DESIGN, MODELING, AND EVALUATION OF HIGH PERFORMANCE I/O SUBSYSTEMS

Shenze Chen

CMPSCI Technical Report 93-39

DESIGN, MODELING, AND EVALUATION OF HIGH PERFORMANCE I/O SUBSYSTEMS

A Dissertation Presented

by

SHENZE CHEN

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1992

Department of Computer Science

Dedicated to

My wife, Wen-Xiong Wu

and

My parents, Cong-Wen Zhao and Shi-Jing Chen

ACKNOWLEDGMENTS

First of all, I wish to thank Prof. Don Towsley, my thesis advisor, for his invaluable support and guidance during the course of this research. I have greatly benefited from his talent and skills when doing my research. He was always enthusiastic and accessible. He encouraged me to pursue new ideas and helped me to solve technical problems. I thank him for his continuous support, his hours, and his friendship throughout my years at UMass.

Special thanks are due to Prof. James F. Kurose and Prof. John A. Stankovic. During the course of my research, I have had a great pleasure to work with both of them. They always provided me excellent advice and spent a tremendous number of hours discussing ideas and methodologies with me. When serving as my thesis committee members, Jim and Jack read my thesis with extreme care and corrected a number of technical and grammatical errors. I would also like to thank Prof. C. Mani Krishna, my thesis committee member, for his advice and feedback, which were invaluable to my thesis. Thanks also go to Prof. Krithi Ramamritham for his constructive comments on a draft which constitutes a part of this dissertation.

I sincerely thank all members, old and new, in the *Networks and Performance Modeling Laboratory* at UMass. All of their efforts have provided me with a friendly, stimulating, and enjoyable research environment. I wish to thank Henning Schulzrinne, Ramesh Nagarajan, Ren-Hung Hwang, David Yates, Jayanta Dey, Shridhar Pingali, Wei Chen, and Jim Salehi for their help and friendship. I would also like to thank Rahul Simha for his warm friendship which continues even after his leaving UMass. Special thanks also go to Jim Huang, who provided me valuable data collected from the RT-CARAT testbed.

Finally, I wish to thank my wife, Wen-Xiong Wu. She has supported me with her love throughout the years that I struggled in graduate school. She shared my difficult times as well as my enjoyable times. I also thank my parents for their con-

tinuous encouragement and support. All of their efforts have made this dissertation possible.

DESIGN, MODELING, AND EVALUATION OF HIGH PERFORMANCE I/O SUBSYSTEMS

A Dissertation Presented

by

SHENZE CHEN

Approved as to style and content by:

Don Towsley, Chair

James F. Kurose, Member

John A. Stankovic, Member

C. Mani Krishna, Member

Arnold L. Rosenberg, Department Head
Department of Computer Science

ABSTRACT

DESIGN, MODELING, AND EVALUATION OF HIGH PERFORMANCE I/O SUBSYSTEMS

SEPTEMBER 1992

SHENZE CHEN

B.S., SHANGHAI MARITIME INSTITUTE
M.S.E., SHANGHAI JIAO TONG UNIVERSITY
M.S., UNIVERSITY OF MASSACHUSETTS
Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor Don Towsley

In today's computer systems, the disk I/O subsystem is often identified as a major bottleneck to system performance. Removing this bottleneck has proven to be a challenging research problem. The research reported in this dissertation is motivated by the design and performance issues of disk I/O subsystems for a variety of today's computing systems. The goal is to design and evaluate high performance I/O subsystems for computing systems which support various applications. Our contributions are three-fold: for different application areas, we (1) propose new architecture and scheduling policies for the disk I/O subsystems; (2) develop analytic models and simulators for these disk subsystems; and (3) study and compare the performance of these architectures and scheduling policies.

First, we study a mirrored disk which can be found in various fault tolerant systems, where each data item is duplicated. While the primary purpose of disk mirroring is to provide the fault tolerance, we are interested in how to achieve performance gain by taking advantage of the two data copies. In particular, we

propose and examine several policies which differ according to the manner in which read requests are scheduled to one of the two copies. Analytic models are developed to model the behavior of these policies and the best policies are proposed. In addition, the modeling techniques developed in this study is of independent interest, which can be used or extended to other system studies.

Second, we investigate existing and propose new disk array architectures for high performance computing systems. Depending on the applications, we propose scheduling policies suitable for these architectures. In particular, we study three variations of RAID 1, *mirrored declustering*, *chained declustering*, and *group-rotate declustering*. We compare the performance of RAID 5 versus Parity Striping. Finally, we examine the performance/cost trade-off of RAID 1 and RAID 5. The performance studies of the various disk array architectures coupled with the proposed scheduling policies are based on our analytic models and simulators.

Third, we examine the disk I/O subsystems for real-time systems, where each I/O is expected to complete before a deadline. The goal is to minimize the fraction of requests that miss their deadlines. In doing so, we first propose two new real-time disk scheduling algorithms, and compare them with other known real-time or conventional algorithms in an integrated real-time transaction system model. We then extend this study to a mirrored disk and disk arrays by combining the real-time algorithms with the architectures and policies studied before. Last, we study disk I/O in a *non-removal* real-time system environment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In the late 1960's, Gene Amdahl put forth a notion, which was later known as "Amdahl's Law", that if a computer system consists of several components, and if some of these components are made much faster while leaving other components unchanged, then the unimproved components are likely to become a performance bottleneck [4]. Unfortunately, this phenomenon has been observed in many of today's computer systems in the existence of a large gap between the speeds of the CPU, the main memory, and the I/O devices. The I/O bottleneck has resulted in a great obstacle to the improvement of computer system performance.

This problem is compounded because, as technology advances, a great variety of applications have been developed. For each of these applications, I/O carries certain specific requirements and is characterized by certain properties. Building a disk I/O subsystem so as to satisfy these special requirements and to achieve high performance is a challenging issue.

The research reported in this dissertation is motivated by the design and performance issues of disk I/O subsystems for a variety of today's computing systems. The goal is to design and evaluate high performance I/O subsystems for computing systems which support various applications. In particular, we study a mirrored disk for fault tolerant systems, disk arrays for high performance transaction processing, scientific computations and image processing, and disk subsystems for real-time systems. The contributions are three-fold: for each of these applications, we (1) propose new architecture and scheduling policies for the disk I/O subsystems; (2) develop analytic models and simulators for these disk subsystems; and (3) study and compare the performance of these architectures and scheduling policies.

The remainder of this chapter is organized as follows: we first describe the I/O crisis, identify the special I/O requirements of different systems and applications,

and address the issues in designing high performance disk I/O subsystems for such systems. Next, a brief review of previous related work is presented. We then list the main contributions of this thesis work, and conclude with an overview of the remainder of this dissertation.

1.1 The I/O Crisis

During the past several years, CPU speed has increased at a rate of 40% to 70% each year [14, 54] and the distinctions between supercomputer, mainframe, minicomputer, and workstation have blurred considerably. For example, today's high performance workstations can achieve 20 to 50 MIPS, whereas the original CRAY-1 achieved only 33 scalar MIPS. Improving VLSI technology also leads to faster and larger main memories; state-of-art workstations have been equipped with main memories of 8-120 megabytes with an access time of 20-80 ns. However, during the same period, disk seek times have only improved by 7% per year and the rotation speeds have remained almost unchanged [46, 87]. Currently, disks are at least four orders of magnitude slower than main memory [84]. Moreover, while CPU and main memory speeds continue to increase, it is unlikely that the access time of magnetic disks will significantly decrease in the near future because of their mechanical nature. Based on these observations, it can be expected that the gap in CPU and I/O speeds will become even larger and as a result, further increases in the processor speed will bring little or no gain to the overall system performance. For example, suppose that an application program today spends 10% of its time waiting for I/O. If CPU speed improves by a factor of 10 without any disk improvements, then the application will only run about 5 times faster and will spend about 50% of its time waiting for I/O. If CPU speed improves by another factor of 10, still without disk improvements, then the application will only run about twice again as fast, and will spend about 90% of its time waiting for I/O. The hundred-fold overall improvement in CPU speed will only result in a ten-fold improvement in application speed [87]. This presents the following serious problem: given the current foreseeable future magnetic disk technology, how can one reduce the speed gap between the CPU, main memory and the I/O subsystems?

