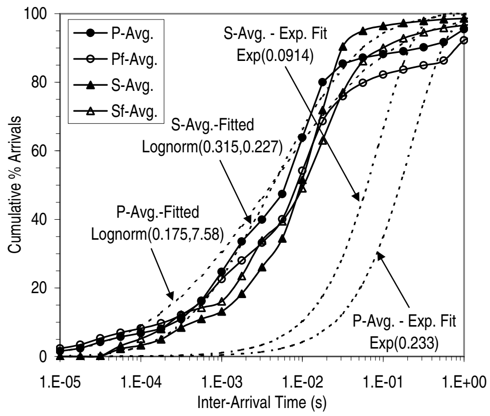


Figure 2.7: Distribution of I/O Inter-Arrival Time.

2.5.3 Request Arrival Rate

In Table 2.10, we present the first, second and third moments of the distribution of I/O inter-arrival time. The distribution is plotted in Figure 2.7. Since this distribution is often needed in modeling I/O systems, we fitted standard probability distributions to it. As shown in the figure, the commonly used exponential distribution, while easy to work with mathematically, turns out to be a rather poor fit for all the workloads. Instead, *the lognormal distribution (denoted $Lognorm(\mu, \sigma)$ where μ and σ are respectively the mean and standard deviation) is a reasonably good fit*. Recall that a random variable is lognormally distributed if the logarithm of the random variable is normally distributed. Therefore, the lognormal distribution is skewed to the right or towards the larger values, meaning that there exists long intervals with no I/O arrivals. The long tail of the inter-arrival distribution could be a manifestation of different underlying behavior such as correlated arrival times but regardless of the cause, the net effect is that *I/O requests seldom occur singly but tend to arrive in groups* because if there are long intervals with no arrivals, there must be intervals that have far more arrivals than their even share. We will analyze the burstiness of the I/O traffic in greater detail in the next section.

Another interesting way to analyze the arrival process of I/O requests is relative to the completion of preceding requests. In particular, if the workload supports multiple outstanding I/O requests, there will be more potential for improving the average I/O performance, for instance, through request scheduling. Figure 2.8 presents the distribution of queue

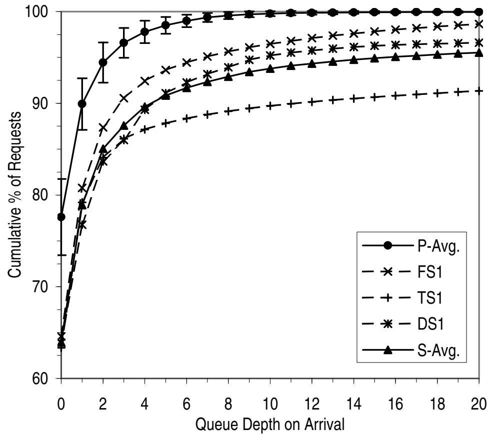


Figure 2.8: Distribution of Queue Depth on Arrival. Bars indicate standard deviation.

depth, which we define to be the length of the request queue as seen by an arriving request. In Table 2.11 and Figure A.2 in Appendix A, we break down the outstanding requests into reads and writes. Note that we consider a request to be in the queue while it is being serviced.

We find that across all the workloads, the read queue tends to be shallow – more than 85% of the requests arrive to find the queue devoid of read requests, and the average number of reads outstanding is only about 0.2. Nevertheless, the read queue can be deep at times. If there are read requests in the queue, the average number of them is almost 2 (denoted $Avg.| > 0$ in Table 2.11). In addition, the maximum read queue depth can be more than 90 times higher than the average. Notice that *the server workloads do not appear to have a deeper read queue than the personal system workloads. This finding suggests that read performance in personal system workloads could benefit as much from request scheduling as in server workloads.* We will examine request scheduling in detail in Chapter 3. Observe further from Table 2.11 that *the write queue is markedly deeper than the read queue for all the workloads*, as we would expect given that a greater fraction of writes are asynchronous compared to reads (Table 2.4). The PC workloads appear to have a significantly shallower write queue than the server workloads.

Note that we are looking at the number of outstanding requests from the perspective of the operating system layer at which the trace data were collected. This reflects the potential for request scheduling at any of the levels below, and not just at the physical storage system, which is typically not handed hundreds of requests at a time. Some of the differences among the workloads could be the result of collecting the traces at different levels on the different platforms. In general, the operating system and/or the disk device driver will queue up the requests and attempt to schedule them based on some simple performance model of the storage system (*e.g.,* minimize seek distance). There is a tendency for the operating system and/or device driver to hold back the requests and issue only a small number of them at any one time so as to avoid overloading the storage system. In reality,

	# I/Os Outstanding					# Reads Outstanding					# Writes Outstanding				
	Avg.	Avg.|>0	Std. Dev.	90%-tile	Max.	Avg.	Avg.|>0	Std. Dev.	90%-tile	Max.	Avg.	Avg.|>0	Std. Dev.	90%-tile	Max.
P1	0.377	1.73	0.937	1	24	0.273	1.66	0.802	1	23	0.104	1.36	0.407	0	10
P2	0.421	1.9	1.01	2	13	0.28	1.93	0.864	1	12	0.141	1.45	0.473	0	6
P3	0.553	2.52	1.51	2	20	0.177	2.34	0.856	0	14	0.376	2.41	1.23	1	20
P4	0.796	2.67	1.96	3	74	0.332	2.15	1.1	1	27	0.464	2.33	1.55	1	74
P5	0.304	1.92	0.958	1	22	0.0985	1.97	0.601	0	20	0.206	1.71	0.704	1	22
P6	0.27	1.52	0.684	1	10	0.0169	1.36	0.181	0	8	0.253	1.52	0.66	1	8
P7	0.47	2.09	1.26	2	55	0.139	1.92	0.766	0	54	0.331	1.91	0.967	1	22
P8	0.365	1.96	1.07	1	26	0.196	1.65	0.699	1	14	0.168	1.82	0.673	0	16
P9	0.718	2.77	2.41	2	73	0.233	1.72	0.837	1	24	0.484	3.3	2.27	1	73
P10	0.573	2.33	1.81	2	60	0.252	1.62	0.766	1	19	0.321	2.53	1.62	1	60
P11	0.454	2.22	1.29	1	37	0.251	2.06	0.948	1	17	0.204	1.73	0.728	1	35
P12	0.341	1.99	1.06	1	19	0.201	2.37	0.897	0	17	0.14	1.35	0.464	1	8
P13	0.664	2.26	1.47	2	24	0.393	2.33	1.17	1	17	0.272	1.7	0.859	1	24
P14	0.541	2.11	1.28	2	23	0.184	1.62	0.677	1	17	0.358	1.98	1.05	1	23
P-Avg.	0.489	2.14	1.34	1.64	34.3	0.216	1.91	0.797	0.643	20.2	0.273	1.94	0.975	0.786	28.6
FS1	1.49	4.19	4.62	3	181	0.186	1.38	0.538	1	13	1.3	4.74	4.56	3	181
TS1	9.98	27.2	41.1	12	1530	0.214	1.42	0.574	1	20	9.76	36.4	41.1	11	1530
DS1	3.13	8.68	15.9	5	257	0.203	1.95	0.904	1	9	2.93	8.93	15.7	5	256
S-Avg.	4.87	13.4	20.5	6.67	656	0.201	1.58	0.672	1	14	4.66	16.7	20.5	6.33	656

Table 2.11: Queue Depth on Arrival.

modern storage systems, specifically modern disks, have the ability to do more elaborate and effective [WGP94] request scheduling based on whether a request will hit in the disk cache, and on the seek and rotational positions.

2.6 Variability in I/O Traffic over Time

When I/O traffic is smooth and uniform over time, system resources can be very efficiently utilized. However, when the I/O traffic is bursty as is the case in practice (Section 2.5.3), resources have to be provisioned to handle the bursts so that during the periods when the system is relatively idle, these resources will be wasted. There are several approaches to try to even out the load. The first is to aggregate multiple workloads in the hope that the peak and idle periods in the different workloads will tend to cancel out one another. This idea is one of the premises of the storage utilities model. Whether the aggregation of multiple workloads achieves the desired effect of smoothening the load depends on whether the workloads are dependent or correlated. We will examine the dependence among our workloads in Section 2.6.1.

The second approach to smoothening the traffic is to try to shift the load temporally. For instance, by deferring or offloading some work from the busy periods to the relative lulls (*e.g.,* write buffering and logging disk arrays [CHY00, SHCG94]) or by eagerly or

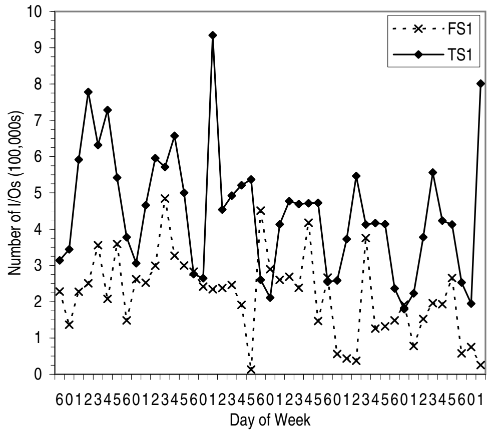


Figure 2.9: Daily Volume of I/O Activity.

speculatively performing some work in the hope that such work will help improve performance during the next busy period (*e.g.,* prefetching and reorganizing data based on access patterns as suggested in Chapter 4). The effectiveness of these attempts at time-shifting the load to even out the traffic depends on the extent to which the traffic is autocorrelated. We will analyze the autocorrelation of I/O traffic to determine whether they are long-range dependent or self-similar in Section 2.6.2. In Section 2.6.3, we characterize in detail the idle periods to help in the design of techniques that try to exploit idle resources.

2.6.1 Dependence among Workloads

In general, two processes are said to be dependent or correlated if the value a process takes on constrains the possible values that the other process can assume. In the current context, the process is the discretized time series of the I/O traffic generated by a given workload. For example, in Figure 2.9, we plot the daily volume of I/O activity for FS1 and TS1 as a function of the day of week (0 = Sunday). If the two workloads are positively correlated, we should see the peaks in the two workloads appearing on the same day so that if the two workloads are aggregated, the resulting workload will have higher peaks. If the workloads are negatively correlated, the peaks of one will occur when the other workload is relatively idle. If the workloads are independent, there should be no relation between the volume of activity for the two workloads. When many independent workloads are aggregated, the resulting traffic will tend to be smooth.

To more formally characterize the dependence among the workloads, we calculate the cross-correlation. The cross correlation between two processes P(i) and Q(i) where i=0,1,2...n-1 is defined as

$$r_{PQ} = \frac{\sum_i (P(i) - \overline{P})(Q(i) - \overline{Q})}{\sqrt{\sum_i (P(i) - \overline{P})^2}\sqrt{\sum_i (Q(i) - \overline{Q})^2}} \tag{2.1}$$

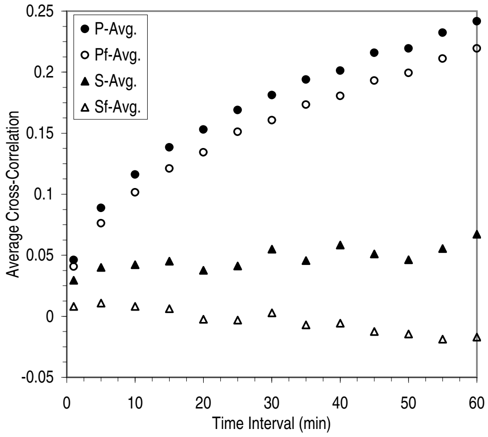


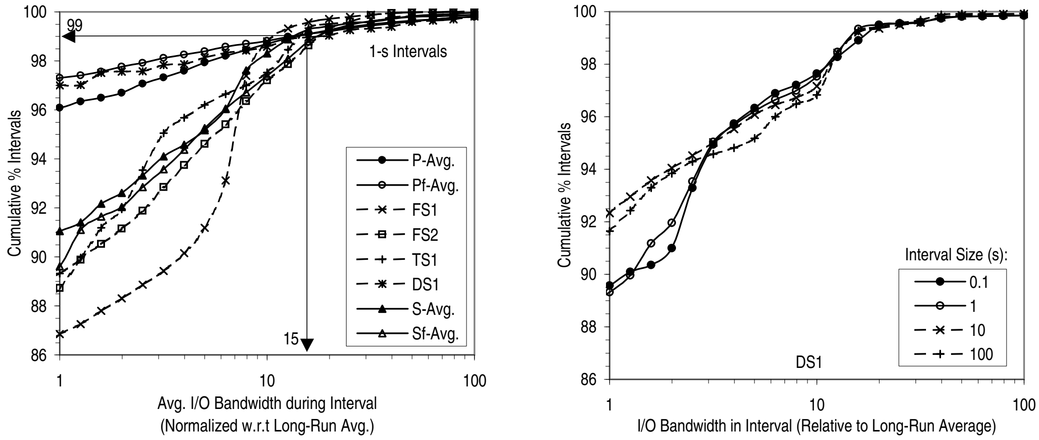
Figure 2.10: Cross-Correlation of Volume of I/O Activity vs. Time Interval Used to Aggregate Volume.

The possible values of r_{PQ} range from -1 to 1 with -1 indicating perfect negative correlation between the two processes, 0 indicating no correlation, and 1 indicating perfect positive correlation. For each workload, we consider the I/O arrival process aggregated over fixed intervals that range from one minute to a day. We synchronize the processes by the time of day and the day of week. The results are available in Tables A.1 to A.4 in Appendix A.

To summarize the dependence among a set of workloads W, we introduce the average cross-correlation which is defined as $\overline{r_{PQ}}$ where $P \in W$, $Q \in W$ and $P \neq Q$. Figure 2.10 plots the average cross-correlation for the PC workloads as a function of the time interval used to aggregate the arrival process. The same figure also plots the average cross-correlation among the server workloads. We find that, in general, *there is little cross-correlation among the server workloads, suggesting that aggregating them will likely help to smooth out the traffic and enable more efficient utilization of resources.* Our PC workloads are taken mostly from office environments with flexible working hours. Nevertheless *the cross-correlation among the PC workloads is still significant except at small time intervals. This suggests that multiplexing the PC workloads will smooth out the high frequency fluctuations in I/O traffic but some of the time-of-day effects will remain unless the PCs are geographically distributed in different time zones.* Note that the filtered workloads tend to be less correlated but the difference is small.

2.6.2 Self-Similarity in I/O Traffic

In many situations, especially when outsourcing storage, we need rules of thumb to estimate the I/O bandwidth requirement of a workload without having to analyze the workload in detail. In Section 2.5.2, we computed the access density and found that the server workloads average about 5 I/Os or about 30 KB worth of I/O per second per GB of data. This result can be used to provide a baseline estimate for the I/O bandwidth required by



(a) (b)

Figure 2.11: Distribution of I/O Traffic Averaged over Various Time Intervals.

a workload given the amount of storage it uses. To account for the variability in the I/O traffic, Figure 2.11(a) plots the distribution of I/O traffic averaged over one-second intervals and normalized to the average bandwidth over the entire trace. The figure shows that *to satisfy the bandwidth requirement for 99% of the 1-second intervals, we would need to provision for about 15 times the long-run average bandwidth.* Notice that for all the workloads, there is an abrupt knee in the plots just beyond 99% of the intervals. This means that *to satisfy requirements beyond 99% of the time would require disproportionately more resources.*

In analyzing the data, we notice that for many of the workloads, the distribution of I/O traffic is relatively insensitive to the size of the interval over which the traffic is averaged. For instance, in Figure 2.11(b), the distributions for time intervals of 0.1s, 1s, 10s, 100s for the database server DS1 are very similar. This scale-invariant characteristic is apparent in Figure A.4 on page 184, which shows the traffic variation over time for different time scales for TS1 and DS1. The topmost plot shows the throughput averaged over time intervals of 0.3s. In the second plot, we zoom out by a factor of ten so that each data point is the average traffic volume over a three-second interval. The third plot zooms out further by a factor of ten. Observe that rescaling the time series does not smooth out the burstiness. Instead the three plots look similar. It turns out that for these workloads, such plots look similar for time scales ranging from tens of milliseconds to tens of seconds.

Definition of Self-Similarity

The phenomenon where a certain property of an object is preserved with respect to scaling in space and/or time is described by self-similarity and fractals [Man82]. Let X be the incremental process of a process Y, *i.e.,* $X(i) = Y(i+1) - Y(i)$. In this context, Y counts the number of I/O arrivals and $X(i)$ is the number of I/O arrivals during the ith time interval. Y is said to be self-similar with parameter H if for all integers m,

$$X = m^{1-H} X^{(m)} \tag{2.2}$$

where

$$X^{(m)}(k) = (1/m) \sum_{i=(k-1)m+1}^{km} X(i),\ k = 1, 2, ...$$

is the aggregated sequence obtained by dividing the original series into blocks of size m and averaging over each block, and k is the index that labels each block. In this chapter, we focus on second-order self-similarity, which means that $m^{1-H}X^{(m)}$ has the same variance and autocorrelation as X. The interested reader is referred to [Ber94] for a more detailed treatment.

The single parameter H expresses the degree of self-similarity and is known as the Hurst parameter. For smooth Poisson traffic, the H value is 0.5. For self-similar series, $0.5 < H < 1$, and as $H \to 1$, the degree of self-similarity increases. Mathematically, self-similarity is manifested in several equivalent ways and different methods that examine specific indications of self-similarity are used to estimate the Hurst parameter. Many of the statistical methods used to estimate the Hurst parameter assume that the arrival process is stationary. In order to avoid potential non-stationarity, we selected two one-hour periods from each trace. The first period is chosen to be a high-traffic period, specifically one that contains more I/O traffic than 95% of other one-hour periods in the trace. The second period is meant to reflect a low traffic situation and is chosen to be one that contains more I/O traffic than 30% of other one-hour periods in the trace.

The interested reader is referred to Appendix B for details about how we estimate the degree of self-similarity for our various workloads. Here, we simply summarize the Hurst parameter values we obtain (Table 2.12) and state the finding that *for time scales ranging from tens of milliseconds to tens and sometimes even hundreds of seconds, the I/O traffic is well-represented by a self-similar process.* Note that filtering the workloads does not affect the self-similar nature of their I/O traffic.

Implications of Self-Similar I/O Traffic

That the I/O traffic is self-similar implies that burstiness exists over a wide range of time scales and that attempts at evening out the traffic temporally will tend to not remove all the variability. More specifically, the I/O system may experience concentrated periods of congestion with associated increase in queuing time. Furthermore, resource (*e.g.,* buffer, channel) requirements may skyrocket at much lower levels of utilization than expected with the commonly assumed Poisson model in which arrivals are mutually independent and are separated by exponentially distributed intervals. This behavior should be considered when

	P-Avg.	Pf-Avg.	FS1	FS2	TS1	DS1	S-Avg.	Sf-Avg.
H	0.81	0.79	0.88	0.92	0.91	0.91	0.90	0.80
μ (KB/s)	188	91.6	108	229	227	1000	445	367
σ^2 (KB/s)^2	769080	528538	122544	345964	502879	1256360	627261	528439

Table 2.12: Hurst Parameter, Mean and Variance of the Per-Second Traffic Arrival Rate during the High-Traffic Period.

designing storage systems, especially when multiple workloads are to be isolated so that they can coexist peacefully in the same storage system, as is required in many storage utilities. Such burstiness should also be accounted for in the service level agreements (SLAs) when outsourcing storage.

More generally, I/O traffic has been known to be bursty but describing this variability has been difficult. The concept of self-similarity provides us with a succinct way to characterize the burstiness of the traffic. We recommend that *I/O traffic be characterized by a three-tuple consisting of the mean and variance of the arrival rate and some measure of the self-similarity of the traffic such as the Hurst parameter.* The first two parameters can be easily understood and measured. The third is more involved but can still be visually explained. Table 2.12 summarizes these parameter values for our various workloads.

It turns out that self-similar behavior is not limited to I/O traffic or to our workloads. Recently, file system activities [GMR$^+$98] and I/O traffic [GS99] have been found to exhibit scale-invariant burstiness. Local and wide-area network traffic may also be more accurately modeled using statistically self-similar processes than the Poisson model (*e.g.,* [LTWW94]). However, analytical modeling with self-similar inputs has not been well developed yet. (See [PW00] for some recent results on modeling network traffic with self-similar processes). This, coupled with the complexity of storage systems today, means that the effect of self-similar I/O traffic has to be analyzed, for the most part, through simulations. In Appendix B, we present a method that uses the parameters in Table 2.12 to generate self-similar traffic for such simulations.

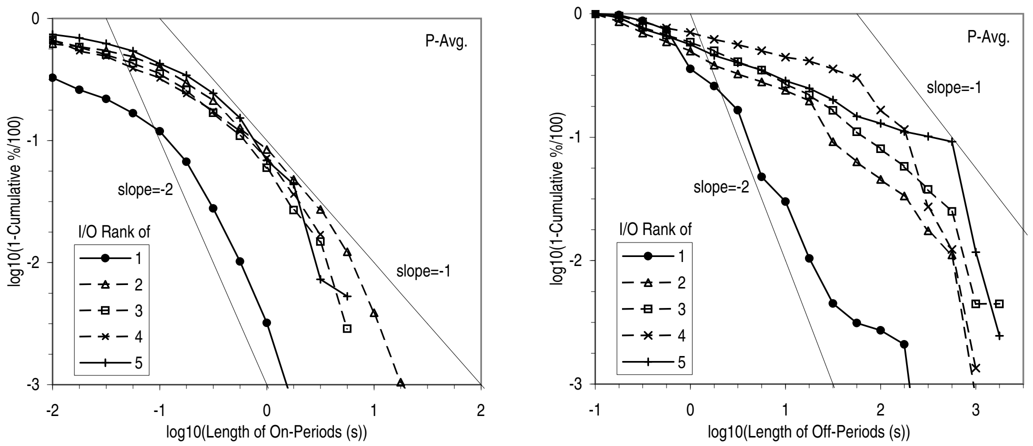
Underpinnings of Self-Similar I/O Traffic

We have seen that the I/O traffic in our workloads is self-similar but self-similarity is a rather abstract concept. To present a more compelling case and provide further insights into the dynamic nature of the traffic, we relate this phenomenon to some underlying physical cause, namely the superposition of I/O from multiple processes in the system where each process behaves as an independent source of I/O with on periods that are heavy-tailed.

A random variable, X, is said to follow a heavy-tailed distribution if

$$P(X > x) \sim cx^{-\alpha}, as\ x \to \infty, c > 0, 1 < \alpha < 2. \tag{2.3}$$

Such a random variable can give rise to extremely large values with non-negligible probability. The superposition of a large number of independent traffic sources with on and/or off



(a) On-Periods.

(b) Off-Periods.

Figure 2.12: Length of On/Off Periods for the Five Most I/O-Active Processes.

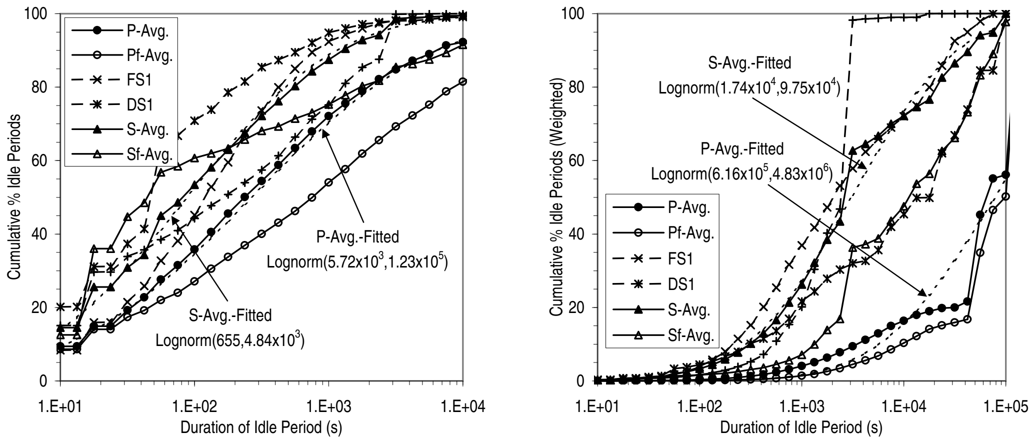
periods that are heavy-tailed is known to result in traffic that is self-similar[1] [WTSW97]. In the current context, we consider each process in the system as an independent source of I/O. As in [GS99], we define an off period for a process as any interval longer than 0.2s during which the process does not generate any I/O. All other intervals are considered to be on periods for the process. This analysis has been shown to be relatively insensitive to the threshold value used to distinguish the on and off periods [WTSW97].

Taking logarithm on both sides of Equation 2.3, we get

$$log(P(X > x)) \sim log(c) - \alpha log(x), as\ x \rightarrow \infty. \tag{2.4}$$

Therefore, if X is heavy-tailed, the plot of $P(X > x)$ versus x on log-log scale should yield a straight line with slope α for large values of x. Such log-log plots are known as complementary cumulative distribution plots or "qq-plots" [KR96]. In Figure 2.12, we present the qq-plots for the lengths of the on and off periods for the five processes that generate the most I/O traffic in each of our PC workloads. Unfortunately, none of our other workloads contain the process information that is needed for this analysis. As shown in the figure, the on periods appear to be heavy-tailed but not the off periods. This is consistent with results reported in [GS99] where the lack of heavy-tailed behavior for the off periods is attributed to periodic activity such as the sync daemon traffic. Having heavy-tailed on periods is sufficient, however, to result in self-similar aggregate traffic.

[1]Not Poisson; assumptions of Palm-Khintchine theorem are not satisfied.



(a) (b)

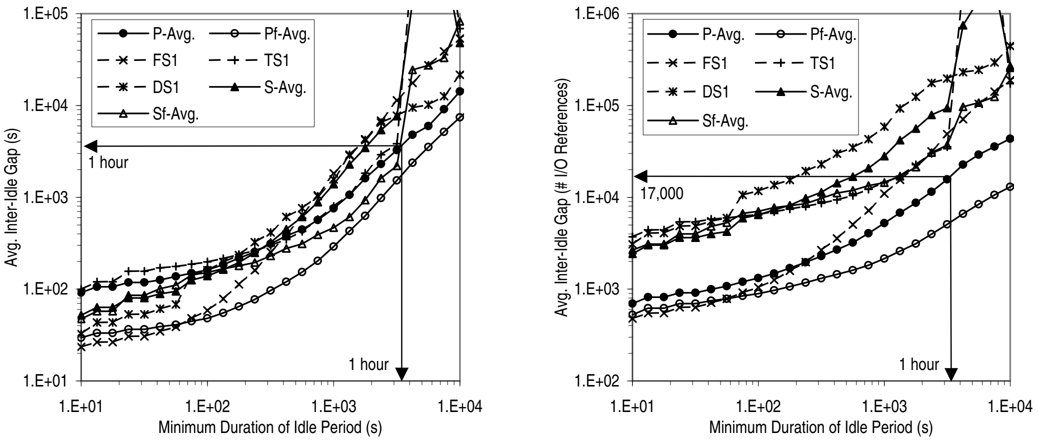
Figure 2.13: Distribution of Idle Period Duration. For the weighted distribution in (b), an idle period of duration s is counted s times, *i.e.,* it is the distribution of idle time.

2.6.3 The Relative Lulls

As discussed earlier, when the I/O load is not constant but varies over time, there may be opportunities to use the relatively idle periods to do some useful work. The reader is referred to [GBS$^+$95] for an overview of idle-time processing and a general taxonomy of idle-time detection and prediction algorithms. Here, we characterize in detail the idle periods, focusing on specific metrics that will be helpful in designing techniques that try to exploit idle time.

We consider an interval to be idle if the average number of I/Os per second during the interval is less than some value k. The term idle period refers to a sequence of intervals that are idle. The duration of an idle period is simply the product of the number of idle intervals it contains and the interval size. In this study, we use a relatively long interval of 10 seconds because we are interested in long idle periods during which we can perform a substantial amount of work. Note that storage systems tend to have some periodic background activity so that treating an interval to be idle only if it contains absolutely no I/O activity would be far too conservative. Since disks today are capable of supporting in excess of 100 I/Os per second, we select k to be 20 for all our workloads except DS1. DS1 contains several times the allocated storage in the other workloads so its storage system will presumably be much more powerful. Therefore, we use a k value of 40 for DS1.

Based on this definition of an idle interval, we find that for the PC workloads, more than 99% of the intervals are idle. The corresponding figure for the server workloads on average is more than 93%. Such results indicate that *there are clearly a lot of idle resources in the storage system that can potentially be put to good use.* Figure 2.13 presents the distribution of idle period duration for our workloads. We fitted standard probability



(a) Measured in seconds.

(b) Measured in Number of I/O References.

Figure 2.14: Average Duration of Busy Periods.

distributions to the data and found that the lognormal distribution is a reasonably good fit for most of the workloads. Notice that although most of the idle periods are short (less than a thousand seconds), long idle periods account for most of the idle time. This is consistent with previous results [GBS+95] and implies that *a system that exploits idle time can get most of the potential benefit by simply focusing on the long idle periods*.

Inter-idle Gap

An important consideration in utilizing idle resources is the frequency with which suitably long idle periods can be expected. In addition, the amount of activity that occurs between such long idle periods determines the effectiveness and the feasibility of exploiting the idle periods. For instance, a log-structured file system or array [Men95, RO92] where garbage collection is performed periodically during system idle time may run out of free space if there is a lot of write activity between the idle periods. In the disk block reorganization scheme proposed in Chapter 4, the inter-idle gap, *i.e.,* the time span between suitably long idle periods, determines the amount of trace data that has to be accumulated on the disk.

In Figure 2.14, we consider this issue by plotting the average inter-idle gap as a function of the duration of the idle period. The results show that for the PC workloads on average, idle periods lasting at least an hour are separated by busy periods of about an hour and with just over 17,000 references. As we would expect, the server workloads have longer busy periods separated by shorter idle periods. But in both environments, the results indicate that *there are long idle periods that occur frequently enough to be interesting for offline optimizations such as disk block reorganization.* In the server environments, we may have

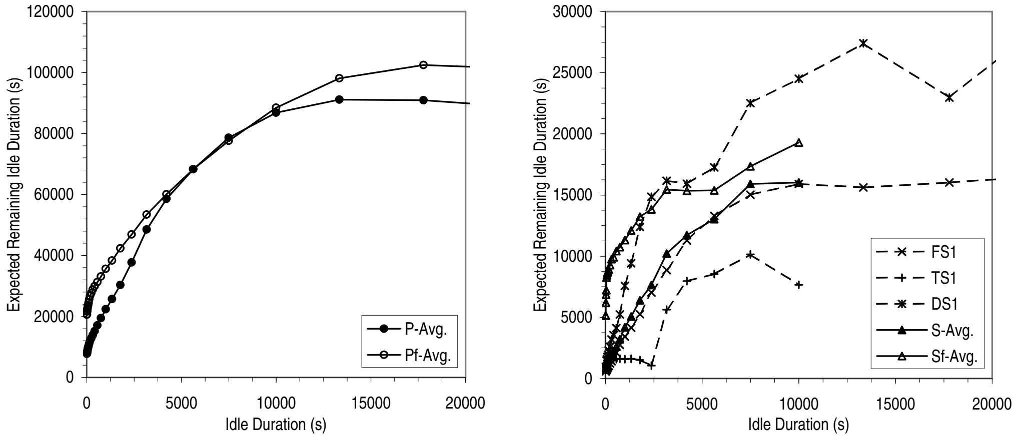


Figure 2.15: Remaining Idle Duration.

to be more meticulous about using the idle time by, for instance, dividing an offline task into several finer-grained steps that can be scheduled whenever there is a short idle period.

Idle Length Prediction

In some cases, there is a recovery cost associated with stopping an offline task before it is completed. Therefore, it is important to predict how long an idle period will last so that the system can decide whether a task should be initiated. In Figure A.3 on page 184, we plot the autocorrelation of the sequence of idle period duration at different lags. For all our workloads, there is little correlation between the length of one idle period and the lengths of the immediately preceding periods. In other words, *how long the system will remain idle is not predictable from the lengths of its recent idle periods.* This is in stark contrast to the strong correlation that has previously been observed for a personal Unix workstation [GBS$^+$95]. In that study, the idle period was taken to be an interval during which there was no I/O activity. We conjecture that because the personal UNIX workstation in the previous study was not heavily used, the idle periods are determined primarily by the periodic background activity that exists in the system, hence the strong autocorrelation.

In Figure 2.15, we plot the expected future idle duration, $E[I(x)]$, which is defined as the expected remaining idle duration given that the system has already been idle for x units of time. More formally,

$$E[I(x)] = \sum_{i=x+1}^{\infty} \frac{(i-x)l(i)}{1-L(i)} \tag{2.5}$$

where $l(\cdot)$ is the probability distribution of the idle period duration, *i.e.,* $l(j)$ is the probability that an idle period has a duration of j and $L(\cdot)$ is the cumulative probability distribution of the idle period duration, *i.e.,* $L(j) = \sum_{i=1}^{j} l(i)$. Observe from Figure 2.15 that $E[I(x)]$ is generally increasing. In other words, the longer the system has been idle, the longer it is likely to remain idle. This phenomenon suggests prediction policies that progressively raise the predicted remaining idle duration as the system remains idle.

To better understand how such prediction policies should be designed, we also calculated the hazard rate of the idle period duration (Figure A.5 in Appendix A). The hazard rate is simply the likelihood that an idle period ends with a duration of at most $k + r$ time units given that it is already k units long. In other words, given that the system has been idle for k units, $H(k, r)$ is the probability that a task initiated now and requiring r units of time will not be completed before the system becomes busy again. More formally,

$$H(k, r) = \frac{\sum_{i=0}^{r} l(k + i)}{1 - L(k - 1)} \tag{2.6}$$

We find that the hazard rate increases with r, meaning that the chances for the task not to be completed before the system becomes busy again increases with the length of the task, as we would expect. In addition, the hazard rate generally declines as the length of time the system has already been idle increases. *This result again supports the idea of predicting the remaining idle period duration by conditioning on the amount of time the system has already been idle.*

2.7 Interaction of Reads and Writes

In general, the interaction between reads and writes complicates a computer system and throttles its performance. For instance, static data can be simply replicated to improve not only the performance of the system but also its scalability and durability. But if the data is being updated, the system has to ensure that the writes occur in the correct order. In addition, it has to either propagate the results of each write to all possible replicated copies or to invalidate these copies. The former usually makes sense if the updated data is unlikely to be updated again but is likely to be read. The latter is useful when it is highly likely that the data will be updated several more times before it is read. In cases where the data is being both updated and read, replication may not be useful. The read-write composition of the traffic, together with the flow of data from writes to reads, is therefore an extremely important workload characteristic. This is the focus of this section.

2.7.1 Read/Write Ratio

A wide range of read/write ratio has been reported in the literature. In addition to intrinsic workload differences, the read-to-write ratio also depends to a large extent on how much of the reads and writes have been filtered by caching, and on the kinds of I/Os (*e.g.,* user data, paging, file system metadata) that are tabulated. Because main memory is volatile, the amount of write buffering performed by the file system cache is typically limited. For example, UNIX systems have traditionally used a policy of periodically (once

	Requests (# Read Requests / # Write Requests)	Traffic (MB Read / MB Written)	Footprint (# Unique Blocks Read / # Unique Blocks Written)
P1	2.51	1.99	1.05
P2	1.37	1.79	1.55
P3	0.429	0.430	0.563
P4	0.606	0.550	0.585
P5	0.338	0.475	1.02
P6	0.147	0.231	0.322
P7	0.288	0.299	0.399
P8	1.23	1.14	0.941
P9	0.925	1.02	1.38
P10	0.937	1.41	2.17
P11	0.831	1.38	0.787
P12	0.758	0.883	0.904
P13	0.566	0.744	1.40
P14	0.481	0.710	0.770
P-Avg.	0.816	0.932	0.988
Pf-Avg.	0.965	0.607	0.888
FS1	0.718	0.633	1.50
TS1	0.794	0.740	1.15
DS1	0.607	1.24	1.06
S-Avg.	0.706	0.870	1.24
Sf-Avg.	1.12	0.843	1.19

Table 2.13: Read/Write Ratio.

every 30s) flushing the dirty blocks in the file cache to disk so as to limit the amount of data that can potentially be lost in a system failure. In Windows NT, one quarter of the dirty data in the file cache is written back to disk every second [Rus98]. Therefore, more of the reads than writes are filtered by the file system cache. The file system also adds metadata writes which may account for more than half of the physical writes (more than 72% in [RW93] and more than 53% in our PC workloads). Thus at the logical level, the read/write ratio is generally much higher than at the physical level.

For instance, the ratio of logical read to write traffic has been reported to be between 3.7 and 6.3 for desktop workstation workloads [RLA00], and the ratio of logical read to write operations has been found to be between 3 and 4.5 in various office environments [RBK92]. But at the physical level, the read/write ratio has been observed to range from only about 0.4 to 1 for Novell NetWare file servers [HH95] and from about 0.7 to 0.8 for several HP-UX systems [RW93]. These figures are comparable to the physical read/write ratio we obtained, which are presented in Table 2.13. Observe that *for the various server workloads and the PC workloads on average, the ratio of read to write requests varies from 0.6 to 0.82, which means that writes account for about 60% of the requests*. Interestingly, mainframe data processing workloads appear to have a higher read/write ratio. For example, measurements conducted at the physical level at 12 moderate-to-large MVS installations

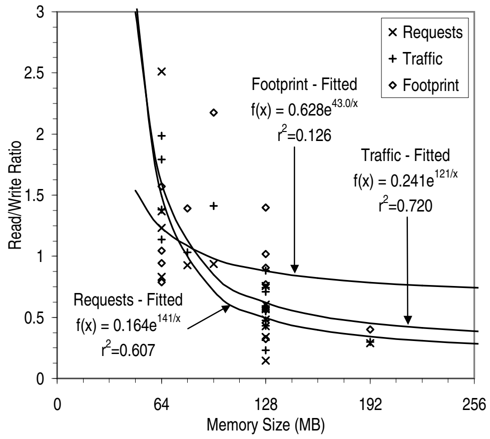
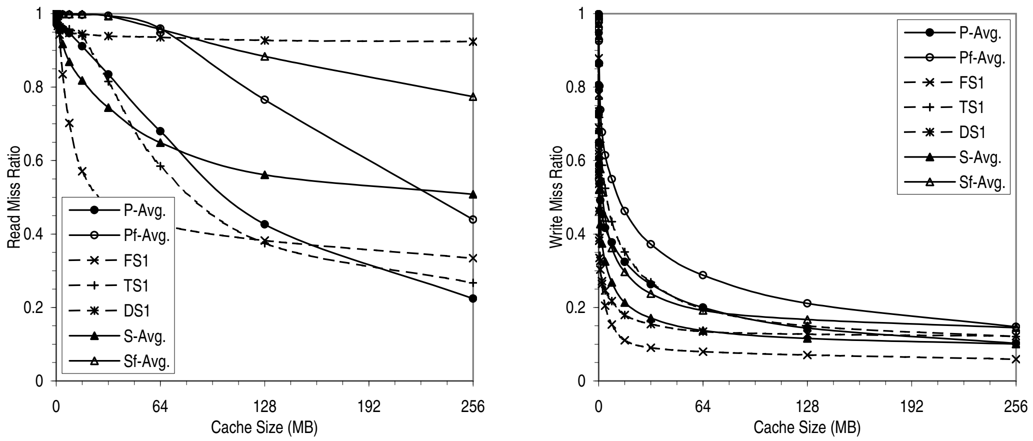


Figure 2.16: Read/Write Ratio as Function of Memory Size.

running mainly data processing applications (circa 1993) found the read/write ratio to be about 3.5 [McN95]. Analysis of the logical I/O traffic of the production database workloads of ten of the world's largest corporations of about the same period found the read/write ratio to average roughly 10 [HSY01a, HSY01b].

Observe from Table 2.13 that for the PC workloads, the read/write ratio appears to be negatively correlated with the memory size of the system. Unfortunately, we do not have enough data points to observe any trends for the server workloads. In Figure 2.16, we plot the read/write ratio for the PC workloads as a function of the memory size. As shown in the figure, the read/write ratio is approximately related to the memory size by an exponential function of the form $f(x) = ae^{b/x}$ where a and b are constants. The model is limited by the few data points we have but it predicts that with an infinitely large memory, *i.e.,* as $x \to \infty$, there will be about 6 writes for every read. Such results support the prediction that almost all reads will be absorbed by the larger buffer caches in the future so that physical I/O will become dominated by writes [OD89]. However, that the read/write ratio remains relatively consistent across all our workloads, which span a time period of eight years, suggests that workload changes may have a counter effect. Also, the fact that the ratio of read footprint to write footprint decreases, albeit slowly, with memory size, suggests that effects (*e.g.,* workload differences) other than an increase in caching, could also be at work here.

If writes become increasingly dominant, a pertinent question to ponder is whether physical read performance really matters. In Figure 2.17, we plot the read and write cache miss ratios assuming a write-back cache with the least-recently-used (LRU) cache replacement policy. We define the miss ratio to be the fraction of requests that cannot be satisfied by the cache but that result in a request to the underlying storage system. Observe that the plots for the filtered workloads are simply a translation of those for the original workloads; the behavior is qualitatively similar. In this experiment, we are in essence simulating a second level cache. The upstream file system cache and/or the database buffer pool have captured significant portions of any read reuse but because they are volatile, they cannot



(a) Read Miss Ratio.

(b) Write Miss Ratio.

Figure 2.17: Miss Ratio with LRU Write-Back Cache (512-Byte Blocks).

safely cache the writes. Therefore, the *writes observed at the storage level exhibit much stronger locality than the reads. In other words, although read caching by the file system or the database buffer can eliminate most of the reads, if writes are delayed long enough by using non-volatile memory, write requests can similarly be very significantly reduced. In fact, for practically all the workloads, a small cache of 1 MB eliminates more than half the writes.*

Furthermore, unlike reads which tend to be synchronous, writes can be effectively rendered asynchronous through the use of write caching or buffering [Chapter 3]. In addition, the effective latency of writes can often be reduced by writing data asynchronously or in a log [RO92, WAP99] or by using write-ahead logging [MHL$^+$92]. Recent results (*e.g.,* [Dah95]) further suggest that because of the widening performance gap between processor and disk-based storage, file system read response times may be dominated by disk accesses even at very high cache hit rates. Therefore, *the performance of read I/Os continues to be very important.*

2.7.2 Working Set Overlap

The working set $W(t, \tau)$ is defined as the set of blocks referenced within the last τ units of time [Den68]. More formally,

$$W(t, \tau) = \{b : Count(b, t-\tau, t) >= 1\} \tag{2.7}$$

where $Count(b, t-\tau, t)$ denotes the number of times block b is referenced between $t-\tau$ and t. In Figure 2.18, we plot the average and maximum daily working set size for our workloads. Note that we define the working set of day x as $W(t$=midnight of day x,

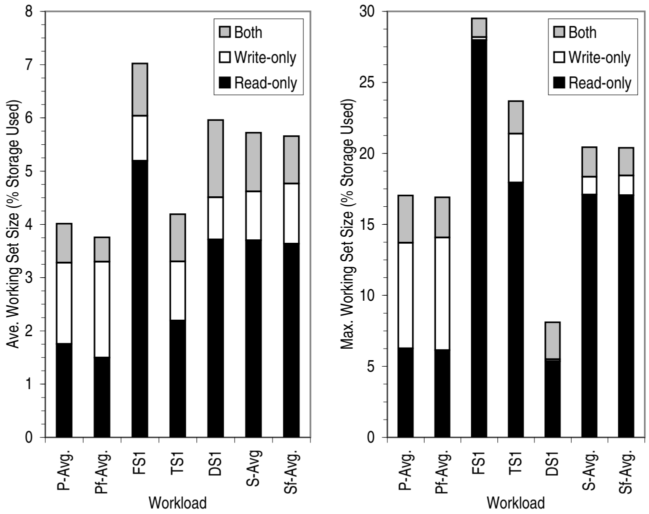


Figure 2.18: Daily Working Set Size.

τ=1 day). To understand the interaction between reads and writes, we classify the blocks referenced into those that are read, written, and both read and written. Specifically,

$$\begin{aligned} W_{read}(t,\tau) &= \{b : ReadCount(b, t-\tau, t) >= 1\} \\ W_{written}(t,\tau) &= \{b : WriteCount(b, t-\tau, t) >= 1\} \\ W_{both}(t,\tau) &= W_{read}(t,\tau) \cap W_{written}(t,\tau) \end{aligned}$$

Observe that on average, the daily working set for the various workloads range from just over 4% (PC workloads) to about 7% of the storage used (FS1). The size of the working set is not constant but fluctuates day to day so that the maximum working set can be several times larger than the average. Notice further from Figure 2.18 that the working set of blocks that are both read and written is small, representing less than 25% of the total working set size for all the workloads. To better understand the interaction between the blocks that are read and those that are written, we introduce the idea of the generalized working set $W(t,\tau,c) = \{b : Count(b, t-\tau, t) >= c\}$. The working set first introduced in [Den68] is simply the special case where $c = 1$. Figure 2.19 presents the average daily generalized working set size for our workloads as a function of c, the minimum number of times a block is referenced in a day for it to be considered part of the working set. The figure shows that for all the workloads, the relationship between the average size of the daily generalized working set and c can be approximately described by a reciprocal function of the form $f(c) = \frac{a}{c^b}$ where a and b are positive constants. That the working set decreases sharply as c increases beyond one indicates that *only a small fraction of the data stored is in active use, suggesting that it is probably a good idea to identify the blocks that are in use and to optimize their layout. We will consider this in detail in Chapter 4.* Notice also that *the amount of data that is both actively read and updated is clearly very small.* In the

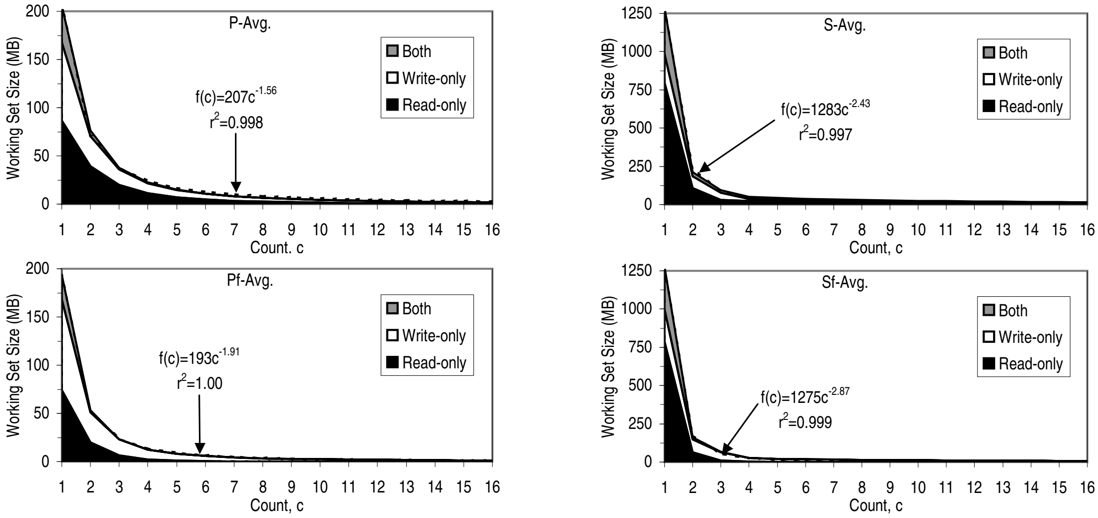


Figure 2.19: Average Daily Generalized Working Set Size.

next section, we will examine this further by looking at the dependencies between reads and writes.

2.7.3 Read/Write Dependencies

Dependencies are generally classified into three categories – *true dependencies* (Read After Write or RAW), *output dependencies* (Write After Write or WAW) and *anti dependencies* (Write After Read or WAR). A RAW is said to exist between two operations if the first operation writes a block that is later read by the second operation and there is no intervening operation on the block. WAW and WAR are similarly defined.

In Figure 2.20, we plot the percentage of reads for which there is a write within τ references that constitute a WAR. We refer to τ as the window size. Observe that even for a large window size of 100,000 references, less than 25% of the reads fall into this category for all the workloads. In other words, *blocks that are read tend not to be updated so that if disk blocks are replicated or reorganized based on their read access patterns, write performance will not be significantly affected.* Notice from Figures 2.21 and 2.22 that all the workloads contain more WAW than RAW. This implies that *updated blocks are more likely to be updated again than to be read. Therefore, if we do replicate blocks, we should only update one of the copies and invalidate the rest rather than update all the copies.* In other words, a write-invalidate policy will work better than a write-broadcast policy. Again, we see that the results for the filtered traces are quantitatively different from those for the original traces but they lead to the same conclusions.

For the PC traces, we are able to match up I/O requests with the corresponding filename. To better understand the dependencies, we rerun the RAW and WAW analysis for these workloads excluding references to the file system metadata and to log files, which

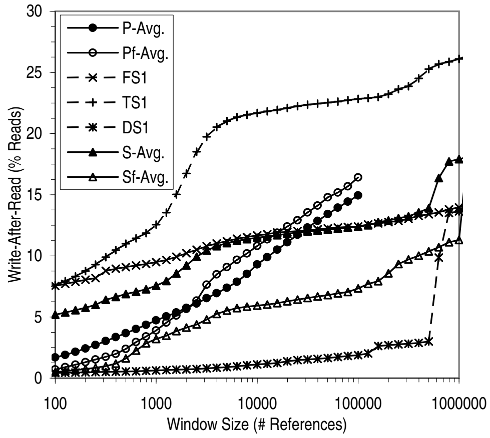


Figure 2.20: Frequency of Occurrence of Write after Read (WAR).

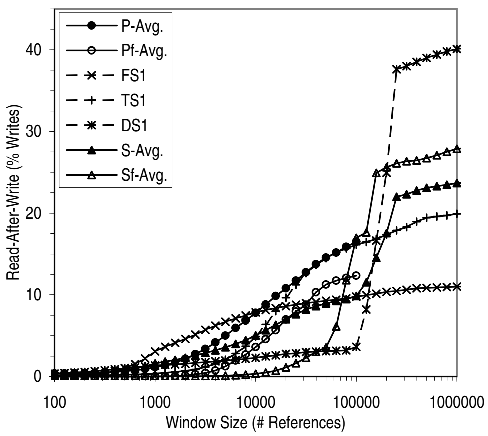


Figure 2.21: Frequency of Occurrence of Read after Write (RAW).

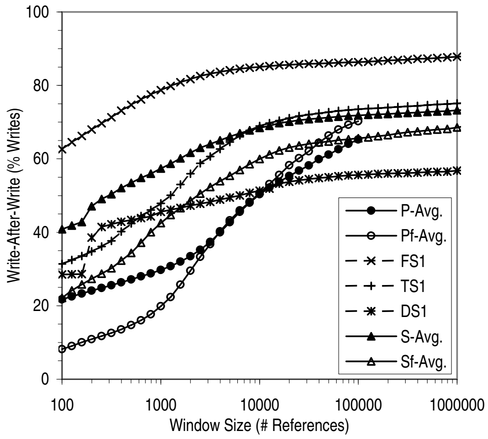


Figure 2.22: Frequency of Occurrence of Write after Write (WAW).

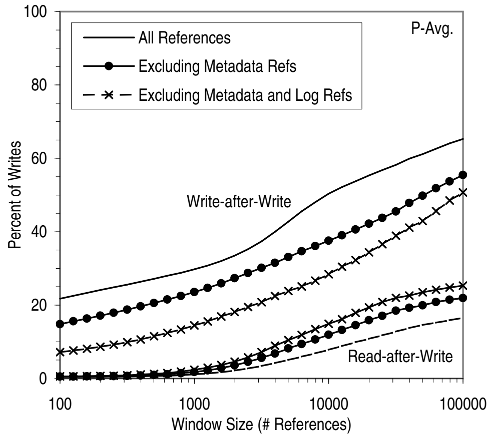


Figure 2.23: Frequency of Occurrence of Read-After-Write (RAW) and Write-After-Write (WAW) when Metadata and/or Log References are Excluded. Metadata references are those that access blocks belonging to directories and to the following files: \$mft, \$attrdef, \$bitmap, \$boot, \$logfile, \$mftmirr and \$upcase. Log references are those that access blocks belonging to the following files: *\config*, *\profiles*, *.log, log.* and netscape.hst.

respectively constitute about half and a quarter of all the write requests. The results are summarized in Figure 2.23. Observe that once the metadata and log references are filtered out, it is still the case that updated blocks are more likely to be updated again than to be read, but less so. In other words, although metadata and log writes make it more likely for an updated block to be updated again than to be read, they are not the only cause for this behavior.

2.8 Conclusions

In this chapter, we empirically analyze the I/O traffic of a wide range of real workloads with an emphasis on understanding how these workloads will respond to new storage developments such as network storage, storage utilities, and intelligent self-optimizing storage. As part of our analysis, we also study the effect of increased upstream caching on the traffic characteristics seen by the storage system and discover that it affects our analysis only quantitatively. Our major findings include:

- Importance of I/O Innovation/Optimization

  I/O is known to be a major component of server workloads and improving the I/O performance for these workloads is critical. Our results suggest that if processors continue to increase in performance according to Moore's Law, I/O is likely to also become a dominant component of personal computer workloads in the next few years. Our data show that consistently across all the workloads, writes account for about

60% of the requests. However, just as read caching by the file system or the database buffer can eliminate most of the reads, if writes are delayed long enough (*e.g.,* by using non-volatile memory), write requests can similarly be very significantly reduced. In fact, for practically all the workloads, a small write-back cache of 1 MB eliminates more than half the writes. Therefore, we believe that the performance of read I/Os is likely to continue to have a direct impact on application performance. As part of our analysis, we reexamine Amdahl's rule of thumb for a balanced system and discover that our server workloads generate on the order of 0.05 bits of physical I/O per instruction, consistent with our earlier work using the production database workloads of some of the world's largest corporations [HSY01a]. The figure for the PC workloads is seven times lower at about 0.007 bits of physical I/O per instruction. We also find that the average request size is about 8 KB.

- Burstiness of I/O Traffic

  Across all the workloads, read and write I/O requests seldom occur singly but tend to arrive in groups. We find that the write queue is very much deeper than the read queue. Our analysis also indicates that there is little cross-correlation in traffic volume among the server workloads, suggesting that aggregating them will likely help to smooth out the traffic and enable more efficient utilization of resources. As for the PC workloads, multiplexing them will remove the high frequency fluctuations in I/O traffic but some of the time-of-day effects are likely to remain unless the PCs are geographically distributed in different time zones. In addition, our results show that to satisfy I/O bandwidth requirements 99% of the time, we would need to provision for 15 times the long-run average bandwidth. Going beyond 99% of the time would require disproportionately more resources. It turns out that for time scales ranging from tens of milliseconds to tens and sometimes even hundreds of seconds, the I/O traffic is well-represented by a self-similar process. This implies that the I/O system may become overwhelmed at much lower levels of utilization than expected with the commonly assumed Poisson model. Such behavior has to be taken into account when designing storage systems, and in the service level agreements (SLAs) when outsourcing storage. We recommend that I/O traffic be characterized by a three-tuple consisting of the mean and variance of the arrival rate, and the Hurst parameter.

- Potential for Harnessing "Free" Resources

  We find that our PC workloads contain a lot of processor idle time for performing background tasks, even without having to deliberately leave the computer on when the user is away. The storage system is also relatively idle. For all the workloads, a system that exploits idle time can get most of the potential benefit by simply focusing on the long idle periods. In both the PC and server environments, there are idle periods that are both long enough and that occur frequently enough to be interesting for offline optimizations such as block reorganization. In the server environment, we might have to be more meticulous in using the available idle time, for instance, by dividing an idle-time task into several finer-grained steps that can be scheduled whenever there is a short idle period. Our results suggest that the length of an idle period can be predicted more accurately by conditioning on the amount of time the system has already been idle than from the lengths of the recent idle periods.

- Opportunity for Block Reorganization

  In general, I/O traffic is low enough for it to be feasible to collect a daily trace of the blocks referenced for later analysis and optimization. We discover that only a small fraction of the data stored is in active use, suggesting that it is probably a good idea to identify the blocks that are in use and to optimize their layout. In addition, the amount of data that is both actively read and updated is very small. Moreover, blocks that are read tend not to be updated so that if blocks are reorganized or replicated based on their read access patterns, write performance will not be significantly affected. Because updated blocks are more likely to be updated again than to be read, if blocks are replicated, a write-invalidate policy will tend to work better than a write-broadcast policy.

Chapter 3

The Real Effect of I/O Optimizations and Disk Improvements

3.1 Synopsis

Many optimization techniques have been invented to mask the slow mechanical nature of storage devices, most importantly disks. Data on the effectiveness of these techniques for real workloads, however, are either lacking or are not comparable. Disk technology has also improved steadily in multiple ways but it is not clear how the various physical improvements relate to the actual performance experienced by real workloads. Therefore, in this chapter, we use an assortment of real server and personal computer workloads to systematically analyze the various optimization techniques and technology improvements so as to determine their true performance impact. The techniques we study include read caching, sequential prefetching, opportunistic prefetching, write buffering, request scheduling, striping and short-stroking. We also break down the steady improvement in disk technology into four major basic effects, and analyze each separately to determine their actual benefit. In addition, we examine their historical rates of improvement and use the trends to project the effect of disk technology scaling. As part of this study, we develop a methodology for replaying the real workloads that more accurately models the timing of I/O arrivals and that allows the I/O rate to be more realistically scaled than previous practice.

Our results show that sequential prefetching and write buffering are the two most effective techniques for improving read and write performance respectively. For our workloads, improvement in the mechanical components of the disk reduces the average response time by 8% per year. Most of this improvement results from increases in the rotational speed rather than reduction in the seek time. In addition, we discover that increases in the recording density of the disk can achieve an equally sizeable improvement in real performance, with most of the gain coming from linear density improvement, which increases the transfer rate, rather than track density scaling. For a given workload, disk technology evolution at the historical rates can be expected to increase performance by about 8% per year if the disk occupancy rate is kept constant. We also observe that the disk is spending most of its time positioning the head rather than transferring data. We believe that to effectively utilize the available disk bandwidth, blocks should be reorganized in such a way that accesses become more sequential.

3.2 Introduction

Because of the slow mechanical nature of many storage devices, the importance of optimizing I/O operations has been well recognized. As a result, a plethora of optimization techniques including caching, write buffering, prefetching, request scheduling and parallel I/O have been invented. The relative effectiveness of these techniques, however, is not clear because they have been studied in isolation by different researchers using different methodologies. Furthermore, many of the techniques have not been evaluated with real workloads thus their actual effect is not known. Some of the ideas have just been proposed or implemented with little or no performance results published (*e.g.,* opportunistic prefetching). As the performance gap between the processor and disk-based storage continues to widen [Gro00, PK98], increasingly aggressive optimization of the storage system is needed, and this requires a good understanding of the real potential of the various techniques and how they work together. In this chapter, we systematically investigate how the different I/O optimization techniques affect actual performance by using trace-driven simulations with a large set of traces gathered from a wide range of real-world settings, including both server and personal computer (PC) environments. To make our findings more broadly applicable, we focus on general rules of thumb about what can be expected from each of these techniques rather than precise quantification of improvement for a particular workload and a specific implementation.

Tremendous efforts have also gone into improving the underlying technology of disks. The improvement in disk technology is usually quantified by using physical metrics such as the tracks or bits per inch, the average seek time and the rotational speed. Relating such physical metrics to the performance delivered to real workloads is, however, difficult. Thus it is not apparent how an improvement in one metric compares with an improvement in another in terms of their real-world impact. Furthermore, some of the metrics are not focused on performance but have a significant effect on it. Increasing the recording density, for example, could improve performance because if the bits are packed more closely together, they can be accessed with a smaller physical movement. While these metrics are considered too low-level from a systems perspective, some of them may also not be very useful for designing disks because they compound several basic physical effects. For instance, a seek could be faster because of an increase in the track density, a reduction in the width of the data band, mechanical improvement in the disk arm actuator, *etc.* Therefore, in this thesis, we break down the steady improvement in disk technology into four major basic effects, and analyze each separately to determine their effect on real workloads. In addition, we examine their historical rates of improvement and use the trends to project the actual performance improvement that can be expected from disk technology scaling.

In the previous chapter, we analyzed in detail the characteristics of the various workloads we use, specifically, (1) the I/O intensity of the workloads and the overall significance of I/O in the workloads, (2) how the I/O load varies over time and how it will behave when aggregated, and (3) the interaction of reads and writes and how it affects performance. Although this chapter is self-contained, readers are encouraged to also read the preceding chapter to better understand the workloads on which this analysis is based. The insights gained from the current study motivated the idea of Automatic Locality-Improving Storage (ALIS), which is a storage system that continually monitors the way it is accessed and then automatically reorganizes selected disk blocks so that accesses become effectively more sequential [Chapter 4]. In fact, the results we derive here serve as the baseline for the analysis of ALIS in the next chapter. Therefore, this chapter has an emphasis on the optimizations that directly affect ALIS, in particular, the prefetching.

The rest of this chapter is organized as follows. Section 3.3 contains a brief overview of previous work in evaluating I/O optimization techniques. Section 3.4 discusses our methodology and describes the traces that we use. In Section 3.5, we analyze the effect of the various optimization techniques. In Section 3.6, we consider the real impact of disk technology improvement over time. Section 3.7 concludes this chapter. Because of the huge amount of data that is involved in this study, we can only present a characteristic cross-section in the main text. More detailed graphs and data are presented in Appendix C. Note that figures and tables in the appendix are identified by a prefix C.

3.3 Related Work

The various I/O optimization techniques have been individually evaluated by different researchers using dissimilar methodologies including discrete event simulation and analytical modeling. In some cases, the simulations are based on traces of real workloads and in others, randomly generated synthetic workloads are used. For instance, disk caching is extensively analyzed in [Smi85, ZS97], prefetching in [GA94, Smi78], write buffering in [BRT93, VJ98], request scheduling in [JW91, SCO90, WGP94] and striping in [CL95, CLG+94]. At the logical level, caching, prefetching and write buffering are well covered in [HSY01b, NWO88]. Several researchers have also explored ways to improve the various techniques in special situations where the reference pattern is known ahead of time (*e.g.,* [PGG+95]). Because of the importance of improving I/O performance, there has been a lot of research on I/O optimization techniques. We mention only some of the more recent work. The reader is referred to [Smi81] for a comprehensive survey of early work on I/O optimization.

3.4 Methodology

The methodology used in this chapter is trace-driven simulation [Smi94, UM97]. In trace-driven simulation, relevant information about a system is collected while the system is handling the workload of interest. This is referred to as tracing the system and is usually achieved by using hardware probes or by instrumenting the software. In the second phase, the resulting trace of the system is played back to drive a model of the system under study. Trace-driven simulation is thus a form of event-driven simulation where the events are taken from a real system operating under conditions similar to the ones being simulated. A common difficulty in using trace-driven simulations to study I/O systems is to realistically model timing effects, specifically to account for events that occur faster or slower in the simulated system than in the original system. This difficulty arises because information about how the arrival of subsequent I/Os depend upon the completion of previous requests cannot be easily extracted from a system and recorded in the traces.

3.4.1 Modeling Timing Effects

In general, simulation models used for evaluating storage system performance can be broadly classified into open and closed models, depending on how request arrivals are choreographed. The closed model traditionally maintains a constant population of outstanding requests. Whenever a request is completed, a new request is issued in its place, sometimes after a simulated "think" time. These models essentially assume that all the I/Os are time-critical [Gan95] so that a new I/O is issued only after a previous request is completed. By maintaining a constant population of outstanding requests, these models effectively smooth out any burstiness in the I/O traffic. Such an approach is clearly not representative of real workloads, which have been shown in several studies (*e.g.,* Chapter 2) to have bursty I/O traffic patterns.

In the open model, requests arrive at predetermined times (*e.g.,* traced time in [RW91] and traced inter-arrival time scaled by a constant factor in [WGP94]), independent of the performance of the storage system. Such models assume that the workload consists exclusively of time-noncritical requests [Gan95] so that whether a preceding request is completed has no bearing on when the system is able to issue subsequent I/Os. Again, this is clearly not true in real systems where an overloaded storage system, by being slow, automatically exerts back pressure on the processes generating the I/Os. For example, 66-91% of the I/Os are flagged as synchronous in PC workloads [Chapter 2] and 52-74% in UNIX workloads [RW93]. In other words, the system generally has to wait for I/Os to be completed before it can continue with subsequent processing. Such data highlights the importance of accounting for the feedback effect between request completion and subsequent request issuance. From a practical perspective, having a feedback mechanism also ensures that the number of outstanding requests will not grow without bound whenever the storage system is unable to handle the incoming workload.

Modeling the feedback effect and thereby limiting the number of outstanding requests is especially helpful in this study because we have a diverse set of workloads collected over the span of several years, and a wide range of experiments in which the performance of the storage system is significantly varied. Some of our experiments evaluate techniques that are opportunistic, *i.e.,* they take advantage of idle time. Therefore, we have to account for the burstiness seen in real I/O traffic. With these requirements in mind, we came up with a methodology that is designed to incorporate feedback between request completion and subsequent I/O arrivals, and model burstiness.

Results in Chapter 2 show that there is effectively little multiprocessing in PC workloads and that most of the I/Os are synchronous. Such predominantly single-process workloads can be modeled by assuming that after completing an I/O, the system has to do some processing and the user, some "thinking", before the next set of I/Os can be issued. The "think" time between the completion of a request and the issuance of its dependent requests can be adjusted to speed up or slow down the workload. In short, we consider a request to be dependent on the *last* completed request, and we issue a request only after its parent request has completed. For multiprocessing workloads, this dependence relationship should be maintained on a per process basis but unfortunately process information is typically not available in I/O traces. Therefore, in order to account for multiprocessing workloads, we merge multiple traces to form a workload with several independent streams of I/O, each obeying the dependence relationship described above.

In essence, we have built an out-of-order multiple issue machine that tries to preserve the dependency structure between I/O requests. We maintain an issue window of 64 requests. A request within this window is issued when the request on which it is dependent completes and the think time has elapsed. Inferring the dependencies based on the last completed request is the best we can do given the block level traces we have. If the workloads were completely described using logical and higher-level system events (*e.g.,* system calls and interrupts), we might be able to more accurately model feedback effects using a system-level model (*e.g.,* [Gan95]). In the limit, we can run the workloads on a system simulator where we have control over the timing of events [RHWG95] or on a virtual machine [CFH+80] or on a real system with a timing-accurate storage emulator [GSS+02]. However, getting real users to release traces of reference address is difficult enough. Asking them for logical data about their computer operations is next to impossible. Moreover, "capturing" a workload so that it can be realistically replayed may be relatively easy for batch jobs but it is very difficult for interactive workloads. We essentially end up with the same problem of having to decide what happens when the system reacts faster. For instance, will the user click the mouse earlier?

3.4.2 Workloads and Traces

The traces analyzed in this study are primarily those characterized in Chapter 2. We do not include FS2 in the analysis here since it does not contain the addresses of individual I/Os. Most of the traces were gathered over periods of several months but to keep the simulation time manageable, we use only the first 45 days of the traces of which the first 20 days are used to warm up the simulator. For the DS1 trace, which is only seven days long, we use the first three days to warm up our simulator.

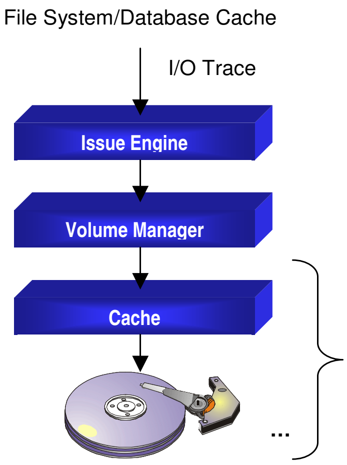
In addition to the base workloads as recorded in the traces, we scale up the traces to obtain workloads that are more intense. Results reported in the previous chapter show that for the PC workloads, the processor utilization during the intervals between the issuance of an I/O and the last I/O completion is related to the length of the interval by a function of the form $f(x) = 1/(ax + b)$ where $a = 0.0857$ and $b = 0.0105$. To model a processor that is n times faster than was in the traced system, we would scale only the system processing time by n, leaving the user portion of the think time unchanged. Specifically, we would replace an interval of length x by one of length $x[1 - f(x) + f(x)/n]$. In this thesis, we run each workload preserving the original think time. For the PC workloads, we also evaluate what happens in the limit when systems are infinitely fast, *i.e.,* we replace an interval of length x by one of $x[1 - f(x)]$. We denote these sped-up PC workloads as P1s, P2s, ..., P14s and the arithmetic mean of their results as Ps-Avg.

We also merge ten of the longest PC traces to obtain a workload with ten independent streams of I/O, each of which obeys the dependence relationship discussed above. We refer to this merged trace as Pm. The volume of I/O traffic in this merged PC workload is similar to that of a server supporting multiple PCs. Its locality characteristics are, however, different because there is no sharing of data among the different users so that if two users are both using the same application, they end up using different copies of the application. Pm might be construed as the workload of a system on which multiple independent PC workloads are consolidated. For the server workloads, we merge the FS1 and TS1 traces to obtain Sm. Note that neither method for scaling up the workloads is perfect but we believe that they are more realistic than simply scaling the inter-arrival time, as is commonly done. In this thesis, we often use the term *PC workloads* to refer collectively to the base PC workloads, the sped-up PC workloads and the merged PC workload. The term *server workloads* likewise refers to the base server workloads and the merged server workload.

3.4.3 Simulation Model

The major components of our simulation model are presented in Figure 3.1. In practice, optimizations such as caching, prefetching, write buffering, request scheduling and striping may be performed at multiple levels in the storage system. For instance, there may be several storage controllers, storage adaptors and disk drives, and they may all perform some of the optimizations to some extent. The number of combinations of who does what and to what extent is large, and the interaction between the optimizations performed at the various levels is complicated and obscure. In order to gain fundamental insights into the effectiveness of each of the optimizations, we collapse the different levels and model each of the optimizations at most once.

For instance, we model only a single level of cache instead of a disk cache, an adaptor cache, a controller cache, *etc.* This approach does not expose the interference that occurs



	Resource-Poor	Resource-Rich
Read Caching	8MB per disk, Least-Recently-Used (LRU) replacement.	1% of storage used, Least-Recently-Used (LRU) replacement.
Prefetching	32KB read-ahead. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks.	Conditional sequential prefetch, 16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks, 8MB per disk opportunistic prefetch buffer.
Write Buffering	4MB, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 30s age limit.	0.1% of storage used, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 1 hour age limit.
Request Scheduling	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.
Striping	Stripe unit of 2MB.	Stripe unit of 2MB.

Figure 3.1: Block Diagram of Simulation Model Showing the Base Configurations and Default Parameters Used to Evaluate the Various I/O Optimization Techniques and Disk Improvements. The parameters pertaining to each technique will be described in detail in Section 3.5.

when the different levels in the storage stack are all trying to do some of the same optimizations. But cutting down on the interference is the only way we can look at the real effect of each of the optimizations. The interference is interesting but is beyond the scope of the current thesis. Furthermore, a well-designed system will have a level at which a particular technique dominates. For instance, for caching, the adaptor cache should be bigger than the disk cache so that its effect dominates. For other techniques such as request scheduling, there is a level where it can best be implemented. Throughout the chapter, we discuss such issues and how we handle them in our simulator.

Even though we simulate only a single instance of each of the optimization techniques, there are many parameters for each technique and their combination makes for a huge design space. In order to systematically examine the effect of each technique, we pick two reasonable base configurations and perturb them in *one* dimension at a time. The *default* parameters used in these base configurations are summarized in Figure 3.1. As we study each technique individually, the relevant parameters will be analyzed and described in detail. As its name suggests, the resource-rich configuration is meant to represent an environment in which resources in the storage system are plentiful, as may be the case when there is a large outboard controller. The resource-poor environment is supposed to mimic a situation where the storage system consists of only disks and low-end disk adaptors.

Our simulator is written in C++ using the CSIM simulation library [Mes94]. It is based upon a detailed model of the mechanical components of the IBM Ultrastar 73LZX [IBM01b] family of disks that is used in disk development and that has been validated against test measurements obtained on several batches of the disk. The level of detail in this model is similar to that in the publicly available DiskSim package [GWP99]. However,

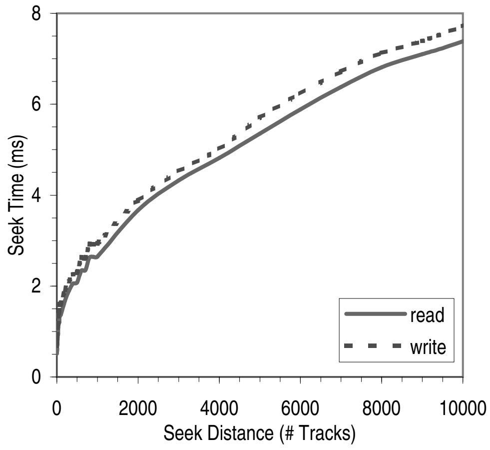


Figure 3.2: Seek Profile for the IBM Ultrastar 73LZX Family of Disks.

instead of using the same seek profile for reads and writes and accounting for the difference by a constant write settling delay, we use separate read and write seek curves to more accurately model the disk. As shown in Figure 3.2, the seek curves for this disk can be approximated by a power function for seeks of less than 5000 tracks and a linear function for longer seeks.

The IBM Ultrastar 73LZX family of 10K RPM disks was introduced in early 2001 and consists of four members with storage capacities of 9.1 GB, 18.3 GB, 36.7 GB and 73.4 GB. The performance characteristics of each is almost identical, with the difference in capacity coming from the number of platters. The higher-capacity disk should have a longer seek time because of the increased inertia of the disk arm but the effect is small. The average seek time is specified to be 4.9 ms and the data rate varies from 29 MB/s at the inner edge to 57 MB/s at the outer edge. The track density for this series of disks is 27,000 tracks per inch while the linear density is as high as 480,000 bits per inch. The tracks range in size from 160KB to 340KB. More details about the specifications of this family of disks can be found in [IBM01b]. In order to understand the effect of disk technology evolution, in the later part of this chapter, we scale these disk characteristics according to technology trends which we derive by analyzing the specifications of disks introduced in the last ten years.

For workloads with multiple disk volumes, we concatenate the volumes to create a single address space. In the base configurations, each workload is fitted to the smallest disk from the IBM Ultrastar 73LZX family that is bigger than the total volume size. We leave a headroom of 20% because the results presented here are part of a larger study that examines replicating up to 20% of the disk blocks and laying them out in a specially set aside area of the disk [Chapter 4]. When we study parallel I/O, we will look at the effect of striping the data across multiple disks. Note that we have a separate read cache and write buffer to allow us to adjust the size of each independently. Results in Chapter 2 show that there is not a lot of interaction between the reads and the writes.

3.4.4 Performance Metrics

I/O performance can generally be measured at different levels in the storage hierarchy. In order to quantify the effect of a wide variety of storage optimization techniques, we measure performance from when requests are issued to the storage system, before they are potentially broken up by the volume manager for requests that span multiple disks. The two important metrics in I/O performance are *response time* and *throughput*. Response time includes both the time needed to service the request and the time spent waiting or queueing for service. Throughput is the maximum number of I/Os that can be handled per second by the system. Quantifying the throughput is generally difficult with trace-driven simulation because the workload, as recorded in the trace, is constant. We can try to scale or speed up the workload to determine the maximum workload the system can sustain but this is difficult to achieve in a realistic manner.

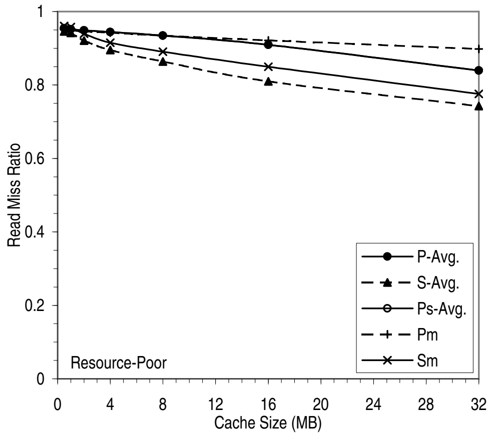
In this thesis, we estimate the throughput by considering the amount of critical resource each I/O consumes. Specifically, we look at the average amount of time the disk arm is busy per request, deeming the disk arm to be busy both when it is being moved into position to service a request and when it has to be kept in position to transfer data. We refer to this metric as the *service time*. Throughput can be approximated by taking the reciprocal of the average service time. One thing to bear in mind is that there are opportunistic techniques, especially for reads (*e.g.,* preemptible read-ahead), that can be used to improve performance. The service time does not include the otherwise idle time that the opportunistic techniques exploit. Thus the service time of a lightly loaded disk will tend to be optimistic of its maximum throughput.

To gain insight into the workings of the different optimization techniques, we also examine the effective *miss ratio* of the read cache and the write buffer. The miss ratio is generally defined as the fraction of I/O requests that are not satisfied by the cache or buffer, or in other words, the fraction of requests that requires physical I/O. In order to make our results more useful for subsequent mathematical analyses and modeling by others, we fitted our data to various functional forms through non-linear regression, which we solved by using the Levenberg-Marquardt method [PFTV90].

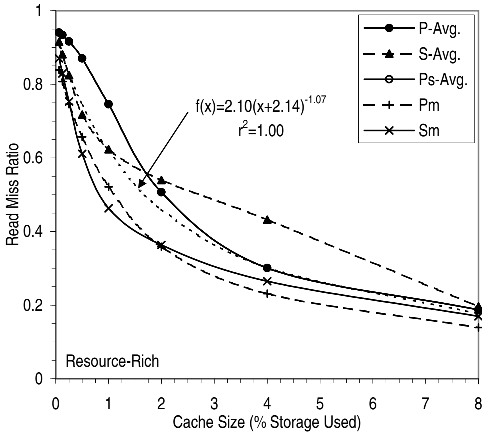
3.5 Effect of I/O Optimizations

3.5.1 Read Caching

Caching is a general technique for improving performance by temporarily holding in a faster memory data items that are (believed to be) likely to be used. The faster memory is called the *cache*. In the context of this thesis, the data items are disk blocks requested from the storage system, and the faster memory refers to dynamic random access memory (DRAM). The fraction of requests satisfied by the cache is commonly called the *hit ratio*. The fraction of requests that have to be handled by the underlying storage system is referred to as the *miss ratio*. The data items can be entered into the cache when they are demand



(a) Resource-Poor.



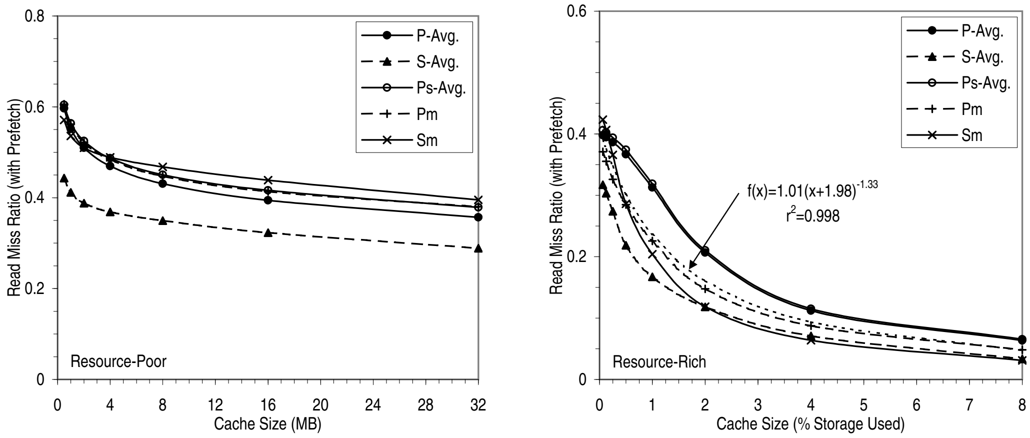
(b) Resource-Rich.

