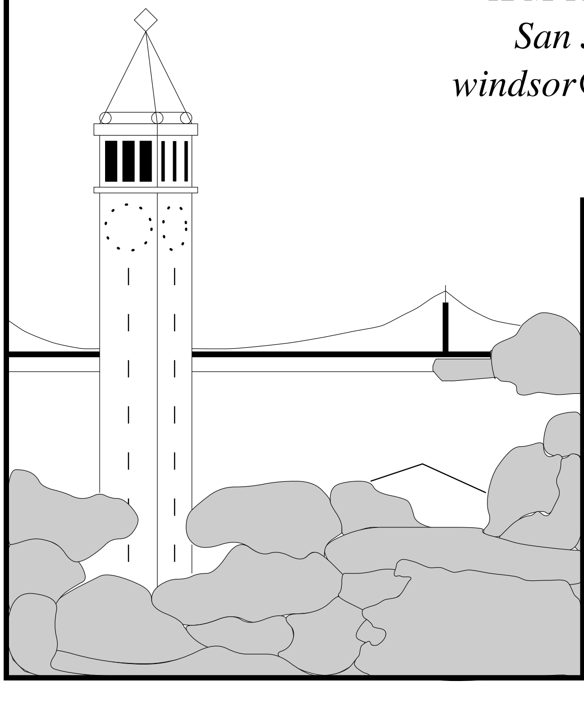
The Real Effect of I/O Optimizations and Disk Improvements

Windsor W. Hsu[†‡]
Alan Jay Smith[†]

[†]*Computer Science Division*
EECS Department
University of California
Berkeley, CA 94720
{windsorh,smith}@cs.berkeley.edu

[‡]*Storage Systems Department*
Almaden Research Center
IBM Research Division
San Jose, CA 95120
windsor@almaden.ibm.com



Report No. UCB/CSD-03-1263

July 2003

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

The Real Effect of I/O Optimizations and Disk Improvements

Windsor W. Hsu[†‡] Alan Jay Smith[†]

[†]Computer Science Division
EECS Department
University of California
Berkeley, CA 94720
{windsorh,smith}@cs.berkeley.edu

[‡]Storage Systems Department
Almaden Research Center
IBM Research Division
San Jose, CA 95120
windsor@almaden.ibm.com

Abstract

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 difficult to relate the various physical improvements to the actual performance experienced by real workloads. In this paper, we use an assortment of real server and personal computer workloads to systematically analyze the various optimization techniques and technology improvements 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 – faster seeks, higher RPM, linear density improvement and increase in track density – and analyze each separately to determine its actual benefit. In addition, we examine the 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 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 performance, reducing the average read and write response time by about 50% and 90% 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.

Funding for this research has been provided by the State of California under the MICRO program, and by AT&T Laboratories, Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation and Veritas Software Corporation.

1 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 [14, 29], 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 paper, 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. In this paper, we break down the steady improvement in disk technology into four major basic effects – seek time reduction due to actuator improvement, spin rate increase, linear density improvement and increase in track density – 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 a companion paper [17], we analyze 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 the current paper is self-contained, readers are encouraged to also read the companion paper 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) [19], 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. In fact, the results we derive here serve as the baseline for the analysis of ALIS in [19]. Therefore, this paper has an emphasis on the optimizations that directly affect ALIS, in particular, the prefetching.

The rest of this paper is organized as follows. Section 2 contains a brief overview of previous work in evaluating I/O optimization techniques. Section 3 discusses our methodology and describes the traces that we use. In Section 4, we analyze the effect of the various optimization techniques. In Section 5, we consider the real impact of disk technology improvement over time. Section 6 concludes and summarizes this paper. 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 the Appendix.

2 Related Work

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 [38, 44], prefetching in [13, 36], write buffering in [1, 42], request scheduling in [23, 35, 43] and striping in [4, 5]. At the logical level, caching, prefetching and write buffering are well covered in [18, 27]. 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.,* [30]). 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 [37] for a comprehensive survey of early work on I/O optimization.

3 Methodology

The methodology used in this paper is trace-driven simulation [39, 41]. 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. As described below, we create and use a new method for replaying I/O traces that more accurately models the timing of I/O arrivals and that allows the I/O rate to be more realistically scaled (*e.g.,* when processor power is increased) than previous practice.

3.1 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 [10] 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.,* [17]) to have bursty I/O traffic patterns.

In the open model, requests arrive at predetermined times (*e.g.,* traced time in [33] and traced inter-arrival time scaled by a constant factor in [43]), independent of the performance of the storage system. Such models assume that the workload consists exclusively of time-noncritical requests [10] 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 [17] and 52-74% in UNIX workloads [34]. 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.

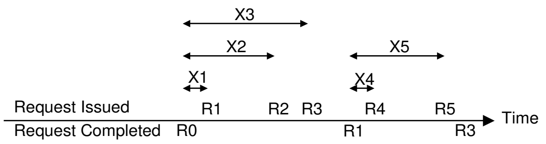


Figure 1: Intervals between Issuance of I/O Requests and Most Recent Request Completion.

Results in [17] 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. For instance, in the timeline in Figure 1, 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. 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.,* [10]). In the limit, we can run the workloads on a system simulator where we have control over the timing of events [32] or on a virtual machine [2] or on a real system with a timing-accurate storage emulator [12]. 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?

3.2 Workloads and Traces

The traces analyzed in this study were collected from both server and PC systems running real user workloads on three different platforms – Windows NT, IBM AIX and HP-UX. All of them were collected downstream of the database buffer pool and the file system cache. Thus these are *real* I/O traces, not logical ones. The PC traces were collected by using VTrace [24], a software tracing tool for Intel x86 PCs running Windows NT/2000. In this study, we are primarily interested in 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. Both the IBM AIX and HP-UX traces were collected using kernel-level trace facilities built into the respective operating systems. 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.

The PC traces were collected from the primary systems of a wide-variety of users, including engineers, graduate students, 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 or environment. Despite this disclaimer, the fact that many of their characteristics correspond to those obtained previously (see [17]), albeit in somewhat different environments, suggest that our findings are to a large extent generalizable. Table 1(a) summarizes the characteristics of these traces. We denote the PC traces as P1, P2, ..., P14 and the arithmetic mean of their results as P-Avg. As detailed in [17], the PC traces contain only I/Os that occur when the user is actively interacting with the system. Specifically, we consider the system to be idle from ten minutes after the last user keyboard or mouse activity until the next such user action, and we assume that there is no I/O activity during the idle periods. 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, and are not likely to be noticed by the user.

The servers traced include two file servers, a timesharing system and a database server. The characteristics of these traces are summarized in Table 1(b). Throughout this paper, we use the term S-Avg. to denote the arithmetic mean of the results for these server workloads. The first file server trace (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 [34]. The trace denoted TS1 was gathered on a timesharing 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 [34]. 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 system. This trace is only seven days long and the first three days are used to warm up the simulator. More details about the traces and how they were collected can be found in [17].

In addition to these base workloads, we scale up the traces to obtain workloads that are more intense. Results reported in [17] 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 paper, 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.

Designation	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)	R/W Ratio
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	2.51
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	1.37
P3	Engineer	450MHz P6	128	4, 2GB NTFS	6	1	45 days (7/26/99 - 9/8/99)	0.708	5.01	0.679	0.429
P4	Engineer	450MHz P6	128	3, 3GB NTFS	6	1	29 days (7/27/99 - 8/24/99)	4.72	26.6	2.56	0.606
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	0.338
P6	Manager	166MHz P6	128	3, 2GB NTFS	5	2	45 days (7/23/99 - 9/5/99)	0.513	2.43	0.324	0.147
P7	Engineer	266MHz P6	192	4GB NTFS	4	1	45 days (7/26/99 - 9/8/99)	1.84	20.1	2.27	0.288
P8	Secretary	300MHz P5	64	1, 3GB NTFS	4	1	45 days (7/27/99 - 9/9/99)	0.519	9.52	1.15	1.23
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	0.925
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	0.937
P11	Director	350MHz P6	64	2, 2GB NTFS	4	1	45 days (8/25/99 – 10/8/99)	0.73	11.4	1.58	0.831
P12	Director	400MHz P6	128	2, 4GB NTFS	6	1	45 days (9/10/99 – 10/24/99)	1.36	6.2	0.514	0.758
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	0.566
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	0.481
P-Avg.	-	318MHz	109	-	5.07	1.43	41.2 days	1.59	13.9	1.67	0.816

(a) Personal Systems.

Designation	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)	R/W Ratio
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	0.718
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	0.794
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	0.607
S-Avg.	-	-	299	-	18.5	8	32.3 days	4.22	74.6	12.1	0.706

[i] Sum of all the file systems and allocated volumes.
[ii] Amount of data referenced at least once
[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

(b) Servers.

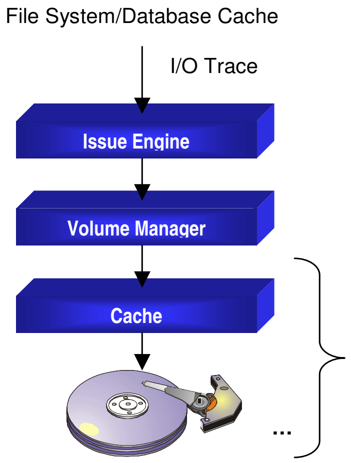
Table 1: Trace Description.

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 paper, 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.

3.3 Simulation Model

The major components of our simulation model are presented in Figure 2. 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 example, we model only a single level of cache instead of a disk drive cache, an adaptor cache, a controller cache, *etc.* This approach does not expose the interference that occurs when the different levels in the



	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 per disk, 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.
Parallel I/O	Stripe unit of 2MB.	Stripe unit of 2MB.

Figure 2: 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 4.

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 paper. 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 drive 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 paper, 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 2. 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 [26]. It is based upon a detailed model of the mechanical components of the IBM Ultrastar 73LZX [21] family of disks that is used in disk development and that has been validated against test measure-

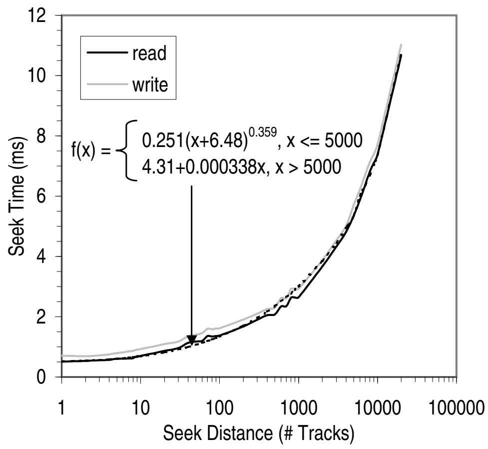


Figure 3: Seek Profile for the IBM Ultrastar 73LZX Family of Disks.

ments obtained on several batches of the disk. The level of detail in this model is similar to that in the publicly available DiskSim package [11]. However, 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, 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 160 KB to 340 KB. More details about the specifications of this family of disks can be found in [21]. In order to understand the effect of disk technology evolution, in the later part of this paper, 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 [19]. 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 [17] show that there is not a lot of interaction between the reads and the writes.

3.4 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 paper, 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. This means that the reciprocal of the service time will tend to be an optimistic estimate of the maximum throughput, especially in the case of a lightly loaded disk where opportunistic techniques are likely to have a bigger effect.

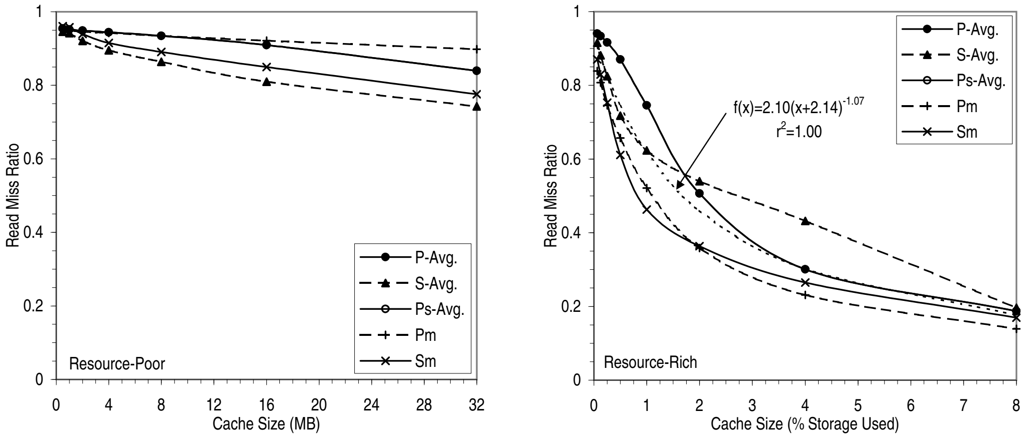
To gain insight into the workings of the different optimization techniques, we also examine the effective *miss ratio* of the read cache and the write buffer. The miss ratio is generally defined as the fraction of I/O requests that are not satisfied by the cache or buffer, or in other words, the fraction 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 [31].

4 Effect of I/O Optimizations

4.1 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 paper, 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 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 2.

Figure 4 shows the effectiveness of read caching at reducing physical reads. Unless otherwise noted, the cache block size is 4 KB. We use the Least-Recently-Used (LRU) replacement policy since variations of it are commonly used throughout computer systems. No-



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

Figure 4: Effectiveness of Read Caching at Reducing Physical Reads.

tice from Figure 4(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, much 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 drive cache is not very effective. It serves primarily as a buffer for prefetching.

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 4(b) the miss ratio 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 due to addressing and packaging limitations, and cost. 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 [8, 16, 22]. 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. 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. This level of cost, though high, is likely to be acceptable since it is about half that incurred by sites that mirror instead of parity-protect their disks. Note also that as disks become a lot bigger and PCs have at least one disk, the amount of cache needed in the PC environment to hold 1% of the data stored (or data in use) may be much less than the amount of cache needed to store 1% of the disk capacity.

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 4(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 [18]. However, at the logical level, the c value is about 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.

In Table 2, we summarize the effectiveness of read caching at improving performance. Throughout this paper, 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 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 2 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.

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

[i] Improvement over 512KB cache (buffer) ([original value – new value]/[original value]).
Resource-Poor: 8MB per disk, Least-Recently-Used (LRU) replacement.
Resource-Rich: 1% of storage used, cently-Used (LRU) replacement.

Table 2: Performance with Read Caching. Table shows percentage improvement over a system with practically no (512 KB) cache.

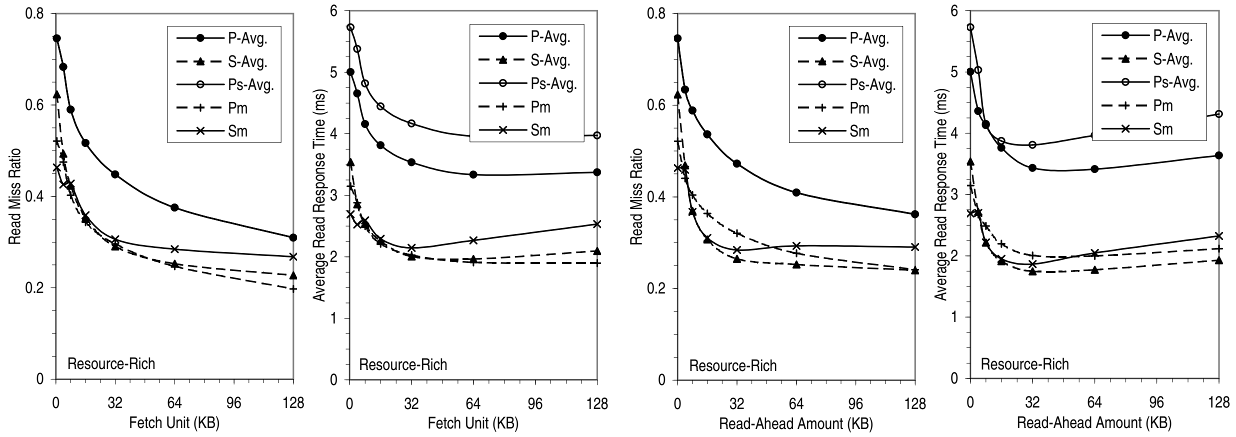
Note that these numbers are for a cache block size of 4 KB. For historical reasons, the sector or smallest addressable unit in most disks and storage controllers today is 512 B. Managing the cache at such a small granularity of 512 B is very inefficient because of the large data structures needed to manage them and because most I/O transfers are much larger than 512 B. To reduce the management overhead, a larger cache block can be used together with valid bits to indicate whether each sector within the block is present in the cache. This is similar to the sector cache approach in processor cache. In Figure A-1, 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 paper. 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.

4.2 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 at improving performance hinges on (1) the accuracy of the prediction, (2) the amount of extra resources (memory use, disk and data path busy time, *etc.*) 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 [18, 36] although in certain situations, system-generated plans [15, 40], user-disclosed hints [30] and guidance from speculative execution [3] 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. For many storage devices, particularly disk drives, however, 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 paper. By default, we assume that data is prefetched into the cache and managed as if it were demand fetched. The prefetched data could instead be placed in a separate buffer or be handled in the cache differently than demand fetched data (*e.g.,* be evicted earlier). The interested reader is referred to [18] 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.,* [13]). 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 look at an alternative to context-based prefetch in [19].



(a) Large Fetch Unit.

(b) Read-Ahead.

Figure 5: Effect of Large Fetch Unit and Read-Ahead on Read Miss Ratio and Response Time (Resource-Rich).

4.2.1 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. Thus a large fetch unit, effectively a large block size, will generally prefetch blocks both preceding and following the target blocks. Because the preceding blocks are fetched before the target blocks to avoid an extra disk revolution, there is a response time penalty for having a large fetch unit. Furthermore, the entire transfer must be complete before an I/O interrupt is received, although in an alternate design the fetch could be broken into one that terminated at the target blocks and a second one that obtained the remaining blocks.

In Figures 5(a) and A-2(a), 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 [38] 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.

4.2.2 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 Figures 5(b) and A-2(b), 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.

In Table 3, we summarize the effectiveness of the different prefetching schemes at improving performance over a non-prefetching system. 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 re-

		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 (64KB), RA: Read-Ahead (32KB), CSP: Conditional sequential prefetch (16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2).
[ii] Improvement over no prefetch ([original value – new value]/[original value]).

Table 3: Performance Improvement with Prefetching. Table shows percentage improvement over a system that does not prefetch.

quest, it performs worse than read-ahead in terms of response time, especially for the server workloads.

4.2.3 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 [18, 36] 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 paper, we use a general sequential detection scheme patterned after that proposed in [18].

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 detector maintains an LRU organized list of segments. Each entry in the segment directory has a sequential 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 paper, 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 6, A-4 and A-5, 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 could 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 A-6 and A-7, except for a slight performance improvement in some of the PC workloads,

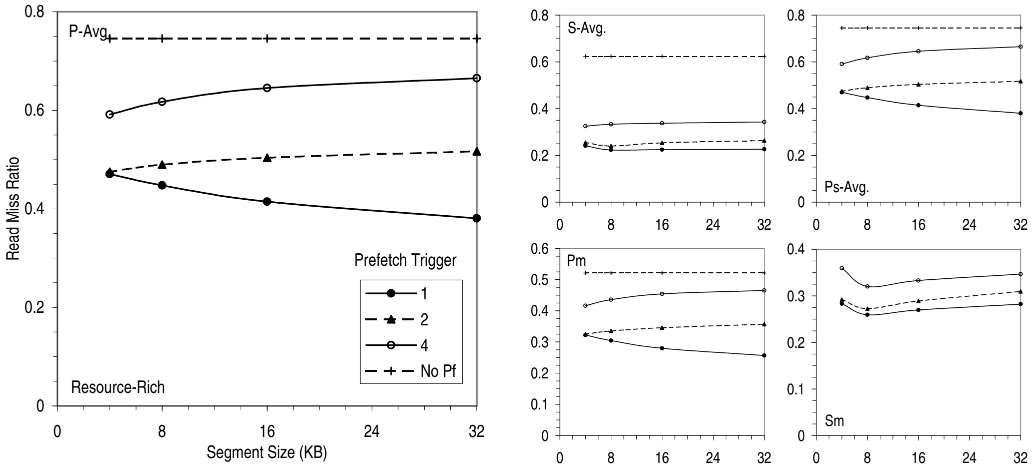


Figure 6: Read Miss Ratio with Conditional Sequential Prefetch (Resource-Rich).

backward conditional sequential prefetch turns out not to be very useful.