1.2 Disk I/O for a Variety of Computing Systems

In this section, we list some of the computer systems and application areas which are of particular interest to us, and briefly describe their associated I/O requirements and I/O characteristics.

Fault Tolerant Systems:

In many computer systems, high reliability is required and of critical importance. This requirement has spurred considerable interest in designing fault tolerant computer systems. These systems are typically designed to provide redundancy so as to survive single component failures [11, 56, 45]. Because of the mechanical nature of disk drives, they are one of the weakest components in a computer system. This forms the basis behind the concept of *mirrored disks*, whereby each data item is stored on a pair of disks. The presence of the additional disk introduces new issues to be addressed in the design of appropriate architectures. Since there are two data copies available, a mirrored disk may support two read requests concurrently, while a single disk can serve only one request at a time. On the other hand, a write has to be performed on both disks.

Transaction Processing Systems:

Transaction processing systems support a wide variety applications, such as airline reservation systems, banking systems, library indexing systems, etc. I/O for transaction systems is typically characterized by large request rates. Each request randomly accesses the disk and transfers a small amount of data [41, 75]. For example, in a banking system, it is quite common that many customers begin their transactions at different teller machines, each withdrawing or depositing a certain amount of money from/to his/her own account. Therefore, the disk I/O subsystem for a transaction system should be designed to efficiently handle a large volume of small random I/O requests so as to speed up the transaction processing. Current databases are growing larger and larger, and therefore the disk subsystems are required to sustain hundreds or even thousands of I/O requests per second.

Engineering/Office Applications:

In [87], Ousterhout and Douglis used the term *engineering/office applications* to describe the computing environment in which programs access large numbers of small files, such as source files for a program or font libraries for a laser printer. Many program development and word processing environments fall into this category. A typical example is the UNIX[1] environment, in which multiple users read several small files, edit them and then write them back to the disk. Similarly, program executions may open several files, perform a sequence of reads or updates on them, and then close these files.

Scientific Applications:

This category includes supercomputer systems and image processing systems. As observed by Bucher and Hayes [19], supercomputer I/O can be characterized almost entirely by sequential I/O. Since multiprogramming levels are lower in a supercomputing or image processing environment, fewer I/O requests are issued per unit time. However, computation parameters are typically moved in bulk from disk to in-memory data structures, and results are periodically written back to disk [55]. Thus, the I/O access patterns for scientific applications have the property of large sequential transfers.

Real-Time Systems:

Recently, interest in real-time systems has been growing rapidly. In a real-time system, each entity (job, process, transaction) carries a deadline when submitted to the system. If an entity cannot complete its task before its deadline, it is said to be lost. Generally, two categories of real-time systems are identified. In a *hard real-time* system, missing a deadline may lead to a catastrophe [110], while in a *soft real-time* system, missing a deadline only reduces the 'value' of the entity to the system [53]. In this dissertation, we are more interested in *soft real-time* systems, such as real-time transaction systems.

The performance criteria for soft real-time systems are quite different from those of conventional systems. The performance metric of interest is the loss ratio, i.e.,

[1]UNIX is a trademark of AT&T Bell Laboratories.

the fraction of jobs that miss their deadlines before completing services, rather than system throughput and device utilization. Because each entity has a time constraint, timing is a critical factor in real-time systems. Hence, I/O for real-time systems needs also to take into account the time constraints of entities.

1.3 Problem Formulation

In this dissertation, we will address various questions in designing high performance reliable I/O subsystems for a variety of computing systems.

Disk Mirroring:

As stated before, disk mirroring, where each data item is stored on a pair of disks, has found widespread use as a mechanism for providing fault tolerance. Read requests can be satisfied by either disk, whereas write operations have to be performed on both disks. Thus, disk mirroring has the advantage of supporting multiple reads in parallel. In addition, by allowing a read request to access that disk whose arm is currently positioned closest to the target data, the access time can be reduced. Both of these actions have the effect of increasing the disk bandwidth and decreasing I/O response times. The question of interest is how to schedule I/O requests to a pair of mirrored disks so as to achieve higher performance. We address these issues in Chapter 3.

Disk Arrays:

An attractive idea for reducing the CPU-IO gap is the so called "disk array", where the disk I/O subsystem consists of multiple disks with data spread over them. When a request needs to access a large amount of data, the array can initiate multiple disks simultaneously to serve the request. On the other hand, if less data is required by each request, so that fewer disks are accessed, the disk array can provide concurrent service to multiple requests. Since "small" disks for PC's and workstations are becoming much cheaper and physically smaller, it is practical to build disk systems containing tens or hundreds of such disks. Currently, workstation disks may have capacities of 500M-1G bytes. Thus a disk array consisting 100 such disks may yield a 50G-100G bytes of storage capacity.

A problem associated with disk arrays is the reliability of disk subsystems. As noted in [104], a disk array built from 49 small disks (ample capacity to replace conventional disks), each with a MTTF of 40,000 hours, yields an overall MTTF of only 816 hours. This is far less than the MTTF of conventional mainframe disks (50,000 hours) with the same capacity. Thus, disk arrays must be redundant so as to reconstruct data loss in a failure, and the recovery should be transparent to the users.

Although the ideas behind the disk arrays are simple, there are many challenging issues that need to be addressed when building a disk array, such as the choice of architecture, how to layout data on the array, and how to schedule I/O requests to the target disks. We address these, and other, issues in Chapter 4.

Disk I/O for Real-Time Systems:

In a real-time system, each computational entity has to be completed within certain amount of time, and thus timing is the most important factor to be considered in system design. As stated before, I/O devices are orders of magnitude slower than CPU and memory speeds, and therefore the improvement in I/O efficiency is extremely important to the performance of a real-time system. In such a system, various I/O requests may carry different deadlines. Some of these requests might be "impending", i.e., their deadlines are very close, while some others are not. Therefore it is reasonable to assign a higher priority to those "impending" I/O requests so as to let them be served first.

In a real-time disk I/O context, an important issue is how to assign priorities to I/O requests so that the request loss ratio is minimized. This is quite different from the conventional arm scheduling algorithms such as the *shortest seek time first* (SSTF) or the *elevator* algorithm, whose purpose is to optimize the arm move distance and therefore reduce the disk service time. Here, in a real-time environment, scheduling policies should concerned not only with the reduction of service time but also with the time constraints carried by different requests.

1.4 Related Work

In the past three decades, a great amount of effort has been spent on the design of disk and I/O subsystems, and it is impossible to treat it exhaustively. Instead, we will focus on the related work which has been reported in the literature on the topics introduced above.

Disk Mirroring and the Fork/Join Queueing System Model

There is little work which has appeared in the literature on the design and evaluation of a two-disk mirrored or disk shadowed systems. Matloff [69] developed an approximate analysis of a different form of disk redundancy where *three or more disks* are used to maintain two copies of each data item. The performance of this system is shown to be better than that of a single disk that maintains a single copy of each data item. Bitton and Gray [16] studied a shadowed disk system consisting of K disks. They proposed that a read request access the disk which requires the minimum arm movement; the authors presented an approximate analysis of the mean seek time. Later in [17], Bitton also provided simulation results on the mean seek distance. However, the analysis in [16] assumed "zero-load", i.e., each request sees an idle disk system and can start service immediately without queueing delay.