Figure 3.3: Effectiveness of Read Caching at Reducing Physical Reads.

fetched or when it is anticipated that they will likely be referenced soon. Caching usually refers only to the former. The latter is generally called *prefetching* and will be studied in detail in the next section. Note that to focus on the effect of caching, we disable prefetching. This is an exception to our general approach of perturbing, at any one time, only the parameters for *one* technique from their default values listed in Figure 3.1.

Figure 3.3 shows the effectiveness of read caching at reducing physical reads. We use the Least-Recently-Used (LRU) replacement policy since variations of it are commonly used throughout computer systems. Notice from Figure 3.3(a) that the cache is not very useful for sizes up to 32 MB. This is expected because we are looking at the physical reference stream, which has been filtered by the caching going on upstream in the host system. Today, it is common even for PC systems to have more than 100 MB of main memory, most of which can be used for file caching. Yet most disks have only 2-4 MB of cache with some offering an 8 MB option. Our results suggest that at such sizes, the disk cache is not very effective. It serves primarily as a buffer for prefetching. In Figure 3.4(a), we present the cache miss ratio when data is prefetched into the cache using the default parameters for the resource-poor environment. Notice that with prefetching, more than 50% of the reads can be satisfied by a 4 MB cache. Increasing the cache beyond 4 MB to 32 MB achieves only diminishing returns.

Note that if the cache is large enough to hold all the blocks that will be referenced again, the performance will obviously be very good. However, we will need a huge cache because from Figures 3.3(b) and 3.4(b), the miss ratio, even with prefetch, continues to improve at cache sizes that are beyond 4% of the storage used (allocated). In practice, there is a limit to the size of the cache. For instance, physical constraints such as addressing and packaging limitations place an upper limit on the amount of DRAM that can be installed in a system.



(a) Resource-Poor.

(b) Resource-Rich.

Figure 3.4: Sensitivity to Cache Size when Data is Prefetched into the Cache.

Economics also constrains the size of the cache. The cost per GB for DRAM is currently about 50 times higher than for disk storage. This means that a cache that is 1% of the storage space, and that does nothing but helps to mask the poor performance of the disks, will cost as much as half the disk storage. Today, most enterprise class storage controllers, when fully loaded, have cache sizes that are in the range of 0.05% to 0.2% of the storage space [EMC01, Hit02, IBM00]. In this study, we set the cache size aggressively to 1% of the storage used in the resource-rich environment and 8 MB per disk in the resource-poor environment.

In order to establish a rule of thumb relating the read miss ratio to the size of the cache, we took the average of the five plots in Figure 3.3(b) and fitted various functional forms to it. As shown in the figure, a good fit is obtained with a power function of the form $f(x) = a(x - b)^c$ where a, b and c are constants. This relationship based on the physical I/O stream turns out to be functionally similar to what has been found at the logical level for large database systems [HSY01b]. However, at the logical level, the c value is about -0.5, half of the -1 in our case. This means that the physical read miss ratio for our workloads improves faster with increase in the cache size than is the case at the logical level for large database systems. Such results suggest that caching can be effective at the physical level provided that the cache is large enough. When data is prefetched into the cache, the c value, at about -1.3, is even lower (Figure 3.4(b)), meaning that the read miss ratio improves even faster as the cache size is increased.

In Table 3.1, we summarize the effectiveness of read caching at improving performance. Throughout this thesis, we define improvement as $(value_{old} - value_{new})/value_{old}$ if a smaller value is better and $(value_{new} - value_{old})/value_{old}$ otherwise. Note that some amount of cache memory is needed as a speed matching buffer between the disk media and

		Read					
		Avg. Resp. Time		Avg. Serv. Time		Miss Ratio	
		ms	%[i]	ms	%[i]		%[i]
Resource-Poor	P-Avg.	6.27	2.46	4.31	2.11	0.934	2.12
	S-Avg.	5.34	9.01	3.88	8.34	0.864	8.93
	Ps-Avg.	6.96	2.33	4.34	2.09	0.934	2.13
	Pm	6.04	2.26	4.18	1.83	0.935	1.81
	Sm	5.69	6.36	4.10	6.34	0.891	7.27
Resource-Rich	P-Avg.	5.00	22.9	3.42	22.3	0.746	22.0
	S-Avg.	3.54	38.8	2.72	35.7	0.623	34.2
	Ps-Avg.	5.73	20.6	3.49	21.5	0.746	22.0
	Pm	3.15	49.0	2.27	46.6	0.521	45.2
	Sm	2.69	55.7	1.99	54.5	0.463	51.8

[i] Improvement over 512KB cache (buffer) ([original value – new value]/[original value]).

Table 3.1: Performance with Read Caching.

the disk interface with the host. In other words, we need to configure our simulator with some small but non-zero amount of cache memory. Therefore, the improvement reported in Table 3.1 is relative to the performance with a small 512 KB cache. As discussed earlier, in the resource-poor environment, caching is relatively ineffective, achieving only about 6% improvement in average read response time and about 4% in average read service time. In the resource-rich environment, the improvement ranges from about 20% in the base PC workloads to more than 50% for the merged workloads.

Note that these numbers are for a cache block size of 4 KB. The sector or smallest addressable unit in most disks and storage controllers today is 512 B. Managing the caches at such a small granularity of 512 B requires a lot of control blocks, one for each cache block, many of which may have to be updated for a single operation. To reduce the number of control blocks needed, a cache block that is many times larger than the sector can be used together with a bit array in each control block to indicate whether a sector within the block is present in the cache. This is similar to the sector cache approach in processor cache. However, a larger cache block generally reduces cache effectiveness because of internal fragmentation. Additionally, the replacement information may not be as accurate because it is maintained at a coarser granularity. In Figure 3.5, we evaluate the impact of using a large cache block on the effectiveness of the cache. Observe that a cache block size of 4 KB is reasonable for our workloads. We will use this block size for the rest of the chapter. Note that the cache block size is the unit of cache management. It is independent of the fetch or transfer size, which we will analyze in the following section.

3.5.2 Prefetching

Prefetching is the technique of predicting blocks that are likely to be used in the future and fetching them before they are actually needed. The overall effectiveness of prefetching

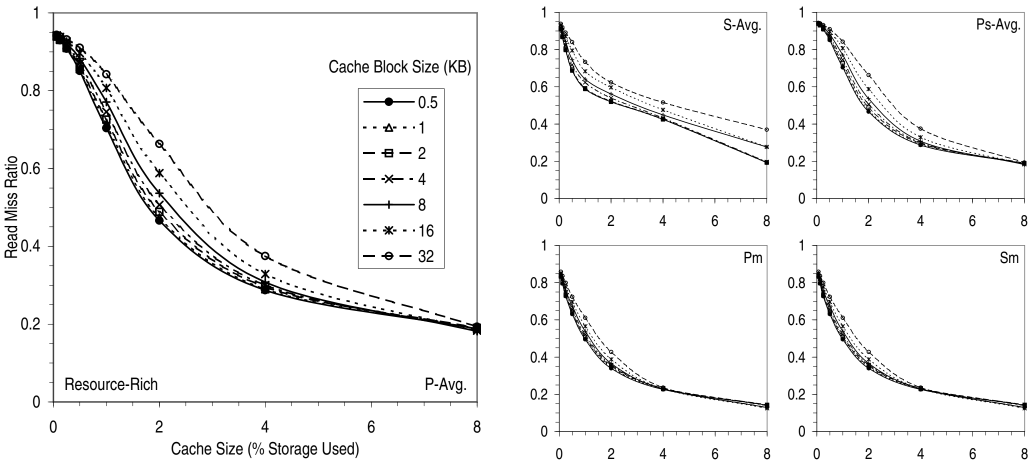


Figure 3.5: Sensitivity to Cache Block Size.

at improving performance hinges on (1) the accuracy of the prediction, (2) the amount of extra resources that are consumed by the prefetch, and (3) the timeliness of the prefetch, *i.e.,* whether the prefetch is completed before the blocks are actually needed.

The prediction is usually based on past access patterns [Smi78, HSY01b] although in certain situations, system-generated plans [TG84, HCL$^+$90], user-disclosed hints [PGG$^+$95] and guidance from speculative execution [CG99] may be available to help with the prediction. In general, the prediction is not perfect so that prefetching consumes more resources than demand fetching. Specifically, it congests the I/O system and may pollute memory with unused pages. Memory pollution is the loading of pages which are not referenced and the displacement of pages that will be referenced. However, for many storage devices, particularly disk drives, a large sequential access is much more efficient than multiple small random accesses. For such devices, prefetching of sequential pages has the potential to increase I/O efficiency by transforming several small block I/Os into one large block I/O, which can be more efficiently handled by the I/O device. Moreover, most workloads exhibit sequentiality in their I/O access patterns so that sequential prefetch, especially if performed on a cache miss, scores well on all three criteria (prediction accuracy, cost, timeliness) listed above. Therefore, practically all storage systems today implement some form of sequential prefetch on cache miss. We will focus on such prefetch in this chapter. By default, we assume that data is prefetched into the cache as if it is demand fetched. The prefetched data can also be placed in a separate buffer or be managed in the cache differently than demand fetched data. The interested reader is referred to [HSY01b] for an evaluation of such alternatives.

Recently, several researchers have proposed schemes for automatically matching up access patterns with previously observed contexts, and then prefetching according to the previously recorded reference patterns (*e.g.,* [GA94]). Such prefetching schemes should score well in the accuracy criteria but because they incur additional random I/Os, which are slow and inefficient, to perform the prefetch, they may not do as well in the cost and timeliness criteria. We will look at an alternative to context-based prefetch in the following chapter.

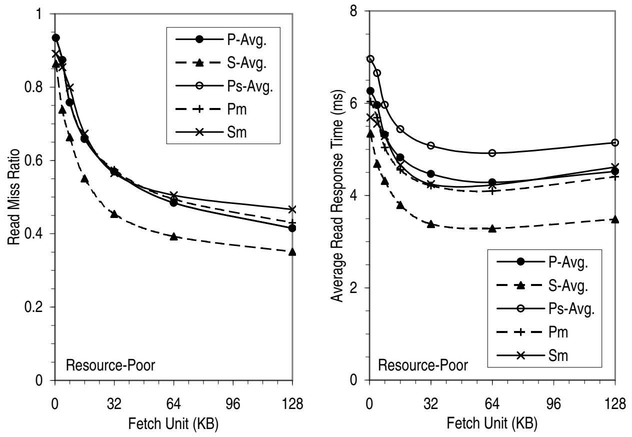
Large Fetch Unit

Sequential prefetch can be achieved relatively easily by using a *large fetch unit* or transfer size. For example, if the fetch unit is 64 sectors or blocks, a read request for blocks 60-68 will cause blocks 0-127 to be fetched. Notice that besides prefetching the blocks immediately following the requested data, a large fetch unit effectively results in the prefetch of some of the preceding blocks. Because these preceding blocks are fetched before the requested data to avoid having to wait for the preceding blocks to rotate under the disk head in the next revolution of the disk, there is a response time penalty for having a large fetch unit. Furthermore, we assume that the read is not considered complete until the entire fetch unit has been returned, although this could be avoided at the cost of taking an interrupt after the last target sector has arrived or by issuing a separate I/O to fetch the following blocks. We consider such techniques under read-ahead below. Note that using a large fetch unit is sometimes referred to as having a large block size.

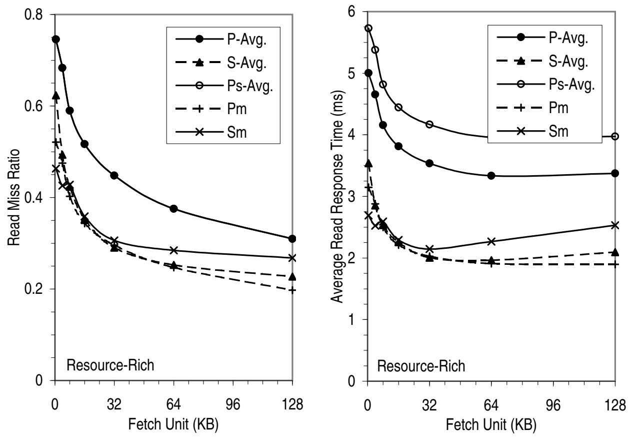
In Figure 3.6, we plot the effect of having a large fetch unit on the read miss ratio and the average read response time. Observe that a large fetch unit significantly reduces the read miss ratio, with most of the effect occurring at fetch units that are smaller than about 64 KB. As the fetch unit is increased beyond 64 KB, the average read response time starts to rise because the penalty of having to wait for the entire fetch unit begins to outweigh the benefit of the relatively small marginal improvement in read miss ratio. Previously, a one-track fetch unit was recommended [Smi85] but since then physical track sizes have grown from the 10 KB range to about 512 KB today. The ability of workloads to effectively use larger fetch units have not, however, kept pace. For all our workloads, a relatively small fetch unit of 64 KB or $\frac{1}{8}$ of a track works well.

Read-Ahead

In *read-ahead*, after the system has fetched the blocks needed to satisfy a read request, it continues to read the blocks following, *i.e.,* it reads ahead of the current request, hence its name. We consider the read request to be completed once all the requested blocks have been fetched. This typically means that two start I/Os are issued – one for the requested blocks and another to read ahead and prefetch data. In Figure 3.7, we explore the performance effect of reading ahead by various amounts. Observe from the figure that read-ahead of 32 KB performs well for all our workloads. Beyond 32 KB, the read response time begins to rise slightly for some of the workloads because the read-ahead is holding up subsequent demand requests, and the marginal improvement in read miss ratio at such large read-ahead amounts is not enough to overcome the effect of this delay. Later in this section, we will look at preempting the read-ahead whenever a demand request arrives.

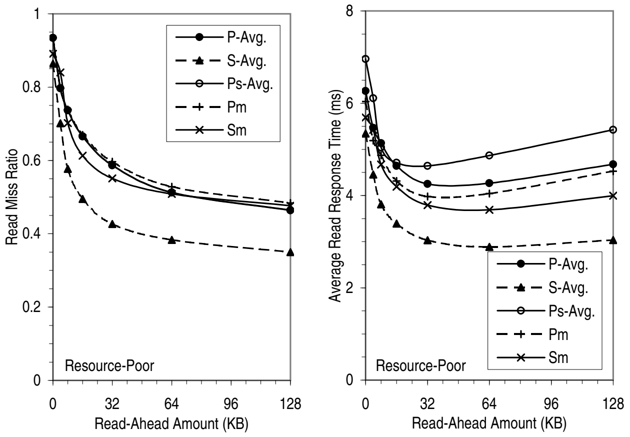


(a) Resource-Poor.

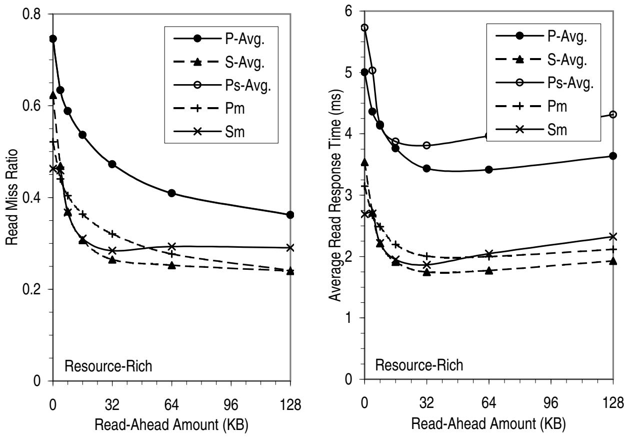


(b) Resource-Rich.

Figure 3.6: Effect of Large Fetch Unit on Read Miss Ratio and Response Time.



(a) Resource-Poor.



(b) Resource-Rich.

Figure 3.7: Effect of Read-Ahead on Read Miss Ratio and Response Time.

		Avg. Read Response Time						Avg. Read Service Time						Read Miss Ratio					
		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]	
		ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]		%[ii]		%[ii]		%[ii]
Resource-Poor	P-Avg.	4.29	32.1	4.25	32.4	4.14	34.3	2.74	36.2	2.99	30.4	2.93	31.9	0.484	48.2	0.587	37.2	0.518	44.6
	S-Avg.	3.29	39.3	3.03	44.5	2.75	49.6	2.27	41.3	2.26	41.9	2.08	46.5	0.393	54.4	0.427	50.6	0.383	55.6
	Ps-Avg.	4.92	29.8	4.64	33.4	4.65	33.4	2.78	35.8	2.84	34.4	2.81	35.0	0.484	48.2	0.587	37.2	0.518	44.6
	Pm	4.10	32.1	3.98	34.1	3.89	35.6	2.74	34.4	2.94	29.5	2.91	30.3	0.495	47.1	0.596	36.2	0.533	43.0
	Sm	4.23	25.7	3.79	33.4	3.61	36.6	3.13	23.6	3.07	25.3	2.93	28.6	0.505	43.3	0.551	38.1	0.523	41.3
Resource-Rich	P-Avg.	3.33	33.8	3.43	31.5	3.33	33.5	2.12	37.8	2.40	29.5	2.35	30.7	0.375	49.4	0.473	36.4	0.415	44.0
	S-Avg.	1.96	39.0	1.75	47.7	1.52	53.7	1.41	37.9	1.34	43.7	1.18	49.6	0.253	53.5	0.265	52.9	0.223	59.3
	Ps-Avg.	3.96	31.2	3.81	33.4	3.82	33.3	2.16	37.5	2.28	34.1	2.26	34.4	0.375	49.4	0.472	36.5	0.414	44.1
	Pm	1.91	39.3	2.01	36.3	1.94	38.3	1.33	41.2	1.56	31.3	1.54	32.3	0.247	52.6	0.321	38.5	0.280	46.3
	Sm	2.27	15.7	1.87	30.7	1.72	36.1	1.72	13.7	1.53	23.2	1.41	29.1	0.285	38.6	0.284	38.6	0.260	44.0

[i] LFU: Large fetch unit, RA: Read-Ahead, CSP: Conditional sequential prefetch
[ii] Improvement over no prefetch ([original value – new value]/[original value]).

Table 3.2: Performance Improvement with Prefetching.

In Table 3.2, we summarize the performance improvement of the different prefetching schemes. Observe that a large fetch unit tends to reduce the read miss ratio more than read-ahead does. It also has a slight advantage in read service time for the PC workloads. This is because the PC workloads tend to exhibit spatial locality and not just sequentiality. In other words, blocks that are near, not just those following, blocks that have been recently referenced are likely to be accessed in the near future. Thus a large fetch unit, by causing the blocks around the requested data to be prefetched, can achieve a higher hit ratio. However, because large fetch unit fetches the surrounding blocks before returning from servicing a request, it performs worse than read-ahead in terms of response time, especially for the server workloads.

Conditional Sequential Prefetch

To reduce resource wastage due to unnecessary prefetch, sequential prefetch can be initiated only when the access pattern is likely to be sequential. Generally, the amount of resources committed to prefetching should increase with the likelihood that the prediction is correct. For instance, previous studies [Smi78, HSY01b] have shown the benefit of determining the prefetch amount by conditioning on the length of the run or sequential pattern observed thus far. We refer to such schemes as *conditional sequential prefetch*. In order to condition on the run length, we need to be able to discover the sequential runs in the reference stream. This is generally difficult because of the complex interleaving of references from different processes. In this chapter, we use a general sequential detection scheme patterned after that proposed in [HSY01b].

The sequential detector keeps track of references at the granularity of multiple sectors or blocks, a unit we refer to as the *segment*. A segment is considered to be referenced if any page within that segment is referenced. By detecting sequentiality in segment references, we can very effectively capture pseudo-sequential reference patterns. The sequential detec-

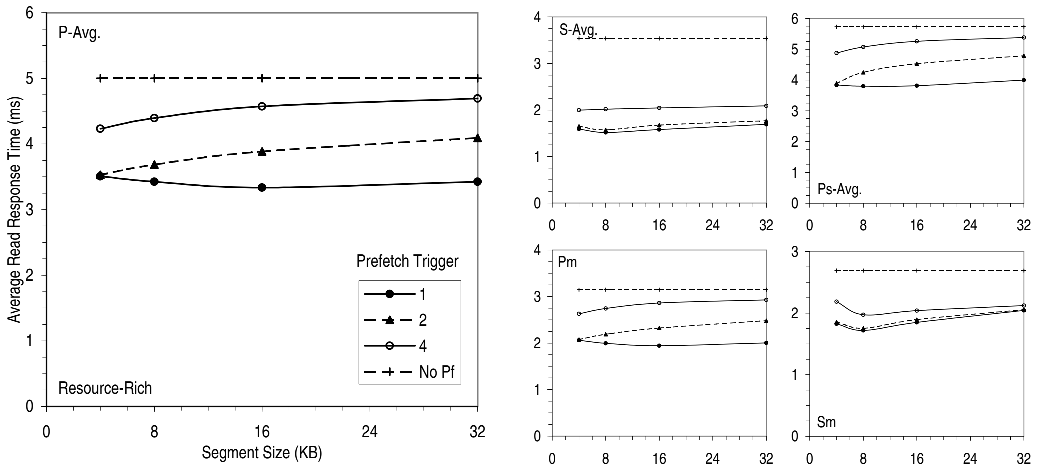


Figure 3.8: Read Response Time with Conditional Prefetch (Resource-Rich).

tor maintains a directory of segments using LRU replacement. Each entry in the segment directory has a run counter that tracks the length of the run ending at that segment. On a read, if the corresponding segment is not already in the segment directory, we insert it. The run counter value of the new segment entry is set to one if the preceding segment is not in the directory, and to one plus the counter value of the preceding segment otherwise. In the latter case, we remove the entry corresponding to the preceding segment. Note that the segment directory tracks sequential patterns in the actual reference stream. It is therefore updated only when read requests are encountered and not when blocks are prefetched. On a read miss, if the run counter for the segment exceeds a threshold known as the prefetch trigger, we initiate sequential prefetch. In this chapter, the prefetch amount is set to 2*(run counter value)*segment size, subject to a maximum of 256 KB. The size of the segment directory governs the number of potential sequential or pseudo-sequential streams that can be tracked by the sequential detector. We use a generous 64 entries for all our simulations.

In Figures 3.8, 3.9, C.2 and C.3, we explore the performance sensitivity to the segment size and the prefetch trigger. As we would expect, lower settings for the prefetch trigger perform better because the cost of fetching additional blocks once the disk head is properly positioned is minuscule compared to the cost of a random I/O that might have to be performed later if the blocks are not prefetched. For all the workloads, the best performance is obtained with a prefetch trigger of one, meaning that prefetch is triggered on every cache miss. A segment size of 16 KB works well for the PC workloads. For the server workloads, the optimal segment size is 8 KB.

In a similar fashion, we can additionally prefetch preceding blocks when a backward sequential pattern is detected. To prevent having to wait a disk revolution for the preceding blocks to appear under the disk head, we fetch the preceding blocks before the requested blocks. As shown in Figures C.4 and C.5, except for a slight performance improvement in some of the PC workloads, backward conditional prefetch turns out not to be very useful.

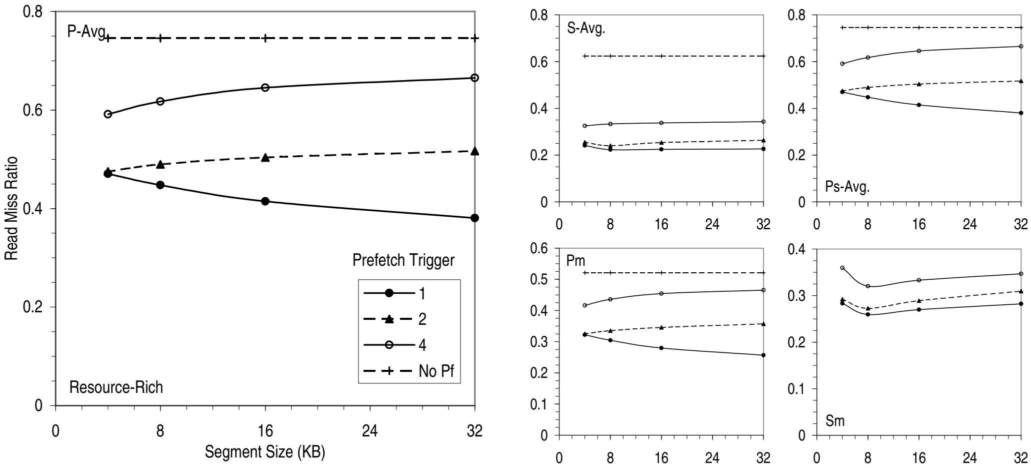


Figure 3.9: Read Miss Ratio with Conditional Prefetch (Resource-Rich).

In Table 3.2, we compare the performance of conditional sequential prefetch to that of large fetch unit and read-ahead. The three schemes achieve roughly the same average read response time for the PC workloads, reducing it by over 30%. For the server workloads, conditional sequential prefetch is clearly superior, improving the average read response time by between 36% and 54%. As for read service time, the PC workloads are improved by between 30 and 40% with large fetch unit having an edge. For the server workloads, conditional sequential prefetch again reigns supreme with improvement of between 29% and 50%. In the resource-poor environment, about 40-60% of the reads remain after caching and prefetching. In the resource-rich environment, about 25-45% remain.

Opportunistic Prefetch

Another way to reduce the potential negative impact of prefetch is to perform the prefetch using only resources that would otherwise be idle or wasted. We refer to this as opportunistic prefetch. In general, opportunistic prefetch can best be performed close to the physical device where detailed information is available about the critical physical resources. The net effect of prefetching, however, should be somewhat independent of which layer in the storage stack the data is prefetched into because the cost of a disk access is much higher than that of a semiconductor memory access. One exception is that data prefetched into the disk cache will tend to be evicted sooner, sometimes even before they are used, because the disk cache is typically smaller than the adaptor/controller cache. Therefore, in the resource-rich environment, we place opportunistically prefetched data into an 8 MB prefetch buffer. The prefetch buffer turns out to significantly reduce pollution of the large cache in the resource-rich environment.

The simplest form of opportunistic prefetch is to read-ahead until a demand request arrives at which point the read-ahead is terminated. This is known as preemptible read-ahead. Preemptible read-ahead may not be practical high up in the storage stack. For

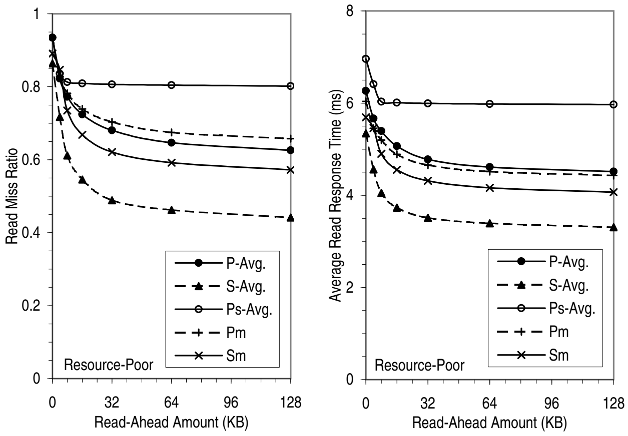
		Avg. Read Response Time						Avg. Read Service Time						Read Miss Ratio					
		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]	
		ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]		%[ii]		%[ii]		%[ii]
Preemptible Read-Ahead	P-Avg.	3.98	7.62	4.06	4.89	4.10	1.38	2.47	10.02	2.74	8.36	2.81	4.26	0.455	6.17	0.535	8.98	0.496	4.32
	S-Avg.	3.09	7.13	2.86	6.62	2.69	2.91	2.09	9.79	2.07	10.0	1.98	5.50	0.357	11.2	0.383	12.1	0.362	6.56
	Ps-Avg.	4.74	3.95	4.66	-0.203	4.70	-1.07	2.61	6.01	2.77	2.66	2.77	1.55	0.486	-0.288	0.567	3.39	0.509	1.67
	Pm	3.84	6.36	3.85	3.27	3.87	0.384	2.52	8.12	2.74	6.80	2.81	3.33	0.472	4.64	0.553	7.34	0.516	3.27
	Sm	4.01	5.19	3.64	4.14	3.54	1.91	2.97	5.17	2.88	5.92	2.82	3.86	0.478	5.33	0.511	7.32	0.500	4.40
+ Read Any Free Blocks[iii]	P-Avg.	3.83	11.3	3.34	22.3	3.42	18.2	2.37	13.8	2.22	25.9	2.31	21.4	0.433	10.8	0.431	26.8	0.404	22.2
	S-Avg.	3.03	9.1	2.67	13.6	2.52	9.38	2.05	11.3	1.91	17.4	1.84	13.0	0.350	13.1	0.349	20.1	0.332	14.7
	Ps-Avg.	4.55	8.03	3.83	18.0	3.96	15.4	2.48	10.7	2.18	23.2	2.25	20.2	0.460	5.2	0.450	23.4	0.412	20.5
	Pm	3.69	9.9	3.14	21.1	3.21	17.5	2.42	11.7	2.23	24.4	2.32	20.4	0.451	8.8	0.447	25.0	0.422	20.8
	Sm	3.95	6.64	3.37	11.2	3.30	8.68	2.92	6.6	2.67	12.7	2.62	10.7	0.468	7.34	0.468	15.1	0.460	12.1
+ Just-in-Time Seek[iv]	P-Avg.	3.77	12.7	3.45	19.8	3.62	13.5	2.08	24.4	1.96	34.4	2.13	27.4	0.404	17.0	0.413	29.9	0.399	23.3
	S-Avg.	2.99	10.2	2.64	14.8	2.52	9.12	1.84	21.2	1.66	29.1	1.61	24.5	0.340	15.4	0.336	23.9	0.327	16.5
	Ps-Avg.	4.45	9.9	4.06	13.4	4.27	8.76	1.95	29.8	1.91	32.8	2.10	25.4	0.418	13.8	0.433	26.4	0.410	21.0
	Pm	3.61	12.0	3.26	18.1	3.43	11.9	2.06	24.9	1.93	34.5	2.12	27.3	0.422	14.7	0.434	27.3	0.420	21.3
	Sm	3.92	7.45	3.34	11.9	3.28	9.10	2.64	15.6	2.37	22.6	2.32	20.7	0.456	9.8	0.459	16.8	0.455	13.1

[i] LFU: Large fetch unit, RA: Read-Ahead, CSP: Conditional sequential prefetch
[ii] Improvement over non-opportunistic prefetch ([original value – new value]/[original value]).
[iii] Preemptible Read-Ahead + Read Any Free Blocks.
[iv] Preemptible Read-Ahead + Read Any Free Blocks + Just-in-Time Seek.

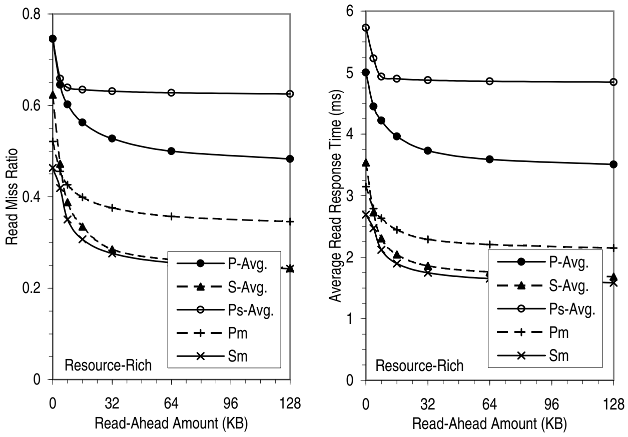
Table 3.3: Additional Effect of Opportunistic Prefetch (Resource-Poor).

example, read-ahead by the disk is usually preemptible. But at the adaptor/controller level, once the request is issued to the disk, it is difficult to cancel. By terminating the read-ahead as soon as another demand request arrives, preemptible read-ahead avoids holding up subsequent requests. Thus its performance does not degrade as the maximum read-ahead amount is increased (Figure 3.10). However, preemptible read-ahead tends not to perform as well as non-preemptible read-ahead, especially for the sped-up workloads, because it may get preempted before it can perform any effective prefetch. Such results suggest a hybrid approach of performing preemptible read-ahead in addition to the non-opportunistic prefetching schemes discussed above. We find that with the hybrid approach, an opportunistic prefetch limit of 128 KB works well in almost all the cases (Figures C.7 - C.12). This is the value that we will assume for the rest of the chapter. An opportunistic prefetch limit of 128 KB means that blocks will only be opportunistically prefetched until a total of 128 KB of data has been prefetched.

In Tables 3.3 and 3.4, we summarize the performance impact of augmenting the various non-opportunistic prefetching schemes with preemptible read-ahead. Notice that in the resource-poor environment, preemptible read-ahead improves average read response time by about 5% for large fetch unit and read-ahead. The improvement is less for conditional sequential prefetch because conditional sequential prefetch already uses resources carefully by determining the amount to prefetch based on how likely the prefetch will be useful. In the resource-rich environment, preemptible read-ahead has a bigger effect, especially for the server workloads which are improved by about 15-20%.



(a) Resource-Poor.



(b) Resource-Rich.

Figure 3.10: Effect of Preemptible Read-Ahead on Read Miss Ratio and Response Time.

		Avg. Read Response Time						Avg. Read Service Time						Read Miss Ratio					
		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]		LFU[i]		RA[i]		CSP[i]	
		ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]	ms	%[ii]		%[ii]		%[ii]		%[ii]
Preemptible Read-Ahead	P-Avg.	2.97	11.9	3.13	10.2	3.15	6.84	1.81	14.9	2.10	13.1	2.15	9.25	0.336	10.8	0.411	13.7	0.381	8.95
	S-Avg.	1.71	15.2	1.45	18.8	1.32	13.5	1.20	17.7	1.04	22.8	0.98	16.4	0.212	19.1	0.204	23.7	0.186	16.1
	Ps-Avg.	3.76	5.40	3.65	4.99	3.69	4.49	1.99	7.79	2.13	7.24	2.13	6.43	0.371	1.15	0.437	8.07	0.391	6.28
	Pm	1.67	12.3	1.96	2.46	1.98	-2.15	1.14	14.6	1.45	6.99	1.50	2.25	0.223	9.90	0.296	7.75	0.275	1.60
	Sm	2.05	9.8	1.44	22.8	1.38	19.6	1.56	9.47	1.14	25.4	1.10	22.1	0.257	9.79	0.212	25.3	0.205	21.2
+ Read Any Free Blocks[iii]	P-Avg.	2.76	18.3	2.57	26.7	2.66	22.2	1.68	20.8	1.71	29.6	1.79	25.0	0.310	17.8	0.332	30.5	0.313	25.6
	S-Avg.	1.65	18.7	1.32	27.4	1.20	22.6	1.16	20.8	0.947	31.3	0.886	25.7	0.204	22.6	0.182	32.8	0.167	26.0
	Ps-Avg.	3.47	13.1	3.03	22.1	3.15	19.3	1.81	16.3	1.69	26.7	1.75	23.8	0.336	10.7	0.348	27.0	0.319	24.0
	Pm	1.57	17.8	1.59	20.9	1.65	15.0	1.07	19.9	1.17	25.3	1.24	19.4	0.207	16.0	0.238	25.8	0.226	19.4
	Sm	1.99	12.4	1.44	22.8	1.38	19.9	1.52	11.8	1.15	24.7	1.10	22.2	0.249	12.6	0.212	25.3	0.204	21.3
+ Just-in-Time Seek[iv]	P-Avg.	2.71	19.9	2.66	24.3	2.80	18.0	1.50	29.7	1.52	37.4	1.65	30.7	0.287	24.1	0.318	33.6	0.308	26.8
	S-Avg.	1.63	20.1	1.29	29.5	1.21	22.1	1.04	29.7	0.800	42.6	0.769	36.0	0.198	24.7	0.171	37.5	0.162	28.5
	Ps-Avg.	3.38	15.5	3.20	17.8	3.37	13.5	1.46	32.9	1.49	35.4	1.64	28.5	0.303	19.9	0.335	30.0	0.317	24.5
	Pm	1.54	19.3	1.65	17.8	1.76	9.31	0.927	30.5	1.01	35.1	1.13	26.4	0.190	22.9	0.229	28.7	0.223	20.1
	Sm	1.97	13.0	1.42	24.1	1.33	22.4	1.38	20.0	1.01	34.4	0.94	33.3	0.242	14.9	0.204	28.1	0.194	25.1

[i] LFU: Large fetch unit, RA: Read-Ahead, CSP: Conditional sequential prefetch
[ii] Improvement over non-opportunistic prefetch ([original value – new value]/[original value]).
[iii] Preemptible Read-Ahead + Read Any Free Blocks.
[iv] Preemptible Read-Ahead + Read Any Free Blocks + Just-in-Time Seek.

Table 3.4: Additional Effect of Opportunistic Prefetch (Resource-Rich).

Another opportunistic prefetching technique is to start reading once the disk head is positioned over the correct track. This may prefetch some blocks before the requested data and/or some blocks after, depending on when the head is properly positioned. Such a scheme is known as *read any free blocks* or *zero latency read*. Basically, it uses the rotational delay to perform some prefetching for free. As shown in Tables 3.3 and 3.4, read any free blocks is quite effective at improving performance. In the resource-poor environment, read any free block with preemptible read-ahead is able to reduce the average read response time with read-ahead by about 20% for the PC workloads and over 10% for the server workloads. In the resource-rich environment, the additional improvement is over 20% for all the workloads. Again, conditional sequential prefetch is improved less because it performs large prefetches only when they are warranted. As for large fetch unit, it is improved the least by read any free blocks because it already prefetches some of the preceding blocks.

The dual of read any free blocks is *just-in-time seek* or *delayed preemption* [GW02]. The idea here is that when a request arrives while the disk is performing preemptible read-ahead, the disk should continue with the read-ahead and move the head to service the incoming request only in time for the head to be positioned over the correct track before the requested data rotates under. Basically, this allows the disk to prefetch more of the succeeding blocks. As shown in Tables 3.3 and 3.4, for large fetch unit, the additional use of just-in-time seek improves performance slightly over performing only read any free blocks and preemptible read-ahead. For read-ahead and conditional sequential prefetch, just-in-time seek offers a marginal performance improvement on top of read any free blocks and

		Resource-Poor									Resource-Rich								
		Avg. Read Resp. Time			Avg. Read Service Time			Read Miss Ratio			Avg. Read Resp. Time			Avg. Read Service Time			Read Miss Ratio		
		LFU[i]	RA[i]	CSP[i]	LFU[i]	RA[i]	CSP[i]	LFU[i]	RA[i]	CSP[i]	LFU[i]	RA[i]	CSP[i]	LFU[i]	RA[i]	CSP[i]	LFU[i]	RA[i]	CSP[i]
Preemptible Read-Ahead	P-Avg.	37.1	35.6	35.1	42.5	36.2	34.7	51.3	42.8	46.9	41.5	38.5	38.1	46.9	38.8	37.2	54.9	45.1	49.1
	S-Avg.	43.2	47.8	50.8	45.8	46.8	48.9	58.5	55.8	58.0	46.6	57.9	61.1	45.9	57.2	59.5	60.0	64.4	66.9
	Ps-Avg.	32.5	33.2	32.7	39.6	36.1	36.0	48.0	39.3	45.5	34.81	36.8	36.4	42.3	38.9	38.7	50.0	41.6	47.7
	Pm	36.4	36.3	35.8	39.7	34.3	32.6	49.5	40.9	44.8	46.8	37.8	37.0	49.8	36.1	33.9	57.3	43.2	47.2
	Sm	29.5	36.1	37.8	27.6	29.7	31.4	46.3	42.7	43.9	24.0	46.4	48.7	21.9	42.7	44.7	44.6	54.1	55.8
+ Read Any Free Blocks[ii]	P-Avg.	39.5	47.3	46.1	44.9	48.3	46.3	53.7	53.9	56.8	45.7	49.7	48.2	50.6	50.4	48.1	58.3	55.8	58.4
	S-Avg.	44.3	51.7	54.2	46.8	51.1	52.9	59.4	59.8	61.7	48.7	61.8	64.6	47.9	61.0	63.2	61.7	68.0	70.2
	Ps-Avg.	35.3	45.3	43.5	42.6	49.5	48.0	50.8	51.8	55.9	40.0	48.0	46.1	47.5	51.7	50.0	54.8	53.6	57.5
	Pm	38.8	48.0	46.8	42.1	46.7	44.5	51.7	52.1	54.8	50.1	49.6	47.6	52.9	48.6	45.5	60.2	54.4	56.7
	Sm	30.6	40.8	42.1	28.7	34.8	36.2	47.5	47.5	48.4	26.2	46.5	48.8	23.9	42.2	44.8	46.3	54.2	55.9
+ Just-in-Time Seek[iii]	P-Avg.	40.5	45.6	42.9	51.7	54.3	50.5	56.8	55.8	57.3	46.7	48.1	45.4	56.1	55.9	52.0	61.5	57.7	59.0
	S-Avg.	45.0	52.3	54.1	52.2	57.4	58.6	60.5	61.3	62.4	49.4	62.6	64.5	52.7	66.5	67.7	62.8	69.8	71.0
	Ps-Avg.	36.6	42.1	39.0	54.9	55.9	51.4	55.2	53.7	56.1	41.7	45.2	42.2	58.0	57.5	53.2	59.4	55.5	57.8
	Pm	40.3	46.1	43.2	50.7	53.9	49.3	54.8	53.6	55.1	51.0	47.6	44.0	59.2	55.4	50.2	63.5	56.1	57.1
	Sm	31.2	41.3	42.3	35.5	42.2	43.4	48.8	48.5	49.0	26.7	47.4	50.4	30.9	49.6	52.7	47.7	55.9	58.0

[i] LFU: Large fetch unit, RA: Read-Ahead, CSP: Conditional sequential prefetch
[ii] Preemptible Read-Ahead + Read Any Free Blocks.
[iii] Preemptible Read-Ahead + Read Any Free Blocks + Just-in-Time Seek.

Table 3.5: Overall Effect of Opportunistic and Non-Opportunistic Prefetch. Table shows percentage improvement over a system that does not prefetch.

preemptible read-ahead for the server workloads, but loses out for the PC workloads. During the rotational delay, the disk can also be used to perform lower-priority or background I/Os in what is known as freeblock scheduling [LSG02]. For instance, if the next block to be read is halfway round the track, the disk head could be positioned to service a background request before being moved back in time to read the block as it rotates under the head. But given that read any free blocks and just-in-time seek are effective at improving performance, such background I/Os may not be totally free for our workloads.

In general, in both the resource-poor and resource-rich environments, read-ahead with preemptible read-ahead and read any free blocks performs the best for the PC workloads, improving average read response time by almost 50% over the case when there is no prefetch (Table 3.5). For the server workloads, conditional sequential prefetch with preemptible read-ahead and read any free blocks offers performance improvement of between 42% and 54% in the resource-poor environment and up to 65% in the resource-rich environment.

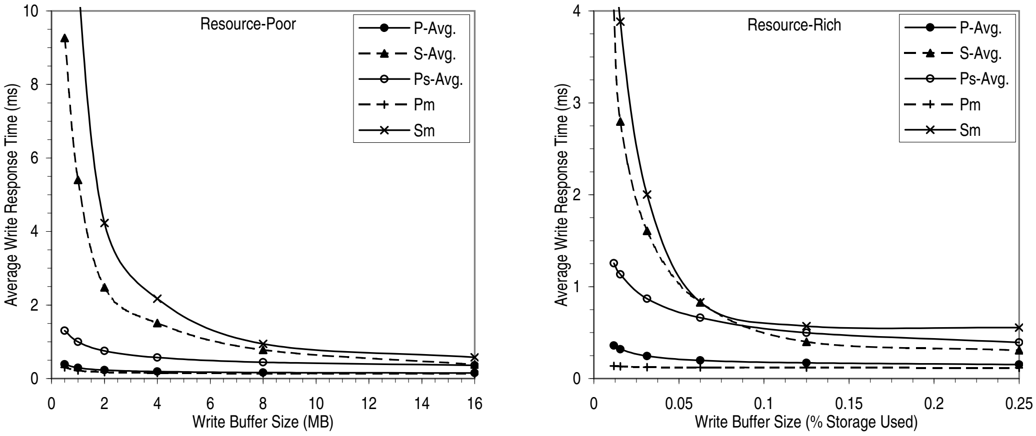
3.5.3 Write Buffering

Write buffering refers to the technique of holding written data temporarily in fast, typically semiconductor, memory before destaging the data to permanent storage. A write operation can be reported as completed once its data has been accepted into the buffer. Because writes tend to come in bursts [Chapter 2], the write buffer helps to better regulate the flow of data to permanent storage. To prevent any loss of data if the system fails before the buffered data is hardened to permanent storage, the write buffer is typically implemented with some form of non-volatile storage (NVS). In some environments, (*e.g.,* UNIX file system, PC disks), a less expensive approach of periodically flushing (usually every 30s) the buffer contents to disk is considered sufficient. By delaying when the written data is destaged to permanent storage, write buffering allows multiple writes to the same location to be combined into a single physical write, thereby reducing the number of physical writes that have to be performed by the system. It may also increase the efficiency of writes by allowing multiple consecutive writes to be merged into a single big-block I/O. In addition, more sophisticated techniques can be used to schedule the writes to take advantage of the characteristics and the state of the storage devices.

In short, the write buffer achieves three main effects. First, it hides the latency of writes by deferring them to some later time. Second, it reduces the number of physical writes, and third, it enables the remaining physical writes to be performed efficiently. In this chapter, we evaluate write buffering using a general framework that is flexible enough for us to examine the three effects of write buffering separately. In this framework, a background destage process is initiated whenever the *fraction* of dirty blocks in the write buffer exceeds a high limit threshold, $highMark$. This ensures that buffer space is available to absorb the incoming writes. To avoid impacting the read response time, destage requests are not serviced unless there are no pending read requests or the write buffer is full. In the latter case, destage requests are serviced at the same priority as the reads. Analysis in the prior chapter shows that the I/O workload is bursty, which implies that the storage system has idle periods during which the destage requests can be handled.

To reduce the number of physical writes, we use the Least-Recently-Written (LRW) policy to decide which blocks to destage [HSY01b]. The LRW policy is similar to the LRU policy for read caching and is so named because it selects for destage the block that was least recently written. In order to examine the effect of limiting the age of dirty data in the buffer, we also destage a block when its age exceeds the maximum allowed. Destage policies have been studied in some detail recently but the focus has been on selecting blocks to destage based on how efficiently buffer space can be reclaimed. For instance, in [BRT93], the track with the most dirty blocks is selected for destage. In [VJ98], the blocks that can be written most quickly are selected. But a destage policy that strives to quickly reclaim buffer space may not be effective if the blocks that are destaged will be dirtied again in the near future. Moreover, with the layered approach of building systems, estimates of the cost of destage operations may not be available to the destage process. For instance, the adaptor or controller housing the write buffer typically has no accurate knowledge of the state and geometry of the underlying disks.

The approach we take is to focus on reducing the number of physical writes, and to rely on request scheduling to perform the remaining writes efficiently. We achieve the latter by ensuring that there is a sizeable number of outstanding destage requests to be scheduled. Specifically, we allow as many outstanding destage requests as the maximum queue depth seen by the host, and once the destage process is initiated, it stops only when the *fraction* of dirty blocks in the buffer drops below a low limit threshold, $lowMark$. By



(a) Resource-Poor. (b) Resource-Rich.

Figure 3.11: Improvement in Average Write Response Time from Absorbing Write Bursts.

setting $lowMark$ to be significantly lower than $highMark$, we achieve a hysteresis effect which prevents the destage process from being constantly triggered whenever new blocks become dirty. Therefore, instead of a continual trickle of destage requests, we periodically get a burst of destage requests which can be effectively scheduled. To further increase the efficiency of the destage operations, whenever a destage request is issued, we include in the same request contiguous blocks that are also dirty. The resulting disk write may span tracks but it is a large sequential write which can be efficiently handled by the disk.

The use of hysteresis in the destage process has been previously proposed in [BRT93, VJ98] but for a different purpose. Because the destage algorithms suggested there do not take into account the age of the blocks, the same blocks may be selected for destage every time the algorithms are run. For instance, if the destage algorithm selects the track with the most dirty blocks, a track that is continuously written will tend to be selected every time the destage process is initiated. In such situations, hysteresis, by preventing the destage process from being constantly triggered, helps to reduce the chances that the same blocks will be repeatedly destaged.

Absorbing Write Bursts

To investigate the amount of buffer space needed to absorb the write bursts, we set both the $highMark$ and $lowMark$ to zero. This ensures that dirty blocks are destaged at the earliest opportunity to make room for buffering the incoming writes. In Figure 3.11, we plot the average write response time as a function of the buffer size. In order to generalize our results across the different workloads, we also normalize the buffer size to the amount of storage used.

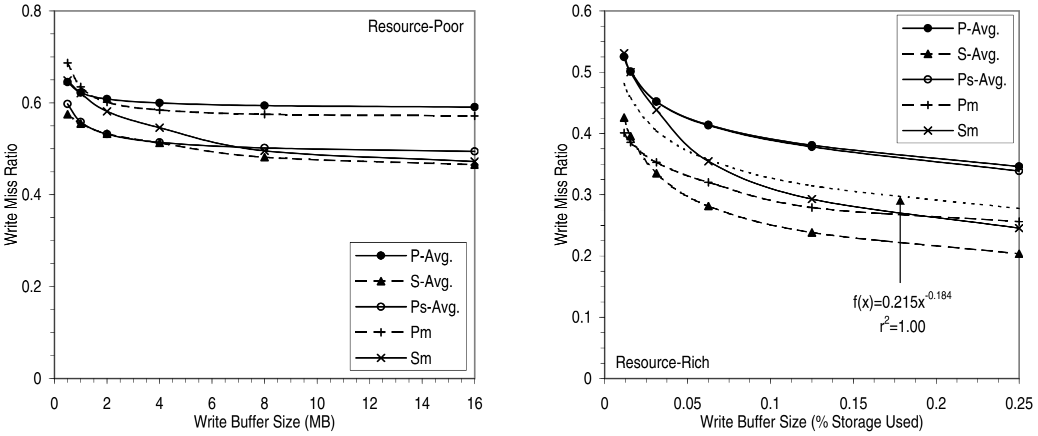
When the write buffer is not large enough to absorb the write bursts, some of the writes will stall until buffer space is reclaimed by destaging some of the dirty blocks. When the buffer is large enough, all the write requests can be completed without stalling. Notice that for all the workloads, a write buffer of between 4 MB and 8 MB or between 0.05 and 0.1% of the storage used is sufficient to effectively absorb the write bursts. In fact, for the PC workloads, a small write buffer of about 1 MB or 0.01% of the storage used is able to hide most of the write latency. As in the case of the read cache, we investigated the effect of different buffer block sizes and found that 4 KB is reasonable for our workloads (Figure C.13).

Eliminating Repeated Writes

As mentioned earlier, when data is updated again before it is destaged, the second update effectively cancels out the previous update, thereby reducing the number of physical writes to the storage system. In this section, we focus on how much buffer space is needed to effectively allow repeated writes to the same location to be cancelled. We set the $highMark$ and $lowMark$ to one so as to maximize the probability that a write will "hit" in the write buffer.

In Figure 3.12, we plot the write miss ratio as a function of the buffer size. We define the write miss ratio as the fraction of write requests that causes one or more buffer blocks to become dirty. Thus the write miss ratio is essentially the fraction of write requests that are not cancelled. As in the case of the read cache, we took the arithmetic mean of the plots for the five different classes of workloads and fitted various functional forms to it. As shown in Figure 3.12(b), a power function of the form $f(x) = a(x - b)^c$ is again a good fit. However, the exponent c at about -0.2 is significantly bigger than it is for reads, meaning that for large buffer sizes, the write miss ratio decreases much more slowly with buffer size increase than is the case for reads. Such a behavior of the physical I/O stream turns out to parallel what has been observed at the logical level for large database systems where the read and write exponents are about -0.5 and -0.25 respectively [HSY01b].

Observe from Figure 3.12(b) that for all the workloads, 60-75% of the writes are eliminated at buffer sizes that are less than 0.1% of the storage used. In Figure C.14, we plot the corresponding improvement in the average write service time. In the resource-poor environment, we limit the age of dirty blocks in the buffer to be less than 30s. There is, therefore, less write cancellation (about 40-50%) and most of it occurs at very small buffer sizes of about 2 MB. In general, when there is concern about losing buffered data, limits have to be placed on the maximum age of the buffered data. In Figure 3.13, we analyze the effect of such constraints and find that a maximum age of 1-hour is able to achieve most of the write elimination.



(a) Resource-Poor. (b) Resource-Rich.

Figure 3.12: Effectiveness of Write Buffering at Reducing Physical Writes.

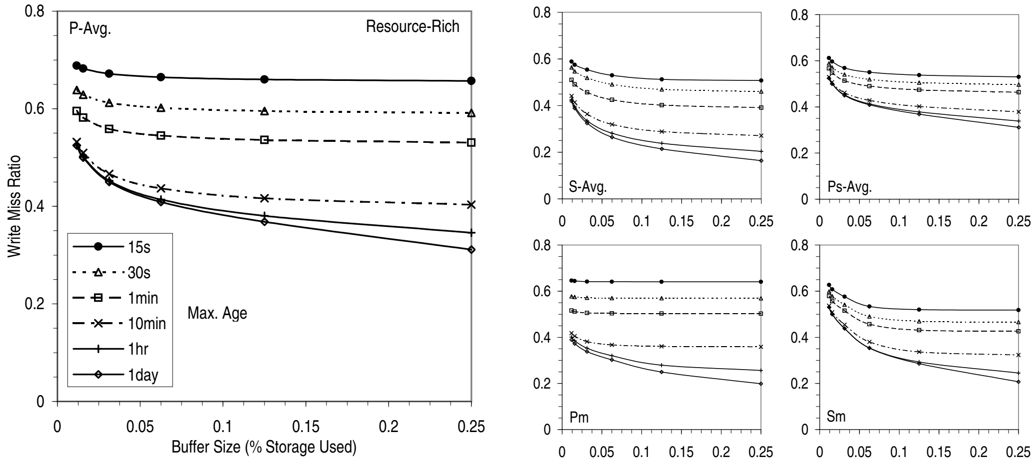


Figure 3.13: Sensitivity to Maximum Dirty Age.

Combined Effect

We have studied the effects of absorbing write bursts and eliminating repeated writes independent of each other. In practice, the two effects compete for buffer space. They also work together because eliminating writes makes it possible to absorb write bursts in less buffer space. Striking a balance between the two is therefore key to effective write buffering. In this section, we investigate how to achieve this balance by appropriately setting the $highMark$ and $lowMark$ threshold values.

In Figures 3.14 and C.15, we plot the average write response time as a function of $highMark$. The corresponding plots for the write miss ratio are presented in Figures 3.15

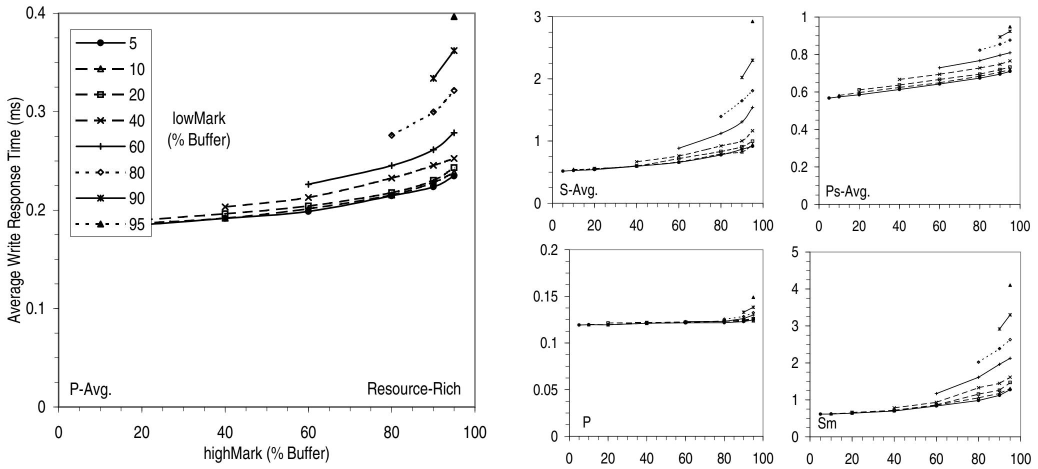


Figure 3.14: Effect of $lowMark$ and $highMark$ on Average Write Response Time (Resource-Rich).

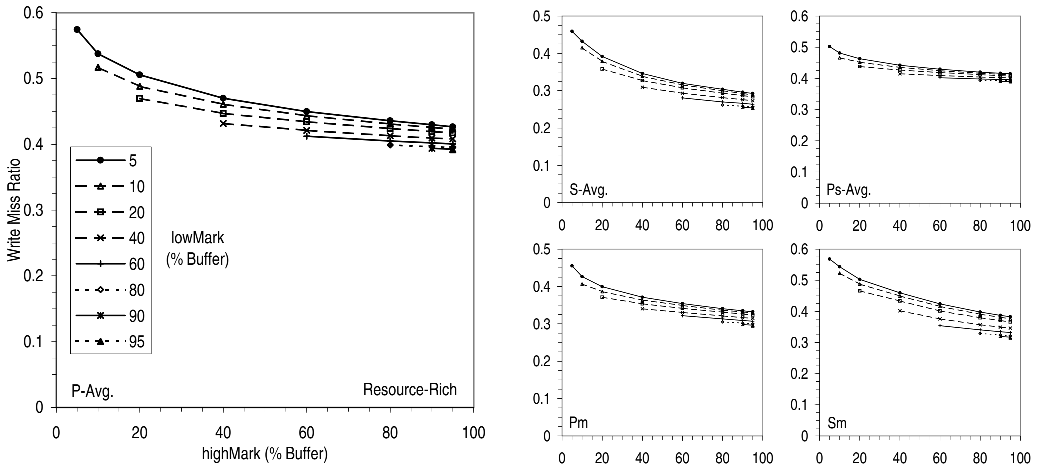


Figure 3.15: Effect of $lowMark$ and $highMark$ on Write Miss Ratio (Resource-Rich).

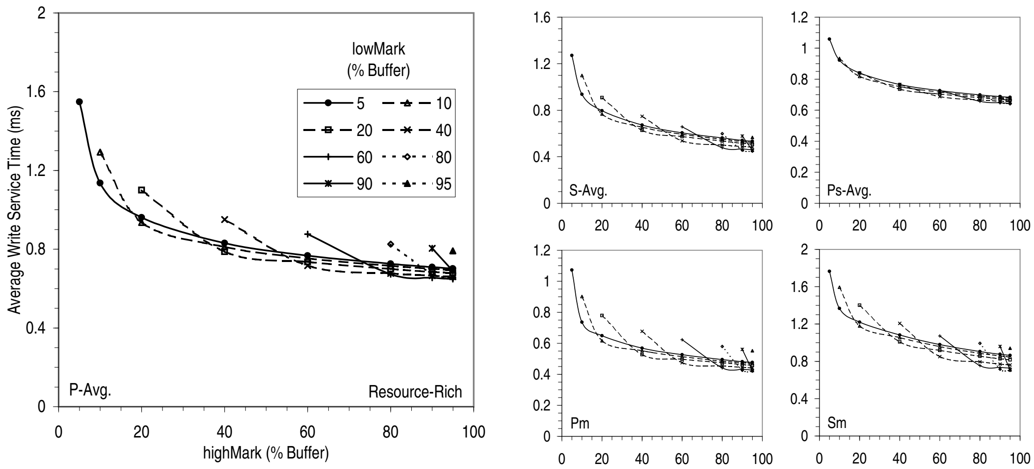


Figure 3.16: Effect of $lowMark$ and $highMark$ on Average Write Service Time (Resource-Rich).

and C.16. As we would expect, if destage is initiated only when most of the buffer is dirty, some of the writes will stall until buffer space becomes available and this will result in large average write response time. On the other hand, if destage is initiated at a small $highMark$ value, there will be less opportunities for write cancellation. For all the workloads, we find that a $highMark$ value of 0.8 and a $lowMark$ value of 0.2 strikes a good compromise. The $highMark$ and $lowMark$ settings also affect how efficiently the destage operations can be performed. From Figures 3.16 and C.17, we find that setting $lowMark$ to significantly less than $highMark$ is essential to allowing the destage operations to be scheduled effectively.

A concern with background destage operations is that they may negatively impact the read response time. For instance, when the write buffer becomes full, background destage requests become foreground operations which may interfere with the incoming read requests. Moreover, the first read request after an idle period may encounter a destage in progress. In this study, we assume that destage operations are not preemptible. This is generally true at the adaptor/controller level because a write request cannot be easily cancelled once it has been issued to the disk. From Figures 3.17 and C.18, we find that the read response time is not significantly affected by write buffering provided that there is some hysteresis, that is $lowMark$ is significantly lower than $highMark$. When there is no hysteresis, destage operations take longer and tend to occur after every write request, thereby increasing the chances that a read will be blocked. In addition, the constant trickle of destage operations may lead to disk head thrashing because the locality of reference for destage operations, which are essentially delayed writes, is not likely to coincide with that of current read requests.

In Table 3.6, we summarize the performance benefit of write buffering. In the resource-poor environment, about 40-50% of the writes are eliminated by write buffering. The average write service time is reduced by between 60-80% while the average write response

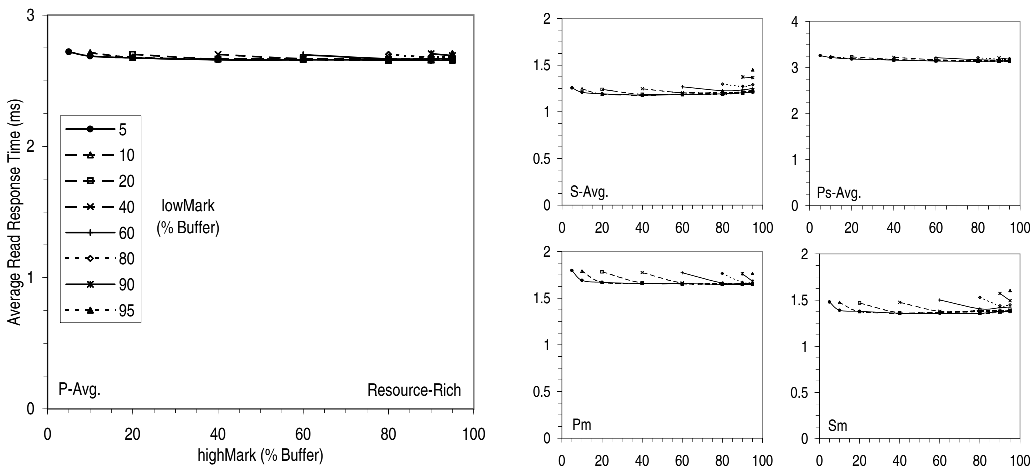


Figure 3.17: Effect of $lowMark$ and $highMark$ on Average Read Response Time (Resource-Rich).

		Write					
		Avg. Resp. Time		Avg. Serv. Time		Miss Ratio	
		ms	%[i]	ms	%[i]		%[i]
Resource-Poor	P-Avg.	0.227	96.9	1.41	70.9	0.606	39.4
	S-Avg.	2.13	92.7	1.32	70.7	0.525	47.3
	Ps-Avg.	0.646	91.6	1.05	78.2	0.520	47.9
	Pm	0.190	97.7	1.30	74.2	0.598	40.1
	Sm	3.48	90.1	1.57	65.2	0.572	42.7
Resource-Rich	P-Avg.	0.218	97.0	0.700	85.6	0.424	57.6
	S-Avg.	0.831	97.0	0.535	87.8	0.293	70.6
	Ps-Avg.	0.695	90.9	0.681	85.9	0.412	58.8
	Pm	0.123	98.5	0.474	90.5	0.332	66.8
	Sm	1.16	96.7	0.855	81.0	0.380	62.0

[i] Improvement over no write buffer ([original value – new value]/[original value]).

Table 3.6: Performance with Write Buffering.

time is reduced by more than 90%. The improvement in the resource-rich environment is even more significant, with about 60-70% of the writes being eliminated, and as much as a 90% reduction in the average write service time.

3.5.4 Request Scheduling

The time required to satisfy a request depends on the state of the disk, specifically whether the requested data is present in the cache and where the disk head is relative to the requested data. In request scheduling [Den67], the order in which requests are handled is optimized to improve performance. Request scheduling can be performed at different levels in the storage stack. For instance, at the level of the operating system, device driver, disk adaptor or even the disk itself. Request scheduling by the disk used to be a feature of high-end server disks but is beginning to appear in disks that are going into PCs. One of the aims of this section is to quantify the actual benefit such scheduling will bring to PC workloads.

For request scheduling to be effective, the scheduler needs to have good estimates of the service time of different requests. Such estimates are difficult to make high in the storage stack because little information is available there. For example, modern disk protocols (*e.g.,* SCSI, IDE) present a flat address space so that any level above the disk has little correct knowledge of the physical geometry of the disk. In addition, it is hard to predict which requests will hit in the onboard disk cache. As we have seen in the previous sections, there are a lot of hits in the onboard disk cache, and such hits can substantially affect the effectiveness of request scheduling [WGP94]. In this chapter, we schedule only requests that miss in the cache, since the critical resource is the disk arm and what we really want to do is arm scheduling.

Our experiments are based on the scheduling algorithm that has been variously referred to as Shortest Time First [SCO90], Shortest Access Time First [JW91] and Shortest Positioning Time First [WGP94]. This is a greedy algorithm that always selects the pending request with the smallest estimated access time (seek + rotational latency). By selecting the request with the shortest access time, the algorithm tries to reduce the amount of time the disk arm spends positioning itself, thereby increasing the effective utilization of the critical resource. The algorithm can be adapted to select the request with the shortest service time so as to minimize waiting time. In order to reduce the chances of request starvation, the requests can be aged by subtracting from each access time or positioning delay (T_{pos}) a weighted value corresponding to the amount of time the request has been waiting for service (T_{wait}). The resulting effective positioning delay (T_{eff}) is used in selecting the next request:

$$T_{eff} = T_{pos} - (W * T_{wait}) \tag{3.1}$$

We refer to this variation of the algorithm as Aged Shortest Access Time First (ASATF).

		Average Response Time						Average Service Time					
		All Requests		Reads		Writes		All Requests		Reads		Writes	
		ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]
Resource-Poor	P-Avg.	1.51	14.2	3.34	12.8	0.227	24.6	1.74	10.8	2.22	4.03	1.41	16.5
	S-Avg.	2.39	26.0	2.67	18.7	2.13	33.0	1.57	19.0	1.91	3.13	1.32	30.6
	Ps-Avg.	1.96	19.0	3.83	14.4	0.646	31.1	1.52	16.4	2.18	6.08	1.05	27.1
	Pm	1.24	14.4	3.14	12.6	0.190	28.1	1.63	15.4	2.23	4.24	1.30	23.8
	Sm	3.43	34.8	3.37	15.1	3.48	44.7	2.06	18.8	2.67	2.71	1.57	33.6
Resource-Rich	P-Avg.	1.19	13.5	2.66	11.6	0.218	25.4	1.14	18.1	1.79	3.40	0.700	34.7
	S-Avg.	1.00	22.3	1.20	13.3	0.831	22.6	0.689	21.6	0.886	2.68	0.535	40.9
	Ps-Avg.	1.67	18.9	3.15	12.4	0.695	32.0	1.11	19.1	1.75	5.05	0.681	35.1
	Pm	0.67	7.71	1.65	8.06	0.123	5.07	0.746	19.8	1.24	2.99	0.474	35.8
	Sm	1.253	39.4	1.38	13.2	1.16	52.8	0.963	25.4	1.10	2.32	0.855	39.8

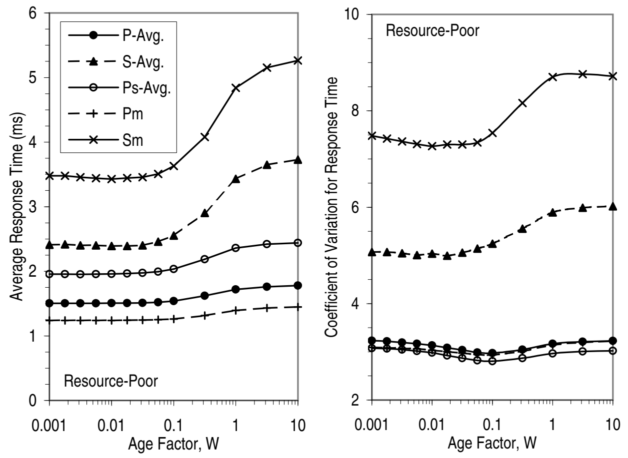
[i] Improvement over FCFS ([original value – new value]/[original value]).

Table 3.7: Performance with ASATF Scheduling.

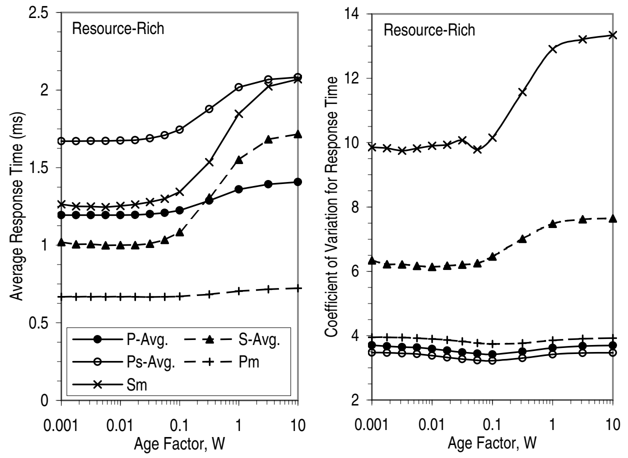
With a sufficiently large aging factor W, ASATF degenerates to First Come First Served (FCFS). A W value of 0.006[1] is recommended in [JW91, WGP94] but the range of "good" values for W is found to be wide. In Figure 3.18, we plot the average response time including both reads and writes, and its coefficient of variation as a function of W. The corresponding plots considering the reads and writes separately can be found in Figures C.19 and C.20 while the plots of the average service time as a function of W are in Figure C.21. For all our workloads, the average response time is almost constant for $W < 0.03$. Observe that as W increases, the coefficient of variation for response time decreases gradually to a minimum and then increases rather sharply beyond that. The improvement in the coefficient of variation is gradual as we increase the aging factor from zero because our model, unlike those used in [JW91, WGP94], takes into consideration feedback between request completion and subsequent request arrivals so that requests are less likely to be starved. Since the variability in response time increases rather sharply for W values beyond the optimal, we err on the side of caution and select a value of 0.01 as the baseline for our other simulations.

By comparing the response time at large values of W with that at small values of W, we can quantify the net effect of request scheduling. We summarize the results in Table 3.7. In general, scheduling tends to have a bigger impact in the server environments. Improvement of up to 39% in average response time is seen for the server workloads. For the PC workloads, the improvement is about 15% on average. Looking at the reads and writes separately, we find that in most cases, the improvement in write response time is about two to three times that for reads. This is because writes tend to come in big bursts so that if

[1][WGP94] recommends 6 but if T_{pos} and T_{wait} are in the same units, as one would reasonably expect, the correct value should be 0.006.



(a) Resource-Poor.



(b) Resource-Rich.

Figure 3.18: Effect of Age Factor, W, on Response Time.

the destage operations are not scheduled efficiently, the write buffer is likely to become full and cause the incoming writes to stall.

Note that request scheduling actually has two separate effects – one is to reduce the time needed to service a request, the other is to reduce the waiting time by letting the shortest job proceed first. The improvement in service time is also presented in Table 3.7. Observe that the service time improvement is more consistent across the PC and server workloads than the improvement in response time. This suggests that a lot of the response time improvement for the server workloads is due to less waiting. Across all our workloads, read service time is barely improved by request scheduling while write service time is improved by between 20-30% in the resource-poor environment and 35-40% in the resource-rich environment. The poor improvement for read requests is expected because the number of read requests that are outstanding and can be scheduled tends to be low (see Chapter 2). The dramatic improvement in write service time reflects our write buffering strategy, which is specifically designed to maintain a sizeable number of outstanding destage requests so that they can be effectively scheduled.

As we have alluded to several times, the effectiveness of request scheduling generally increases with the number of requests that are available to be scheduled. In most systems, the maximum number of requests that are outstanding to the storage system can be set. The actual queue depth depends on the workload. In Table 3.8, we summarize the effect of allowing multiple requests to be outstanding to the storage system. The data considering reads separately from the writes are plotted in Figures C.22 and C.23. Note that as the maximum queue depth is increased, the average service time is improved but because some requests are deferred, the average response time may rise. For our workloads, a maximum queue depth of eight works well. With this maximum queue depth, the average response time for the server workloads is improved by between 30% and 40% in both the resource-poor and resource-rich environments while the PC workloads are improved by about 20%. In terms of average service time, both the PC and server workloads are improved by about 20%. Breaking down the requests into reads and writes, we again find that most of the improvement is due to the writes (Table C.1).

Note that the benefit of increasing the maximum queue depth is more than that due to scheduling the disk arm. This is because the storage system cache in effect performs another level of scheduling by allowing subsequent cache hits to proceed. This effect, however, tends to be secondary since the improvement in response time when the maximum queue depth is increased from one to eight (Table 3.8) generally exceeds by only a small amount the improvement due to scheduling a maximum of eight outstanding requests (Table 3.7).

3.5.5 Parallel I/O

A widely used technique to improve I/O performance is to distribute data among several disks so that multiple requests can be serviced by the different disks concurrently. In addition, a single request than spans multiple disks can be sped up if it is serviced by the

		Average Response Time								Average Service Time							
		Max. Q Depth = 2		4		8		16		Max. Q Depth = 2		4		8		16	
		ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]
Resource-Poor	P-Avg.	1.68	8.59	1.56	14.8	1.51	17.8	1.52	17.2	1.99	1.44	1.79	11.1	1.74	13.7	1.70	15.8
	S-Avg.	3.71	6.46	2.77	24.1	2.39	30.4	2.17	34.8	1.97	1.58	1.69	15.5	1.57	20.9	1.49	25.0
	Ps-Avg.	2.29	5.83	2.08	14.4	1.96	19.3	1.99	18.6	1.79	2.89	1.59	13.9	1.52	18.0	1.46	21.0
	Pm	1.43	15.0	1.31	22.7	1.24	26.6	1.25	25.8	1.97	1.61	1.71	14.3	1.63	18.6	1.57	21.6
	Sm	5.32	4.41	4.02	27.7	3.43	38.3	3.12	44.0	2.52	1.76	2.19	14.5	2.06	19.8	1.95	23.8
Resource-Rich	P-Avg.	1.35	8.94	1.25	15.3	1.19	19.0	1.23	17.2	1.38	1.71	1.21	13.7	1.14	18.8	1.08	22.9
	S-Avg.	1.70	9.42	1.23	24.9	1.00	30.4	0.889	34.6	0.895	1.72	0.747	16.3	0.689	21.2	0.635	27.4
	Ps-Avg.	1.98	4.99	1.80	13.9	1.67	19.8	1.72	18.3	1.35	2.96	1.18	14.9	1.11	19.9	1.06	23.7
	Pm	0.753	16.2	0.703	21.8	0.667	25.8	0.689	23.3	0.923	1.36	0.800	14.5	0.746	20.2	0.703	24.9
	Sm	2.02	6.33	1.48	31.3	1.25	41.8	1.14	47.0	1.27	1.04	1.06	17.8	0.963	25.1	0.889	30.9

[i] Improvement over queue depth of one ([original value – new value]/[original value]).

Table 3.8: Average Response and Service Times as Maximum Queue Depth is Increased from One.

disks in parallel. The latter tends to make more sense for workloads dominated by very large transfers, specifically scientific workloads. For most other workloads where requests are small and plentiful, the ability to handle many of them concurrently is usually more important.

In general, data can be distributed among the disks in various ways. The two most common approaches are to organize the disks into a volume set or a stripe set. In a volume set, data is laid out on a disk until it is full before the next disk is used. In a stripe set, data is divided into units called stripe units and the stripe units are laid out across the disks in a round robin fashion. Note that the volume set is essentially a stripe set with a stripe unit that is equal to the size of the disk. In RAID (Redundant Array of Inexpensive Disks) [CNC$^+$96] terminology, the stripe set is known as RAID-0. A shortcoming of striping data across the disks is that each disk contains some blocks of many files so that a single disk failure could wipe out many files. There are well-known techniques such as mirroring and parity protection to overcome this weakness but they are beyond the scope of this study. The interested reader is referred to [CLG$^+$94] for more details.

The choice of stripe unit has a major bearing on the performance of the storage system. A small stripe unit could result in single requests spanning multiple disks, thereby increasing the number of physical I/Os and tying up many disks. More importantly, it results in many small random requests, which the disks are not very efficient at handling. Furthermore, a small stripe unit makes sequential prefetch by the disk less effective because data that appears contiguous on a disk are likely to be logically interspersed by data that are on other disks. On the other hand, a small stripe unit evens out the load across the multiple disks and reduces the chances that a subset of the disks will be disproportionately busy, a condition often referred to as access skew. For parity-protected arrays of disks (*e.g.,* RAID-5), a large stripe unit would make it more difficult to do a full-stripe write so that

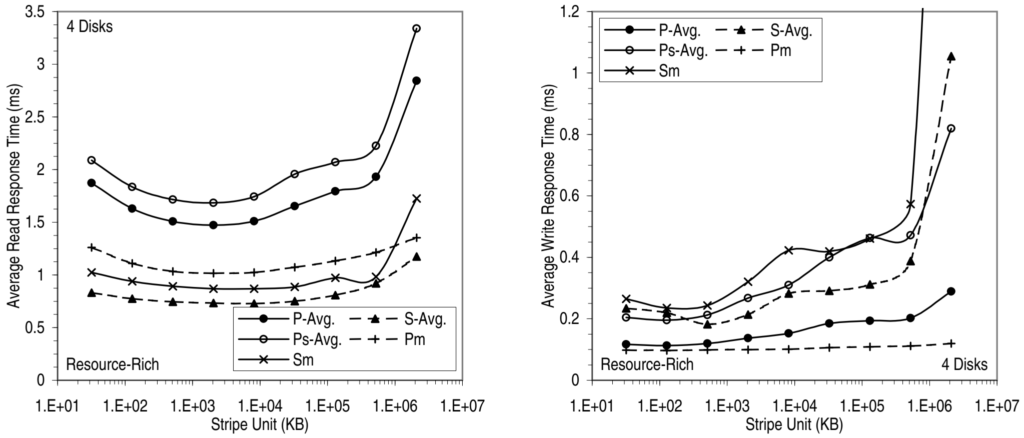
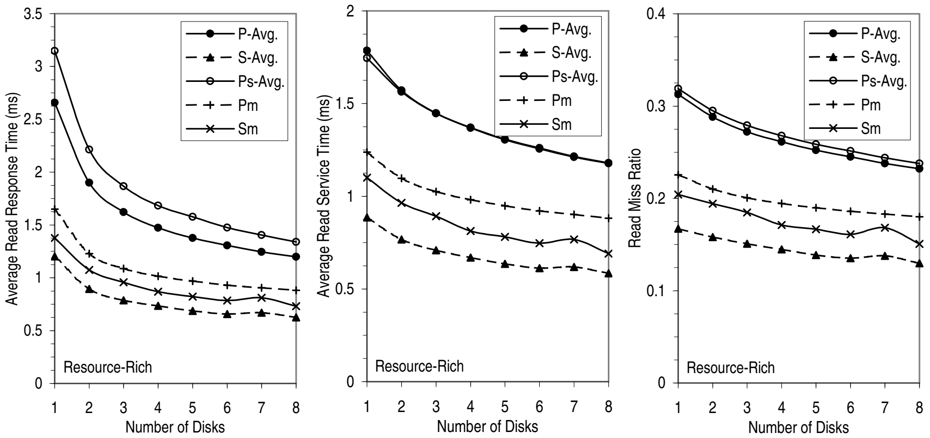


Figure 3.19: Average Read and Write Response Time as a Function of Stripe Unit (Resource-Rich).