In Table 3, 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.

4.2.4 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 such an approach 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. Because a disk access costs much more than a semiconductor memory access, the cost of accessing prefetched data should be largely independent of the layer in the storage stack the data is prefetched into. However, data prefetched into the disk drive cache will tend to be evicted sooner, sometimes even before they are used, because the disk drive cache is typically smaller than the adaptor/controller cache. To model this effect, we enter opportunistically prefetched data into an 8 MB (LRU) prefetch buffer instead of the large cache in the resource-rich environment. The prefetch buffer turns out to significantly reduce pollution of the large cache.

The simplest form of opportunistic prefetch is to read-ahead up to a maximum amount or 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 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 (Figures 7 and A-8). 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. In such an approach, we would always perform some amount of prefetch (non-opportunistic), and if idle resources are available, we would prefetch more (opportunistic). We find that with the hybrid approach, an opportunistic prefetch limit of 128 KB works well in almost all the cases (Figures A-10 - A-15). This is the value that we will assume for the rest of the paper. 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.

Tables 4 and A-1 summarize the performance impact of performing preemptible read-ahead in addition to the various non-opportunistic prefetching schemes. In the resource-poor environment, preemptible read-ahead im-

		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 (64KB), RA: Read-Ahead (32KB), CSP: Conditional sequential prefetch (16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2).
[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 4: Additional Effect of Opportunistic Prefetch (Resource-Rich). Table shows percentage improvement over a system that performs only non-opportunistic prefetch.

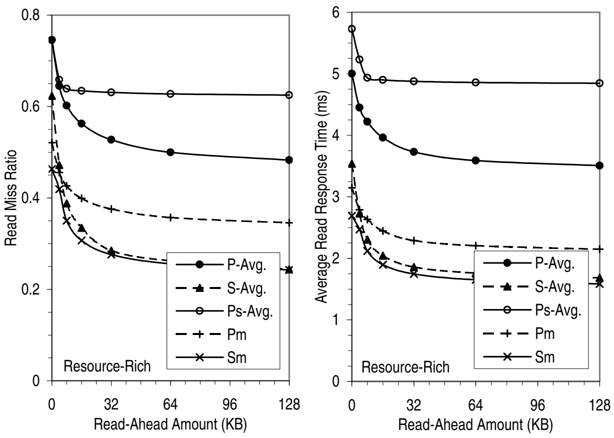


Figure 7: Effect of Preemptible Read-Ahead on Read Miss Ratio and Response Time (Resource-Rich).

proves 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%.

Another opportunistic prefetching technique is to start reading once the disk head is positioned over the correct track. 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. Such a scheme may prefetch some blocks that precede the requested data and/or some blocks that come after, depending on when the head is properly positioned. For example, if the head is positioned to read just after the requested data has rotated under, read any free blocks will fetch the succeeding blocks until the end of the track and then continue reading the blocks at the beginning of the track. As shown in Tables 4 and A-1, 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* [9]. 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

	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]
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

[i] LFU: Large fetch unit (64KB), RA: Read-Ahead (32KB), CSP: Conditional sequential prefetch (16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2).

Table 5: Overall Effect of Performing Preemptible Read-Ahead and Read Any Free Blocks in Addition to Non-Opportunistic Prefetch. Table shows percentage improvement over a system that does not prefetch.

correct track before the requested data rotates under. Basically, this allows the disk to prefetch more of the succeeding blocks. As shown in Tables A-1 and 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 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 I/Os that are tagged as lower-priority. This technique is called *freeblock scheduling* [25] and is meant to allow tasks such as disk scrubbing and data mining to be performed in the background without any impact on the foreground work. For instance, if the next block to be read is halfway round the track, the disk head could be positioned to service background requests "for free" as long as it could be 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, the best performance is obtained for the PC workloads when preemptible read-ahead and read any free blocks are performed in addition to simple read-ahead. Specifically, this means starting to read once the disk head is positioned over the correct track, and reading 32 KB and, if there are no incoming requests, up to 128 KB beyond the requested data. The average read response time in this case is improved by almost 50% over a system that does not prefetch (Table 5). For the server workloads, performance improvement of between 42% and 54% in the resource-poor environment and up to 65% in the resource-rich environment are achieved when conditional sequential prefetch is supplemented by preemptible read-ahead and read any free blocks.

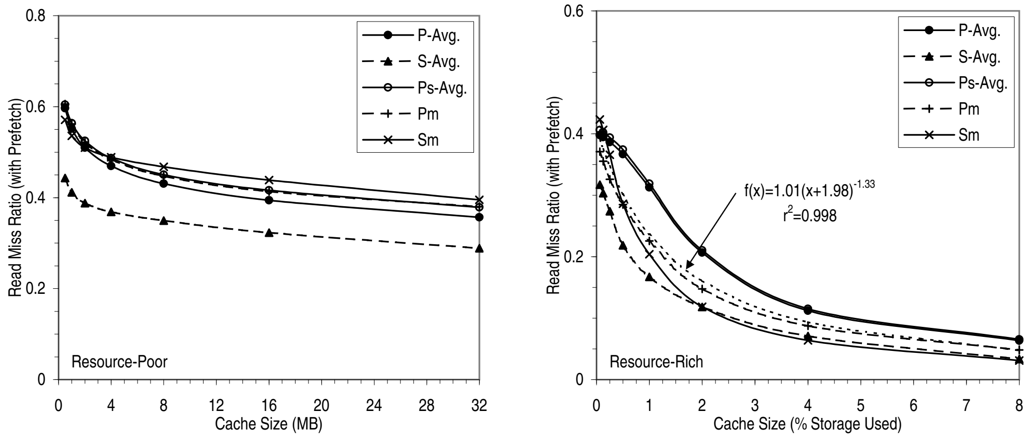
4.2.5 Sensitivity to Cache Size

In Figure 8, we analyze the performance sensitivity to cache size when data is prefetched into the cache using the default parameters (Figure 2). The default parameters mean that in the resource-poor environment, we read-ahead by at least 32 KB on every cache miss and up to 128 KB if there are no incoming request. We also perform read any free blocks. In the resource-rich environment, we perform conditional sequential prefetch, together with preemptible read-ahead and read any free blocks.

Observe that with prefetching, more than 50% of the reads can be satisfied by a 4 MB cache. Increasing the cache size beyond 4 MB to 32 MB achieves only diminishing returns. Such results suggest that disk drive caches in the MB range are sufficient. On the other hand, for very large caches, the miss ratio continues to improve as the cache size is increased beyond 4% of the storage used. As we have discussed earlier, most enterprise class storage controllers today, when fully loaded, have a front-end (cache size) to back-end (storage space) ratio of between 0.05% and 0.2% [8, 16, 22]. Our results suggest that increasing the cache size for these systems is likely to continue to be useful. Note, however, that the desirable amount of cache may not scale linearly with the size of the system.

4.3 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 [17], 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 written to permanent storage, the write buffer is typically implemented with some



(a) Resource-Poor.

(b) Resource-Rich.

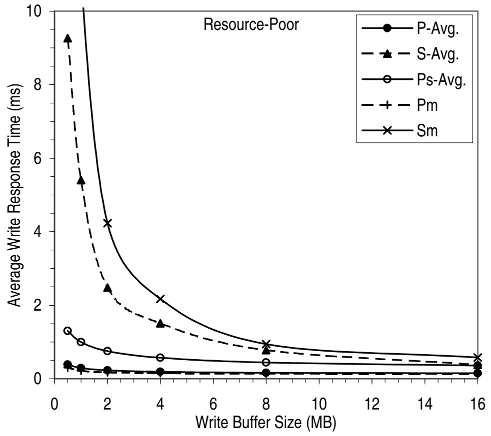
Figure 8: Sensitivity to Cache Size when Data is Prefetched into the Cache.

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.

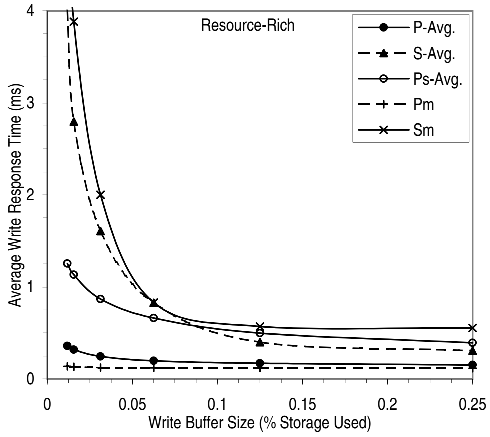
In short, the write buffer achieves three main effects. First, it hides the latency of writes by deferring them to some later time. Second, it reduces the number of physical writes, and third, it enables the remaining physical writes to be performed efficiently. In this paper, we evaluate write buffering 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, $highMark$, and is suspended once the *fraction* of dirty blocks in the buffer drops below a low limit threshold, $lowMark$. By appropriately setting $highMark$, we can ensure 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 [17] 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 [18]. 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 [1], the track with the most dirty blocks is selected for destage. In [42], 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 example, 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 first focus on reducing the number of physical writes by destaging blocks that are less likely to be rewritten and to then perform the remaining writes efficiently. To achieve the latter, 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. Also, we allow as many outstanding destage requests (contiguous blocks) 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 setting $lowMark$ to be significantly lower than



(a) Resource-Poor.



(b) Resource-Rich.

Figure 9: Improvement in Average Write Response Time from Absorbing Write Bursts.

$highMark$, 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.

4.3.1 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 9, 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.

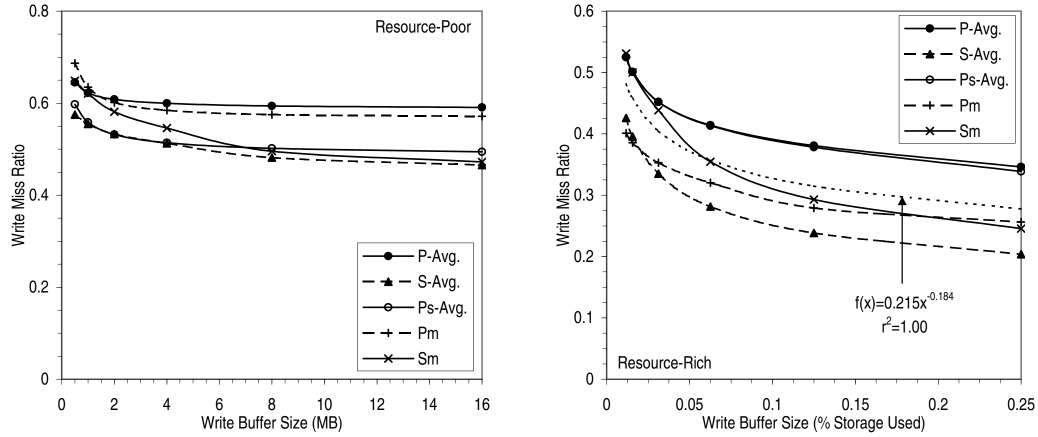
When the write buffer is not large enough to absorb the write bursts, some of the writes will stall until buffer space is reclaimed by destaging some of the dirty blocks. When the buffer is large enough, all the write requests can be completed without stalling. Notice that for all the workloads, a write 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 or units of buffer management and again found that 4 KB is reasonable for our workloads (Figure A-16).

4.3.2 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 10, 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 10(b), a power function of the form $f(x) = a(x - b)^c$ is again a good fit. However, the magnitude of the exponent c at about 0.2 is significantly lower 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 size of the read and write exponents are about 0.5 and 0.25 respectively [18].

Observe from Figure 10(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 A-17, 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 can-



(a) Resource-Poor.

(b) Resource-Rich.

Figure 10: Effectiveness of Write Buffering at Reducing Physical Writes.

cellation (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 A-18, 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.

4.3.3 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 11 and A-19, we plot the write miss ratio as a function of $highMark$. As we would expect, if destage is initiated whenever a small fraction of the buffer is dirty, there will be less opportunities for write cancellation. The write miss ratio is therefore high for small values of $highMark$. For our various workloads, we find that the miss ratio curves tend to flatten beyond a $highMark$ value of about 0.6. On the other hand, if the $highMark$ value is set high meaning that destage is initiated only when most of the buffer is dirty, response time will suffer because some of the writes will arrive to find the buffer full and will stall until buffer space becomes available. In Figures 12 and A-20, we plot the average write response time as a function of $highMark$. Observe that the average write response time rises as $highMark$ increases beyond about 0.8-0.9. In general, we find that a $highMark$ value of about 0.6-0.9 and a $lowMark$ value of less than 0.4 strike a reasonable compromise between absorbing write bursts and eliminating repeated writes. In the rest of this paper, we use as default a $highMark$ value of 0.8 and a $lowMark$ value of 0.2.

In Figures 13 and A-21, we plot the write service time as a function of the threshold settings. Notice that the service time curves are steeper than the corresponding miss ratio curves in Figures 11 and A-19. This is because the $highMark$ and $lowMark$ settings also affect how efficiently the destage operations can be carried out. In particular, when $lowMark$ is set close to $highMark$, the destage requests will be issued in a continuous trickle but when $lowMark$ is set significantly lower than $highMark$, the destage operations will be issued in batches so that they can be scheduled to be efficiently performed (Section 4.4).

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 A-23 and A-22, 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

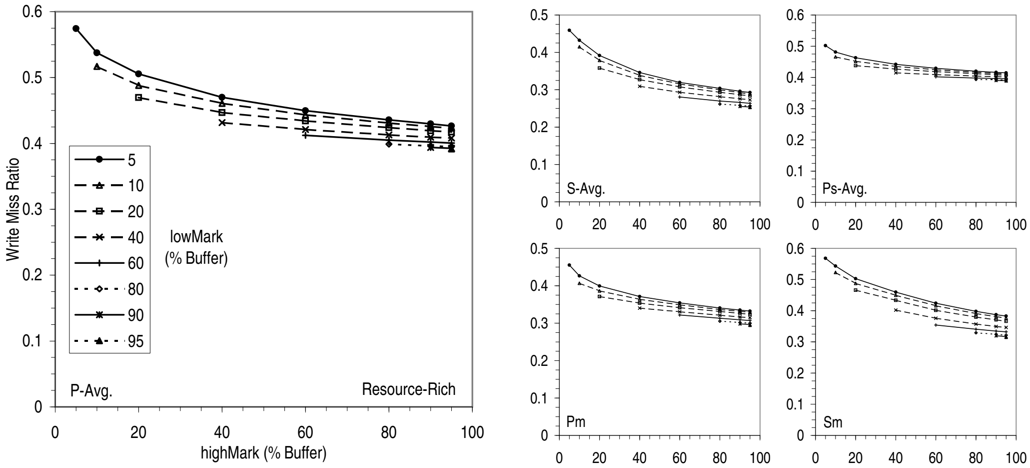


Figure 11: Effect of $lowMark$ and $highMark$ on Write Miss Ratio (Resource-Rich).

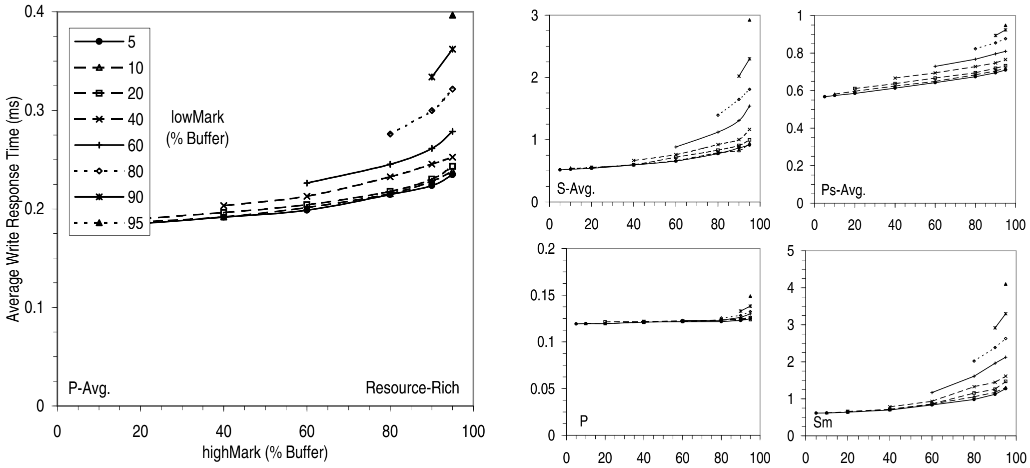


Figure 12: Effect of $lowMark$ and $highMark$ on Average Write Response Time (Resource-Rich).

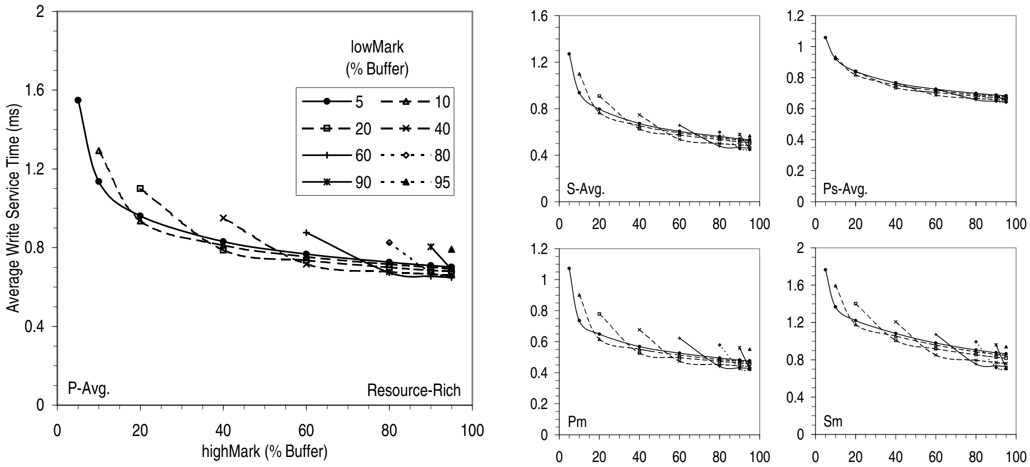


Figure 13: Effect of $lowMark$ and $highMark$ on Average Write Service Time (Resource-Rich).

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

[i] Improvement over write-through or no write buffer ([original value – new value]/[original value]).
Resource-Poor: 4MB, lowMark = 0.2, highMark = 0.8, Least-Recently-Written (LRW) replacement, 30s age limit.
Resource-Rich: 0.1% of storage used, lowMark = 0.2, highMark = 0.8, LRW replacement, 1 hour age limit.

Table 6: Performance with Write Buffering. Table shows percentage improvement over a write-through system.

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 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% over the write-through case while the average write response 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. Note that this large reduction in write service time with a relatively small write buffer, albeit non-volatile to avoid any data loss, puts into doubt the premise of log-structured file systems [28], which are based on the idea that with large disk caches, I/O systems will have almost no reads, and will be bottlenecked on writes.

4.4 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 [7], the order in which requests are handled is optimized to improve performance. 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.

Request scheduling can in principle be performed at different levels in the storage stack (*e.g.,* operating system, device driver, disk adaptor, disk drive), provided that the necessary information is available to estimate the service time of different requests. High in the storage stack, it is difficult to make good estimates 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 drive has little knowledge of the physical geometry of the disk, unless it knows the disk model number and has a table of the track and sector configuration. In addition, it is hard to predict the angular position of the disk or which requests will hit in the disk drive cache. As we have seen in the previous sections, there are a lot of hits in the disk drive cache, and such hits can substantially affect the effectiveness of request scheduling [43].