Other related work can be found in the area of replicated database systems. Using the matrix geometric methodology developed by Neuts [81], Nelson and Iyer [79] described and analyzed two protocols for a replicated database that correspond to two single queue policies that we studied in [117]. Abbadi *et al.*[39] presented a fault-tolerant protocol for distributed database systems, which employed the same idea as Bitton: to read a logical object from the *nearest available* physical copy of that object. Recently, Matloff and Lo [70] studied a shadowed disk system in which each data item is assumed to be stored in the same location on all k shadowed disks and I/O requests are served sequentially. In order to improve the read performance, they suggested that after each request is completed, the arm of all idle drives move to positions calculated to minimize the seek time of the next request. Other work on the performance evaluation of replicated database systems can be found in [33, 32, 7].

It is useful to observe that the disk mirroring system corresponds very closely to the fork/join queueing system which has received considerable attention in recent

years. Specifically, a mirrored disk can be modeled as a fork/join queueing system consisting of two servers serving a mixture of regular and fork/join customers. A read request corresponds a regular customer which can be served by either server, and a write request corresponds to a fork/join customer which must be served by both servers.

Flatto and Hahn [40] performed an exact analysis of a system with two exponential servers. Fork/join customers arrive according to a Poisson process and the complete system is modeled by a Markov chain containing two state variables each of which is unbounded. They obtain the stationary probability distribution for this chain by transforming the problem into a boundary value problem. Rao and Posner [95] performed an approximate analysis of this model by truncating one of the two state variables and solving the resulting model using the matrix-geometric methodology developed by Neuts [81]. We will show that their approximation to the response time distribution for fork/join customers provides a lower bound on the correct distribution. In [58], Kim and Agrawala presented an approach to obtain approximate solutions for a K-server fork/join queueing system, in which the mean response time of fork/join customers is obtained by tracking the *virtual waiting time* of each queue, i.e., time to empty a server when no more arrivals occur. These three studies only considered systems serving fork/join customers.

Using a different approach, Nelson and Tantawi [80] developed an accurate approximation for the case of K identical servers, which is easily extended to include regular customers as well. Baccelli, Makowski, and Shwartz [8] developed simple (but loose) computational bounds for a K-server system that processes several classes of customers that differ from each other according to the subset of servers required. This approach can be used to model one of our distributed policies, termed DP-RJ (see Section 3.1), in which regular customers are randomly routed to one of the two servers. However, with the exception of the approach used by Rao and Posner [95], none of the approaches appear to generalize easily to policies other than the aforementioned DP-RJ policy.

Other work relevant to one of our distributed policies, DP-SQ (see Section 3.1), can be found in [96], in which Rao and Posner presented an approximate analysis of

the shortest queue model. However, only regular customers are accounted for in their model.

Finally, if read requests (regular customers) are given higher priority than write requests (fork/join customers), the overall mean I/O response time can be expected to be improved. Readers are referred to [5] for a discussion on this topic.

Disk Arrays

Since the ideas of "disk interleaving" and "disk striping" were first introduced by Kim [59] and Salem *et al.* [101], tremendous effort has been directed to disk array research. In [59], Kim studied a synchronized disk array, in which data are *byte interleaved* across multiple disks which work together as one logical device with a fast transfer rate, whereas in [101], Salem and Garcia-Molina considered the effect of *block interleaving* which they called "striping", where the stripe width is defined as the number of disks over which data are spread. Ng, Lang, and Selinger [84] discussed some of the design options of disk arrays, so as to provide a better understanding of the trade-offs for some of these choices and to point out some pitfalls to be avoided. In another paper [82], Ng considered the trade-offs between the number of devices and the number of data paths. The performance advantages and limitations of alternative implementations were analyzed under the assumption of byte-level interleaving. Kim and Tantawi [60] investigated asynchronous disk interleaving, where the access delay (seek, rotation, and transfer time) of a request directed to n disks is the maximum of n accesses. By using an approximate analysis, a simple expression for the mean value of such a maximum delay was obtained, which in turn was verified by simulating trace data. Again, queueing delays were ignored in this study.

Based on previous discussions, Patterson *et al.* [92, 91, 55] classified 5 levels of RAID (*Redundant Arrays of Inexpensive Disks*). They differ from each other according to the manner in which redundancy is added to provide fault tolerance. RAID 1, called *mirrored array* or *mirrored declustering*, maintains two copies for each data item. RAID 2 and RAID 3 adopt *bit* or *byte interleaving* across the stripe. RAID 2 employs a Hamming code error correction which, in addition to data disks, needs one check disk for detecting a single error but more check disks if error correction is to be performed. RAID 3 assumes that a disk failure can be detected immediately

by the disk controller, and therefore only a single parity bit is needed to reconstruct the lost data. This can reduce the number of check disks to one. *Block interleaving* is adopted for RAID 4 and RAID 5, which allows concurrent I/O to occur within a stripe and avoids the necessity of mechanically synchronized disks. While RAID 4 places all parity bits in a check disk, the RAID 5 tries to overcome the potential bottleneck of the check disk by distributing parity blocks over the stripe in a rotated manner. Among the five, they concluded that the most promising candidates for high performance computing systems are the *mirrored disk array (RAID 1)* and the *rotated parity array (RAID 5)*. Later in [42], Gibson further discussed some disk array architectures which can tolerate two disk failures by adding more redundant information.

The performance evaluation of disk arrays is an important issue to both designers and users. Livny, Khoshafian, and Boral [66] compared two schemes to place data on multiple disks: *declustering* which spreads each file across several disks and *clustering* which places each file on a single disk. They showed that the former is inherently better than the latter. In [97], based on simulation experiments, Reddy and Banerjee discussed the performance of several disk array configurations, such as disk synchronization, data declustering/disk striping, and a combination of these approaches. They also examined the effect of block size and other parameters of the disk array system. Ogata and Flynn [85] considered similar problems by using a simple $M/G/1$ queueing model, and their results agree with those of Reddy and Banerjee in [97]. In all of these three studies, however, reads and writes were treated in the same way and therefore the high cost suffered by write requests which access part of disks in a stripe was ignored. In fact, in order to allow a disk array consisting of tens or hundreds disks to work properly, the array must be redundant so that the data lost due to a device failure can be reconstructed automatically. As we will see later, different fault tolerant techniques may have a great impact on the system performance.

Olson [86] reported an experiment performed on Prime EXL^{TM} 316 (a 16 MHz 80386 system) containing up to 4 disks. By using the Prime $IOBENCH^{TM}$ benchmark, which defined a transaction with seven random reads and one write, each of

which accesses 1K bytes, they found that both RAID 1 and RAID 5 have acceptable performance in a read environment, while RAID 5 degrades significantly in an update environment. Another experiment was reported by Peter Chen *et al.* [24] on an Amdahl 5890 mainframe with up to 20 disks (stripe width = 10) executing a synthetically generated workload. The performance of the mirrored array (consisting of 20 disks) was compared with RAID 5 (consisting of 11 disks) in terms of throughput per disk. As a result, they concluded that for applications in which I/O requests typically access small amount of data, such as transaction processing systems, RAID 5 is outperformed by RAID 1, but for applications with predominantly large transfers, RAID 5 clearly outperforms RAID 1.

It is interesting to observe that most of the performance results above were obtained through simulation or experimental studies. Little analytic work on the performance of RAID architectures has been reported in the literature. Menon, Mattson, and Ng [75] studied RAID 5 by modeling each disk in the array by a $M/M/1$ queue, and by calculating the response time for a write request according to Nelson and Tantawi's fork/join queueing model [80]. By using this simple $M/M/1$ model, they studied the performance of RAID 5 in normal mode (all disks operational), in degraded mode (one disk broken, rebuild not started), and in rebuild mode (rebuild started but not finished). A noticeable feature of their study is that they first integrated and examined the effect of caching on the performance of disk arrays.