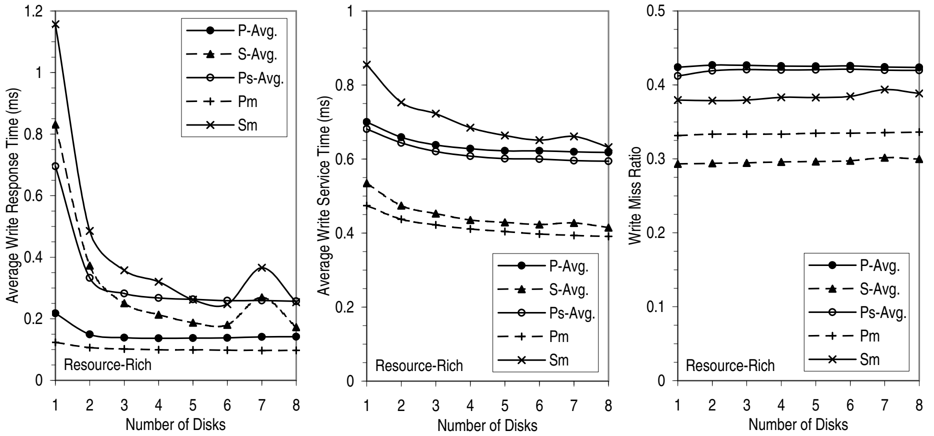
write performance might be degraded. However, full-stripe writes are not very common in most workloads. Results of a previous study on RAID-5 striping [CL95] indicate that for workloads that are meant to model time sharing and transaction processing workloads, read throughput increases with stripe unit until the megabyte range while write throughput is within 20% of the maximum at a stripe unit of 1 MB.

In Figures 3.19 and C.24, we plot the average read and write response time for our various workloads as a function of the stripe unit, assuming that data is striped across four disks. The corresponding plots for the service time are in Figure C.25. Observe that for our workloads, access skew, or imbalance in the amount of work borne by the different disks, does not seem to be a major issue until the stripe unit is larger than 100 MB. As we increase the number of disks, access skew becomes a bigger issue so that the upward surge in response time at large stripe units is more apparent (Figure C.26). From Figures 3.19 and C.24, a stripe unit of less than about 2 MB works well for the writes. For the reads, performance is generally good with a stripe unit in the megabyte range with the best performance being achieved by a stripe unit of 2 MB. In the rest of this chapter, we will assume a stripe unit of 2 MB.

Figures 3.20 and C.27 show the performance achieved as we increase the number of disks that are striped across. Recall that the read miss ratio is defined as the fraction of read requests that requires physical I/O. Therefore, when there are multiple disks each with a cache, the read miss ratio is the arithmetic mean of the read miss ratio of each disk, weighted by the number of reads to that disk. The write miss ratio is similarly defined. Observe that for all our workloads, striping data across four disks is sufficient to reap most of the performance benefit. In Table 3.9, we summarize the improvement in average response time, service time and miss ratio when data is striped across four disks. Overall, average read response time is improved by about 45% in the resource-poor environment and by about 40% in the resource-rich environment. Write response time is reduced a lot



(a) Read.



(b) Write.

Figure 3.20: Performance as a Function of the Number of Disks (Resource-Rich).

		Read						Write					
		Avg. Resp. Time		Avg. Serv. Time		Miss Ratio		Avg. Resp. Time		Avg. Serv. Time		Miss Ratio	
		ms	%[i]	ms	%[i]		%[i]	ms	%[i]	ms	%[i]		%[i]
Resource-Poor	P-Avg.	1.72	48.0	1.56	30.1	0.333	22.8	0.105	43.7	1.34	4.75	0.596	1.62
	S-Avg.	1.37	49.6	1.30	34.6	0.275	22.8	0.149	70.4	1.04	18.2	0.480	7.72
	Ps-Avg.	1.91	50.1	1.56	28.6	0.350	22.5	0.149	74.6	0.915	13.4	0.503	3.50
	Pm	1.77	43.7	1.66	25.5	0.364	18.7	0.102	46.4	1.26	3.14	0.577	3.52
	Sm	1.93	42.7	1.91	28.5	0.385	17.7	0.223	93.6	1.13	28.1	0.495	13.4
Resource-Rich	P-Avg.	1.47	43.7	1.37	24.0	0.261	16.9	0.137	27.0	0.628	10.4	0.425	-0.434
	S-Avg.	0.734	35.1	0.669	23.3	0.145	12.6	0.214	54.0	0.435	20.1	0.296	-0.936
	Ps-Avg.	1.68	46.2	1.37	22.1	0.268	16.5	0.268	57.9	0.608	10.8	0.420	-2.15
	Pm	1.02	38.5	0.981	20.7	0.195	13.7	0.100	19.4	0.411	13.3	0.333	-0.582
	Sm	0.869	36.8	0.812	26.2	0.171	16.2	0.321	72.3	0.685	19.9	0.383	-0.979

[i] Improvement over single disk ([original value – new value]/[original value]).

Table 3.9: Performance with Striping across Four Disks.

more for the server workloads than the PC workloads – as high as 94% in the resource-poor environment and 74% in the resource-rich environment. This is because, as noted earlier, writes tend to come in large bursts in the server workloads and with more disks, these writes can be handled with much less waiting time.

Note that we are using identical disks so the total capacity of the system grows as we increase the number of disks. Therefore, part of the performance improvement reported in Table 3.9 results from *short-stroking* or using less of each disk. An alternative is to compare performance using smaller-capacity disks as the number of disks increases so as to keep the total storage capacity constant, but such a comparison is not necessarily more insightful. Moreover, the storage required for many of our workloads is already smaller than the capacity offered by a 1-surface disk. We believe that a more interesting approach would be to quantify the performance effect of short-stroking the disks.

Observe from Table 3.9 that in the resource-poor environment, the read miss ratio decreases with the number of disks. This is because in this environment, each disk has an 8 MB cache so that as the number of disks increases, we in effect have more cache. In the resource-rich environment, the cache is supposed to reflect a large adaptor/controller cache and thus the total size is fixed at 1% of the storage used[2]. However, there is a per-disk prefetch buffer, which explains why the read miss ratio also decreases with the number of disks. But notice that the read service time improves more than the read miss ratio as we increase the number of disks from one to four. In the resource-poor environment, the read miss ratio improves by about 20% while the read service time is reduced by around 30%. In the resource-rich environment, the read miss ratio improves by about 15% while the

[2]Note that we distributed the cache equally among the disks to simplify our simulation model. This is a reasonable approximation except when the stripe unit is very large, causing access skew among the disks.

		Average Read Service Time							Average Write Service Time						
		Number of Disks							Number of Disks						
		2	3	4	5	6	7	8	2	3	4	5	6	7	8
Resource-Poor	P-Avg.	5.16	7.83	9.37	10.2	10.8	11.3	11.2	0.610	2.08	2.58	3.39	3.72	4.63	4.13
	S-Avg.	8.88	12.4	14.1	15.1	15.9	16.2	15.8	7.40	8.11	10.3	12.1	10.4	10.4	12.3
	Ps-Avg.	4.07	6.78	8.33	9.11	9.71	10.2	10.3	2.93	4.87	5.54	6.08	5.94	6.66	5.94
	Pm	5.34	7.79	9.06	9.75	10.3	10.7	10.9	0.523	1.29	1.17	2.02	2.24	1.96	1.83
	Sm	8.04	11.2	13.4	14.8	15.7	16.5	16.8	8.52	10.1	13.9	14.5	15.3	16.0	16.0
Resource-Rich	P-Avg.	3.83	6.13	7.23	7.91	8.09	8.69	8.44	5.82	8.92	10.3	11.1	11.1	11.5	11.6
	S-Avg.	7.89	10.6	11.7	12.0	12.2	12.2	11.0	13.1	17.2	20.0	20.4	21.1	21.0	22.0
	Ps-Avg.	2.59	4.69	5.71	6.35	6.51	7.11	7.01	5.30	8.76	10.6	11.7	11.7	12.4	12.4
	Pm	4.58	6.68	7.49	8.07	8.60	8.75	8.87	7.72	11.0	13.3	14.7	16.2	17.0	17.5
	Sm	7.76	10.5	14.7	13.4	13.9	15.5	16.4	11.9	15.4	19.8	22.3	23.8	22.6	26.0

Table 3.10: Improvement in Average Service Time as the Number of Disks Striped Across is Increased from One while the Total Cache and Buffer Space are Kept Constant.

read service time is reduced by almost 25%. Such data suggests that short-stroking could account for roughly a 10% performance improvement.

To further confirm that the benefit of short-stroking is in the 10% range, we rerun the experiments keeping the total cache and buffer size constant as we increase the number of disks. Results presented in Figures C.28 and C.29 show that the miss ratio is constant as we increase the number of disks in these experiments. Therefore, all the resulting service time improvement can be attributed to the effect of short-stroking. In Table 3.10, we summarize this improvement. Observe that the service time improvement saturates beyond about four disks or when less than a quarter of the disk is used. Largely in agreement with the results above, we find that short-stroking improves the average read service time by up to 10-15% for our workloads. For writes, the improvement ranges from 2% to 16% in the resource-poor environment and from 12% to 26% in the resource-rich environment.

3.6 Effect of Technology Improvement

At its core, disk storage is composed of a set of rotating platters on the surfaces of which data is recorded. There is typically a read-write head for each surface and all the heads are attached to the disk arm so that they move in tandem. A simple high-level description such as this already suggests that there are multiple dimensions to the performance of the disk. For instance, the rate at which the platters rotate, how fast the arm moves, and how closely packed the data is, all affect, in some way, how quickly data can be accessed. Moreover, the effective performance of a disk depends on which blocks are accessed and in what order. Therefore, it is not clear what effect technology improvement or scaling in any one dimension has on real-world performance. In this section, we try to relate scaling in the underlying technology to the actual performance of real workloads. The goal is to quantify the real impact of improvement in each dimension so as to establish some rules of thumb that can be used by disk designers and system builders who select and qualify the disks. Note that there are sometimes discontinuities in the technology. For instance, the transition from $5\frac{1}{4}$-inch disk to $3\frac{1}{2}$-inch disk. Our analysis focuses on the overall trend rather than such discrete effects.

The result of technology improvement in the different dimensions are generally difficult to isolate and systematically quantify because the performance metrics that we are familiar with (*e.g.,* seek time, rotational latency) are often metrics that compound the effect of improvement in multiple dimensions. For instance, the often quoted ten percent yearly improvement in the access time of disks results from a combination of increase in rotational speed which reduces the rotational latency, decrease in seek time due to improvement in the disk arm actuator, and smaller diameter disks or narrower data band which reduces the seek distance. In practice, for a given workload, the actual seek time is also affected by improvement in areal density because the head has to move a smaller physical distance to get to the data. Changes in areal density also lead to changes in storage capacity which could potentially affect the number of disks and the mapping of data to disks. In this section, we break down the continuous improvement in disk technology into four major basic effects, namely seek time reduction due to actuator improvement, spin rate increase, linear density improvement and increase in track density.

Note that the disk heads for the different surfaces are attached to the disk arm and move in tandem. In the past, this means that tracks within a cylinder are vertically aligned and no additional seek was required to read the next track in the cylinder. However, in modern disks, only one of the heads is positioned to read or write at any one time because the disk arm flexes at the high frequency it is operated at. Therefore, when the head reaches the end of a track, there is a delay before the next head is positioned to start transferring the data. To prevent having to wait an entire revolution after a track switch, the tracks in a cylinder are laid out at an offset known as the track-switch skew. There is also a delay for moving the head to an adjacent cylinder so tracks are laid out at an offset known as the cylinder-switch skew across cylinder boundaries. As we scale the performance of the disk, we adjust the skews to make sure that the disk does not "miss revolutions" for transfers that span multiple tracks.

3.6.1 Mechanical Improvement

We begin by examining the improvement in the mechanical or moving parts of the disk. Figure 3.21 presents the historical rates of change in the average seek time and rotational speed for the IBM family of server disks. The average seek time is generally taken to be the average time needed to seek between two random blocks on the disk. The average access time is defined as the sum of the average seek time and the time needed for half a rotation of the disk. Observe that on average, seek time decreases by about 8% per year while rotational speed increases by about 9% per year. Putting the two together, average random access performance improves by just over 8% per year.

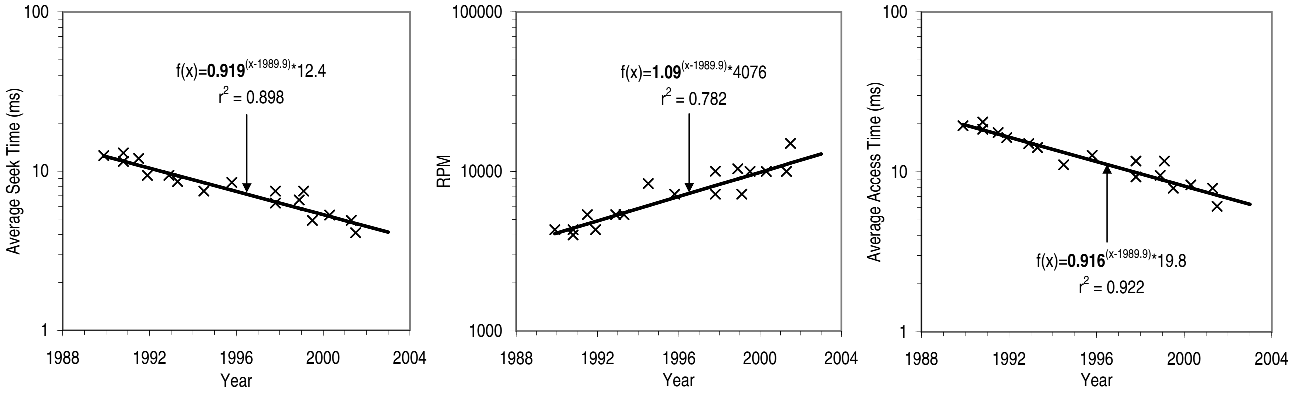


Figure 3.21: Historical Rates of Change in Average Seek Time, Rotational Speed and Access Time (IBM Server Disks).

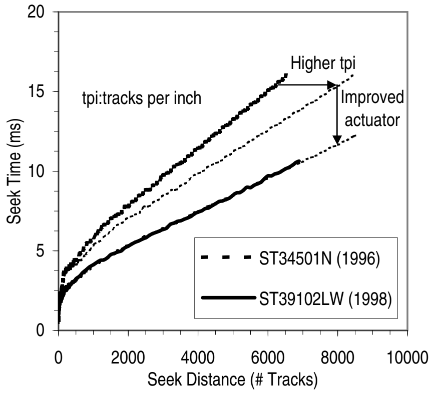
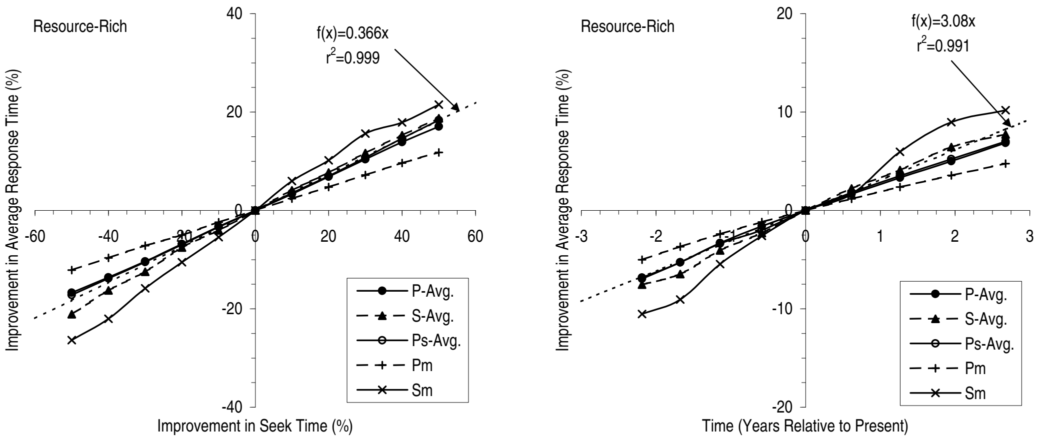


Figure 3.22: Change in Seek Profile over Time.

Seek Time

As shown in Figure 3.2 on page 52, the seek time is a non-linear function of the seek distance. We know that historically, the average seek time improves by about 8% per year but how does this affect the seek time for different seek distances? It turns out that a good way to model the improvement in seek time is to simply scale the seek profile vertically by a constant factor. For instance, in Figure 3.22, we show how the seek profile changes across two generations of a disk family. Beginning with the seek profile of the earlier disk, we first scale it horizontally to account for the increase in the track density. Subsequent scaling in the vertical direction results in a curve that fits the seek profile of the later disk almost perfectly.

In Figures 3.23 and C.30, we plot the effect of seek time improvement on the average response time for our various workloads. The corresponding plots for the average service time are similar and are presented in Figure C.31. Besides plotting the improvement in average response time as a function of the improvement in seek time (Figure 3.23(a)), we



(a) Function of Seek Time Improvement.

(b) Function of Years of Seek Time Improvement at Historical Rate (8% Per Year).

Figure 3.23: Effect of Improvement in Seek Time on Average Response Time (Resource-Rich).

also show how the improvement in average read response time varies over time, assuming the historical 8% yearly improvement in seek time (Figure 3.23(b)). To generalize our results, we fitted a curve to the arithmetic mean of the five classes of workloads. As shown in the figures, a linear function of the form $f(x) = ax$ where a is a constant turns out to be a good fit. Specifically, we find that a 10% improvement in seek time translates roughly into a 4% gain in the actual average response time, and that a year of seek time improvement at the historical rate of 8% per year results in just over 3% improvement in the average response time.

Rotational Speed

Figures 3.24 and C.32 show how increasing the rotational speed of the disk affects the average response time for our various workloads. Again, the corresponding plots for the service time are similar and are in Figure C.33. Notice that the S-Avg. plot in Figure C.32 shows a little performance loss as the rotational speed is increased. This is due to the fact that DS1, one of the components of S-Avg., is sensitive to how the blocks are laid out in tracks because some of its accesses, especially the writes, occur in specific patterns. As we scale the rotational speed and adjust the track and cylinder-switch skews, there are cases where consecutively accessed blocks are poorly positioned rotationally, even with request scheduling. Such situations highlight the need for automatic block reorganization such as that proposed in the following chapter.

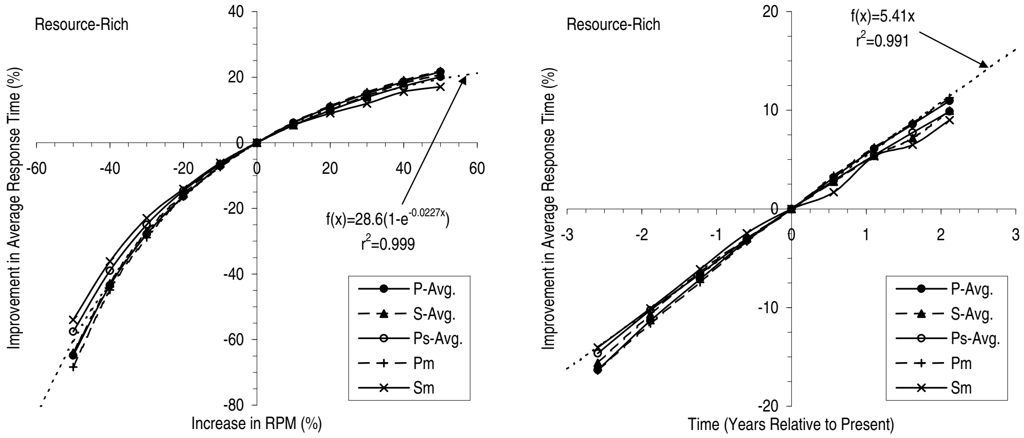


Figure 3.24: Effect of RPM Scaling on Average Response Time (Resource-Rich).

Observe from the figures that the improvement in average response time as a function of the increase in rotational speed can be accurately described by a function of the form $f(x) = a(1 - e^{-bx})$ where a and b are constants. Such a function suggests that as we increase the rotational speed keeping other factors constant, the marginal improvement diminishes so that the maximum improvement is a. Taking into account the historical rate of increase in rotational speed (9% per year), we find that a year's worth of scaling in rotational speed corresponds to about a 5% improvement in average response time.

3.6.2 Increase in Areal Density

In Figure 3.25, we present the rate of increase in the recording or areal density of disks over the last ten years. Observe that the linear density has been increasing by approximately 21% per year while the track density has been going up by around 24% per year. Areal density has increased especially sharply in the last few years so that with a least squares estimate (no weighting), the compound growth rate is as high as 62%. If we minimize the sum of squares of the relative (instead of absolute) distances of the data points from the fitted line so that the large areal densities do not dominate ("$1/y^2$ weighting"), the compound growth rate is about 49%. Combining the growth rate in rotational speed and in linear density, we obtain the rate of increase in the disk data rate. As shown in Figure 3.26, this turns out to be 40% per year, which is dramatically higher than the 8% annual improvement in average access time. The result is a huge gap between random and sequential performance, and is one of the primary motivations for reorganizing disk blocks to improve the spatial locality of reference [Chapter 4].

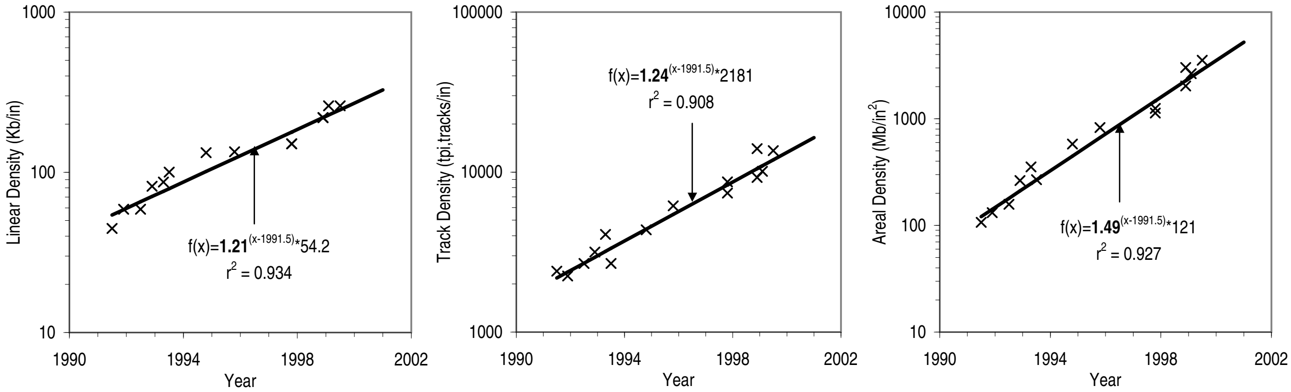


Figure 3.25: Historical Rates of Increase in Linear, Track and Areal Density (IBM Server Disks).

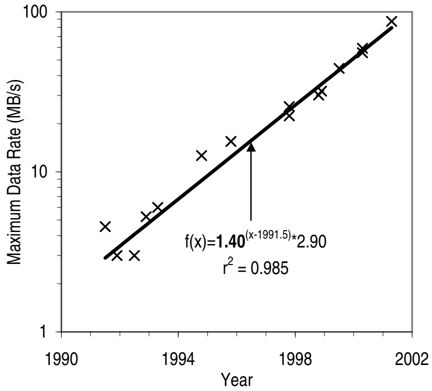


Figure 3.26: Historical Rate of Increase in Maximum Data Rate (IBM Server Disks).

Linear Density

Increasing the areal density reduces the cost and therefore the price-performance of disk-based storage. Areal density improvement also directly affects performance because as bits are packed more closely together, they can be accessed with a smaller physical movement. Figures 3.27 and C.34 show how increases in the linear density reduce the average response time for our various workloads. Observe that there is a discontinuity in the plots at a linear-density improvement of around -20%. This is because as linear density is reduced, we require more disks to hold the same amount of data. The jump reflects the performance gain from having an additional disk arm and the associated disk cache or prefetch buffer. Most of the gain comes from the ability to service more requests concurrently. As we would expect, the service time is impacted less by having an additional disk (Figure C.35).

Focusing on the long continuous segment of the plots, we find that the improvement in average response time as a function of the increase in linear density can again be accurately

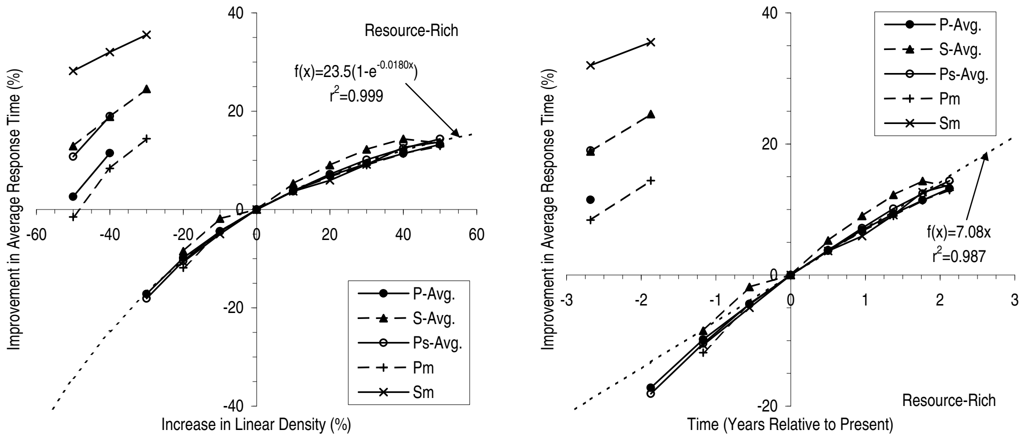


Figure 3.27: Effect of Increased Linear Density on Average Response Time (Resource-Rich).

modeled by a function of the form $f(x) = a(1 - e^{-bx})$ where a and b are constants. The effect is similar to that of increasing the rotational speed but is quantitatively less per unit of improvement because increasing the linear density does not reduce the rotational latency. We find that every year of improvement in linear density at the historical rate of 21% per year results in a 6-7% reduction in average response time.

Track Density

Packing the tracks closer together means that the arm has to move over a shorter physical distance to get to the same track. This effect is similar to that of improving the seek time but the quantitative effect on the average response time per unit of improvement tends to be much smaller because of the shape of the seek profile. In particular, the marginal cost of moving the arm is relatively small once it is moved. In Figures 3.28 and C.36, we present the effect of increasing the track density on the average response time. Observe that a year's worth of track density scaling (24%) buys only about 3-4% improvement in average response time.

Again, DS1 is not well-behaved because it is sensitive to how blocks are laid out in tracks, and this sensitivity causes the jagged nature of the plot for S-Avg. On the surface, this result is surprising because changing the track density should not affect how blocks are laid out in tracks. A deeper analysis reveals that the block layout does get affected because changes in the track density lead to changes in the zoning of the disk. In general, to take advantage of the fact that tracks are longer the further they are from the center of the disk, the disk is divided into concentric zones or bands within which each track has the same number of sectors. As track density changes, we assume that the physical dimensions of each zone or band remains constant but the number of tracks within each zone increases.

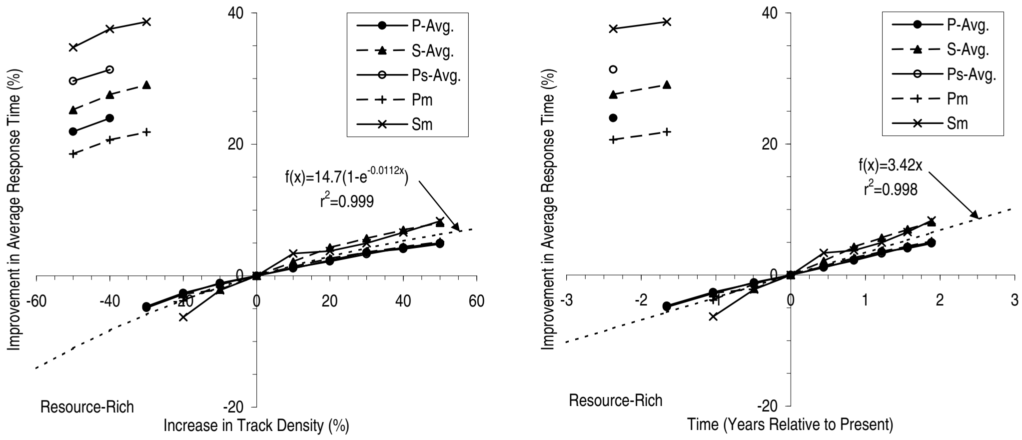


Figure 3.28: Effect of Increased Track Density on Average Response Time (Resource-Rich).

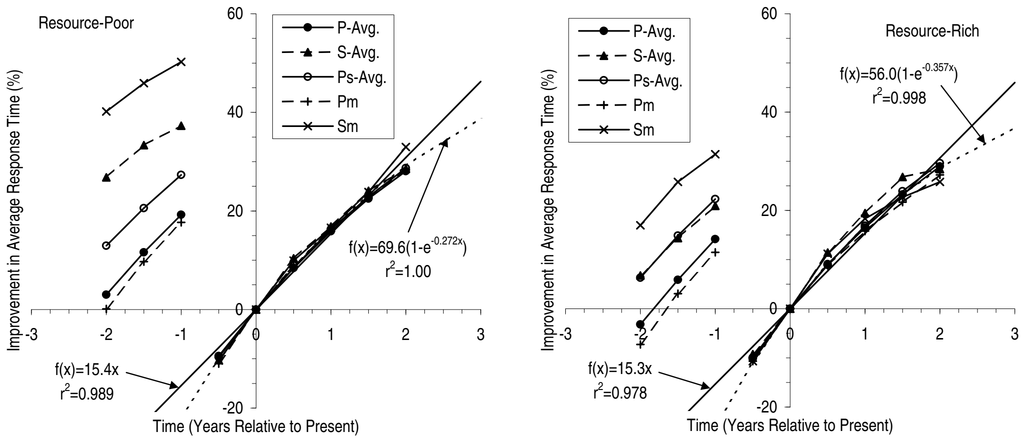


Figure 3.29: Overall Effect of Disk Improvement on Average Response Time.

3.6.3 Overall Improvement over Time

In Figures 3.29 and C.38, we put together the effect of mechanical improvement and areal density scaling to obtain the overall performance effect of disk technology evolution. As shown in the figures, the actual improvement in average response and service times as a function of the years of disk improvement at the historical rates can best be described by an exponential function of the form $f(x) = a(1 - e^{-bx})$ where a and b are constants. However, to project outward for the next couple of years, a linear function is a reasonably good fit. Observe that for our various workloads, the average response time and service time are projected to improve by about 15% per year. The different classes of workloads have almost identical plots, which increases confidence in our result. The rate of actual performance improvement (15%) turns out to be significantly higher than the widely quoted

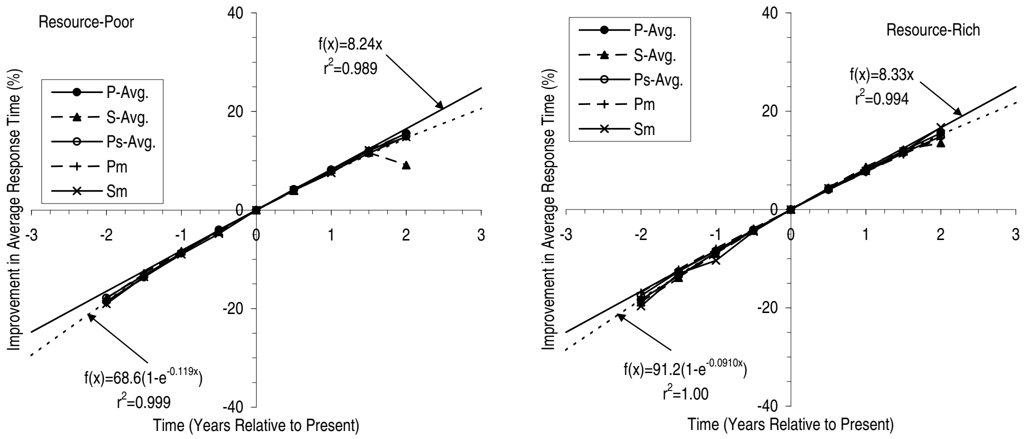


Figure 3.30: Effect of Mechanical Improvement on Average Response Time.

"less than 10%" yearly improvement in disk performance because it takes into account the improvement in areal density and assumes that the workload and the number of disks used remain constant so that the disk occupancy rate is diminishing.

To estimate the yearly improvement in the more realistic situation where the increased capacity of the newer disks is utilized so that the disk occupancy rate is kept constant, we examine the effect of improving only the mechanical portions of the disk (seek and rotational speed). This is presented in Figures 3.30 and C.39 which show that the average response and service times improve by about 8% per year. We also explore the scenario where only the areal density is increased (Figure 3.31 and C.40) and discover that the average response and service times are improved by about 9% per year. This improvement comes about because as areal density rises, the data is packed closer together and can be accessed with a smaller physical movement. Note that the overall yearly performance improvement, at 15%, is slightly lower than the sum of the effects of the mechanical improvement and the areal density increase. This is because the two effects are not orthogonal. For instance, as the recording density is increased, each access will likely entail less mechanical movement so that the benefit of having faster mechanical components is diminished.

Another rule of thumb that is useful to system designers is one that relates the actual access time to the advertised or specified performance parameters of a disk. There is often a wide disparity between the actual and specified performance numbers because the specified figures are obtained under assumptions that the workload exhibits no locality. Specifically, the average seek time is defined as the time taken to seek between two random blocks on the disk and the rotational latency is generally taken to be the time for half a revolution of the disk. In practice, there is locality in the reference stream so we would expect the actual access time to be significantly lower. In Figure 3.32, we look at the actual seek time and rotational latency of our various workloads as a percentage of the average values specified by the disk manufacturer. As shown in the figure, the actual seek time is about 35% of the advertised average seek time and the time taken for the correct block to rotate under

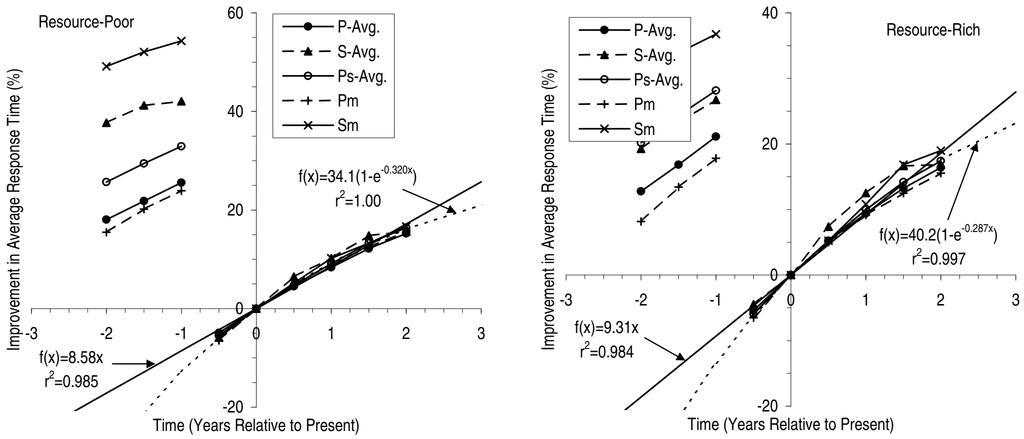


Figure 3.31: Effect of Areal Density Increase on Average Response Time.

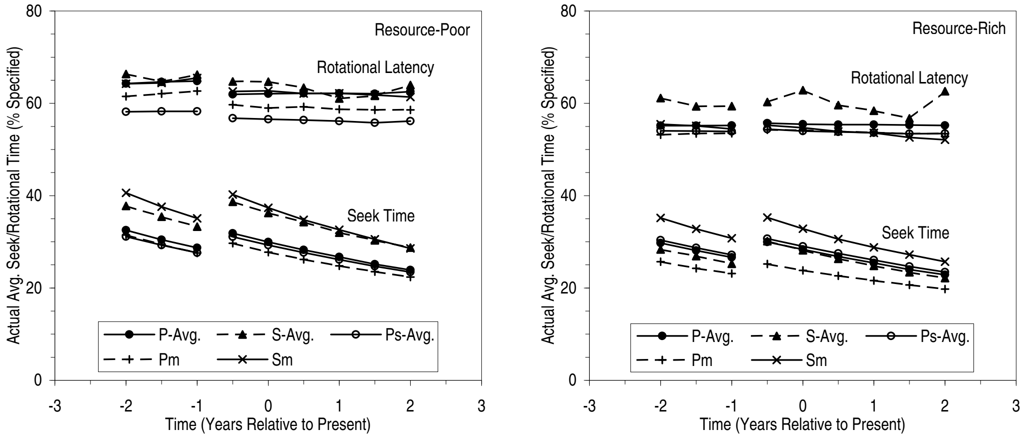
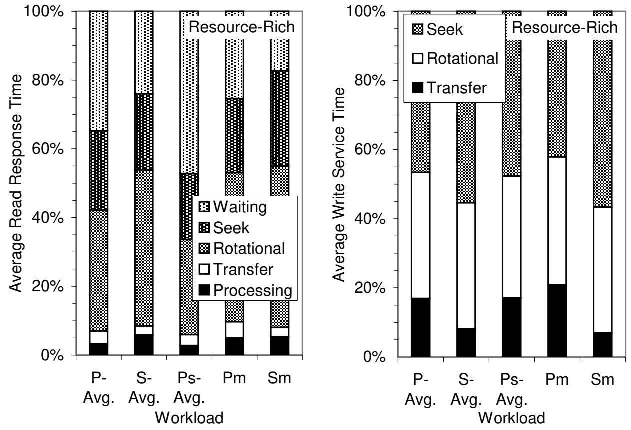


Figure 3.32: Actual Average Seek/Rotational Time as Percentage of Manufacturer Specified Values.

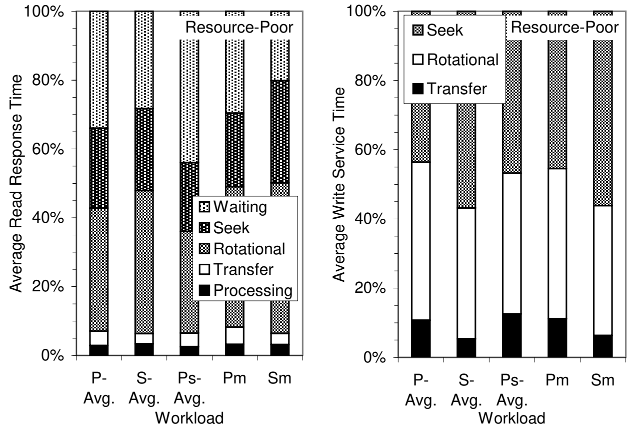
the head is about 80% of that specified. The seek percentage decreases slightly over time because of the improvement in areal density but the effect is not very significant.

To gain further insight into where a request is spending most of its time, we break down the average read response time and write service time into their components in Figure 3.33. In the figure, the component identified as "processing" refers to the disk command processing time, which varies with the type of request (read or write) and with whether the previous request is a cache hit. For all our workloads, the command processing time is not significant and averages less than 5% of the read response time for all our workloads. We define *waiting time*, also known as *queueing time*, as the difference between response time and the sum of service time and processing time.

Notice that even with a 10,000 RPM disk, rotational latency constitutes a major portion (30-40%) of both the read response time and the write service time. The seek time is also



(a) Resource-Rich.



(b) Resource-Poor.

Figure 3.33: Breakdown of Average Read Response and Write Service Time.

very significant, accounting for about 25% of the read response time and 45% of the write service time. Note that request scheduling affects how the disk head positioning time is proportioned between seek and rotational time, especially for writes which we issue in batches. In any case, for both reads and writes, most of the time is spent positioning the disk head. The transfer time, on the other hand, accounts for less than 5% of the read response time and only about 10% of the write service time. As the data rate continues to rise dramatically, the transfer time will diminish further. Note that the transfer time is the only time during which data is being read or written. In other words, the disk bandwidth will become less and less effectively utilized. Thus we should consider reorganizing disk blocks to better take advantage of the available disk bandwidth [Chapter 4]. Observe further that the waiting time is very significant for reads and is in fact the largest component for some workloads. This, however, does not mean that the read response time will ultimately be limited by the waiting time because improving the performance of the disk will reduce the waiting time proportionately.

3.7 Conclusions

In this chapter, we systematically study various I/O optimization techniques to establish their actual effectiveness at improving I/O performance. Our results, which are based on analyzing the physical I/Os of a variety of real server and PC workloads, are summarized in Table 3.11. The table shows for each technique, the average improvement over five classes of workloads – PC workloads, server workloads, sped-up PC workloads, merged PC workloads and merged server workloads. We find that prefetching offers by far the most significant improvement in read performance, reducing the average read response and service times by about half. As for writes, buffering has the potential to most dramatically increase performance, reducing the average write response time by more than 90% and the average write service time by more than 70%.

More specifically, we find that at the storage or physical level, small read caches are not very useful because some amount of caching is performed upstream. The small amount of memory in the disk serves primarily as a prefetch buffer. However, if the cache is large enough, read caching at the storage level can be effective. We find that the read miss ratio decreases as the inverse of the ratio of cache size to storage used. In addition, our results clearly indicate that sequential prefetch is extremely valuable. In a resource-poor environment such as one where the storage system consists of only disks and low-end disk adaptors, sequential prefetch together with caching is able to filter out 40-60% of the read requests. In a resource-rich environment where there is a large outboard controller, only about 40% of the read requests require a physical I/O if caching and sequential prefetching are performed. The additional use of opportunistic prefetch makes a significant difference, further reducing the miss ratio to about 35-45% in the resource-poor environment and to 20-30% in the resource-rich environment.

%		Read			Write		
		Avg. Resp. Time	Avg. Service Time	Miss Ratio	Avg. Resp. Time	Avg. Service Time	Miss Ratio
Resource-Poor	Read Caching	4.49	4.14	4.45	0	0	0
	Prefetching	46.6	46.1	53.0	0	0	0
	Write Buffering	0	0	0	93.8	71.8	43.5
	Request Queuing	16.2	2.8	0	49.9	30.5	0
	Parallel I/O	46.8	29.5	20.9	65.7	13.5	5.96
Resource-Rich	Read Caching	37.4	36.1	35.1	0	0	0
	Prefetching	51.1	50.3	59.7	0	0	0
	Write Buffering	0	0	0	96.0	86.2	63.1
	Request Queuing	17.0	1.8	0	46.2	38.4	0
	Parallel I/O	40.0	23.3	15.2	46.1	14.9	-1.02

Table 3.11: Performance Effect of Various I/O Optimization Techniques. Table shows improvement (|[value without technique - value with technique]|/[value without technique]) averaged over the five classes of workloads. Table entries are shaded to reflect the relative magnitude of improvement with darker shades representing larger improvements.

Our analysis demonstrates that write buffering can dramatically improve write performance through three distinct effects. First, by absorbing the incoming writes and performing them in the background, write buffering is able to improve write response time by over 90%. Second, by delaying when the physical writes are carried out, repeated writes to the same blocks can be eliminated. For all our workloads, 40% of the writes are eliminated by a small write buffer of less than 1 MB. For larger write buffers, we find that the write miss ratio follows a fifth root rule, meaning that the miss ratio goes down as the inverse fifth root of the ratio of buffer size to storage used. We also discover that most of the benefit of write elimination can be achieved without requiring dirty data to remain in the buffer beyond an hour. Third, by carrying out the physical writes in batches, write buffering makes it possible to schedule the writes so that they are carried out more efficiently. On average, write buffering is able to reduce the write service time by about 70% in the resource-poor environment and by as much as 90% in the resource-rich environment.

Compared to caching, prefetching and write buffering, we find that request scheduling tends to have a smaller effect. In particular, we observe that having a queue depth beyond one improves average response time by 30-40% for the server workloads and by about 20% for the PC workloads. Most of the improvement comes from scheduling the requests to minimize the time spent positioning the disk arm. The remaining benefit comes from allowing requests that can be satisfied by the read cache or the write buffer to proceed out of order. As for striping data across multiple disks to allow parallel I/O, we discover that a large stripe unit in the megabyte range is good. By striping at such a granularity across four disks, average read response time can be reduced by 40-45% over the one-disk case. Practically all the improvement stems from being able to service requests in parallel because of the extra disk arms. The short-stroking effect of using less of each disk only accounts for up to a 10-15% reduction in the service time. For the server workloads, writes

%	Resource-Poor		Resource-Rich	
	Avg. Resp. Time	Avg. Service Time	Avg. Resp. Time	Avg. Service Time
Linear Density	6.21	5.39	7.08	6.73
Track Density	3.48	3.28	3.42	3.29
Areal Density	8.58	7.97	9.31	9.07
Disk Arm (Seek Time)	3.24	3.39	3.08	3.18
Rotational Speed	5.08	5.11	5.41	5.30
Mechanical Components	8.24	8.49	8.33	8.45
Overall	15.4	14.9	15.3	15.9

Table 3.12: Performance Effect of Disk Technology Evolution at the Historical Rates. Table shows yearly improvement(|[original value - new value]|/[new value]) averaged over the five classes of workloads. Table entries are shaded to reflect the relative size of improvement with darker shades representing larger improvements.

tend to come in bursts so the write response time is improved even more by the additional disk arms.

In addition to evaluating the various I/O optimization techniques, we also analyze how the continuous improvement in disk technology affects the actual I/O performance seen by real workloads. The results are summarized in Table 3.12, which shows the yearly performance improvement that can be expected if disk technology were to continue evolving at the historical rates. In the last ten years, the average seek time of the disk has decreased by about 8% per year while the disk rotational speed has gone up by around 9% per year. At these rates of improvement, seek time reduction achieves about a 3% per year improvement in the actual response time seen by a workload while increases in rotational speed account for around 5% per year. Together, the mechanical improvements bring about an 8% improvement in performance per year. Increases in the recording density are often neglected when projecting effective disk performance. But our results clearly demonstrate that areal density improvement has as much of an impact on the effective I/O performance as the mechanical improvements. Historically, linear density increases at a rate of 21% per year while track density grows at 24% per year. Such growth rates translate into respective yearly improvement of 6-7% and 3-4% in the actual average response time, and a combined 9% per year improvement in performance. Overall, we expect the I/O performance for a given workload with a constant number of disks to increase by about 15% per year due to the evolution of disk technology. In the more realistic situation where we take advantage of the larger storage capacity of the newer disks so that the disk occupancy rate is kept constant, the yearly improvement in performance should be approximately 8%.

Because of locality of reference and request scheduling, we find that for our workloads, the average actual seek time is about 35% of the advertised average seek time for the disk, and the average actual rotational latency is about 80% of the value specified. Further analysis shows these figures to be relatively stable as disk technology evolves. We also observe that the disk spends most of its time positioning the head and very little time actually transferring data. With technology trends being the way they are, it will become increasingly difficult to effectively utilize the available disk bandwidth. Therefore, we have to consider reorganizing disk blocks in such a way that accesses become more sequential. This is the subject of the following chapter.

Chapter 4

The Automatic Improvement of Locality in Storage Systems

4.1 Synopsis

Disk I/O is increasingly the performance bottleneck in computer systems. Furthermore, managing the performance of disk-based storage has become progressively more difficult. In this chapter, we note that although disk access time has been relatively stable, disk transfer rates have risen dramatically. Given that processing power is increasingly available, we propose Automatic Locality-Improving Storage (ALIS), an introspective storage system that automatically reorganizes selected disk blocks based on the dynamic reference stream to increase the spatial locality of reference and leverage the rapidly growing disk transfer rate. ALIS is based on the observation that only a small fraction of the stored data is in active use, and there often exist long repeated read sequences. Unlike earlier work which was aimed at reducing the seek distance, ALIS focuses on eliminating physical I/Os by increasing the effectiveness of sequential prefetch, an effect unlikely to diminish over time with disk technology trends. Using trace-driven simulation with a large set of real workloads, we demonstrate that ALIS considerably outperforms prior techniques, improving the average read performance by up to 50% for server workloads and by about 15% for personal computer workloads. Since disk performance in practice is increasing only by about 8% per year [Chapter 3], the benefit of ALIS may correspond to as much as several years of technological progress.

4.2 Introduction

Processor performance has been increasing by more than 50% per year [HP96] while disk access time, being limited by mechanical delays, has improved by only about 8% per year [Gro00]. As the performance gap between the processor and the disk continues to widen, disk-based storage systems are increasingly the bottleneck in computer systems,

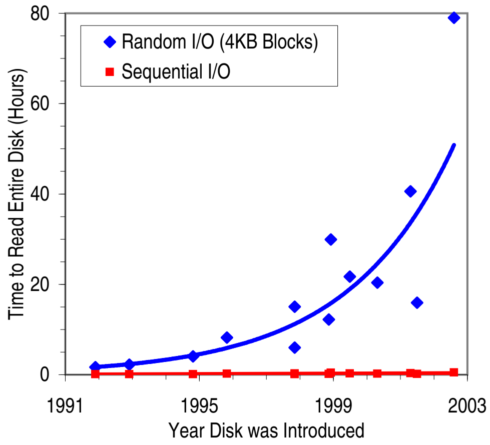


Figure 4.1: Time Needed to Read an Entire Disk as a Function of the Year the Disk was Introduced.

even in personal computers (PCs) where I/O delays have been found to highly frustrate users [Cor98]. To make matters worse, disk recording density has risen by more than 50% per year [Gro00], far exceeding the rate of decrease in access density (I/Os per second per gigabyte of data), which has been estimated to be only about 10% per year in mainframe environments [McN95]. The result is that although the disk arm is only slightly faster in each new generation of the disk, each arm is responsible for serving a lot more data. For example, Figure 4.1 shows that the time to read an entire disk using random I/O has increased from just over an hour for a 1992 disk to almost 80 hours for a disk introduced in 2002.

Although the disk access time has been relatively stable, disk transfer rates have risen by as much as 40% per year due to the increase in rotational speed and linear density [Gro00]. Given the technology and industry trends, such improvement in the transfer rate is likely to continue, as is the almost annual doubling in storage capacity. Therefore, a promising approach to increasing effective disk performance is to replicate and reorganize selected disk blocks so that the physical layout mirrors the logically sequential access. As more computing resources become available or can be added relatively easily to the storage system [Chapter 5], sophisticated techniques that accomplish this transparently, without human intervention, are increasingly possible.

The ability to autonomically [IBM01a] manage storage performance is especially attractive because of the growing cost and complexity of managing storage performance. For instance, even in the simple case of a storage system with only a single disk, its performance depends not just on where data is placed and how data is accessed, but also on what was accessed previously. The dynamics of even a relatively simple operation is further complicated by the increasingly complicated storage hierarchy with its various levels of virtualization. Therefore, in this chapter, we propose an autonomic [IBM01a] storage system that adapts itself to a given workload by automatically reorganizing selected disk blocks to improve the spatial locality of reference. We refer to such a system as Automatic Locality-Improving Storage (ALIS).

ALIS currently optimizes disk block layout based on the observation that only a portion of the stored data is in active use [Chapter 2] and that workloads tend to have long repeated sequences of reads. ALIS exploits the former by clustering frequently accessed blocks together while largely preserving the original block sequence, unlike previous techniques (*e.g.,* [AS95, BHMW98, RW91]) which fail to recognize that spatial locality exists and end up rendering sequential prefetch ineffective. For the latter, ALIS analyzes the reference stream to discover the repeated sequences from among the intermingling requests and then lays the sequences out sequentially so that they can be effectively prefetched. Trace-driven simulations using a large collection of real server and PC workloads show that ALIS considerably outperforms previous techniques to improve read performance by up to 50% and write performance by as much as 22%.

The key insight behind such impressive results is that placing data items close together to reduce the seek distance (*e.g.,* [AS95, BHMW98, RW91]) is not very effective at improving performance since it does not lessen the rotational latency, which constitutes about 40% of the read response time [Chapter 3]. Moreover, because of inertia and head settling time, there is but a relatively small time difference between a short seek and a long seek, especially with newer disks. Therefore, for ALIS, reducing the seek distance is only a secondary effect. Instead, ALIS focuses on reducing the number of physical I/Os, an effect that should not diminish over time with disk technology trends. Specifically, it locates data close together in the sequence that they are likely to be accessed so that sequential prefetch performed by the storage system to exploit the increasing disk transfer rate will likely be useful. In other words, ALIS transforms the request stream to exhibit more sequentiality.

By operating at the level of the storage system, ALIS transparently improves the performance of all I/Os, including memory-mapped I/O, paging I/O, file system metadata I/O, and other system generated I/O which may constitute well over 60% of the I/O activity in a system [RW91]. Moreover, it increases the performance of applications, even legacy ones, without requiring any changes to the operating system, file system software or application. By reducing the number of physical I/Os, ALIS is also likely to improve disk acoustics and reduce power consumption but these effects are beyond the scope of this thesis.

The rest of this chapter is organized as follows. Section 4.3 contains an overview of related work. In Section 4.4, we present the architecture of ALIS. This is followed in Section 4.5 by a discussion of the methodology used to evaluate the effectiveness of ALIS. Details of some of the algorithms are presented in Section 4.6 and are followed in Section 4.7 by the results of our performance analysis. Section 4.8 concludes this chapter. To keep the chapter focused, we highlight only portions of our results in the main text. More detailed graphs and data are presented in Appendix D. Figures and tables in the appendix are identified by a prefix D.

4.3 Background and Related Work

Various heuristics have been used to lay out data on disk so that items (*e.g.,* files) that are expected to be used together are located close to one another (*e.g.,* [MJLF84, Pea88, MK91, GK97]. The shortcoming of these *a priori* techniques is that they are based on static information such as the name space relationships of files, which may not reflect the actual reference behavior. Furthermore, files become fragmented over time. The blocks belonging to individual files can be gathered and laid out contiguously in a process known as defragmentation [McD88, Exe01]. But defragmentation does not handle inter-file access patterns and its effectiveness is limited by the file size which tends to be small [BHK+91, RLA00]. Moreover, defragmentation assumes that blocks belonging to the same file tend to be accessed together which may not be true for large files [RLA00] or database tables, and during application launch when many seeks remain even after defragmentation [Cor98].

The *posteriori* approach utilizes information about the dynamic reference behavior to arrange items. An example is to identify data items – blocks [RW91, AS95, BHMW98], cylinders [VC90], files [SGM91, Whi94, Sym01] – that are referenced frequently and to relocate them to be close together. Rearranging small pieces of data was found to be particularly advantageous [AS95] but in doing so, contiguous data that used to be accessed together could be split up. There were some early efforts to identify dependent data and to place them together [CR89, RW91], but for the most part, the previous work assumed that references are independent, which has been shown to be invalid for real workloads (*e.g.,* [Smi85, HSY01a, HSY01b]). Furthermore, the previous work did not consider the aggressive sequential prefetch common today, and was focused primarily on reducing only the seek time.

The idea of co-locating items that tend to be accessed together has been investigated in several different domains – virtual memory (*e.g.,* [Fer76]), processor cache (*e.g.,* [Hei94]), object database (*e.g.,* [TN92]) *etc.* The basic approach is to pack items that are likely to be used contemporaneously into a superunit, *i.e.,* a larger unit of data that is transferred and cached in its entirety. Such clustering is designed mainly to reduce internal fragmentation of the superunit. Thus the superunits are not ordered nor are the items within each superunit. The same approach has been tried to pack disk blocks into segments in the log-structured file system (LFS) [MRC+97]. However, storage systems in general have no convenient superunit. Therefore such clustering merely moves related items close together to reduce the seek distance. A superunit could be introduced but concern about the response time of requests in the queue behind will limit its size so that ordering these superunits will still be necessary for effective sequential prefetch.

Some researchers have also considered laying out blocks in the sequence that they are likely to be used. However, the idea has been limited to recognizing very specific patterns such as sequential, stepped sequential and reverse sequential in block address [CLTR98], and to the special case of application starts [Cor98] where the reference patterns likely to be repeated are identified with the help of external knowledge.

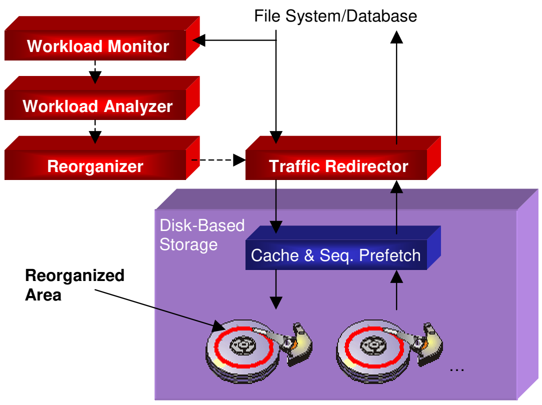


Figure 4.2: Block Diagram of ALIS.

There has also been some recent work on identifying blocks or files that tend to be used together or in a particular sequence so that the next time a context is recognized, the files and blocks can be prefetched accordingly (*e.g.,* [PZ91, GAN93, GA94, KL96, LD97]). The effectiveness of this approach is constrained by the amount of locality that is present in the reference stream, by the fact that it does not improve fetch efficiency, and by the burstiness in the I/O traffic which makes it difficult to prefetch in time.

4.4 Architecture of ALIS

ALIS consists of four major components. These are depicted in the block diagram in Figure 4.2. First, a *workload monitor* collects a trace of the addresses referenced as requests are serviced. This is a low overhead operation and involves logging four to eight bytes worth of data per request. Since the ratio of I/O traffic to storage capacity tends to be small [Chapter 2], collecting a reference trace is not expected to impose a significant overhead. For instance, we find in Chapter 2 that logging eight bytes of data per request for the Enterprise Resource Planning (ERP) workload at one of the nation's largest health insurers will create only 12 MB of data on the busiest day.

Periodically, typically when the storage system is relatively idle, a *workload analyzer* examines the reference data collected to determine which blocks should be reorganized and how they should be laid out. Because workloads tend to be bursty, there should generally be enough lulls in the storage system for the workload analysis to be performed daily [Chapter 2]. The analysis can also be offloaded to a separate machine if necessary. The workload analyzer uses two strategies, each targeted at exploiting a different workload behavior. The first strategy attempts to localize *hot*, *i.e.,* frequently accessed, data in a process that we refer to as *heat clustering*. Unlike previously proposed techniques, ALIS localizes hot data while preserving and sometimes even enhancing spatial locality. The second strategy that ALIS uses is based on the observation that there are often long read sequences or *runs* that are repeated. Thus it tries to discover these runs to lay them out sequentially so that they can be effectively prefetched. We call this approach *run clustering*. The various clustering strategies will be discussed in detail in Section 4.6.

Based on the results of the workload analysis, a *reorganizer* module makes copies of the selected blocks and lays them out in the determined order in a preallocated region of the storage space known as the *reorganized area (RA)*. This reorganization process can proceed in the background while the storage system is servicing incoming requests. The use of a specially set aside reorganization area as in [AS95] is motivated by the fact that only a relatively small portion of the data stored is in active use [Chapter 2] so that reorganizing a small subset of the data is likely to achieve most of the potential benefit. Furthermore, with disk capacities growing very rapidly [Gro00], more storage is available for disk system optimization. For the workloads that we examined, a reorganized area 15% the size of the storage used is sufficient to realize nearly all the benefit.

In general, when data is relocated, some form of directory is needed to forward requests to the new location. Because ALIS moves only a subset of the data, the directory can be simply a lookaside table mapping only the data in the reorganized area. Assuming 8 bytes are needed to map 8 KB of data and the reorganized area is 15% of the storage space, the directory size works out to be equivalent to only about 0.01% of the storage space ($15\%*8/8192 \approx 0.01\%$). The memory required for the directory can be further reduced by using well-known techniques such as increasing the granularity of the mapping or restricting the possible locations that a block can be mapped to. The directory can also be paged. Such actions may, however, affect performance.

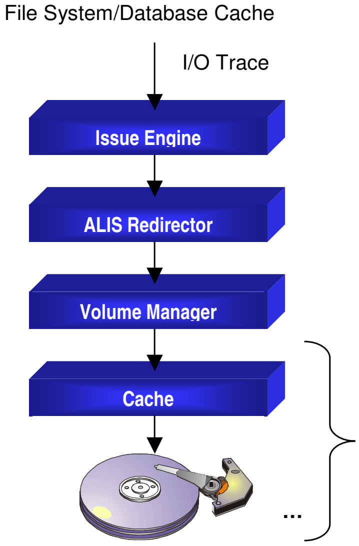
Note that there may be multiple copies of a block in the reorganized area because a given block may occur in the heat-clustered region and in multiple runs. The decision of which copy to fetch, either original or one of the duplicates in the reorganized area is determined by the *traffic redirector* which sits on the I/O path. For every read request, the traffic redirector looks up the directory to determine if there are any up-to-date copies of the requested data in the reorganized area. If there is more than one up-to-date copy of a block in the system, the traffic redirector can select the copy to fetch based on the estimated proximity of the disk head to each of the copies and the expected prefetch benefit. A simple strategy that works well in practice is to give priority to fetching from the runs. If no run matches, we proceed to the heat-clustered data, and if that fails, the original copy of the data is fetched. We will discuss the policy of deciding which copy to select in greater detail in Section 4.6.

For reliability, the directory is stored and duplicated in known, fixed locations on disk. The on-disk directory is updated only during the process of block reorganization. When writes to data that have been replicated and laid out in the reorganized area occur, one or more copies of the data have to be updated. Any remaining copies are invalidated. We will discuss which copy or copies to update later in Section 4.7.3. It suffices here to say that such update and invalidate information is maintained in addition to the directory. At the beginning of the block reorganization process, any updated blocks in the reorganized region are copied back to the home or original area. Since there is always a copy of the data in the home area, it is possible to make the reorganization process resilient to power failures by using an intentions list. With care, block reorganization can be performed while access to data continues.

The on-disk directory is read on power-up and kept static during normal operation. The update or invalidate information is, however, dynamic. Losing the memory copy of the directory is thus not catastrophic, but having non-volatile storage (NVS) would make things simpler for maintaining the update/invalidate information. Without NVS, a straightforward approach is to periodically write the update/invalidate information to disk. When the system is first powered up, it checks to see if it was shut down cleanly the previous time. If not, some of the update/invalidate information may have been lost. The update/invalidate information in essence tracks the blocks in the reorganized area that have been updated or invalidated since the last reorganization. Therefore, if the policy of deciding which blocks to update and which to invalidate is based on regions in the reorganized area, copying all the blocks in the update region back to the home area and copying all the blocks from the home area to the invalidate region effectively resets the update/invalidate information.

ALIS can be implemented at different levels in the storage hierarchy, including the disk itself, if predictions about embedding intelligence in disks [Gra98, KPH98b, RGF98] come true. We are particularly interested in the storage adaptor and the outboard controller, which can be attached to a storage area network (SAN) or an internet protocol network (NAS), because they provide a convenient platform to host significant resources for ALIS, and the added cost can be justified, especially for high performance controllers that are targeted at the server market and which are relatively price-insensitive [Chapter 5]. For the personal systems, a viable alternative is to implement ALIS in the disk device driver. More generally, ALIS can be thought of as a layer that can be interposed somewhere in the storage stack. ALIS does not require a lot of knowledge about the underlying storage system. So far, we have simply assumed that the storage system downstream is able to service requests that exhibit sequentiality much more efficiently than random requests. This is a characteristic true of all disk-based storage systems and it turns out to be extremely powerful, especially in view of the disk technology trends.

Note that some storage systems such as RAID (Redundant Array of Inexpensive Disks) [CLG$^+$94] adaptors and outboard controllers implement a virtualization layer or a virtual to physical block mapping so that they can aggregate multiple storage pools to present a flat storage space to the host. ALIS assumes that the flat storage space presented by these storage systems performs largely like a disk so that virtually sequential blocks can be accessed much more efficiently than random blocks. This assumption tends to be valid because, for practical reasons such as to reduce overhead, any virtual to physical block mapping is typically done at the granularity of large extents so that virtually sequential blocks are likely to be also physically sequential. Moreover, file systems and applications have the same expectation of the storage space so it behooves the storage system to ensure that the expectation is met.



	Resource-Poor	Resource-Rich
Read Caching	8MB per disk, Least-Recently-Used (LRU) replacement.	1% of storage used, Least-Recently-Used (LRU) replacement.
Prefetching	32KB read-ahead. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks.	Conditional sequential prefetch, 16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2. Preemptible read-ahead up to maximum prefetch of 128KB, read any free blocks, 8MB per disk opportunistic prefetch buffer.
Write Buffering	4MB, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 30s age limit.	0.1% of storage used, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 1 hour age limit.
Request Scheduling	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.	Shortest Access Time First with age factor = 0.01 (ASATF(0.01)), queue depth of 8.
Striping	Stripe unit of 2MB.	Stripe unit of 2MB.

Figure 4.3: Block Diagram of Simulation Model Showing the Optimized Parameters [Chapter 3] for the Underlying Storage System.

4.5 Performance Evaluation Methodology

We apply the methodology developed in Chapter 3 to evaluate the effectiveness of ALIS. The methodology is essentially trace-driven simulation with an improved method for replaying the traces that allows us to model both the feedback effect between request completion and subsequent I/O arrivals, and the burstiness in the I/O traffic. We could have simply played back the trace maintaining the original timing as in [RW91] but that would result in optimistic performance results for ALIS because it would mean more free time for prefetching and fewer opportunities for request scheduling than in reality. The same traces as in Chapter 3 are used here.

4.5.1 Simulation Model

The simulator that we use to quantify the benefit of ALIS is based on that described in detail in the previous chapter. Its major components are presented in Figure 4.3.

A wide range of techniques such as caching, prefetching, write buffering, request scheduling and striping have been invented for optimizing I/O performance. Each of these optimizations can be configured with different policies and parameters, resulting in a huge design space for the storage system. In Chapter 3, we systematically explore the entire design space to establish the effectiveness of the various techniques for real workloads, and to determine the best practices for each technique. Here, we leverage our previous results and use the optimal settings we derived to set up the baseline storage system for evaluating the effectiveness of ALIS. As in the preceding chapter, we consider two reasonable configurations the parameters of which are summarized in Figure 4.3.

Recall that for workloads with multiple disk volumes, we concatenate the volumes to create a single address space. Each workload is fitted to the smallest disk from the IBM Ultrastar 73LZX [IBM01b] family that is bigger than the total volume size, leaving a headroom of 20% for the reorganized area. When we scale the capacity of the disk and require more than one disk to hold the data, we stripe the data using the previously determined stripe size of 2 MB. We do not simulate RAID protection since it is for the most part orthogonal to ALIS.

As we shall see, infrequent block reorganization is sufficient to realize most of the benefit of ALIS. In addition, our analysis of the workloads reveals that there are relatively idle periods during which the reorganization can be performed [Chapter 2]. Therefore, in this study, we make the simplifying assumption that block layout can be changed instantaneously, as in previous work (*e.g.,* [RW91]). Later in Section 4.7.4, we will empirically validate this assumption.

4.5.2 Performance Metrics

I/O performance can be measured at different levels in the storage hierarchy. In order to fully quantify the effect of ALIS, we measure performance from when requests are issued to the storage system, before they are potentially broken up by the ALIS redirector or the volume manager for requests that span multiple disks. The two important metrics in I/O performance are *response time* and *throughput*. As in the previous chapter, we approximate throughput by the reciprocal of the average *service time*, which we define as the average amount of time the disk arm is busy per request, deeming the disk arm to be busy both when it is being moved into position to service a request and when it has to be kept in position to transfer data.

Recall that the main benefit of ALIS is to transform the request stream so as to increase the effectiveness of sequential prefetch performed by the storage system downstream. To gain insight into this effect, we also examine the read *miss ratio* of the cache (and prefetch buffer) in the storage system. The read miss ratio is defined as the fraction of read requests that are not satisfied by the cache (or prefetch buffer), or in other words, the fraction of requests that requires physical I/O. It should take on a lower value when sequential prefetch is more effective.

Note that we tend to care more about read response time and less about write response time because write latency can often be effectively hidden by write buffering [Chapter 3]. In fact, write buffering can also dramatically improve write throughput. Moreover, because workloads tend to be bursty [Chapter 2], the physical writes can generally be deferred until the system is relatively idle. On the other hand, despite predictions to the contrary, both measurements of real systems (*e.g.,* [BHK$^+$91]) and simulation studies (*e.g.,* [Chapter 3] and [DMW$^+$94, HSY01b]) show that large caches, while effective, have not eliminated read traffic.

	Resource-Poor				Resource-Rich			
	Average Read Response Time (ms)	Average Read Service Time (ms)	Read Miss Ratio	Average Write Service Time (ms)	Average Read Response Time (ms)	Average Read Service Time (ms)	Read Miss Ratio	Average Write Service Time (ms)
P-Avg.	3.34	2.22	0.431	1.41	2.66	1.79	0.313	0.700
S-Avg.	2.67	1.91	0.349	1.32	1.20	0.886	0.167	0.535
Ps-Avg.	3.83	2.18	0.450	1.05	3.15	1.75	0.319	0.681
Pm	3.14	2.23	0.447	1.30	1.65	1.24	0.226	0.474
Sm	3.37	2.67	0.468	1.57	1.38	1.10	0.204	0.855

Table 4.1: Baseline Performance Figures.

Therefore in this chapter, we focus primarily on the read response time, read service time, read miss ratio, and to a lesser extent, on the write service time. In particular, we look at how these metrics are improved with ALIS where improvement is defined as $(value_{old} - value_{new})/value_{old}$ if a smaller value is better and $(value_{new} - value_{old})/value_{old}$ otherwise. We use the performance figures obtained previously in Chapter 3 as the baseline numbers. These are summarized in Table 4.1.

4.6 Clustering Strategies

In this section, we present in detail various techniques for deciding which blocks to reorganize and how to lay these blocks out relative to one another. We refer to these techniques collectively as clustering strategies. We will use empirical performance data to motivate the various strategies and to substantiate our design and parameter choices. General performance analysis of the system will appear in the next section.

4.6.1 Heat Clustering

In Chapter 2, we observe that only a relatively small fraction of the data stored on disk is in active use. The rest of the data are simply there, presumably because disk storage is the first stable or non-volatile level in the memory hierarchy, and the only stable level that offers online convenience. Given the exponential increase in disk capacity, the tendency is to be less careful about how disk space is used so that data will be increasingly stored on disk just in case they will be needed. Figure 4.4 depicts the typical situation in a storage system. Each square in the figure represents a block of data and the darkness of the square reflects the frequency of access to that block of data. The squares are arranged in the sequence of the corresponding block address such that the square at the extreme right of a row immediately precedes that at the extreme left of the next row. There are some hot or frequently accessed blocks and these are distributed throughout the storage system.

Accessing such active data requires the disk arm to seek across a lot of inactive or cold data, which is clearly not the most efficient arrangement.

This observation suggests that we should try to determine the active blocks and cluster them together so that they can be accessed more efficiently with less physical movement. As discussed in Section 4.3, over the last two decades, there have been several attempts to improve spatial locality in storage systems by clustering together hot data [VC90, SGM91, RW91, Whi94, AS95, BHMW98, Sym01]. We refer to these schemes collectively as *heat clustering*. The basic approach is to count the number of requests directed to each unit of reorganization over a period of time, and to use the counts to identify the frequently accessed data and to rearrange them using a variety of block layouts. For the most part however, the previously proposed block layouts fail to effectively account for the fact that real workloads exhibit sequentiality, and that there is a dramatic and increasing difference between random and sequential disk performance.

Organ Pipe and Heat Placement

For instance, previous work rely almost exclusively on the *organ pipe layout* [Knu98] in which the most frequently accessed data is placed at the center of the reorganized area, the next most frequently accessed data is placed on either side of the center, and the process continues alternating between the two sides of the center until the least-accessed data has been placed at the edges of the reorganized region. This is illustrated in Figure 4.4. If we visualize the block access frequencies in the resulting arrangement, we get an image of organ pipes, hence the name.

Considering the disk as a 1-dimensional space, the organ pipe heuristic minimizes disk head movement under the conditions that data is demand fetched, and that the references are derived from an independent random process with a known fixed distribution and are handled on a first-come-first served (FCFS) basis [Knu98]. However, disks are really 2-dimensional in nature, and the cost of moving the head is not an increasing function of the distance moved. A small backward movement would, for example, require almost an entire disk revolution. Furthermore, storage systems today perform aggressive prefetch and request scheduling, and in practice, data references are not independent nor are they drawn from a fixed distribution. See for instance [Smi85, HSY01a, HSY01b] where real workloads are shown to clearly generate dependent references. In other words, the organ pipe arrangement is not optimal in practice and may in fact end up splitting contiguous data, thereby rendering sequential prefetch ineffective and causing some requests to require multiple I/Os. This is especially the case when the unit of reorganization is small, as was recommended previously (*e.g.,* [AS95]) in order to cluster hot data as closely as possible.

In Figures 4.5 and D.1, we present the performance effect of heat clustering with the organ pipe layout on our various workloads. These figures assume that reorganization is performed daily and that the reorganized area is 10% the size of the total volume and is located at a byte offset 30% into the volume with the volume laid out inwards from the outer edge of the disk. We will evaluate the sensitivity to these parameters in Section 4.7. As

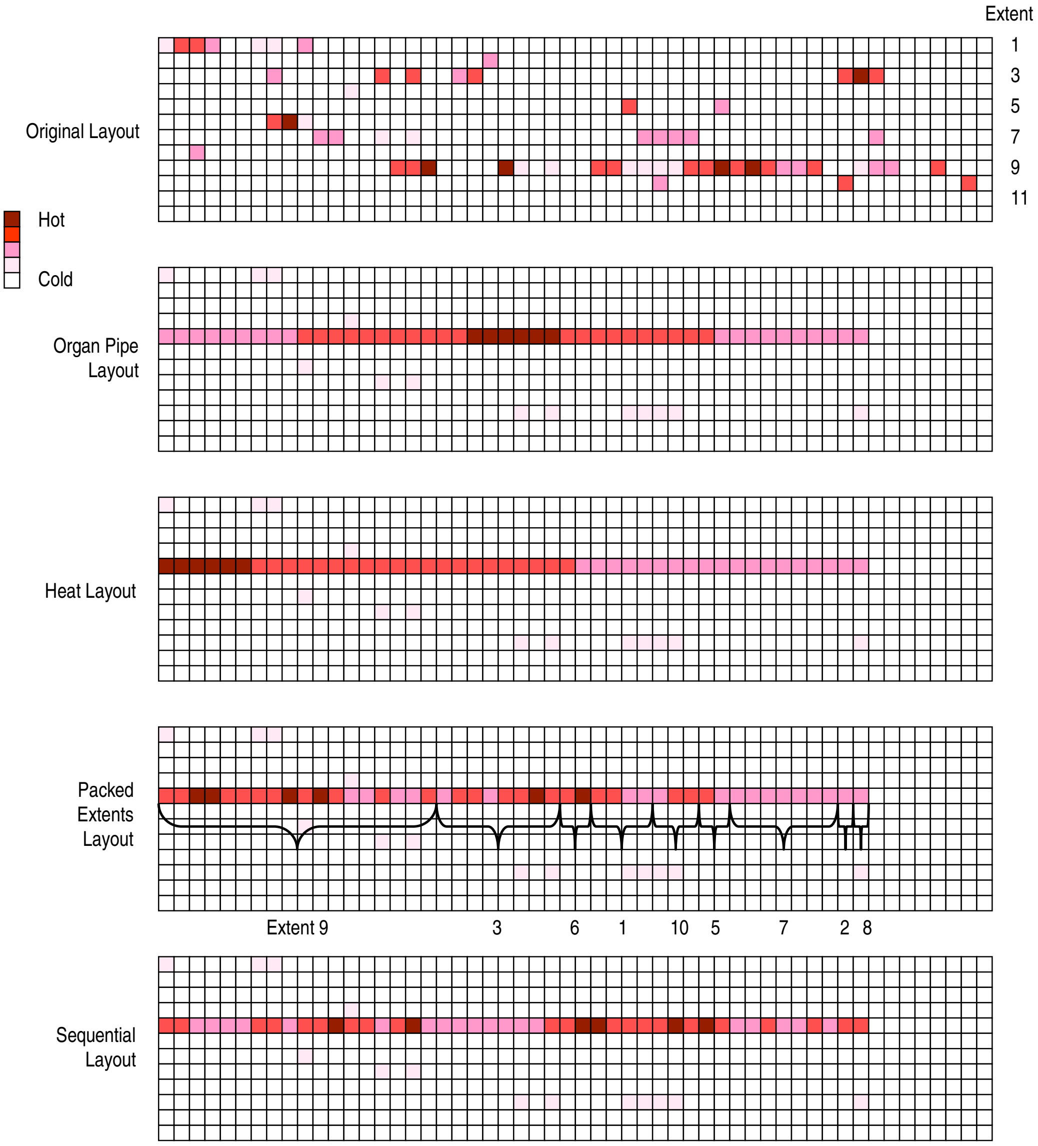


Figure 4.4: Block Layout Strategies for Heat Clustering.

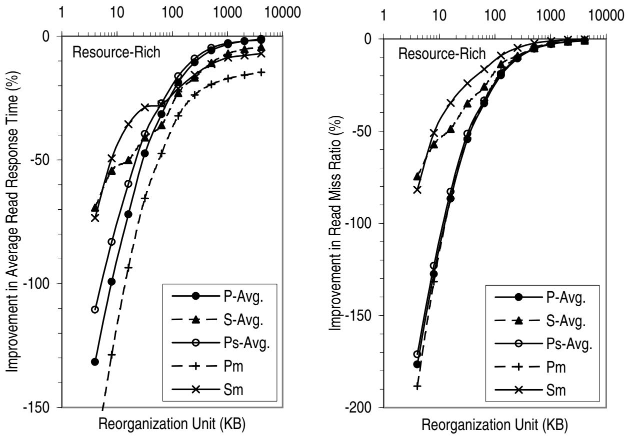


Figure 4.5: Effectiveness of Organ Pipe Placement at Improving Read Performance (Resource-Rich).

discussed, the organ pipe heuristic performs very poorly. Contrary to previous recommendations, a small reorganization unit is especially bad, dramatically degrading performance for all the workloads. Observe that with larger reorganization units, the performance degradation is reduced. This is because spatial locality is preserved within the reorganization unit. In the limit, the organ pipe layout converges to the original block layout and achieves performance parity with the base unreorganized case.