In this paper, we first consider scheduling the requests that miss in the cache since the critical resource is the disk arm. We term this *arm scheduling*. Our arm scheduling experiments assume a maximum queue depth of eight and are based on the scheduling algorithm that has been variously referred to as Shortest Time First [35], Shortest Access Time First [23] and Shortest Positioning Time First [43]. 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:

$$T_{eff} = T_{pos} - (W * T_{wait}) \quad (1)$$

We refer to this variation of the algorithm as Aged Shortest Access Time First (ASATF) [23].

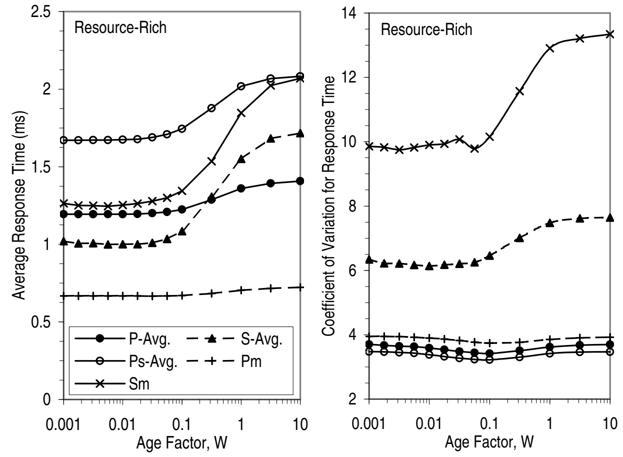


Figure 14: Effect of Age Factor, W, on Response Time (Resource-Rich).

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 [23, 43] but the range of "good" values for W is found to be wide. In Figures 14 and A-24, 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 Figure A-25 while the plots of the average service time as a function of W are in Figure A-26. 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 [23, 43], 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 arm scheduling. We summarize the results in Table 7. In general, arm 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 the destage operations are not scheduled efficiently, the write buffer is likely to become full and cause the incoming writes to stall.

Note that arm 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. Observe from Table 7 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 [17]. The sizeable improvement of up to 40% 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.

So far in this section, we have assumed a maximum queue depth of eight and focused on the effectiveness of arm scheduling. In practice, when there are multiple outstanding requests, the storage system cache in effect performs an *additional* level of scheduling by allowing subsequent cache hits to proceed. We refer to this effect as *cache scheduling*. In Table 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 Figure A-27. The improvement in response time reported in Table 8 includes the effect of both cache and arm scheduling. That it exceeds by only a small amount the improvement due to arm scheduling alone (Table 7) suggests that the cache scheduling effect tends to be secondary.

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 A-2).

[1][43] 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.

		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]).
Age factor: 0.01, queue depth: 8.

Table 7: Performance with Aged Shortest Access Time First (ASATF) Scheduling. Table shows percentage improvement over FCFS scheduling.

		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]).
Shortest Access Time First with Age Factor of 0.01.

Table 8: Average Response and Service Times as Maximum Queue Depth is Increased from One.

4.5 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 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. In RAID (Redundant Array of Inexpensive Disks) [6] terminology, the stripe set is known as RAID-0. Note that the volume set is essentially a stripe set with a stripe unit that is equal to the size of the disk. 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 [5] 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

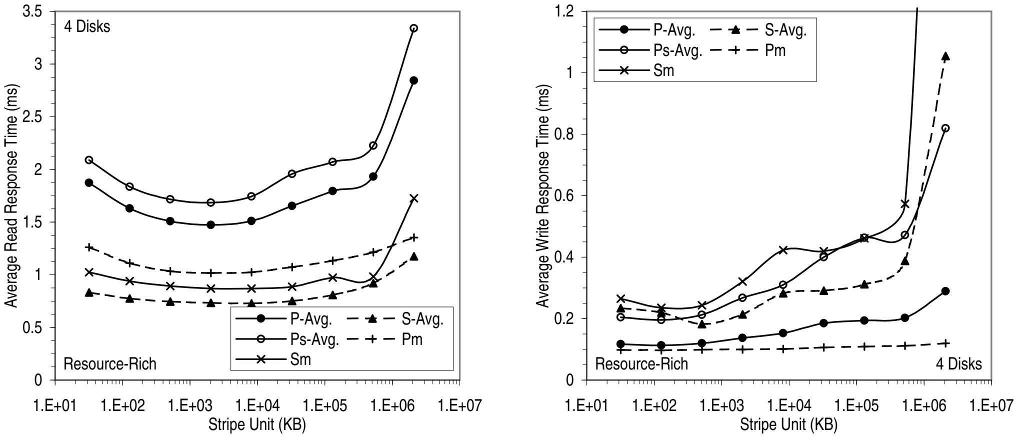


Figure 15: Average Read and Write Response Time as a Function of Stripe Unit (Resource-Rich).

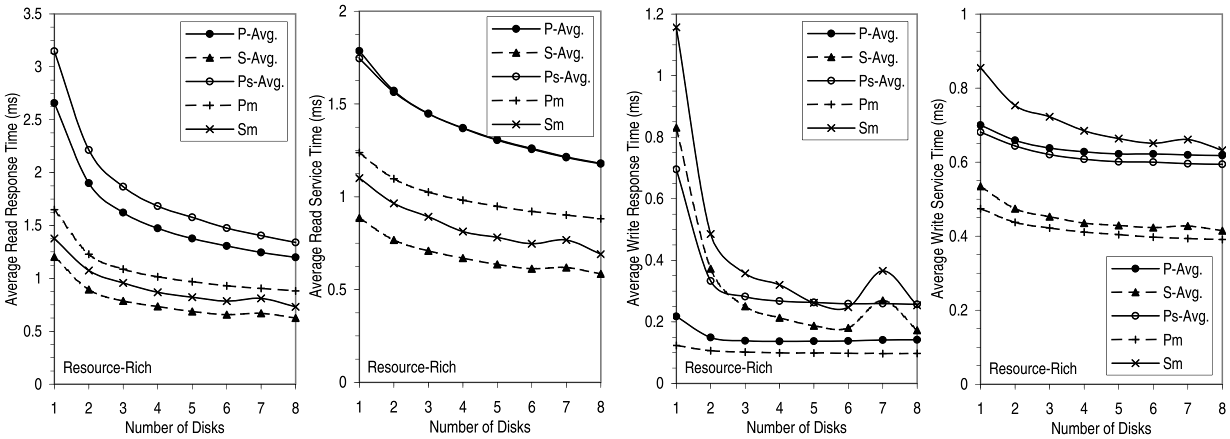
disks, thereby increasing the number of physical I/Os and causing many disks to be busy. More importantly, it results in many small random requests, which the disks are not very efficient at handling. Furthermore, a small stripe unit makes sequential prefetch by the disk less effective because data that 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 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 [4] 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 15 and A-28, 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 A-29. Observe that the response time does not rise dramatically until the stripe unit is well beyond 100 MB. This suggests that for our workloads, access skew, or imbalance in the amount of work borne by the different disks, is not a major issue unless the stripe unit is larger than 100 MB. As we increase the number of disks, it becomes more difficult to keep all the disks equally busy so that the upward surge in response time at large stripe units is more apparent (Figure A-30). From Figures 15 and A-28, 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 paper, we will assume a stripe unit of 2 MB.

Figures 16 and A-31 show the performance achieved as we increase the number of disks that are striped across. For all our workloads, striping data across four disks is sufficient to reap most of the performance benefit. In Table 9, we summarize the improvement in performance 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 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 the performance improvement reported in Table 9 is not due solely to less waiting for the disk arm. As more disks are used, there are more caches, prefetch buffers and disk arm idle time with which to perform opportunistic prefetch. The combined effect of these additional resources is reflected in the decrease in miss ratio. In the resource-poor environment, the read miss ratio improves by about 20% when data is striped across four disks. The corresponding improvement in the resource-rich environment is about 15%. Recall that we define the miss ratio as the fraction of requests that requires physical I/O. Therefore, when there are multiple disks each with a cache, the miss ratio is the arithmetic mean of the miss ratio of each disk, weighted by the number of requests to that disk.

Notice further that the read service time improves by about 10% more than the read miss ratio as we increase the number of disks from one to four. This improvement in service time beyond the reduction in miss ratio is due



(a) Read.

(b) Write.

Figure 16: 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]).
Stripe unit: 2MB.

Table 9: Performance with Striping across Four Disks. Table shows percentage improvement over a single disk.

to less disk arm movement. When data is striped across the disks, the locality of reference is affected. For example, each of the active regions (*e.g.,* active files) could be mapped contiguously to a different disk in which case each of the disk arms would not have to travel far. Conversely, an active region could be distributed among the multiple disks, requiring all the arms to move to that region. More importantly, when data is distributed across more disks of the same capacity which is what we are doing, the total capacity of the system grows and each disk arm has a narrower range of movement. An alternative would be to compare performance using smaller-capacity disks as the number of disks increases so as to keep the total storage capacity constant, but the storage required for many of our workloads is already smaller than the capacity offered by a 1-surface disk.

More generally, when only a portion of the disk capacity is used, the disk performs better because the seek distance is reduced. This effect is called *short-stroking*. To directly quantify the short-stroking effect, we go back to our base configurations of using a single disk and increase the capacity of that disk by adding disk platters (recording surfaces). In Table A-3, we summarize the performance improvement achieved when these larger disks are used. Observe that the service time improvement saturates when disks that are four times larger than required are used. Largely in agreement with the results above, we find that short-stroking a disk that is four times larger then necessary improves the average read service time by 10-15% for our workloads. For writes, the improvement ranges from 15% to 20%. The improvement is rather low because short-stroking reduces only the seek time, which, as we shall see, constitutes only about 25% of the read response time. Moreover, because of inertia and head settling time, there is

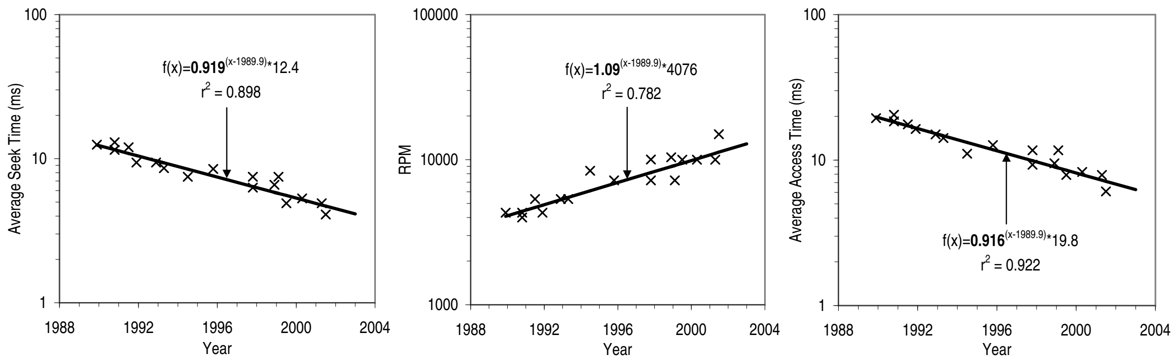


Figure 17: Historical Rates of Change in Average Seek Time, Rotational Speed and Access Time (IBM Server Disks).

but a relatively small time difference between a short seek and a long seek, especially with newer disks.

5 Effect of Technology Improvement

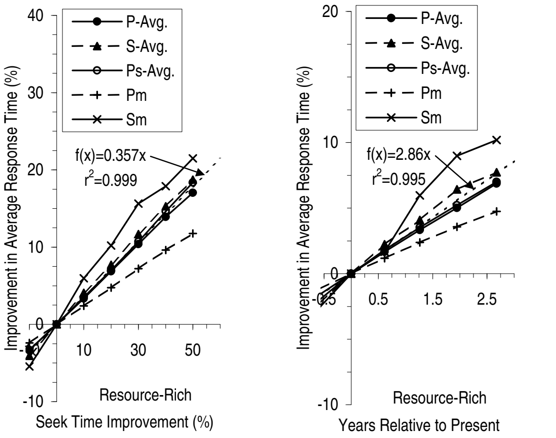
At its core, disk storage is composed of a set of rotating platters on the surfaces of which data is recorded. There is typically a read-write head for each surface and all the heads are attached to the disk arm so that they move in tandem. A simple high-level description such as this already suggests 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 at which it is operated. 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.

5.1 Mechanical Improvement

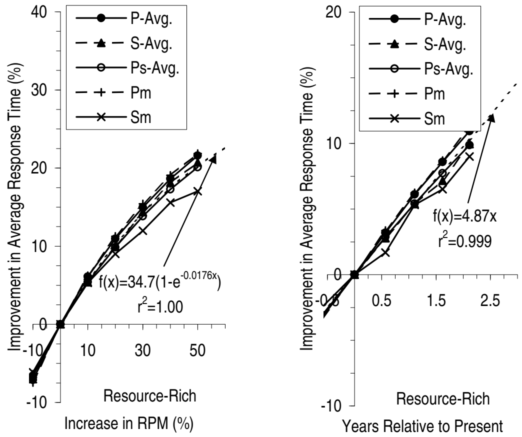
We begin by examining the improvement in the mechanical or moving parts of the disk. Figure 17 presents the historical rates of change in the average seek time and rotational speed for the IBM family of server disks. The average seek time is generally taken to be the av-



(a) Function of Improvement in Seek Time.

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

Figure 18: Effect of Improvement in Seek Time on Average Response Time (Resource-Rich).



(a) Function of Increase in RPM.

(b) Function of Years of RPM Increase at Historical Rate (9% Per Year).

Figure 19: Effect of RPM Scaling on Average Response Time (Resource-Rich).

erage 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.

5.1.1 Seek Time

As shown in Figure 3, 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? We find 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 A-32, 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 Figure 18, 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 A-33. Note that the physical I/Os in the resource-rich environment are not exactly those in the resource-poor environment because there are different amounts of caching and write buffering in the two environments. But it turns out that the performance effect of disk technology improvement is almost identical in both environments. We therefore present only the figures for the resource-rich environment in the main text. The largely similar plots for the resource-poor environment can be found in the appendix (Figure A-34).

Besides plotting the improvement in average response time as a function of the improvement in seek time (Figure 18(a)), we also show how the improvement in average read response time varies over time, assuming the historical 8% yearly improvement in seek time (Figure 18(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 under 3% improvement in the average response time.

5.1.2 Rotational Speed

Figures 19 and A-35 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 A-36. Notice that the S-Avg. plot in Figure A-35 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, espe-

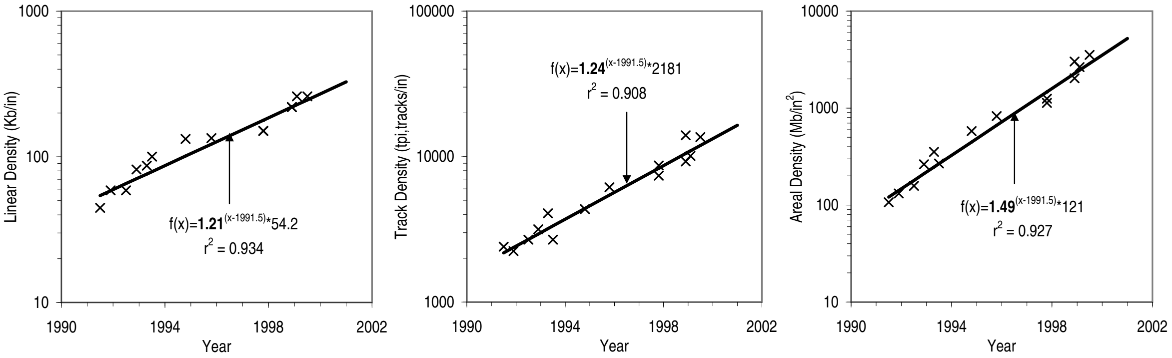


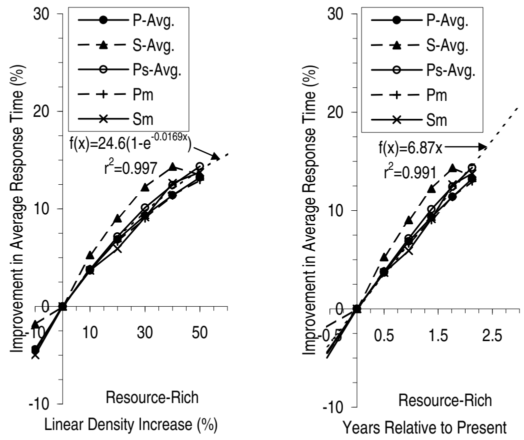
Figure 20: Historical Rates of Increase in Linear, Track and Areal Density (IBM Server Disks).

cially 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 [19].

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.

5.2 Increase in Areal Density

In Figure 20, 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 A-37, 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 [19].



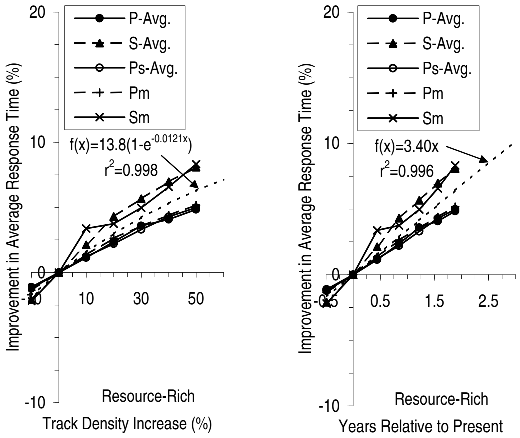
(a) Function of Increase in Linear Density.

(b) Function of Years of Linear Density Increase at Historical Rate (21% Per Year).

Figure 21: Effect of Increased Linear Density on Average Response Time (Resource-Rich).

5.2.1 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 21 and A-39 show how increases in the linear density reduce the average response time for our various workloads. We find that the improvement in average response time as a function of the increase in linear density can again be accurately 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



(a) Function of Increase in Track Density.

(b) Function of Years of Track Density Increase at Historical Rate (24% Per Year).

Figure 22: Effect of Increased Track Density on Average Response Time (Resource-Rich).

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.

5.2.2 Track Density

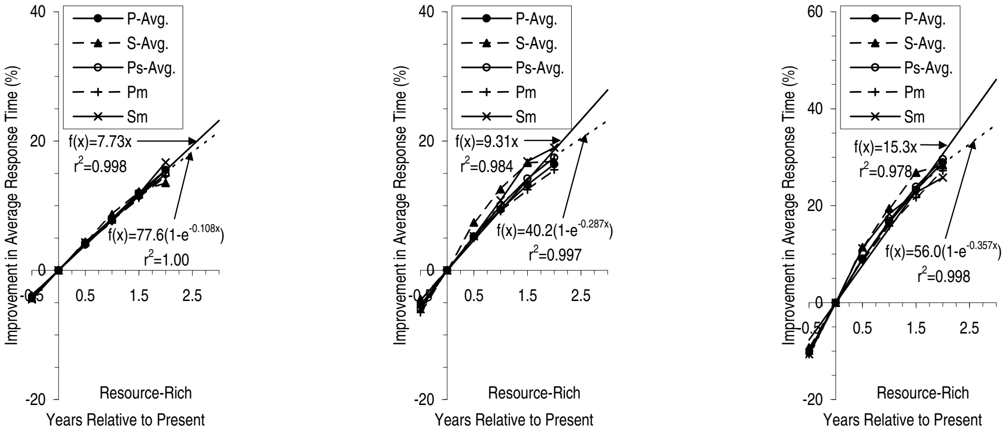
Packing the tracks closer together means that the arm has to move over a shorter physical distance to get to the same track. This effect is similar to that of improving the seek time but the quantitative effect on the average response time per unit of improvement tends to be much smaller because of the shape of the seek profile. In particular, the marginal cost of moving the arm is relatively small once it is moved. In Figures 22 and A-40, 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.