Research on the disk array reliability issue and recovery technique has been reported in [104, 34, 77]. Schulz *et al.* [104] developed a model of reliability that includes support hardware, such as power supplies, controller electronics, cables, and fans, so as to provide a better understanding of disk array systems. From another perspective, Copeland and Keller [34] examined two recovery techniques for database systems, in which two copies of data are maintained. *Mirrored declustering*, which corresponds to the RAID 1 structure, spreads each relation across two identical sets of disks. *Interleaved declustering* [115], on the other hand, places the primary copy on one disk and spreads the secondary copy across other disks in the cluster. Read requests are given higher priority over updates, and transaction response times are affected only by read operations. Consequently, they concluded that *interleaved*

declustering performs better than *mirrored declustering* in the case of media failure. In [77], Muntz and Lui proposed a modified RAID 5 organization, where the group size (i.e., stripe width) may be less than the number of disks in an array. They analyzed the recovery time and MTTF for the array system in the face of maintaining a minimum level of I/O service to customers during the rebuild period. While the modified RAID 5 works well in a database environment, it might be less efficient in supporting large I/O's, where data are transferred from multiple disks in parallel.

Recently, Lee and Katz [64] considered the problem of placing parity blocks in a disk array. They found that for a relatively large request size of hundreds kilobytes, a good choice of parity placement may result in up to a 20% to 30% performance improvement. Properties that are generally desirable of good parity placement are also proposed. In another study, Chen and Patterson [25] examined, via simulations, how to choose the striping unit, i.e., the number of logically contiguous data on each disk, so as to maximize the performance. The high overhead for write requests was ignored in their discussions. In [111], the idea of RAID 5 was extended to distributed systems, in which each site has its own disk system. Parity blocks are rotated in the same way as in RAID 5 on each site, so that the whole system can survive a single site failure. The price paid for this fault tolerance is that for each update operation of n blocks at a site, up to n messages are sent to other sites to initiate updates of corresponding parity blocks. Since a disk array usually has one or more hot spare disk(s), Menon *et al.* [74, 73] and Reddy *et al.* [98] studied how to use the spare space efficiently to improve the performance. Other references on disk array studies include [94] and [21].

Disk I/O for Real-Time Systems

As mentioned before, in a real-time system, each computational entity or customer[2] carries a time constraint. Some of these entities' deadlines are very tight while others are not. All of these entities should be scheduled and processed in such a way that they complete by their corresponding deadlines. A common scheduling discipline in

[2]In the following discussions, we will use the term *entity* and *customer* interchangeably.

real-time systems is for customers with tight deadlines to be given a high priority for service.

Scheduling time-constrained customers in real-time systems has been studied from several perspectives. Typically, a real-time system is referred *removal system* if customers who have missed deadlines are allowed to be removed from the system without receiving service. Theoretically, for such a removal system, under certain circumstances the optimal scheduling policy has been shown to be the *earliest deadline policy (ED)*, which always selects the customer with the smallest deadline (most emergent) for service [90, 89, 15]. Here a policy is said to be optimal, if it minimizes the fraction of customers that are lost.

Previous work on non-removal real-time systems, in which all customers have to be served finally, even though some of them have already missed their deadlines, was focused mainly on reducing the customer lateness and tardiness. Su and Sevcik [113] looked at the problem of scheduling customers with deadlines in a queue. They showed that *earliest due date (EDD)* rule minimizes performance parameters of expected lateness and tardiness when every customer has to be served. In [6, 9], Baccelli *et al.* discussed various issues regarding queueing systems which serves impatient customers on the first-come-first-serve (FCFS) bases. In [18] Blanc *et al.* considered the problem of optimal control of customers arrived to a FCFS single server queue. The objective of the study was to minimize the discounted reward associated with the successful departure of customers. Pinedo [93] considered the problem of minimizing the number of late jobs when the processing times are exponentially distributed and the deadlines are randomly distributed. Dertouzos [36] has shown that for any set of tasks with arbitrary service times and deadlines, the EDD policy is optimal if preemptions are allowed. Both of Pinedo and Dertouzos considered only *deterministic* scheduling policies among a set of n jobs, i.e., no new jobs are allowed into the system once processing begins. Other work of interest can be found in [37, 121].

Extensive work has been done in the context of real-time CPU scheduling, multiprocessor scheduling, and real-time transaction processing [1, 20, 31, 53, 67, 108, 122, 123]. However, it is interesting to observe that, to our knowledge, only a few papers have dealt with I/O scheduling in a real-time environment. In [3], Abbott

and Garcia-Molina suggested a real-time disk scheduling algorithm called FD-SCAN (*Feasible Deadline SCAN*), which is a variant of the SCAN algorithm. Specifically, FD-SCAN differs from SCAN in the way that it dynamically adapts the scan direction towards the request with the earliest feasible deadline and services every request on the way, where a deadline is said to be *feasible* if it is estimated that it can be met. The performance of FD-SCAN has been shown to be better than ED-SCAN, which scans towards the request with the earliest deadline, and other conventional algorithms, SSTF, SCAN, and FCFS. In [23], Carey, Jauhari and Livny presented a priority disk scheduling algorithm which is also based on the SCAN algorithm. Although in their work they only looked at a conventional database system, the idea can be easily extended to a real-time environment. In particular, I/O requests are partitioned into several priority levels, and the SCAN algorithm is used within each priority level. After servicing a request, it checks to see if there is any high priority request waiting to be served. If there exist high priority requests, the algorithm switches to the highest level containing a request and continues to scan, during which time all requests with lower priority on the way are skipped. Since the performance criteria for conventional database systems and real-time database systems are quite different, the performance reported there is on the response time rather than the loss ratio.

We have attempted to briefly review some related work reported in the literature. As stated before, this review is by no means intended to be exhaustive. As a consequence, in subsequent chapters, we will continue to discuss some of these and/or other work, as appropriate.

1.5 Research Contributions

The contributions of this dissertation research are in the area of designing and evaluating high performance I/O subsystems for a variety of today's computing systems. In particular, we study mirrored disks for fault tolerant systems, disk arrays for transaction processing/workstation environments and scientific computing/image processing systems, and disk subsystems for real-time systems. The main results are listed below:

Disk Mirroring: In this part of the dissertation, we propose several scheduling policies for a mirrored disk, which differ from each other by the manner in which read requests are scheduled to one of the two disks. Each of these policies is then modeled by a multiple dimensional Markov chain. The performance of these policies is studied by using the results obtained from these analytic models. The results show that:

- The best policy is the minimum response policy DP-MR, which sends a read request to both disks and, whenever the first one finishes, aborts the other one. This requires controller support to abort a request during its service.
- If disk access is assumed to be non-preemptive, then the shortest queue policy DP-SQ-MS and the central queue policy CP-AQ-MS are the best choices. Under DP-SQ-MS, two queues are maintained, one for each disk, and a read request is routed to the shortest queue at its arrival. Under CP-AQ-MS, all requests join a central queue but an auxiliary queue may be built for write requests for the disk which lags behind.
- When the two disks are physically located on different sites, then the distributed DP-RJ policy, which randomly routes read requests to one of the two disks, can be used.
- In addition, we point out that although our discussions are focused on disk I/O subsystems, some of the modeling techniques, such as our method of obtaining the performance bounds, are also of independent interest and are likely to be applicable to a large variety of analysis problems. For example, these modeling and bounding techniques have been used in our subsequent study of RAID 1 disk array and its variants.

Disk Arrays: In this part of the dissertation, we study the behavior of existing disk array architectures and propose a new architecture, which is a variation of RAID 1, called "group-rotate declustering". The performance of these architectures is examined in both "small" I/O as well as "large" I/O environments.

- We consider RAID 1 and its variants, *mirrored declustering*, *chained declustering*, as well as our proposed *group-rotate declustering*. We have observed that, for applications with predominantly small transfers, all three architectures coupled with the random join policy DP-RJ perform basically the same. However, when coupled with the shortest queue policy DP-SQ-MS, group-rotate declustering is the best and mirrored declustering is the worst. For applications in which I/O requests typically transfer a large amount of data, mirrored declustering and group-rotate declustering are equivalent, and they perform much better than chained declustering. While group-rotate declustering has been observed to be equivalent to or outperform mirrored declustering when the disk array is in a normal mode, it is also expected to perform better than mirrored declustering *in* the presence of a disk failure.