To prevent the disk arm from having to seek back and forth across the center of the organ pipe arrangement, an alternative is to arrange the hot reorganization units in decreasing order of their heat or frequency counts. In such an arrangement, the most frequently accessed reorganization unit is placed first, followed in order by the next most frequently accessed unit, and so on. This is referred to as *heat layout* in Figure 4.4. As shown in Figures D.2 and D.3, the heat layout, while better than the organ pipe layout, still degrades performance substantially for all but the largest reorganization units.

Link Closure Placement

An early study did identify the problem that when sequentially accessed data is split on either side of the organ pipe arrangement, the disk arm has to seek back and forth across the disk resulting in decreased performance [RW91]. A technique of maintaining forward and backward pointers from each reorganization unit to the reorganization unit most likely to precede and succeed it was proposed. This scheme is similar to the heat layout except that when a reorganization unit is placed, the *link closure* formed by following the forward and backward pointers is placed at the same time. As shown in Figures 4.6 and D.4, this scheme, though better than the pure organ pipe heuristic, still suffers from the same problems because it is not accurate enough at identifying data that tend to be accessed in sequence.

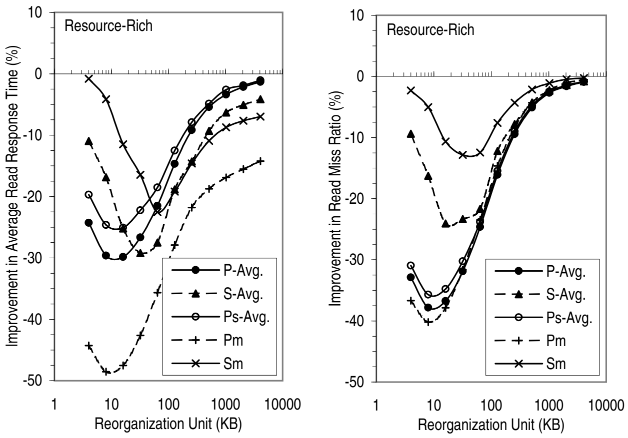


Figure 4.6: Effectiveness of Link Closure Placement at Improving Read Performance (Resource-Rich).

Packed Extents and Sequential Layout

Since only a portion of the stored data is in active use, clustering the hot data, in effect short-stroking or using a small fraction of the disk, should yield a substantial improvement in performance. The key to realizing this potential improvement is to recognize that there is some existing spatial locality in the reference stream, especially since the original block layout is the result of many optimizations (see Section 4.3). When clustering the hot data, we should therefore attempt to preserve the original block sequence, particularly when there is aggressive read-ahead as is the case today.

With this insight, we develop a block layout strategy called *packed extents*. As before, we keep a count of the number of requests directed to each unit of reorganization over a period of time. During reorganization, we first identify the n reorganization units with the highest frequency count, where n is the number of reorganization units that can fit in the reorganized area. These are the target units, *i.e.,* the units that should be reorganized. Next, the storage space is divided into extents each the size of many reorganization units. These extents are sorted based on the highest access frequency of the reorganization units within each extent. If there is a tie, the next highest access frequency is compared. For instance, if the extent size is a row of blocks in Figure 4.4 and the extents are numbered from top to bottom, the sorted extent list is 9,3,6,1,10,5,7,2,8. In the packed extents layout, the target reorganization units are arranged in the reorganized region in ascending order of their extent rank in the sorted extent list, and their offset within the extent. In the example of Figure 4.4, there are 18 target units in extent nine and eight in extent three. Because extent nine and extent three are respectively the first and second extents in the sorted extent list, the 18 target units of extent nine are laid out in address sequence before the eight target units in extent three, and so on. The packed extents layout essentially packs hot data together while preserving the sequence of the data within each extent, hence its name.

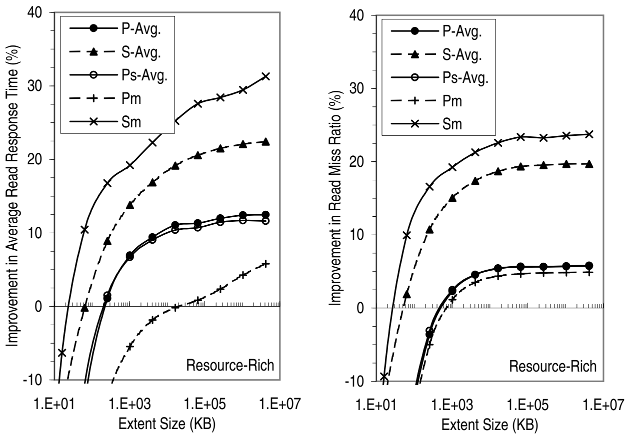


Figure 4.7: Effectiveness of Packed Extents Layout at Improving Read Performance (Resource-Rich).

The main effect of the packed extents layout is to reduce seek distance without decreasing prefetch effectiveness. By moving data that are seldom read out of the way, it actually also improves the effectiveness of sequential prefetch, as can be seen by the reduction in the read miss ratio in Figures 4.7 and D.5, which assume a 4 KB reorganization unit. Observe from the figures that this scheme turns out to perform very well for large extents, improving average read response time in the resource-rich environment by up to 12% and 31% for the PC and server workloads respectively. That the performance increases with extent sizes up to the gigabyte range implies that for laying out the hot reorganization units, preserving existing spatial locality is more important than concentrating the heat.

This suggests that simply arranging the target reorganization units in increasing order of their original block address should work well. Such a *sequential layout* is the special case of packed extents with a single extent. While straightforward, the sequential layout tends to be sensitive to the original block layout, especially to user/administrator actions such as the order in which workloads are migrated or loaded onto the storage system. But for our workloads, the sequential layout works well. Observe from Figures 4.8 and D.6 that with a reorganization unit of 4KB, the average read response time is improved by up to 29% in the resource-poor environment and 31% in the resource-rich environment. It turns out that sequential layout was considered in [AS95] where it was reported to perform worse than the organ pipe layout. The reason was that the earlier work did not take into account the aggressive caching and prefetching common today, and was focused primarily on reducing the seek time. In this chapter, unless explicitly mentioned otherwise, we use the sequential layout for heat clustering.

To increase stability in the effectiveness of heat clustering, the reference counts can be aged such that

$$Count_{new} = \alpha Count_{current} + (1 - \alpha) Count_{old} \tag{4.1}$$

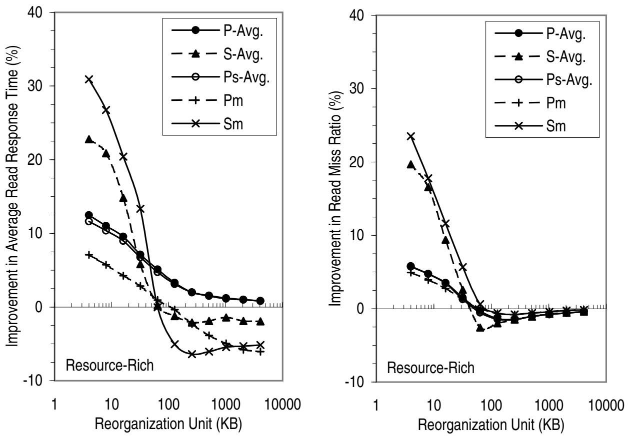


Figure 4.8: Effectiveness of Sequential Layout at Improving Read Performance (Resource-Rich).

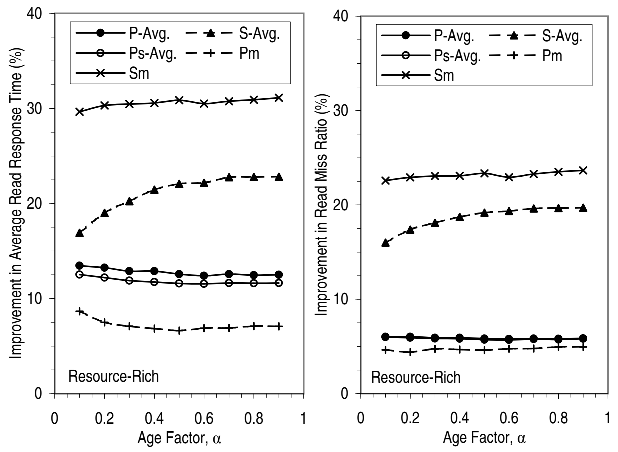


Figure 4.9: Sensitivity of Heat Clustering to Age Factor, α (Resource-Rich).

where $Count_{new}$ is the reference count used to drive the reorganization, $Count_{current}$ is the reference count collected since the last reorganization and $Count_{old}$ is the previous value of $Count_{new}$. The parameter α controls the relative weight placed on the current reference counts and those obtained in the past. For example, with an α value of 1, only the most recent reference counts are considered. In Figures 4.9 and D.9, we study how sensitive performance with the sequential layout is to the value of the parameter α. The PC workloads tend to perform slightly better for smaller values of α, meaning when more history is taken into account. The opposite is true, however, of the server workloads on average but both effects are small. As in [RW91], all the results in this chapter assume a value of 0.8 for α unless otherwise indicated.

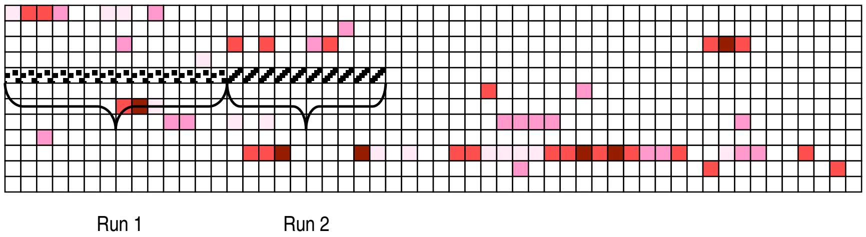


Figure 4.10: Block Layout with Run Clustering.

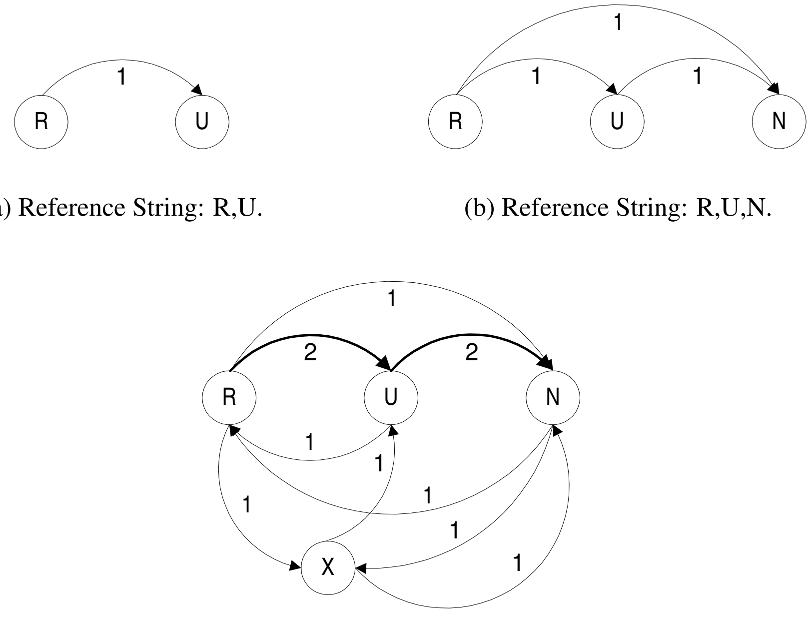
4.6.2 Run Clustering

Our analysis of the various workloads also reveals that the reference stream contains long read sequences that are often repeated. The presence of such repeated read sequences or *runs* should not be surprising since computers are frequently asked to perform the same tasks over and over again. For instance, PC users tend to use a core set of applications, and each time the same application is launched, the same set of files [ZS99] and blocks [Cor98] are read. The existence of runs suggest a clustering strategy that seeks to identify these runs so as to lay them out sequentially in the reorganized area. We refer to this strategy as *run clustering*. Figure 4.10 illustrates the basic idea behind run clustering assuming that two runs are discovered. Note that the blocks in each of the runs are laid out in the reorganized region in the discovered sequence so that they can be effectively prefetched the next time the same sequence is encountered.

Representing Access Patterns

The reference stream contains a wealth of information. The first step in run clustering is to extract relevant details from the reference stream and to represent the extracted information compactly and in a manner that facilitates analysis. This is accomplished by building an access graph where each vertex represents a unit of reorganization and the weight of an edge from vertex i to vertex j represents the desirability for reorganization unit j to be located close to and after reorganization unit i. For example, a straightforward method for building the access graph is to set the weight of edge $i \rightarrow j$ equal to the number of times reorganization unit j is referenced *immediately after* unit i. But this method represents only pair-wise patterns. Moreover, at the storage level, any repeated pattern is likely to be interspersed by other requests because the reference stream is the aggregation of many independent streams of I/O, especially in multi-tasking and multi-user systems. Furthermore, the I/Os may not arrive at the storage system in the order they were issued because of request scheduling or prefetch. Therefore, the access graph should represent not only sequences that are exactly repeated but also those that are *largely* or *pseudo* repeated.

Such an access graph can be constructed by setting the weight of edge $i \rightarrow j$ equal to the number of times reorganization unit j is referenced *shortly after* accessing unit i, or more specifically the number of times unit j is referenced within some number of references,



(a) Reference String: R,U.

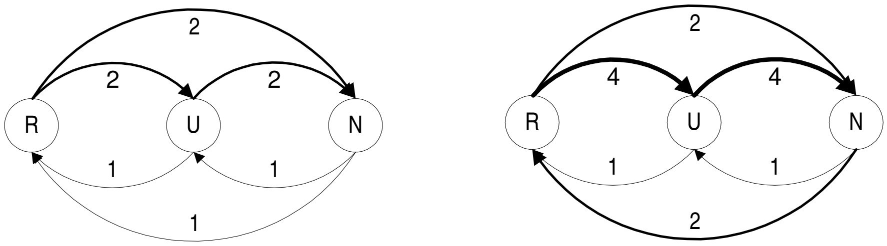
(b) Reference String: R,U,N.

(c) Reference String: R,U,N,R,X,U,N.

Figure 4.11: Access Graph with a Context Size of Two.

τ, of accessing unit i [MSNO74]. We refer to τ as the *context size*. As an example, Figure 4.11(a) illustrates the graph that represents a sequence of two requests where the first request is for reorganization unit R and the second is for unit U. In Figure 4.11(b), we show the graph when an additional request for unit N is received. The figures assume a context size of two. Therefore, edges are added from both unit R and unit U to unit N. Figure 4.11(c) further illustrates the graph after an additional four requests for data are received in the sequence R, X, U, N. The resulting graph has three edges of weight two among the other edges of weight one. These edges of higher weight highlight the largely repeated sequence R, U, N. This example shows that by choosing an appropriate value for the context size or τ, we can effectively filter out intermingled references. We will investigate good values for τ for our workloads later.

An undesirable consequence of having the edge weight represent the number of times a reorganization unit is referenced *within* τ references of accessing another unit is that we lose information about the exact sequence in which the references occur. For instance, Figure 4.12(a) depicts the graph for the reference string R,U,N,R,U,N. Observe that reorganization unit R is equally connected to unit U and unit N. The edge weights indicate that units U and N should be laid out after unit R but they do not tell the order in which these two units should be arranged. We could potentially figure out the order based on the edge of weight two from unit U to unit N. But to make it easier to find repeated sequences, the



(a) Uniform Edge Weight. (b) Graduated Edge Weight.

Figure 4.12: The Effect of Graduated Edge Weight (Reference String = R,U,N,R,U,N, Context Size = 2).

actual sequence of reference can be more accurately represented by employing a graduated edge weight scheme where the weight of edge $i{\rightarrow}j$ is a decreasing function of the number of references between when those two data units are referenced. For instance, suppose X_i denotes the reorganization unit referenced by the i-th read. For each X_n, we add an edge of weight $\tau - j + 1$ from X_{n-j} to X_n, where $j \leq \tau$. In the example of Figure 4.11(b), we would add an edge of weight *one* from unit R to unit N and an edge of weight *two* from unit U to unit N. Figure 4.12(b) shows that with such a graduated edge weight scheme, we can readily tell that unit R should be immediately followed by unit U when the reference string is R,U,N,R,U,N.

More generally, we can use the edge weight to carry two pieces of information – the number of times a reorganization unit is accessed within τ references of another, and the distance or number of intermediate references between when these two units are accessed. Suppose f is a parameter that determines the fraction of edge weight devoted to representing the distance information. Then for each X_n, we add an edge of weight $1 - f + f * \frac{\tau - j + 1}{\tau}$ from X_{n-j} to X_n, where $j \leq \tau$. We experimented with varying the weighting of these two pieces of information and found that $f = 1$ tends to work better although the difference is small (Figure D.10). The effect of varying f is small because our run discovery algorithm (to be discussed later) is able to determine the correct reference sequence most of the time, even without the graduated edge weights.

In Figures 4.13 and D.11, we analyze the effect of different context sizes on our various workloads. The context size should generally be large enough to allow pseudo repeated reference patterns to be effectively distinguished. For our workloads, performance clearly increases with the context size and tends to stabilize beyond about eight. Unless otherwise indicated, all the results in this chapter assume a context size of nine. Note that the reorganization units can be of a fixed size but the results presented in Figures 4.14 and D.12 suggest that this is not very effective. In ALIS, each reorganization unit represents the data referenced in a request. By using such variable-sized reorganization units, we are able to achieve much better performance since the likelihood for a request to be split into multiple

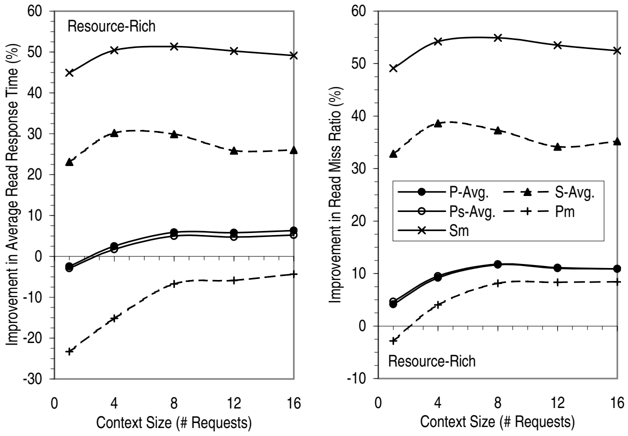


Figure 4.13: Sensitivity of Run Clustering to Context Size, τ (Resource-Rich).

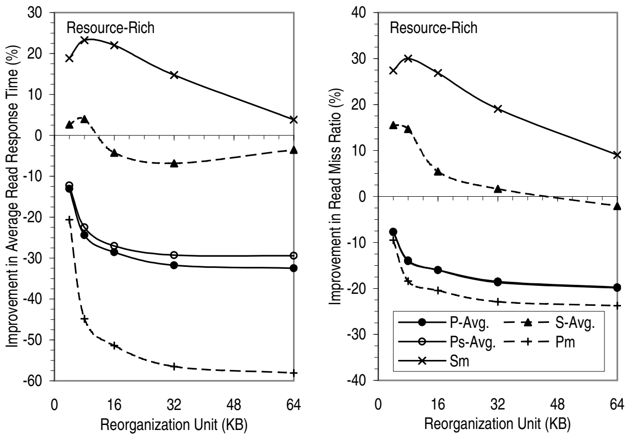


Figure 4.14: Effectiveness of Run Clustering with Fixed-Sized Reorganization Units (Resource-Rich).

I/Os is reduced. Prefetch effectiveness is also enhanced because internal fragmentation is avoided so that any discovered sequence is likely to contain only the data that is actually referenced. In addition, this approach allows the same data block to potentially appear in multiple runs, and helps to distinguish among different access sequences that include the same block.

Various pruning algorithms can be used to limit the size of the graph. In this thesis, we remove the vertices and edges with weight below some threshold which we set at the respective 10th percentile. In other words, we remove vertices weighing less than 90% of all the vertices and edges with weight in the bottom 10% of all the edges. The weight of a vertex is defined as the weight of its heaviest edge. This simple *bottom pruning* policy adds no additional memory overhead and preserves the relative connectedness of the vertices so that the algorithm for discovering the runs is not confused when it tries to determine how

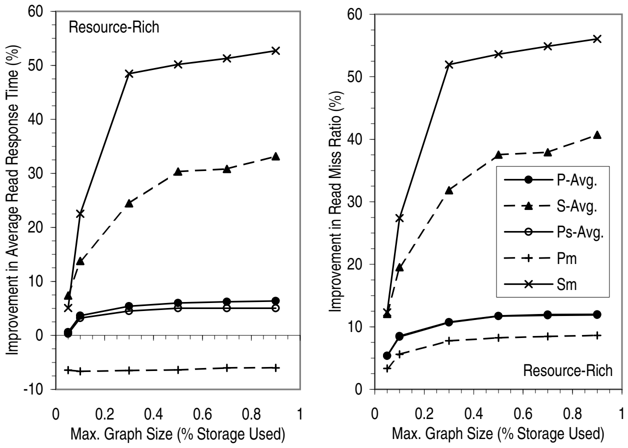


Figure 4.15: Sensitivity of Run Clustering to Graph Size (Resource-Rich).

to sequence the reorganization units. To model the graph size, we tabulated the size of the vertex and adjacency list structures. The vertex structure requires the following fields - vertex ID (6 bytes), a pair of adjacency list pointers (2x4 bytes), pointer to next vertex (4 bytes), status byte (1 byte). Note that the graph is directed so we need a pair of adjacency lists for each vertex to be able to quickly determine both the incoming and the outgoing edges. Accordingly, there are two adjacency list entries (an outgoing and an incoming) for each edge. Each of these entries consists of the following fields – pointer to vertex (4 bytes), weight (2 bytes), pointer to next edge (4 bytes). Therefore each vertex in the graph requires 19 bytes of memory while each edge occupies 20 bytes.

Figures 4.15 and D.13 show the performance improvement that can be achieved as a function of the size of the graph. Observe from the figures that a graph smaller than 0.5% of the storage used is sufficient to realize most of the benefit of run clustering. This is the default graph size we use for the simulations reported in this chapter. Memory of this size should be available when the storage system is relatively idle because caches larger than this are needed to effectively hold the working set [Chapter 3]. A multiple-pass run clustering algorithm can be used to further reduce memory requirements. Note that high-end storage systems today host many terabytes of storage but the storage is used for different workloads and are partitioned into logical subsystems or volumes. These volumes can be individually reorganized so that the memory required at any one time for run clustering is greatly reduced from 0.5% of the total storage in use.

An idea for speeding up the graph build process and reducing the graph size is to pre-filter the reference stream to remove requests that do not occur frequently. The basic idea is to keep reference counts for each reorganization unit as in the case of heat clustering. In building the access graph, we ignore all the requests whose reference count falls below some threshold. Our experiments show that pre-filtering tends to reduce the performance gain (Figure D.14) because it is, for the most part, not as accurate as the graph pruning

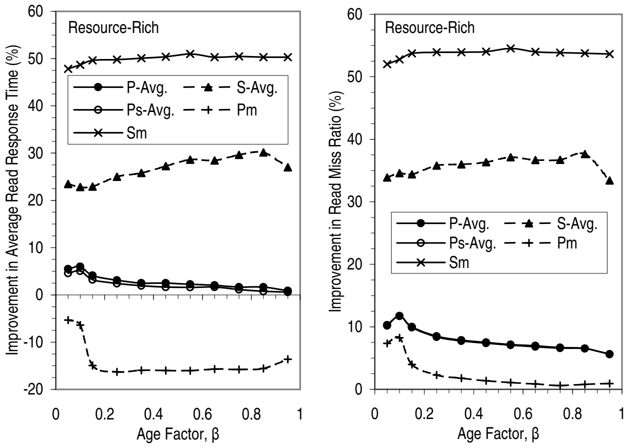


Figure 4.16: Sensitivity of Run Clustering to Age Factor, β (Resource-Rich).

process in removing uninteresting information. But in cases where we are constrained by the graph build time, it could be a worthwhile option to pursue.

As in heat clustering, we age the edge weights to provide some continuity and avoid any dramatic fluctuations in performance. Specifically, we set

$$Weight_{new} = \beta Weight_{current} + (1 - \beta) Weight_{old} \tag{4.2}$$

where $Weight_{new}$ is the edge weight used in the reorganization, $Weight_{current}$ is the edge weight collected since the last reorganization and $Weight_{old}$ is the previous value of $Weight_{new}$. The parameter β controls the relative weight placed on the current edge weight and those obtained in the past. In Figures 4.16 and D.15, we study how sensitive run clustering is to the value of the parameter β. Observe that as in heat clustering, the workloads are relatively stable over a wide range of β values with the PC workloads performing better for smaller values of β, meaning when more history is taken into account, and the server workloads preferring larger values of β. Such results reflect that fact that the PC workloads are less intense and have reference patterns that are repeated less frequently so that it is useful to look further back into history to find these patterns. This is especially the case for the merged PC workloads where the reference pattern of a given user can quickly become aged out before it is encountered again.

Throughout the design of ALIS, we try to desensitize its performance to the various parameters so that it is not catastrophic for somebody to "configure the system wrongly". To reflect the likely situation that ALIS will be used with a default setting, we base our performance evaluation on parameter settings that are good for an entire class of workloads rather than on the best values for each individual workload. Therefore, the results in this chapter assume a default β value of 0.1 for all the PC workloads and 0.8 for all the server workloads. A useful piece of future work would be to devise ways to set the various parameters dynamically to adapt to each individual workload. Figures 4.16 and D.15 suggest that the approach of using hill-climbing to gradually adjust the value of β until a local optimum is reached should be very effective because the local optimum is also the global optimum. This characteristic is generally true for the parameters in ALIS.

Mining Access Patterns

The second step in run clustering is to analyze the access graph to discover desirable sequences of reorganization units, which should correspond to the runs in the reference stream. This process is similar to the graph-theoretic clustering problem with an important twist that we are interested in the sequence of the vertices. Let G be an access graph built as outlined above and R, the target sequence or run. We use $|R|$ to denote the number of elements in R and $R[i]$, the ith element in R. By convention we refer to $R[1]$ as the front of R and $R[|R|]$ as the back of R. The following outlines the algorithm that ALIS uses to find R.

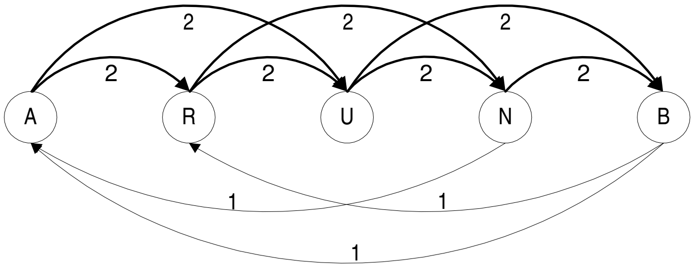
1. Find the heaviest edge linking two unmarked vertices.
2. Initialize R to the heaviest edge found and mark the two vertices.
3. Repeat
4. Find an unmarked vertex u such that $headweight = \sum_{i=1}^{Min(\tau,|R|)} Weight(u, R[i])$ is maximized.
5. Find an unmarked vertex v such that $tailweight = \sum_{i=1}^{Min(\tau,|R|)} Weight(R[|R|-i+1], v)$ is maximized.
6. if $headweight > tailweight$
7. Mark u and add it to the front of R.
8. else
9. Mark v and add it to the back of R.
10. Goto Step 3.

In steps 1 and 2, we initialize the target run by the heaviest edge in the graph. Then in steps 3 to 10, we inductively grow the target run by adding a vertex at a time. In each iteration of the loop, we select the vertex that is most strongly connected to either the $head$ or $tail$ of the run, the $head$ or $tail$ of the run being, respectively, the first and last τ members of the run and τ is the context size used to build the access graph. Specifically, in step 4, we find a vertex u such that the weight of all its edges incident on the vertices in the $head$ is the highest. Vertex u represents the reorganization unit that we believe is most likely to immediately precede the target sequence in the reference stream. Therefore, if we decide to include it in the target sequence, we will add it as the first entry (Step 7). Similarly, in step 5, we find a vertex v that we believe is most likely to immediate follow the target run in the reference stream. The decision of which vertex u or v to include in the target run is made greedily. We simply select the unit that is most likely to immediate precede or follow the target sequence, *i.e.,* the vertex that is most strongly connected to the respective ends of the target sequence (Step 6).

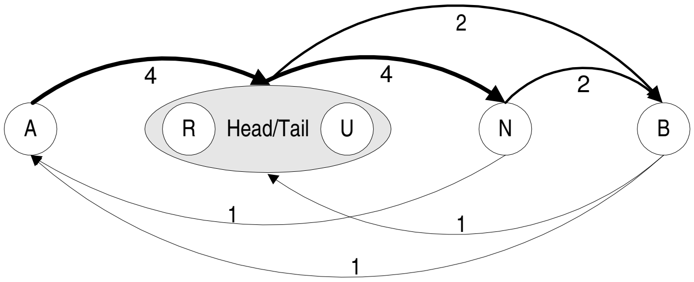
By selecting the next vertex to include in the target run based on its connectedness to the first or last τ members of the run, the algorithm is following the way the access graph is built using a context of τ references to recover the original reference sequence. For instance, in Figure 4.17, we show the operation of the algorithm on a graph for the reference string A,R,U,N,B,A,R,U,N,B. For simplicity, we assume that we use uniform edge weights and the context size is two. Without loss in generality, suppose we pick the edge $R \rightarrow U$ in step 1. Since the target sequence has only two entries at this point, the $head$ and $tail$ of the sequence are identical and contain the units R and U. By considering the edges of *both* unit R and unit U, the algorithm easily figures out that A is the unit most likely to immediate precede the sequence R,U while N is the unit most likely to immediately follow it. Note that looking at unit U alone, we would not be able tell whether N or B is the unit most likely to immediately follow the target sequence. To grow the target run, we can either add unit A to the front or unit N to the rear. Based on Step 6, we add unit N to the rear of the target sequence. Figure 4.17(c) shows the next iteration in the run discovery process where it becomes clear that $head$ refers to the first τ members of the target run while $tail$ refers to the last τ members. Note that the use of a context of τ references also allows the algorithm to distinguish between repeated patterns that share some common reorganization units.

The process of growing the target run continues until $headweight$ and $tailweight$ fall below some edge weight threshold. The edge weight threshold ensures that a sequence (*e.g.,* $u \rightarrow head$) becomes part of the run only if it occurs frequently. The threshold is therefore conveniently expressed relative to the value that $headweight$ or $tailweight$ would assume if *that* sequence were to occur once in every reorganization interval. In Figures 4.18 and D.16, we investigate the effect of varying the edge weight threshold on the effectiveness of run clustering. Observe that being more selective in picking the vertices tends to reduce performance except at very small threshold values for the PC workloads. As we have noted earlier, the PC workloads tend to have less repetition and a lot more churn in their reference patterns so that it is necessary to filter out some of the noise and look further into the past to find repeated patterns. The server workloads are much easier to handle and respond well to run clustering even without filtering. The jagged nature of the plot for the average of the server workloads (S-Avg.) results from DS1, which being only seven days long is too short for the edge weights to be smoothed out. In this chapter, we assume a default value of 0.1 for the edge weight threshold for all the workloads.

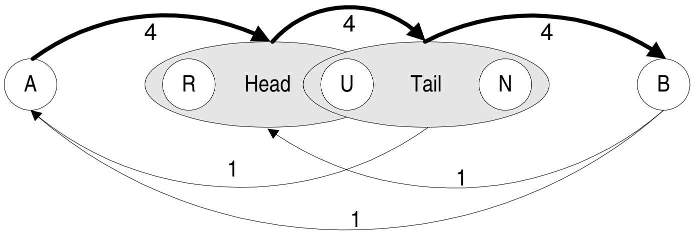
A variation of the run discovery algorithm is to terminate the target sequence whenever $headweight$ and $tailweight$ are much lower than (*e.g.,* less than half) their respective values in the previous iteration of the loop (steps 3-10). The idea is to prevent the algorithm from latching onto a run and pursuing it too far, or in other words, from going too deep down what could be a local minimum. Another variation of the algorithm is to add u to the target run only if the heaviest outgoing edge of u is to one of the vertices in $head$ and to



(a) Discovered Run = ϕ.



(b) Discovered Run = R,U.



(c) Discovered Run = R,U,N.

Figure 4.17: The Use of Context in Discovering the Next Vertex in a Run (Reference String = A,R,U,N,B,A,R,U,N,B, Context Size = 2).

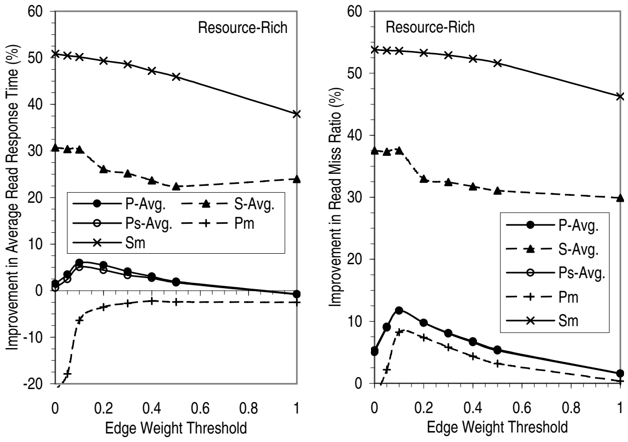


Figure 4.18: Sensitivity of Run Clustering to Edge Threshold (Resource-Rich).

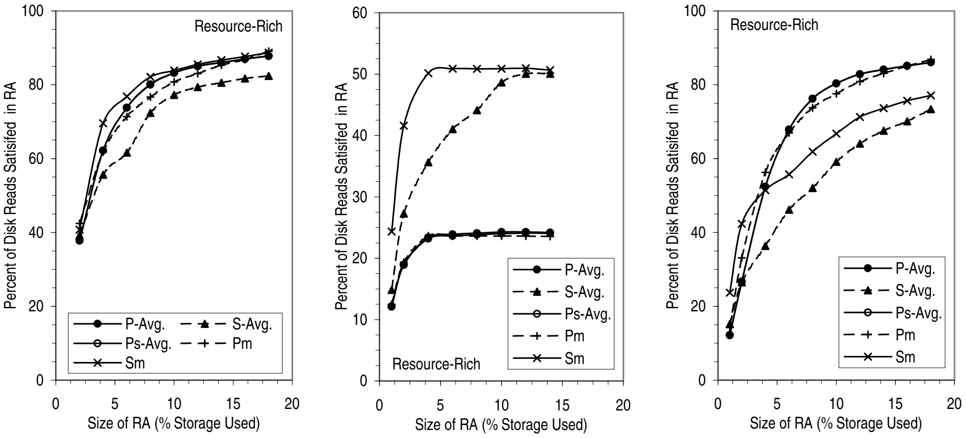
add v to the run only if the heaviest incoming edge of v is from one of the vertices in $tail$. We experimented with both variations and found that they do not offer consistently better performance. As we shall see later, the workloads do change over time so that excessive optimization based on the past may not be productive.

The whole algorithm is executed repeatedly to find runs of decreasing desirability. In Figure D.17, we study whether it makes sense to impose a minimum run length. Runs that are shorter than the minimum are discarded. Intuitively, the context size is chosen to allow pseudo repeated patterns to be effectively distinguished. Thus a useful run should be at least as long as the context size. This turns out to agree with our experimental results. Note that the run-discovery process is very efficient, requiring only $O(e \cdot log(e) + v)$ operations, where e is the number of edges and v, the number of vertices. The $e \cdot log(e)$ term results from sorting the edges once to facilitate step 1. Then each vertex is examined at most once.

After the runs have been discovered and laid out in the reorganized area, the traffic redirector decides whether a given request should be serviced from the runs. This decision can be made by conditioning on the context or the recent reference history. For example, suppose that a request matches the kth reorganization unit in run R. We define the context match as the percentage of the previous τ requests that are in $R[k-\tau]$... $R[k-1]$, and we redirect a request to R only if the context match exceeds some value. For our workloads, we find that it is generally better to always try to read from a run (Figure D.18).

4.6.3 Heat and Run Clustering Combined

A trend that stands out throughout our analysis of run clustering is that the improvement in read miss ratio significantly exceeds the improvement in read response and service time, especially for the PC workloads. It turns out that this is because many of the references cannot be clearly associated with any repeated sequence. For instance, we find that for the PC workloads, only about 20–30% of the disk read requests can be satisfied from the



(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure 4.19: Percent of Disk Reads Satisfied in Reorganized Area (Resource-Rich).

reorganized area with run clustering (Figures 4.19(b) and D.19(b)). Thus in practice, the disk head has to move back and forth between the runs in the reorganized area and the remaining hot spots in the home area. In other words, although the number of disk reads is reduced by run clustering, the time taken to service the remaining reads is lengthened. In the next section, we will study placing the reorganized region at different offsets into the volume. Ideally, we would like to locate the reorganized area near the remaining hot spots but these are typically distributed across the disk so that no single good location exists. Besides, figuring out where these hot spots are *a priori* is difficult. We believe a more promising approach is to try to satisfy more of the requests from the reorganized area. One way of accomplishing this is to simply perform heat clustering in addition to run clustering. Figure 4.20 illustrates the idea.

Since the runs are specific sequences of reference that are likely to be repeated, we assign higher priority to them. Specifically, on a read, we first attempt to satisfy the request by finding a matching run. If no such run exists, we try to read from the heat-clustered region before falling back to the home area. The reorganized area is shared between heat and run clustering, with the runs being allocated first. In this chapter, all the results for heat and run clustering combined assume a default reorganized area that is 15% of the storage size and that is located at a byte offset 30% into the volume. We will investigate the performance sensitivity to these parameters in the next section. We also evaluated the idea of limiting the size of the reorganized area that is devoted to the runs but found that it did not make a significant difference (Figure D.20). Sharing the reorganized area dynamically between heat and run clustering works well in practice because a workload with many runs

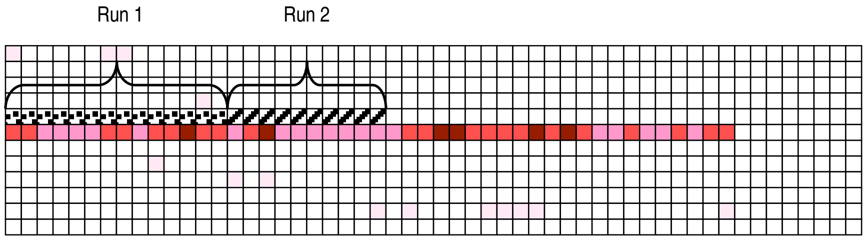


Figure 4.20: Block Layout with Heat and Run Clustering Combined.

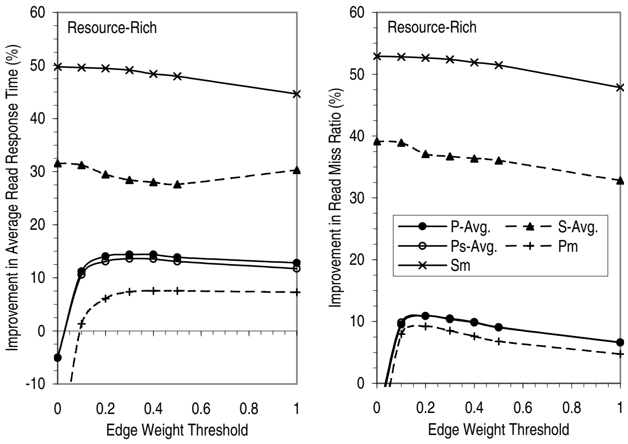


Figure 4.21: Sensitivity of Heat and Run Clustering Combined to Edge Threshold (Resource-Rich).

is not likely to gain much from the additional heat clustering while one with few runs will probably benefit a lot.

As shown in Figures 4.19(c) and D.19(c), when run clustering is augmented with heat clustering, the majority of the disk read requests can be satisfied from the reorganized area. This suggests that when heat clustering is performed in addition to run clustering, we can be more selective about what we consider to be part of a run because even if we are overly selective and miss some blocks, these blocks are likely to be found nearby in the adjacent heat-clustered region so that there is no need to go all the way to the original location of the blocks. We therefore reevaluate the edge weight threshold used in the run discovery algorithm. Figures 4.21 and D.21 summarize the results. Notice that compared to the case of run clustering alone (Figures 4.18 and D.16), the plots are much more stable and the performance is less sensitive to the edge weight threshold, which is a nice property. As expected, the performance is better with larger threshold values when heat clustering is performed in addition to run clustering. Thus, we increase the default edge weight threshold for the PC workloads to 0.4 when heat and run clustering are combined.

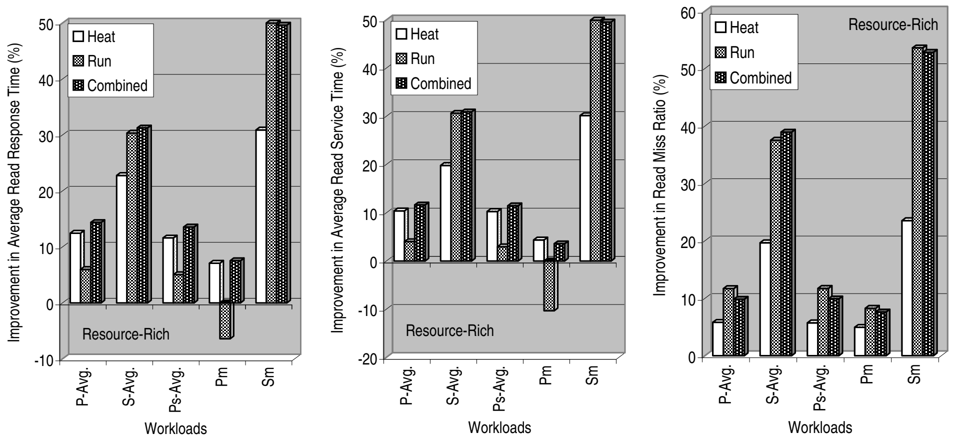


Figure 4.22: Performance Improvement with the Various Clustering Schemes (Resource-Rich).

4.7 Performance Analysis

4.7.1 Clustering Algorithms

In Figures 4.22 and D.22, we summarize the performance improvement achieved by the various clustering schemes. In general, the PC workloads are improved less by ALIS than the server workloads. This could be because the PC workloads, being more recent than the server workloads, have comparatively more caching upstream in the file system where a lot of the repeated references are satisfied. Therefore, the references remaining downstream at the physical level are less predictable. The disparity in the effectiveness of ALIS at speeding up the PC workloads and the server workloads could also result from the different file systems used and the increased availability of storage space in the more recent (PC) workloads. We would expect that when more storage space is available, fragmentation would be reduced so that there is less potential for ALIS to improve the block layout. However, most of the storage space in one of the server workloads, DS1, is updated in place. Such storage should have little fragmentation. Yet ALIS is able to improve the performance for this workload by a lot more than for the PC workloads. Note that fragmentation would not be completely eliminated when more storage space is available. Fragmentation occurs when files are deleted and the freed space is reused. Even though recent PCs have big disks so that the user seldom needs to explicitly remove files to make room, a lot of files (*e.g.,* log, backup) in this environment are automatically installed/created and deleted. Furthermore, in order to preserve locality and reduce fragmentation, recently freed space is reallocated even when there is a big chunk of empty space that has yet to be used.

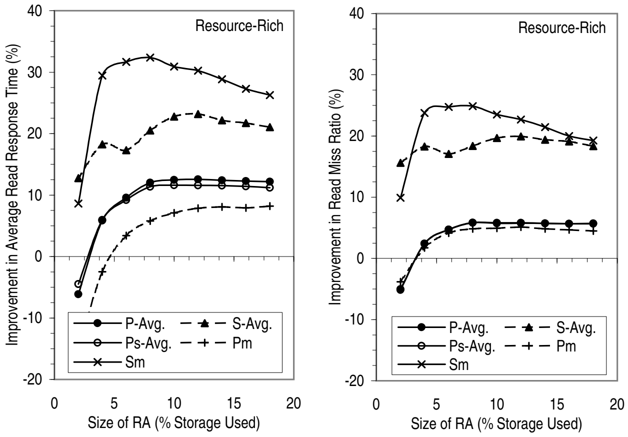
A deeper analysis of some of the PC workloads for which we have process and filename information shows that a user who starts an application will tend not to launch it again for several days. More generally, the PC is a general purpose device that manages the range of activities of a single user while the servers are more specialized, handling the same kinds of tasks for many different users. Thus the access pattern for the PC workloads tends to be more diverse and less predictable. In particular, any repetition in the PC workloads results from the same user performing the same task and this occurs much less frequently than in the case of the server workloads where the same task can be performed by many different users. Therefore for the PC workloads, there is less benefit in optimizing the disk block layout based on how the blocks were previously accessed. This intrinsic difference between the two kinds of workloads also explains why run clustering, which is predicated on the repetition of specific access patterns, is less effective for the PC workloads than heat clustering, which simply clusters frequently accessed data together, while the opposite is true for the server workloads.

Observe further that combining heat and run clustering enables us to achieve the greater benefit of the two schemes. In fact, the performance of the combined scheme is clearly superior to either technique alone in practically all the cases. Specifically, the read response time for the PC workloads is improved on average by about 17% in the resource-poor environment and 14% in the resource-rich environment. The sped-up PC workloads are improved by about the same amount while the merged PC workload is improved by just under 10%. In general, the merged PC workload is difficult to optimize for because the repeated patterns, already few and far in between, are spread further apart than in the case of the base PC workloads. For the base server workloads on average, the improvement in read response time ranges from about 30% in the resource-rich environment to 37% in the resource-poor environment. The merged server workload is improved by as much as 50%. Interestingly, one of the PC users, P10, the chief technical officer, was running the disk defragmenter, Diskeeper [Exe01]. Yet in both the resource-poor and resource-rich environments, run and heat clustering combined improves the read performance for this user by 15%, which is about the average for all the PC workloads.

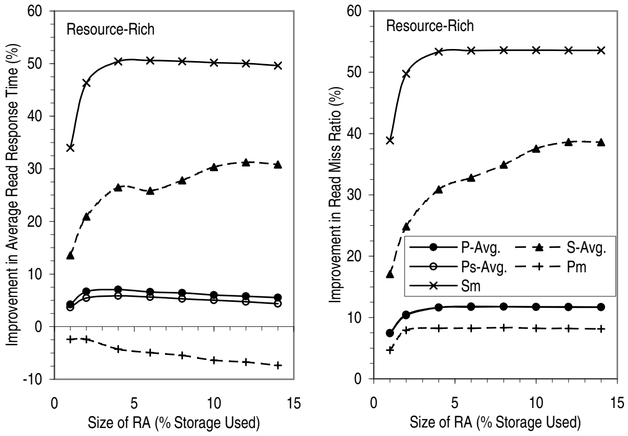
4.7.2 Reorganized Area

Earlier in the chapter, we said that reorganizing a small fraction of the stored data is enough for ALIS to achieve most of the potential performance improvement. In Figures 4.23 and D.23, we quantify what we mean by a small fraction. Observe that for all our workloads, a reorganized area less than 10% the size of the storage used is sufficient to realize practically all the benefit of heat clustering. For run clustering, the reorganized region required to get most of the advantage is even smaller. Combining heat and run clustering, we find that by reorganizing only about 10% of the storage space, we are able achieve most of the potential performance improvement for all the workloads except the server workloads which require on average about 15%. Given the technology trends, we believe that, in most cases, this 15% storage overhead in the form of the reorganized area is well worth the resulting increase in performance.

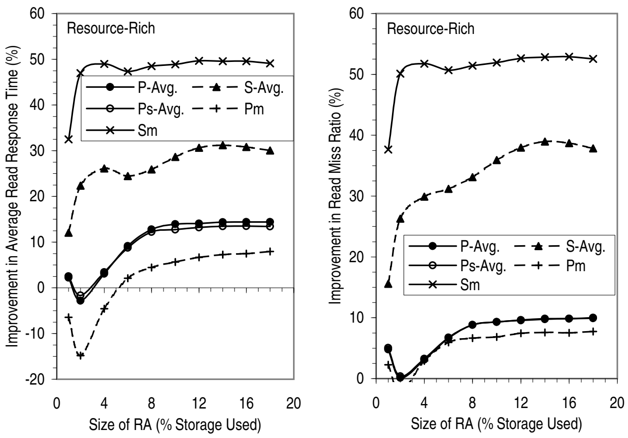
Note that performance does not increase monotonically with the size of the reorganized region, especially for small reorganized areas. In general, blocks are copied into the re-



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

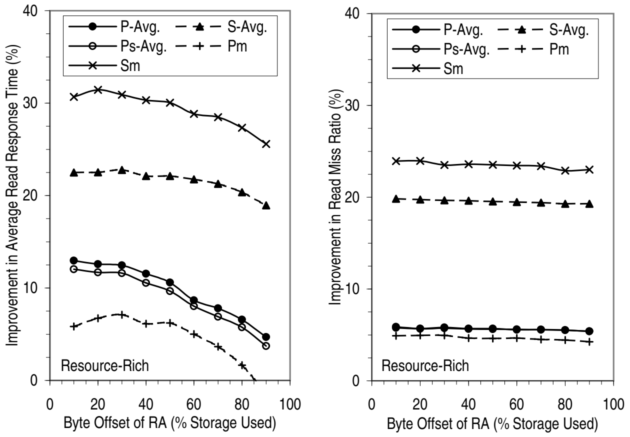
Figure 4.23: Sensitivity to Size of Reorganized Area (Resource-Rich).

organized region and rearranged based on the prediction that the new arrangement will outperform the original layout. For some blocks, the prediction turns out to be wrong so that as more blocks are reorganized, the performance sometimes decline. Heat clustering, in particular, is sensitive to the size of the reorganized area because the size determines the number of blocks that are rearranged. If the reorganized region is small, only a small number of the hottest blocks are reorganized. In this case, these blocks would be moved into the reorganized area and away from other blocks that are not as frequently accessed but that could be accessed in sequence. On the other hand, if the reorganized area is large, many blocks, including those that are not very frequently accessed, would be reorganized and this would dilute the heat that is being clustered. We could introduce a threshold to prevent the reorganization of blocks that are not very frequently accessed but for this study, that essentially only adds a parameter that serves the same function as the size of the reorganized area.

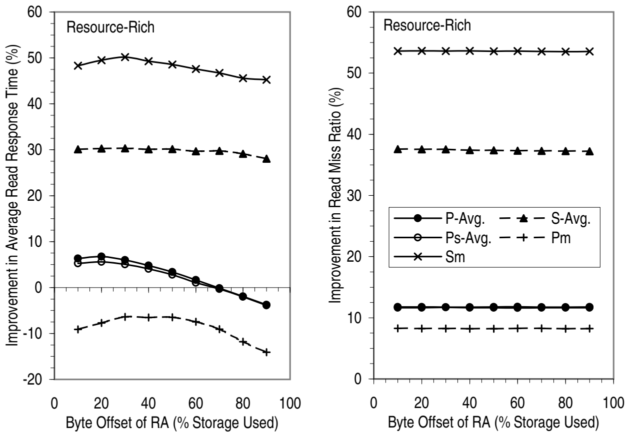
Besides the size of the reorganized region, another interesting question is where to locate it. If there is a constant number of sectors per cylinder and accesses are uniformly distributed, we would want to place the reorganized area at the center of the disk. However, modern disks are zoned so that the linear density, and hence the data rate, at the outer edge is a lot higher than at the inner edge. To leverage this higher sequential performance, the reorganized region should be placed closer to the outer edge. The complicating factor is that in practice, accesses are not uniformly distributed so that the optimal placement of the reorganized area depends on the workload characteristics. Specifically, it is advantageous to locate the reorganized area near to any remaining hot regions in the home area but determining these hot spots ahead of time is difficult. Besides, they are typically distributed across the disk so that no single good location exists.

In Figures 4.24 and D.24, we see these various effects at work in our workloads. We assume the typical situation where volumes are laid out inwards from the outer edge of the disk. Recall that heat and run clustering combined has the nice property that most of the disk reads are either eliminated due to the more effective sequential prefetch, or can be satisfied from the reorganized area. Any remaining disk reads will tend to exhibit less locality and be spread out across the disk. In other words, the remaining disk reads will likely be uniformly distributed. Therefore, in this case, we can predict that placing the reorganized area somewhere in the middle of the disk should minimize any disk head movement between the reorganized region and the home area. Empirically, we find that for all our workloads, locating the reorganized area at a byte offset 30-40% into the storage space works well. Given that there are more sectors per track at the outer edge, this corresponds to roughly a 24-33% radial distance offset from the outer edge.

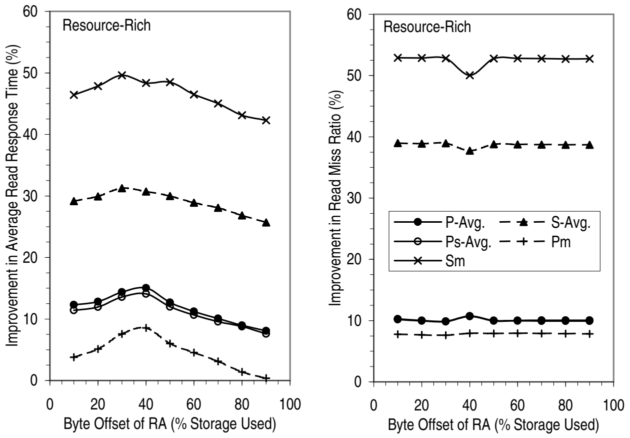
A shortcoming of replicating and reorganizing data is that a piece of data may have multiple names or addresses and that these names could change after each reorganization. The effectiveness of any cache downstream of the redirector could potentially be affected as a result. For example, after a block is copied into the reorganized region, any request for that block would likely be directed to the copy in the reorganized area. If the data is cached downstream of the redirector under its address in the home area, the cached data will be



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure 4.24: Sensitivity to Placement of Reorganized Area (Resource-Rich).

useless. The cached data will, however, eventually be replaced. Therefore, the effect of the changing names should not be significant. In Figures D.25 and D.26, we confirm this by remapping the cache contents after each reorganization. Specifically, if a cached block is copied into the reorganized region, we change its label in the cache to its address in the reorganized area. We find that such remapping does not significantly improve performance except for the merged PC workload in the resource-rich environment where the underlying cache is huge in relation to the working set of the workload. In practice, the effect of the changing names is likely to be even smaller because the process of moving blocks around could alter the contents of the underlying cache and this has not been modeled in our simulations. The results in this thesis assume that the cache contents are remapped after each reorganization. An alternative is to prime the cache, by issuing spurious I/O for example, with the pages most likely to be referenced next but this would introduce an orthogonal effect and obscure our results.

4.7.3 Write Policy

In general, writes or update operations complicate a system and throttle its performance. For a system such as ALIS that replicates data, we have to ensure that all the copies of a block are either updated or invalidated whenever the block is written to. In our analysis of the workloads in Chapter 2, we discover that blocks that are updated tend to be updated again rather than read. This suggests that we should update only one of the copies and invalidate the others. But the question remains of which copy to update. A second issue related to how writes are handled is whether the read access patterns differ from the write access patterns, and if it is possible to lay the blocks out to optimize for both. We know that the set of blocks that are both actively read and written tend to be small [Chapter 2] so it is likely that read performance can be optimized without significantly degrading write performance. But should we try to optimize for both reads and writes by considering writes in addition to reads when tabulating the reference count and when building the access graph?

In Figures 4.25 and D.27, we show the performance effect of the different policies for handling writes. Observe that for heat clustering, updating the copy in the reorganized area offers the best read and write performance. Incorporating writes in the reference count speeds up the writes and in the case of the PC workloads, also improves the read performance. The results in this chapter therefore assume that writes are counted in heat clustering. Figures 4.26 and D.28 present the corresponding results for run clustering. Notice that including write requests in the access graph improves the write performance for some of the workloads but decreases read performance across the board. Therefore, the default policy in this chapter is to consider only reads for run clustering. As for which copy to update, the simulation results suggest updating the runs for the server workloads. For the PC workloads, updating the runs increases read performance slightly but markedly degrades write performance. Therefore, the default policy for the PC workloads is to update the home copy. That the read performance for PC workloads increases only slightly when runs

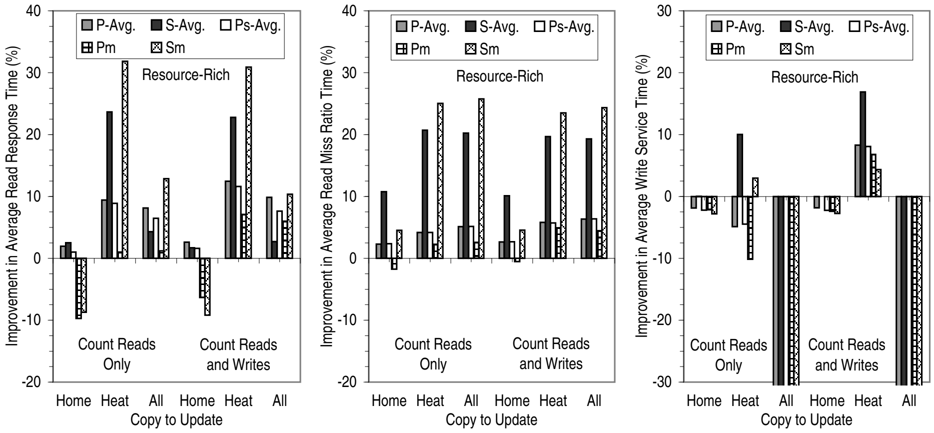


Figure 4.25: Effect of Various Write Policies on Heat Clustering (Resource-Rich).

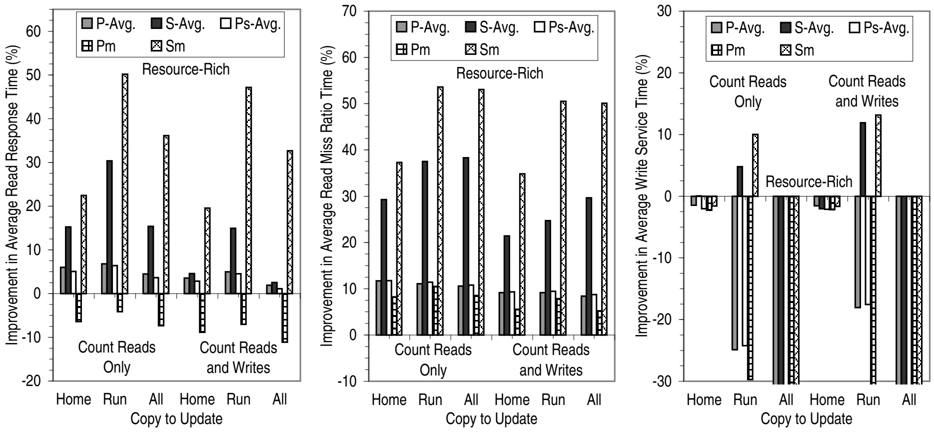


Figure 4.26: Effect of Various Write Policies on Run Clustering (Resource-Rich).

are updated is not surprising since the runs in this environment are often repeated reads of application binaries, and these are written only when the applications were first installed.

Next, we investigate write policies for the combination of heat and run clustering in Figures 4.27 and D.29. We introduce a policy called *RunHeat* that performs a write by first attempting to update the affected blocks in the run-clustered region of the reorganized area. If a block does not exist in the run-clustered region, the policy tries to update that block in the heat-clustered region. If the block does not exist anywhere in the reorganized area, the original copy in the home area is updated. As shown in the figures, RunHeat is the best write policy for all the workloads as far as read performance is concerned. Furthermore, it does not degrade write performance for any of the workloads, and in fact achieves a sizeable improvement of about 5-10% in the average write service time for most of the workloads and up to 22% for the base server workloads in the resource-poor environment. This is the default write policy we use for heat and run clustering combined.

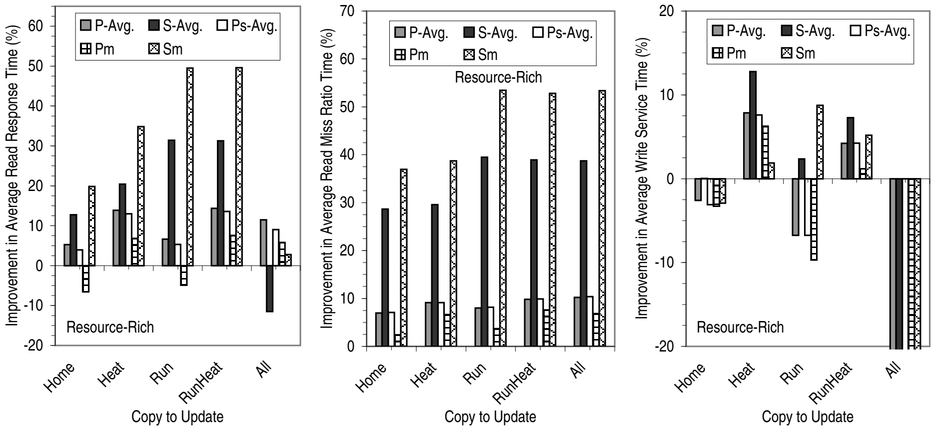


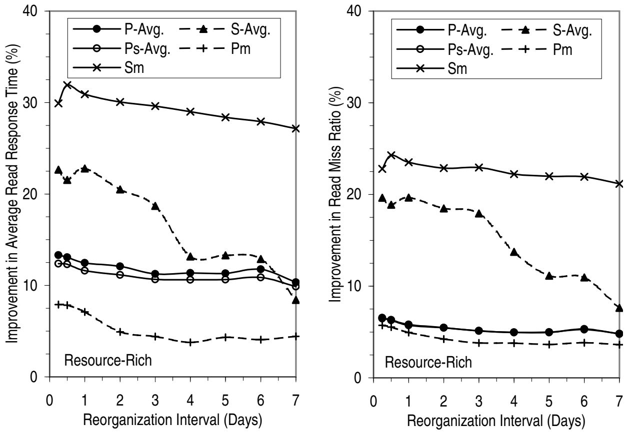
Figure 4.27: Effect of Various Write Policies on Heat and Run Clustering Combined (Resource-Rich).

4.7.4 Workload Stability

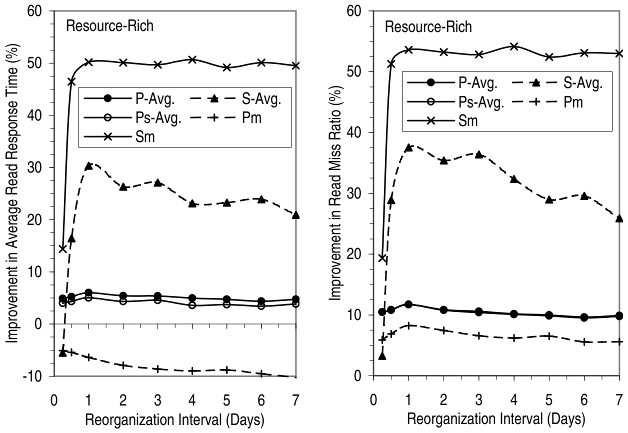
The process of reorganizing disk blocks entails a lot of data movement and may potentially affect the service rate of any incoming I/Os. In addition, resources are needed to perform the workload analysis and the optimization that produces the reorganization plan. Therefore it is important to understand how frequently the reorganization needs to be performed and the tradeoffs involved. Figures 4.28 and D.30 present the sensitivity of the various clustering strategies to the reorganization interval.

We would expect daily reorganization to perform well because of the diurnal cycle. Our results confirm this. They also show that less frequent reorganization tends to only affect the improvement gradually so that reorganizing daily to weekly is generally adequate. This is consistent with findings in [RW91]. The only exception is for the average of the server workloads where the effectiveness of ALIS plummets if we reorganize less than once every three days. Recall that one of the components of this average is DS1, which is only seven days long. Because we use the first three days of this trace for warm up purposes, if we reorganize less frequently than once every three days, the effect of the reorganization will not be fully reflected. Note that for run clustering and combined heat and run clustering, reorganizing more than once a day performs poorly. This is because the workloads do vary over the course of a day so that if we reorganize too frequently, we are always “chasing the tail” and trying to catch up. Unless otherwise stated, all the results in this chapter are for daily reorganization.

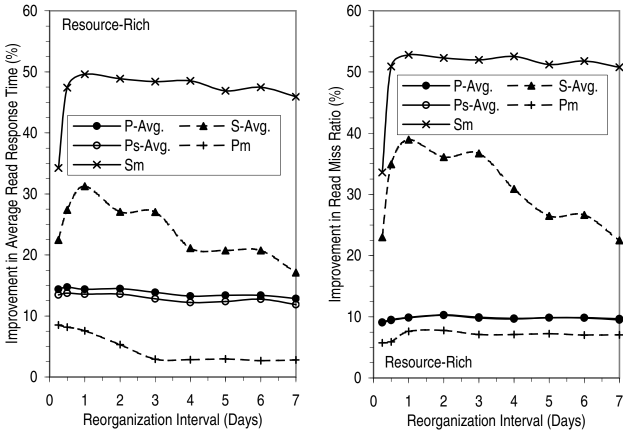
More generally, the various clustering strategies are all based on the reference history. They try to predict future reference patterns by assuming that these patterns are likely to be those that have occurred in the past. The effectiveness of these algorithms is therefore limited by the extent to which workloads vary over time. The above results suggest that there are portions of the workloads that are very stable and are repeated daily since there is but a small effect in varying the reorganization frequency from daily to weekly. To gain



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure 4.28: Sensitivity to Reorganization Interval (Resource-Rich).

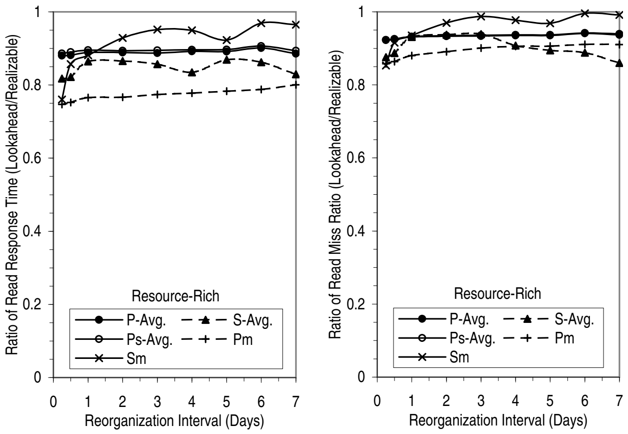
further insight into the stability of the workloads, we consider the ideal case where we can look ahead and see the future references of a workload. In Figures 4.29 and D.31, we show how much better these algorithms perform when they have knowledge of future reference patterns as compared to when they have to predict the future reference patterns from the reference history.

In the figures, the "realizable" performance is that obtained when the reorganization is performed based on the past reference stream or the reference stream seen so far. This is what we have assumed all along. The "lookahead" performance is that achieved when the various clustering algorithms are operated on the future reference stream, specifically the reference stream in the next reorganization interval. Observe that for heat clustering, the difference is small, meaning that the workload characteristic heat clustering exploits is relatively stable. On the other hand, for run clustering and combined heat and run clustering, the reference history is not that good a predictor of the future. With a reorganization frequency of once a day, having forward knowledge outperforms history-based prediction by about 40-50%. In other words, significant portions of the workloads are time-varying or cannot be predicted from past references. This suggests that excessive mining of the reference history for repeated sequences or runs may not be productive.

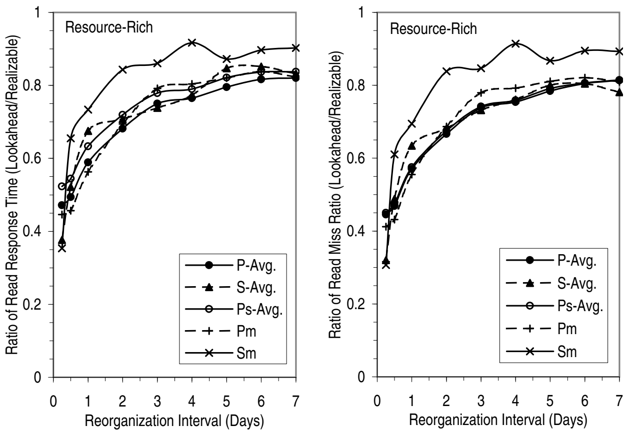
Based on our prior result that there is a lot of time during which the storage system is relatively idle [Chapter 2], we made the assumption earlier in the chapter that the block layout can be changed instantaneously. Here we validate the assumption by showing that the process of physically copying blocks into the reorganized region takes up only a small fraction of the idle time available between reorganizations, given that the reorganizations are performed at most daily. In this set of experiments, we populate the reorganized area from beginning to end by performing the reads in batches so that request scheduling by the disk is likely to be effective. Since the disk in our simulations has a default maximum queue depth of eight, we use a batch size of eight. The data read in each batch is written to the reorganized area in large sequential writes before the next batch of reads are issued.

As shown in Figures 4.30 and D.32, such a simple process is able to efficiently copy selected disk blocks into the reorganized area, achieving a rate of about 10 MB/s. The copy rate is especially high for heat clustering because the sequential layout means that the reads occur in elevator order. Notice that the copy rate is higher for the PC workloads than the server workloads, suggesting that the reads for copying the blocks are more sequential for these workloads, or in other words, that the original block layout for the PC workloads is closer to the optimized layout. This is consistent with our finding that the PC workloads is less responsive to ALIS than the server workloads. Observe further that the copying process is slightly faster in the resource-poor environment than in the resource-rich environment. This is because the extra caching and buffering in the resource-rich environment is not effective when there is no block reuse, as is the situation when copying blocks into the reorganized region.

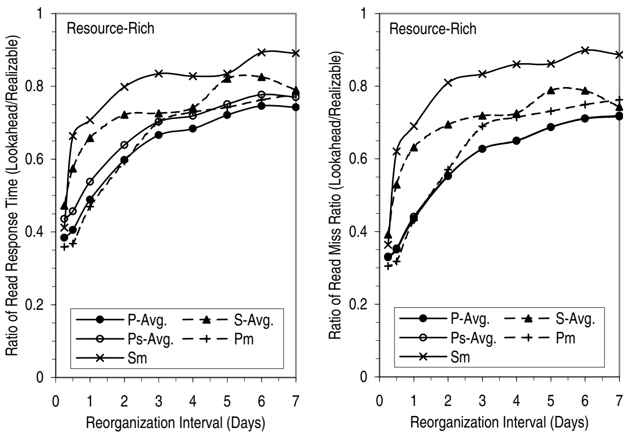
In the worst case, it takes just over 10 minutes to copy the selected disk blocks into the reorganized region. Since the results presented in Chapter 2 show that the storage system is relatively idle more than 90% of the time, 10 minutes represent but a tiny fraction



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure 4.29: Performance with Knowledge of Future Reference Patterns (Resource-Rich).

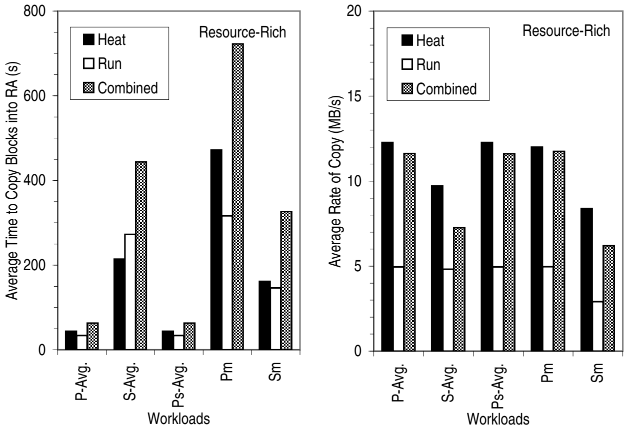


Figure 4.30: Rate of Copying Blocks into the Reorganized Region (Resource-Rich).

of the idle time available in a day. Therefore, it is likely that ALIS will be able to find time to physically reorganize the blocks without affecting the performance of incoming I/Os. Furthermore, as we have described earlier, the block reorganization process may be performed incrementally, whenever the load on the storage system is low. In addition, if the physical block copy time do become an issue, we can try to adjust the current contents of the reorganized region to match a subsequent reorganization plan rather than to start from scratch and copy all the selected blocks into the reorganized area.

4.7.5 Effect of Improvement in the Underlying Disk Technology

Disk technology is constantly evolving. Mechanically, the disk arm is becoming faster over time and the disk is spinning at a higher rate. The recording density is also increasing with each new generation of the disk so that there are more sectors per track and more tracks per inch. The net effect of these trends is that less physical movement is needed to access the same data, and the same physical movement takes a shorter amount of time. We have demonstrated that ALIS is able to achieve dramatic improvement in performance for a variety of real workloads. An interesting issue is whether the effect of ALIS, which tries to reduce both the number of physical movements and the distance moved, will be increased or diminished as a result of these technology trends. It has been pointed out previously that as disks become faster, the benefit of reducing the seek distance will be lessened [RW91]. In ALIS, we emphasize clustering strategies that not only reduce the seek distance but, more importantly, also eliminate some of the I/Os by increasing the effectiveness of sequential prefetch. This latter effect should not diminish with technology trends. In this section, we will empirically verify that the benefit of ALIS persists as disk technology evolves.

Mechanical Improvement

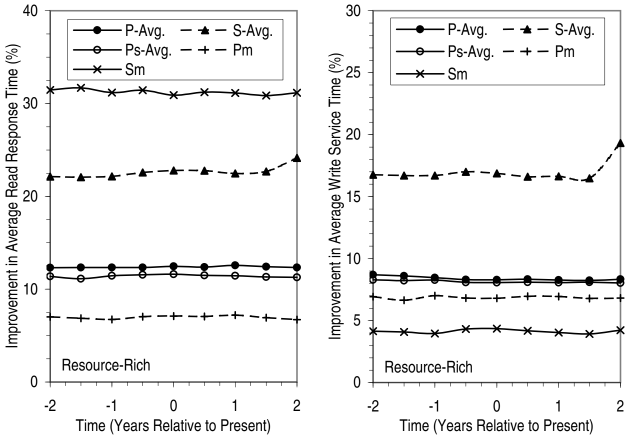
We begin by examining the effect of improvement in the mechanical or moving parts of the disk, specifically, the reduction in seek time and the increase in rotational speed. The average seek time is generally taken to be the average time needed to seek between two random blocks on the disk. Based on the performance characteristics of server disks introduced by IBM over the last ten years, we found that on average, seek time decreases by about 8% per year while rotational speed increases by about 9% per year [Chapter 3]. Together, the improvement in the seek time and the rotational speed translate into an 8% yearly improvement in the average response and service times for our various workloads [Chapter 3]. This scenario of improving only the mechanical portions of the disk provides insights into situations where the increased capacity of newer disks are utilized so that the disk occupancy rate is kept constant.

In Figures 4.31, D.33 and D.34, we investigate how the effect of ALIS changes as the disk is improved mechanically at the historical rates. Observe from the figures that the benefit of ALIS is practically constant over the four-year period. In fact, the plots show a slight upward trend, especially for the server workloads in the resource-poor environment. This slight increase in the effectiveness of ALIS stems from the fact that as the disk becomes faster, it will have more free resources (time) to perform more opportunistic prefetch and with ALIS, such opportunistic prefetch is more likely to be useful. There is an abrupt rise in some of the S-Avg. plots at the end of the four-year period because DS1, one of the components of S-Avg., is sensitive to how the blocks are laid out in tracks since some of its accesses, especially the writes, occur in specific patterns. As the disk is sped up, the layout of blocks in the tracks have to be adjusted to ensure that transfers spanning multiple tracks do not "miss revolutions". In some cases, consecutively accessed blocks become poorly positioned rotationally [Chapter 3] so that the benefit of ALIS is especially pronounced. Such situations highlight the value of ALIS in ensuring good performance and reducing the likelihood of unpleasant surprises due to poor block layout.

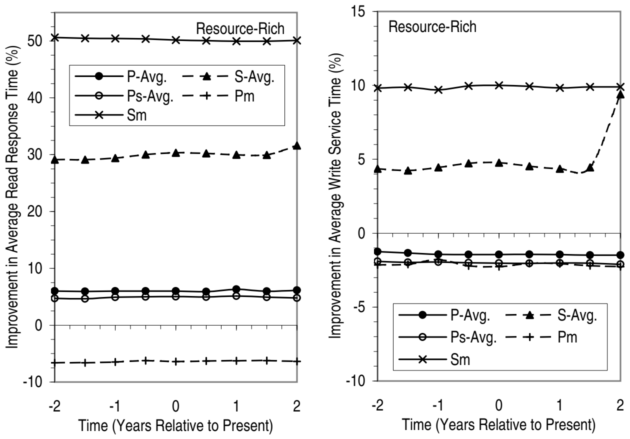
That the effectiveness of ALIS does not decrease with faster disks may be somewhat surprising. At the very least, we would expect any performance improvement resulting from seek distance reduction to be diminished because the seek profile should become flatter as disks become faster so that reducing the seek distance would become less effective. It turns out that as shown in Figure 4.32, while a flatter seek profile is usually associated with faster disks (*e.g.,* [RW91]), the main cause of a flatter seek profile is the increase in track density, which we will examine next.

Increase in Recording Density

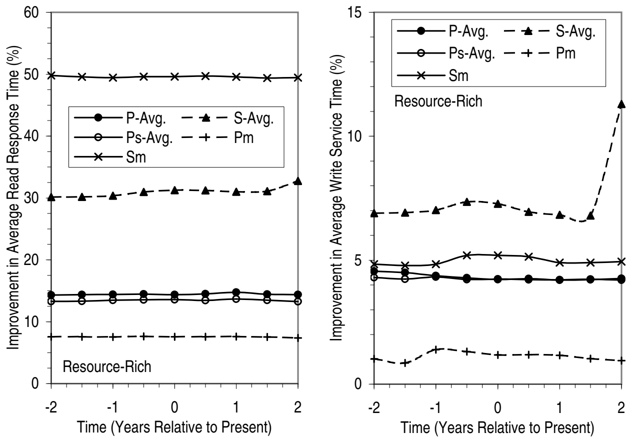
Increasing the recording or areal density reduces the cost of disk-based storage. Areal density improvement also directly affects performance because as bits are packed more closely together, they can be accessed with a smaller physical movement. Historically, linear density increases by about 21% per year while track density rises by approximately



(a) Heat Clustering.

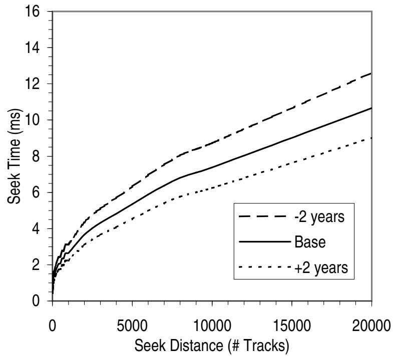


(b) Run Clustering.

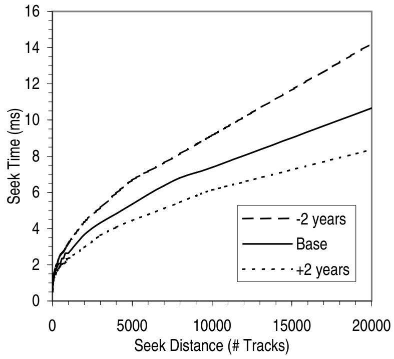


(c) Heat and Run Clustering Combined.

Figure 4.31: Effectiveness of the Various Clustering Techniques as Disks are Mechanically Improved over Time (Resource-Rich).



(a) Due to Mechanical Improvement.