5.3 Overall Improvement over Time

In Figures 23(c), A-42(c), A-43(c) and A-44(c), 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 "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. In any case, we note that CPUs are doubling in speed every year or two, so the demands on the I/O system are increasing faster than the capability of the I/O system.

To estimate the yearly improvement in the more realistic situation where the increased capacity of the newer disks are 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 23(a), A-42(a), A-43(a) and A-44(a) 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 (Figures 23(b), A-42(b), A-43(b) and A-44(b)) and discover that the average response and service times are improved by about 9% per year. This improvement comes about because as areal density is increased, the data is packed more closely 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 increase in areal density. 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



(a) Mechanical Improvement. (b) Areal Density Increase. (c) Disk Technology Improvement.

Figure 23: Effect on Average Response Time (Resource-Rich).

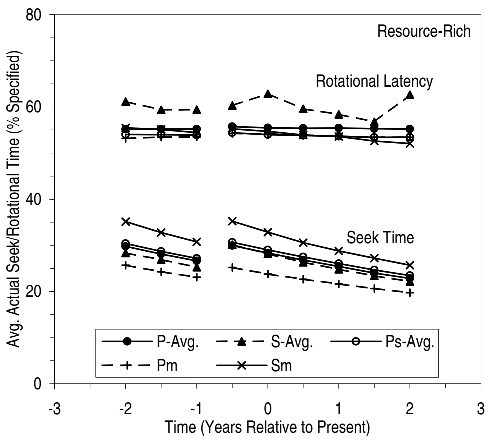


Figure 24: Actual Average Seek and Rotational Time as Percentage of Manufacturer Specified Values (Resource-Rich).

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 Figures 24 and A-45, 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 the head is about 60% of that specified. The seek percentage decreases slightly over time because of the improvement in areal density but the effect is not very significant. The non-monotonic nature of the rotational latency curve for S-Avg. is again due to the fact that DS1 is sensitive to how blocks are laid out in tracks. As the rotational speed and track density increase over time, a poor block layout sometimes results.

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 Figures 25 and A-46. 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 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

%			Read			Write		
			Avg. Resp. Time	Avg. Service Time	Miss Ratio	Avg. Resp. Time	Avg. Service Time	Miss Ratio
Resource-Poor	Read Caching	8MB per disk, LRU replacement.	4.49	4.14	4.45	0	0	0
	Prefetching	32KB read-ahead, preemptible read-ahead up to 128KB, read any free blocks.	46.6	46.1	53.0	0	0	0
	Write Buffering	4MB per disk, Least-Recently-Written (LRW) replacement, 30s age limit.	0	0	0	93.8	71.8	43.5
	Request Scheduling	Shortest Access Time First , queue depth of 8.	16.2	2.8	0	49.9	30.5	0
	Parallel I/O	Stripe unit of 2MB.	46.8	29.5	20.9	65.7	13.5	5.96
Resource-Rich	Read Caching	1% of storage used, LRU replacement.	37.4	36.1	35.1	0	0	0
	Prefetching	Conditional sequential prefetch, preemptible read-ahead up to 128KB, read any free blocks.	51.1	50.3	59.7	0	0	0
	Write Buffering	0.1% of storage used, Least-Recently-Written (LRW) replacement, 1 hour age limit.	0	0	0	96.0	86.2	63.1
	Request Scheduling	Shortest Access Time First, queue depth of 8.	17.0	1.8	0	46.2	38.4	0
	Parallel I/O	Stripe unit of 2MB.	40.0	23.3	15.2	46.1	14.9	-1.02

Table 10: Performance Effect of Various I/O Optimization Techniques. Table shows percentage improvement (|[time without technique - time with technique]|/[time without technique]). Table entries are shaded to reflect the relative magnitude of improvement with darker shades representing larger improvements.

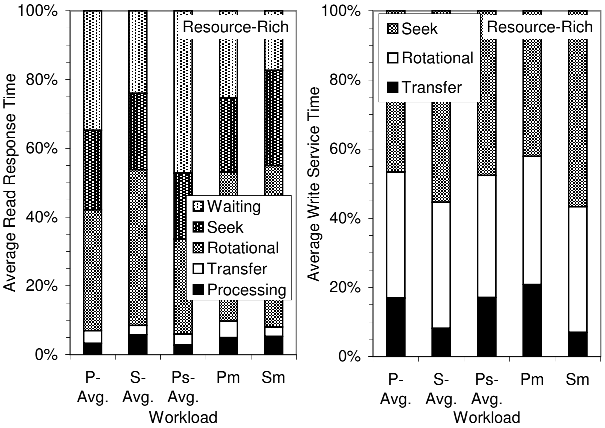


Figure 25: Breakdown of Average Read Response and Write Service Time (Resource-Rich).

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 [19]. 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.

6 Conclusions, Summary and Synthesis

In this paper, 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 10. 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 the most effective approach to improving I/O performance is to reduce the number of physical I/Os that have to be performed. When designing a storage system, we would therefore first focus on caching, prefetching, and write buffering. Because caching is already performed upstream in the host, small caches in the megabyte range are not useful at the storage level. The small amount of memory in the disk drive should be designed more as a prefetch buffer than a cache that captures block reuse. Increasing its size beyond the megabyte range is thus not very useful. If cost is not a big constraint, a large cache on the order of 1% of the storage capacity can be effective at the storage level. Further increasing the size of this cache is almost always a good idea as the miss ratio continues to decrease at cache sizes that are beyond 4% of the storage used.

Our results clearly indicate that sequential prefetch is extremely effective. We highly recommend performing simple read-ahead and, in more sophisticated implementations, setting the prefetch amount by conditioning on the length of the sequential run already observed. In a resource-poor environment such as one where the

%	Annual Rate of Improvement	Resource-Poor		Resource-Rich	
		Avg. Resp. Time	Avg. Service Time	Avg. Resp. Time	Avg. Service Time
Linear Density	21	6.21	5.39	7.08	6.73
Track Density	24	3.48	3.28	3.42	3.29
Areal Density	49	8.58	7.97	9.31	9.07
Disk Arm (Seek Time)	8	3.24	3.39	3.08	3.18
Rotational Speed	9	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 11: Performance Effect of Disk Technology Evolution at the Historical Rates. Table shows percentage yearly improvement(|[original value - new value]|/[new value]). Table entries are shaded to reflect the relative size of improvement with darker shades representing larger improvements.

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 when 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. We therefore advocate that opportunistic prefetch be enabled on the disks.

The write buffer should be designed in the same spirit of reducing physical operations by allowing repeated writes to the same blocks to be eliminated. Using a Least-Recently-Written (LRW) replacement policy, we find that 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. For all our workloads, most of the benefit of write elimination can be achieved without requiring dirty data to remain in the buffer beyond an hour. The write buffer should also be sized to absorb incoming write bursts. We recommend a write buffer size that is on the order of 0.1% of the storage capacity. Our results show that such a buffer can improve write response time by over 90%.

After investing in techniques that reduce the number of physical operations, it is worthwhile to consider optimizations that increase the efficiency in performing the remaining I/Os. For instance, when the writes are buffered, the remaining physical writes should be issued in batches so that they can be effectively scheduled and efficiently performed. In general, we should try to queue multiple requests at the disk (*e.g.,* by setting the disk adaptor queue depth to more than one) so that the disk can optimize the order in which the requests are carried out. 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. If data is striped across multiple disks to allow parallel I/O, we would do it with a large stripe unit in the megabyte range. By striping at such a granularity across four disks, average read response time can be reduced by 40-45% over the one-disk case. We would generally not recommend short-stroking the disk since using a disk that is four times larger than necessary results in only about a 10-20% improvement in performance.

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 11, 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 60% 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 [19].

Acknowledgments

The authors would like to thank Ruth Azevedo, Jacob Lorch, Bruce McNutt and John Wilkes for providing the traces used in this study. Thanks are also owed to William Guthrie who shared with us his expertise in modeling disk drives, and to Ed Grochowski who provided the historical performance data for IBM disk drives. In addition, the authors are grateful to Jai Menon, John Palmer and Honesty Young for helpful comments on versions of this paper.

References

[1] P. Biswas, K. K. Ramakrishnan, and D. Towsley, "Trace driven analysis of write caching policies for disks," *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, (Santa Clara, CA), pp. 13–23, May 1993.

[2] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rossel, "A virtual machine emulator for performance evaluation," *Communications of the ACM*, 23, 2, pp. 71–80, 1980.

[3] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (New Orleans, LA), pp. 1–14, Feb. 1999.

[4] P. M. Chen and E. K. Lee, "Striping in a RAID level 5 disk array," *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, (Ottawa, Canada), pp. 136–145, May 15–19 1995.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: high-performance, reliable secondary storage," *ACM Computing Surveys*, 26, 2, pp. 145–185, June 1994.

[6] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The rio file cache: Surviving operating system crashes," *Proceedings of ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (Cambridge, MA), pp. 74–83, Oct. 1996.

[7] P. J. Denning, "Effects of scheduling on file memory operations," *Proceedings of AFIPS Spring Joint Computer Conference*, (Atlantic City, NJ), pp. 9–21, Apr. 1967.

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

[9] M. A. Gaertner and J. L. Wach, "Rotationally optimized seek initiation." U.S. Patent 6339811. Filed Dec. 28, 1999. Issued Jan. 15, 2002.

[10] G. 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.

[11] G. R. Ganger, B. L. Worthington, and Y. N. Patt, *The DiskSim Simulation Environment Version 2.0 Reference Manual*, 1999.

[12] J. L. Griffin, J. Schindler, S. W. Schlosser, J. S. Bucy, and G. R. Ganger, "Timing-accurate storage emulation," *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, (Monterey, CA), pp. 75–88, Jan. 2002.

[13] J. Griffioen and R. Appleton, "Reducing file system latency using a predictive approach," *Proceedings of the Summer 1994 USENIX Conference*, pp. 197–207, June 1994.

[14] E. Grochowski, "IBM magnetic hard disk drive technology," 2002. http://www.hgst.com/hdd/technolo/grochows/grocho01.htm.

[15] L. Haas, W. Chang, G. Lohman, M. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita, "Starburst mid-flight: As the dust clears," *IEEE Transactions on Knowledge and Data Engineering*, 2, 1, pp. 143–160, Mar. 1990.

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

[17] W. W. Hsu and A. J. Smith, "Characteristics of I/O traffic in personal computer and server workloads," *IBM Systems Journal*, 42, 2, pp. 347–372, 2003.

[18] W. W. Hsu, A. J. Smith, and H. 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, pp. 96–143, Mar. 2001.

[19] W. W. Hsu, A. J. Smith, and H. C. Young, "The automatic improvement of locality in storage systems." Technical Report, CSD-03-1264, Computer Science Division, University of California, Berkeley, July 2003. Also available as Chapter 4 of [20].

[20] W. W. Hsu, *Dynamic Locality Improvement Techniques for Increasing Effective Storage Performance*, PhD thesis, University of California, Berkeley, 2002. Available as Technical Report CSD-03-1223, Computer Science Division, University of California, Berkeley, Jan. 2003.

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

[22] IBM Corporation, "IBM TotalStorage™ Enterprise Storage Server Models F10 and F20," 2000.

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

[24] J. R. Lorch and A. J. Smith, "The VTrace tool: Building a system tracer for Windows NT and Windows 2000," *MSDN Magazine*, 15, 10, pp. 86–102, Oct. 2000.

[25] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle, "Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives," *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (San Diego, CA), pp. 87–102, Oct. 2000.

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

[27] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the SPRITE network file system," *ACM Transactions on Computer Systems*, 6, 1, Feb. 1988.

[28] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," *Operating Systems Review*, 23, 1, pp. 11–28, Jan. 1989.

[29] D. A. Patterson and K. 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.

[30] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, (Copper Mountain, CO), pp. 79–95, Dec. 1995.

[31] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1990.

[32] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE parallel and distributed technology: systems and applications*, 3, 4, pp. 34–43, Winter 1995.

[33] C. Ruemmler and J. Wilkes, "Disk shuffling," Techical Report HPL–91–156, Hewlett-Packard Laboratories, Palo Alto, CA, USA, Oct. 1991.

[34] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," *Proceedings of USENIX Winter Conference*, (San Diego, CA), pp. 405–420, Jan. 1993.

[35] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," *Proceedings of Winter USENIX Conference*, (Washington, DC), pp. 313–324, Jan. 1990.

[36] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems*, 3, 3, pp. 223–247, Sept. 1978.

[37] A. J. Smith, "Input/output optimization and disk architectures: A survey," *Performance Evaluation*, 1, 2, pp. 104–117, 1981.

[38] A. J. Smith, "Disk cache — miss ratio analysis and design considerations," *ACM Transactions on Computer Systems*, 3, 3, pp. 161–203, Aug. 1985.

[39] A. J. Smith, "Trace driven simulation in research on computer architecture and operating systems," *Proceedings of Conference on New Directions in Simulation for Manufacturing and Communications*, (Tokyo, Japan), pp. 43–49, Aug. 1994.

[40] J. Z. Teng and R. A. Gumaer, "Managing IBM Database 2 buffers to maximize performance," *IBM Systems Journal*, 23, 2, pp. 211–218, 1984.

[41] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Computing Surveys*, 29, 2, pp. 128–170, June 1997.

[42] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," *IEEE Transactions on Computers*, 47, 2, pp. 228–235, 1998.

[43] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," *Proceedings of ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, (Nashville, TN), pp. 241–251, May 1994.

[44] B. T. Zivkov and A. J. Smith, "Disk cache design and performance as evaluated in large timesharing and database systems," *Proceedings of Computer Measurement Group (CMG) Conference*, (Orlando, FL), pp. 639–658, Dec. 1997.

Appendix

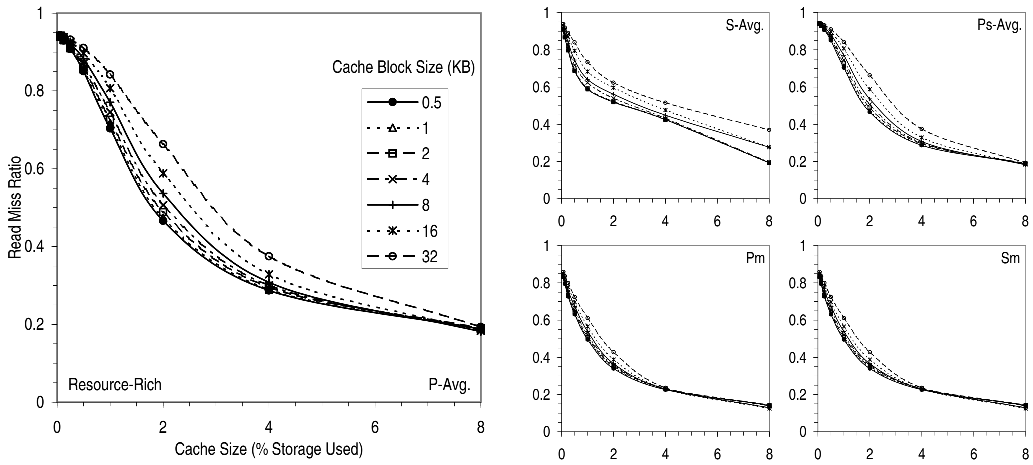
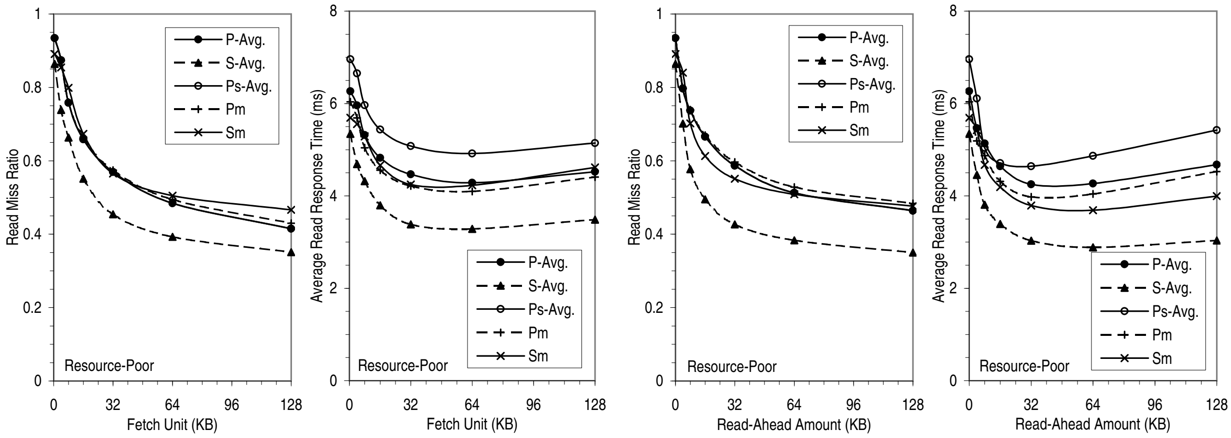


Figure A-1: Sensitivity to Cache Block Size.



(a) Large Fetch Unit.

(b) Read-Ahead.

Figure A-2: Effect of Large Fetch Unit and Read-Ahead on Read Miss Ratio and Response Time (Resource-Poor).

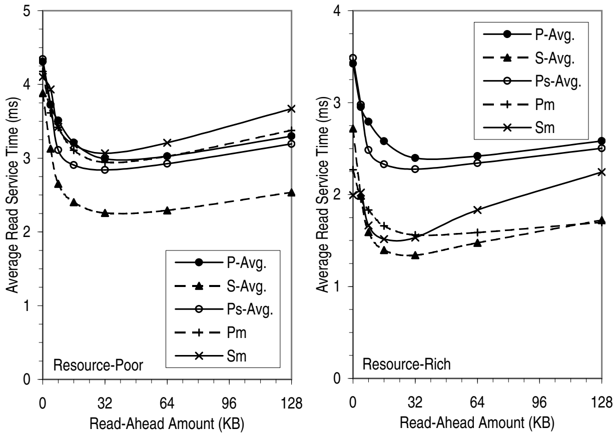


Figure A-3: Effect of Read-Ahead on Average Read Service Time.