- We examine the RAID 5 and Parity Striping architectures in a small I/O environment. Our work differs from previous work by explicitly modeling "write synchronization" which is required in both RAID 5 and Parity Striping but has been ignored in previous performance studies. In particular, we propose two synchronized scheduling policies. A priority queueing model is developed for these architectures coupled with the proposed scheduling policies. As a consequence, we observe that RAID 5 is sensitive to the increase in mean request size, and Parity Striping is sensitive to access skew. We identify workloads for which each architecture provides the best performance.

- We compare RAID 1 and RAID 5 to study the performance gain achievable by RAID 1. Two cases are identified: (1) the two arrays having the same capacity, and (2) the two arrays containing the same number of disks. The results show that RAID 1 provides much higher performance than RAID 5, especially for applications in which each request typically transfers a small amount of data. The only case where RAID 5 is observed to outperform RAID 1 is when most of the requests are writes, each request accesses a large amount of data, and most of these writes perform a *full stripe write*.

Real-Time Disk I/O: We propose two new scheduling algorithms, SSEDO and SSEDV, for real-time systems which set priorities for I/O requests by accounting for both time constraints and arm-move distances.

- These two algorithm are compared with other existing real-time algorithms, P-SCAN, FD-SCAN, and ED, as well as conventional algorithms, SSTF, SCAN, C-SCAN, and FCFS. Most of the performance studies are conducted in an integrated removal real-time transaction system model, in which a transaction that misses its deadline is removed from the system without receiving service. Hence, the performance metric of interest is the system level transaction loss ratio rather than individual I/O request loss ratio. The transaction system model has been validated by an actual real-time transaction system testbed, called RT-CARAT. As a result, we are able to show that SSEDO and SSEDV can improve performance up to 38% over other real-time algorithms, and up to 53% over conventional algorithms.

- We extend our algorithms to real-time disk I/O subsystems consisting of a mirrored disk and disk arrays by combining them with the previously studied disk mirroring and disk array architectures and policies. Results from simulation show that applying real-time algorithms to a mirrored disk or disk arrays may lead up to a 17% to 20% performance improvement over the corresponding non-real-time versions.

- We investigate disk I/O subsystems for non-removal real-time systems by presenting a useful paradigm for transferring algorithms from removal systems to non-removal systems. We show that the performance ordering of different algorithms obtained by using the paradigm in a non-removal system is the same as those algorithms originally studied in a removal real-time system environment. This allows us to quickly locate good algorithms suitable for non-removal real-time systems.

We believe that the proposed architectures and scheduling policies (algorithms), their analytic models and predicted performance developed as a result of this research

can serve as valuable aids to the system and I/O subsystem designers in making design choices.

1.6 Overview of the Dissertation

The rest of the dissertation is organized as follows: in Chapter 2, we first describe some techniques for improving the I/O subsystem performance, such as caching, device buffering to avoid RPS missing, and the concept of log-structured file systems (LFS). While this dissertation focuses on the architectural and service discipline aspects of disk I/O subsystems, these techniques described can be used in combination with those studied in this dissertation to achieve high performance. In addition, we also briefly describe some potential alternatives to magnetic disks, e.g., RAM disks and optical disks. Then, we present a disk model which will be used in the subsequent chapters.

In Chapter 3, we concentrate on the behavior of a mirrored disk. In particular, we present several policies for serving I/O requests and develop mathematical models which model the behavior of these policies. In solving these mathematical models, a new bounding technique is used, which we believe to be applicable to other analytic problems. Last, the performance of these policies is studied and the best policies are recommended.

In Chapter 4, we study disk arrays for transaction processing and scientific applications. The disk array architectures of particular interest are RAID 1, RAID 5, and their variations. Scheduling policies are proposed for different disk array architectures, and their performance is studied through analysis and simulation.

While Chapters 3 and 4 deal with conventional systems, in Chapter 5, we examine disk I/O subsystems for real-time systems. Two new real-time disk I/O scheduling algorithms are proposed. Their performance is studied and compared with other existing algorithms in an integrated real-time transaction system model. Then, modified versions of the two algorithms are studied in a non-removal real-time system, in which a computational entity which has missed its deadline is still served to completion.

Finally, Chapter 6 summarizes our work and indicates directions for future extensions.

CHAPTER 2

ALTERNATE TECHNIQUES AND THE DISK MODEL

In this dissertation research, we will focus on the architectural and service discipline aspects of I/O subsystems. There are, however, many other techniques for improving I/O performance. In the next section, we will briefly describe several such techniques. Then we will discuss some other alternatives to the magnetic disks, such as optical disks and RAM disks. Last, we will present a disk model which will be used in our analysis in the subsequent chapters.

2.1 Techniques for Improving I/O Performance

There are many ways to improve I/O performance. In this section, we only describe several such techniques which either have shown to improve, or have the potential of improving, I/O performance. Discussions of other techniques can be found in [42].

2.1.1 Data Caching

Caching has been used in memory hierarchies for many years and is based on the principle of locality of accesses. If disk data are likely to be used again, then they can be found immediately in the cache.

As main memory becomes larger and larger, it is possible to cache more and more data in main memory. Consequently, for applications such as workstation/engineering environments in which disk access patterns exhibit high locality, most read requests can be satisfied by the cache, and therefore accesses to disk are reduced. The cache can be placed at different locations along the data path from CPU to device [106, 107]. Typically, file systems cache data in the main memory. Since it has been observed in workstation/engineering environments that frequently used files are usually small and

short lived [88, 10, 63, 102], a file cache adopting the *write-back* policy can efficiently reduce the I/O traffic to the disk. This is especially beneficial when the disk is on a remote site connected through a local area network. A problem with the file cache is the maintenance of cache consistency in a multiple processor system, since each processor may cache a shared file in its local cache. Whenever a copy of a shared file in a local cache is updated, other processors should be made immediately aware of this.

On the other hand, disk manufactures usually supply their products with a disk cache installed at the controller. The main advantage of such a disk cache is that it is easily integrated into existing operating systems, and it avoids the above cache consistency problem. However, its efficiency may be less than that of a main memory file cache.

While caching techniques can effectively reduce the disk traffic for applications with high access locality, such as workstation/engineering environments, it is less efficient for some other applications. One such an example is found in database applications where disk data access patterns are observed to be random [41] and it is impractical to cache the entire database in the main memory when the database is large. Another example can be found in the scientific applications, in which an I/O request may transfer a huge amount of data which is larger than the cache. Also in such an environment, data are less likely to be reused in the near future. Hence, caching by itself may not be sufficient to solve the I/O bottleneck problem.

As indicated in [55], another potential problem with a disk or file cache is that it must be made nonvolatile to avoid data loss and inconsistency between cache and magnetic disks when system crashes or power failure occurs. One solution is to provide battery backup for the cache. However, it is difficult to verify that the battery will be fully charged when conventional power fails. It is also difficult to determine how long is enough to power the cache with batteries, since the repair time of a system crash resulting from software bugs or hardware failures may vary from several hours to a couple of days.

Other comments regarding the use of a disk or file cache can be found in [120, 12], and the implementations of file caches are reported in [50, 71, 78, 103, 109].

2.1.2 Log-Structured File System

Since cache sizes are becoming larger and larger, most of the reads may be expected to be satisfied by the cache for some applications. In this case, disk accesses consist of mostly write operations. Moreover, by using the write-back policy, many small writes can be accumulated in the cache. Based on these observations, a radical idea is to reorganize file systems in the form of a circular append-only log [87, 38, 99, 100], called a *Log-Structured File System (LFS)*. Unlike a conventional file system in which a file update is performed in place on the disk, a LFS only appends a modified or a new file, together with the file header, to a log. The purpose is to provide sequential writes so that most seeks during update operations can be eliminated. The performance improvement for the file access time is expected to be an order of magnitude or more.