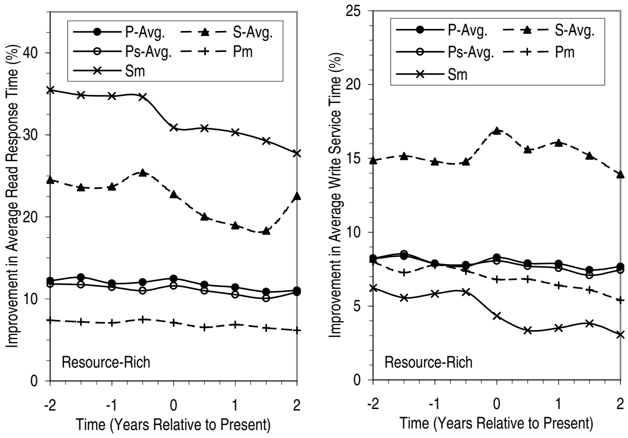
(b) Due to Increase in Recording Density.

Figure 4.32: Change in Seek Profile Over Time.

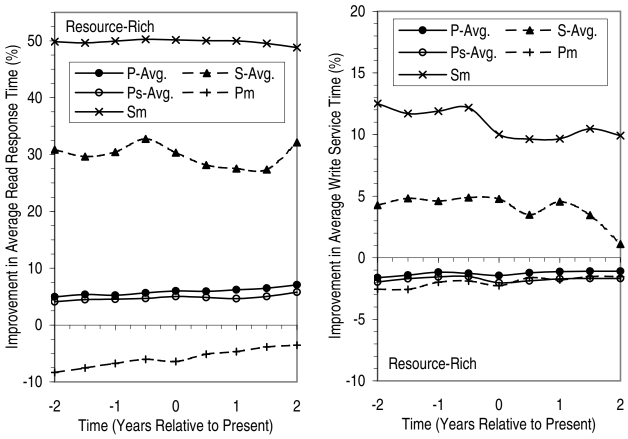
24% per year [Chapter 3]. For our workloads, the average response and service times are improved by about 9% per year as a result of the increase in areal density [Chapter 3].

In Figures 4.33, D.35 and D.36, we analyze how areal density improvement affects the effectiveness of ALIS. Observe that there is a sharp change in some of the plots when we go backwards in time. This is because as areal density is reduced, we require more disks to hold the same amount of data and with more disks, there are more resources to effectively perform prefetch and to better take advantage of the improved locality that ALIS offers. This is especially the case in the resource-poor environment where the cache size increases with the number of disks. In the resource-rich environment, there is also a noticeable increase in the effectiveness of heat clustering at reducing the read miss ratio when there are more disks. A deeper analysis reveals that with heat clustering, most of the data that are opportunistically prefetched turn out to be useful so that having more disks and therefore more prefetch buffers helps to increase the benefit of ALIS.

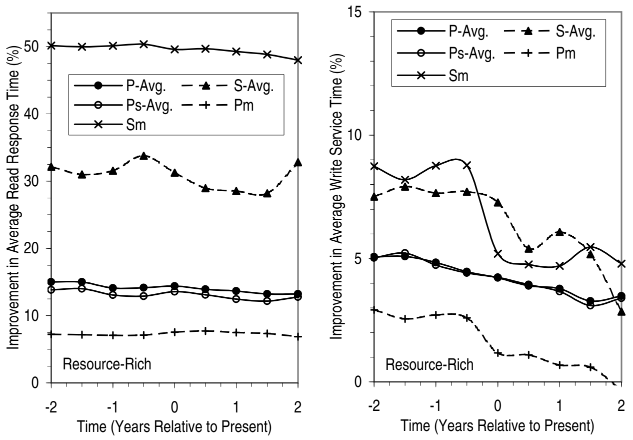
Observe further that there is a downward trend in most of the plots, especially those that relate to heat clustering. This is expected because heat clustering derives part of its benefit from seek distance reduction which is less effective as disks become denser and the difference between a long and short seek is reduced. The improvement in write performance, being also dependent on a reduction in seek distance, shows a similar downward trend. On the other hand, the performance benefit of run clustering is relatively insensitive to the increase in areal density since the main effect of run clustering is to reduce the number of I/Os by increasing the effectiveness of sequential prefetch. The same is true of heat and run clustering combined. Such a result is quite remarkable because at the historical rate of increase in areal density, two years of improvement translates into more than a doubling of disk capacity. This means that in going forward two years in time in Figures 4.33, D.35



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure 4.33: Effectiveness of the Various Clustering Techniques as Disk Recording Density is Increased over Time (Resource-Rich).

and D.36, we are seriously short-stroking the disk by using less than half the available disk space. Yet ALIS is still able to achieve a dramatic improvement in performance.

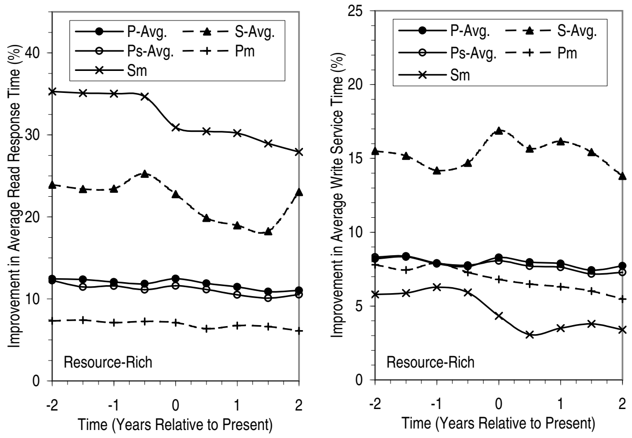
Overall Effect of Disk Technology Trends

Putting together the effect of mechanical improvement and areal density scaling, we obtain the overall performance effect of disk technology evolution. For our various workloads, the average response and service time are projected to improve by about 15% per year [Chapter 3]. In Figures 4.34, D.37 and D.38, we plot the additional performance improvement that ALIS can provide. Note that except for the aforementioned transition effects as the number of disks required changes, the benefit of ALIS with heat and run clustering combined is generally stable over time with only a very slight downward inclination. In the worst case, going forward two years in time reduces the improvement with ALIS from 50% to 48% for the merged server workload in the resource-rich environment. As discussed earlier, this is quite impressive because at the end of the two-year period, we are using less than half the available disk space. In the more realistic situation where we try to take advantage of the increased disk space, the benefit of ALIS will be even more enduring. Note that we did not re-optimize any of the parameters of the underlying storage system for ALIS. We simply use the settings that have been found to work well for the base system [Chapter 3]. If we were to tune the underlying storage system to exploit the increasing gap between random and sequential performance, and the improved locality that ALIS provides, the benefit of ALIS should be all the more stable and substantial.

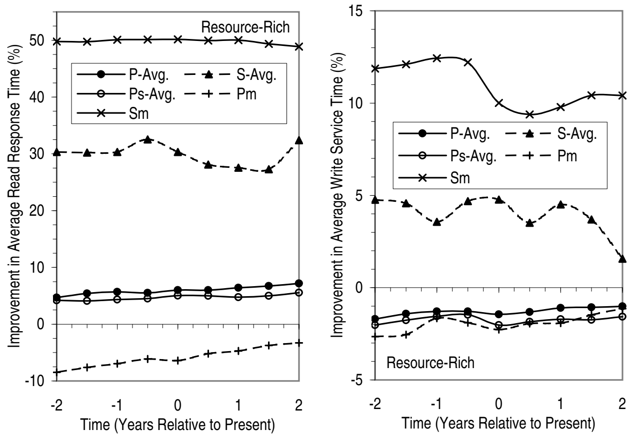
4.8 Conclusions

In this chapter, we propose ALIS, an introspective storage system that continually analyzes I/O reference patterns to replicate and reorganize selected disk blocks so as to improve the spatial locality of reference, and hence leverage the dramatic improvement in disk transfer rate. Our analysis of ALIS suggests that disk block layout can be effectively optimized by an autonomic storage system, without human intervention. Specifically, we find that the idea of clustering together hot or frequently accessed data has the potential to significantly improve storage performance, if handled in such a way that existing spatial locality is not disrupted. In addition, we show that by discovering repeated read sequences or runs and laying them out sequentially, we can greatly increase the effectiveness of sequential prefetch. By further combining these two ideas, we are able to reap the greater benefit of the two schemes and achieve performance that is superior to either technique alone. In fact, with the combined scheme, most of the disk reads are either eliminated due to the more effective prefetch or can be satisfied from the reorganized data, an outcome which greatly simplifies the practical issue of deciding where to locate the reorganized data.

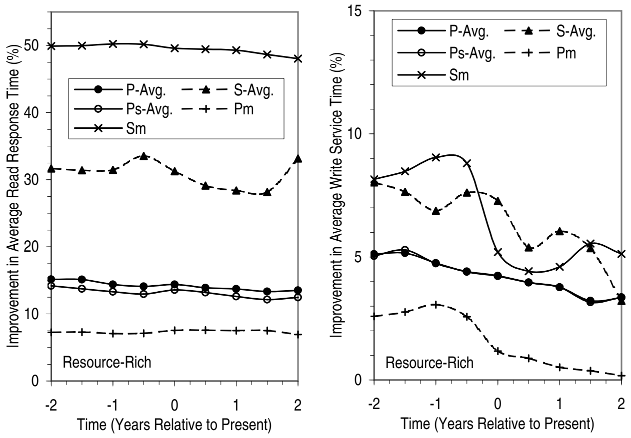
Using trace-driven simulation with a large collection of real server and PC workloads, we demonstrate that ALIS is able to far outperform prior techniques in both an environ-



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure 4.34: Effectiveness of the Various Clustering Techniques as Disk Technology Evolves over Time (Resource-Rich).

ment where the storage system consists of only disks and low-end disk adaptors, and one where there is a large outboard controller. For the server workloads, read performance is improved by between 31% and 50% while write performance is improved by as much as 22%. The read performance for the PC workloads is improved by about 15% while the writes are faster by up to 8%. Given that disk performance, as perceived by real workloads, is increasing by about 8% per year assuming that the disk occupancy rate is kept constant [Chapter 3], such improvements may be equivalent to as much as several years of technological progress at the historical rates. As part of our analysis, we also examine how improvement in disk technology will impact the effectiveness of ALIS and confirm that the benefit of ALIS is relatively insensitive to disk technology trends.

Chapter 5

The Performance Effect of Offloading Application Processing to the Storage System

5.1 Synopsis

Recent developments in both hardware and software have made it increasingly feasible to offload general purpose processing from the host system to the storage system. In particular, low-cost processing power is increasingly available, and software can be made robust, secure and mobile. In this chapter, we propose a general Smart Storage (SmartSTOR) architecture in which a processing unit that is coupled to one or more disks can be used to perform such offloaded processing. A major part of the chapter is devoted to understanding the performance potential of the SmartSTOR architecture for decision support workloads. Our analysis suggests that there is a definite performance advantage in using fewer but more powerful processors. As for software architecture, we find that the offloading of database operations that involve only a single relation is not very promising. In order to achieve significant speedup, we have to consider the offloading of multiple-relation operations. In general, for the storage system to effectively handle application processing such as running decision support queries, we need parallel software systems that are scalable and that can efficiently utilize the large number of processing units that will likely be in such a storage system.

5.2 Introduction

Typical I/O devices consist of the physical device hardware (*e.g.,* disk platters, read/write heads), device specific electronics (*e.g.,* sense amplifiers) and generic electronics to control the device and handle the interface to the host. With the rapid growth in processing power per processor (estimated at a rate of 60% per year [HP96]), it is reasonable to consider implementing and treating some of the generic electronics in the I/O device as a general purpose processor, and not just as a dedicated microprogrammed embedded controller. For instance, a 33 MHz ARM7TDMI embedded processor has recently been used to implement all the functions of a disk controller, including the servo control [AO97]. In addition, many of today's storage adapters as well as outboard and Network Attached Storage (NAS) controllers already contain several general purpose commodity processors to handle functions such as RAID [CLG+94] protection and network protocol processing. *If a moderately powerful general purpose microprocessor is combined with a reasonable amount of local memory, and placed either in a disk controller or a storage controller (i.e., a controller which controls multiple devices), then there will exist a general purpose outboard CPU with substantial excess processing capacity.*

Recent advances in software technology make using this processing capacity easier than previously. In particular, software fault isolation techniques [WLAG93] as well as robust and secure languages such as Java [GM96] enable applications to be effectively isolated so that they can be safely executed on a machine without causing malicious side effects. Recent emphasis on architectural neutrality and the portability of languages [GM96] further enhances code mobility and eases the way for code to be moved to different machines for execution. For example, in SUN's Jini framework [Sun99], application code can be downloaded to the device as needed. The convergence of these hardware and software developments provide an opportunity for a fundamental shift in system design by potentially allowing application code to be offloaded to the peripherals for execution.

In this chapter, we propose a general *Smart Storage* (SmartSTOR) architecture in which a processing unit that is coupled to one or more disks can be used to perform general purpose processing offloaded from the host. Essentially, we envision a system in which the host supervises a number of SmartSTORs, each of which consists of a powerful processing unit, a useful amount of local memory, and a number of I/O devices, usually disks. The host processor may generate tasks specific to one SmartSTOR (*i.e.,* only needing data local to that SmartSTOR) and delegate that work to the SmartSTOR, which would then deliver the result to the host. Alternatively, the SmartSTOR can be handed more complicated tasks that require coordination with other SmartSTORs. If the generation and delegation of these tasks can be sufficiently automated and reliable, and if the load balancing is successful, then the processing power of the SmartSTORs and the host becomes additive, and the result is a much more powerful system.

Besides allowing processing to be offloaded from the host processor, the Smart Storage architecture also reduces data movement between the host and storage subsystem. In addition, it allows processing power to be automatically scaled with increasing storage demand. An important feature of a SmartSTOR is that it may be configured as NAS so that the processing power in the SmartSTOR would be available to any system mounting that storage. The processing power embedded in the storage system could also be used to simplify the costly task of system management [BOK+98].

An essential element to the success of the Smart Storage architecture lies in convincing software developers that SmartSTOR is a viable and attractive architecture. Projecting the performance potential of the SmartSTOR architecture relative to the required software effort is an important first step in this direction. Since decision support workloads are increasingly important commercially [Ber98], a major part of this chapter is devoted to understanding how these workloads will perform on the SmartSTOR architecture. In particular, we evaluate the performance of the Transaction Processing Performance Council Benchmark D (TPC-D) [Tra97], which is the industry-standard decision support benchmark, on various SmartSTOR-based systems.

Our methodology is based on projecting SmartSTOR performance from current system performance and parameters. More specifically, we use the system configurations of published TPC-D results to determine the number of SmartSTORs that will be needed. In addition, we examine the query execution plans from two certified TPC-D systems to establish the fraction of work that can be offloaded to the SmartSTORs. We also use the TPC-D results to empirically derive the system scalability relationship so that we can estimate the effectiveness of distributing a query among many SmartSTORs. There are clearly limits to this projection approach but we believe that it is the most effective and appropriate methodology at this early stage.

The rest of this chapter is organized as follows. In the next section, we discuss related work and highlight some of the unique features of the SmartSTOR architecture that make it more viable than other previous proposals. In Section 5.4, we describe the hardware and software architecture for SmartSTOR. This is followed in Section 5.5 by a discussion of the methodology used to project the performance of TPC-D on systems with SmartSTOR. Performance analysis results are presented in Section 5.6. Section 5.7 concludes this chapter.

5.3 Related Work

There have been some recent proposals for embedding intelligence in disks [Gra98] and these include the Intelligent Disk (IDISK) [KPH98b] and the Active Disk [AUS98a, RGF98]. The processors that can be used in these disk-centric proposals are subject to the power budget and stringent cost constraints of the disk environment – generally disks are fungible and are sold almost entirely on the basis of price. The market for high cost/high performance/high functionality disks is very limited, and thus prices for disks in this market segment are higher than they would otherwise be due to the loss of efficiencies of scale.

On the other hand, SmartSTOR, by operating at the level of the storage (*i.e.,* multiple device) controller, can offer processing units that are more substantial and therefore easier to effectively use. Moreover, by allowing a processing unit to be coupled to one or more disks, the SmartSTOR architecture allows for more flexible scaling of processing power to increasing storage demand. In the nearer term, the SmartSTOR architecture is likely to be easier to accomplish because increasing the processing power on an adapter or controller to handle general purpose processing is less risky than modifying the actual disk design. It also lowers the barrier of entry and opens up the architecture to the creativity of more than

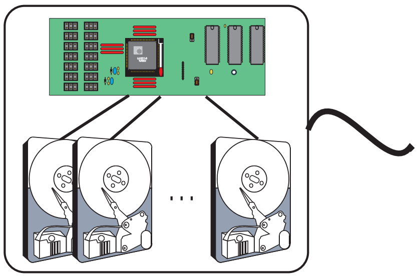


Figure 5.1: SmartSTOR Hardware Architecture.

just the few disk companies. Finally, it separates the manufacturing of low cost disks, most of which go into personal computers (PCs), from high performance controllers which can go into servers, clusters and mainframes, and which are relatively price-insensitive.

The idea of moving processing closer to the disk was studied extensively in the form of database machines during the late 1970s and early 1980s [DH81, HMP89]. Most of those database machines relied on costly special-purpose hardware which had to be specifically programmed and which prevented the database machines from taking advantage of algorithmic advancements and improvements in commodity hardware. In most cases, the functionality of the database machines was limited; they could not do arbitrary database operations. Furthermore, the reliance on highly-specialized hardware made it difficult to develop succeeding generations of the system so that it was not worthwhile to expend significant effort programming these machines.

In contrast, the SmartSTOR architecture leverages commodity general purpose hardware which allows the system to track the continual improvements in both hardware and software. In particular, a SmartSTOR can be based on a standard CPU (*e.g.,* PowerPC, MIPS, X86, ARM *etc.*), for which there are extensive software tools, a great deal of support, and a long projected life. In addition, the technology that is now available for developing portable and architecturally-neutral software can help reduce the need to program specifically for any particular implementation of the SmartSTOR architecture. Furthermore, shared nothing database algorithms and technology have matured to the point where we should be able to exploit some of the parallelism present in the SmartSTOR architecture.

5.4 The SmartSTOR Architecture

The proposed Smart Storage architecture consists of a processing unit that is coupled to one or more disks. Figure 5.1 depicts such an architecture. We define the *cardinality* of a SmartSTOR to be the number of disks it contains. A SmartSTOR with a cardinality of one contains a single disk and is conceptually equivalent to an IDISK/Active Disk in our performance analysis.

A SmartSTOR-based system is similar in many aspects to a cluster of general purpose computing nodes made up of commodity parts but there are several important differences.

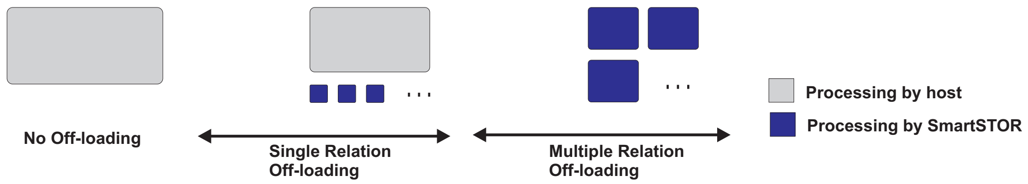


Figure 5.2: Possible Software Architectures.

In general, SmartSTORs can be built by increasing the processing power of existing storage adapters or controllers, many of which are based on commodity components. Such an approach allows us to save on the supporting infrastructure (*e.g.,* chipset, power distribution, physical packaging). Another approach to building SmartSTORs is to perform a limited amount of custom packaging and "decontenting", *i.e.,* removing parts that are not needed or will not be noticed, on a standard PC design. For example, we could easily put together a package consisting of an X86 processor, a power supply, a network card and several disks to serve as the hardware for a SmartSTOR. This allows us to leverage the most cost effective parts and to achieve more efficient packaging and reduced component count. Besides saving on the upfront equipment cost, SmartSTOR also has the potential to reduce operating costs through more efficient packaging which takes up less floor space. In addition, SmartSTORs can be made easier to manage than general purpose PCs, especially since they are designed to handle specific tasks that are offloaded from the host through a well-defined interface.

Ultimately, the success of the SmartSTOR architecture hinges on the availability of software that can take advantage of its unique capabilities. Figure 5.2 shows a spectrum of software options, each having different performance potential and requiring different amounts of software engineering effort. At this point in time, it is not apparent which software architecture, if any, will provide enough benefits to justify its development cost but through the performance projection that we will perform later in this chapter, we hope to gain some understanding that will help developers reach their own conclusions.

Intuitively, data intensive operations like filtering and aggregation should be offloaded to the SmartSTOR. More generally, operations that rely solely on local data belonging to a single base relation are good candidates for offloading. In this study, we refer to operations that work on a single relation as single-relation operations. Such operations are the basis of database queries and includes table/index scan, sort, group by and partial aggregate functions. Note that single-relation operations can operate on base relations as well as derived relations that result, for example, from join operations. Because the derived relations may not be local to the smartSTOR, the offloading of single-relation operations that work on derived rather than base relations is generally more complicated and the benefit less clear. In this chapter, when we refer to the offloading of single-relation operations, we generally mean the offloading of single-relation operations that use only base relations.

Although single-relation operations are the basis of database queries, a typical decision support query involves a lot more than just these basic operations. To distribute more processing to the SmartSTORs, we have to consider offloading multiple-relation operations such as joins that may involve data in one or more SmartSTORs. At the extreme, this is functionally equivalent to running a complete shared-nothing database management system (DBMS) [DGS$^+$90, LDSY91] such as IBM's DB2/EEE [IBM98] and an operating system on each SmartSTOR. Such an approach has hefty resource requirements but it may be possible to trim the DBMS to contain only the functionality profitable for offloading and to use less memory-intensive algorithms.

5.5 Projection Methodology

In this section, we outline the methodology that we use to assess the effectiveness of the SmartSTOR architecture and the relative merits of the various hardware and software organizations. There has been some recent work on evaluating the performance of Active Disks [AUS98a, AUS98b, RGF98, UAS98] but these have concentrated on image processing applications and basic database operations. Because decision support workloads represent an increasing fraction of the commercial workload [Ber98] and are growing so rapidly as to be pushing the limits of current system designs [WA97, WA98], we focus primarily on projecting how well they will perform on a SmartSTOR architecture. Our projection is based on the Transaction Processing Performance Council Benchmark D (TPC-D) [Tra97], which is the industry standard benchmark for decision support. A brief description of the benchmark is provided in Appendix A. Readers who are interested in the characteristics of the benchmark are referred to [HSY01a, HSY01b], which contain a comprehensive analysis of the benchmark characteristics and how they compare with those of real production database workloads.

The results reported in this chapter are based on TPC-D version 1 since at the time of this study, it has the largest number of published results. When there are enough TPC-H [Tra99] results, it would be interesting to redo this analysis to see whether the same trends are observed with the new benchmark. The current study examines results that were published between July 1998 and January 1999. Results after January 1999 have been omitted because we believe that these setups have been so fine-tuned for running the benchmark that attempting to lump them in with the other results would be meaningless. In particular, the aggressive use of Automatic Summary Tables (ASTs), *i.e.,* auxiliary tables that contain partially aggregated data, enable processing to be effectively pushed to the database load phase, which is not part of the TPC-D performance metric, so that very little processing needs to be performed when executing the queries. Nevertheless, if we have enough such results, it might be interesting to apply the same analysis to them as a separate group.

5.5.1 I/O Bandwidth

There are two likely performance advantages to the SmartSTOR architecture: (a) the amount of data that needs to be moved from the disks to the host for processing should be significantly reduced; (b) the actual processing can be offloaded from the host and done in parallel by the many processors within the whole system. Since decision support workloads are very data intensive, it is generally believed that they will benefit substantially from the potential decrease in I/O traffic.

Based on measurements[1] performed on several certified TPC-D setups, we have been able to establish a simple rule of thumb relating the TPC-D scale factor to the physical I/O bandwidth required. More specifically, we find that for a database of scale S, a total of about $3 \cdot S$ GB of data are transferred between the host and storage system during a TPC-D power test. With improvements in the memory capacity of the host system and more sophisticated database optimization, the constant 3 is expected to gradually decrease over time. Our measurements also indicate that the peak bandwidth requirement is about 3.3 times the average. Therefore, we can estimate the I/O bandwidth consumed during a TPC-D run by:

$$\textit{Average I/O bandwidth} \approx \frac{3 \cdot S}{\textit{total run time}} \tag{5.1}$$

$$\textit{Peak I/O bandwidth} \approx \frac{10 \cdot S}{\textit{total run time}} \tag{5.2}$$

Note that these rules of thumb are based on measurements conducted without the use of ASTs. With ASTs, the I/O bandwidth required will be lower.

In Table 5.1, we apply these rules of thumb to estimate the I/O bandwidth consumed in the TPC-D benchmark runs with results published during the period between July 1998 and January 1999. The highest per node I/O bandwidth consumption (1252 MB/s peak) is observed on a 32-processor system with a 12.5 GB/s system bus and which can be configured with 32 PCI buses each having a peak bandwidth of 528 MB/s. This puts the peak bandwidth consumed at about 10% of the bandwidth available. The highest per processor I/O bandwidth consumption is about 48.4 MB/s peak and occurs on an 8-processor system with a 3.2 GB/s system bus. This system can be configured with eight 528 MB/s PCI buses. Such results suggest that decision support workloads similar to TPC-D may not impose extra I/O bandwidth burden over that required for other workloads that today's systems are designed to handle.

To further understand this rather surprising finding, we examine the query execution plans from one of the TPC-D setups certified during the selected period. These plans are presented in Appendix E. Of the 17 TPC-D queries, only Query 16 uses a table scan and it is of the SUPPLIER table which contains only about 0.1% of the total number of records in the database. All the other accesses rely on an index in one way or another. In this particular TPC-D setup, a total of twenty-six indices are defined over the eight relations.

[1] Measurements taken in IBM benchmark labs.

	System	Average MB/s		Peak MB/s	
		Per Node	Per Processor	Per Node	Per Processor
100GB	Sun Enterprise 3500	116	14.5	387	48.4
	NEC Express 5800 HV8600	74.3	9.28	248	30.9
	IBM Netfinity 7000 M10	36.7	9.16	122	30.5
	IBM RS/6000 S70	55.0	4.58	183	15.3
	IBM NetFinity 7000 M10	37.1	9.27	124	30.9
	Compaq ProLiant 7000	41.3	10.3	138	34.4
	NCR 4400	24.4	6.09	81.2	20.3
	Compaq Digital Alpha 4100	20.2	5.06	67.4	16.9
	Average	50.6	8.54	169	28.5
300GB	IBM RS/6000 SP model 550	10.8	2.69	35.9	8.97
	Compaq Alpha Server GS140	42.8	4.27	143	14.2
	Sequent NUMA-Q 2000	150	4.68	499	15.6
	SGI Origin 2000	91.3	2.85	304	9.50
	HP 9000 V2250	70.6	4.41	235	14.7
	HP NetServer LXr 8000	25.4	6.36	84.7	21.2
	NCR 4400	22.8	5.71	76.1	19.0
	Average	59.1	4.42	197	14.8
1TB	Sun Starfire Enterprise 10000	375	5.87	1252	19.6
	IBM Netfinity 7000 M10	9.81	2.45	32.7	8.17
	Sequent NUMA-Q 2000	239	3.73	796	12.4
	Sun Starfire Enterprise 10000	281	4.40	938	14.7
	Average	226	4.11	754	13.7

Table 5.1: Estimated I/O Bandwidth Consumed during TPC-D Power Test.

	System	# Host Processors	# Disks	Ratio
100GB	Sun Enterprise 3500	8	138	17.3
	NEC Express 5800 HV8600	8	129	16.1
	IBM Netfinity 7000 M10	4	94	23.5
	IBM RS/6000 S70	12	215	17.9
	IBM NetFinity 7000 M10	4	94	23.5
	Compaq ProLiant 7000	4	84	21.0
	NCR 4400	4	43	10.8
	Compaq Digital Alpha 4100	4	57	14.3
	Average	6	107	18.0
300GB	IBM RS/6000 SP model 550	96	816	8.50
	Compaq Alpha Server GS140	40	512	12.8
	Sequent NUMA-Q 2000	32	263	8.22
	SGI Origin 2000	32	209	6.53
	HP 9000 V2250	16	202	12.6
	HP NetServer LXr 8000	4	89	22.3
	NCR 4400	4	63	15.8
	Average	32	308	12.4
1TB	Sun Starfire Enterprise 10000	64	1085	17.0
	IBM Netfinity 7000 M10	128	928	7.25
	Sequent NUMA-Q 2000	64	809	12.6
	Sun Starfire Enterprise 10000	64	1085	17.0
	Average	80	977	13.5

Table 5.2: Disk/Processor Ratio of TPC-D Setups with Results Published between July 1998 and January 1999.

Perhaps as a reflection of the fact that the TPC-D benchmark has been well studied and understood, there are many cases of index-only-access in which all the required fields are defined in the indices. The results suggest that the judicious use of techniques such as indices can be extremely effective at reducing the amount of I/O bandwidth required to support a TPC-D-like decision support workload. Therefore, for the rest of this chapter, we will concentrate on the other potential advantage of SmartSTOR, namely the ability to use the processing power in the storage system to perform some host processing.

5.5.2 System Configuration

The first step in projecting the performance of TPC-D on the SmartSTOR architecture is to determine the number of SmartSTORs that will be in the system and the processing power that they will possess. As is typical of forward-looking studies, we assume that some aspect of the system, in this case the number of disks, will remain the same. Table 5.2 summarizes the relevant configuration information for the selected TPC-D results. Recall that *cardinality* is the number of disks per SmartSTOR. For each setup, we project the number of SmartSTORs in the corresponding future system by:

$$\textit{num-SmartSTOR} = \frac{\textit{num-disk}}{\textit{cardinality}} \tag{5.3}$$

In order to describe the processing power available in the SmartSTORs without using absolute and therefore time frame dependent numbers, we introduce the notion of *performance per disk* (perf-per-disk), which is the effective processing power per disk relative to the host processor.

$$\textit{perf-per-disk} = \frac{\textit{processing power per SmartSTOR}}{\textit{processing power of host processor} \cdot \textit{cardinality}} \tag{5.4}$$

The actual value of perf-per-disk depends on the cardinality, family and generation of processors used, the power budget, the system design, *etc.* and is open to debate. In general, we believe that if the processor is embedded in the disk as opposed to the adapter or outboard controller, it will tend to have lower performance because of the smaller power budget and the much more stringent cost constraints in the disk environment. For an intelligent adapter or controller, the embedded processor may perhaps be even as powerful as a host processor, although that is unlikely to be cost effective. In either case, the embedded processor is likely to be used also for tasks, some of which are real-time, that are previously performed in special-purpose hardware. Since it is premature to specify precise values for perf-per-disk, we perform sensitivity analysis on the parameter later in this chapter.

5.5.3 Performance with Single-Relation Offloading

Recent work has shown that single-relation operations such as SQL select and aggregation can be very effectively offloaded to suitably enhanced disks [AUS98a]. However, a typical decision support query is comprised of many operations other than those that rely on data from a single relation. Moreover, in most cases, the results of the single-relation operations are combined through joins to create new derived relations that are further operated on. Therefore, to establish the actual performance effect of single-relation offloading, we need to determine the fraction of work represented by the single-relation operations and that can be delegated to the SmartSTOR.

Our method for determining the portion of work that can be offloaded is to analyze the query execution plans. Measuring the CPU time needed for each individual operation in a query execution plan is extremely difficult because the operations are executed simultaneously in parallel or in a pipelined fashion. Therefore, we use the CPU costs estimated by the query optimizer. The results presented in this chapter are based on the query execution plans from two certified TPC-D setups. In order to understand the possible range of values in the fraction of work that can be offloaded, we consider both a shared-everything and a shared-nothing DBMS. The first system we consider is a Symmetric Multiprocessor System (SMP) running IBM's DB2/UDB [IBM97], a shared-everything DBMS, while the second

Query	System 1 (Shared-Everything)	System 2 (Shared-Nothing)
1	100	99.9
2	6.8	2.3
3	0.3	12.9
4	4.0	2.7
5	5.2	94.6
6	41.8	44.1
7	0.2	53.4
8	21.8	9.0
9	9.0	5.5
10	2.1	1.6
11	0.4	0.1
12	8.0	7.8
13	0.3	6.4
14	13.3	91.6
15	48.2	41.9
16	0.0	0.2
17	0.1	49.1

Table 5.3: Percent of Work that can be Offloaded by Single-Relation Offloading.

system is a cluster-based setup running the shared-nothing IBM DB2/EEE [IBM98]. The complete set of plans from the first system is available in Appendix F.

Table 5.3 summarizes the fraction of processing that can be offloaded by single-relation offloading for the two TPC-D setups. From the table, Query 1 is the only query that can be offloaded by more than 50% in System 1. Observe further that only five out of the 17 queries can be offloaded by more than 10% in System 1. System 2 is generally more amenable towards single-relation offloading but it is still the case that less than half of the queries can be offloaded by more than 10%. According to Amdahl's Law [HP96], these statistics suggest that the performance potential of single-relation offloading may be limited. However, the fact that there is substantial difference between the figures for the two setups suggest that there may be considerable room for improving the plans generated to better take advantage of the SmartSTOR architecture. This is an area that requires further research.

Suppose that f is the fraction of processing that can be performed by the SmartSTOR. Assuming that host and SmartSTOR processing are maximally overlapped, the speedup that can be achieved by single-relation offloading is:

$$\textit{speedup} = \frac{1}{Max(1-f, \frac{f}{s})}$$

where

$$s = \frac{\textit{num-disk}}{\textit{num-host-proc}} \cdot \textit{perf-per-disk}$$

is the aggregate processing power available in the SmartSTORs relative to that in the host. If we further assume that the system will be intelligent enough to offload operations only when it makes sense to do so, the speedup is:

$$\textit{speedup} = Max(1, \frac{1}{Max(1-f, \frac{f}{s})}) \tag{5.5}$$

As we shall see, even with such optimistic assumptions, the performance potential of single-relation offloading is rather limited.

Assuming that the current run time for query i is $QI(i)$, we can project the run time for the query on a SmartSTOR architecture, $QI(i)'$, by:

$$QI(i)' = \frac{QI(i)}{\textit{speedup}} \tag{5.6}$$

The TPC-D benchmark defines both a power metric and a throughput metric [Tra97]. Since we are primarily interested in speedups in this study, we focus on the power metric, QppD. In determining the average performance improvement possible in a SmartSTOR architecture with single-relation offloading, we use the projected query run times, $QI(i)'s$, to determine the increase in QppD for each of the 19 selected TPC-D systems. Then we take the arithmetic mean over the 19 setups to obtain an average improvement in QppD. Note that QppD includes the execution times of two update functions, which we assume cannot be offloaded by SR. Also, as discussed in Appendix E and [Tra97], the definition of QppD limits the run time of any query to be at most 1000 times shorter than that of the slowest query.

5.5.4 Performance with Multiple-Relation Offloading

In general, when work is distributed across multiple processing units, skew comes into play so that the performance of the system scales sublinearly with the number of processing units. For a well-understood workload such as TPC-D, we can try to distribute the tuples in the base relations evenly across the SmartSTORs so as to minimize any data skew. Therefore, for single-relation offloading, the portion of work offloaded is likely to be sped up by the extra processing power available in the SmartSTORs. However, for more complicated operations that involve redistributing tuples or that involve derived relations, there is likely to be an unequal distribution of relevant tuples across the SmartSTORs.

In order to project the performance of TPC-D when multiple-relation operations are offloaded, we need to understand how effectively the work can be distributed across the SmartSTORs, *i.e.,* we need to understand the scalability of the system. Since we are not aware of any generally accepted model of scalability for TPC-D, we empirically derive a model by using the TPC-D results published during the selected period. Because these results were obtained on systems with different processors, we have to first normalize them. Let:

$$\textit{database efficiency} = \frac{\textit{QppD}}{\textit{SPECintbase95} \cdot \textit{num-host-proc}} \tag{5.7}$$

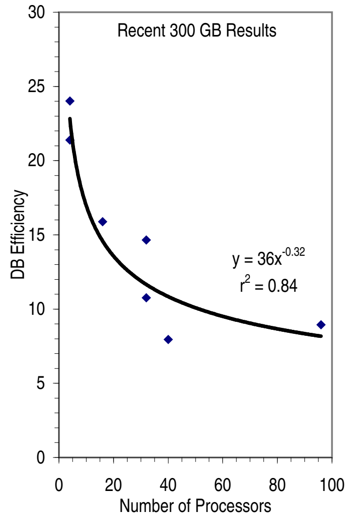


Figure 5.3: Scalability of TPC-D Systems.

SPEC measurements [Sta95] are more indicative of performance on CPU intensive integer and floating point workloads, rather than on the CPU portion of database system and operating system workloads, but we are not aware of any better alternative; we therefore normalize TPC-D performance by SPEC numbers [Sta95]. We believe that the errors introduced should be small.

Figure 5.3 plots the database efficiency of the 300 GB TPC-D results. We choose to use the 300 GB results because the benchmark setups for this scale factor have a wide range in the number of processors used. Observe that the set of points can be roughly approximated by $\frac{C}{\sqrt[3]{\textit{num-host-proc}}}$, where C is a constant. We refer to this scalability rule as the *cube root rule* because the per processor efficiency is halved when the number of processors is increased by a factor of eight. We expect the scalability of the system to improve with advances in both hardware and software. Therefore, we use the *fourth root rule* to consider future TPC-D system scalability. With the fourth root rule, the per processor efficiency is halved when the number of processors is increased by a factor of 16. Note that real workloads are unlikely to be as well understood and tuned as the TPC-D benchmark and the processing will tend to be less well distributed. In other words, real workloads will probably scale more poorly with the number of processors. Therefore, we also consider the *square root rule*.

Using these scalability rules, we can establish a relationship between QppD and the number of processors and their processing power.

$$\begin{aligned} \textit{QppD} &= \textit{database efficiency} \cdot \textit{SPECintbase95} \cdot \textit{num-host-proc} \\ &= C \cdot \textit{SPECintbase95} \cdot \textit{num-host-proc}^{1-\frac{1}{n}} \end{aligned} \tag{5.8}$$

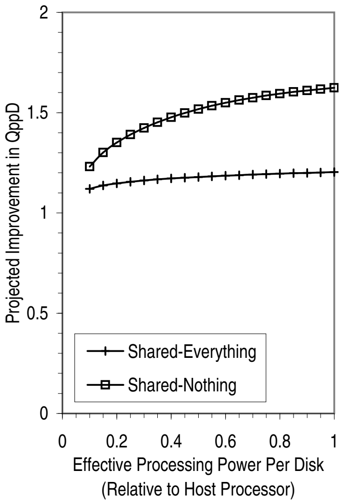


Figure 5.4: Projected Improvement in TPC-D Performance with Single-Relation Offloading.

where

$$n = \begin{cases} 2 & \textit{for the square root rule,} \\ 3 & \textit{for the cube root rule,} \\ 4 & \textit{for the fourth root rule.} \end{cases}$$

In a SmartSTOR environment,

$$\begin{aligned} \textit{QppD} &= C \cdot \textit{SPECintbase95}_{\textit{SmartSTOR}} \cdot \textit{num-SmartSTOR}^{1-\frac{1}{n}} \qquad (5.9) \\ &= C \cdot \textit{perf-per-disk} \cdot \textit{cardinality} \cdot \textit{SPECintbase95}_{\textit{host}} \cdot \textit{num-SmartSTOR}^{1-\frac{1}{n}} \end{aligned}$$

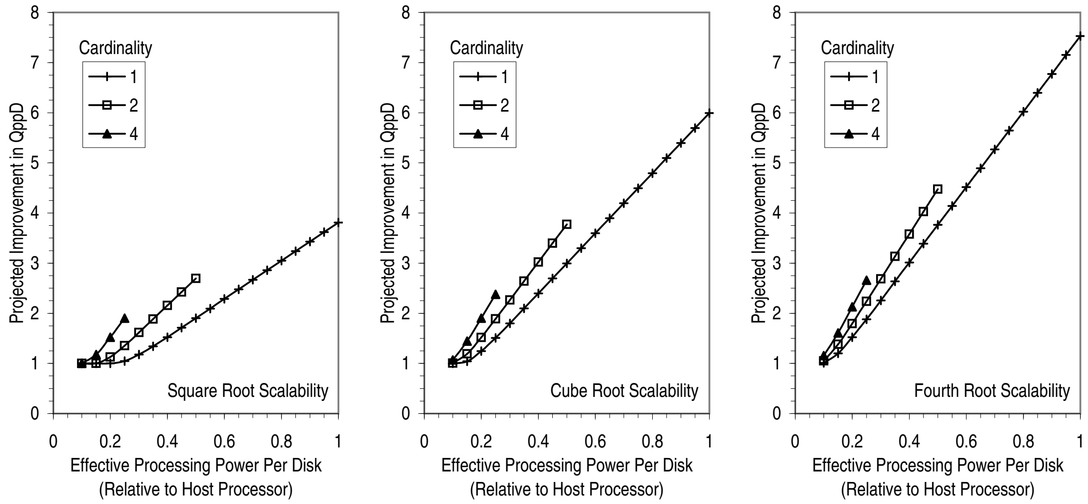
Therefore, assuming that the system will be intelligent enough to offload operations only when it improves performance, the speedup is:

$$\textit{speedup} = Max(1, \quad \textit{perf-per-disk} \cdot \textit{cardinality} \cdot \left(\frac{\textit{num-SmartSTOR}}{\textit{num-host-proc}}\right)^{1-\frac{1}{n}}) \qquad (5.10)$$

Using this result, the improvement in QppD can be projected for each of the 19 selected TPC-D systems. As in the case for single-relation offloading, we take the arithmetic mean over the 19 setups to obtain the average projected improvement in QppD.

5.6 Analysis of Performance Results

Based on the steps outlined in the previous section, we can analytically derive the improvement in QppD for the various hardware and software options. The results for single-relation offloading are summarized in Figure 5.4 while those for multiple-relation



(a) Square Root Scalability. (b) Cube Root Scalability. (c) Fourth Root Scalability.

Figure 5.5: Projected Improvement in TPC-D Performance with Multiple-Relation Offloading.

offloading are in Figure 5.5. In these figures, we plot the projected speedup for the various alternatives as a function of the effective processing power per disk. As in the rest of this dissertation, we define improvement or speedup as $(value_{old} - value_{new})/value_{old}$ if a smaller value is better and $(value_{new} - value_{old})/value_{old}$ otherwise.

For multiple-relation offloading, we plot the projected performance improvement with the square, cube and fourth root scalability rules to indicate the range of speedup that can be expected. For single-relation offloading, we plot the speedup given by the two sets of offloading fractions discussed in Section 5.5.3 and presented in Table 5.3. Note that the figures make no cost statement. This is deliberate since accurate cost information are generally closely guarded and in any case, are very technology and time frame dependent. Given a set of cost estimates, Figures 5.4 and 5.5 can be used to determine whether SmartSTOR is a cost-effective approach and if so, the configuration that should be used.

Recall from our scalability model that for multiple-relation offloading, TPC-D performance tends to scale rather sublinearly with the number of processors used. This shows up in Figure 5.5 in that for the same perf-per-disk, a larger cardinality has a performance advantage. In other words, for a given aggregate processing power, it is significantly more effective to share powerful processors among multiple disks than to have less powerful per-disk processors. This may not be the most cost effective solution, however; current pricing policies are such that prices go up more than linearly with processor speed. Note also that the performance effect of sharing a processor among multiple disks is limited by the fact that there are no arbitrarily powerful processors. In this chapter, we assume that the embedded processor is at most as powerful as the host processor so that the maximum perf-per-disk is given by $\frac{1}{cardinality}$.

Observe from Figures 5.4 and 5.5 that multiple-relation offloading clearly outperforms single-relation offloading for practically all values of perf-per-disk. However, by distributing more processing to the SmartSTORs and harnessing more of the parallelism in the system, multiple-relation offloading is likely to require a lot more resources, particularly memory, in the SmartSTORs. As we have discussed earlier, multiple-relation offloading in its broadest sense is functionally equivalent to running a complete shared-nothing DBMS and an operating system on each SmartSTOR. An interesting research question is whether it is possible to effectively separate out the functionality profitable for offloading and to use new algorithms that are less memory-intensive. However, as we have alluded to earlier, the performance potential of the SmartSTOR architecture has to be evaluated relative to the cost of any software reengineering, and to the cost of building the SmartSTORs themselves.

Our results can also be used to estimate the potential performance benefit of the IDISK and the Active Disk. These per-disk proposals are conceptually identical to an intelligent adapter or controller of cardinality one with the exception that they are likely to have a lower perf-per-disk. As discussed earlier, the exact value of perf-per-disk is arguable but with the much more stringent power and cost constraints in the disk environment, the processors that are embedded in the disk are likely to be significantly less powerful than those in a SmartSTOR. With a perf-per-disk value of 0.25, which is likely in a per-disk environment, the projected improvement in QppD ranges from 1.16 to 1.39 for single-relation offloading and from 1.05 to 1.88 for multiple-relation offloading. For comparison, this ratio of processing power (0.25) is about equivalent to that between a 200 MHz Intel Pentium MMX and a 575 MHz Compaq Alpha 21264 (based on SPECintbase95).

As mentioned earlier, a possible approach to building SmartSTORs is to decontent standard PCs. In this case, a rough value of perf-per-disk for a SmartSTOR may be $\frac{0.8}{cardinality}$. Based on this perf-per-disk value, the projected speedup in QppD for cardinalities of 1, 2 and 4 with multiple-relation offloading ranges from 3.05 to 6.02, from 2.15 to 3.58 and from 1.52 to 2.13 respectively. For single-relation offloading, the corresponding ranges are 1.20-1.59, 1.17-1.48 and 1.15-1.35.

An important point to note is that among all the published TPC-D results so far, the largest number of processors used is only 192 while the largest number of disks used is over 1,500. Therefore, to effectively take advantage of the large number of processors that are likely to be in a SmartSTOR-based system, we have to focus on improving the scalability of parallel software systems.

5.7 Conclusions

In this chapter, we propose a general Smart Storage (SmartSTOR) architecture in which general purpose processing can be performed by a processing unit that is shared among one or more disks. The SmartSTOR architecture provides two key performance advantages, namely a reduction in I/O movement between the host and I/O subsystem, and the ability to offload some of the work from the host processor to the processing units in the SmartSTORs.

In order to understand the performance potential of the SmartSTOR architecture for decision support workloads, as well as the various hardware and software tradeoffs, we project the performance of the industry-standard decision support benchmark, TPC-D, on various SmartSTOR-based systems. In particular, we perform measurements on several certified TPC-D systems to estimate the I/O bandwidth required for supporting such workloads. We also examine the query execution plans from two TPC-D systems to determine the amount of processing that can potentially be offloaded to the SmartSTORs. In addition, we analyze published TPC-D performance figures to empirically establish a scalability rule that can be used to project the effectiveness of distributing query execution among a large number of SmartSTORs.

Our results indicate that I/O bandwidth may not be that serious a bottleneck for TPC-D. Therefore the main advantage of using SmartSTORs for workloads similar to TPC-D appears to be the ability to offload some of the processing from the host. Further analysis reveals that offloading database operations that involve only a single base relation tends to have limited performance benefit. The offloading of operations that involve multiple relations appears much more promising, but requires significantly more resources. For multiple-relation offloading, we find that performance scales rather sublinearly with the number of processors used. Therefore, for the same aggregate processing power, it is much more effective to share powerful processing units among multiple disks than to have a greater number of less powerful processors. Our analysis also suggests that to effectively offload processing to the storage system, we need parallel software systems that can efficiently utilize the large number of processing units that will likely be in the system.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The slow mechanical nature of I/O devices, most importantly disks, compared to the speed of electronic processing, has long been recognized. In order to keep the processor supplied with data, systems rely on aggressive I/O optimization techniques that can be tuned to specific workloads. But as the improvement in processor performance continues to far exceed the improvement in disk access time, the I/O bottleneck is increasingly an issue. We now resort more and more to expensive measures for increasing I/O performance such as configuring systems with large amounts of memory as the I/O cache or using many more disks than storage requirements warrant. As systems continue to grow in complexity over and beyond our ability to cost-effectively manage them, what is really needed is a storage system that delivers good performance without requiring a lot of resources and time to configure and tune, even as the workloads evolve.

This dissertation has explored mechanizable techniques for improving I/O performance by dynamically optimizing disk block layout in response to the actual usage pattern. It is based on the observation that it is much more efficient to read a large contiguous chunk of data than many small chunks scattered throughout the disk. Users, however, typically have only limited knowledge and control of how their data are laid out on the disk, and most would rather not be thus burdened. The file system or application can guess how blocks are likely to be used based on static logical information such as name space relationships but such information may not accurately reflect the actual dynamic usage pattern. On the other hand, technology trends are such that disk space and processing power are increasingly available for performing sophisticated optimizations without user involvement. Therefore we contend that it is useful and practical for the storage system to automatically adapt block layout to the dynamic reference behavior so as to increase the spatial locality of reference and allow it to effectively fetch data in larger chunks.

The major contributions of this work have been as follows. First, we investigated how storage systems are actually being used to gauge both the feasibility and the effectiveness of automatically optimizing the block layout. Our analysis, which is based on multi-week traces of the I/O activity of a wide variety of both personal computer and server systems, indicates that improving I/O performance is important, and that there are idle resources that can potentially be used to perform any optimization. In addition, we find that collecting a daily trace of the requests for later analysis and optimization is feasible. We also discover that only a small fraction of the data stored is in active use, which suggests that it will likely be useful to identify the blocks that are in use and to optimize their layout. A deeper analysis of the dependencies between the read and write requests suggests that if blocks are reorganized or replicated based on their read access patterns, write performance will not be significantly affected, and that if blocks are replicated, we should update only one of the copies and invalidate the rest.

As part of our analysis, we reexamine Amdahl's rule of thumb for a balanced system and discover that our server workloads generate on the order of 0.05 bits of physical I/O per instruction, consistent with our earlier work using the production database workloads of some of the world's largest corporations [HSY01a]. We also find that the average request size is about 8 KB. In addition, we observe that the I/O traffic is bursty in a self-similar sense, which implies that the I/O system may become overwhelmed at much lower levels of utilization than expected with the commonly assumed Poisson model. Such behavior has to be taken into account when designing storage systems, and in the service level agreements (SLAs) when outsourcing storage. To make our results more generally applicable, we also study the effect of increased upstream caching on the traffic characteristics seen by the storage system and show that it affects our analysis only quantitatively.

Based on the results of our workload characterization, we next discuss the importance of modeling both the burstiness in the I/O traffic and the feedback between request completion and subsequent I/O arrivals, and develop a simulation methodology that incorporates both effects. Using this methodology, we systematically study the many previously proposed I/O optimization techniques to establish their actual and relative effectiveness at improving I/O performance. In the process, we obtain several new insights about the various techniques and establish an optimized baseline configuration for our subsequent experiments.

Our results show that sequential prefetch, especially when coupled with the ability to prefetch opportunistically, offers by far the most significant improvement in read performance, reducing the average read response and service times by about half. In addition, we develop a framework for effectively performing write buffering that not only hides the latency of writes but also reduces the number of physical writes and enables the remaining physical writes to be performed efficiently. Our simulations show that write buffering, when properly performed, is able to reduce the average write response time by more than 90% and the average write service time by more than 70%. We also demonstrate that a large stripe unit in the megabyte range performs well, and that short-stroking or using less of each disk improves performance by only up to 10-15%.

In addition to evaluating the various I/O optimization techniques, we also analyze how the continuous improvement in disk technology affects the actual I/O performance seen by real workloads. Increases in the recording density are often neglected when projecting effective disk performance. But our results clearly indicate that areal density improvement has as much of an impact on effective I/O performance as the mechanical improvements. Overall, we expect the I/O performance for a given workload with a constant number of disks to increase by about 15% per year due to the evolution of disk technology. If the workloads are scaled up or fewer disks are used to take advantage of the larger capacity of the new disks, we expect the improvement to be about 9% per year.

Along the way, we also establish several rules of thumb that are generally useful for system designers. For instance, we discover that the read miss ratio of the storage cache decreases as the inverse of the ratio of cache size to storage used (allocated). For larger write buffers, our results indicate that the write miss ratio follows a fifth root rule, meaning that the miss ratio goes down as the inverse fifth root of the ratio of buffer size to storage used. Because of locality of reference and request scheduling, we observe that for our workloads the average actual seek time is about 35% of the advertised average seek time for the disk, and the average actual rotational latency is about 80% of the value specified.

Our results confirm that the disk spends most of its time positioning the disk head and very little time actually transferring data. With technology trends being the way they are, it will become increasingly difficult to effectively utilize the available disk bandwidth. Therefore, we next present ALIS, an introspective storage system that continually analyzes I/O reference patterns to replicate and reorganize selected disk blocks so as to improve the spatial locality of reference, and hence leverage the dramatic improvement in disk transfer rate. Our analysis of ALIS suggests that disk block layout can be effectively optimized by an autonomic storage system, without human intervention. Specifically, we find that the idea of clustering together hot or frequently accessed data has the potential to significantly improve storage performance, if handled in such a way that existing spatial locality is not disrupted. In addition, we show that by discovering repeated read sequences or runs and laying them out sequentially, we can greatly increase the effectiveness of sequential prefetch. By further combining these two ideas, we are able to reap the greater benefit of the two schemes and achieve performance that is superior to either technique alone.

Using extensive trace-driven simulations, we demonstrate that the mechanizable techniques we develop for reorganizing data can dramatically improve performance despite efforts already made by the database and the file system to optimize the layout of data. Our results clearly show that ALIS is able to far outperform prior reorganization techniques in both an environment where the storage system consists of only disks and low-end disk adaptors, and one where there is a large outboard controller. For the server workloads, read performance is improved by between 31% and 50% while write performance is improved by as much as 22%. Given that disk performance, as perceived by real workloads, is increasing by about 9% per year assuming that the disk occupancy rate is kept constant, such improvements may be equivalent to as much as several years of technological progress at the historical rates. The read performance for the PC workloads is improved by about 15% while the writes are faster by up to 8%. In general, the PC workloads tend to be improved less than the server workloads because the tasks typically handled by a PC are more diverse and less repetitive. Servers, on the other hand, tend to perform similar tasks for many different users so that their access patterns are more predictable. We also examine how improvement in disk technology will impact the effectiveness of ALIS and confirm that the benefit of ALIS is relatively insensitive to disk technology trends.

Besides enabling sophisticated optimizations such as ALIS, the increasingly available processing power in the storage system can also be used to offload application processing, parts of file system functionality (*e.g.,* object-based storage), *etc.* from the host system. Furthermore, recent advances in software technology make it easier than previously to use this processing capacity for application processing. In the final part of this dissertation, we turn our attention to analyzing the effectiveness of offloading application processing to the storage system. We devise a methodology to estimate the effectiveness of distributing a database query among multiple processing units in the storage system by empirically deriving the system scalability relationship using the published results for the industry-standard benchmark for decision support workloads, namely the Transaction Processing Performance Council Benchmark D (TPC-D) [Tra97]. In addition, we examine the query execution plans from two certified TPC-D systems to establish the fraction of work that can potentially be offloaded. We find that for the same aggregate processing power, it is much more effective to share powerful processing units among multiple disks than to have a greater number of less powerful processors. Our results also suggest that to take full advantage of the ability to offload database operations to a large number of processing units in the storage system, significant advances and reengineering of the database system are needed.

6.2 Future Directions

We believe that because of the widening gap between processor and disk performance, and the soaring costs of system management, much interesting and important work remains to be done in the area of I/O performance, especially in methods to achieve good performance right out of the box, without intricate tuning.

We have evaluated the various I/O optimization techniques assuming that each is performed at most once in the storage stack. This helps us to systematically focus on the real effect of the optimizations. In practice, however, each of the optimizations may be performed independently at multiple levels in the storage hierarchy. For instance, there may be several storage controllers, storage adaptors and disk drives, and they may all perform some of the optimizations to some extent. It will be interesting to study the interaction between the optimizations performed at the various levels to understand the effect and to minimize any destructive interference.

ALIS currently behaves like an open-loop control system that is driven solely by the workload and a simple performance model of the underlying storage system, namely that it is able to service sequential I/O much more efficiently than random I/O. Because the performance of disks today is so much higher when data is read sequentially rather than randomly, this simple performance model is sufficiently accurate to produce a dramatic performance improvement. But for increased robustness, it would be worthwhile to explore the idea of incorporating some feedback into the optimization process to, for example, turn ALIS off when it is not performing well or, at a finer granularity, influence how blocks are laid out.

In the design of ALIS, we try to desensitize its performance to the various parameters so that it is not catastrophic for somebody to "configure the system wrongly". To reflect the likely situation that ALIS will be used with a default setting, we base our performance evaluation on parameter settings that are good for an entire class of workloads rather than on the best values for each individual workload. A useful piece of future work would be to devise ways to set the various parameters dynamically to adapt to each individual workload. The general approach of using hill-climbing to gradually adjust each knob until a local optimum is reached should be very effective for ALIS because the results of our various sensitivity studies suggest that for the parameters in ALIS, the local optimum is also likely to be the global optimum.

In run clustering, the edges of the access graph represent the desirability for reorganization units to be located one after another. But if the reorganization units are already located sequentially, it might not be necessary to replicate and reorganize them. We believe, however, that copying these units into the reorganized region would still be beneficial. Nevertheless, it would be interesting to explore the effect of a subtle change to make the edges reflect the desirability for reorganization units *to be relocated* so that they are positioned one after another. In this case, when we add an edge from vertex i to vertex j, we would weight it by some estimate of the current cost of reading reorganization unit j after reading unit i.

After a reorganization unit is added to a run, the run clustering algorithm marks it to prevent it from being included again in any run. An interesting variation of the algorithm would be to allow multiple copies of a reorganization unit to exist either in the same run or in different runs. This is motivated by the fact that some data blocks, for instance those corresponding to shared libraries, may appear in more than one access pattern. The basic idea in this case would be to not mark a vertex after it has been added to a run. Instead, we would remove the edges that are used to include that particular vertex in the run.

By using extensive trace-driven simulations, we have demonstrated that ALIS has the potential to very significantly improve I/O performance. The simulations enable us to explore many aspects of the design, especially on the algorithmic front. Our simulator is based on a disk model that is used by a disk development team and that has been validated against several batches of the disk. Furthermore, because our simulator has been used successfully to explore a wide variety of I/O optimization techniques, we are confident that it accurately models most of the important effects. However, a full implementation and deployment of ALIS will help to ensure that all the complexities of the real world have been accounted for, and will give us some indication of the overheads that may be involved.

Finally, in analyzing the effectiveness of offloading application processing to the storage system, our results are somewhat limited by the small number of TPC-D benchmark results that are available. But the same methodology can be applied elsewhere and on different benchmarks. Since our study, the TPC-D benchmark has been superseded by the

TPC-H [Tra99] benchmark. It will be worthwhile to see if the same trends are observed with the new benchmark.

Bibliography

[ADHW99] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. DBMSs on a modern processor: Where does time go. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, September 1999.

[Amd70] Gene Myron Amdahl. Storage and I/O parameters and systems potential. In *Proceedings of IEEE International Computer Group Conference (Memories, Terminals, and Peripherals)*, pages 371–372, Washington, DC, June 1970.

[AO97] Lyle Adams and Michael Ou. Processor integration in a disk controller: Embedding a RISC processor in a complex ASIC to reduce cost and improve performance. *IEEE Micro*, 17(4):44–48, July/August 1997.

[AS95] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[AUS98a] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, October 1998.

[AUS98b] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Structure and performance of decision support algorithms on active disks. Technical Report TRCS98-28, Computer Science Dept., University of California, Santa Barbara, November 1998.

[AV98] Patrice Abry and Darryl Veitch. Wavelet analysis of long-range-dependent traffic. *IEEE Transactions on Information Theory*, 44(1):2–15, 1998.

[BD97] Dileep Bhandarkar and Jason Ding. Performance characterization of the Pentium Pro processor. In *Proceedings of International Symposium on High-Performance Computer Architecture (HPCA)*, pages 288–297, San Antonio, Texas, February 1997.

[Ber94] Jan Beran. *Statistics for Long-Memory Processes*. Chapman & Hall, New York, NY, 1994.

[Ber98] Philip A. Bernstein. Database Technology: What's Coming Next? Keynote speech at *Symposium on High Performance Computer Architecture (HPCA)*, February 1998. Slides available at http://tab.computer.org/tcca/HPCA-4/bernstein.ppt.

[BGW91] G. P. Bozman, H. H. Ghannad, and E. D. Weinberger. A trace-driven study of CMS file references. *IBM Journal of Research and Development*, 35(5/6):815–828, September/November 1991.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 198–212, Pacific Grove, CA, October 1991.

[BHMW98] Brian Eric Bakke, Frederic Lawrence Huss, Daniel Frank Moertl, and Bruce Marshall Walk. Method and apparatus for adaptive localization of frequently accessed, randomly addressed data. U.S. Patent 5765204, June 1998. Filed June 5, 1996.

[BOK+98] Aaron Brown, David Oppenheimer, Kimberly Keeton, Randi Thomas, John Kubiatowicz, and David A. Patterson. ISTORE: Introspective storage for data-intensive network services. Technical Report CSD-98-1030, Computer Science Division, University of California, Berkeley, December 23, 1998.

[BRT93] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. In *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 13–23, Santa Clara, CA, May 1993.

[CFH+80] M. D. Canon, D. H. Fritz, John H. Howard, T. D. Howell, Michael F. Mitoma, and Juan Rodriguez-Rossel. A virtual machine emulator for performance evaluation. *Communications of the ACM*, 23(2):71–80, 1980.

[CG99] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, New Orleans, LA, February 1999.

[CHY00] Ying Chen, Windsor W. Hsu, and Honesty C. Young. Logging RAID - an approach to fast, reliable, and low-cost disk arrays. In *Proceedings of European Conference on Parallel Computing (EuroPar)*, Munich, Germany, August 2000.

[CL95] Peter M. Chen and Edward K. Lee. Striping in a RAID level 5 disk array. In *Proceedings of ACM International Conference on Measurement and*

Modeling of Computer Systems (SIGMETRICS), pages 136–145, Ottawa, Canada, May 15–19 1995.

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[CLTR98] Chye Lin Chee, Hongjun Lu, Hong Tang, and C. V. Ramamoorthy. Adaptive prefetching and storage reorganization in a log-structured storage system. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):824–838, September 1998.

[CNC+96] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: Surviving operating system crashes. In *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Cambridge, MA, October 1996.

[Cor98] Intel Corp. Intel application launch accelerator, March 1998. http://www.intel.com/ial/ala.

[CR89] Scott D. Carson and Paul F. Reynolds Jr. Adaptive disk reorganization. Technical Report UMIACS-TR-89-4, Department of Computer Science, University of Maryland, January 1989.

[Dah95] Michael Donald Dahlin. *Serverless Network File Systems*. PhD thesis, University of California, Berkeley, December 1995.

[Den67] Peter J. Denning. Effects of scheduling on file memory operations. In *Proceedings of AFIPS Spring Joint Computer Conference*, pages 9–21, Atlantic City, NJ, April 1967.

[Den68] Peter J. Denning. The working set model for program behaviour. *Communications of the ACM*, 11(5), May 1968.

[DGS+90] David J. DeWitt, Shaharm Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui i Hsiao, and Rick Rasmusen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[DH81] D. J. DeWitt and P. B. Hawthorn. A performance evaluation of database machine architectures. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 199–214, September 1981.

[DMW+94] Michael Dahlin, Clifford Mather, Randolph Wang, Thomas Anderson, and David Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 150–160, May 1994.

[EMC01] EMC Corporation. Symmetrix™ 8830-36/-73/-181, 2001.

[Exe01] Executive Software International Inc. Diskeeper 6.0 second edition for Windows, 2001. http://www.execsoft.com/diskeeper/diskeeper.asp.

[Fer76] Domenico Ferrari. Improvement of program behavior. *Computer*, 9(11):39–47, November 1976.

[GA94] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of Summer USENIX Conference*, pages 197–207, June 1994.

[GAN93] Knuth Stener Grimsrud, James K. Archibald, and Brent E. Nelson. Multiple prefetch adaptive disk caching. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):88–103, February 1993.

[Gan95] Gregory R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, 1995. Available as Technical Report CSE-TR-243-95, EECS Department, University of Michigan, Ann Arbor, June 1995.

[GBS+95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Proceedings of USENIX Technical Conference*, pages 201–212, New Orleans, LA, January 1995.

[GG97] Jim Gray and Goetz Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[GK97] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *Proceedings of USENIX Technical Conference*, pages 1–17, Anaheim, CA, January 1997.

[GM96] James Gosling and Henry McGilton. The Java^{TM} language environment a white paper, May 1996. http://java.sun.com/docs/white/langenv.

[GMR+98] Steven D. Gribble, Gurmeet Singh Manku, Drew Roselli, Eric A. Brewer, Timothy J. Gibson, and Ethan L. Miller. Self-similarity in file systems. In *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 141–150, Madison, WI, June 1998.

[Gra98] Jim Gray. Put EVERYTHING in the storage device. Talk at NASD Workshop on Storage Embedded Computing, June 1998. Slides available at http://www.nsic.org/nasd/1998-jun/gray.pdf.

[Gro00] Ed Grochowski. IBM leadership in disk storage technology, 2000. http://www.storage.ibm.com/technolo/grochows/grocho01.htm.

[GS99] Maria E. Gómez and Vicente Santonja. Analysis of self-similarity in I/O workload using structural modeling. In *Proceedings of Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 234–242, College Park, MD, October 1999.

[GSS+02] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John S. Bucy, and Gregory R. Ganger. Timing-accurate storage emulation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 75–88, Monterey, CA, January 2002.

[GW02] Mark A. Gaertner and Joseph L. Wach. Rotationally optimized seek initiation. U.S. Patent 6339811, January 2002. Filed Dec. 28, 1999.

[GWP99] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, 1999.

[HCL+90] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[Hei94] R. R. Heisch. Trace-directed program restructuring for AIX executables. *IBM Journal of Research and Development*, 38(5):595–603, September 1994.

[HH95] John R. Heath and Stephen A. R. Houser. Analysis of disk workloads in network file server environments. In *Proceedings of Computer Measurement Group (CMG) Conference*, pages 313–322, Nashville, TN, December 1995.

[Hit02] Hitachi Data Systems. Lightning 9900™ : Specifications, 2002.

[HMP89] Ali R. Hurson, L. L. Miller, and S. H. Pakzad. *Parallel Architectures for Database Systems*. CS Press Tutorial, 1989.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc. San Francisco, CA, second edition, 1996.

[HSY01a] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. Analysis of the characteristics of production database workloads and comparison with the TPC benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[HSY01b] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. *ACM Transactions on Database Systems*, 26(1):96–143, March 2001.

[IBM96] IBM Corp. *AIX Versions 3.2 and 4 Performance Tuning Guide*, 5th edition, April 1996.

[IBM97] IBM Corp. *DB2 UDB V5 Administration Guide*, 1997.

[IBM98] IBM Corp. *DB2 Universal Database Extended Enterprise Edition for UNIX Quick Beginnings Version 5*, 1998.

[IBM00] IBM Corp. IBM TotalStorage™ Enterprise Storage Server Models F10 and F20, 2000.

[IBM01a] IBM Corp. *Autonomic Computing: IBM's Perspective on the State of Information Technology*, 2001. http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

[IBM01b] IBM Corp. *Ultrastar 73LZX Product Summary Version 1.1*, 2001.

[JW91] D. Jacobson and J. Wilkes. Disk scheduling algorithms based on rational position. Technical Report HPL–CSP–91–7, Hewlett-Packard Laboratories, Palo Alto, CA, USA, February 1991.

[KL96] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *Proceedings of USENIX Annual Technical Conference*, pages 319–328, January 1996.

[KLW94] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994.

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, second edition, 1998.

[KPH+98a] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. Performance characterization of a quad Pentium Pro SMP using OLTP workloads. In *Proceedings of ACM International Symposium on Computer Architecture (ISCA)*, pages 15–26, Barcelona, Spain, June 1998.

[KPH98b] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[KR96] Marie F. Kratz and Sidney I. Resnick. The QQ - estimator and heavy tails. *Stochastic Models*, 12:699–724, 1996.

[LD97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Anaheim, CA, January 1997.

[LDSY91] Raymond A. Lorie, Jean-Jacques Daudenarde, James W. Stamos, and Honesty C. Young. Exploiting database parallelism in a message-passing multiprocessor. *IBM Journal of Research and Development*, 35(5/6):681–695, September/November 1991.

[LS00] Jacob R. Lorch and Alan Jay Smith. The VTrace tool: Building a system tracer for Windows NT and Windows 2000. *MSDN Magazine*, 15(10):86–102, October 2000.

[LSG02] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 275–288, January 2002.

[LTWW94] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[Maj81] Joseph. B. Major. Processor, I/O path, and DASD configuration capacity. *IBM Systems Journal*, 20(1):63–85, 1981.

[Man82] Benoit B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, New York, 1982.

[McD88] Shane McDonald. Dynamically restructuring disk space for improved file system performance. Technical Report 88-14, Department of Computational Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada, July 1988.

[McN95] Bruce McNutt. MVS DASD survey: Results and trends. In *Proceedings of Computer Measurement Group (CMG) Conference*, pages 658–667, Nashville, TN, December 1995.

[Men95] Jai Menon. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, pages 167–178, Washington, DC, August 1995.

[Mes94] Mesquite Software Inc. *CSIM18 simulation engine (C++ version)*, 1994.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, , and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[MJLF84] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[MK91] Larry W. McVoy and Steve R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of Winter USENIX Conference*, pages 33–43, Dallas, TX, January 1991.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, pages 238–251, Saint-Malo, France, October 1997.

[MSNO74] Takashi Masuda, Hiroyuki Shiota, Kenichiro Noguchi, and Takashi Ohki. Optimization of program organization by cluster analysis. In *Proceedings of IFIP Congress*, pages 261–265, 1974.

[NWO88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the SPRITE network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[OD89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[ODH+85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, pages 15–24, Orcas Island, WA, December 1985.

[Pea88] J. Kent Peacock. The counterpoint fast file system. In *USENIX Conference Proceedings*, pages 243–249, Dallas, TX, Winter 1988.

[PFTV90] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1990.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 79–95, Copper Mountain, CO, December 1995.

[PK98] David A. Patterson and Kimberly. K. Keeton. Hardware technology trends and database opportunities. Keynote speech at *SIGMOD'98*, June 1998. Slides available at http://www.cs.berkeley.edu/~pattrsn/talks/sigmod98-keynote-color.ppt.

[PW00] Kihong Park and Walter Willinger, editors. *Self-Similar Network Traffic and Performance Evaluation*. John Wiley and Sons Inc., New York, 2000.

[PZ91] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. Technical Report CS-91-15, Department of Computer Science, Brown University, February 1991.

[RBH+95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta. The impact of architectural trends on operating system performance. In *Proceedings of 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 285–298, Copper Mountain, CO, December 1995.

[RBK92] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of file I/O traces in commercial computing environments. In *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 78–90, Newport, RI, June 1992.

[RGF98] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 62–73, New York, NY, August 1998.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The SimOS approach. *IEEE parallel and distributed technology: systems and applications*, 3(4):34–43, Winter 1995.

[RLA00] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of USENIX Annual Technical Conference*, pages 41–54, Berkeley, CA, June 2000.

[RN96] Bong K. Ryu and Mahesan Nandikesan. Real–time generation of fractal atm traffic: Model. Technical Report 440-96-06, Center for Telecommunicaions Research, Columbia University, New York, March 1996.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[Rus98] Mark Russinovich. Inside the cache manager. *Windows & .NET Magazine*, October 1998.

[RW91] Chris Ruemmler and John Wilkes. Disk shuffling. Technical Report HPL-91-156, HP Laboratories, October 1991.

[RW93] Chris Ruemmler and John Wilkes. UNIX disk access patterns. In *Proceedings of USENIX Winter Conference*, pages 405–420, San Diego, CA, January 1993.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of Winter USENIX Conference*, pages 313–324, Washington, DC, January 1990.

[SGM91] Carl Staelin and Hector Garcia-Molina. Smart filesystems. In *Proceedings of USENIX Winter Conference*, pages 45–52, January 1991.

[SHCG94] Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–235, August 1994.

[Smi78] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[Smi81] Alan Jay Smith. Input/output optimization and disk architectures: A survey. *Performance Evaluation*, 1(2):104–117, 1981.

[Smi85] Alan Jay Smith. Disk cache — miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, August 1985.

[Smi94] Alan Jay Smith. Trace driven simulation in research on computer architecture and operating systems. In *Proceedings of Conference on New Directions in Simulation for Manufacturing and Communications*, pages 43–49, Tokyo, Japan, August 1994.

[Sta95] Standard Performance Evaluation Corporation. SPEC CPU95 benchmarks, August 1995. http://www.spec.org/osg/cpu95.

[Sun99] Sun Microsystems. Jini^{TM} technology architectural overview, January 1999. http://www.sun.com/jini/whitepapers/architecture.html.

[Sym01] Symantec Corp. Norton Utilities 2002, 2001. http://www.symantec.com/nu/nu_9x.

[TG84] J. Z. Teng and R. A. Gumaer. Managing IBM Database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[TN92] Manolis M. Tsangaris and Jeffrey F. Naughton. On the performance of object clustering techniques. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 144–153, June 1992.

[Tra97] Transaction Processing Performance Council. *TPC Benchmark^{TM} D Standard Specification Revision 1.3.1*, December 1997.

[Tra99] Transaction Processing Performance Council. *TPC Benchmark^{TM} H Standard Specification Revision 1.1.0*, June 1999.

[UAS98] Mustafa Uysal, Anurag Acharya, and Joel Saltz. An evaluation of architectural alternatives for rapidly growing datasets: Active disks, clusters, SMPs. Technical Report TRCS98-27, Comp. Sc. Dept., University of California, Santa Barbara, November 1998.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[VC90] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software – Practice and Experience*, 20(3):225–242, March 1990.

[VJ98] Anujan Varma and Quinn Jacobson. Destage algorithms for disk arrays with nonvolatile caches. *IEEE Transactions on Computers*, 47(2):228–235, 1998.

[Vog99] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 93–109, December 1999.

[WA97] Richard Winter and Kathy Auerbach. Giants walk the earth: the 1997 VLDB survey. *Database Programming and Design*, 10(9), September 1997.

[WA98] Richard Winter and Kathy Auerbach. The big time: the 1998 VLDB survey. *Database Programming and Design*, 11(8), August 1998.

[WAP99] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file system for a programmable disk. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, pages 29–43, New Orleans, LA, February 1999.

[WGP94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 241–251, Nashville, TN, May 1994.

[Whi94] Albert E. Whipple II. Optimizing a magnetic disk by allocating files by the frequency a file is acessed/updated or by designating a file to a fixed location on a disk. U.S. Patent 5333311, July 1994. Filed Dec. 10, 1990.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, December 1993.

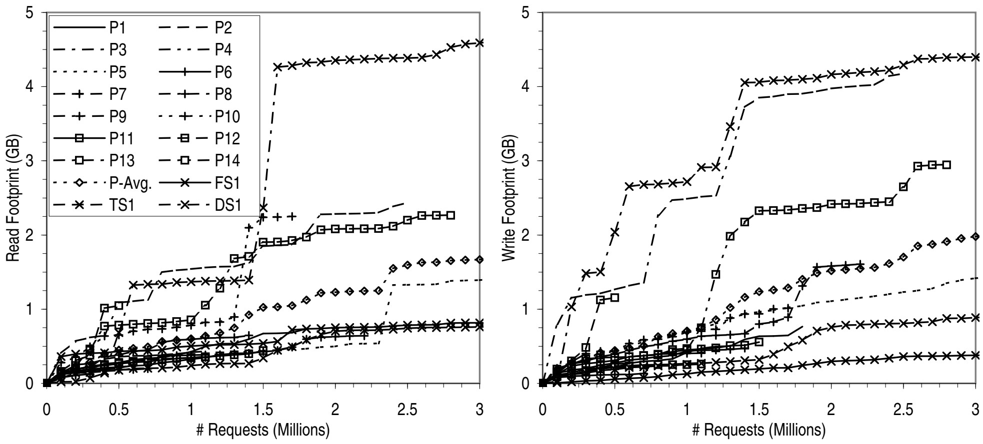
[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, February 1997.

[ZS97] Barbara Tockey Zivkov and Alan Jay Smith. Disk cache design and performance as evaluated in large timesharing and database systems. In *Proceedings of Computer Measurement Group (CMG) Conference*, pages 639–658, Orlando, FL, December 1997.

[ZS99] Min Zhou and Alan J. Smith. Analysis of personal computer workloads. In *Proceedings of International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 208–217, College Park, MD, oct 1999.

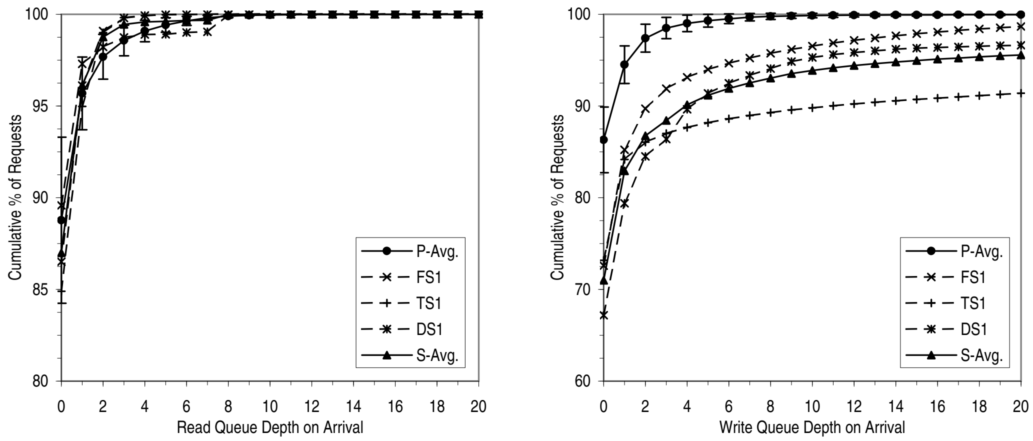
Appendix A

Additional Results for Chapter 2



(a) Read. (b) Write.

Figure A.1: Footprint vs. Number of References.



(a) Read Requests.

(b) Write Requests.

Figure A.2: Average Queue Depth on Arrival. Bars indicate standard deviation.