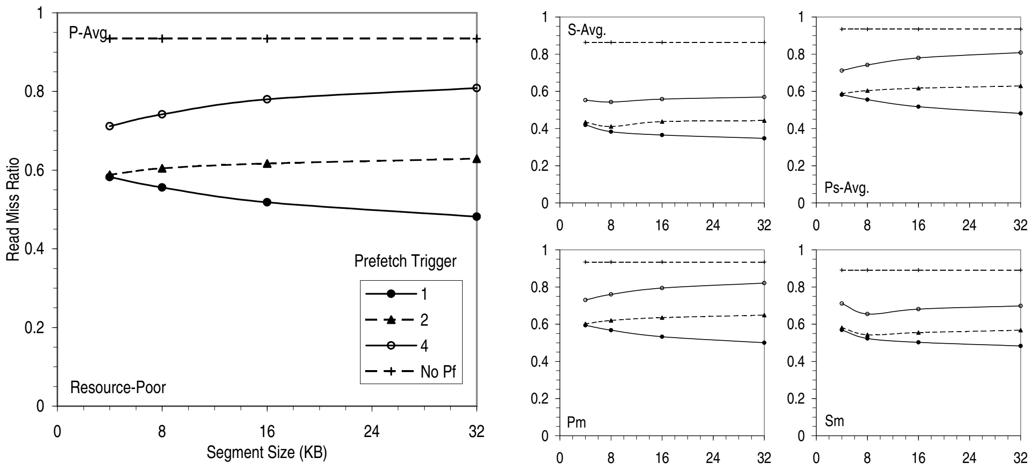
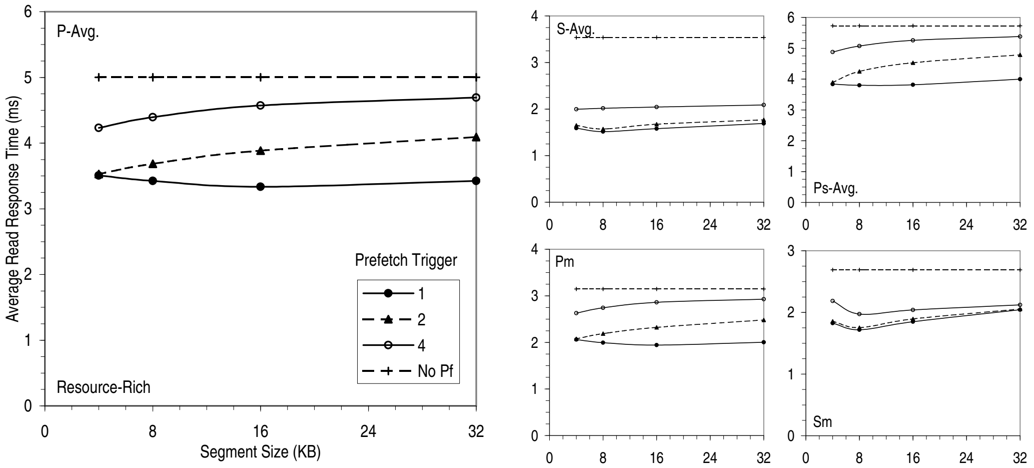
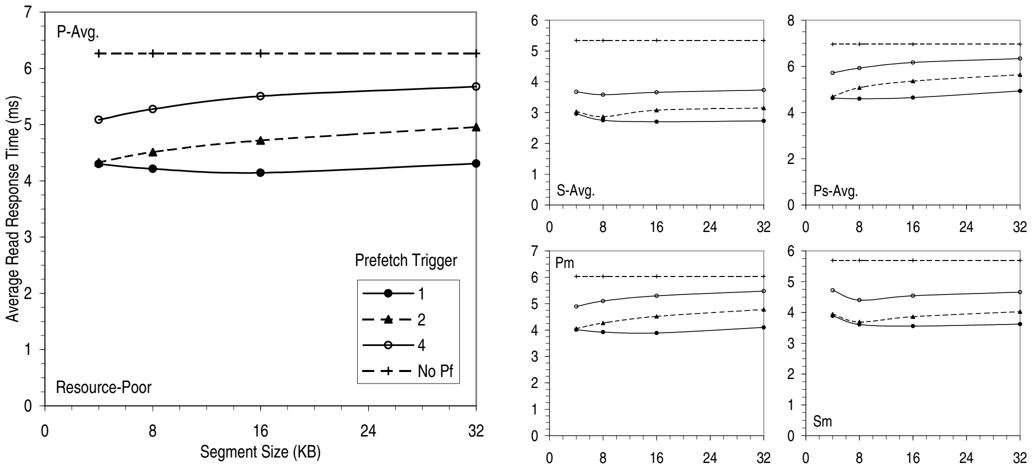


Figure A-4: Read Miss Ratio with Conditional Sequential Prefetch (Resource-Poor).



(a) Resource-Rich



(b) Resource-Poor

Figure A-5: Average Read Response Time with Conditional Sequential Prefetch (Resource-Poor).

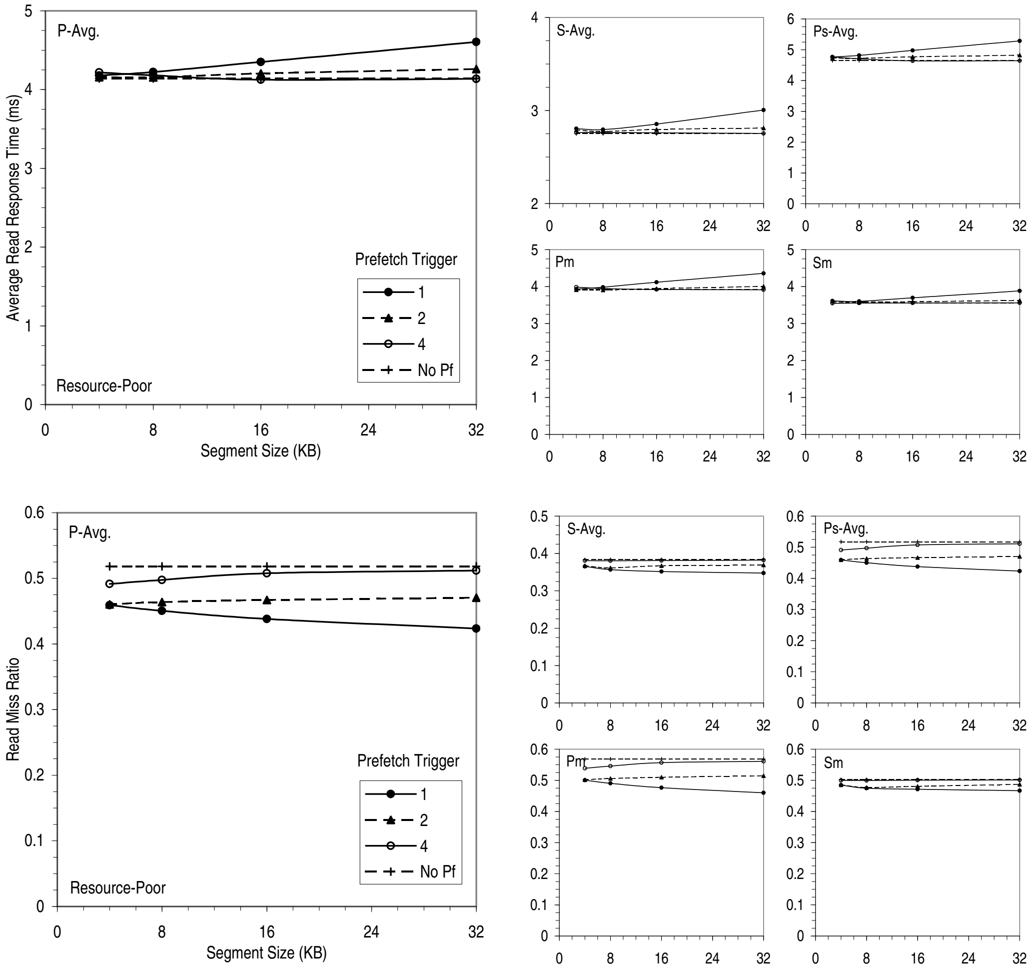


Figure A-6: Additional Effect of Backward Conditional Sequential Prefetch (Resource-Poor).

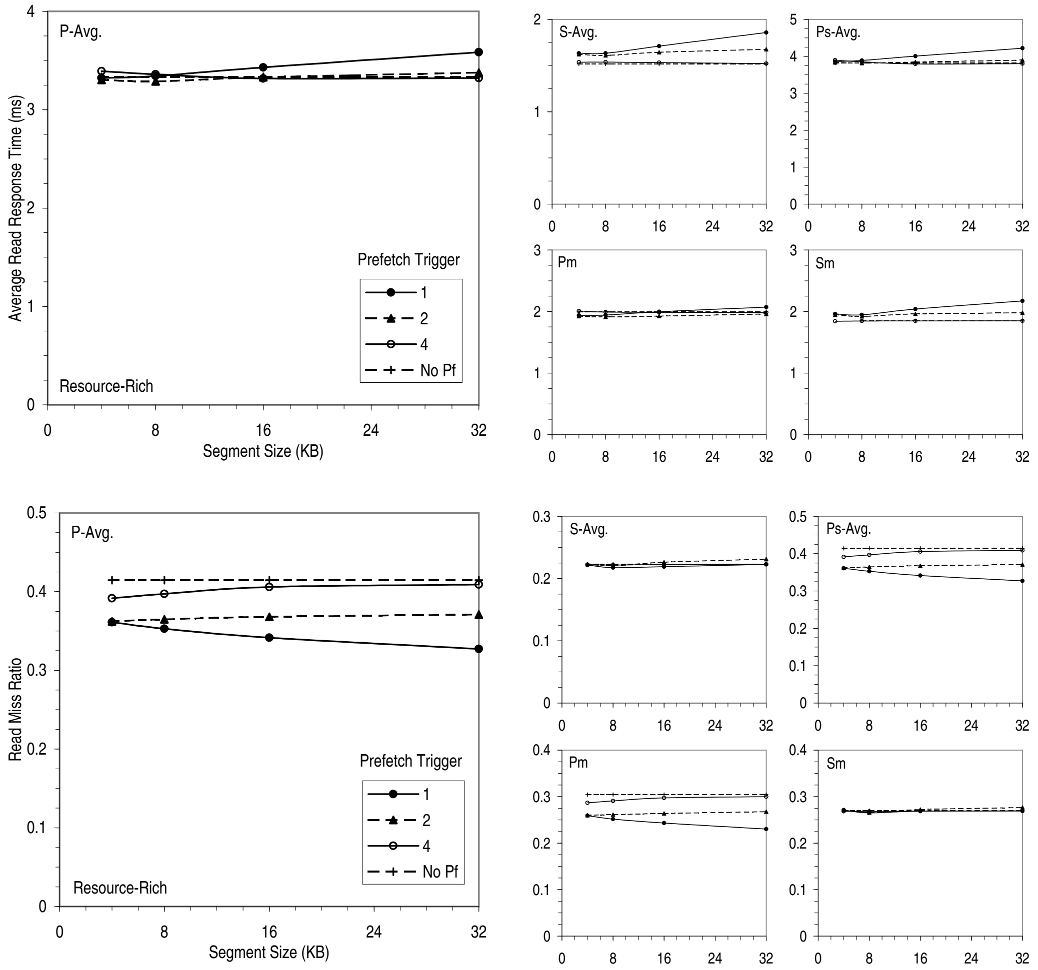


Figure A-7: Additional Effect of Backward Conditional Sequential Prefetch (Resource-Rich).

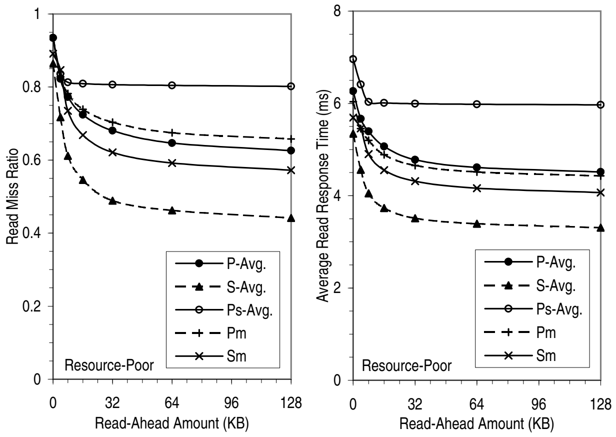


Figure A-8: Effect of Preemptible Read-Ahead on Read Miss Ratio and Response Time (Resource-Poor).

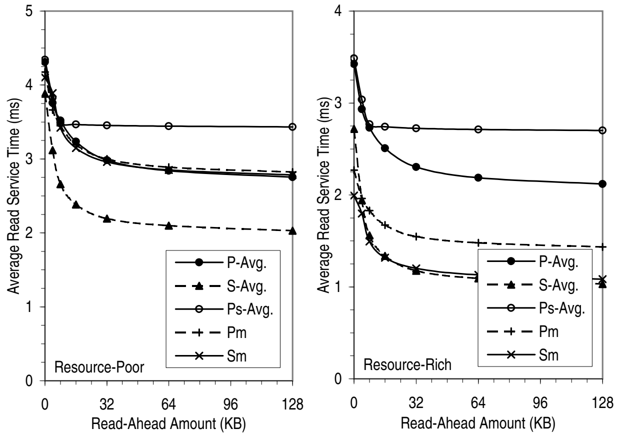


Figure A-9: Effect of Preemptible Read-Ahead on Average Read Service Time.

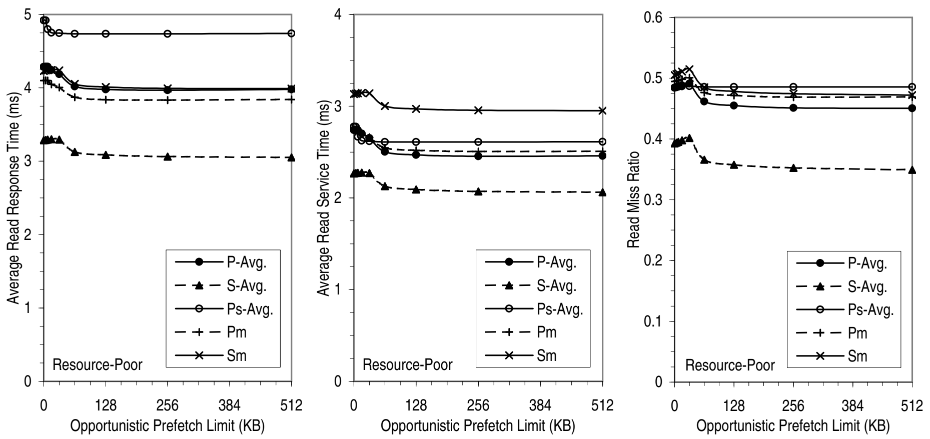


Figure A-10: Performance of Large Fetch Unit with Preemptible Read-Ahead (Resource-Poor).

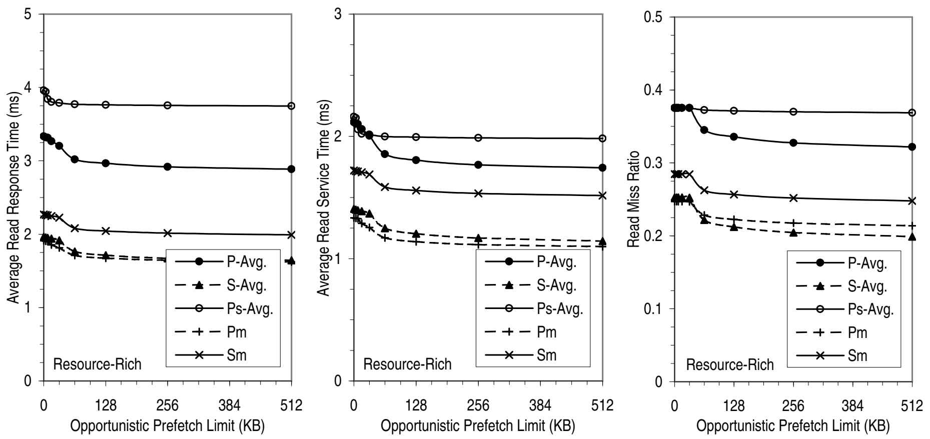


Figure A-11: Performance of Large Fetch Unit with Preemptible Read-Ahead (Resource-Rich).

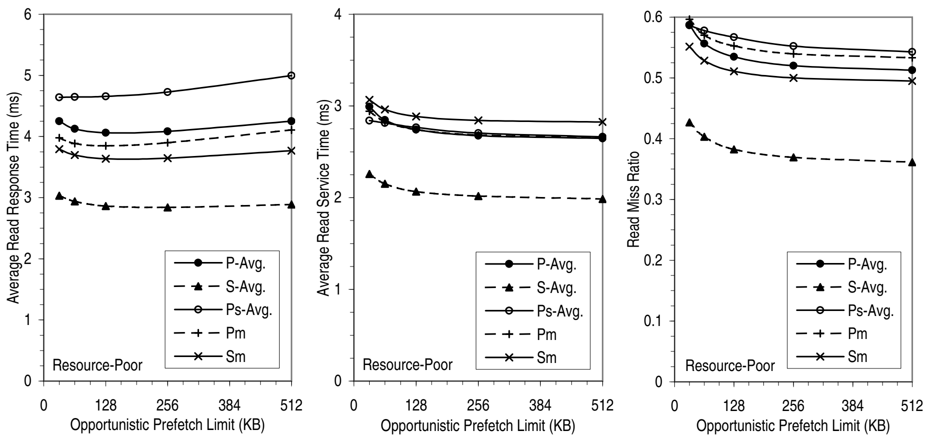


Figure A-12: Performance of Read-Ahead with Preemptible Read-Ahead (Resource-Poor).

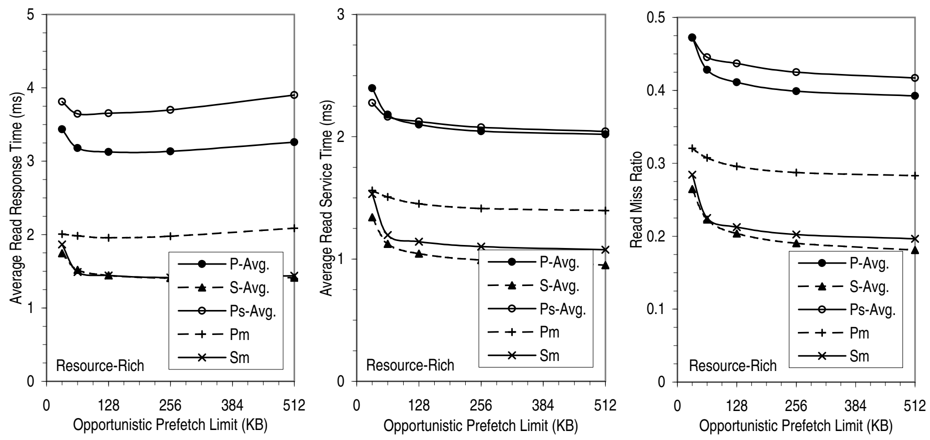


Figure A-13: Performance of Read-Ahead with Preemptible Read-Ahead (Resource-Rich).

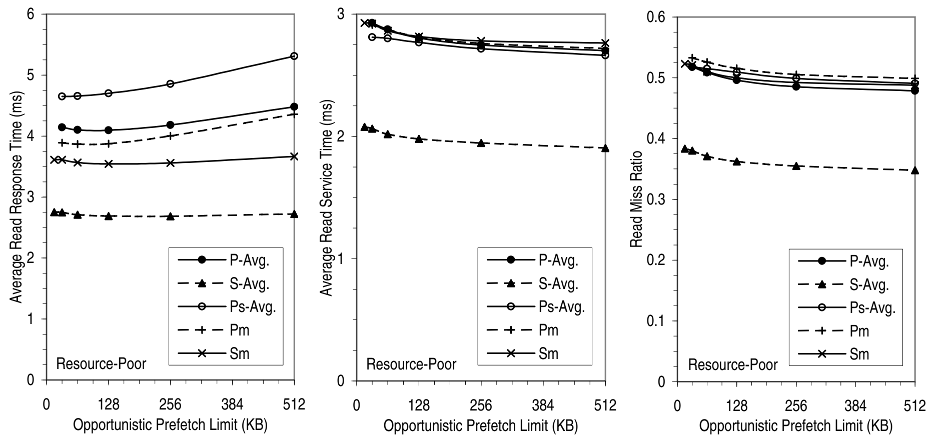


Figure A-14: Performance of Conditional Sequential Prefetch with Preemptible Read-Ahead (Resource-Poor).

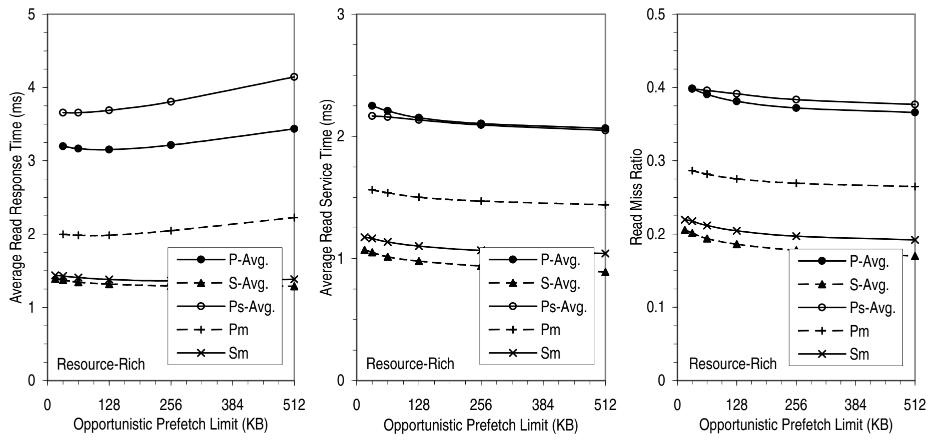


Figure A-15: Performance of Conditional Sequential Prefetch with Preemptible Read-Ahead (Resource-Rich).