There are three difficult issues associated with LFS that have to be resolved,

- how to handle the occasional retrievals that cannot be satisfied by the cache and are required from the log;
- since disk space is limited, how to manage log wrap-around;
- how to achieve efficient disk space utilization

Some possible solutions for these problems are discussed in [87, 38].

Recently, Rosenblum and Ousterhout [99, 100] reported an implementation of LFS for typical UNIX workloads. The LFS treats a disk as a segmented append-only log, with a segment size of 1M bytes. By using the *write back* strategy, bursts of small writes to the file system are aggregated in the cache and converted into a large write of size 1M bytes. A segment write is triggered by three conditions: cache full, user initiation, or when an age threshold of 30 sec is hit. Notice that the last two conditions can cause a partial segment to be written, because the file cache may not contain sufficient data to fill an entire segment. The free disk space can be exhausted quickly. In order to maintain enough free disk space, a cleaning algorithm, which merges segments by copying data from one segment to another, executes whenever the number of free disk segments is less than some thresholds. The performance of

LFS was compared with SunOS file system by using some synthetic workloads, and the results showed that the LFS outperforms the conventional SunOS file system.

2.1.3 Shared Data Path and Concurrent Arm Positioning

It is a common practice to let multiple devices share a component on the data transfer path, such as channel and controller, for economic reasons. A typical example can be found in IBM's I/O subsystems, where multiple devices are connected to a string, multiple strings share a control unit, and multiple control units share one or more channel(s). Another example can be found in today's workstations, where up to 7 SCSI disks can be chained together to share a single controller. In such a system, the controller issues a command to a device. After receiving the command, the device starts to position its arm. In the meantime, the device gives up the controller so that the controller can serve other devices. This allows concurrent arm positioning of multiple devices which share the same controller. When the target data on a device rotates near the read/write head, the device sends a signal to the controller, requesting a reconnection for a data transfer. This is called *rotational position sensing (RPS).* If the controller is busy serving another device at that moment, a RPS miss occurs. In this case, the device has to take another full rotation and then tries to reconnect again. Obviously, the cost for RPS misses is high, especially when multiple devices sharing the same controller are busy serving I/O requests simultaneously. One way to solve the RPS miss problem is to provide a small buffer for each device. Whenever a RPS miss occurs, the data is buffered so that the extra rotation can be avoided. Another way is to provide multiple controllers for multiple strings of disks. In this case, if a disk in a string find a controller busy, it can be assigned to another available controller.

In our discussions above, we have briefly described several ways of improving I/O performance. Although these are interesting approaches, we will not pursue them further in this dissertation because of time and space limitation. However, these techniques can be combined with the mechanisms to be discussed in subsequent chapters to achieve a high performance I/O subsystem. For example, as we will see in Chapter 4, one of the main drawbacks of disk array level 5 (RAID 5) is the high cost

for small write operations. However, this problem can be improved by combining the Log-Structured File System with RAID 5 so that many small writes are converted into a large sequential write.

2.2 Alternatives to Magnetic Disks

Although our main focus is on magnetic disks in this dissertation study, there are potential alternatives for future computer systems.

2.2.1 Optical Disks

The main advantage of an optical disk is its high capacity, which is in the order of gigabytes for 5.25-inch disks [22, 12]. However, for erasable optical disks, magneto-optic technology, which has the longest durability – about 10 million write cycles, requires an erasure before a write. This adds one more full rotation for each write. Moreover, if a verification after a write is needed, yet a third rotation is necessary. These additional rotations result in a high cost for update operations. In addition, current optical devices have a slower seek time and rotation speed compared to magnetic devices. Therefore, optical devices may not completely replace all magnetic devices in the foreseeable future. Importantly, most of the techniques and conclusions presented in our study should also apply to optical disks.

2.2.2 RAM Disks

RAM disks, or *solid state disks (SSD)*, are constructed from memory chips, which are backed by batteries to prevent data loss in case of power failures. As VLSI technology progresses, the capacity of memory chips is growing, while the price has been decreasing rapidly. One can expect that some day in the future, the price of RAM disks will approach that of magnetic disks, and therefore magnetic disks will be replaced by the higher performance RAM disks. How soon will this happen? Gibson estimates 30 years [42]. Other optimistic estimates are a little earlier, but still at least ten years away. Therefore, reducing the speed gap and designing high performance I/O subsystems for a variety of computing systems based on the current

and near future disk technologies are an extremely important task for researchers and designers.

2.3 A Disk Model

In the preceding sections, we have described several techniques for improving I/O performance. From now on, we will focus on the topics of this dissertation study, i.e., examining design and performance issues of I/O subsystems from the architectural and service discipline points of view, both for conventional and real-time systems. In doing so, we first present a disk model, which will be used in most of the subsequent chapters.

Since we are focusing on the disk I/O subsystem, we count the I/O response time as the time elapsed between the arrival of a request to the subsystem and the departure of this request from the subsystem. Typically, the I/O response time to a disk subsystem consists of four components: queueing delay at the controller, seek time, rotational latency, and data transfer time. In the case of a shared data path, there is also a shared resource contention delay. As indicated in Section 2.1.3, there are several ways to help reduce or eliminate the resource contention delay. Hence in the following discussion, we will assume that each device has its own independent data path and we will ignore the resource contention delay. Thus, we will only focus on the queueing delay, which is a function of the I/O load, and the disk access time, which consists of seek, rotation, and transfer times.

Define,

C: the number of cylinders for each disk;

N_b: the number of blocks in a track;

τ: the transfer time for a single block;

S_{max}: the maximum seek time;

R_{max}: the full rotation time;

D: a random variable ($r.v.$) denoting the disk seek distance;

V: a *r.v.* denoting whether an access is sequential ($V = 0$) or not ($V = 1$);

p_s: the probability that the seek distance is equal to zero, $p_s = P\{V = 0\}$;

S: a *r.v.* for disk seek time;

R: a *r.v.* for rotational latency;

X: a *r.v.* for the sum of seek and rotation time, $X = S + R$;

B: a *r.v.* for the number of blocks to be transferred in a request;

T: the data transfer time, $T = \tau B$;

Y_r, Y_w, Y: *r.v.*s for read, write, and overall disk service times.

Z_r, Z_w, Z: *r.v.*'s for read, write, and overall I/O response times.

First, consider the disk seek pattern. Since it is observed that in reality, most of the time the disk arm doesn't move [68, 60], we introduce a *sequential access probability*, p_s, which is defined as the probability that the seek distance is equal to zero. We identify two kinds of access patterns, the sequential access pattern and non-sequential access pattern. Under the *sequential access* pattern, the disk arm does not move, whereas under the *non-sequential access* pattern, the arm has to move to serve a request. In the case of a *non-sequential* access, the arm is assumed to move to any other cylinder with equal probability.

According to the definition of V, we have the following conditional distributions for the seek distance,

$$P\{D = i|V = 0\} = \begin{cases} 1 & i = 0, \\ 0 & i = 1, 2, \cdots, C-1; \end{cases}$$

$$P\{D = i|V = 1\} = \begin{cases} 0 & i = 0, \\ \frac{2(C-i)}{C(C-1)} & i = 1, 2, \cdots, C-1, \end{cases}$$

where the term $2(C-i)/C(C-1)$ is derived based on the assumption that the arm and the target cylinder are randomly located on the disk and that they are distinct.

Removal of the conditioning yields

$$P\{D=i\} = \begin{cases} p_s, & i=0, \\ (1-p_s)\frac{2(C-i)}{C(C-1)}, & i=1,2,\cdots,C-1. \end{cases} \tag{2.1}$$

We also use the following expression for the seek time, S, as a function of the seek distance D [105, 16, 17, 85],

$$S = \begin{cases} 0, & D=0, \\ a+b\sqrt{D}, & D>0, \end{cases}$$

where a is the arm acceleration time and b is the mechanical seek factor.