Average

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	Avg.[i]	FS1	TS1	DS1	Avg.[ii]
P1	1	0.0692	0.0459	0.0784	0.0773	0.027	0.097	0.0798	0.074	0.0393	0.0419	0.0329	0.0262	0.0368	0.0558	0.00962	0.0219	0.00331	-
P2	0.0692	1	0.0244	0.0533	0.0759	0.0251	0.0834	0.0423	0.159	0.0195	0.0285	0.0116	0.0283	0.0544	0.0519	0.00249	0.046	0.00056	-
P3	0.0459	0.0244	1	0.0115	0.0518	0.0263	0.0324	0.0272	0.0371	0.0428	0.0487	0.0132	0.0192	0.0447	0.0327	0.0285	0.0192	0.0175	-
P4	0.0784	0.0533	0.0115	1	0.0399	0.0262	0.0496	0.0484	0.0994	0.0278	0.0593	0.0109	0.0742	0.0446	0.0480	0.0247	0.0376	0.0144	-
P5	0.0773	0.0759	0.0518	0.0399	1	0.0342	0.0939	0.0512	0.0765	0.0281	0.0349	0.0118	0.048	0.04	0.0510	0.021	0.0263	0.00529	-
P6	0.027	0.0251	0.0263	0.0262	0.0342	1	0.0673	0.0333	0.0615	0.0538	0.0352	0.0299	0.0434	0.0528	0.0397	0.0197	0.0201	0.0331	-
P7	0.097	0.0834	0.0324	0.0496	0.0939	0.0673	1	0.105	0.0857	0.0475	0.0532	0.0317	0.0431	0.0722	0.0663	0.0303	0.0315	0.0776	-
P8	0.0798	0.0423	0.0272	0.0484	0.0512	0.0333	0.105	1	0.0509	0.038	0.0294	0.0431	0.0309	0.0362	0.0474	0.015	0.0248	0.0463	-
P9	0.074	0.159	0.0371	0.0994	0.0765	0.0615	0.0857	0.0509	1	0.0497	0.0731	0.0233	0.0366	0.0941	0.0708	0.0288	0.0576	0.0196	-
P10	0.0393	0.0195	0.0428	0.0278	0.0281	0.0538	0.0475	0.038	0.0497	1	0.0353	0.0143	0.0209	0.0429	0.0354	0.00701	0.0149	0.0134	-
P11	0.0419	0.0285	0.0487	0.0593	0.0349	0.0352	0.0532	0.0294	0.0731	0.0353	1	0.0077	0.0311	0.057	0.0412	0.0404	0.0456	0.0164	-
P12	0.0329	0.0116	0.0132	0.0109	0.0118	0.0299	0.0317	0.0431	0.0233	0.0143	0.0077	1	0.0112	0.0149	0.0197	0.000939	0.00489	0.00926	-
P13	0.0262	0.0283	0.0192	0.0742	0.048	0.0434	0.0431	0.0309	0.0366	0.0209	0.0311	0.0112	1	0.0625	0.0366	0.0368	0.0216	0.0246	-
P14	0.0368	0.0544	0.0447	0.0446	0.04	0.0528	0.0722	0.0362	0.0941	0.0429	0.057	0.0149	0.0625	1	0.0502	0.0129	0.0614	0.0775	-
Avg.[i]	0.0558	0.0519	0.0327	0.0480	0.0510	0.0397	0.0663	0.0474	0.0708	0.0354	0.0412	0.0197	0.0366	0.0502	0.0462	-	-	-	-
FS1	0.00962	0.00249	0.0285	0.0247	0.021	0.0197	0.0303	0.015	0.0288	0.00701	0.0404	0.000939	0.0368	0.0129	-	1	0.0242	0.0222	0.0232
TS1	0.0219	0.046	0.0192	0.0376	0.0263	0.0201	0.0315	0.0248	0.0576	0.0149	0.0456	0.00489	0.0216	0.0614	-	0.0242	1	0.042	0.0331
DS1	0.00331	0.00056	0.0175	0.0144	0.00529	0.0331	0.0776	0.0463	0.0196	0.0134	0.0164	0.00926	0.0246	0.0775	-	0.0222	0.042	1	0.0321
Avg.[ii]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.0232	0.0331	0.0321	0.0295

Average

[i] Average of cross correlation with other PC workloads, excluding self.
[ii] Average of cross correlation with other server workloads, excluding self.

Table A.1: Cross-Correlation of Per-Minute Volume of I/O Activity.

Average

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	Avg.[i]	FS1	TS1	DS1	Avg.[ii]
P1	1	0.16	0.108	0.129	0.228	0.118	0.233	0.22	0.16	0.106	0.123	0.08	0.0812	0.0832	0.141	0.0262	0.0505	0.0179	-
P2	0.16	1	0.0807	0.132	0.217	0.0925	0.183	0.126	0.35	0.0486	0.0898	0.0392	0.0858	0.125	0.133	0.00882	0.078	-0.00217	-
P3	0.108	0.0807	1	0.0311	0.141	0.111	0.101	0.0788	0.11	0.113	0.131	0.026	0.0619	0.0944	0.091	0.0619	0.0356	0.0549	-
P4	0.129	0.132	0.0311	1	0.12	0.072	0.115	0.107	0.194	0.061	0.113	0.035	0.176	0.0743	0.105	0.0481	0.0706	0.0321	-
P5	0.228	0.217	0.141	0.12	1	0.137	0.216	0.167	0.211	0.0959	0.107	0.0321	0.147	0.107	0.148	0.0579	0.0506	0.0397	-
P6	0.118	0.0925	0.111	0.072	0.137	1	0.187	0.114	0.183	0.141	0.129	0.0743	0.142	0.145	0.127	0.0524	0.0486	0.0885	-
P7	0.233	0.183	0.101	0.115	0.216	0.187	1	0.255	0.201	0.0971	0.119	0.0667	0.0945	0.136	0.154	0.0608	0.0569	0.141	-
P8	0.22	0.126	0.0788	0.107	0.167	0.114	0.255	1	0.115	0.0825	0.0906	0.0947	0.0832	0.0819	0.124	0.0338	0.0518	0.106	-
P9	0.16	0.35	0.11	0.194	0.211	0.183	0.201	0.115	1	0.108	0.143	0.0441	0.0962	0.173	0.161	0.059	0.108	0.0323	-
P10	0.106	0.0486	0.113	0.061	0.0959	0.141	0.0971	0.0825	0.108	1	0.0771	0.0344	0.0546	0.0914	0.085	0.0184	0.0392	0.0394	-
P11	0.123	0.0898	0.131	0.113	0.107	0.129	0.119	0.0906	0.143	0.0771	1	0.0193	0.108	0.108	0.104	0.0869	0.0993	0.0294	-
P12	0.08	0.0392	0.026	0.035	0.0321	0.0743	0.0667	0.0947	0.0441	0.0344	0.0193	1	0.0229	0.0248	0.046	0.000372	0.00633	0.03	-
P13	0.0812	0.0858	0.0619	0.176	0.147	0.142	0.0945	0.0832	0.0962	0.0546	0.108	0.0229	1	0.145	0.100	0.0815	0.0424	0.0513	-
P14	0.0832	0.125	0.0944	0.0743	0.107	0.145	0.136	0.0819	0.173	0.0914	0.108	0.0248	0.145	1	0.107	0.0267	0.106	0.119	-
Avg.[i]	0.141	0.133	0.091	0.105	0.148	0.127	0.154	0.124	0.161	0.085	0.104	0.046	0.100	0.107	0.116	-	-	-	-
FS1	0.0262	0.00882	0.0619	0.0481	0.0579	0.0524	0.0608	0.0338	0.059	0.0184	0.0869	0.000372	0.0815	0.0267	-	1	0.0462	0.0405	0.0434
TS1	0.0505	0.078	0.0356	0.0706	0.0506	0.0486	0.0569	0.0518	0.108	0.0392	0.0993	0.00633	0.0424	0.106	-	0.0462	1	0.04	0.0431
DS1	0.0179	-0.00217	0.0549	0.0321	0.0397	0.0885	0.141	0.106	0.0323	0.0394	0.0294	0.03	0.0513	0.119	-	0.0405	0.04	1	0.0403
Avg.[ii]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.0434	0.0431	0.0403	0.0422

Average

[i] Average of cross correlation with other PC workloads, excluding self.
[ii] Average of cross correlation with other server workloads, excluding self.

Table A.2: Cross-Correlation of Per-10-Minute Volume of I/O Activity.

Average

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	Avg.[i]	FS1	TS1	DS1	Avg.[ii]
P1	1	0.376	0.298	0.262	0.484	0.329	0.39	0.434	0.292	0.215	0.207	0.126	0.199	0.202	0.293	0.0413	0.147	0.071	-
P2	0.376	1	0.244	0.258	0.479	0.302	0.384	0.344	0.418	0.138	0.186	0.15	0.233	0.342	0.296	0.0272	0.145	0.058	-
P3	0.298	0.244	1	0.106	0.306	0.29	0.181	0.199	0.295	0.206	0.242	0.0813	0.151	0.207	0.216	0.142	0.121	0.062	-
P4	0.262	0.258	0.106	1	0.231	0.168	0.278	0.271	0.323	0.154	0.249	0.0963	0.347	0.165	0.224	0.0812	0.155	0.11	-
P5	0.484	0.479	0.306	0.231	1	0.38	0.372	0.344	0.384	0.227	0.223	0.0619	0.296	0.22	0.308	0.0903	0.131	0.082	-
P6	0.329	0.302	0.29	0.168	0.38	1	0.376	0.258	0.356	0.252	0.213	0.14	0.327	0.291	0.283	0.11	0.142	0.241	-
P7	0.39	0.384	0.181	0.278	0.372	0.376	1	0.454	0.361	0.177	0.171	0.156	0.187	0.241	0.287	0.121	0.119	0.188	-
P8	0.434	0.344	0.199	0.271	0.344	0.258	0.454	1	0.255	0.164	0.183	0.157	0.193	0.187	0.265	0.0764	0.129	0.267	-
P9	0.292	0.418	0.295	0.323	0.384	0.356	0.361	0.255	1	0.263	0.216	0.126	0.197	0.331	0.294	0.088	0.169	0.0909	-
P10	0.215	0.138	0.206	0.154	0.227	0.252	0.177	0.164	0.263	1	0.15	0.0763	0.136	0.209	0.182	0.0247	0.144	0.107	-
P11	0.207	0.186	0.242	0.249	0.223	0.213	0.171	0.183	0.216	0.15	1	0.0297	0.19	0.244	0.193	0.145	0.187	0.0627	-
P12	0.126	0.15	0.0813	0.0963	0.0619	0.14	0.156	0.157	0.126	0.0763	0.0297	1	0.0355	0.0485	0.099	-0.00785	0.0473	0.06	-
P13	0.199	0.233	0.151	0.347	0.296	0.327	0.187	0.193	0.197	0.136	0.19	0.0355	1	0.298	0.215	0.16	0.131	0.12	-
P14	0.202	0.342	0.207	0.165	0.22	0.291	0.241	0.187	0.331	0.209	0.244	0.0485	0.298	1	0.230	0.0355	0.243	0.161	-
Avg.[i]	0.293	0.296	0.216	0.224	0.308	0.283	0.287	0.265	0.294	0.182	0.193	0.099	0.215	0.230	0.242	-	-	-	-
FS1	0.0413	0.0272	0.142	0.0812	0.0903	0.11	0.121	0.0764	0.088	0.0247	0.145	-0.00785	0.16	0.0355	-	1	0.076	0.0832	0.0796
TS1	0.147	0.145	0.121	0.155	0.131	0.142	0.119	0.129	0.169	0.144	0.187	0.0473	0.131	0.243	-	0.076	1	0.0422	0.0591
DS1	0.071	0.058	0.062	0.11	0.082	0.241	0.188	0.267	0.0909	0.107	0.0627	0.06	0.12	0.161	-	0.0832	0.0422	1	0.0627
Avg.[ii]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0.0796	0.0591	0.0627	0.0671

Average

[i] Average of cross correlation with other PC workloads, excluding self.
[ii] Average of cross correlation with other server workloads, excluding self.

Table A.3: Cross-Correlation of Hourly Volume of I/O Activity.

Average

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	Avg.[i]	FS1	TS1	DS1	Avg.[ii]
P1	1	0.579	0.241	0.488	0.659	0.647	0.492	0.684	0.544	0.38	0.116	0.205	0.315	0.18	0.425	0.065	0.341	-0.0254	-
P2	0.579	1	0.115	0.319	0.631	0.513	0.628	0.736	0.565	0.317	0.243	0.253	0.537	0.464	0.454	-0.118	0.445	0.184	-
P3	0.241	0.115	1	0.136	0.312	0.26	-0.0361	0.201	0.37	0.354	0.249	-0.0782	0.171	0.12	0.186	0.288	0.401	0.176	-
P4	0.488	0.319	0.136	1	0.323	0.277	0.219	0.516	0.502	0.35	0.434	0.224	0.469	0.203	0.343	0.0703	0.552	-0.466	-
P5	0.659	0.631	0.312	0.323	1	0.592	0.507	0.618	0.566	0.452	0.0851	-0.0555	0.382	0.17	0.403	0.135	0.344	-0.0191	-
P6	0.647	0.513	0.26	0.277	0.592	1	0.569	0.406	0.619	0.426	0.141	0.25	0.591	0.321	0.432	0.0314	0.476	0.414	-
P7	0.492	0.628	-0.0361	0.219	0.507	0.569	1	0.597	0.563	0.162	0.0476	0.373	0.455	0.324	0.377	0.0792	0.204	0.278	-
P8	0.684	0.736	0.201	0.516	0.618	0.406	0.597	1	0.542	0.224	0.132	0.266	0.369	0.22	0.424	-0.0358	0.333	0.23	-
P9	0.544	0.565	0.37	0.502	0.566	0.619	0.563	0.542	1	0.728	0.0909	0.352	0.404	0.376	0.479	0.175	0.629	-0.0133	-
P10	0.38	0.317	0.354	0.35	0.452	0.426	0.162	0.224	0.728	1	0.116	0.0664	0.431	0.584	0.353	0.062	0.472	-0.0131	-
P11	0.116	0.243	0.249	0.434	0.0851	0.141	0.0476	0.132	0.0909	0.116	1	0.0112	0.272	0.387	0.179	0.163	0.518	0.387	-
P12	0.205	0.253	-0.0782	0.224	-0.0555	0.25	0.373	0.266	0.352	0.0664	0.0112	1	0.23	0.11	0.170	-0.163	0.0531	-0.201	-
P13	0.315	0.537	0.171	0.469	0.382	0.591	0.455	0.369	0.404	0.431	0.272	0.23	1	0.586	0.401	0.0261	0.59	0.133	-
P14	0.18	0.464	0.12	0.203	0.17	0.321	0.324	0.22	0.376	0.584	0.387	0.11	0.586	1	0.311	-0.297	0.523	0.13	-
Avg.[i]	0.425	0.454	0.186	0.343	0.403	0.432	0.377	0.424	0.479	0.353	0.179	0.170	0.401	0.311	0.353	-	-	-	-
FS1	0.065	-0.118	0.288	0.0703	0.135	0.0314	0.0792	-0.0358	0.175	0.062	0.163	-0.163	0.0261	-0.297	-	1	0.133	-0.343	-0.105
TS1	0.341	0.445	0.401	0.552	0.344	0.476	0.204	0.333	0.629	0.472	0.518	0.0531	0.59	0.523	-	0.133	1	0.18	0.157
DS1	-0.0254	0.184	0.176	-0.466	-0.0191	0.414	0.278	0.23	-0.0133	-0.0131	0.387	-0.201	0.133	0.13	-	-0.343	0.18	1	-0.0815
Avg.[ii]	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-0.105	0.157	-0.0815	-0.0100

Average

[i] Average of cross correlation with other PC workloads, excluding self.
[ii] Average of cross correlation with other server workloads, excluding self.

Table A.4: Cross-Correlation of Daily Volume of I/O Activity.

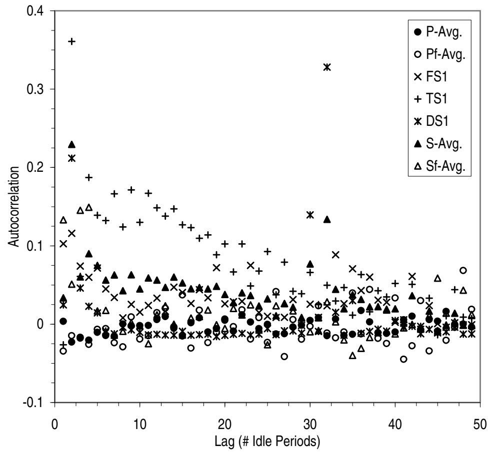
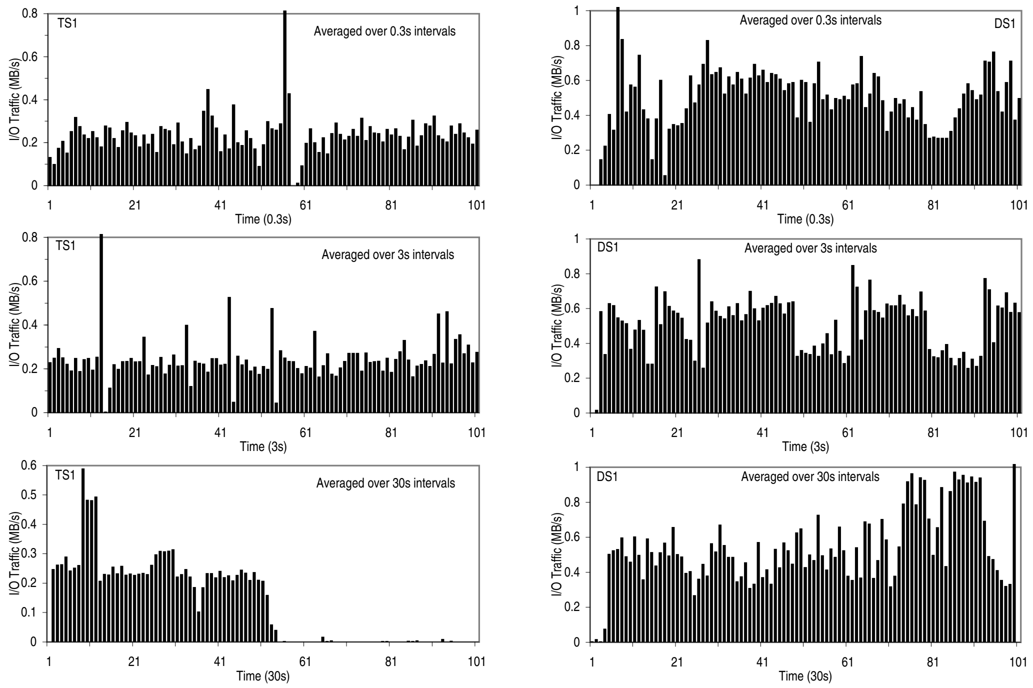


Figure A.3: Autocorrelation of the Sequence of Idle Period Duration.



(a) TS1

(b) DS1

Figure A.4: I/O Traffic at Different Time Scales during the High-Traffic Period (One-hour period that contains more I/O traffic than 95% of other one-hour periods).

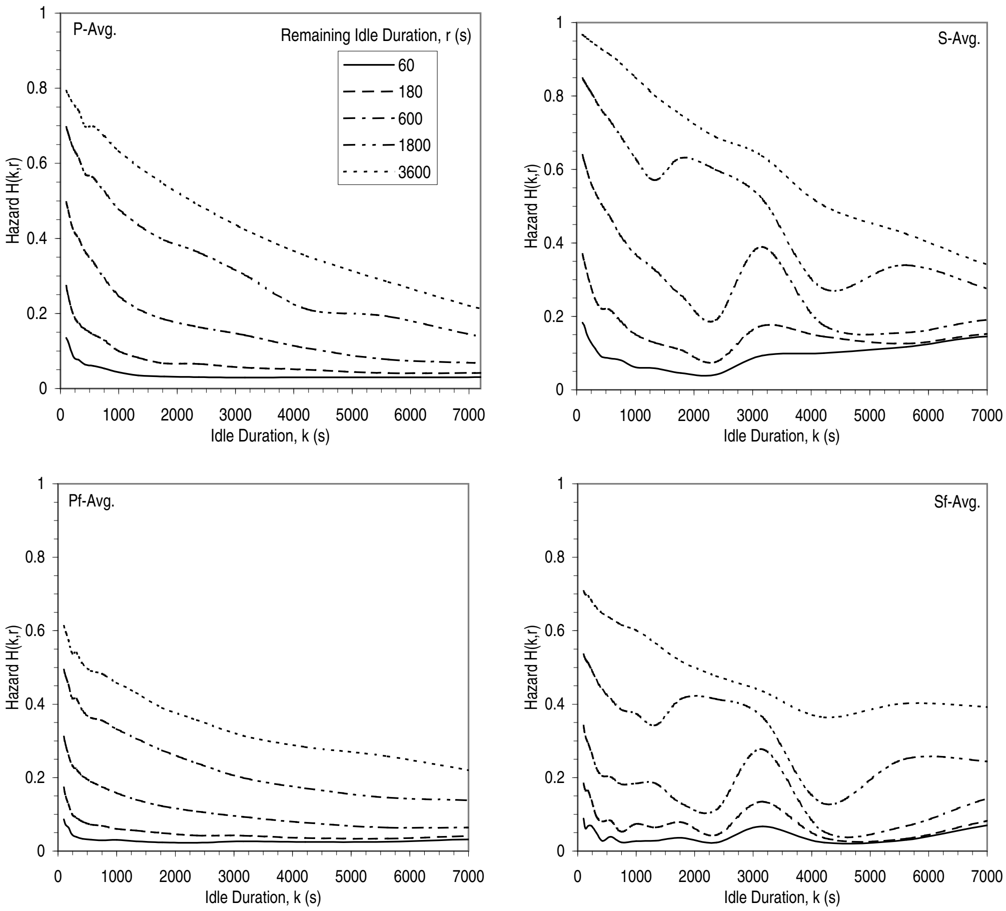


Figure A.5: Hazard Rate for the Distribution of Idle Period Duration.

Appendix B

Details on Self-Similarity

B.1 Estimating the Degree of Self-Similarity

The degree of self-similarity is expressed using a single parameter, the Hurst parameter H. For a self-similar series, $1/2 < H < 1$, and as $H \to 1$, the degree of self-similarity increases. For smooth Poisson traffic, H is 1/2. Mathematically, self-similarity is manifested in several equivalent ways and different methods that examine specific indications of self-similarity are used to estimate the Hurst parameter. In this paper, we focus on the R/S method and the variance-time plot. Newer inference methods that are more sensitive to different types of scaling phenomena (*e.g.,* [AV98]) have been developed but are beyond the scope of this thesis.

B.1.1 The R/S Method

One of the manifestations of the self-similar property is that the autocorrelations of the process decay hyperbolically rather than exponentially. This behavior is known as long-range dependence and it provides an explanation for an empirical law known as the Hurst effect [LTWW94].

The R/S or rescaled adjusted range statistic for a set of observations $X_k : k = 1, 2, ..., n$ having mean $\overline{X}(n)$ and sample variance $S^2(n)$ is defined by

$$\frac{R(n)}{S(n)} = \frac{1}{S(n)}[max(0, W_1, W_2, ..., W_n) - min(0, W_1, W_2, ..., W_n)] \qquad \text{(B.1)}$$

where

$$W_k = (X_1 + X_2 + ... + X_k) - k\overline{X}(n),\ k \geq 1.$$

It turns out that

$$E\left[\frac{R(n)}{S(n)}\right] \sim cn^H \qquad \text{(B.2)}$$

where $H = 0.5$ for short-range dependent processes and $0.5 < H < 1$ for long-range dependent processes. This difference between short and long-range dependent processes is known as the Hurst effect and forms the basis for the R/S method of inferring the Hurst parameter.

Taking logarithm on both sides of Equation B.2,

$$log\left(E\left[\frac{R(n)}{S(n)}\right]\right) \sim Hlog(n) + log(c) \tag{B.3}$$

Therefore, we can estimate H by plotting $log(E[R(n)/S(n)])$ versus $log(n)$ for different values of n. In practice, we divide a set of N observations into K disjoint subsets each of length N/K and compute $log(E[R(n)/S(n)])$ for each of these subsets using logarithmically spaced values of n. The resulting plot of $log(E[R(n)/S(n)])$ versus $log(n)$ is commonly referred to as a pox plot. For a long-range dependent time series, the pox plot should fluctuate in a straight street of slope H, $0.5 < H < 1$ [Ber94].

In Figure B.1, we present the pox plots for our various workloads for the high-traffic period. Observe that the pox plots for all the workloads appear to fluctuate around straight streets with slope ranging from 0.6 to almost 0.9. In other words, *all the workloads exhibit long-range dependence and self-similarity in their I/O traffic patterns.* In Figure B.2, we present the corresponding pox plots for the filtered traces. The same behavior is observed.

B.1.2 Variance-Time Plot

Suppose X is an incremental process indexed by i. Another manifestation of self-similarity is that the variance of the aggregated process $X^{(m)}$ decrease more slowly than the reciprocal of m, where

$$X^{(m)}(k) = (1/m) \sum_{i=(k-1)m+1}^{km} X(i),\ k = 1, 2, ...$$

More formally,

$$Var(X^{(m)}) \sim cm^{-\beta},\ 0 < \beta < 1. \tag{B.4}$$

Taking logarithm on both sides,

$$log(Var(X^{(m)})) \sim log(c) - \beta log(m). \tag{B.5}$$

Thus for a self-similar process, the variance-time plot, *i.e.,* the plot of $log(Var(X^{(m)}))$ against $log(m)$, should be a straight line with a slope $(-\beta)$ between -1 and 0. The degree of self-similarity is given by $H = 1 - \beta/2$.

The variance-time plots for our various workloads are presented in Figures B.3 and B.4. Observe that for the high-traffic period, the variance-time plots for all the workloads are

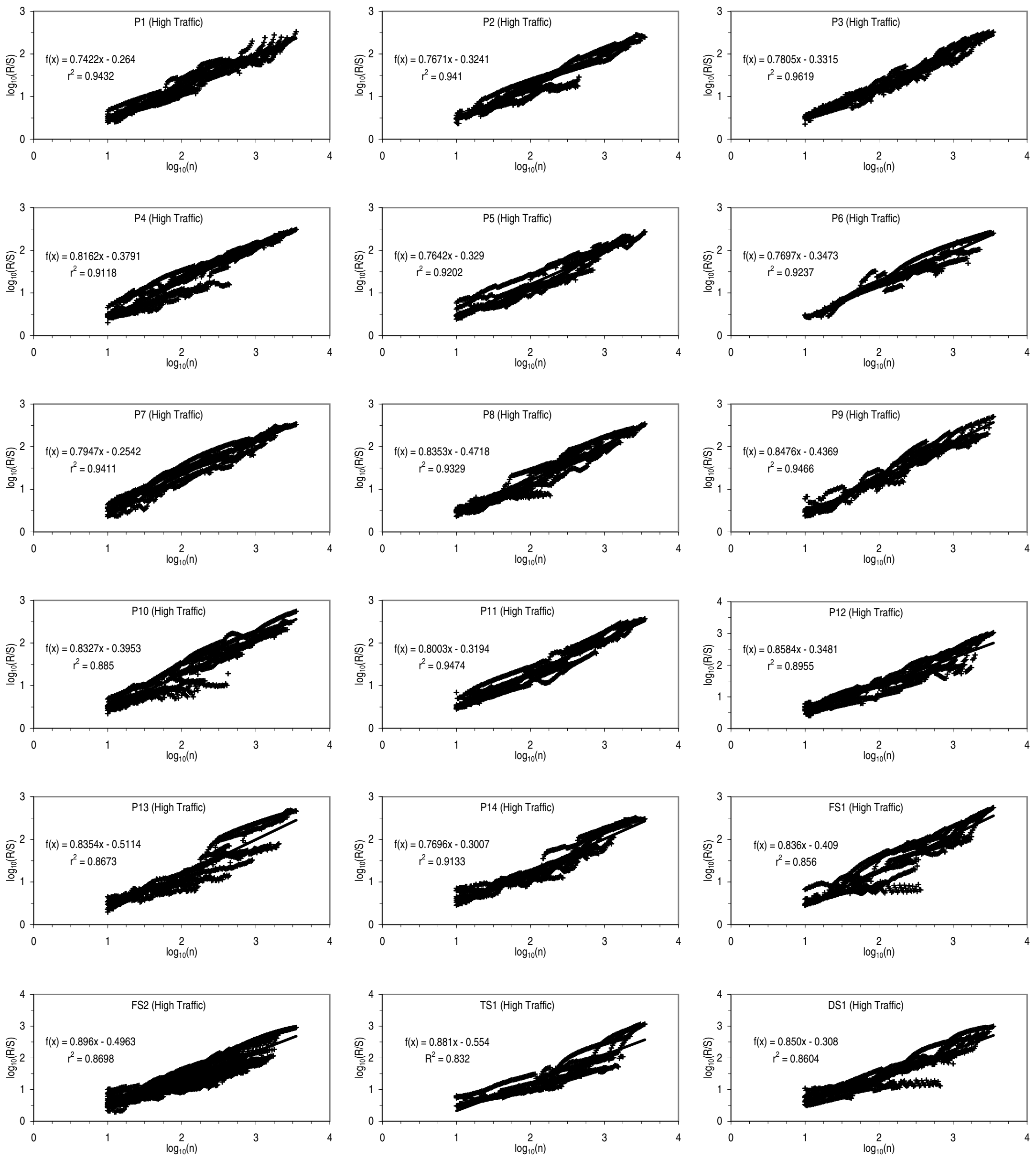


Figure B.1: Pox Plots to Detect Self-Similarity.

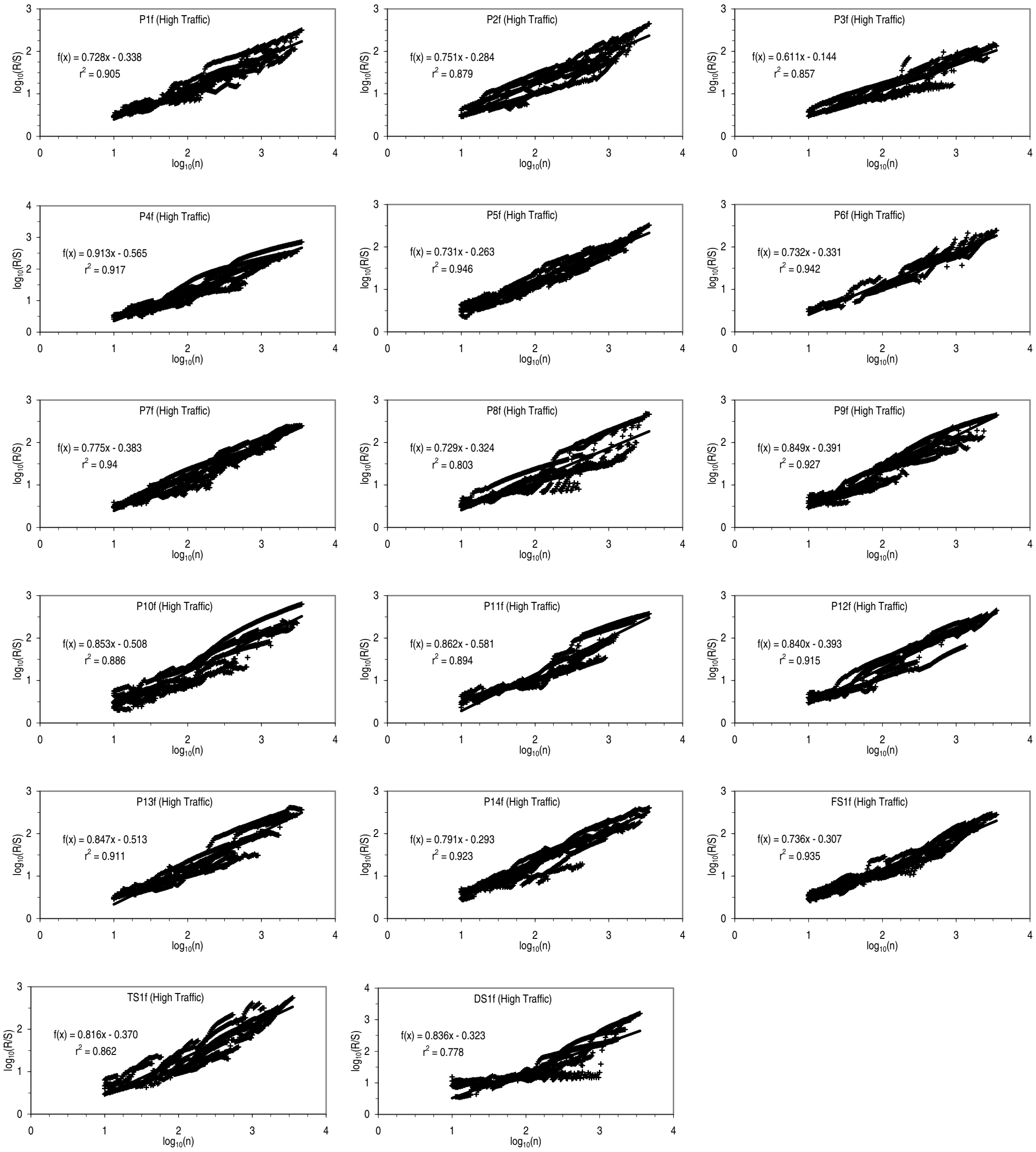
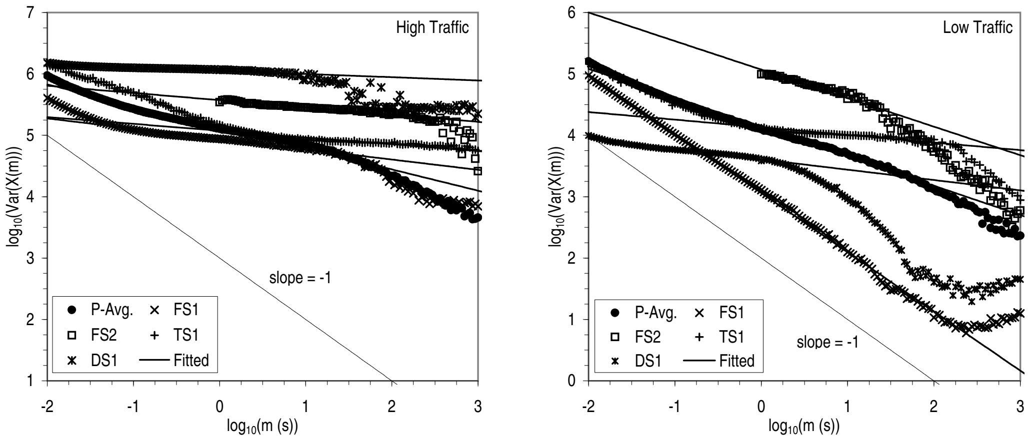


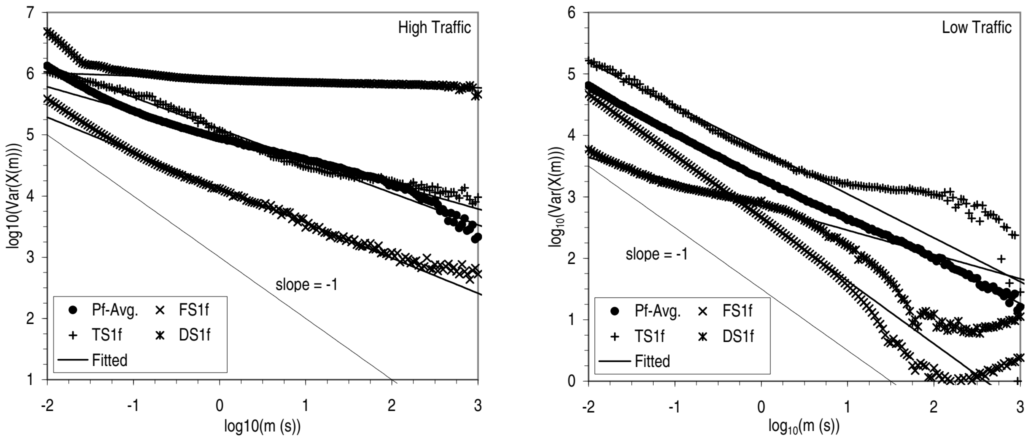
Figure B.2: Pox Plots to Detect Self-Similarity (Filtered Traces).



(a) High-Traffic Period.

(b) Low-Traffic Period.

Figure B.3: Variance-Time Plots to Detect Self-Similarity.



(a) High-Traffic Period.

(b) Low-Traffic Period.

Figure B.4: Variance-Time Plots to Detect Self-Similarity (Filtered Traces).

very linear with slopes that are more gradual than -1. This indicates that the I/O traffic for the workloads is self-similar in nature. Notice though that *the self-similarity does not span all time scales but appears to break down beginning just beyond 10s for the database server. In other words, for time scales ranging from tens of milliseconds to tens and sometimes even hundreds of seconds, the I/O traffic is well-represented by a self-similar process but not beyond that.* Interestingly, the filtered traces appear to be self-similar to larger time scales although some of them have a steeper slope, meaning that they are less self-similar.

For the low-traffic period, all the plots again have linear segments with slope of less than -1 but these segments are shorter than in the high-traffic case, particularly in the case of the database server. In addition, the slope of the linear regions is noticeably steeper than for the high-traffic period. This means that *I/O traffic during the low-traffic period is self-similar but less so and over a smaller range of time scales than during the high-traffic period.* As discussed in the main text, the self-similarity could be caused by the superposition of I/O generated by different processes in the system where each process behaves as an independent I/O source with heavy-tailed on periods. During the low-traffic period, we would expect that there are fewer processes running in the system and therefore fewer independent sources of I/O so that the aggregated traffic is less self-similar. This is in line with observations in [GS99].

Table B.1 summarizes the Hurst parameter values that we obtained using the R/S method and the variance-time plot. These two methods provide independent estimates of the degree of self-similarity and discrepancies between their results can be expected. In view of this, the figures we obtained are reasonably consistent, which adds confidence to our analysis and results.

B.2 Generating Self-Similar I/O Traffic

There are several ways to generate self-similar traffic but models such as those based on F-ARIMA and Fractional Guassian Noise processes are generally computationally expensive. An alternative traffic generator based on the superposition of independent and identical fractal renewal processes is attractive because it has a physical correspondence to the superposition of I/O traffic generated by different processes, and is relatively easy to construct. The Superposition of Fractal Renewal Processes model is completely characterized by M, the number of fractal renewal processes, and $p(\tau)$, the inter-arrival probability density function. A convenient probability density function is the following where the parameter A serves as a threshold between exponential behavior and power-law behavior:

$$p(\tau) = \begin{cases} \frac{\gamma}{A} e^{\frac{-\gamma\tau}{A}}, & \tau \leq A, \\ \gamma e^{-\gamma} A\gamma\tau^{-(\gamma+1)}, & \tau > A \end{cases} \tag{B.6}$$

The interested reader is referred to [RN96] for more details about the model.

High Traffic		P-Avg.	Pf-Avg.	FS1	FS2	TS1	DS1	S-Avg.	Sf-Avg.
Var.-Time	Slope of Fitted Line	-0.35	-0.40	-0.17	-0.12	-0.11	-0.053	-0.11	-0.38
	Hurst Parameter	0.83	0.80	0.92	0.94	0.94	0.97	0.94	0.81
Pox	Slope of Fitted Line	0.80	0.79	0.84	0.90	0.88	0.85	0.86	0.80
	Hurst Parameter	0.80	0.79	0.84	0.90	0.88	0.85	0.86	0.80

(a) High Traffic.

High Read Traffic		P-Avg.	Pf-Avg.	FS1	FS2	TS1	DS1	S-Avg.	Sf-Avg.
Var.-Time	Slope of Fitted Line	-0.26	-0.29	-0.20	-0.10	-0.13	-0.10	-0.14	-0.13
	Hurst Parameter	0.87	0.85	0.90	0.95	0.94	0.95	0.93	0.93
Pox	Slope of Fitted Line	0.77	0.74	0.85	0.92	0.79	0.76	0.80	0.77
	Hurst Parameter	0.77	0.74	0.85	0.92	0.79	0.76	0.80	0.77

(b) High Read Traffic.

High Write Traffic		P-Avg.	Pf-Avg.	FS1	FS2	TS1	DS1	S-Avg.	Sf-Avg.
Var.-Time	Slope of Fitted Line	-0.50	-0.55	-0.29	-0.12	-0.28	-0.068	-0.21	-0.49
	Hurst Parameter	0.75	0.73	0.85	0.94	0.86	0.97	0.89	0.76
Pox	Slope of Fitted Line	0.79	0.78	0.81	0.76	0.88	0.82	0.83	0.79
	Hurst Parameter	0.79	0.78	0.81	0.76	0.88	0.82	0.83	0.79

(c) High Write Traffic.

Table B.1: Degree of Self-Similarity.

B.2.1 The Inputs

The inputs to the traffic generator are:

1. H, the Hurst parameter which measures the degree of self-similarity [Ber94].

2. μ, the average number of arrivals during intervals of duration T_s.

3. σ^2, the variance in the number of arrivals during intervals of duration T_s.

B.2.2 Model Setup

The three inputs described above were chosen to be relatively easy to measure and understand. Before we begin to generate the traffic, however, we need to convert the inputs into a more convenient form:

1. Calculate

$$\alpha = 2H - 1 \tag{B.7}$$

2. Calculate

$$\gamma = 2 - \alpha \tag{B.8}$$

3. Calculate

$$\lambda = \frac{\mu}{T_s} \tag{B.9}$$

4. Calculate

$$T_o = \frac{T_s}{(\frac{\sigma^2}{\lambda T_s})^{\frac{1}{\alpha}} - 1} \tag{B.10}$$

5. Calculate

$$A = \left[\frac{T_o^{\alpha} 2\gamma^2(\gamma - 1)e^{\gamma}}{(2-\gamma)(3-\gamma)[1 + (\gamma - 1)e^{\gamma}]^2} \right]^{\frac{1}{\alpha}} \tag{B.11}$$

6. Calculate

$$M = \left\lfloor \frac{A\lambda}{\gamma} \left[1 + \frac{1}{(\gamma - 1)e^{\gamma}} \right] \right\rfloor \tag{B.12}$$

B.2.3 The Algorithm

Let $T_i^{(j)}$ denote the ith inter-arrival time for process j. The following algorithm calculates the $T_i^{(j)}$ by spawning M independent threads. This multi-threaded approach is useful when actual I/Os are to be issued. For pure simulations or where I/O calls return immediately after they have been issued, a single-threaded version can be easily constructed.

1. Spawn M threads
2. For each thread
3. Generate a random variable U uniformly distributed in [0,1)
4. Calculate

$$V = \frac{1 + (\gamma - 1)e^{\gamma}}{\gamma} U \tag{B.13}$$

5. Calculate

$$\tau_o^{(j)} = \begin{cases} -\gamma^{-1} A \ln[U^{\frac{\gamma V - 1}{\gamma V - U}}], & V \geq 1, \\ A V^{\frac{1}{1-\gamma}}, & V < 1 \end{cases} \tag{B.14}$$

6. Repeat
7. Generate a random variable U uniformly distributed in [0,1)
8. Calculate

$$\tau_i^{(j)} = \begin{cases} -\frac{1}{\gamma} A \ln[U], & U \geq e^{-\gamma}, \\ \frac{1}{e} A U^{\frac{-1}{\gamma}}, & U < e^{-\gamma} \end{cases} \tag{B.15}$$

9. Until enough arrivals are generated

Appendix C

Additional Results for Chapter 3

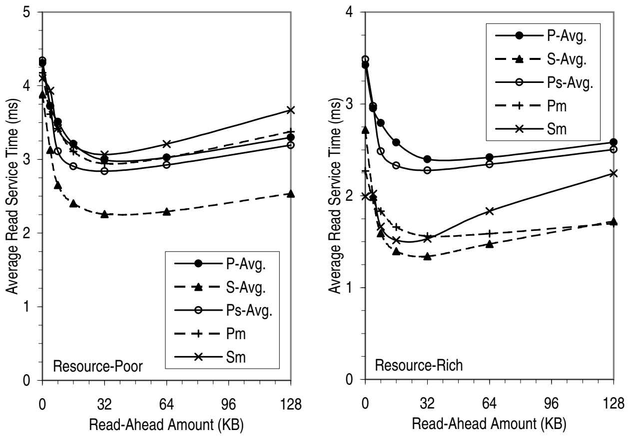


Figure C.1: Effect of Read-Ahead on Average Read Service Time.

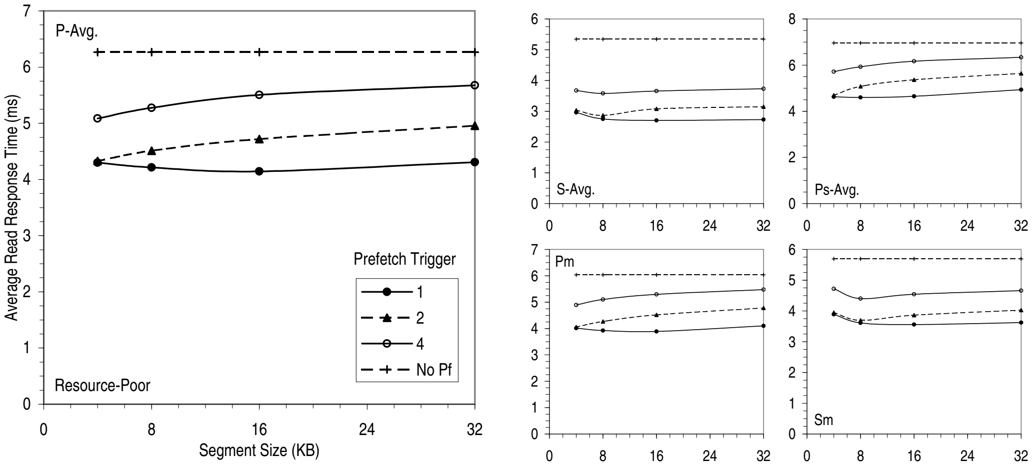


Figure C.2: Response Time with Conditional Prefetch (Resource-Poor).

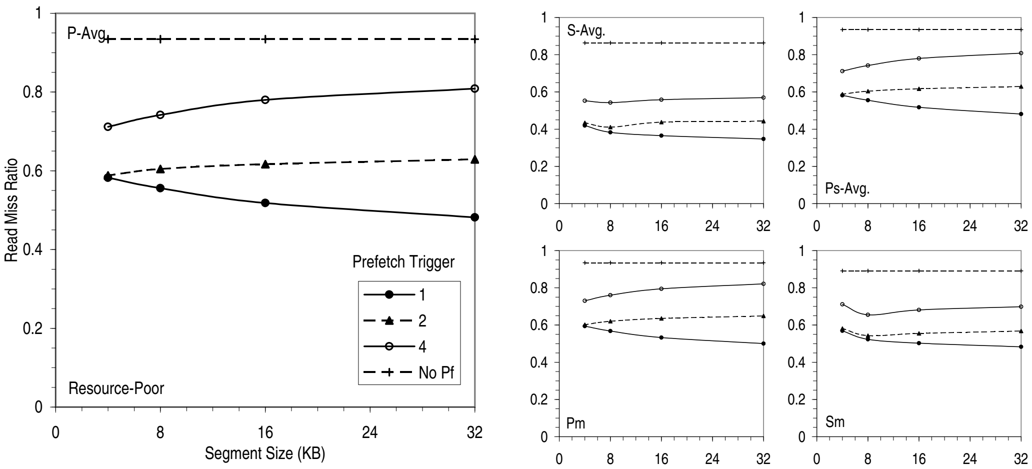


Figure C.3: Read Miss Ratio with Conditional Prefetch (Resource-Poor).

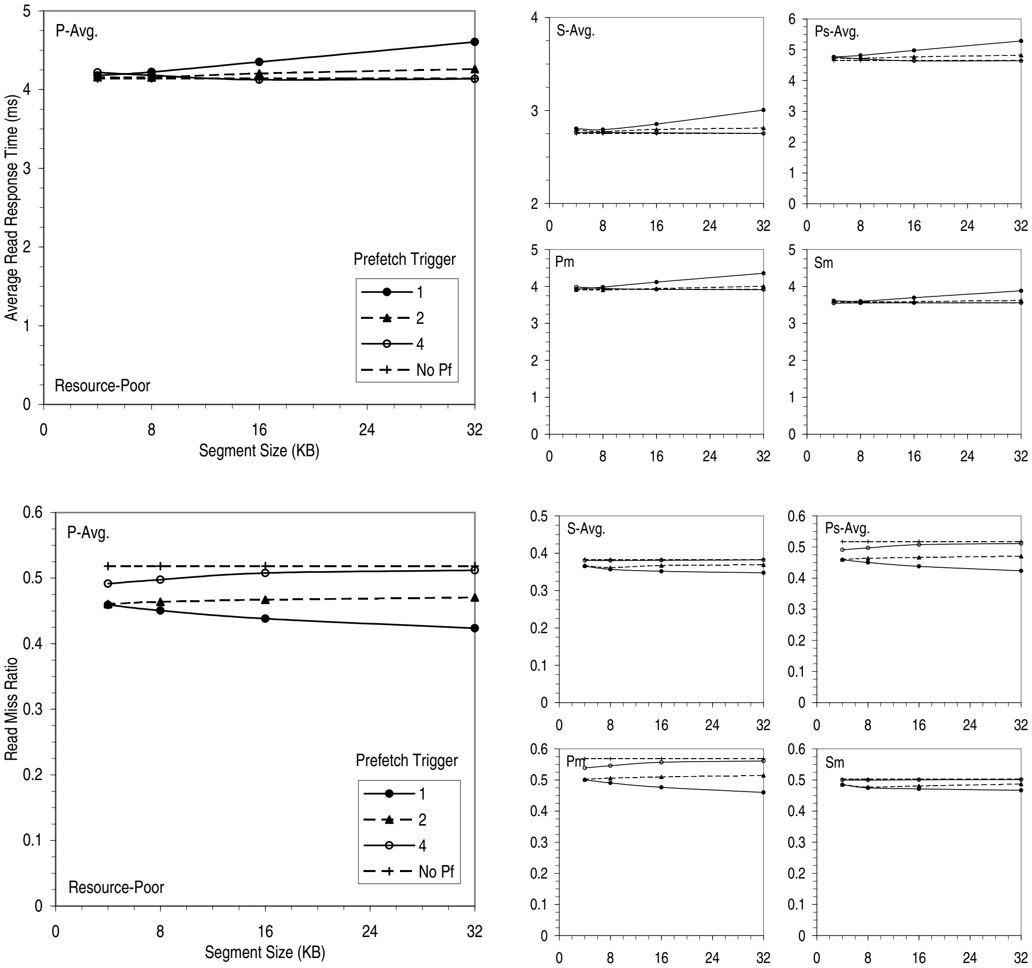


Figure C.4: Additional Effect of Backward Conditional Prefetch (Resource-Poor).

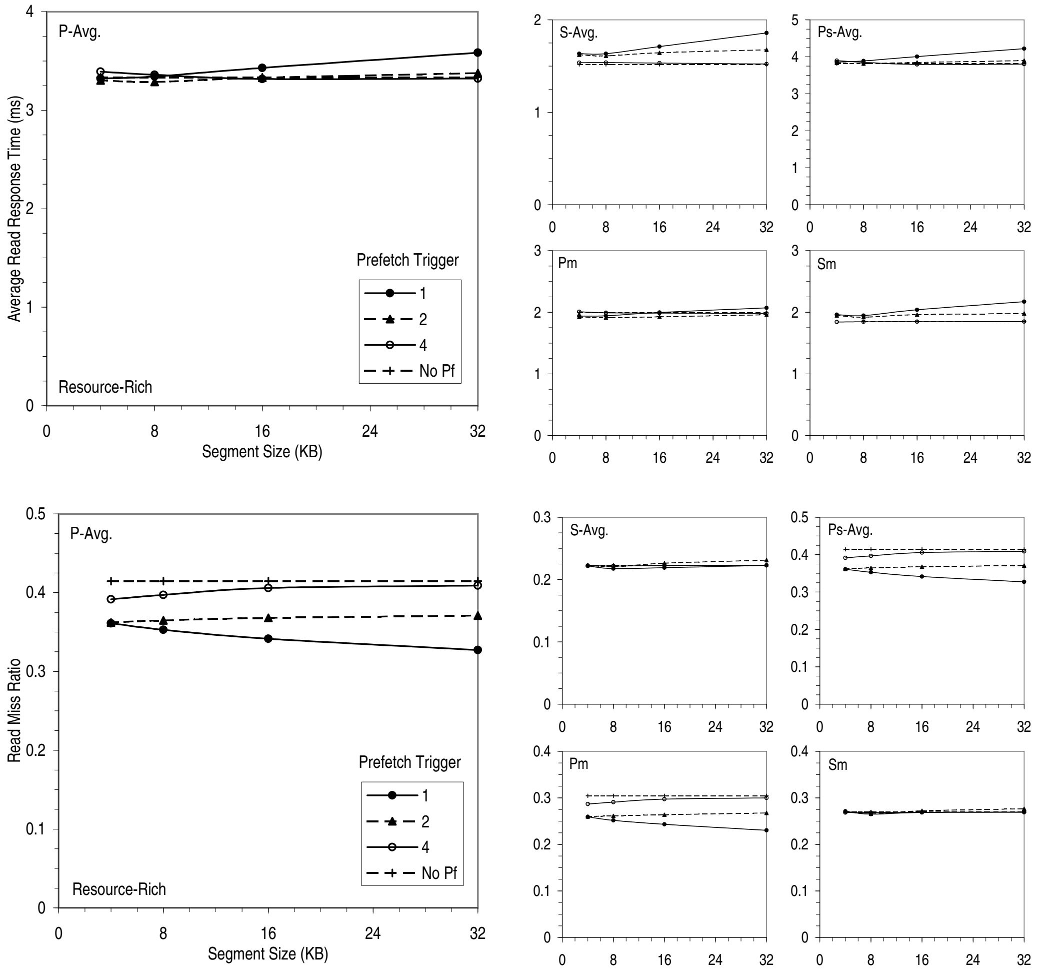


Figure C.5: Additional Effect of Backward Conditional Prefetch (Resource-Rich).

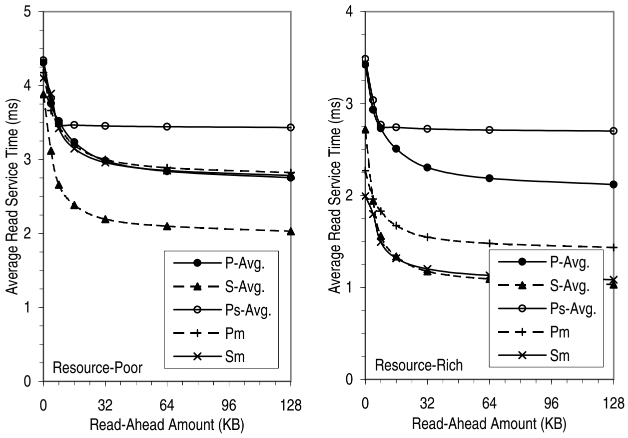


Figure C.6: Effect of Preemptible Read-Ahead on Average Read Service Time.

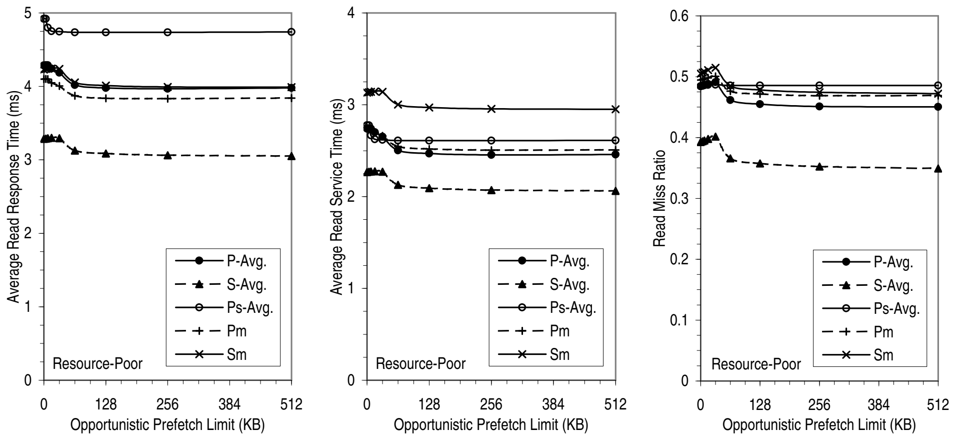


Figure C.7: Performance of Large Fetch Unit with Preemptible Read-Ahead (Resource-Poor).

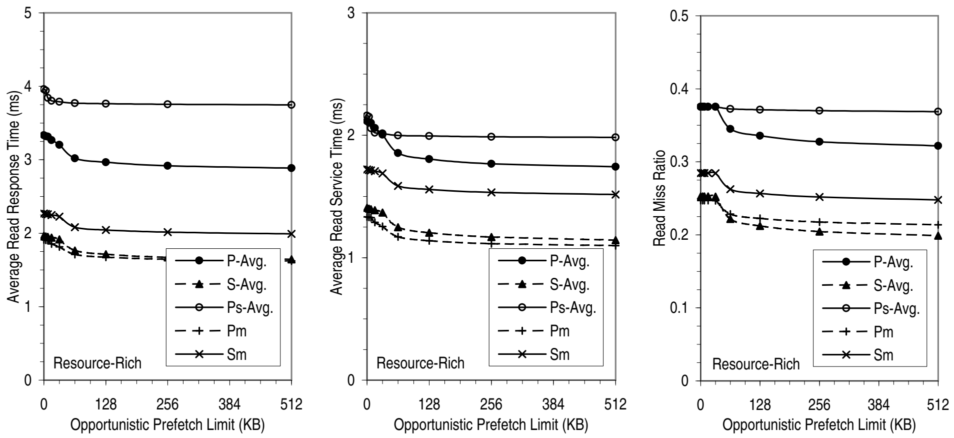


Figure C.8: Performance of Large Fetch Unit with Preemptible Read-Ahead (Resource-Rich).

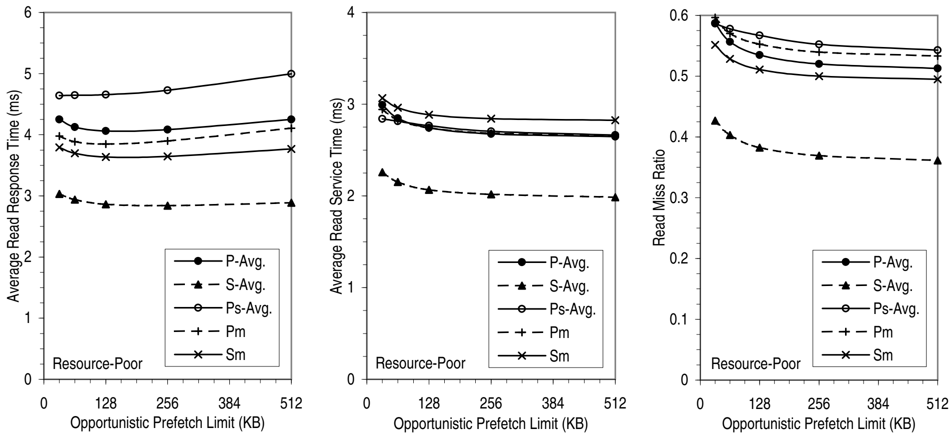


Figure C.9: Performance of Read-Ahead with Preemptible Read-Ahead (Resource-Poor).

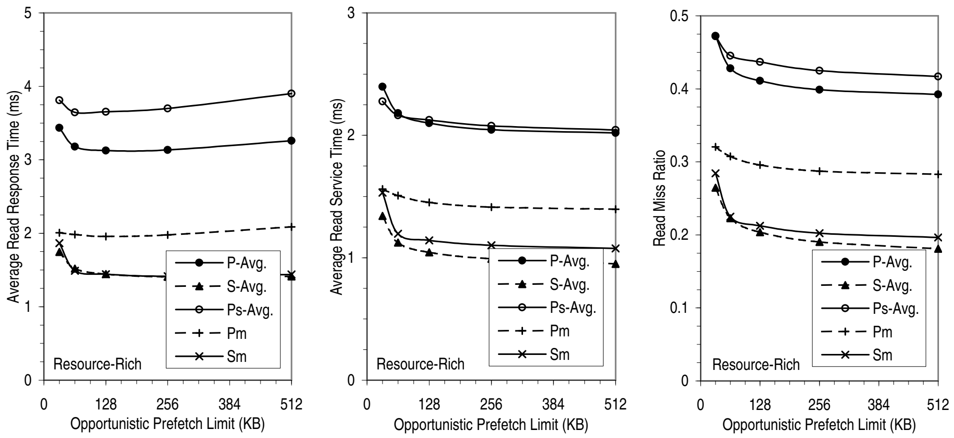


Figure C.10: Performance of Read-Ahead with Preemptible Read-Ahead (Resource-Rich).

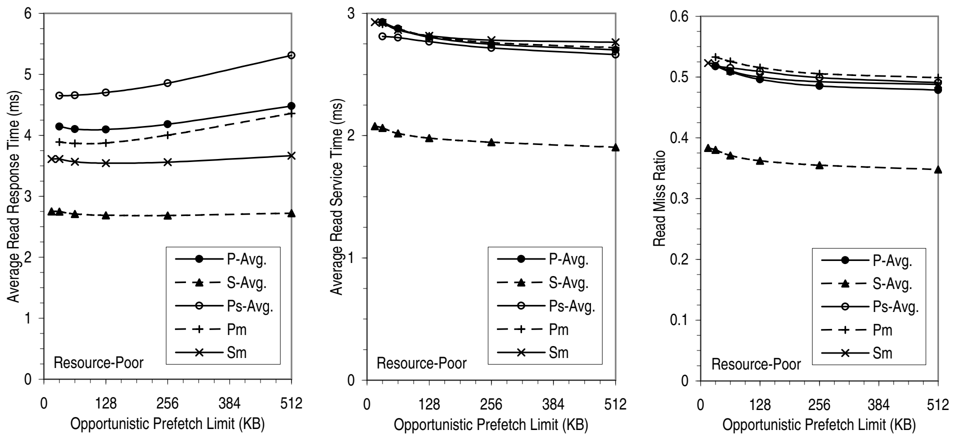


Figure C.11: Performance of Conditional Sequential Prefetch with Preemptible Read-Ahead (Resource-Poor).

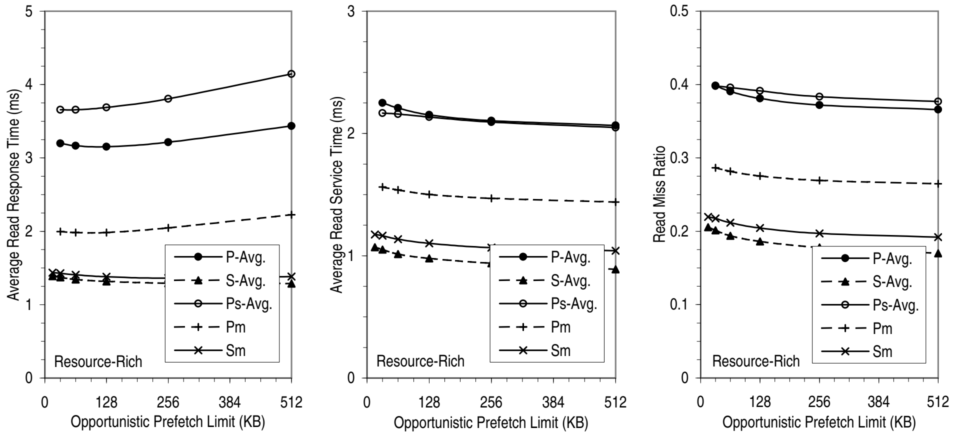


Figure C.12: Performance of Conditional Sequential Prefetch with Preemptible Read-Ahead (Resource-Rich).

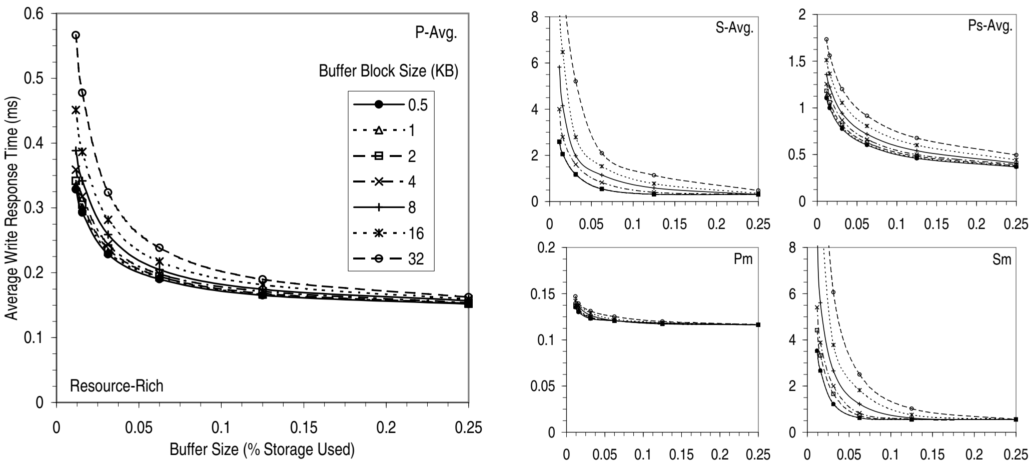
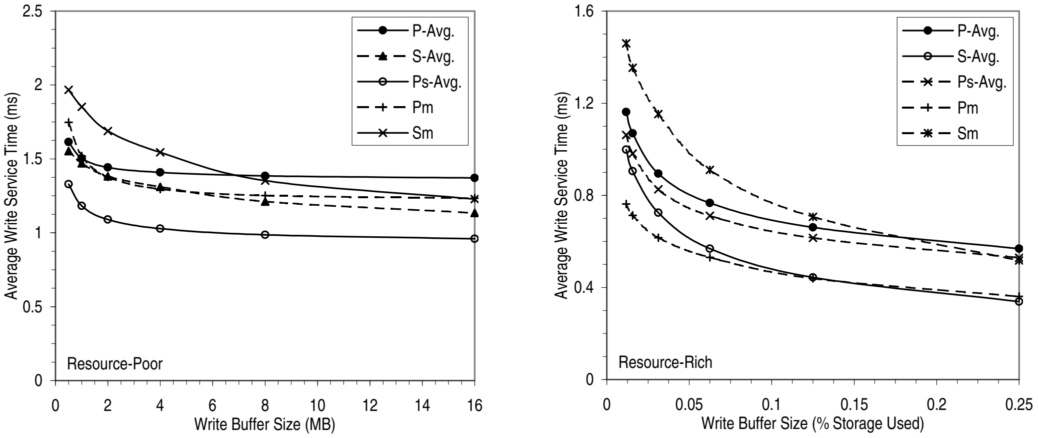


Figure C.13: Sensitivity to Buffer Block Size.



(a) Resource-Poor. (b) Resource-Rich.

Figure C.14: Improvement in Average Write Service Time from Eliminating Repeated Writes.

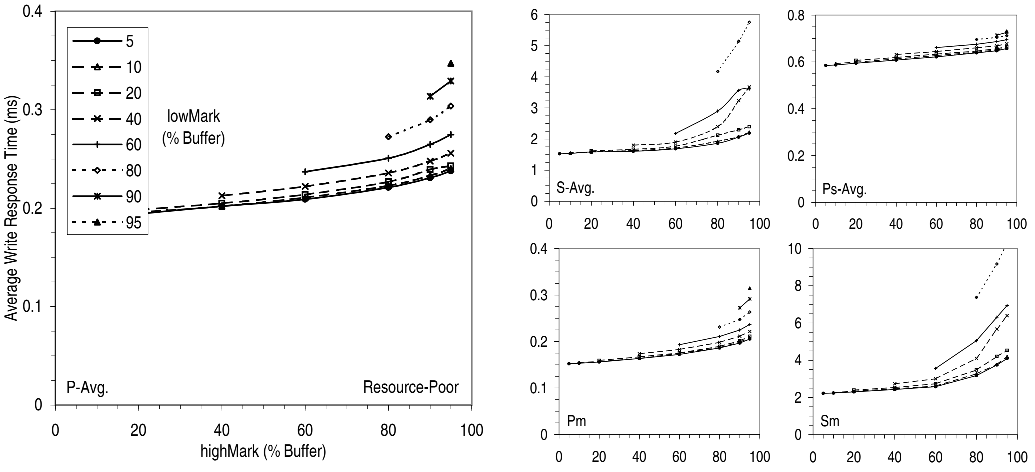


Figure C.15: Effect of $lowMark$ and $highMark$ on Average Write Response Time (Resource-Poor).

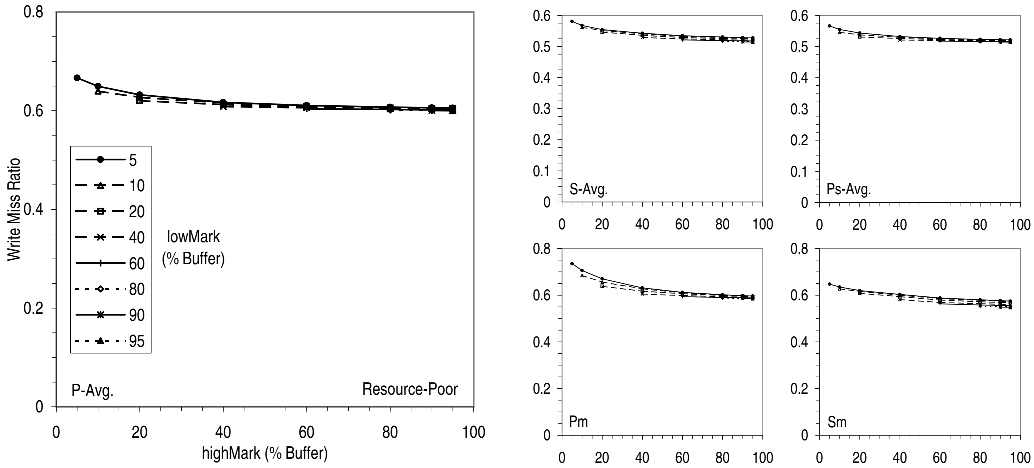


Figure C.16: Effect of $lowMark$ and $highMark$ on Write Miss Ratio (Resource-Poor).

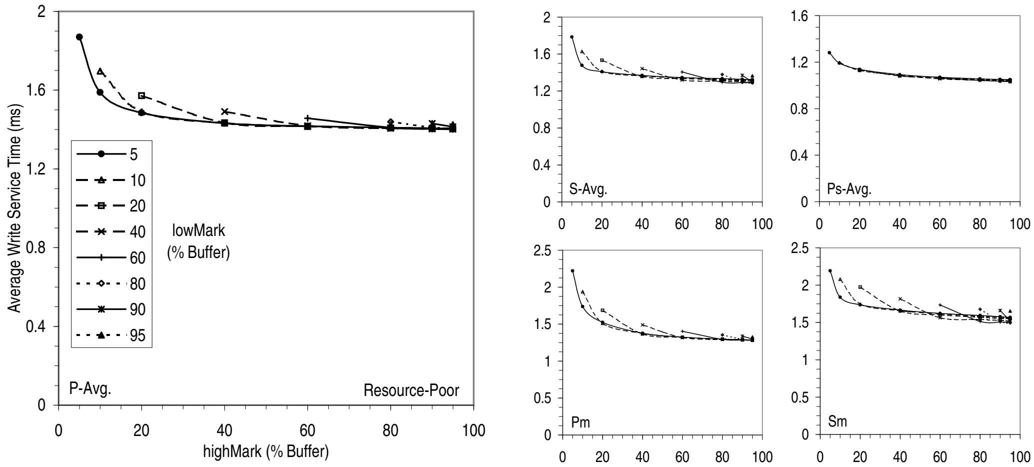


Figure C.17: Effect of $lowMark$ and $highMark$ on Average Write Service Time (Resource-Poor).

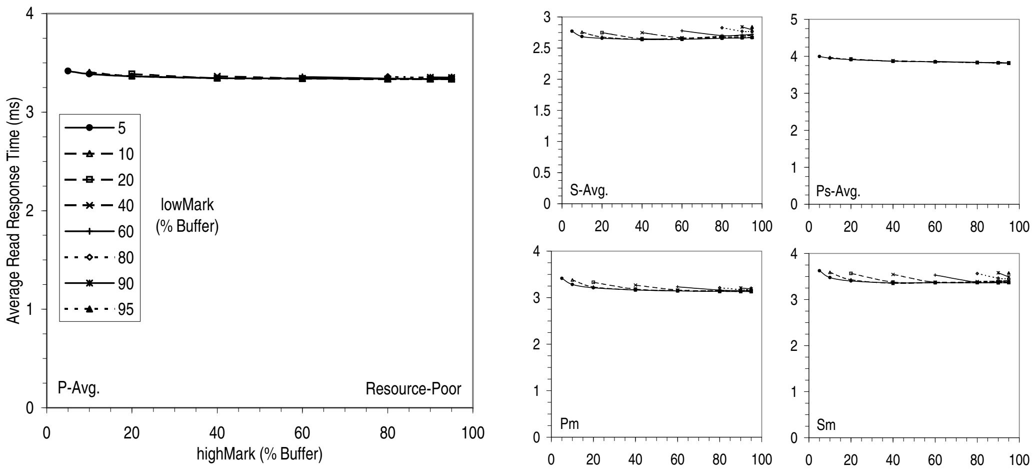
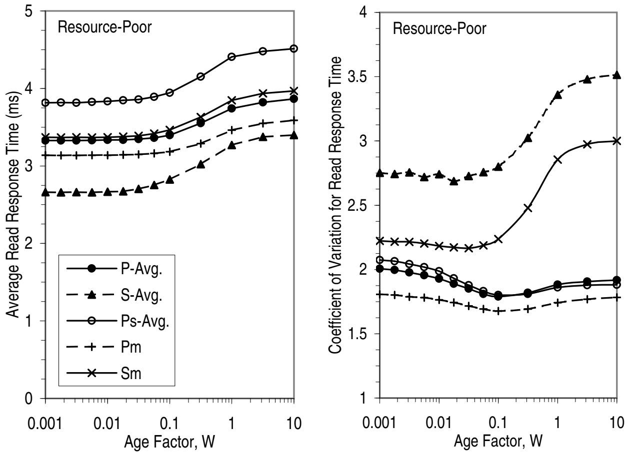
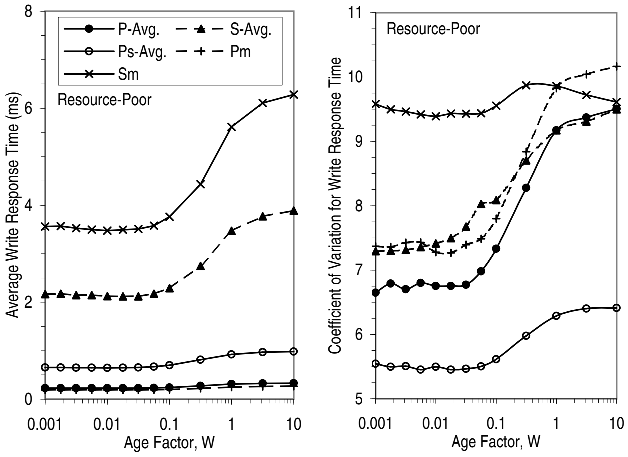


Figure C.18: Effect of $lowMark$ and $highMark$ on Average Read Response Time (Resource-Poor).

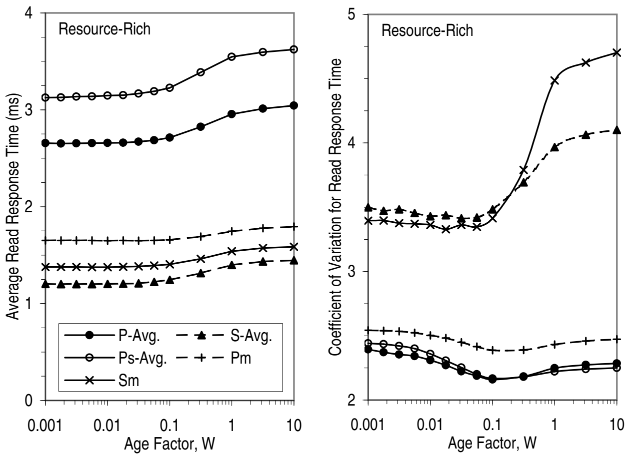


(a) Reads.

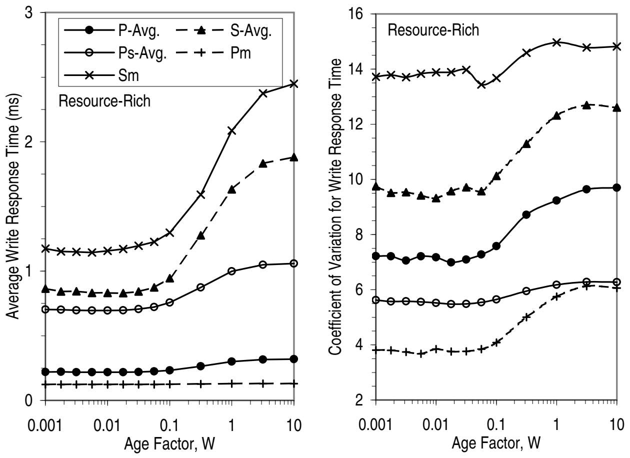


(b) Writes.

Figure C.19: Effect of Age Factor, W, on Response Time (Resource-Poor).

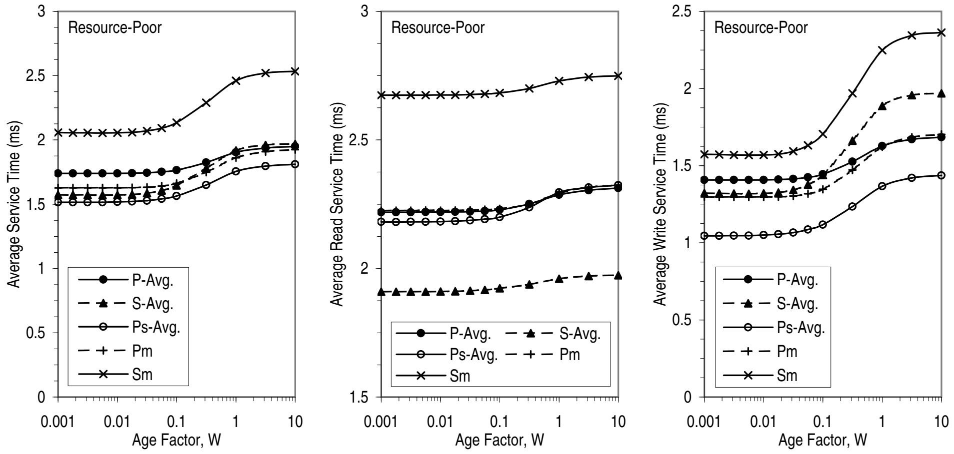


(a) Reads.

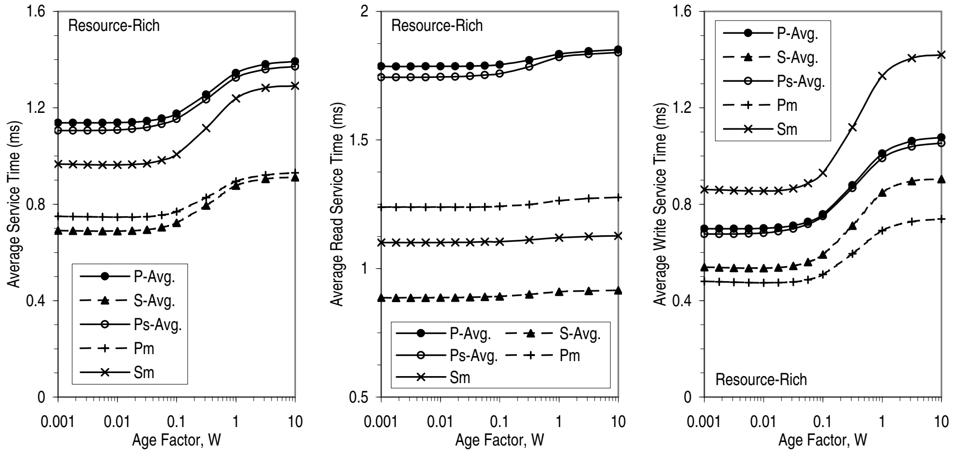


(b) Writes.