		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 (64KB), RA: Read-Ahead (32KB), CSP: Conditional sequential prefetch (16KB segments for PC workloads, 8KB segments for server workloads, prefetch trigger of 1, prefetch factor of 2).
[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 A-1: Additional Effect of Opportunistic Prefetch (Resource-Poor). Table shows percentage improvement over a system that performs only non-opportunistic prefetch.

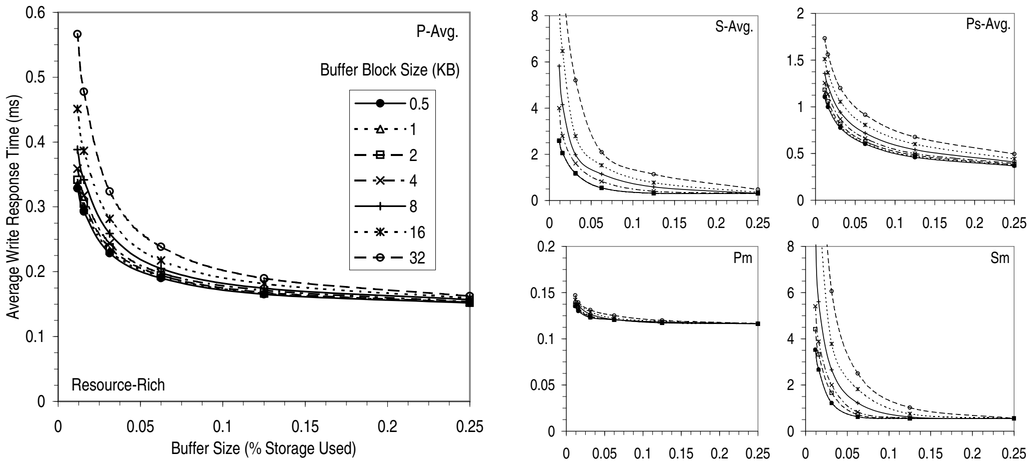
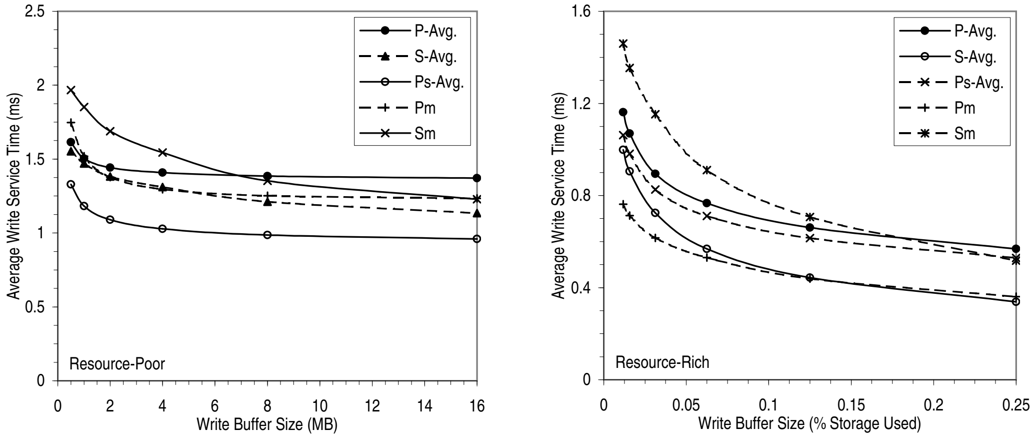


Figure A-16: Sensitivity to Buffer Block Size.



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

Figure A-17: Improvement in Average Write Service Time from Eliminating Repeated Writes.

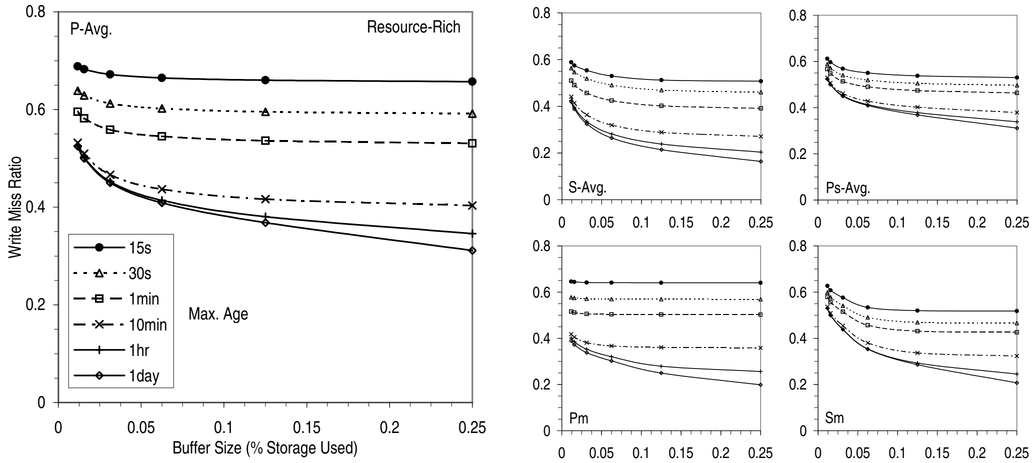


Figure A-18: Sensitivity to Maximum Dirty Age.

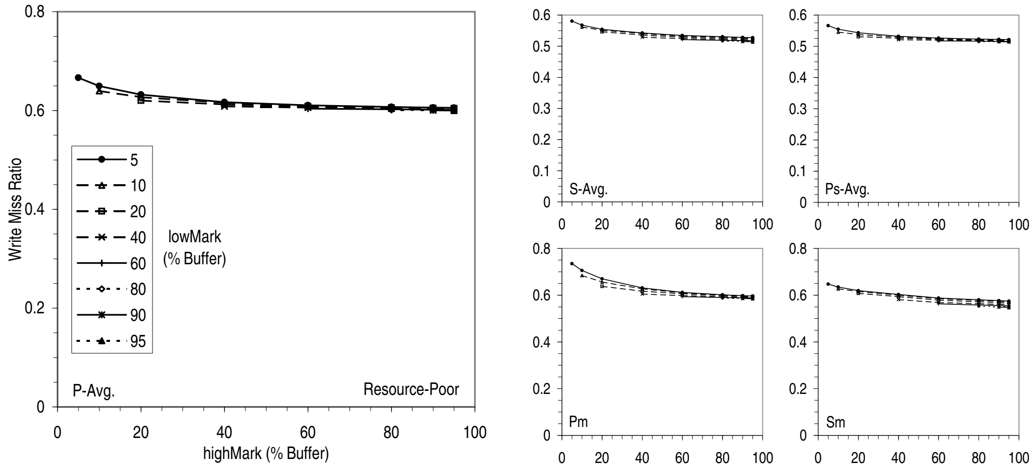


Figure A-19: Effect of $lowMark$ and $highMark$ on Write Miss Ratio (Resource-Poor).

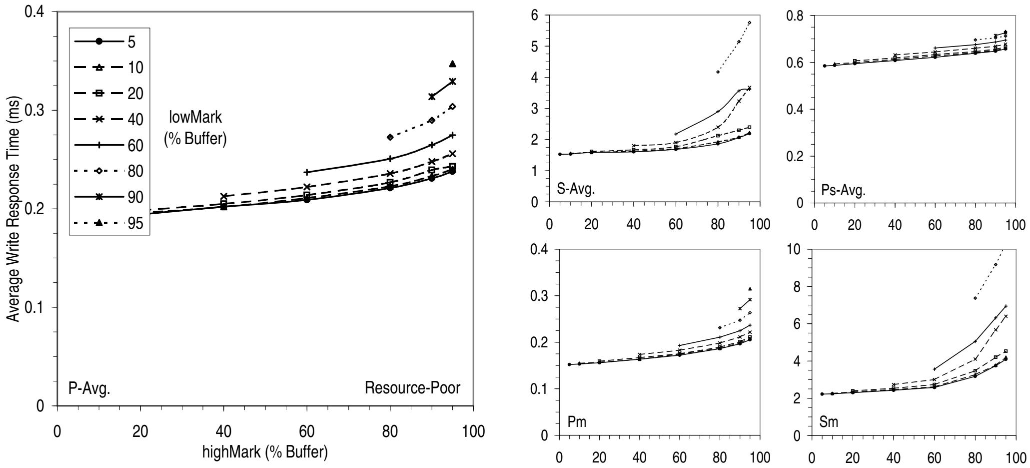


Figure A-20: Effect of $lowMark$ and $highMark$ on Average Write Response Time (Resource-Poor).

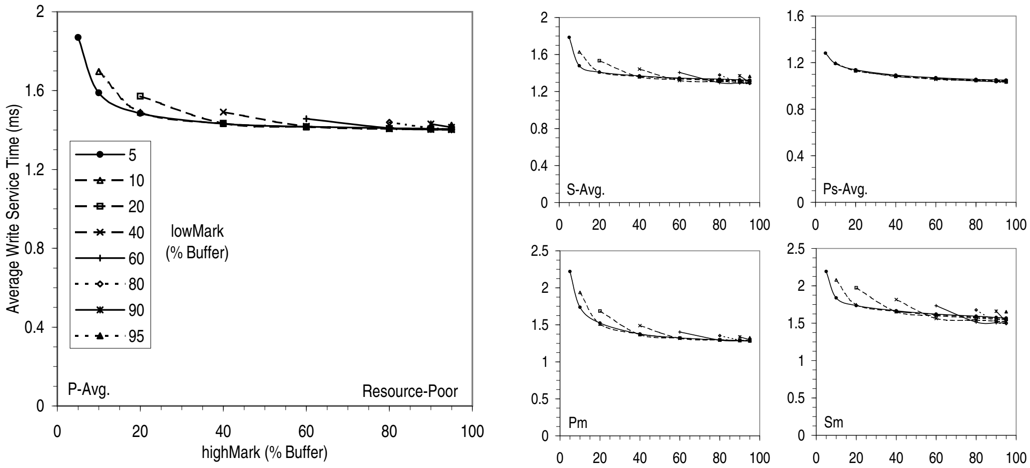


Figure A-21: Effect of $lowMark$ and $highMark$ on Average Write Service Time (Resource-Poor).

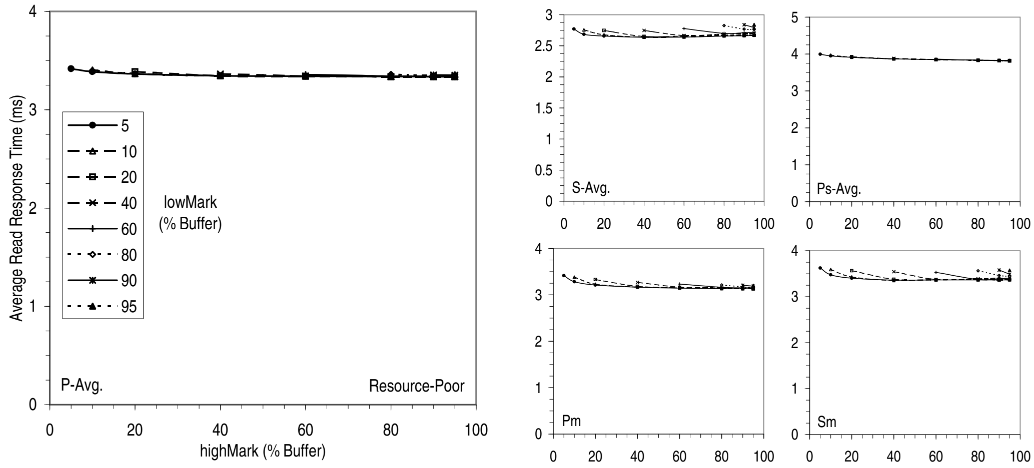


Figure A-22: Effect of $lowMark$ and $highMark$ on Average Read Response Time (Resource-Poor).

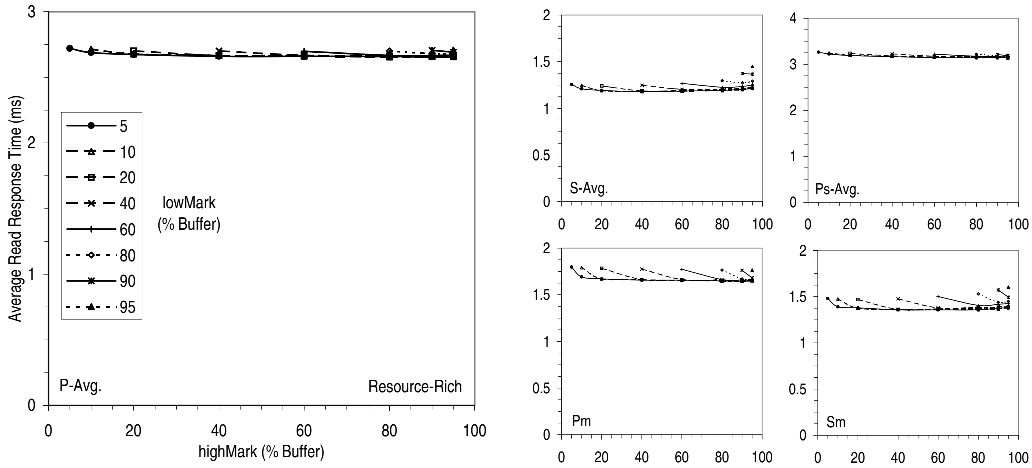


Figure A-23: Effect of $lowMark$ and $highMark$ on Average Read Response Time (Resource-Rich).

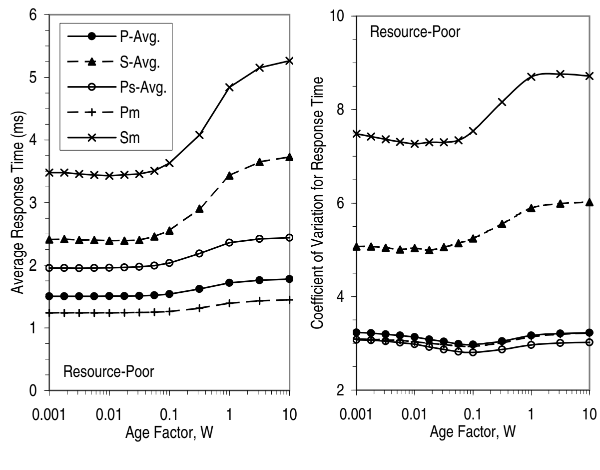
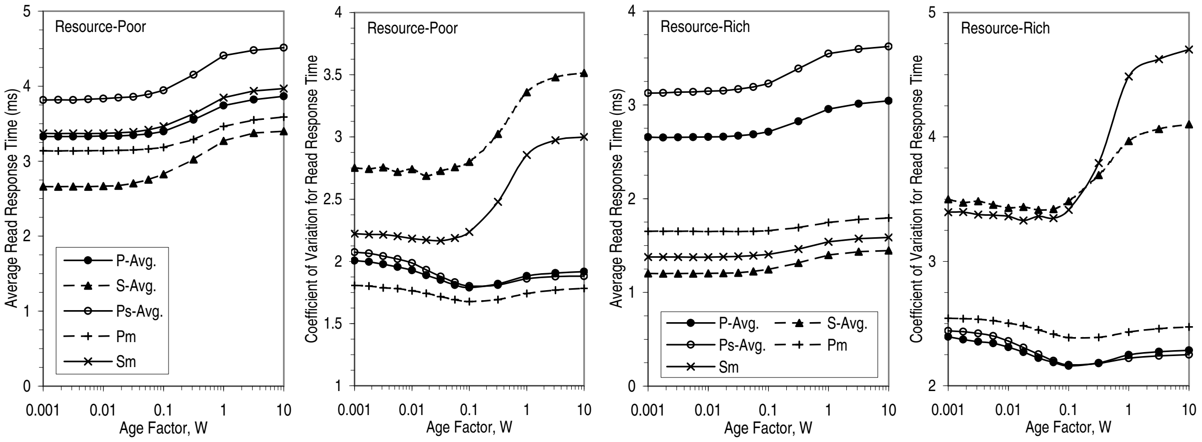
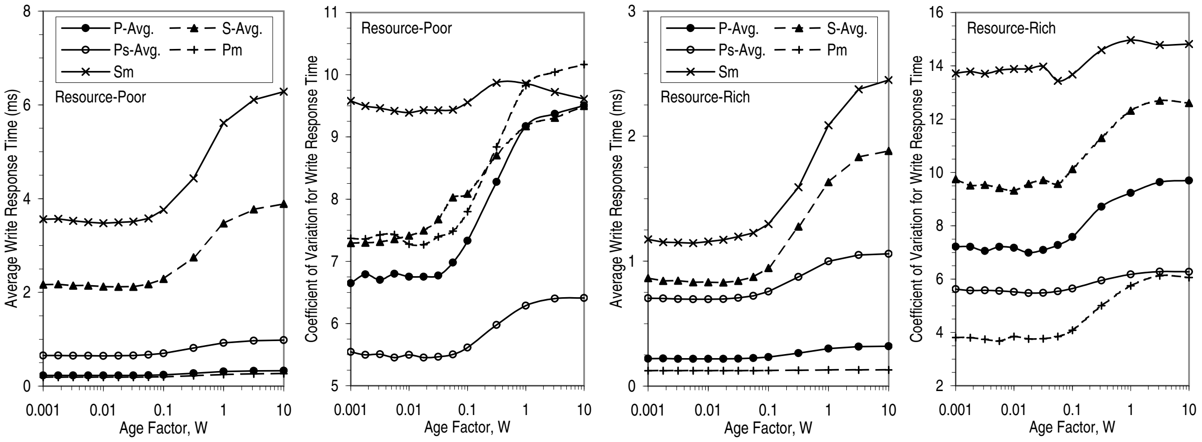


Figure A-24: Effect of Age Factor, W, on Response Time (Resource-Poor).

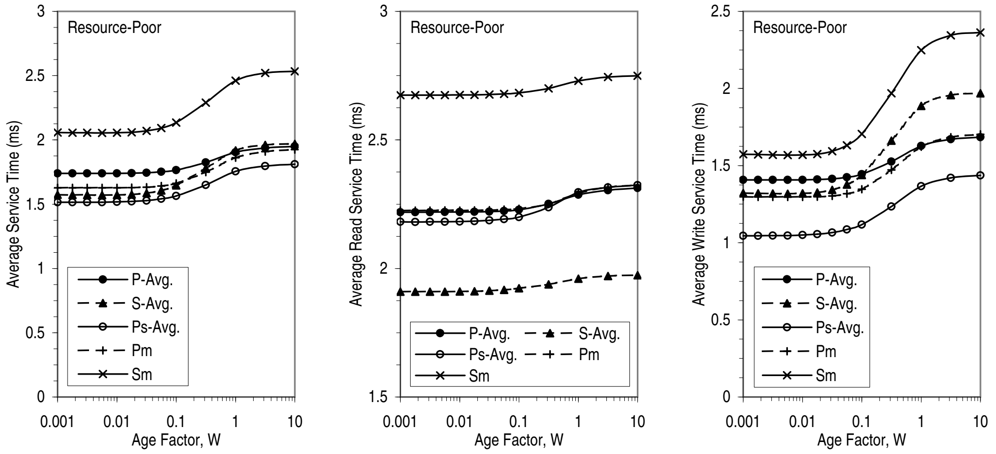


(a) Reads.