For ease of analysis, we approximate the seek distance D as a continuous $r.v.$ with probability density function (pdf)

$$f_D(x) = \begin{cases} p_s u_0(x), & x=0, \\ (1-p_s)\frac{2(C-x)}{C^2}, & 0<x\leq C, \end{cases}$$

where $u_0(x)$ is the unit impulse function defined by (see [61] pp. 342)

$$u_0(x) = \begin{cases} \infty, & x=0, \\ 0, & x\neq 0, \end{cases}$$

$$\int_{-\infty}^{\infty} u_0(x)\,dx = 1.$$

The cumulative distribution function (CDF) of D, $F_D(x) = P\{D<x\}$, is

$$F_D(x) = \begin{cases} p_s, & x=0, \\ (1-p_s)\left(\frac{2x}{C}-\frac{x^2}{C^2}\right), & 0<x\leq C, \\ 1, & C<x. \end{cases}$$

Therefore, the seek time S has the following CDF,

$$F_S(x) = \begin{cases} F_D(0), & 0\leq x\leq a, \\ F_D\left((\frac{x-a}{b})^2\right), & a<x\leq S_{max}, \\ 1, & S_{max}<x, \end{cases} \tag{2.2}$$

where the maximum seek time $S_{max} = a+b\sqrt{C-1}$.

The mean seek time is

$$E[S] = \sum_{i=1}^{C-1} P\{D=i\}(a+b\sqrt{i})$$

$$= (1-p_s)\left[a + b\frac{2}{C-1}\left(\sum_{i=1}^{C-1} i^{\frac{1}{2}} - \frac{1}{C}\sum_{i=1}^{C-1} i^{\frac{3}{2}}\right)\right].$$

By approximating the summations by integrals, e.g., $\sum_{i=1}^{C-1} i^{\frac{1}{2}} = \int_1^{C-1} x^{\frac{1}{2}}\, dx$, we obtain

$$E[S] \approx (1-p_s)\left[a + b\sqrt{C-1}\frac{8}{15}\right]. \tag{2.3}$$

The rotation time is assumed to be uniformly distributed in $[0, R_{max}]$ with *pdf*

$$f_R(x) = \frac{1}{R_{max}}, \qquad 0 \le x \le R_{max}. \tag{2.4}$$

and mean

$$E[R] = \frac{R_{max}}{2}. \tag{2.5}$$

Let $X = S + R$ be the sum of seek and rotation time, then the *pdf* of X is

$$f_X(x) = \int_0^{S_{max}} f_R(x-s)\, dF_S(s).$$

Specifically, since the seek time S is not a purely continuous *r.v.*, $f_X(x)$ has one of the following forms, depending on the disk parameters,

case 1: $R_{max} \le b\sqrt{C-1}$,

$$f_X(x) = \begin{cases} f_R(x)p_s + \int_{a+}^{x} f_R(x-s)\, dF_S(x), & 0 \le x \le R_{max}, \\ \int_{a+}^{x} f_R(x-s)\, dF_S(s), & R_{max} < x \le R_{max} + a, \\ \int_{x-R_{max}}^{x} f_R(x-s)\, dF_S(s), & R_{max} + a < x \le S_{max}, \\ \int_{x-R_{max}}^{S_{max}} f_R(x-s)\, dF_S(s), & S_{max} < x \le S_{max} + R_{max}; \end{cases} \tag{2.6}$$

case 2: $R_{max} > b\sqrt{C-1}$,

$$f_X(x) = \begin{cases} f_R(x)p_s + \int_{a+}^{x} f_R(x-s)\, dF_S(s), & 0 \le x \le R_{max}, \\ \int_{a+}^{x} f_R(x-s)\, dF_S(s), & R_{max} < x \le R_{max} + a, \\ \int_{x-R_{max}}^{S_{max}} f_R(x-s)\, dF_S(s), & R_{max} + a < x \le S_{max} + R_{max}. \end{cases}$$

For our particular disk parameter settings (see page 30), $f_X(x)$ is obtained from case 1, where each of the integrals is a $4th$-order polynomial.

Last, the disk service time for a typical request is

$$Y = X + T \tag{2.7}$$

with mean

$$\begin{aligned} E[Y] &= E[S] + E[R] + E[T] \\ &= E[X] + E[T] \end{aligned} \tag{2.8}$$

where T is the transfer time, $T = \tau B$.

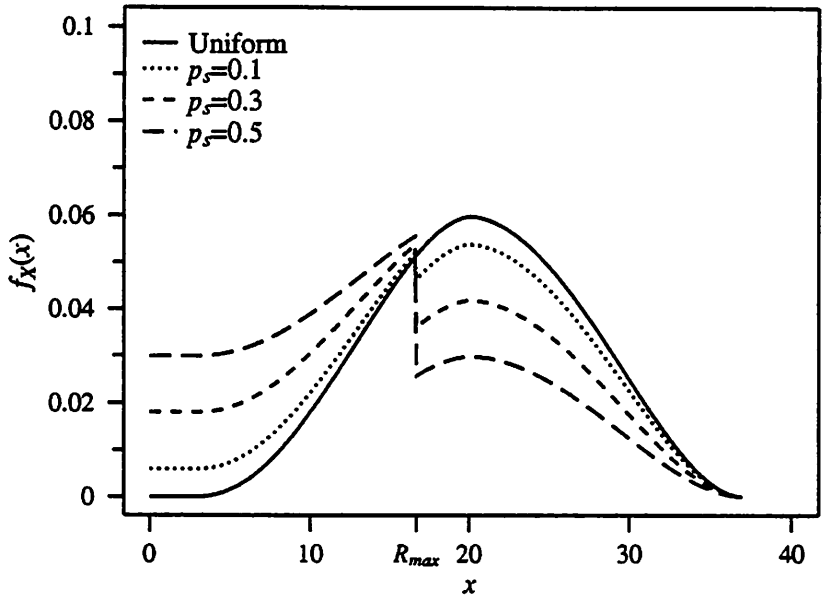


Figure 2.1 Probability Density Function for the Sum of Disk Seek and Rotational Latency.

It is interesting to observe that, according to some real disk traces, in [60], Kim and Tantawi suggested that the distribution of X, the sum of seek and rotation time, can be approximated by a normal distribution. According to our derivation, the curve for the *pdf* of X matches the bell shape of a normal distribution only when $p_s = 1/C$ (Figure 2.1), i.e., the arm moves to any cylinder with equal probability, including the current cylinder. As p_s increases, there is a discontinuity at R_{max}, which becomes more pronounced as p_s increases. This is because when X is greater than R_{max}, the possibility for a zero seek time disappears (notice the difference between the first and second formula in Equation (2.6)). In fact, similar behavior can also be observed from the histograms presented in [60].

In the above discussion, we derived the disk service time distribution for a single disk, which accounts for the access locality. For a mirrored disk, since there are two data copies and a read can be satisfied by either copy, the disk service time distribution is slightly different. If a read arrives the I/O subsystem and find both disks idle, it can start at the disk with minimum arm movement. In this case, the seek distance is a *r.v.* defined by $D_{min} = \min\{D^{(1)}, D^{(2)}\}$, where $D^{(1)} =_d D^{(2)} =_d D$

and independent of each other. (Here "$=_d$" denotes "equal in distribution".) The distribution of D_{min} is

$$P\{D_{min} = i\} = P\{D^{(1)} = i, D^{(2)} > i\} + P\{D^{(1)} > i, D^{(2)} = i\} + P\{D^{(1)} = i, D^{(2)} = i\}.$$

Since $D^{(1)}$ and $D^{(2)}$ are independent and identically distributed ($i.i.d.$), we have

$$P\{D_{min} = i\} = \begin{cases} p_s(2 - p_s), & i = 0, \\ 4(1 - p_s)^2 \frac{(C-i)^3}{C^2(C-1)^2}, & i = 1, 2, \cdots, C-1. \end{cases}$$

In a similar way as above, define $S_{min} = \min\{S^{(1)}, S^{(2)}\}$ which has mean

$$\begin{aligned} E[S_{min}] &= \sum_{i=1}^{C-1} P\{D_{min} = i\}(a + b\sqrt{i}) \\ &\approx (1 - p_s)^2 \left[a + b\sqrt{C-1} \times 0.408\right] \end{aligned} \tag{2.9}$$

and the mean service time, Y_{min}, for these read requests is immediately obtained from equation

$$E[Y_{min}] = E[S_{min}] + E[R] + E[T]. \tag{2.10}$$

Last, the disk parameters are summarized as follows, which are typical for current disk drives: number of cylinders $C = 1200$; transfer rate= 3MB/sec; maximum rotation time $R_{max} = 16.7$ ms; block size = 4096 bytes; acceleration time $a = 3$ ms; seek factor $b = 0.5$ [85]. Thus, the time required to transfer a single block is $\tau = 1.3$ ms.