Figure C.20: Effect of Age Factor, W, on Response Time (Resource-Rich).

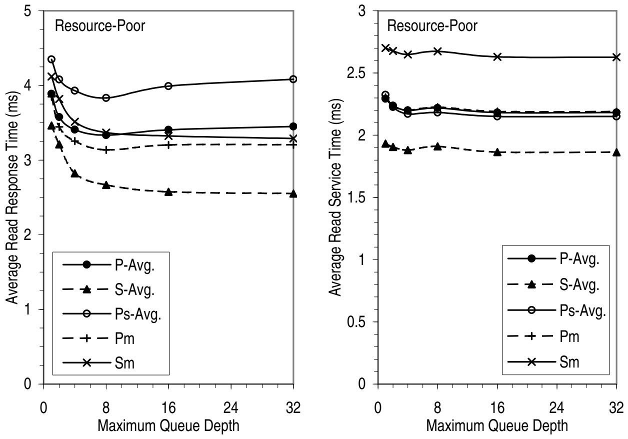


(a) Resource-Poor.

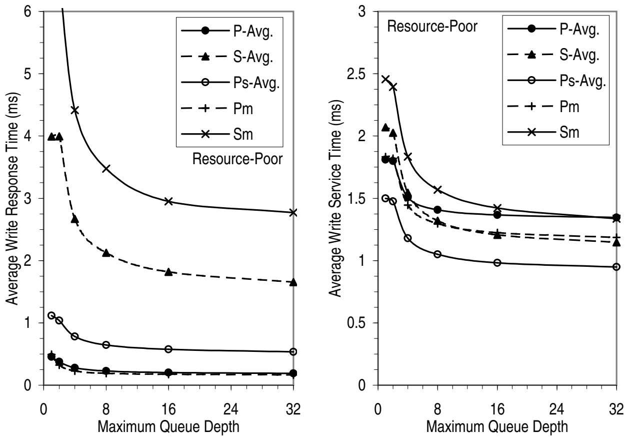


(b) Resource-Rich.

Figure C.21: Effect of Age Factor, W, on Service Time.

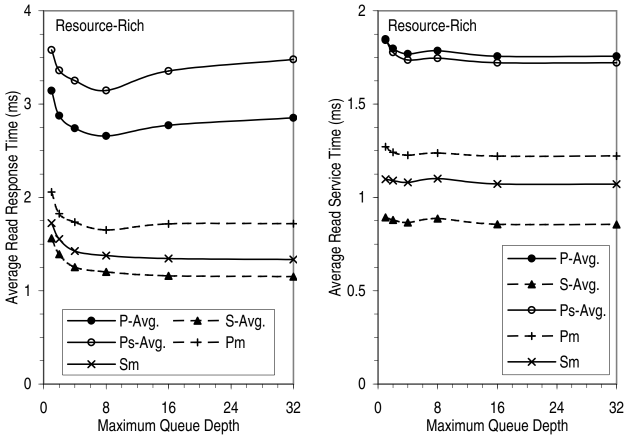


(a) Reads.

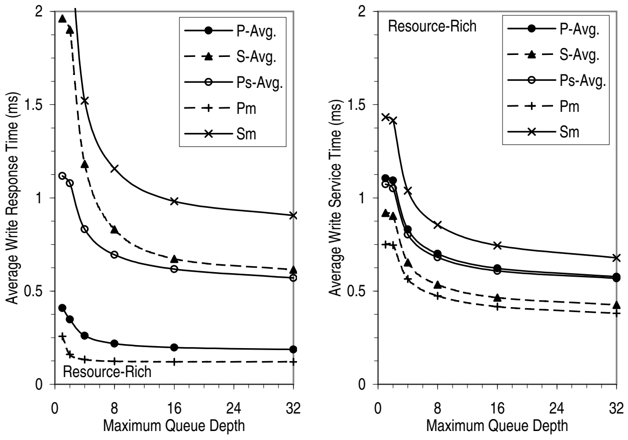


(b) Writes.

Figure C.22: Average Response and Service Times as a Function of the Maximum Queue Depth (Resource-Poor).



(a) Reads.



(b) Writes.

Figure C.23: Average Response and Service Times as a Function of the Maximum Queue Depth (Resource-Rich).

		Average Read Response Time								Average Read Service Time							
		Max. Q Depth = 2		4		8		16		Max. Q Depth = 2		4		8		16	
		ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]
Resource-Poor	P-Avg.	3.58	7.45	3.41	11.5	3.34	13.2	3.41	11.7	2.24	2.34	2.20	4.03	2.22	3.03	2.18	4.76
	S-Avg.	3.21	6.89	2.82	16.8	2.67	20.1	2.58	22.7	1.91	1.58	1.88	2.79	1.91	0.549	1.86	3.52
	Ps-Avg.	4.08	5.73	3.93	8.96	3.83	11.1	3.99	7.75	2.23	4.11	2.17	6.49	2.18	5.94	2.15	7.40
	Pm	3.44	10.5	3.26	15.4	3.14	18.4	3.20	16.8	2.24	2.98	2.20	4.50	2.23	3.44	2.19	4.98
	Sm	3.82	7.30	3.51	14.7	3.37	18.2	3.32	19.3	2.68	0.889	2.65	1.96	2.67	0.982	2.63	2.64
Resource-Rich	P-Avg.	2.88	8.08	2.74	12.2	2.66	14.8	2.77	11.8	1.80	2.31	1.77	3.78	1.79	2.67	1.76	4.43
	S-Avg.	1.39	10.0	1.25	16.9	1.20	18.3	1.16	21.5	0.879	1.37	0.865	2.68	0.886	-0.896	0.856	3.57
	Ps-Avg.	3.36	5.66	3.25	8.61	3.15	11.7	3.35	6.40	1.78	3.65	1.74	5.75	1.75	5.12	1.72	6.48
	Pm	1.83	11.3	1.73	15.7	1.65	19.8	1.72	16.6	1.24	2.32	1.23	3.55	1.24	2.58	1.22	3.93
	Sm	1.55	9.94	1.43	17.4	1.38	20.2	1.35	22.0	1.09	0.619	1.08	1.56	1.10	-0.310	1.07	2.31

[i] Improvement over queue depth of one ([original value – new value]/[original value]).

(a) Reads.

		Average Write Response Time								Average Write Service Time							
		Max. Q Depth = 2		4		8		16		Max. Q Depth = 2		4		8		16	
		ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]	ms	%[i]
Resource-Poor	P-Avg.	0.375	19.2	0.277	38.2	0.227	47.7	0.202	52.3	1.80	0.672	1.51	16.8	1.41	22.3	1.37	24.6
	S-Avg.	3.99	14.7	2.67	40.7	2.13	50.3	1.82	56.2	2.03	1.75	1.55	24.4	1.32	34.8	1.21	39.8
	Ps-Avg.	1.04	8.38	0.782	30.2	0.646	41.8	0.576	48.0	1.48	1.74	1.18	21.5	1.05	30.1	0.982	34.7
	Pm	0.323	34.7	0.228	53.9	0.190	61.7	0.174	64.9	1.82	0.651	1.44	21.1	1.30	29.2	1.22	33.2
	Sm	6.49	3.02	4.42	34.0	3.48	48.0	2.95	55.9	2.39	2.51	1.83	25.3	1.57	36.1	1.42	42.1
Resource-Rich	P-Avg.	0.348	16.4	0.261	34.5	0.218	43.5	0.197	48.1	1.09	1.11	0.831	24.6	0.700	36.3	0.622	43.3
	S-Avg.	1.90	14.1	1.18	37.3	0.831	45.4	0.673	49.0	0.904	2.08	0.654	29.3	0.535	42.4	0.465	50.1
	Ps-Avg.	1.08	4.04	0.832	24.7	0.695	36.4	0.617	43.2	1.05	2.34	0.804	25.0	0.681	36.3	0.608	42.9
	Pm	0.161	37.6	0.133	48.5	0.123	52.1	0.121	53.2	0.746	0.460	0.565	24.7	0.474	36.8	0.416	44.5
	Sm	2.38	4.36	1.52	39.0	1.16	53.6	0.981	60.6	1.41	1.29	1.04	27.5	0.855	40.3	0.745	48.0

[i] Improvement over queue depth of one ([original value – new value]/[original value]).

(b) Writes.

Table C.1: Average Response and Service Times as Maximum Queue Depth is Increased from One.

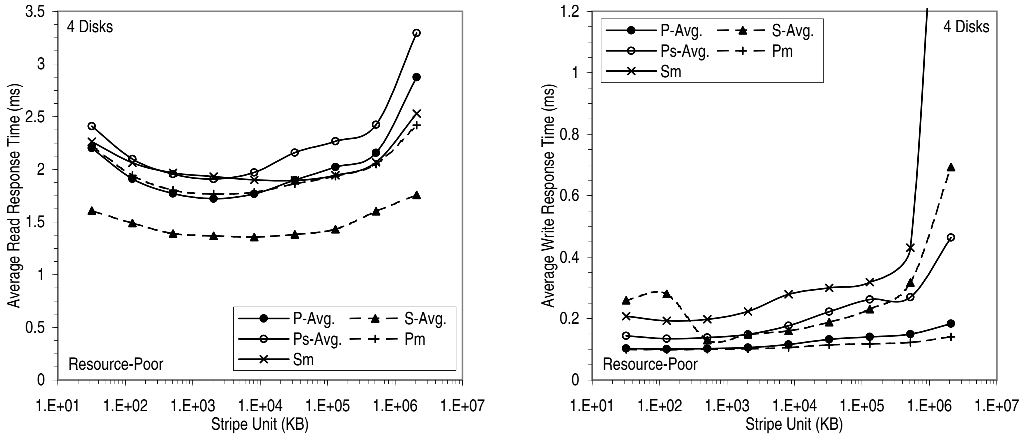
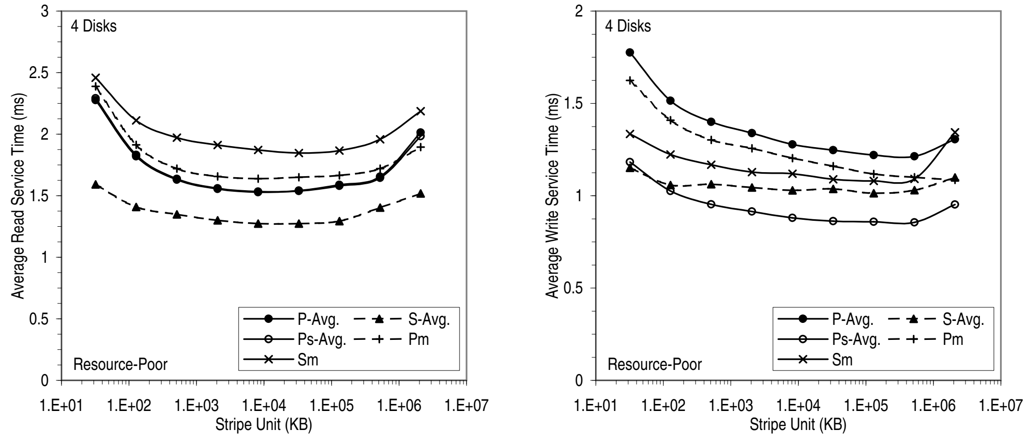
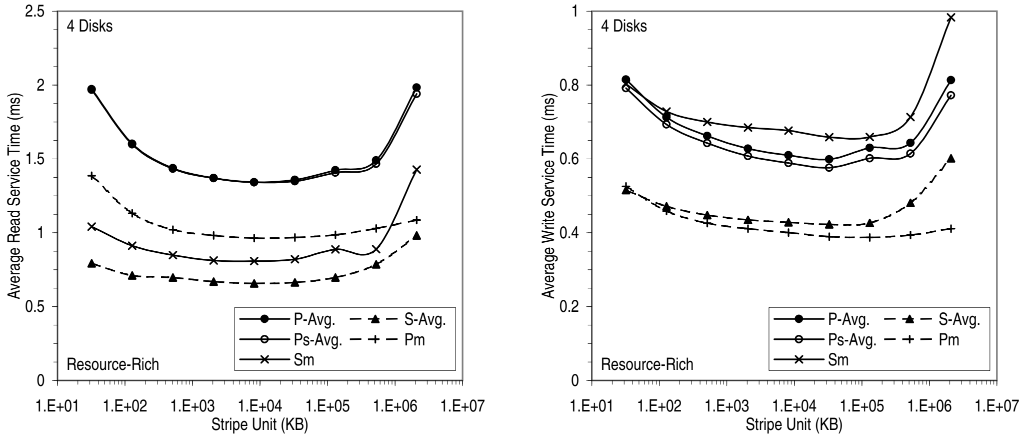


Figure C.24: Average Read and Write Response Time as a Function of Stripe Unit (Resource-Poor).

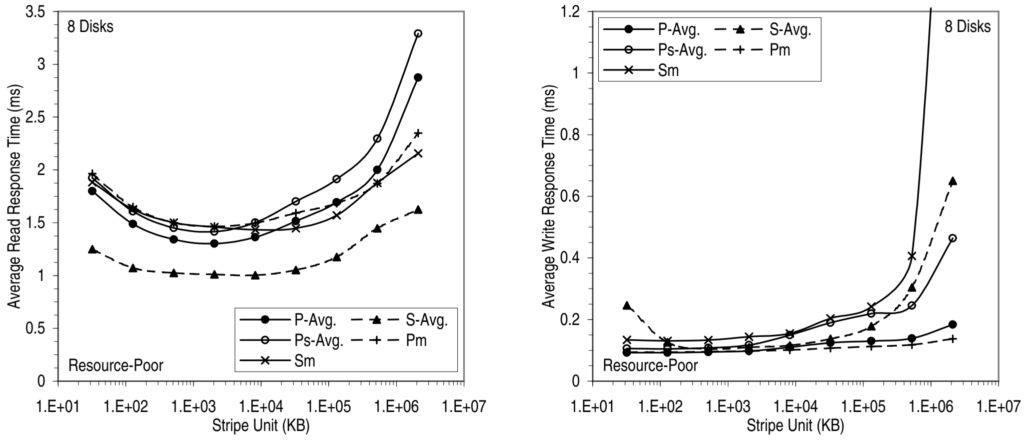


(a) Resource-Poor.

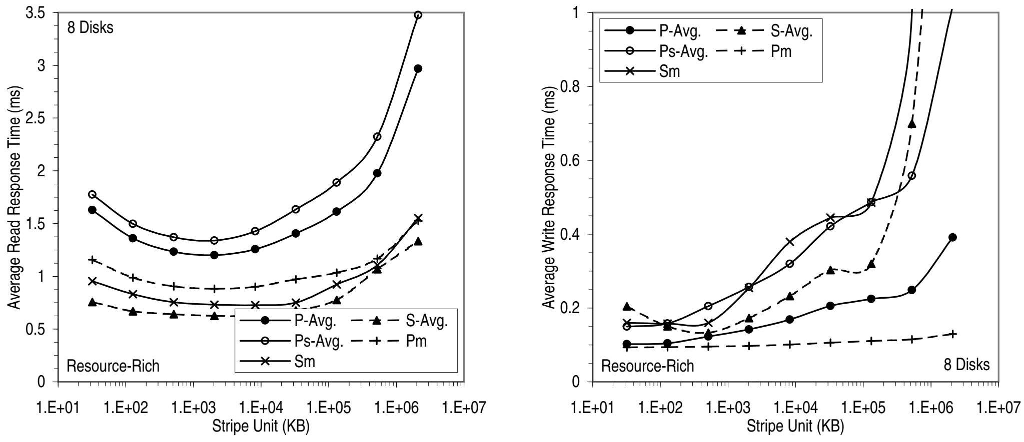


(b) Resource-Rich.

Figure C.25: Average Read and Write Service Time as a Function of Stripe Unit.

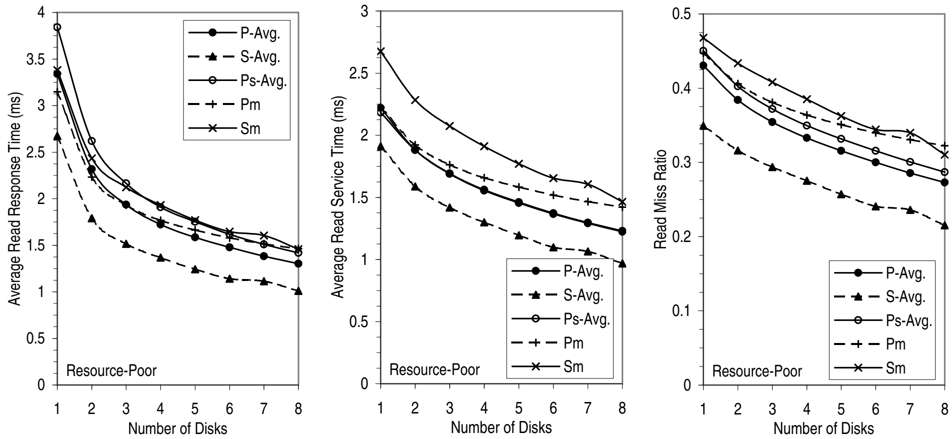


(a) Resource-Poor.

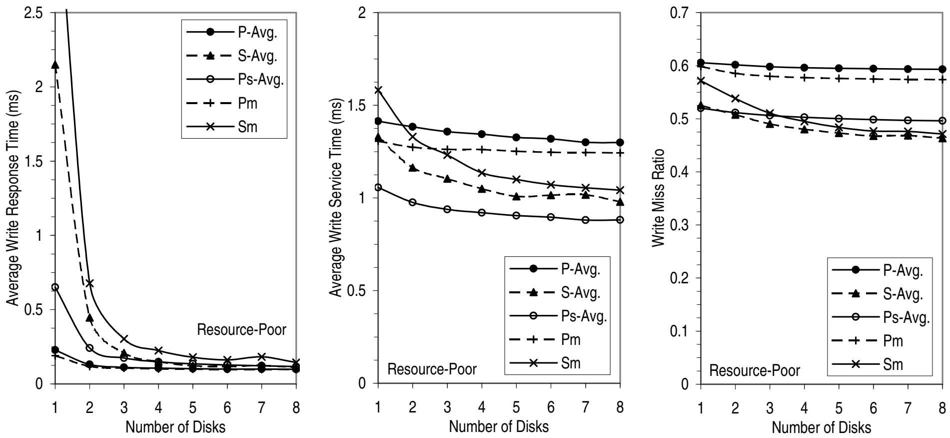


(b) Resource-Rich.

Figure C.26: Average Read and Write Response Time as a Function of Stripe Unit (8 Disks).

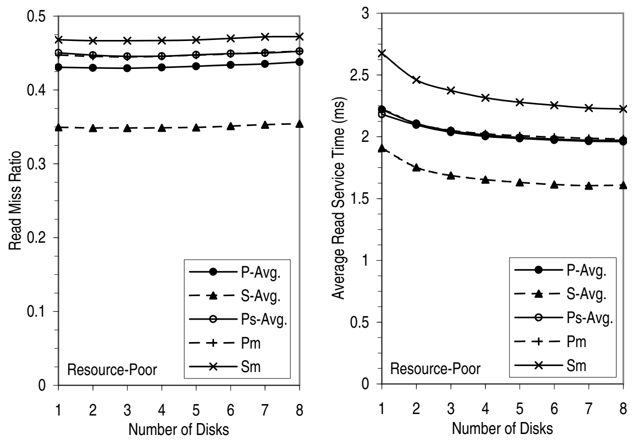


(a) Read.

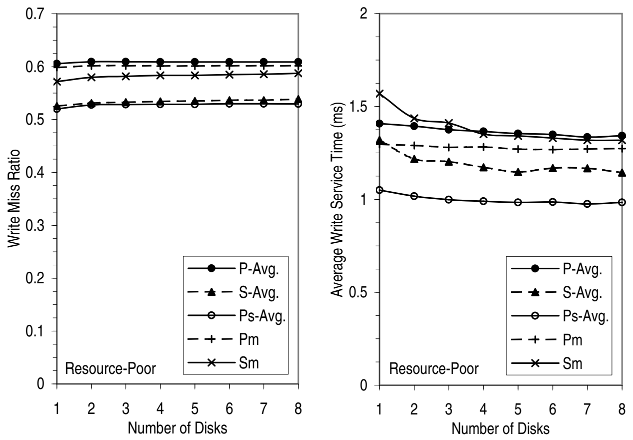


(b) Write.

Figure C.27: Performance as a Function of the Number of Disks (Resource-Poor).

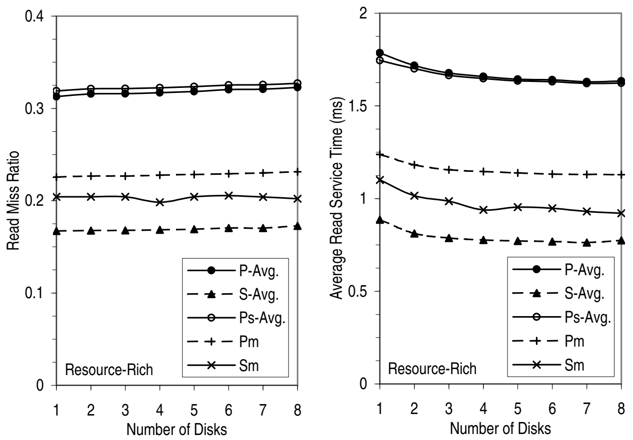


(a) Read.

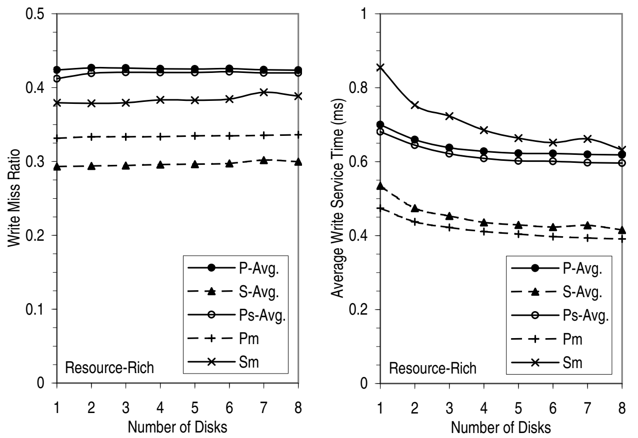


(b) Write.

Figure C.28: Performance as a Function of the Number of Disks and with Constant Total Cache and Buffer Space (Resource-Poor).



(a) Read.



(b) Write.

Figure C.29: Performance as a Function of the Number of Disks and with Constant Total Cache and Buffer Space (Resource-Rich).

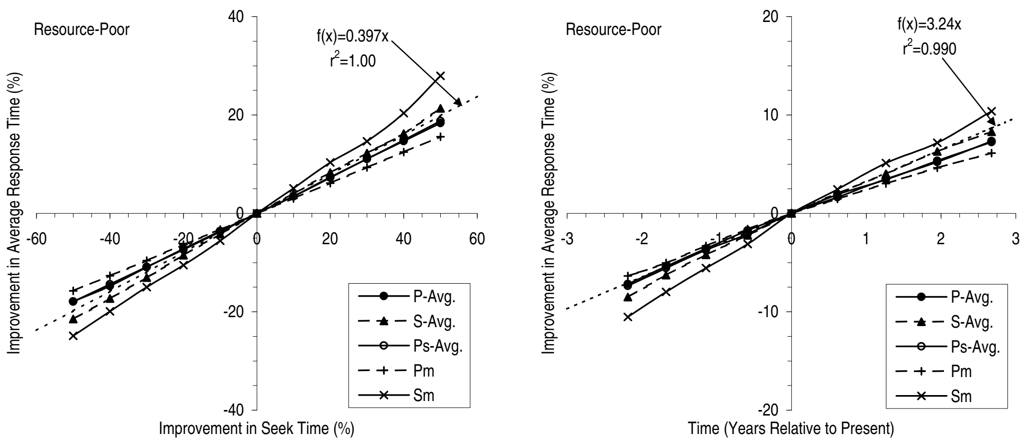
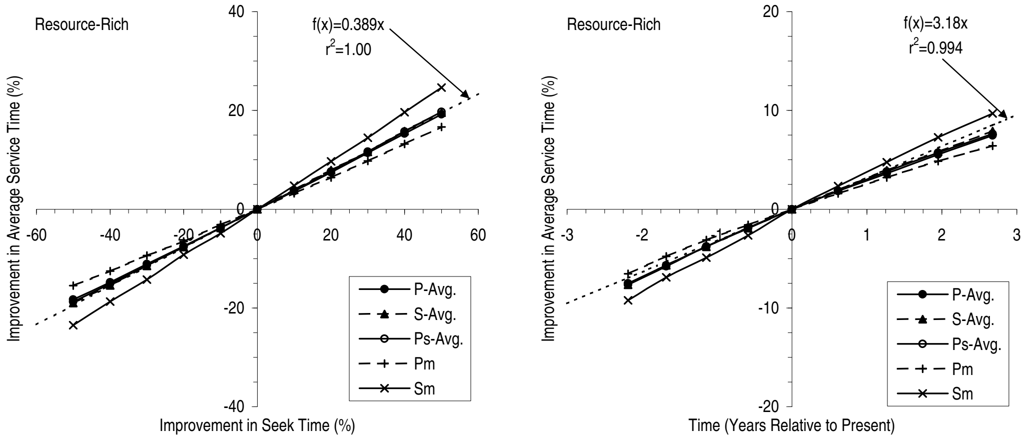
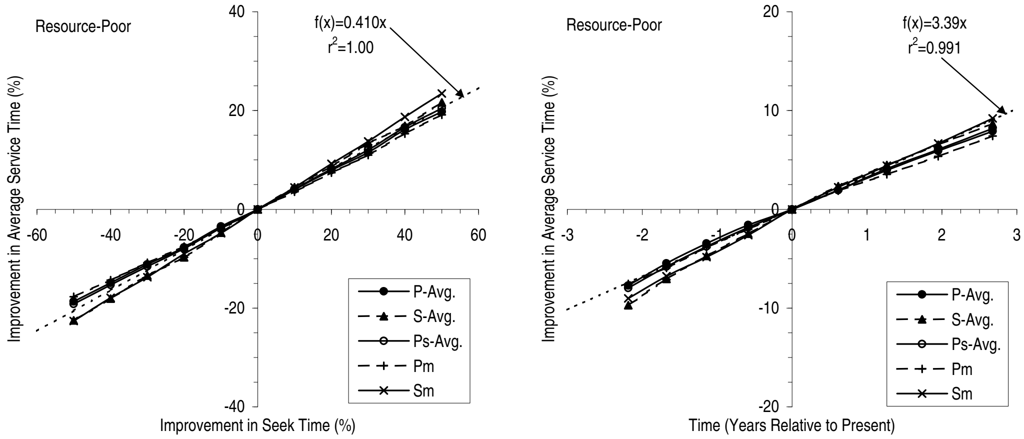


Figure C.30: Effect of Improvement in Seek Time on Average Response Time (Resource-Poor).



(a) Resource-Rich.



(b) Resource-Poor.

Figure C.31: Effect of Improvement in Seek Time on Average Service Time.

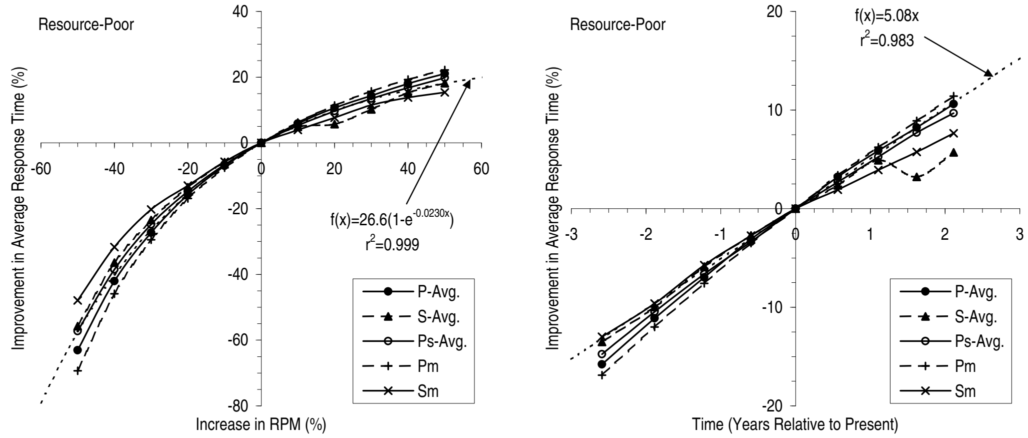
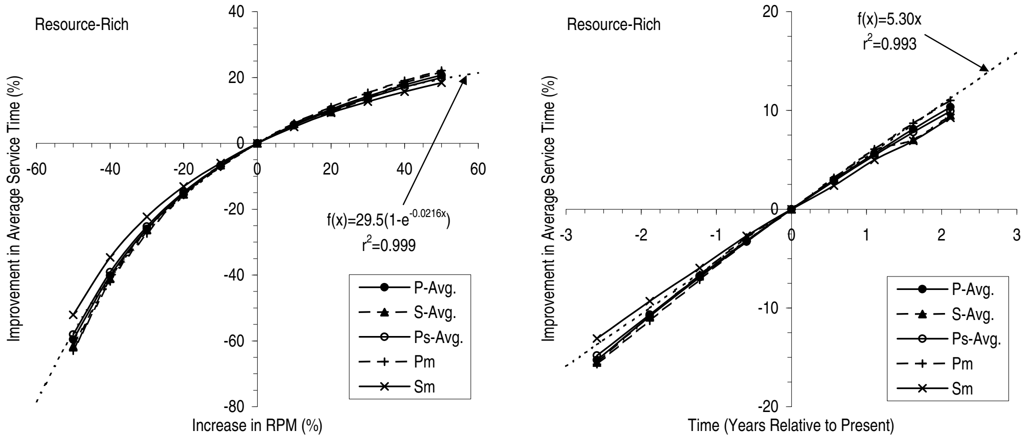
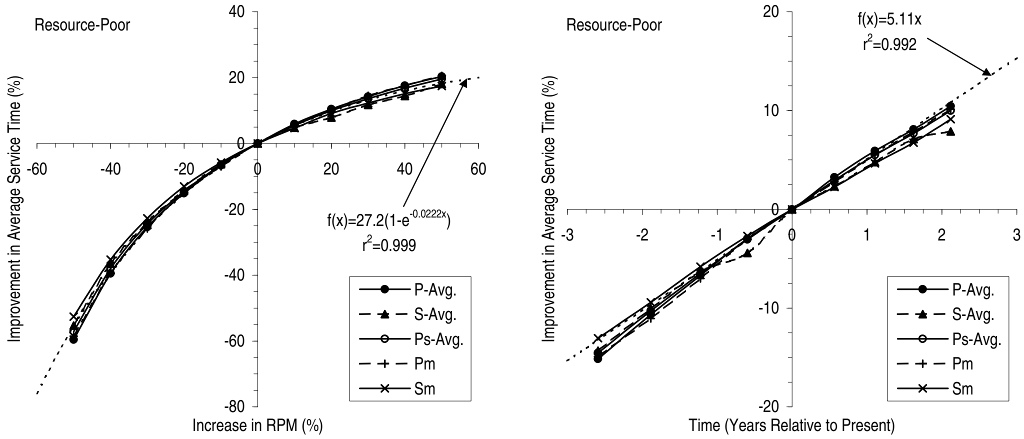


Figure C.32: Effect of RPM Scaling on Average Response Time (Resource-Poor).



(a) Resource-Rich.



(b) Resource-Poor.

Figure C.33: Effect of RPM Scaling on Average Service Time.

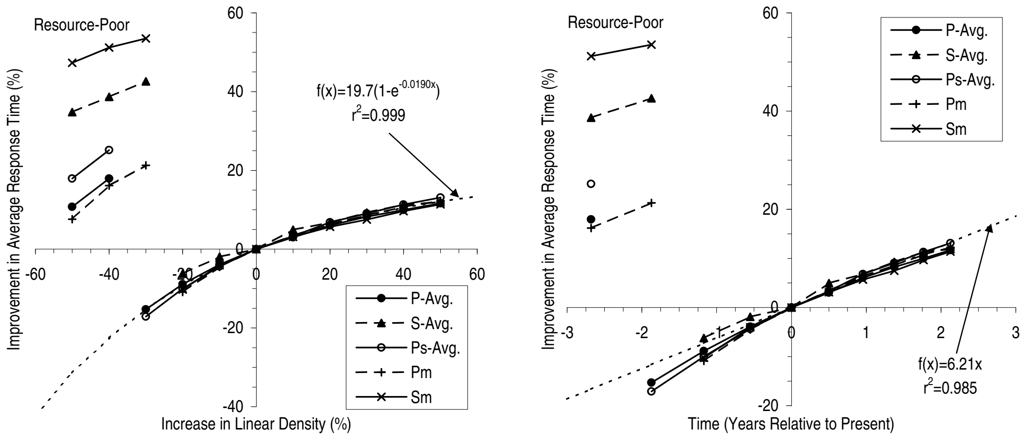
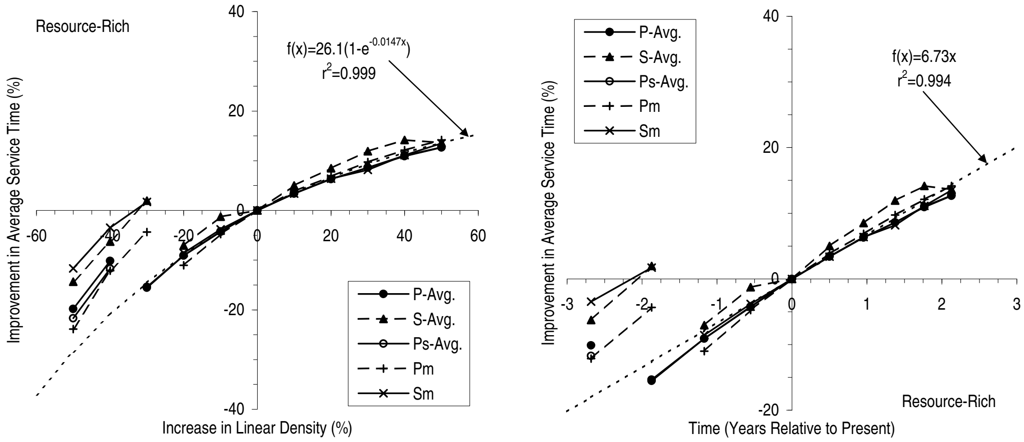
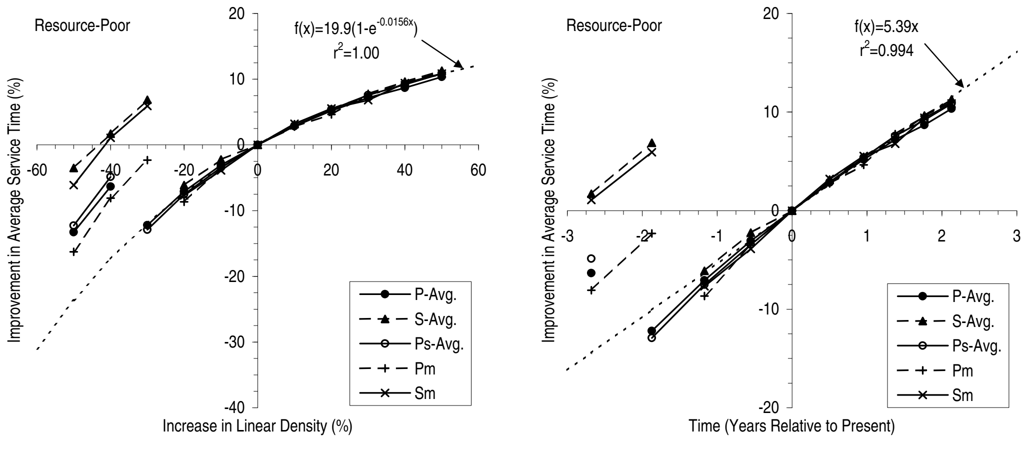


Figure C.34: Effect of Increased Linear Density on Average Response Time (Resource-Poor).



(a) Resource-Rich.



(b) Resource-Poor.

Figure C.35: Effect of Increased Linear Density on Average Service Time.

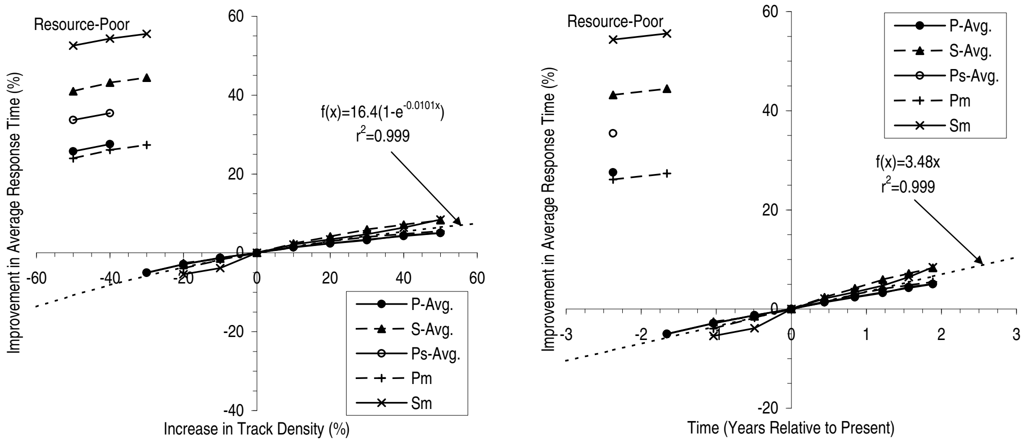
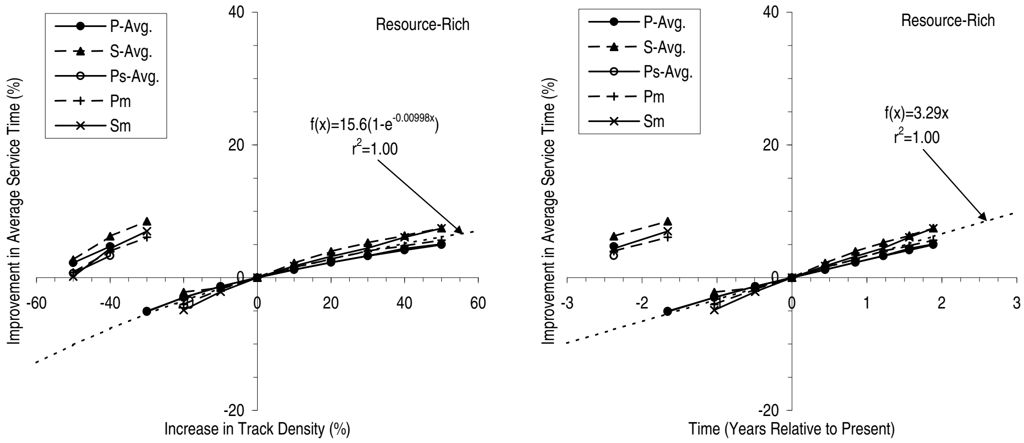
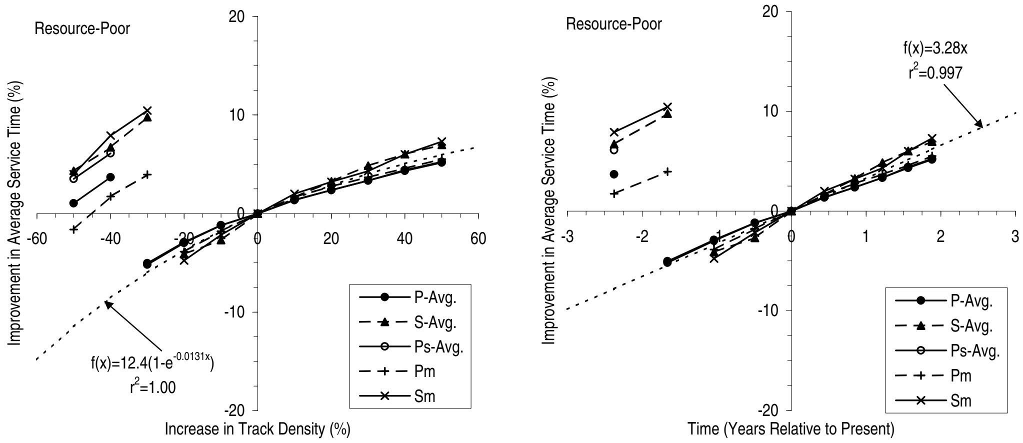


Figure C.36: Effect of Increased Track Density on Average Response Time (Resource-Poor).



(a) Resource-Rich.



(b) Resource-Poor.

Figure C.37: Effect of Increased Track Density on Average Service Time.

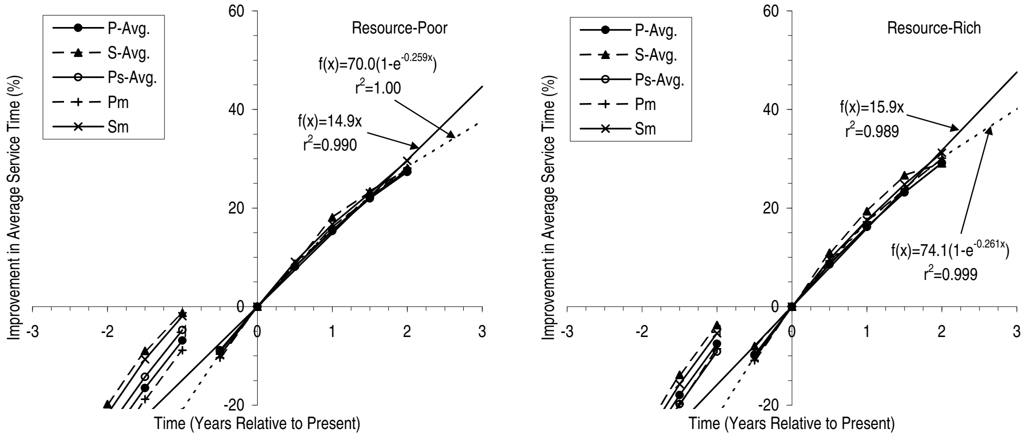


Figure C.38: Overall Effect of Disk Improvement on Average Service Time.

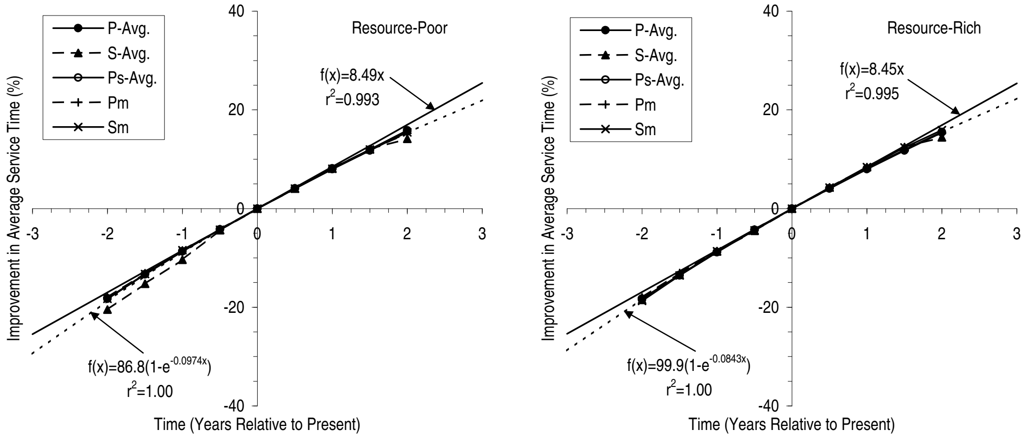


Figure C.39: Effect of Mechanical Improvement on Average Service Time.

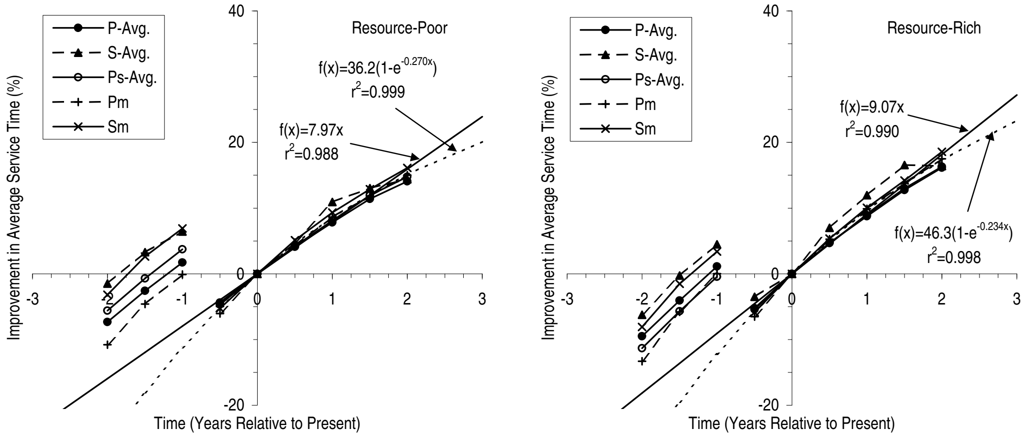
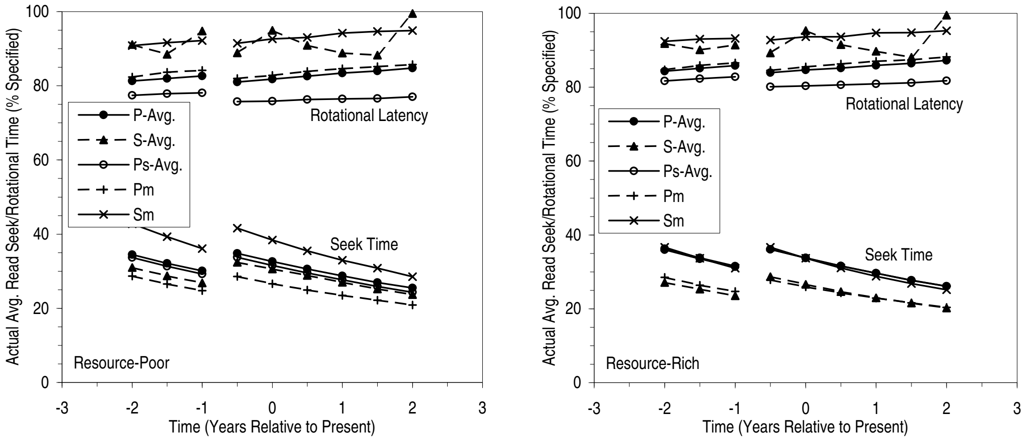
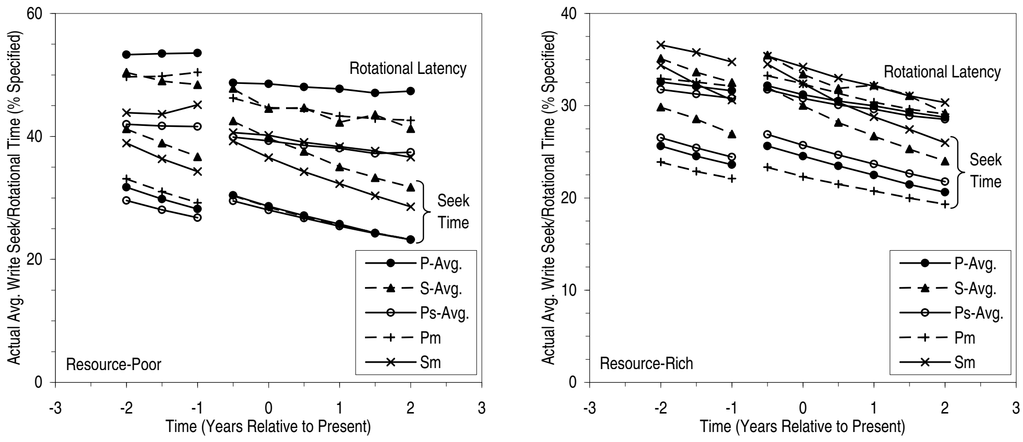


Figure C.40: Effect of Areal Density Increase on Average Service Time.



(a) Reads.



(b) Writes.

Figure C.41: Actual Average Seek/Rotational Time as Percentage of Specified Values (Resource-Rich).

Appendix D

Additional Results for Chapter 4

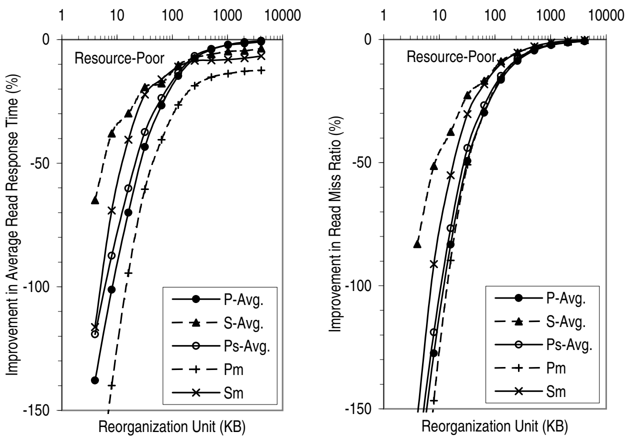


Figure D.1: Effectiveness of Organ Pipe Placement at Improving Read Performance (Resource-Poor).

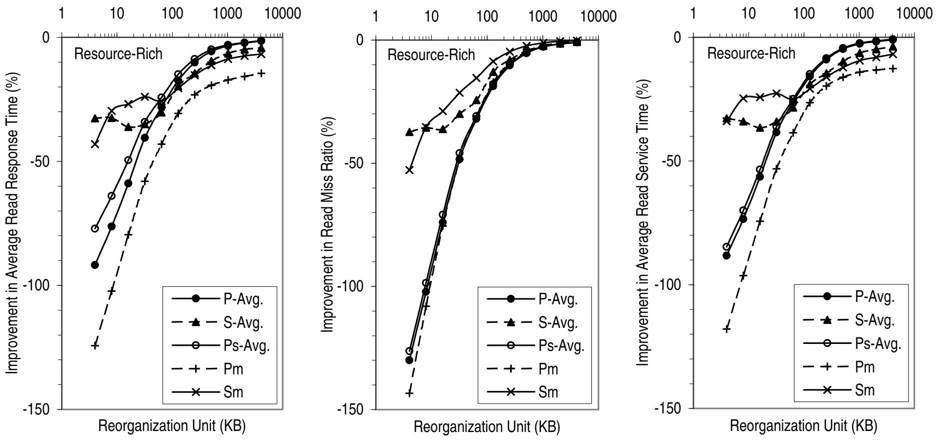


Figure D.2: Effectiveness of Heat Layout at Improving Read Performance (Resource-Rich).

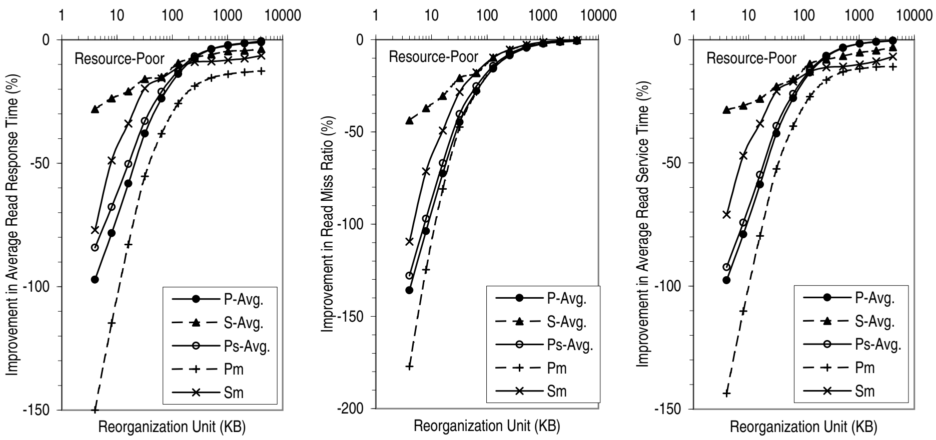


Figure D.3: Effectiveness of Heat Layout at Improving Read Performance (Resource-Poor).

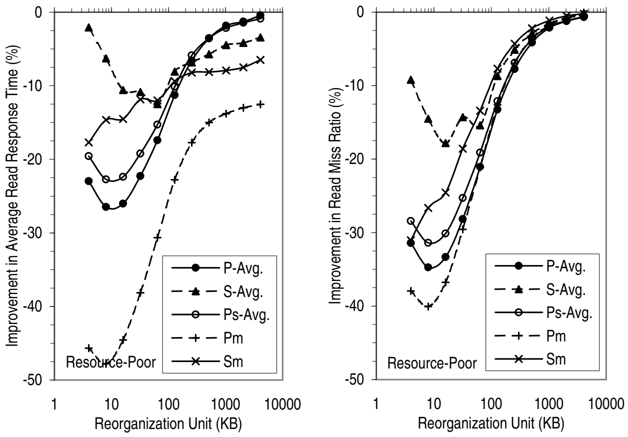


Figure D.4: Effectiveness of Link Closure Placement at Improving Read Performance (Resource-Poor).

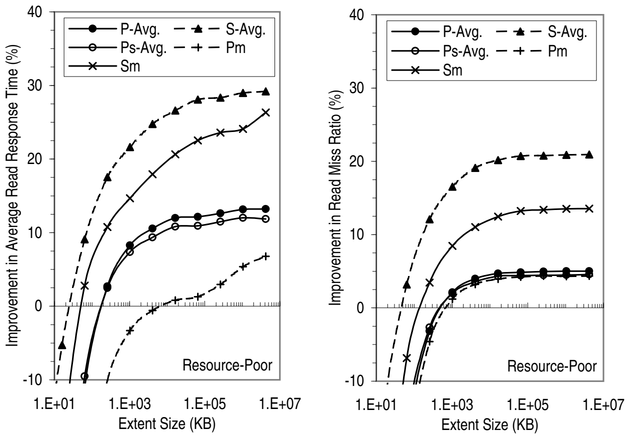


Figure D.5: Effectiveness of Packed Extents Layout at Improving Read Performance (Resource-Poor).

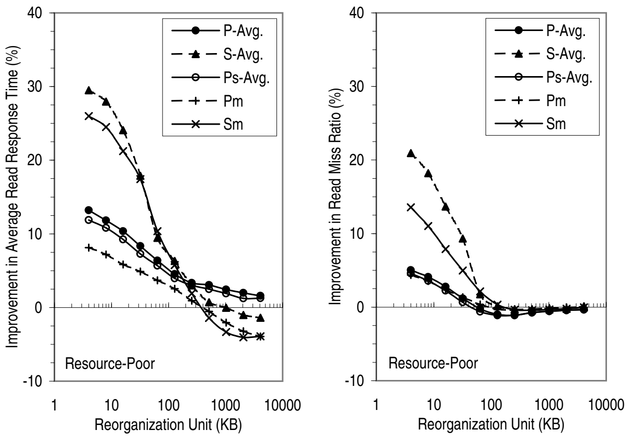
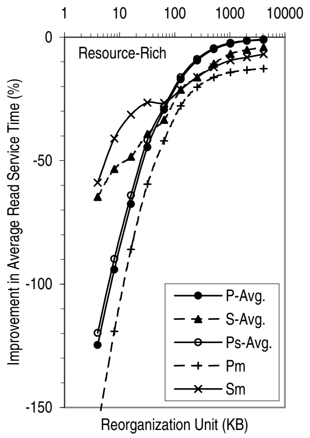
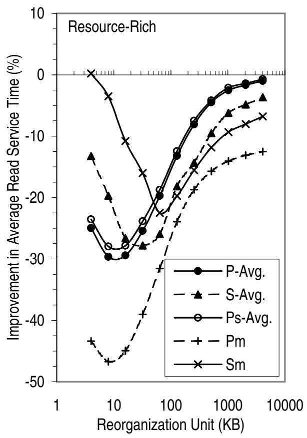


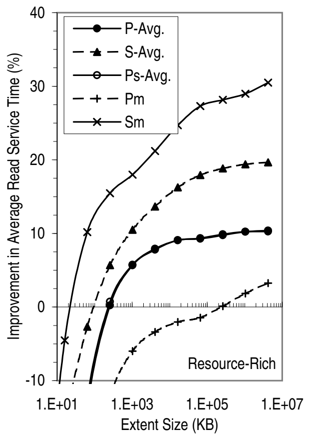
Figure D.6: Effectiveness of Sequential Layout at Improving Read Performance (Resource-Poor).



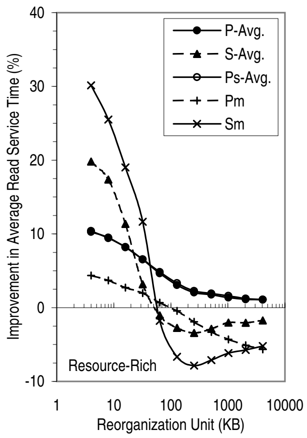
(a) Organ Pipe.



(b) Link Closure.

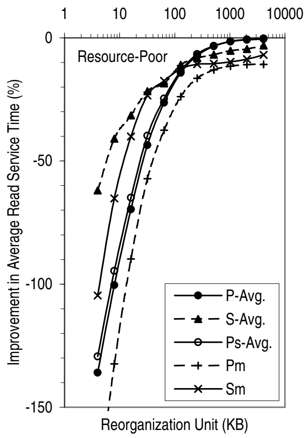


(c) Packed Extents.

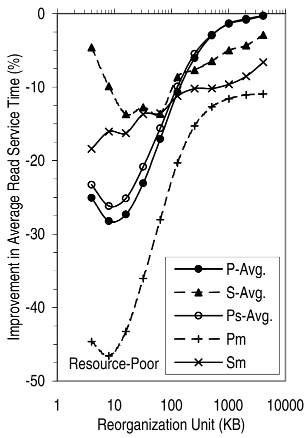


(d) Sequential.

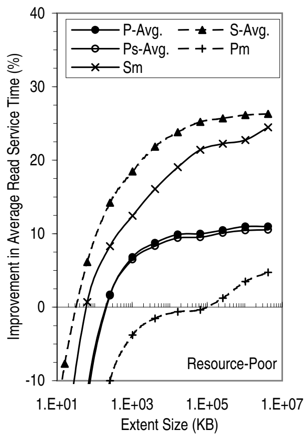
Figure D.7: Improvement in Average Read Service Time for the Various Block Layouts in Heat Clustering (Resource-Rich).



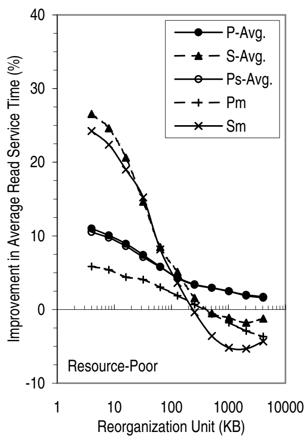
(a) Organ Pipe.



(b) Link Closure.



(c) Packed Extents.



(d) Sequential.

Figure D.8: Improvement in Average Read Service Time for the Various Block Layouts in Heat Clustering (Resource-Poor).

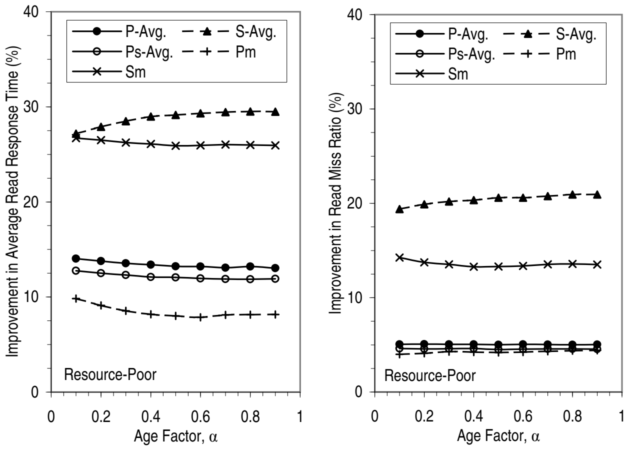
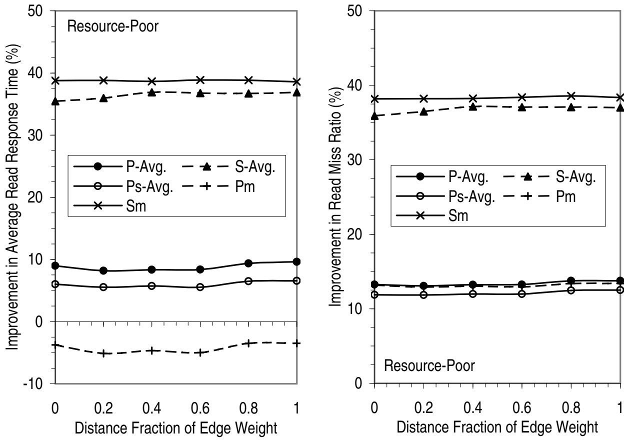
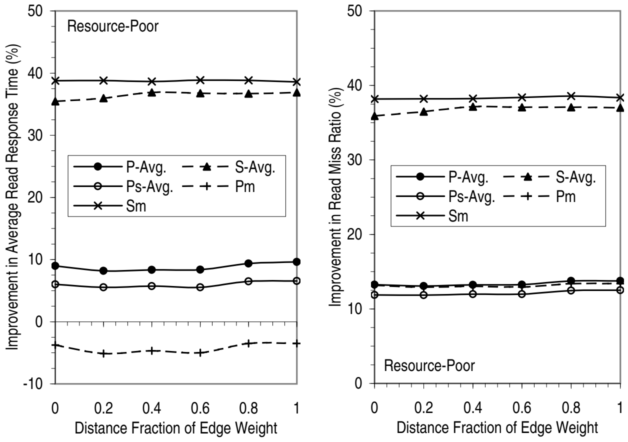


Figure D.9: Sensitivity of Heat Clustering to Age Factor, α (Resource-Poor).



(a) Resource-Poor.



(b) Resource-Rich.

Figure D.10: Sensitivity of Run Clustering to Weighting of Edges.

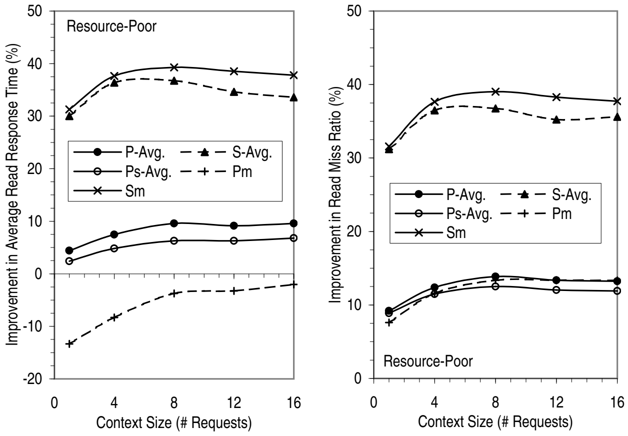


Figure D.11: Sensitivity of Run Clustering to Context Size, τ (Resource-Poor).

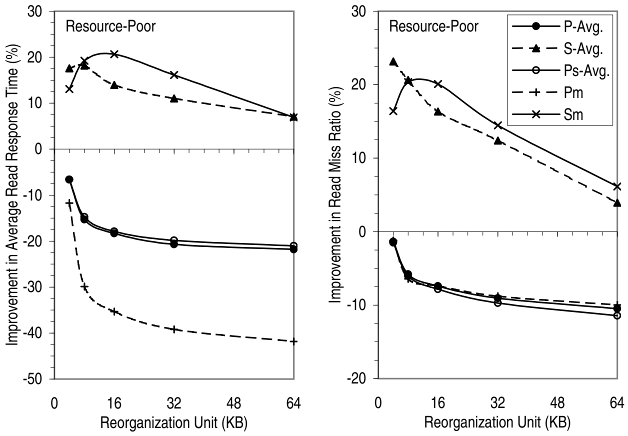


Figure D.12: Effectiveness of Run Clustering with Fixed-Sized Reorganization Units (Resource-Poor).

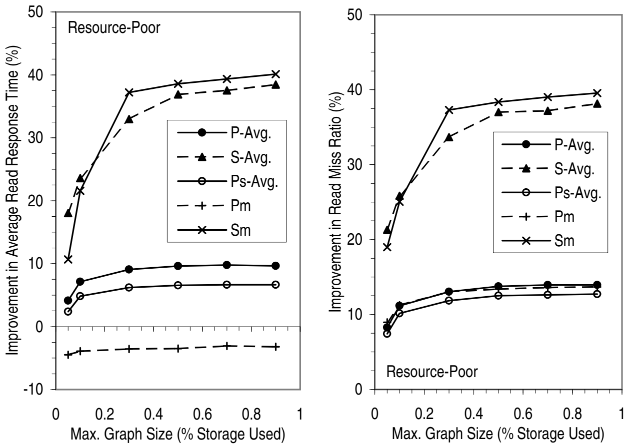
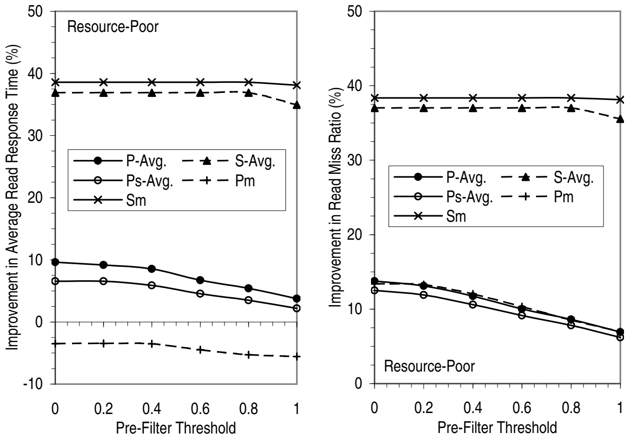
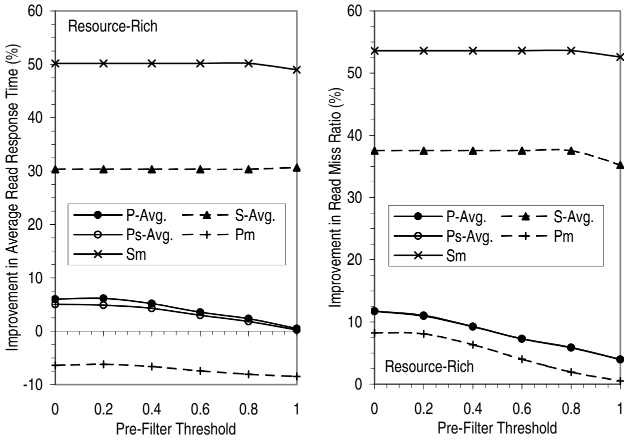


Figure D.13: Sensitivity of Run Clustering to Graph Size (Resource-Poor).



(a) Resource-Poor.



(b) Resource-Rich.

Figure D.14: Effect of Pre-Filtering on Run Clustering.

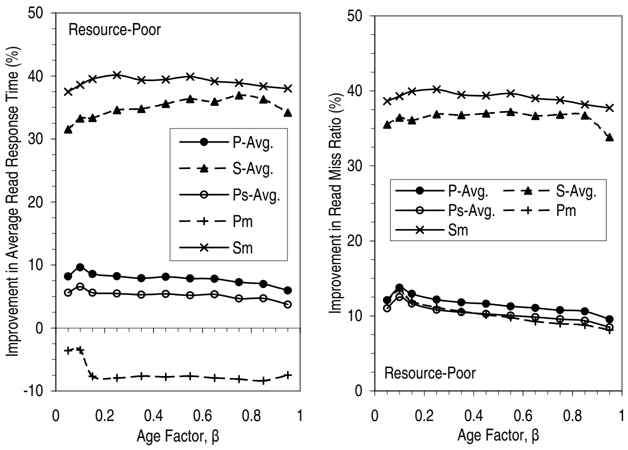


Figure D.15: Sensitivity of Run Clustering to Age Factor, β (Resource-Poor).

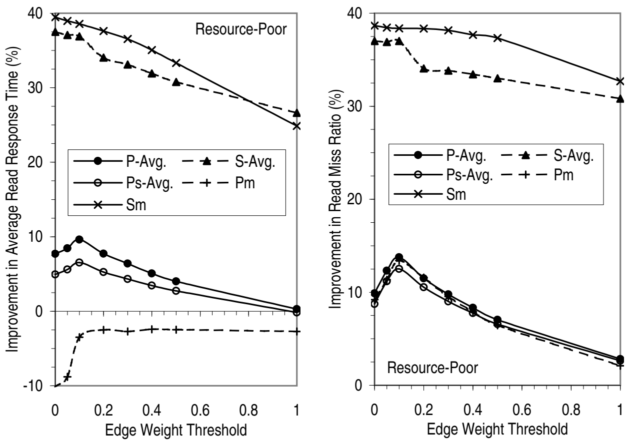
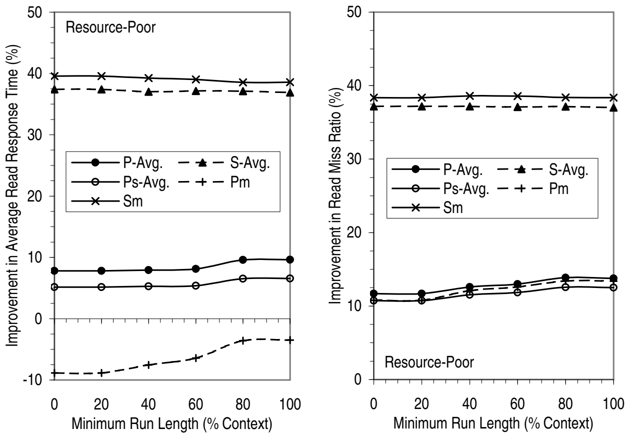
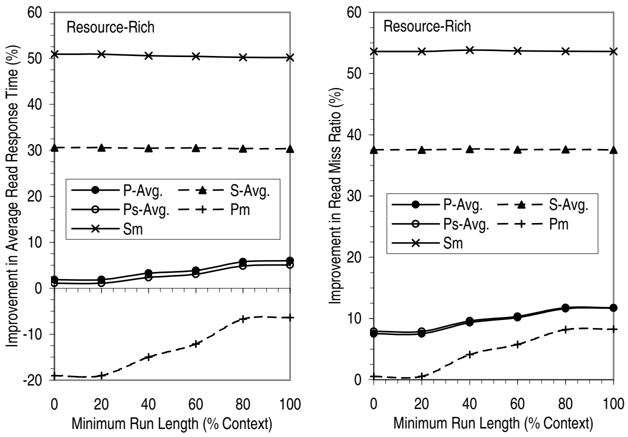


Figure D.16: Sensitivity of Run Clustering to Edge Threshold (Resource-Poor).

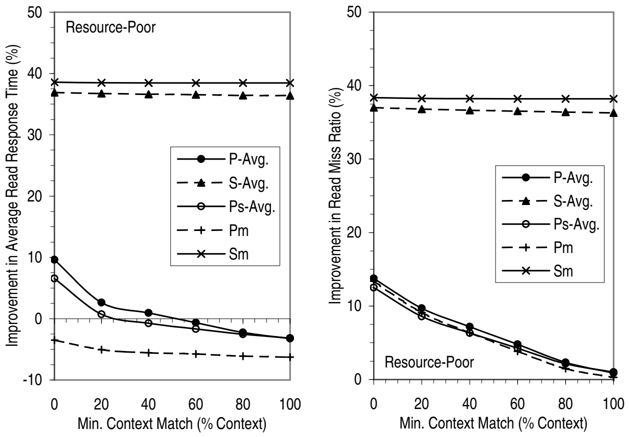


(a) Resource-Poor.

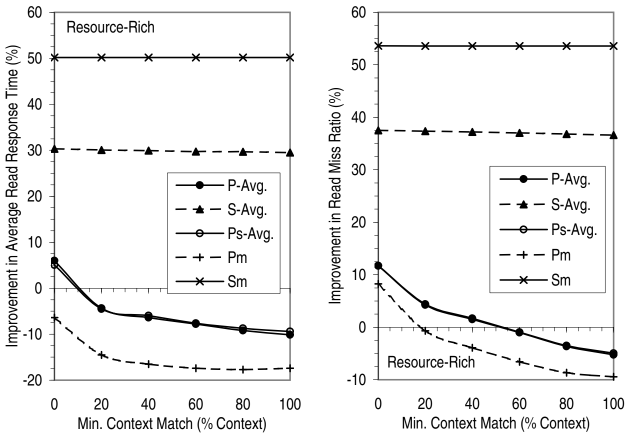


(b) Resource-Rich.

Figure D.17: Effect of Imposing Minimum Run Length.

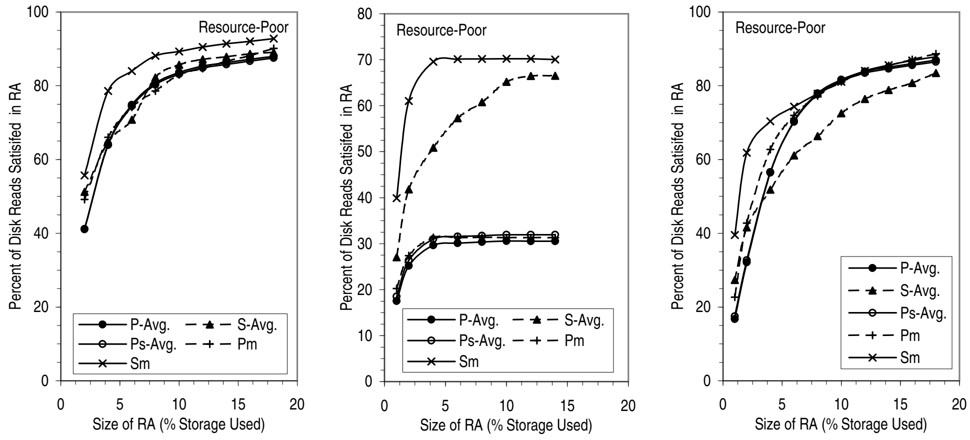


(a) Resource-Poor.



(b) Resource-Rich.

Figure D.18: Effect of Using a Run only when the Contexts Match (Resource-Poor).

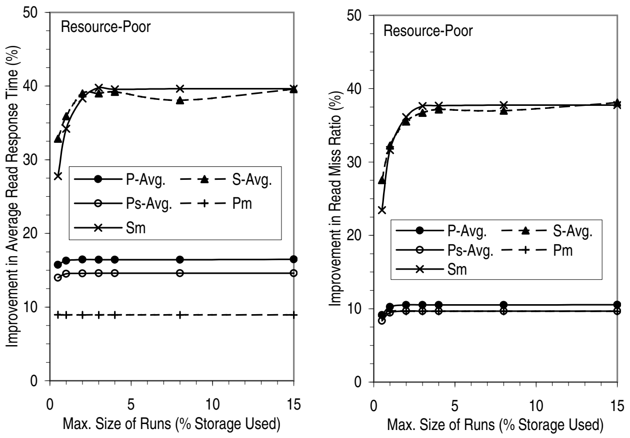


(a) Heat Clustering.

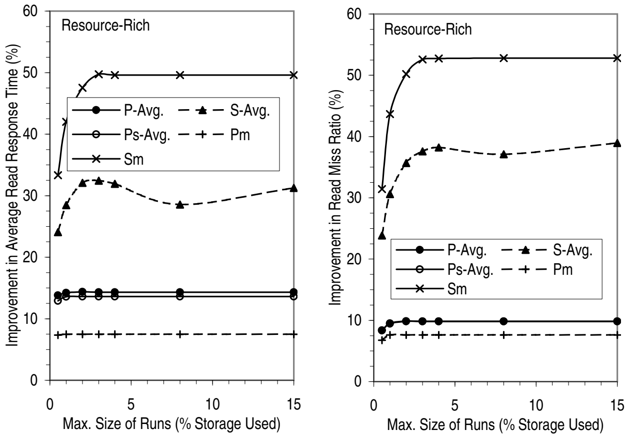
(b) Run Clustering.

(c) Heat and Run Clustering Combined.

Figure D.19: Percent of Disk Reads Satisfied in Reorganized Area (Resource-Poor).



(a) Resource-Poor.



(b) Resource-Rich.

Figure D.20: Effect of Limiting the Total Size of Runs in the Reorganized Area.

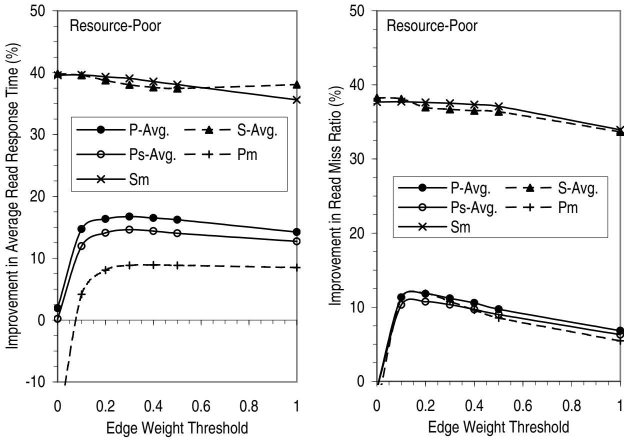


Figure D.21: Sensitivity of Heat and Run Clustering Combined to Edge Threshold (Resource-Poor).

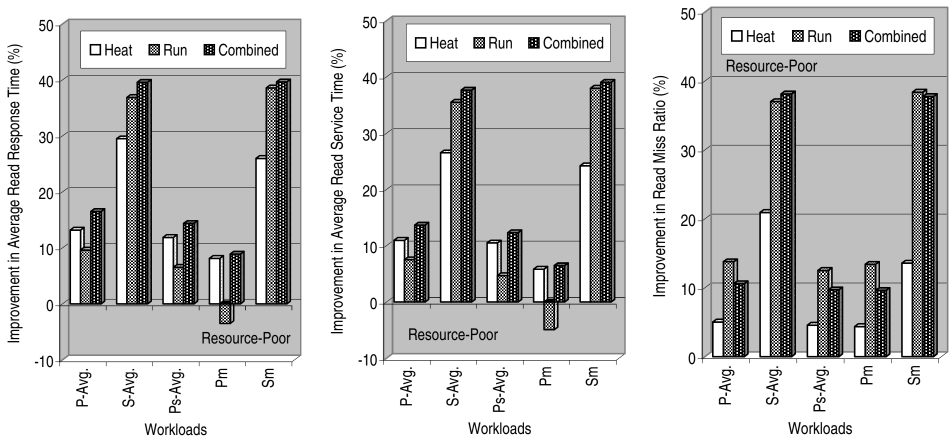
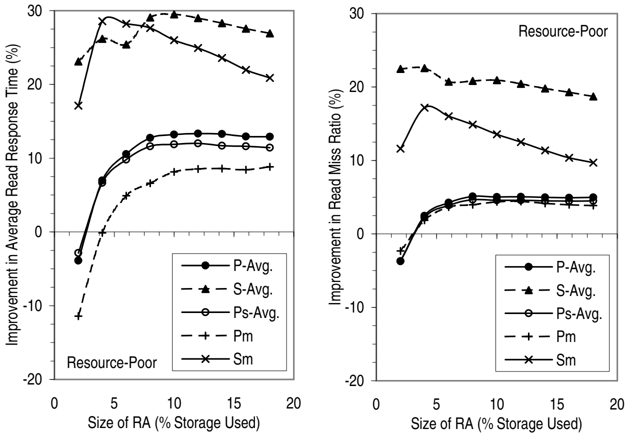
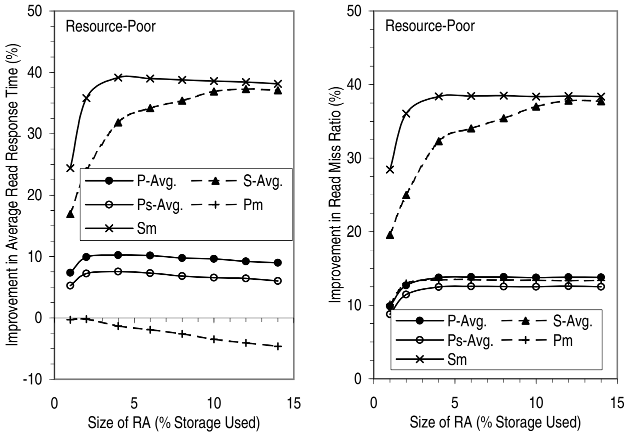


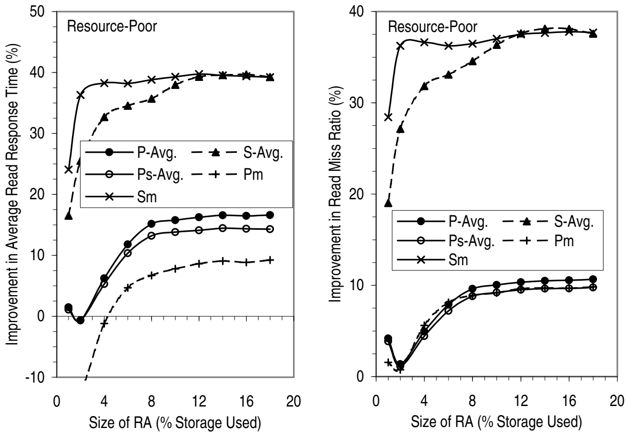
Figure D.22: Performance Improvement with the Various Clustering Schemes (Resource-Poor).