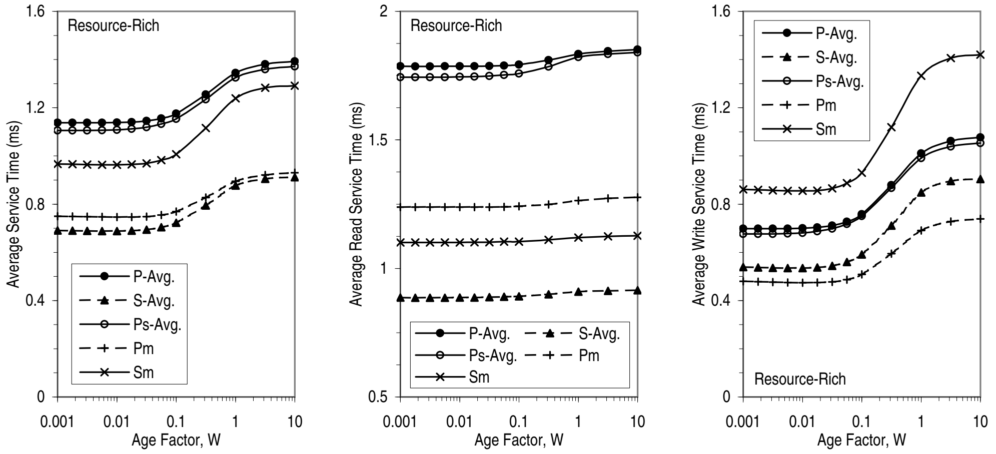


(b) Writes.

Figure A-25: Effect of Age Factor, W, on Response Time.

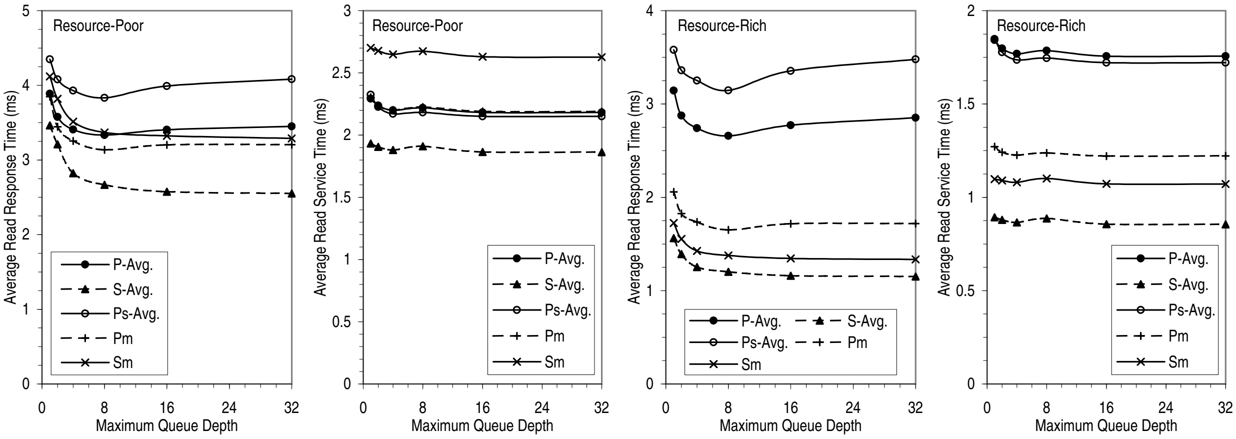


(a) Resource-Poor.

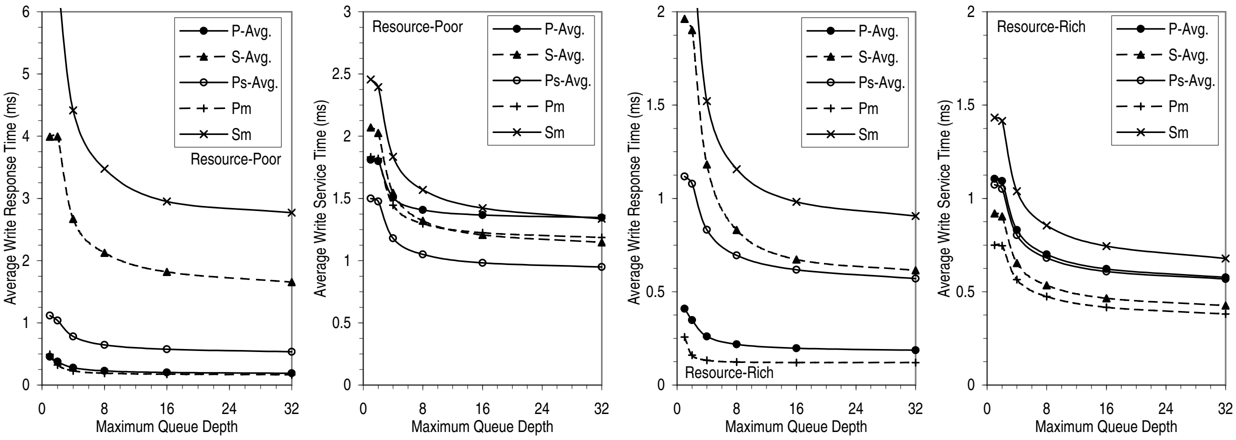


(b) Resource-Rich.

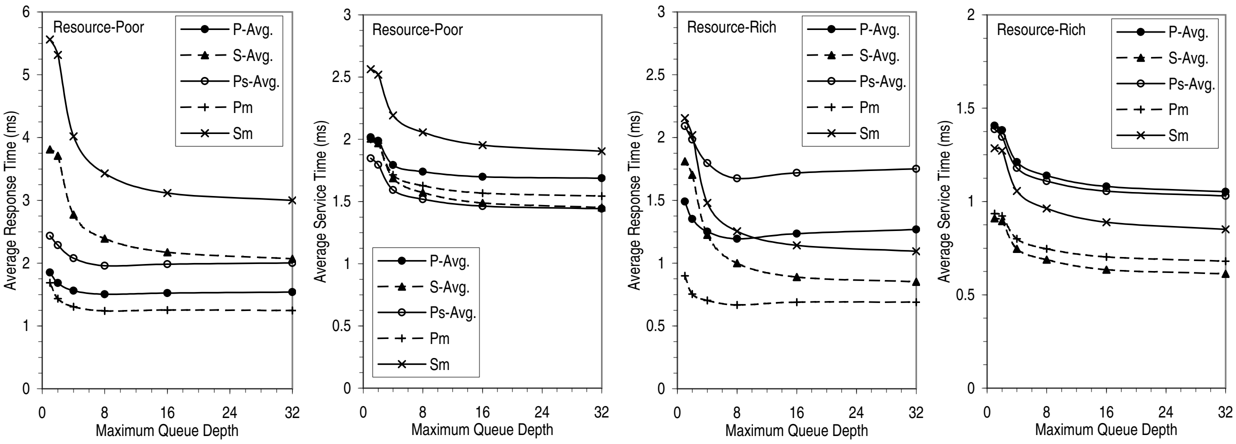
Figure A-26: Effect of Age Factor, W, on Service Time.



(a) Reads.



(b) Writes.



(c) Reads and Writes.

Figure A-27: Average Response and Service Times as a Function of the Maximum Queue Depth.

		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]).
Shortest Access Time First with Age Factor of 0.01.

(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]).
Shortest Access Time First with Age Factor of 0.01.

(b) Writes.

Table A-2: Average Response and Service Times as Maximum Queue Depth is Increased from One.

		Average Read Service Time							Average Write Service Time						
		Disk Capacity (Relative to Base)							Disk Capacity (Relative to Base)						
		2	3	4	5	6	7	8	2	3	4	5	6	7	8
Resource-Poor	P-Avg.	7.29	10.6	12.4	13.8	14.7	15.5	16.0	9.38	13.8	16.8	18.4	19.3	20.6	21.8
	S-Avg.	10.2	13.8	15.8	16.9	17.7	18.4	19.0	13.1	18.2	20.6	23.6	25.1	26.5	29.4
	Ps-Avg.	7.41	10.8	12.8	14.2	15.3	15.9	16.6	10.3	15.0	18.2	20.0	21.0	22.6	24.0
	Pm	7.44	10.1	11.9	12.8	13.8	14.6	15.3	11.5	16.4	19.3	20.4	22.5	23.6	24.9
	Sm	10.0	13.4	15.5	17.3	18.4	19.3	19.7	14.1	20.4	22.1	26.7	28.1	29.5	30.6
Resource-Rich	P-Avg.	7.09	10.3	12.0	13.3	14.3	15.1	15.5	10.2	14.9	18.1	20.0	21.6	22.9	23.9
	S-Avg.	9.45	13.0	14.7	15.7	16.5	17.1	17.7	14.3	20.4	22.2	25.8	27.9	29.2	30.3
	Ps-Avg.	7.36	10.6	12.5	14.0	14.9	15.7	16.3	10.5	15.2	18.3	20.3	22.0	23.4	24.3
	Pm	7.30	9.87	11.4	12.4	13.2	14.0	14.6	11.7	16.3	19.7	21.7	23.4	24.8	25.8
	Sm	8.90	12.2	14.1	15.9	16.9	17.8	18.2	14.6	20.7	21.3	26.9	28.9	30.5	31.6

Table A-3: Improvement in Average Service Time when Larger Capacity Disks are Used.

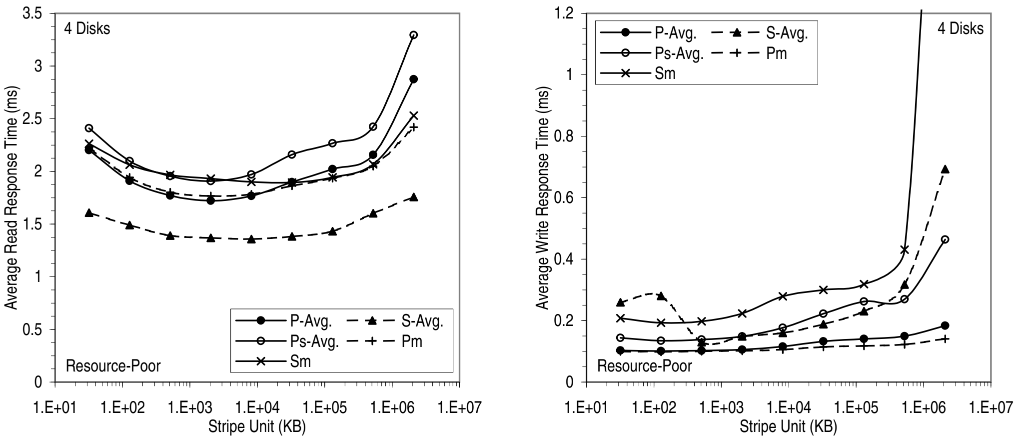
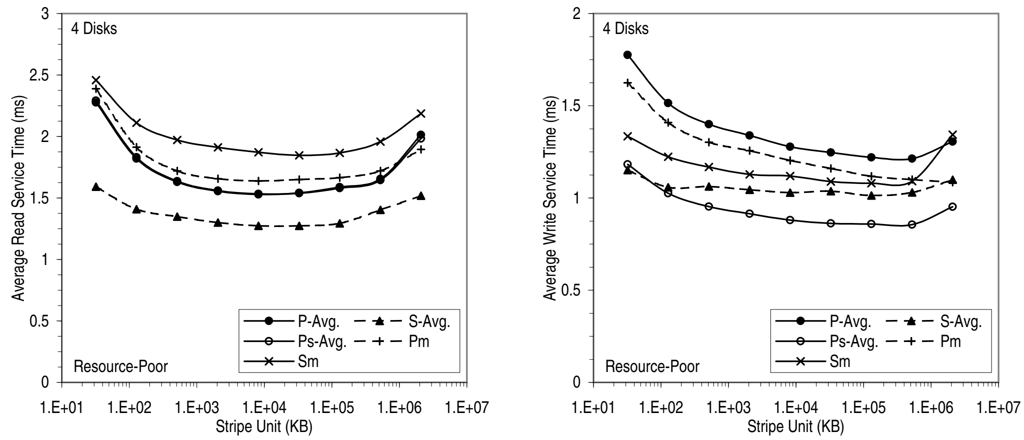
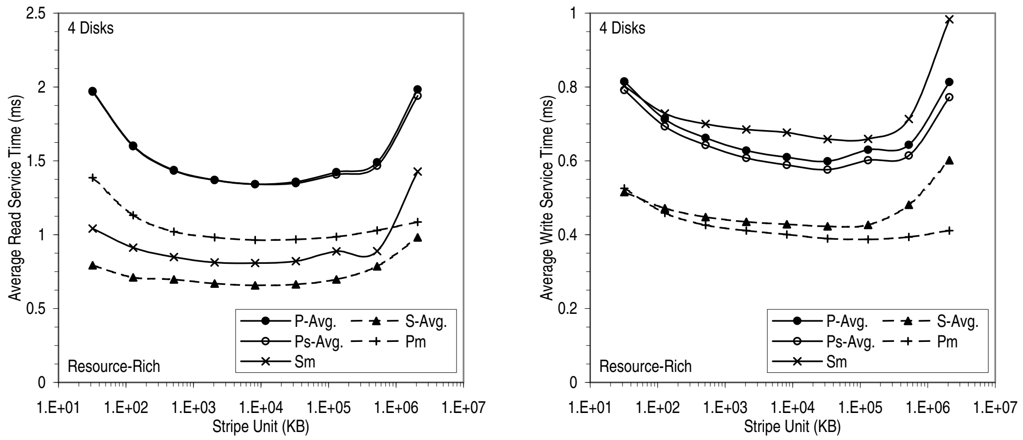


Figure A-28: Average Read and Write Response Time as a Function of Stripe Unit (Resource-Poor).

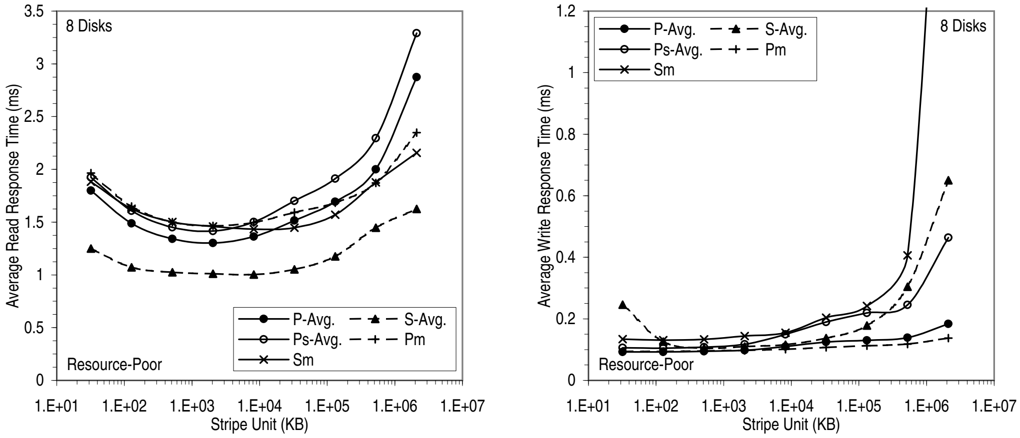


(a) Resource-Poor.

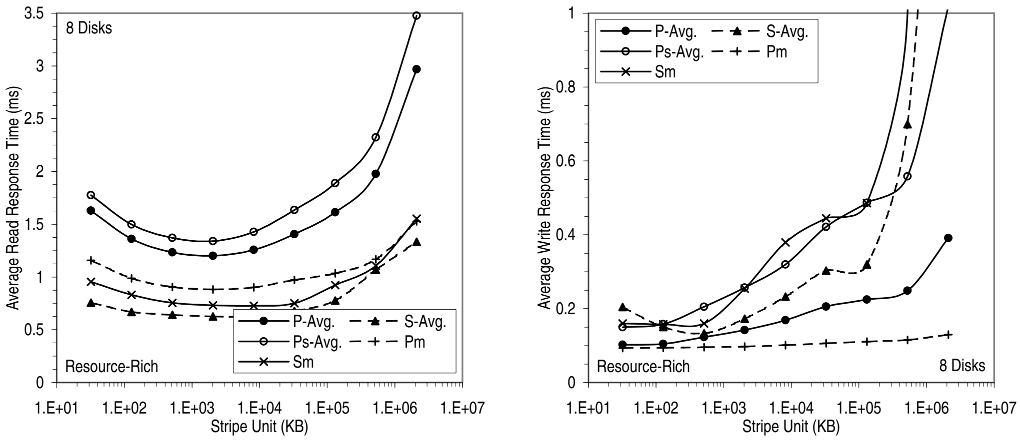


(b) Resource-Rich.

Figure A-29: Average Read and Write Service Time as a Function of Stripe Unit.

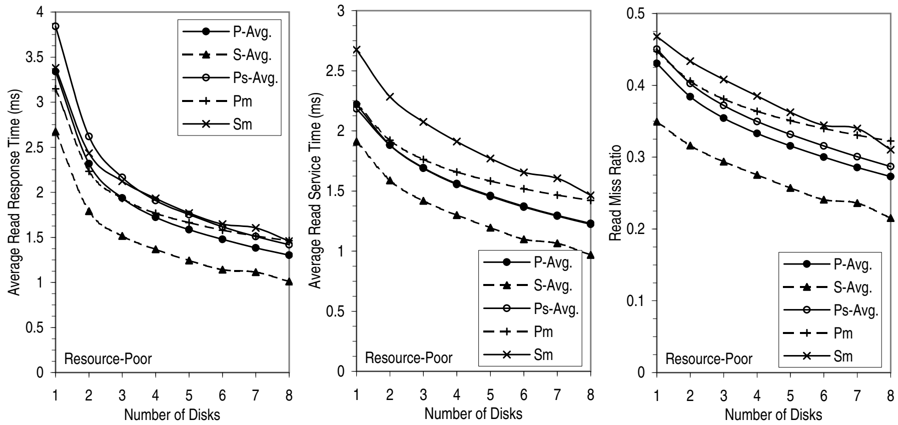


(a) Resource-Poor.

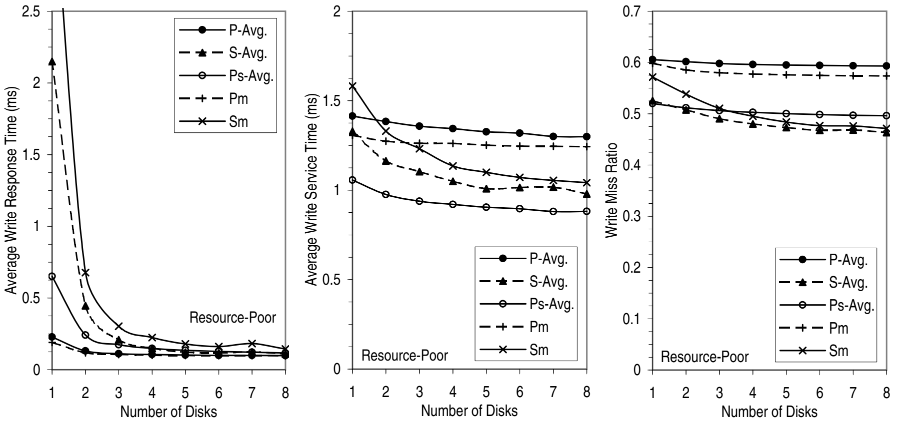


(b) Resource-Rich.

Figure A-30: Average Read and Write Response Time as a Function of Stripe Unit (8 Disks).



(a) Read.



(b) Write.

Figure A-31: Performance as a Function of the Number of Disks (Resource-Poor).

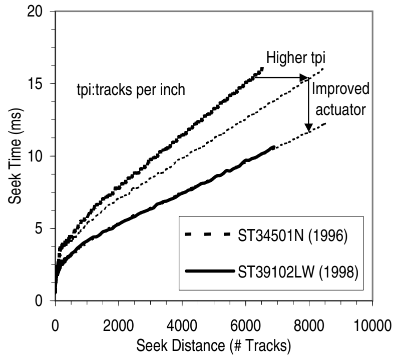
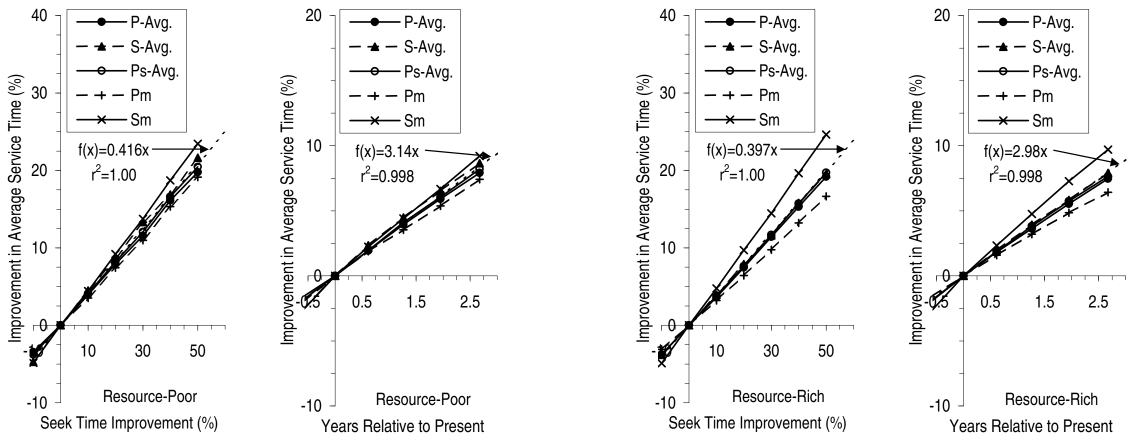


Figure A-32: Change in Seek Profile over Time.



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

Figure A-33: Effect of Improvement in Seek Time on Average Service Time.

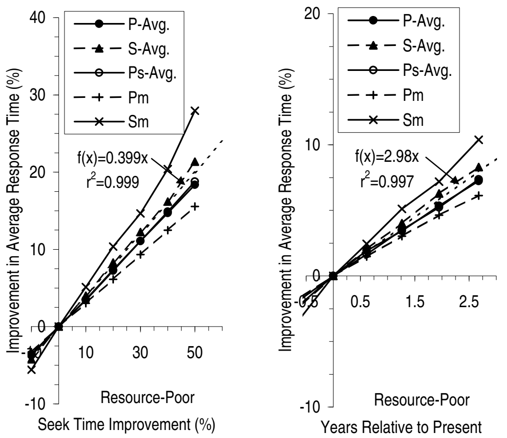


Figure A-34: Effect of Improvement in Seek Time on Average Response Time (Resource-Poor).

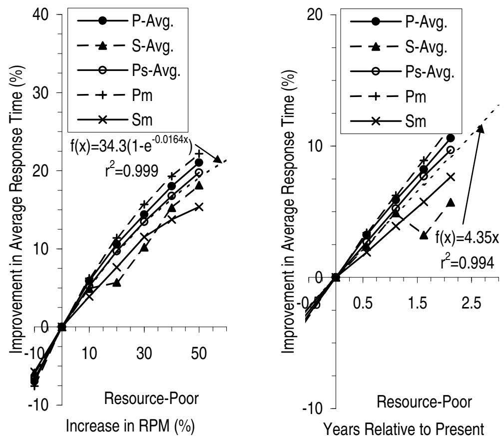
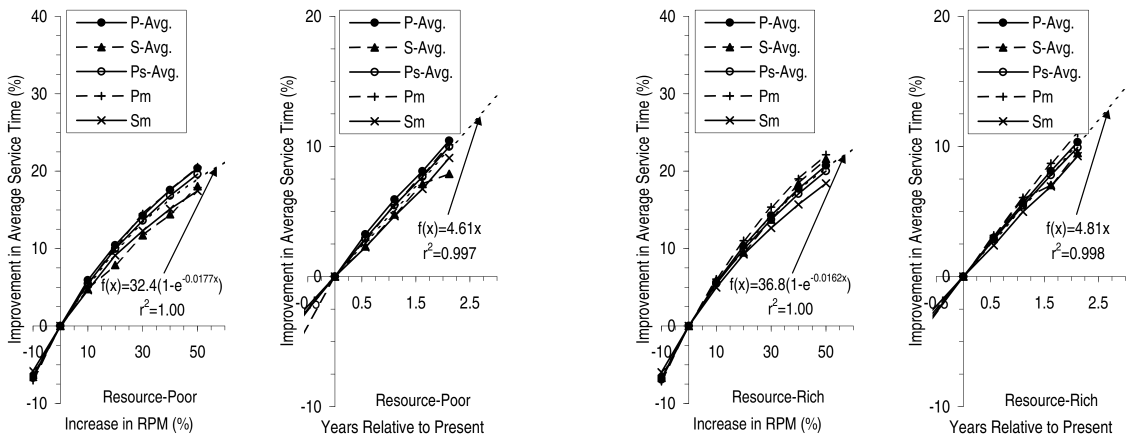


Figure A-35: Effect of RPM Scaling on Average Response Time (Resource-Poor).



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

Figure A-36: Effect of RPM Scaling on Average Service Time.

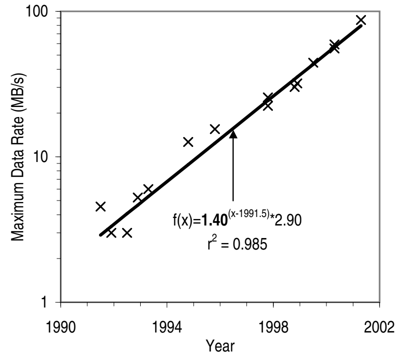
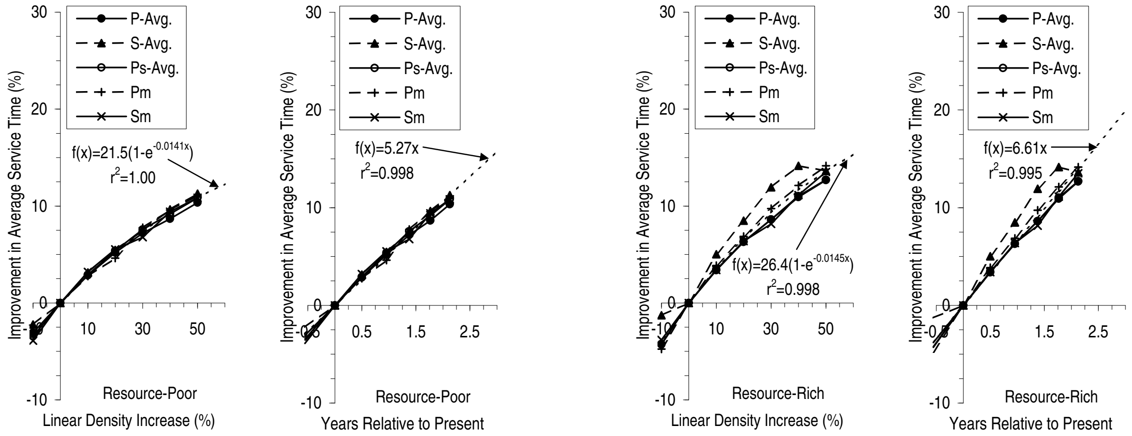


Figure A-37: Historical Rate of Increase in Maximum Data Rate (IBM Server Disks).



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

Figure A-38: Effect of Increased Linear Density on Average Service Time.

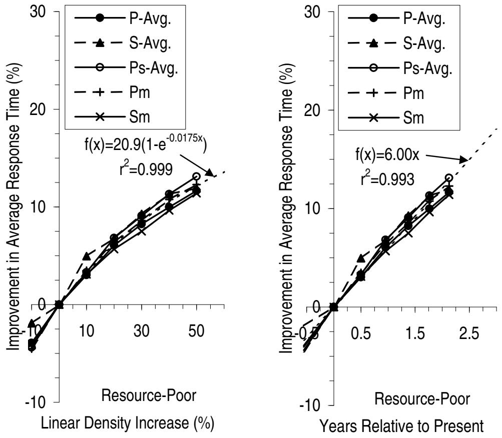


Figure A-39: Effect of Increased Linear Density on Average Response Time (Resource-Poor).

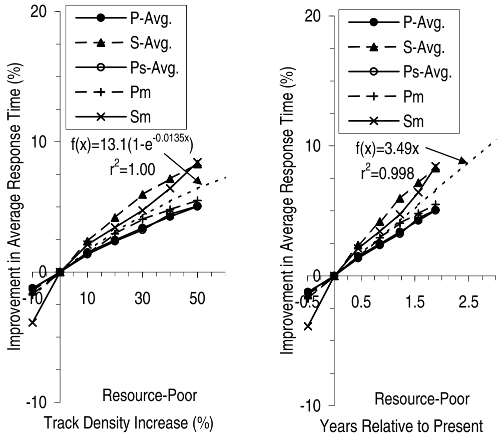
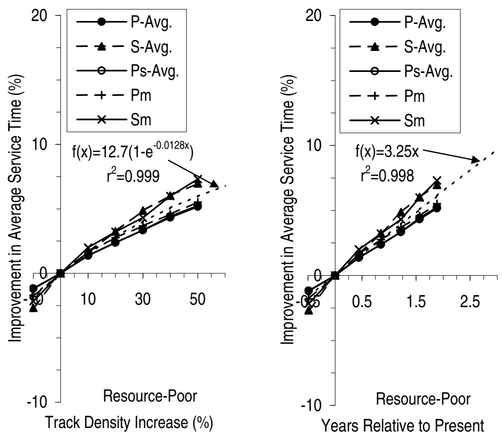
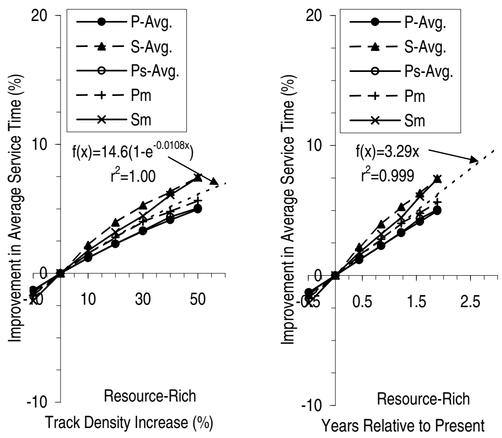


Figure A-40: Effect of Increased Track Density on Average Response Time (Resource-Poor).

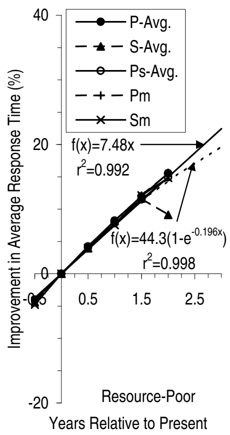


(a) Resource-Poor.

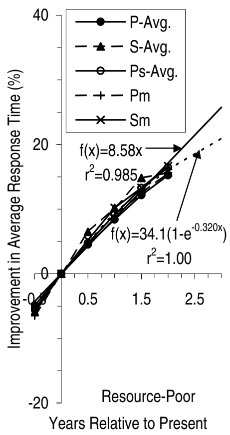


(b) Resource-Rich.

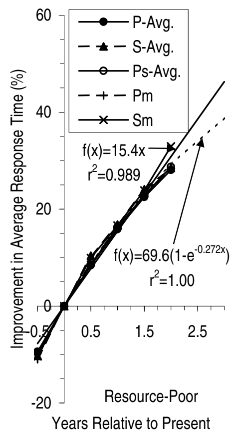
Figure A-41: Effect of Increased Track Density on Average Service Time.



(a) Mechanical Improvement.

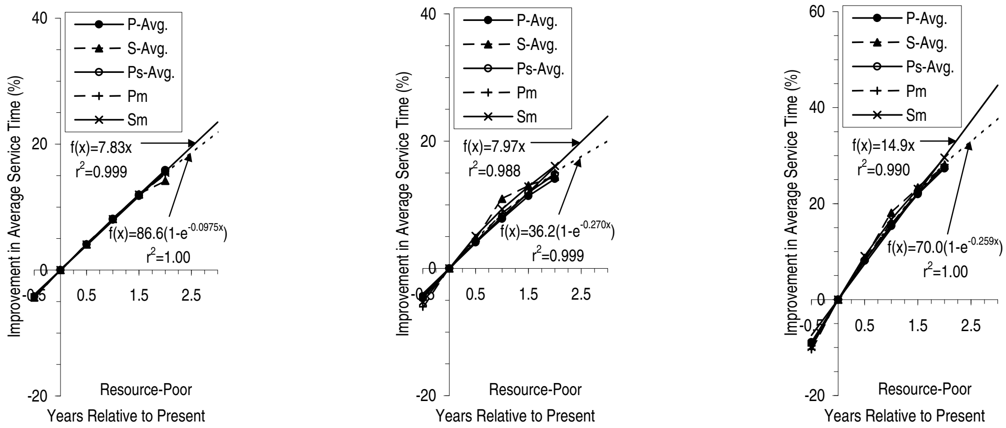


(b) Areal Density Increase.



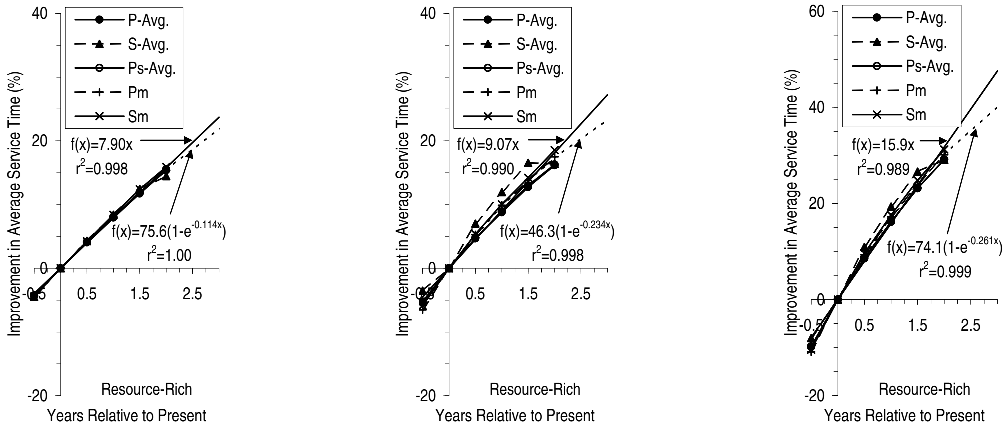
(c) Disk Technology Improvement.

Figure A-42: Effect on Average Response Time (Resource-Poor).



(a) Mechanical Improvement. (b) Areal Density Increase. (c) Disk Technology Improvement.

Figure A-43: Effect on Average Service Time (Resource-Poor).



(a) Mechanical Improvement. (b) Areal Density Increase. (c) Disk Technology Improvement.

Figure A-44: Effect on Average Service Time (Resource-Rich).

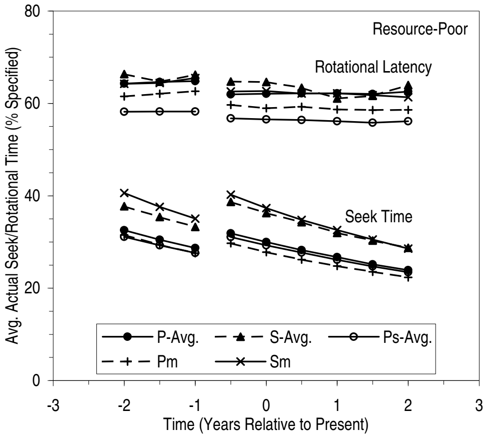


Figure A-45: Actual Average Seek and Rotational Time as Percentage of Manufacturer Specified Values (Resource-Poor).

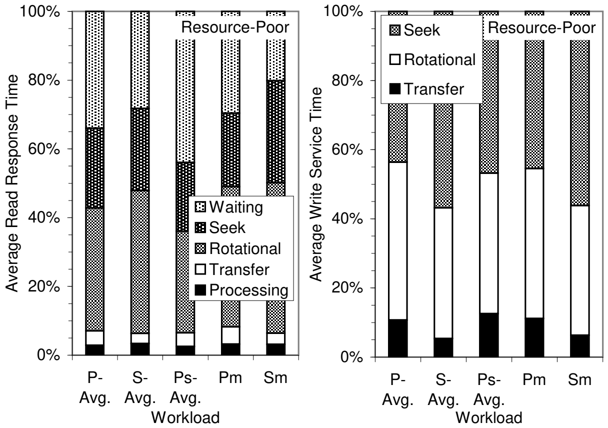
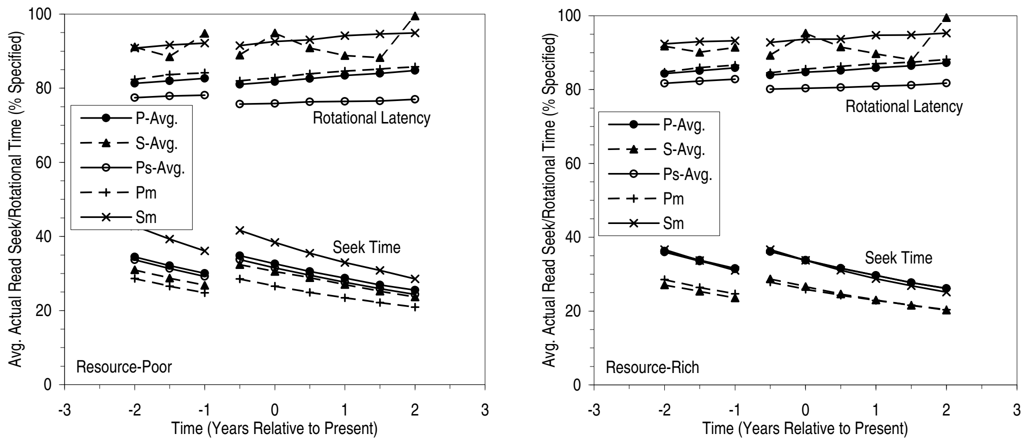
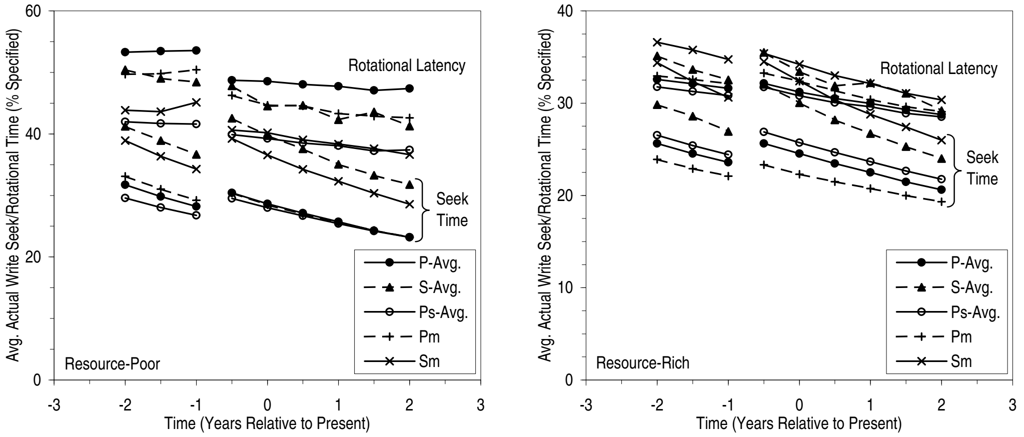


Figure A-46: Breakdown of Average Read Response and Write Service Time (Resource-Poor).



(a) Reads.



(b) Writes.

Figure A-47: Actual Average Seek/Rotational Time as Percentage of Specified Values.