(a) Heat Clustering.

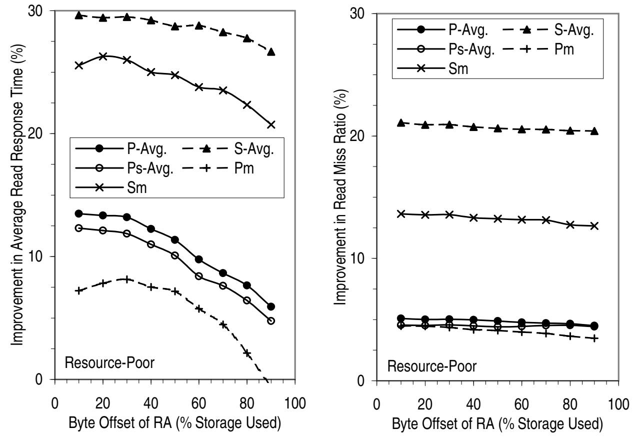


(b) Run Clustering.

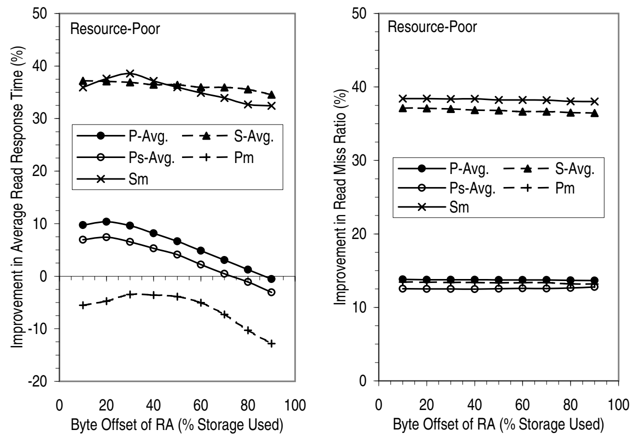


(c) Heat and Run Clustering Combined.

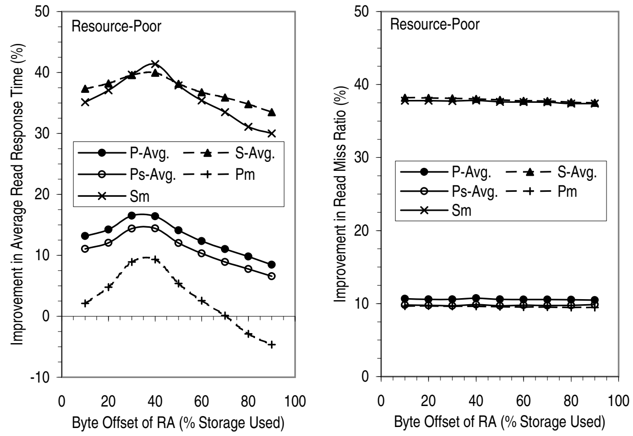
Figure D.23: Sensitivity to Size of Reorganized Area (Resource-Poor).



(a) Heat Clustering.

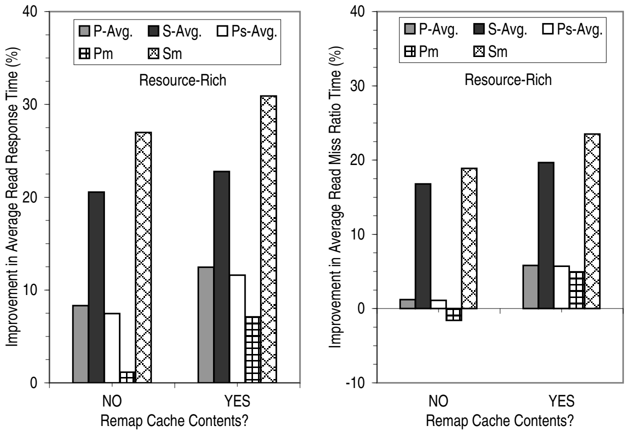


(b) Run Clustering.

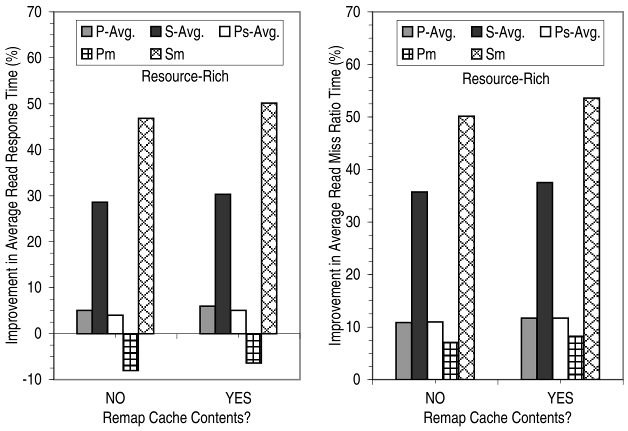


(c) Heat and Run Clustering Combined.

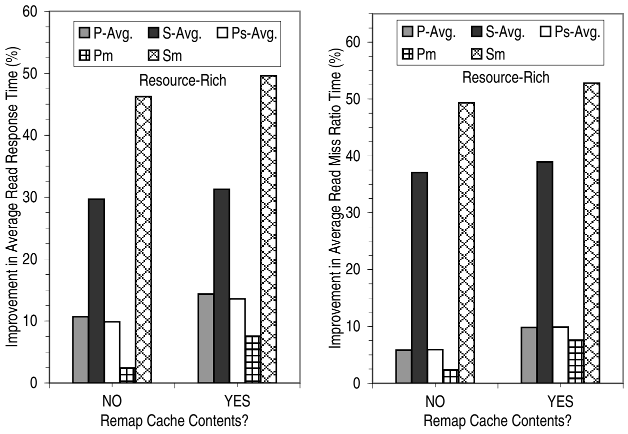
Figure D.24: Sensitivity to Placement of Reorganized Area (Resource-Poor).



(a) Heat Clustering.

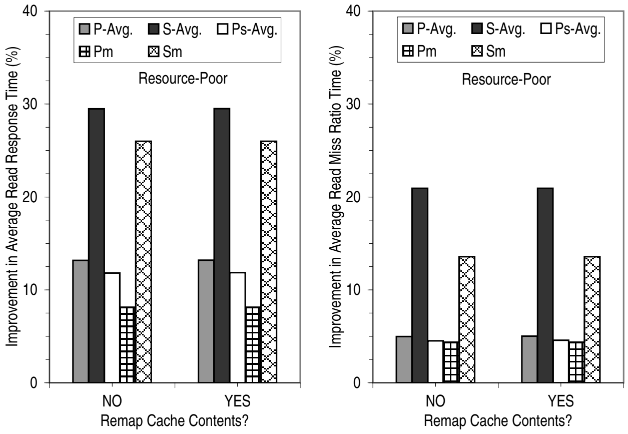


(b) Run Clustering.

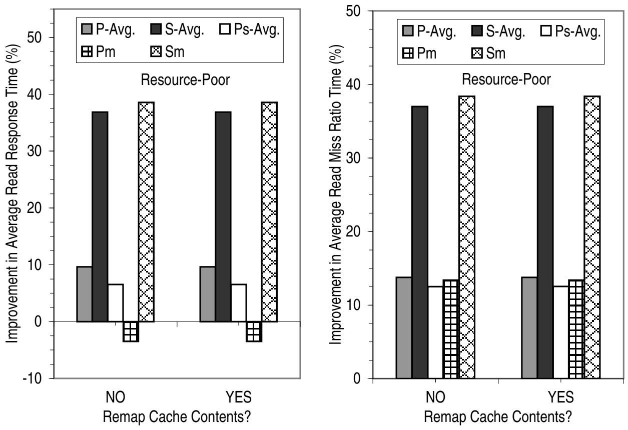


(c) Heat and Run Clustering Combined.

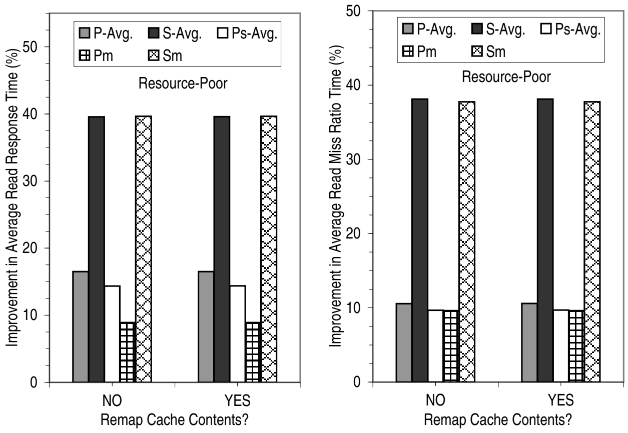
Figure D.25: Effect of Remapping Cache Contents (Resource-Rich).



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure D.26: Effect of Remapping Cache Contents (Resource-Poor).

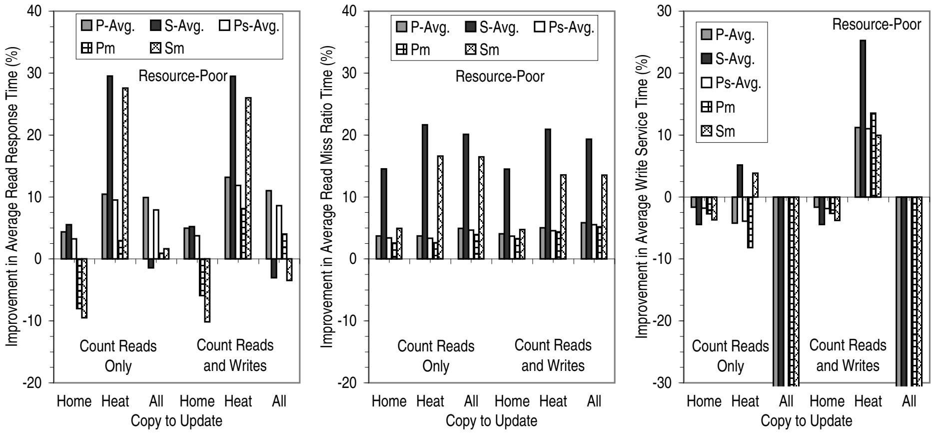


Figure D.27: Effect of Various Write Policies on Heat Clustering (Resource-Poor).

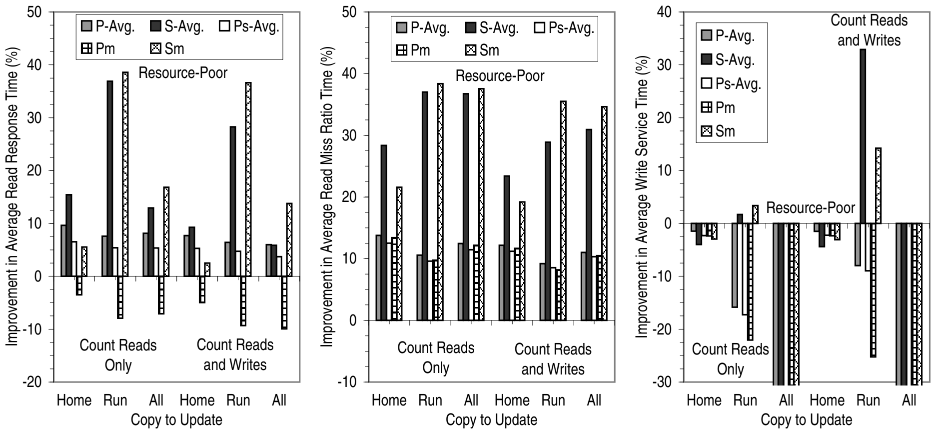


Figure D.28: Effect of Various Write Policies on Run Clustering (Resource-Poor).

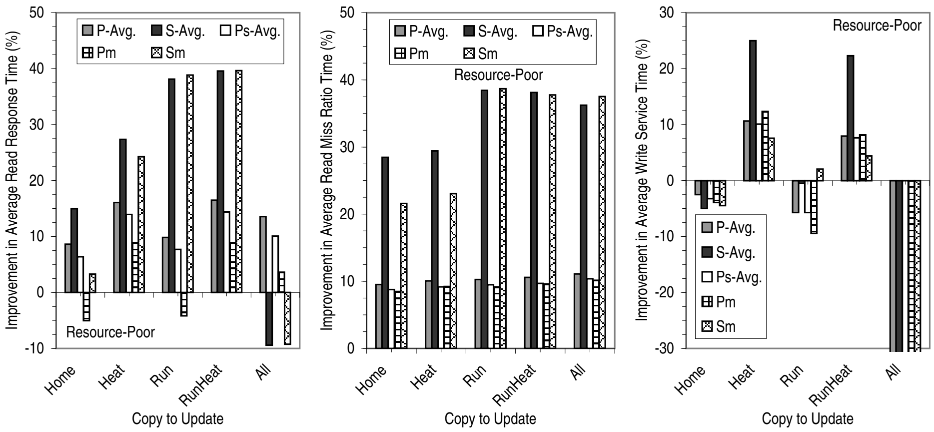
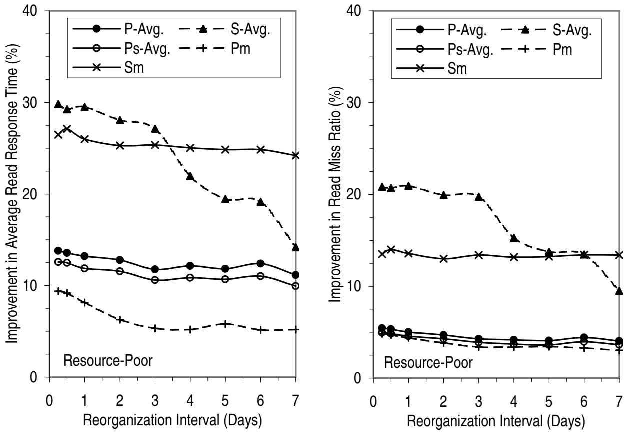
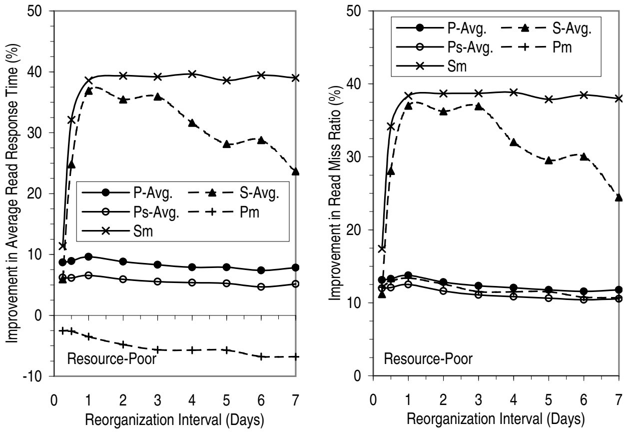


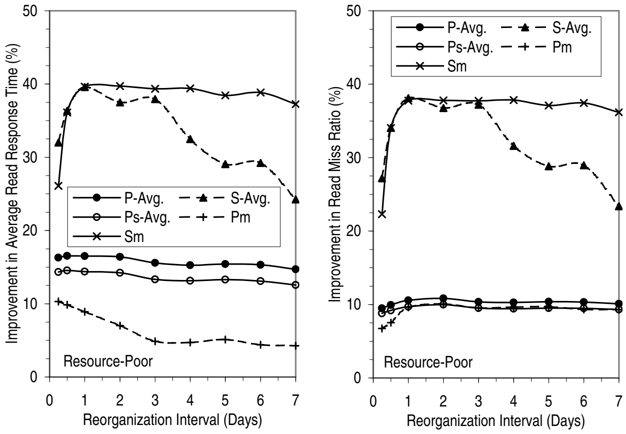
Figure D.29: Effect of Various Write Policies on Heat and Run Clustering Combined (Resource-Poor).



(a) Heat Clustering.

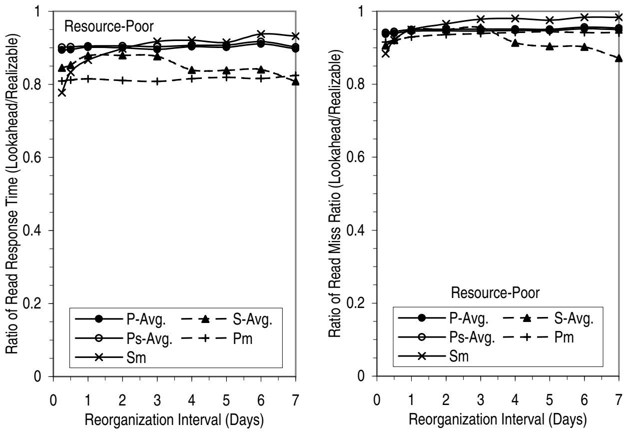


(b) Run Clustering.

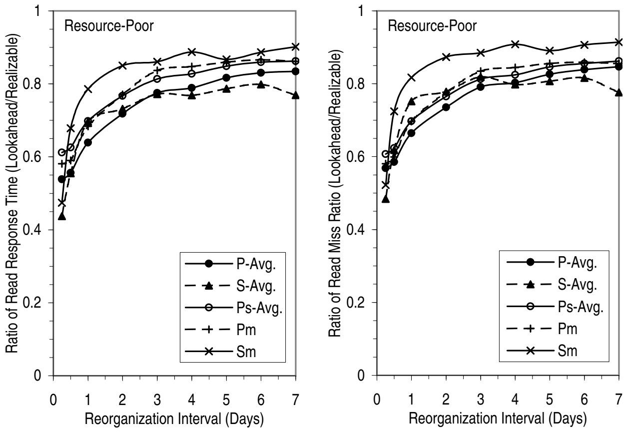


(c) Heat and Run Clustering Combined.

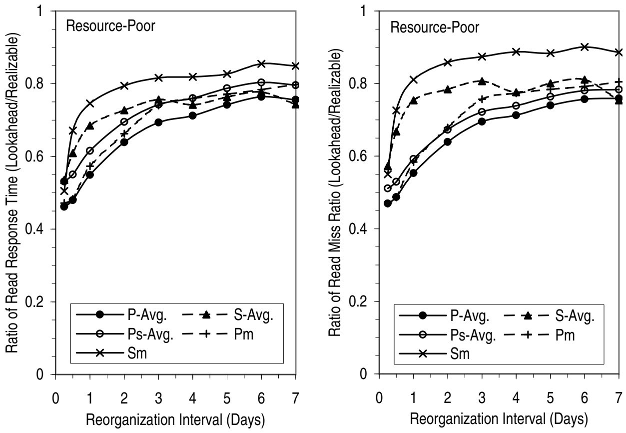
Figure D.30: Sensitivity to Reorganization Interval (Resource-Poor).



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure D.31: Performance with Knowledge of Future Reference Patterns (Resource-Poor).

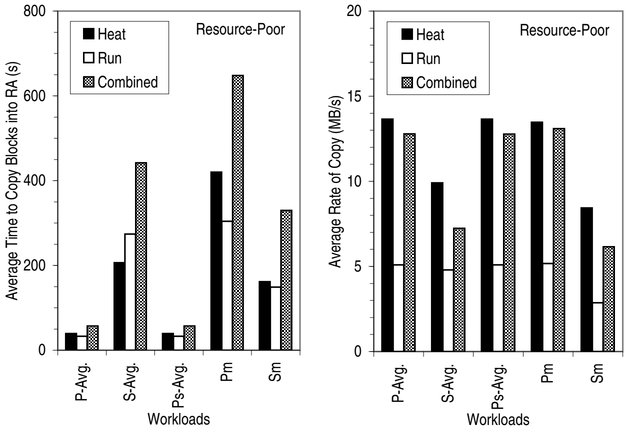
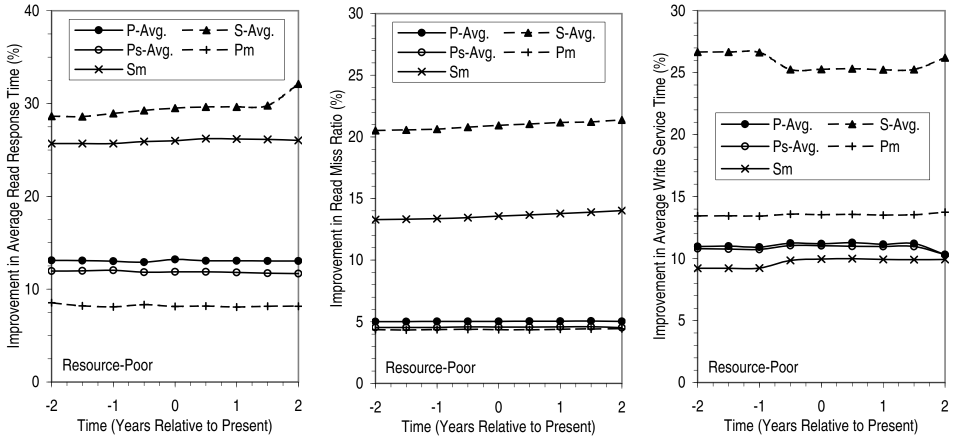
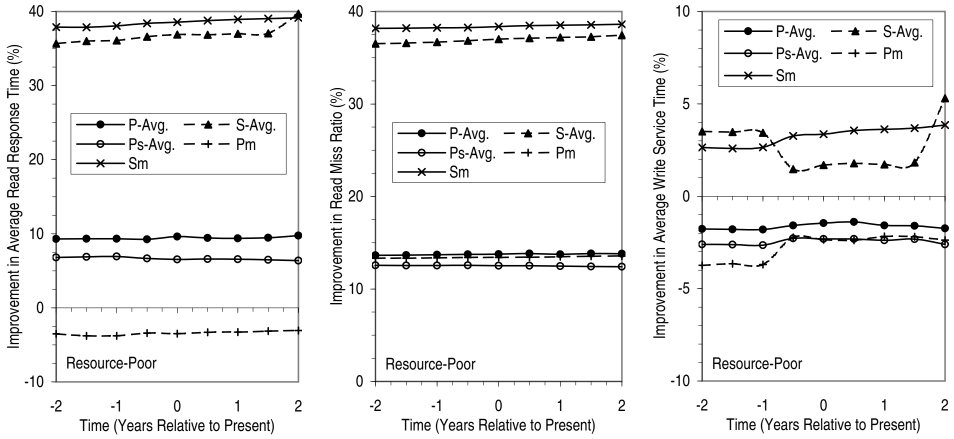


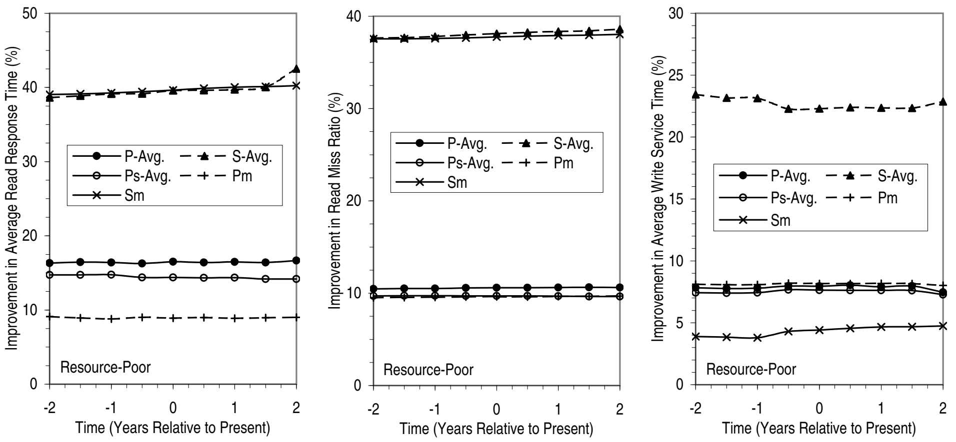
Figure D.32: Rate of Copying Blocks into the Reorganized Region (Resource-Poor).



(a) Heat Clustering.

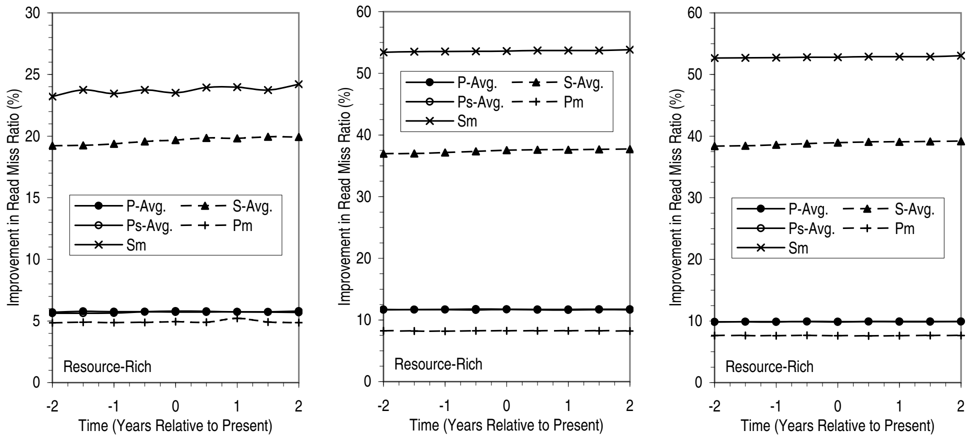


(b) Run Clustering.



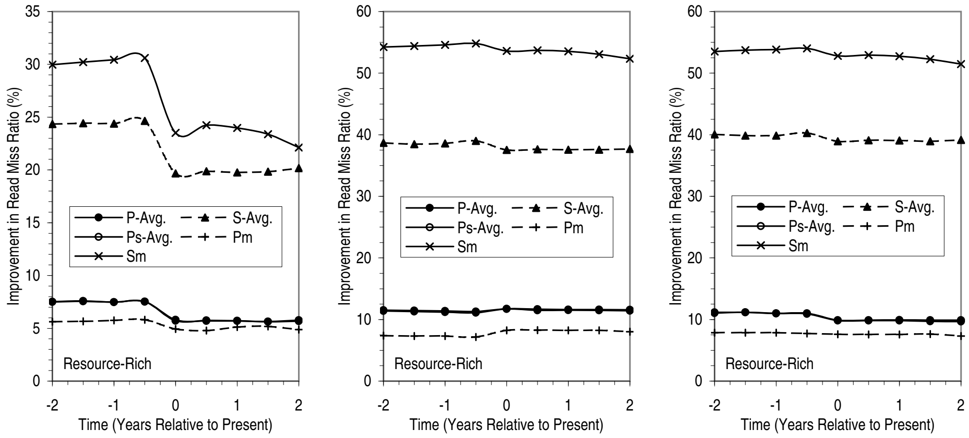
(c) Heat and Run Clustering Combined.

Figure D.33: Effectiveness of the Various Clustering Techniques as Disks are Mechanically Improved over Time (Resource-Poor).



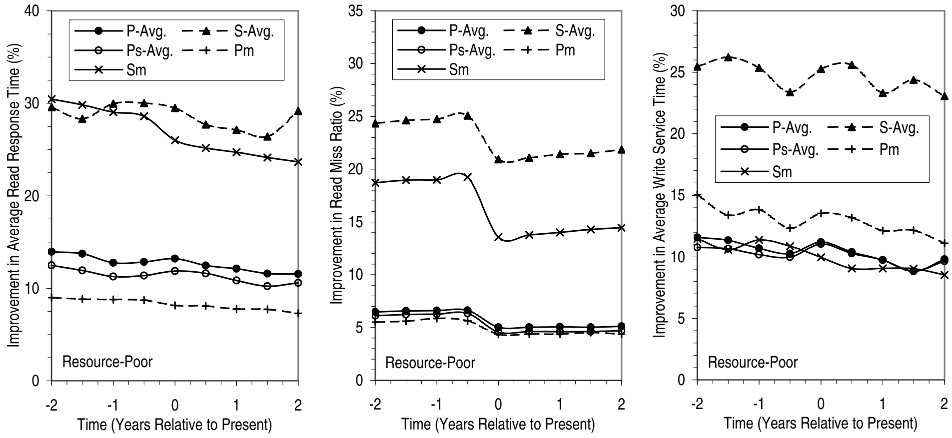
(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

Figure D.34: Effectiveness of the Various Clustering Techniques at Reducing Read Miss Ratio as Disks are Mechanically Improved over Time (Resource-Rich).

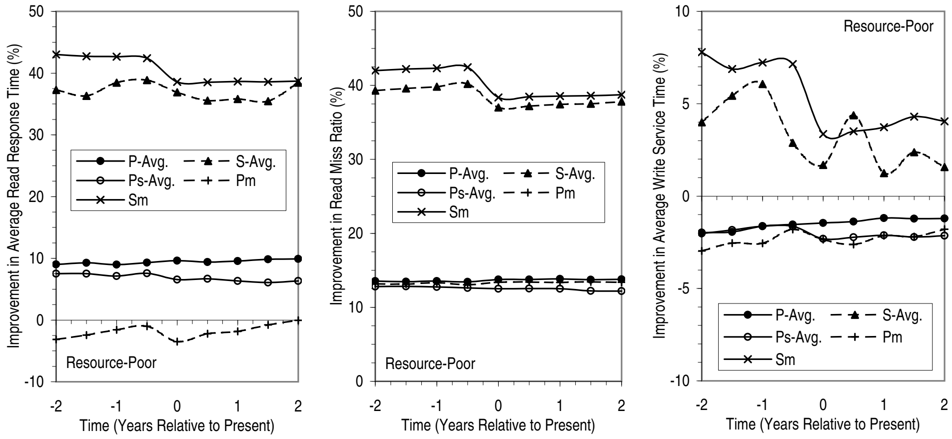


(a) Heat Clustering. (b) Run Clustering. (c) Heat and Run Clustering Combined.

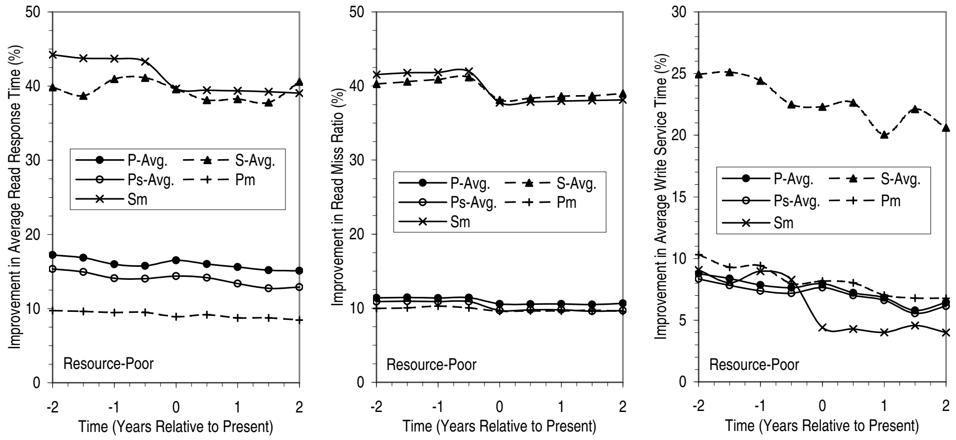
Figure D.35: Effectiveness of the Various Clustering Techniques at Reducing Read Miss Ratio as Disk Recording Density is Increased over Time (Resource-Rich).



(a) Heat Clustering.

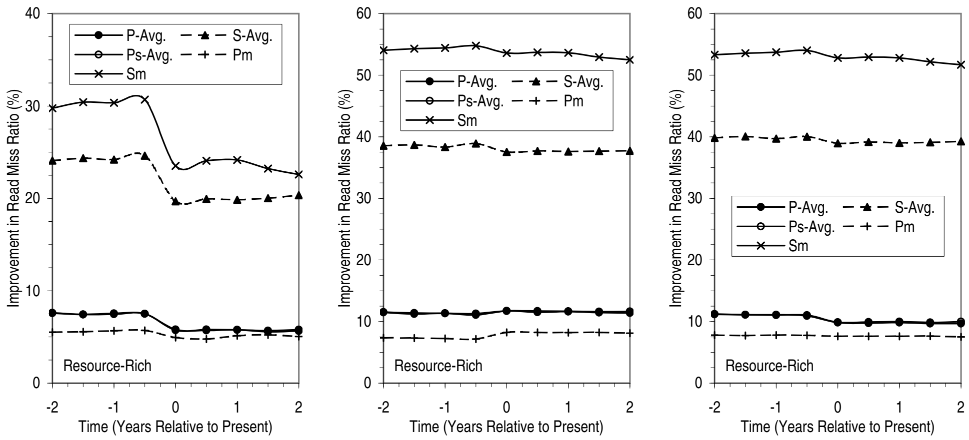


(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure D.36: Effectiveness of the Various Clustering Techniques as Disk Recording Density is Increased over Time (Resource-Poor).

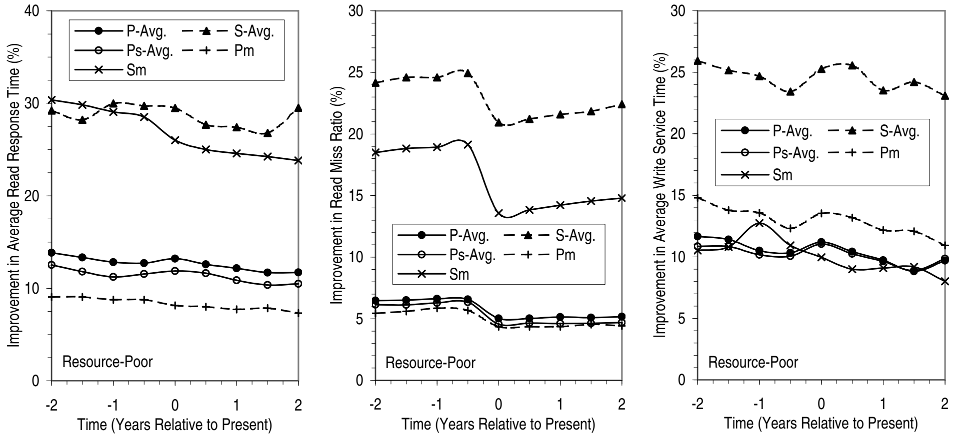


(a) Heat Clustering.

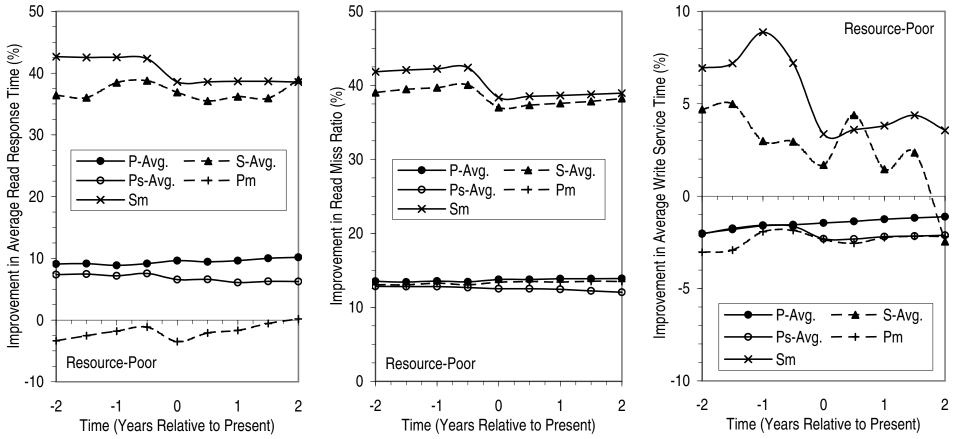
(b) Run Clustering.

(c) Heat and Run Clustering Combined.

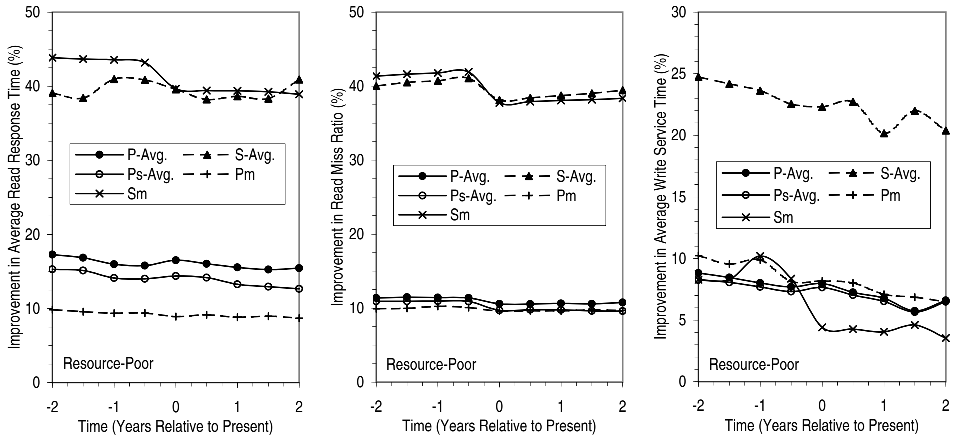
Figure D.37: Effectiveness of the Various Clustering Techniques at Reducing Read Miss Ratio as Disk Technology Evolves over Time (Resource-Rich).



(a) Heat Clustering.



(b) Run Clustering.



(c) Heat and Run Clustering Combined.

Figure D.38: Effectiveness of the Various Clustering Techniques as Disk Technology Evolves over Time (Resource-Poor).

Appendix E

Overview of the TPC-D Benchmark

The Transaction Processing Performance Council Benchmark D (TPC-D) [Tra97] is a decision support benchmark that models the analysis end of the business environment where trends are analyzed and refined to support sound business decisions. It consists of eight relations, 17 read-only queries and two update functions. The 17 read-only queries have different complexities, varying from single table aggregation (*e.g.,* Query 1) to 8-way join (*e.g.,* Query 2).

Eight *scale factors* (SF) are defined – 1, 10, 30, 100, 300, 1,000, 3,000, and 10,000. The scale factor is approximately the logical database size measured in GBs. Each benchmark configuration may define different indices. With index and database storage overhead (*e.g.,* free space), the actual database size may be much bigger than the logical database size defined by the benchmark. Only results measured against the same scale factor are comparable.

TPC-D introduces two performance metrics and a single price-performance metric. They are the TPC-D *power metric* (QppD@Size), TPC-D *throughput metric* (QthD@Size) and TPC-D *price/performance metric* (Price-per-QphD@Size). The power metric is defined as follows:

$$QppD@Size = \frac{3600}{\sqrt[19]{(RI(1) * RI(2) * ... * RI(17) * UI(1) * UI(2))}} * SF \tag{E.1}$$

where

- RI(i) = MAX(QI(i), $\frac{1}{1000}$ MAXQI).
- QI(i) is the run time, in seconds, of query i during the power test.
- MAXQI = MAX(QI(1), QI(2), ..., QI(17)).
- UI(j) is the run time, in seconds, of update function j during the power test.

- Size is the database size chosen for the measurement and SF, the corresponding scale factor.

The power test runs one query at a time in the order defined by the benchmark. The 3600 translates QppD to a query per hour measurement. Since QppD is a geometric mean of query rates, each query or update function has an equal weight. If the performance of any query or update function is improved by a factor of 2, the QppD@Size measurement will be increased by about 3.7%. If a system's execution time scales linearly with SF, QppD at any database size will be the same.

In the throughput test, one or more query streams are run concurrently on the system. The throughput metric is defined as follows:

$$\textit{QthD@Size} = \frac{S * 17 * 3600}{T_s} * SF \tag{E.2}$$

where

- S is the number of query streams used in the throughput test.
- T_s is the interval, in seconds, between when the query streams are started and when the last query stream completes.
- Size is the same as in the definition of QppD.

Notice that QthD@Size is based on the arithmetic mean of the query execution times. Thus queries with longer execution times have more weight in the metric.

The TPC-D power metric and the TPC-D throughput metric are combined to form a composite query-per-hour rating, QphD@Size, which is the geometric mean of QppD@Size and QthD@Size. Finally, the price/performance metric is defined as:

$$\textit{Price-per-QphD@Size} = \frac{\$}{\textit{QphD@Size}} \tag{E.3}$$

Appendix F

Query Execution Plans for TPC-D

In this appendix, we show the query execution plans of all 17 TPC-D queries from a recently certified TPC-D result on an SMP system. In each execution plan, sub-trees rooted by rectangular boxes are what we call SR sub-trees; all operations in a SR sub-tree are single-base-relation operations that can be delegated to a SmartSTOR by single-relation offloading. Each query execution plan tree is rooted by a *return* operation, which returns the qualified tuples to the application. We use a double-circle for tables and indices. These include base tables, base indices, and optimizer generated subqueries and table functions. The legend is as follows:

- AST: automatic summary table
  ASTs are auxiliary tables that contain partially aggregated data.
- FETCH: table scan through index
- FILTER: predicate evaluation
  This is for the predicates that are not pushed down to the scans.
- [HS|MS|NL]JOIN: [hash|merge-scan|nested-loop] join
- [IX|TB]SCAN: [index|table] scan
  IXSCAN is different from FETCH in that the former is an index-only scan, *i.e.,* the corresponding table does not have to be read to get the needed fields.
- L[M]TQ: local [merge] table queue
  Table queue is a mechanism to exchange data among operations. The plans are from an SMP system, thus, all table queues are local table queues. Regular LTQ collects data in any order while LMTQ collects data in a specific order.
- GRPBY: group by
- RETURN: return to host
- SORT: sort

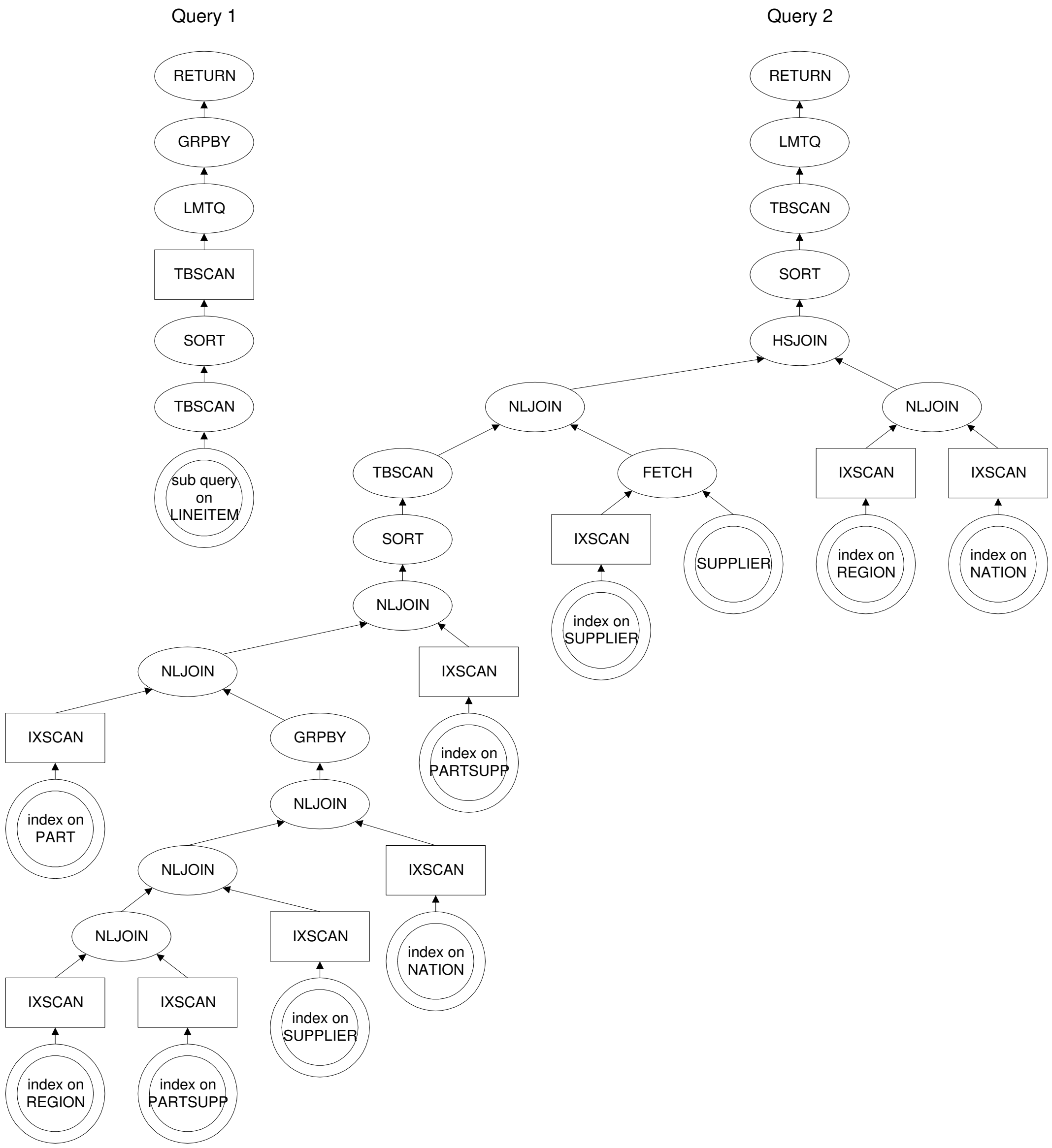


Figure F.1: Execution Plans for Queries 1 and 2.

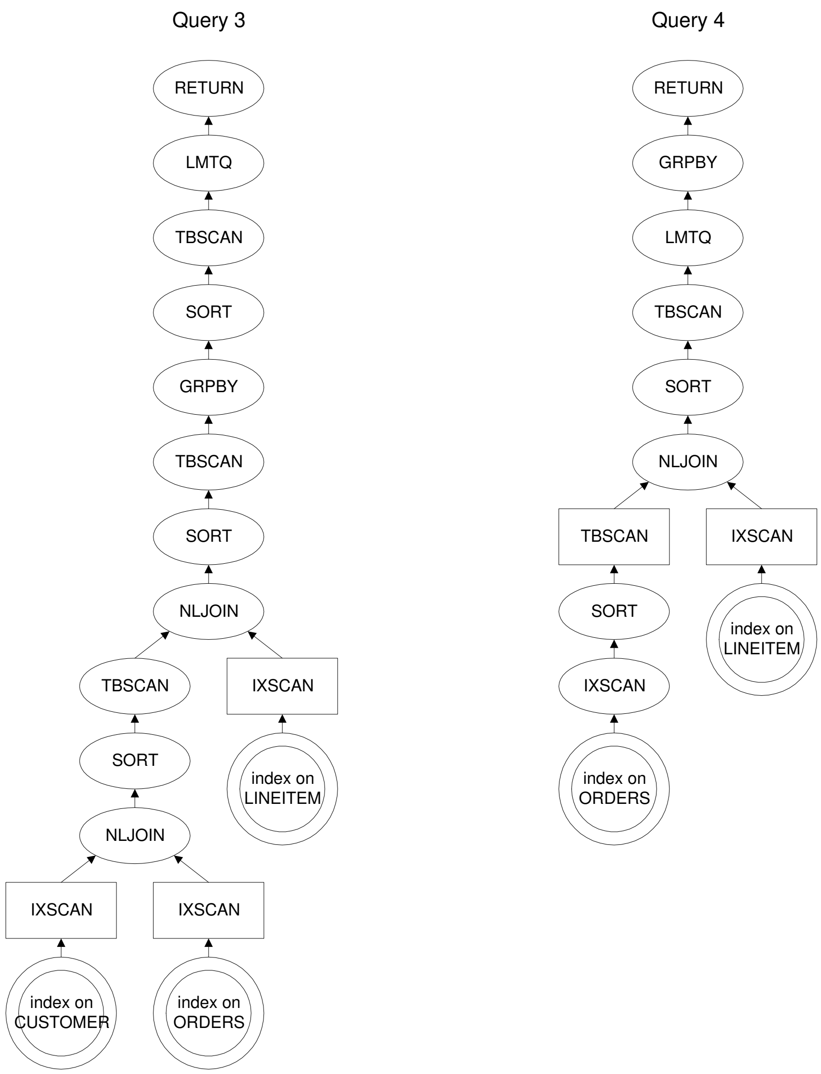


Figure F.2: Execution Plans for Queries 3 and 4.

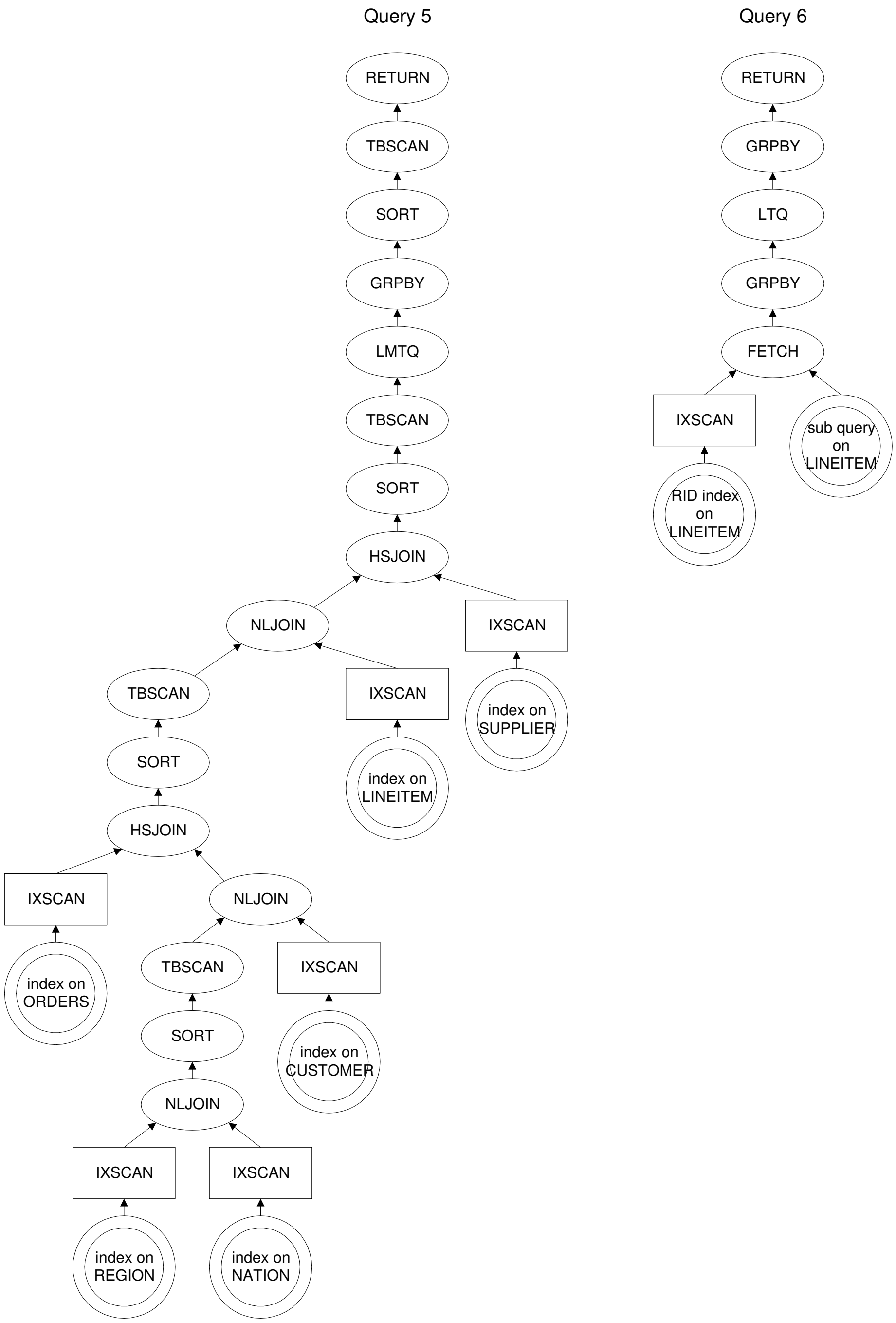


Figure F.3: Execution Plans for Queries 5 and 6.

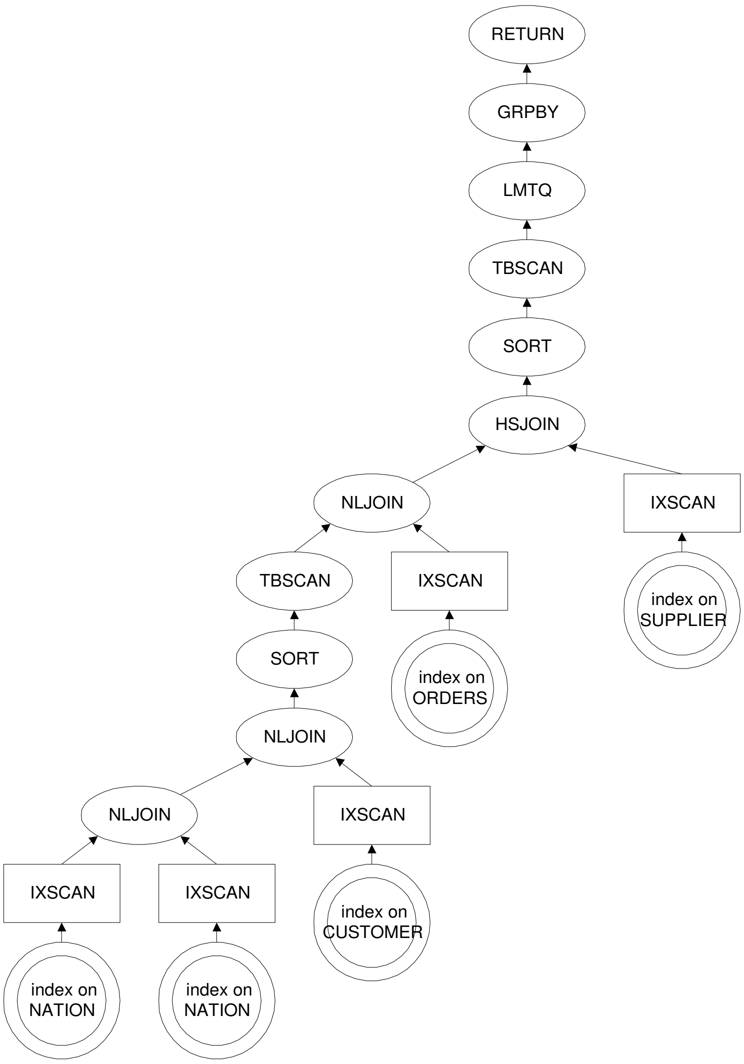


Figure F.4: Execution Plan for Query 7.

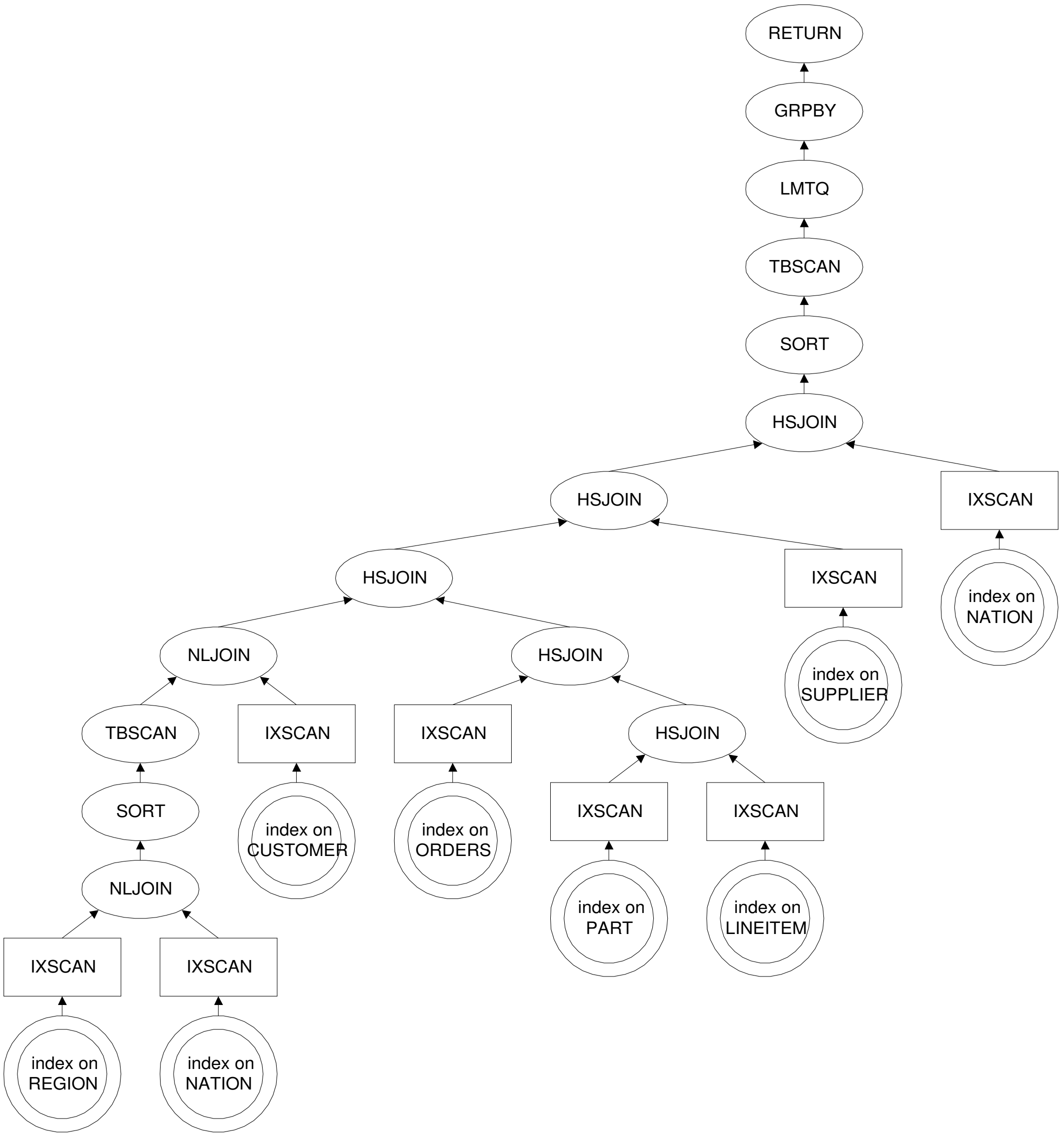


Figure F.5: Execution Plan for Query 8.

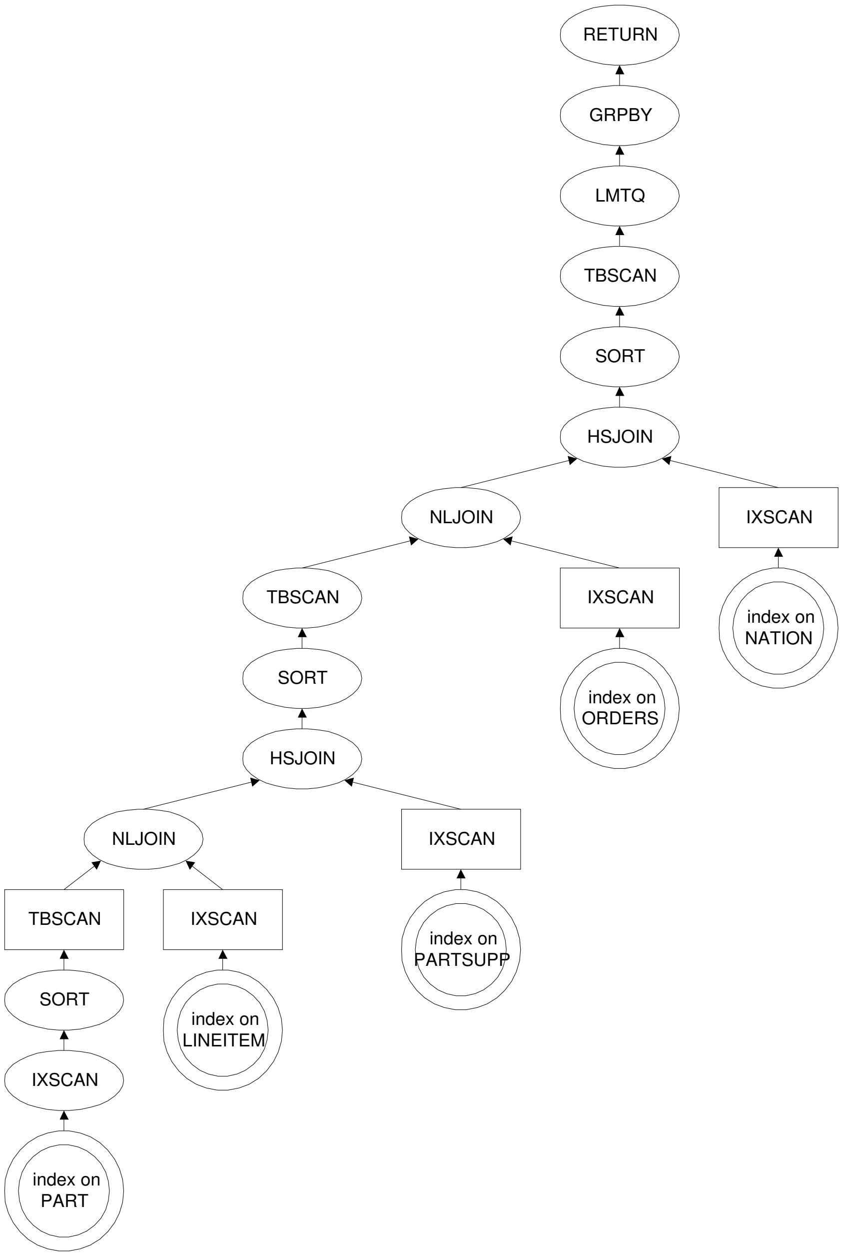


Figure F.6: Execution Plan for Query 9.

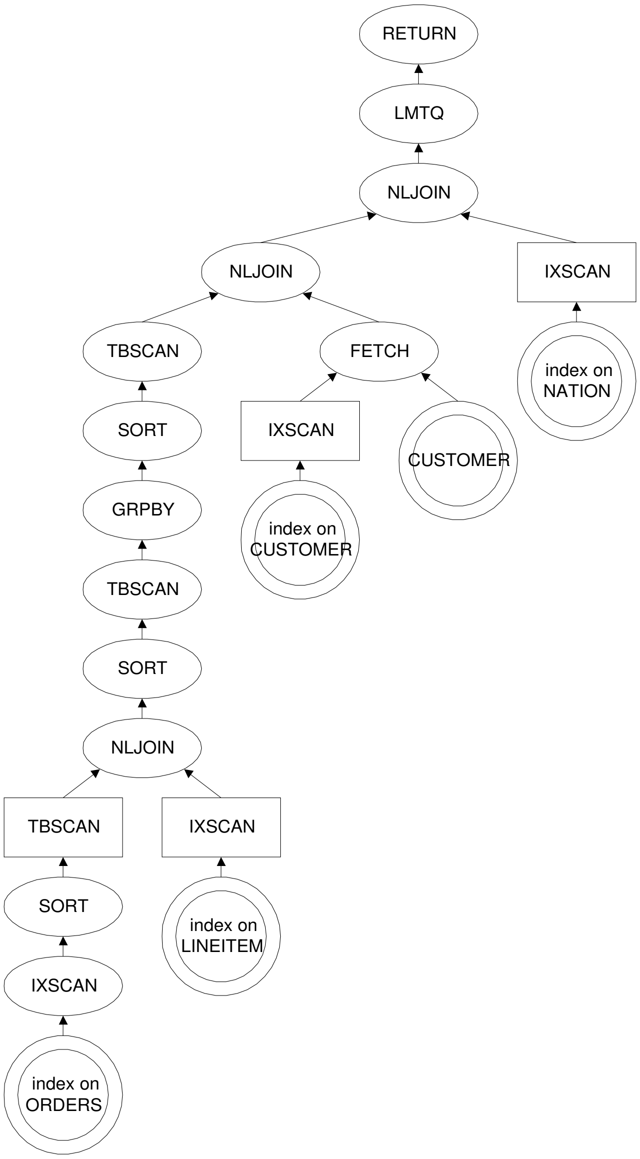


Figure F.7: Execution Plan for Query 10.

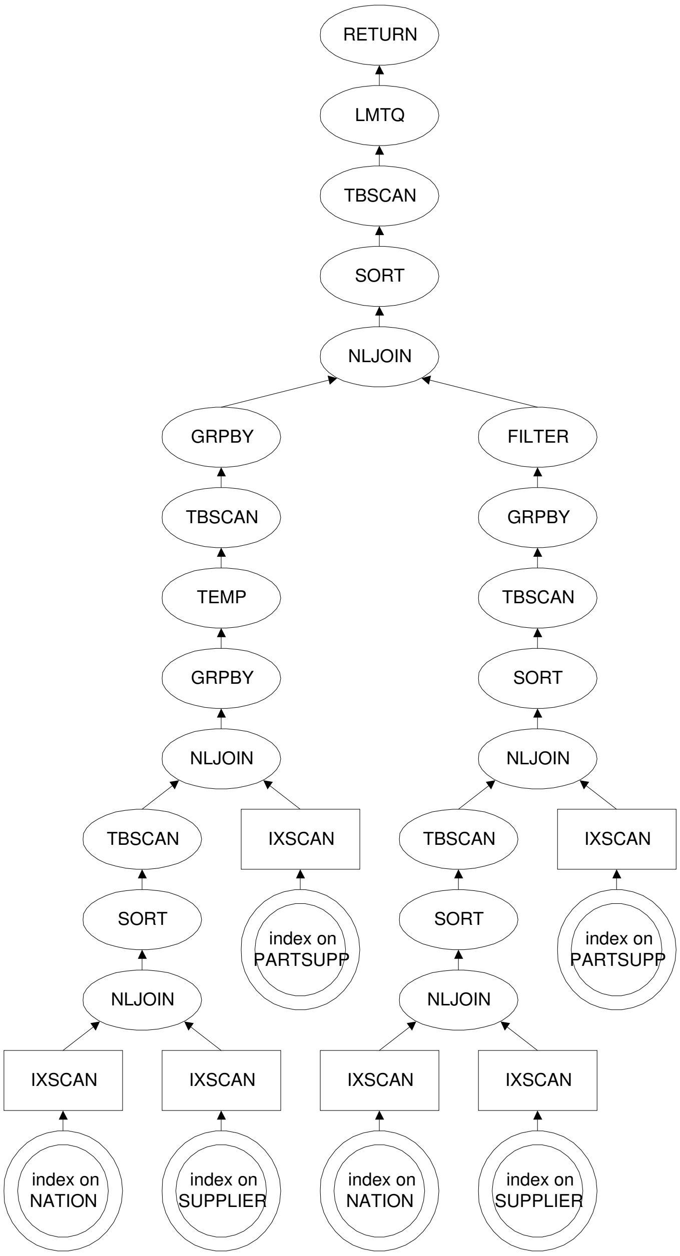


Figure F.8: Execution Plan for Query 11.

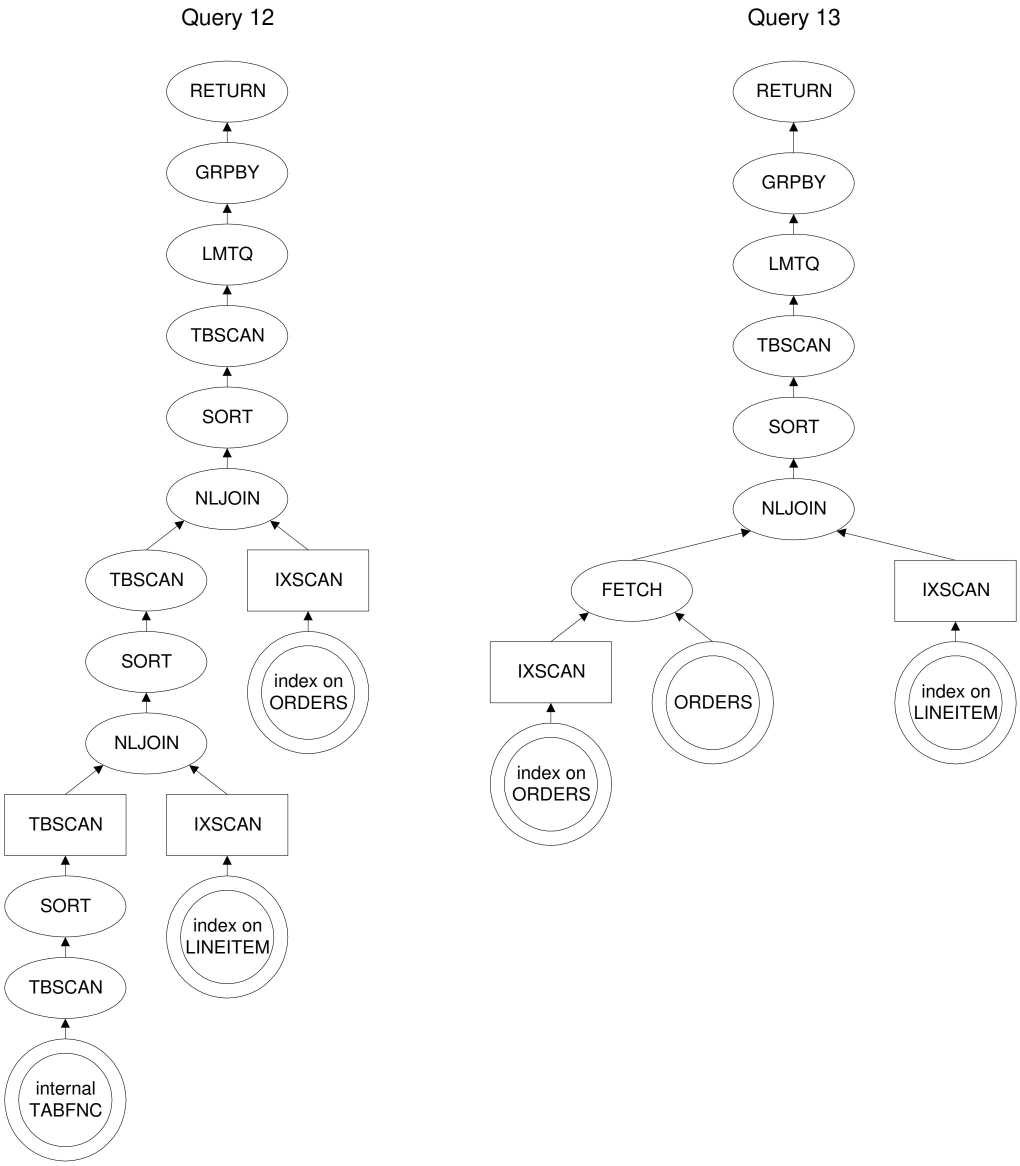


Figure F.9: Execution Plans for Queries 12 and 13.

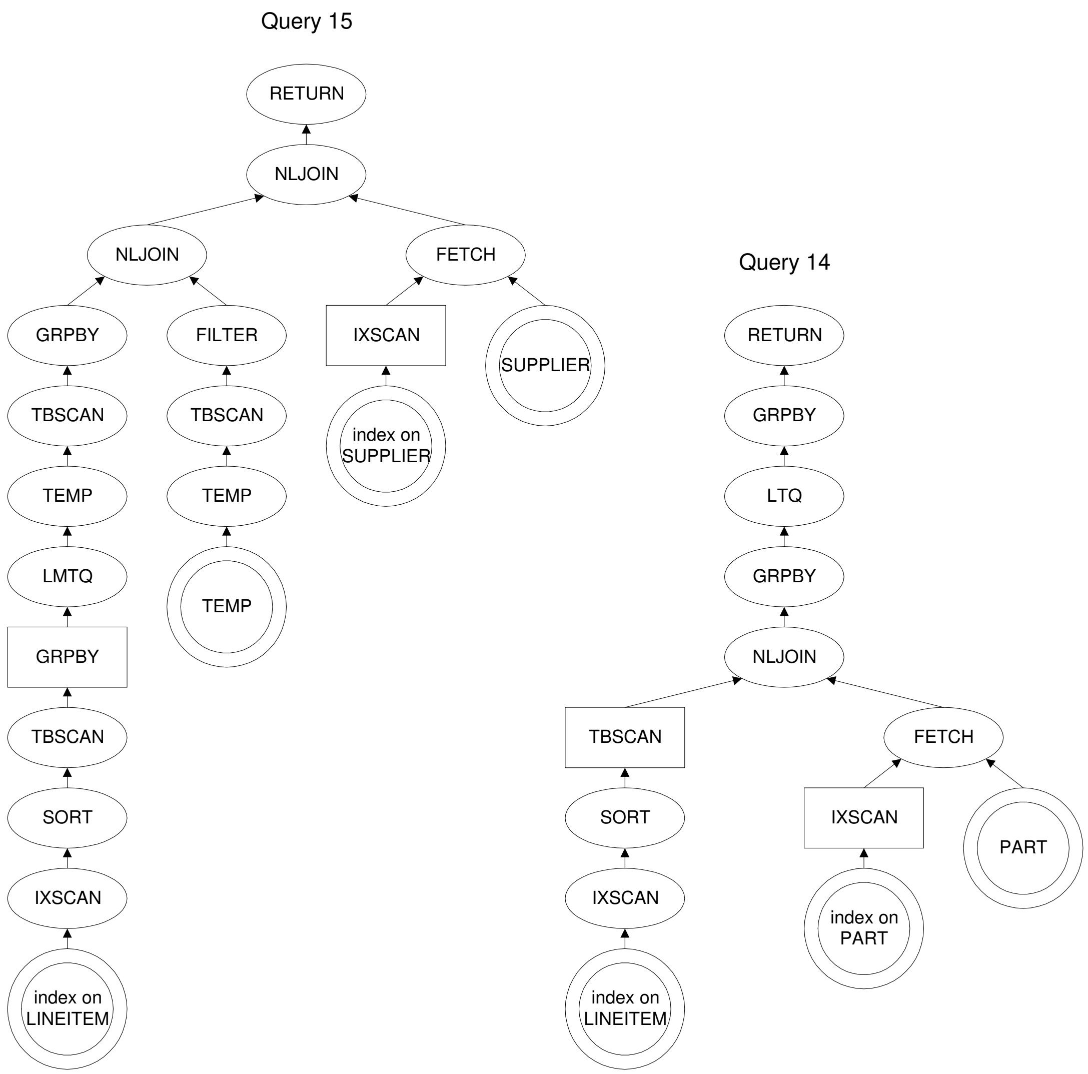


Figure F.10: Execution Plans for Queries 14 and 15.

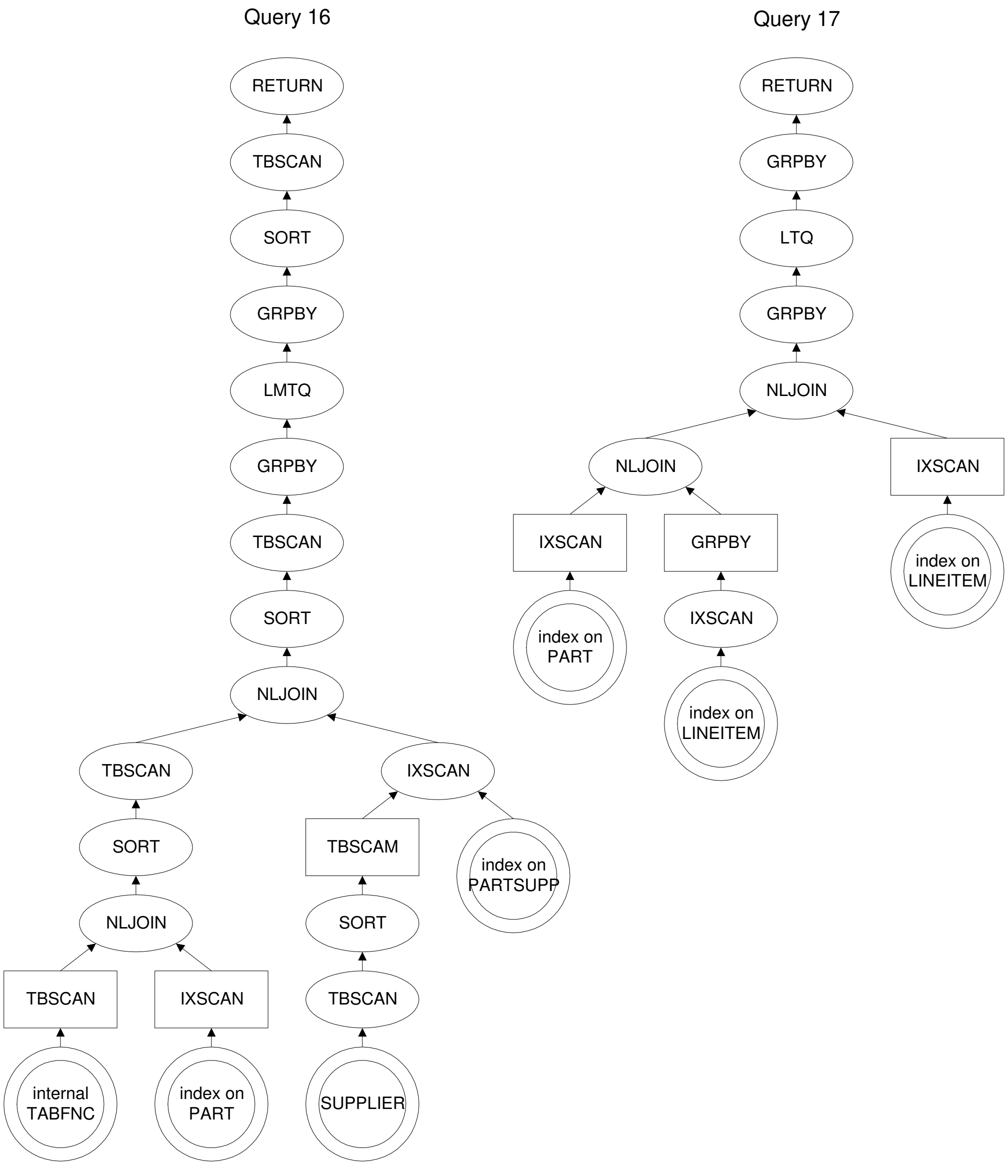


Figure F.11: Execution Plans for Queries 16 and 17.