CHAPTER 3

A MIRRORED DISK FOR FAULT TOLERANT SYSTEMS

We consider a fault tolerant mirrored disk subsystem, in which each data item is stored on a pair of disks. We study the effects of different scheduling policies on the performance of a mirrored disk subsystem. These policies differ from each other according to the number of queues required and the level of concurrency provided between different I/O requests. Based on the number of queues used, we divide all policies into two categories, *single queue policies* where only one queue is maintained in front of the mirrored disk and *multiple queue policies* where two or more queues are maintained. In a previous study [117], we observed that the worst policies are those that do not allow reads to execute on both disks. This can occur in systems in which a *secondary* disk is maintained in a standby mode until the *primary* disk fails. We also observed that single queue policies are worse than multiple queue policies. Hence, we will focus on the performance analysis of various multiple queue policies in this study. Discussions of these policies in a fork/join queueing system context can be found in [116].

This chapter is organized as follows: we first describe different multiple queue policies, and then provide analytic models for these policies. In solving these models, we develop bounding techniques which bound the performance metrics interest to us. Last, we compare the performance of different policies by using these analytic models and simulations.

3.1 Description of Scheduling Policies

As indicated in [117], one common feature of the single queue policies is that write requests may proceed only when both disks are free. By introducing a second queue, a multiple queue policy allows a write request to spawn two tasks that may access each

disk asynchonously. These multiple queue policies can also be further divided into two subclasses, *distributed policies (DP)* and *centralized policies (CP)*. In all cases, I/O requests are served in a first-come-first-serve manner within each queue. Other service disciplines, such as the elevator algorithm, will be briefly discussed at the end of this chapter.

Distributed Policies (DP):

In the class of distributed policies, two queues are maintained by the system, one for each disk (Figure 3.1). At the time of arrival, a write request generates two

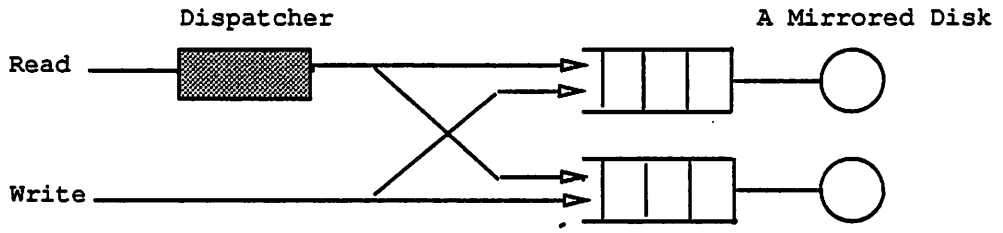


Figure 3.1 A Queueing Model for Distributed Policies.

tasks that enter each of the two queues. The various policies differ from each other according to the way that they decide upon the disk to which the read request will be sent.

- **The Random Join (DP-RJ) Policy.** Under this policy, a read request is randomly assigned to the queue associated with disk i with probability α_i, $i = 1, 2$ ($\alpha_1 + \alpha_2 = 1$), at the time of its arrival.

- **The Shortest Queue (DP-SQ) Policy.** Under the DP-SQ policy, a read request selects the disk with the shortest queue at the time of its arrival. Ties are broken in an arbitrary manner.

- **The Shortest Queue with Minimum Seek (DP-SQ-MS) Policy.** The DP-SQ-MS policy behaves like the DP-SQ policy, except when a read request arrives to an empty system, i.e., both disks are idle. In this case, the read request is assigned to the disk with the minimum seek distance. The performance

improvement gained by adopting this strategy is expected to be noticeable when the system is lightly loaded.

- **The Minimum Waiting (DP-MW) policy.** Under the DP-MW policy, each read request generates two tasks, one for each queue. Whenever any one of these tasks begins service, its counterpart is immediately removed from the other queue.

- **The Minimum Waiting with Minimum Seek (DP-MW-MS) Policy.** The DP-MW-MS policy is defined in a similar way as the DP-SQ-MS policy to achieve better performance under light loads. Namely, if a read request arrives and finds both disks idle, it chooses the disk with the shortest seek time to execute.

- **The Minimum Response (DP-MR) Policy.** The DP-MR policy is similar to the DP-MW policy which allows read requests to enter both queues upon arrival. However, the DP-MR policy aborts one of the tasks associated with a read request once its peer *completes* its service, whereas the DP-MW policy aborts a task once its peer *begins* its service. Therefore, this policy requires the disk controller and device driver to allow the abortion of a request while it is in service.

Remark: Among all of these policies, the DP-RJ policy is suitable for systems in which the two disks are physically located at two distant sites, since it requires no information interchange between the two disks. The other policies, however, are more suitable for systems in which the two disks are located close to each other because those policies have to know the status of each queue in order to schedule read requests.

Centralized Policies (CP):

Policies in this class maintain a common queue for both disks. Both read and write requests enter the common queue at the time of their arrival (Figure 3.2).

- **The Auxiliary Queue (CP-AQ) Policy.** Under the CP-AQ policy, a request at the front of the common queue begins service as soon as a disk becomes

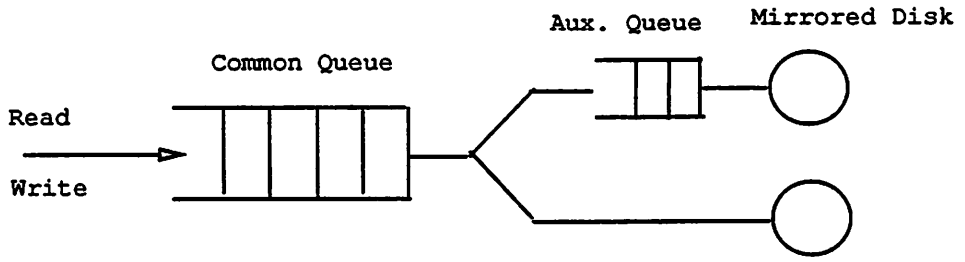


Figure 3.2 A Queueing Model for Centralized Policies.

available. If the request at the head of the common queue is a write request, it also places the second task in an auxiliary queue associated with the other disk. Thus the system requires an auxiliary queue associated with the disk which lags behind. This auxiliary queue contains write requests only (not including the request in service).

- **The Auxiliary Queue with Minimum Seek (CP-AQ-MS) Policy.** The CP-AQ-MS policy is similarly defined as the CP-AQ policy except that a read request is allowed to begin service at the disk requiring the least arm movement to satisfy the request whenever both disks are idle.

3.2 Analytic Models

In this section, we present analytic models for these policies described in the last section. While the primary contribution of this study is the evaluation and comparison of a variety of distributed and centralized policies, the method of analysis is also of interest. Specifically, we develop tight bounds on the response time distributions of read and write requests under these policies. This is accomplished by judiciously truncating one of the state variables in the Markov chain models for these policies and applying the matrix-geometric methodology developed by Neuts [81] to obtain the bounds. The presence of these bounds provides an analyst with the capability of approximating the statistics of the response times of read and write requests to the degree of accuracy required by changing the truncation parameter. For all of these policies, an increase in accuracy is obtained with an increase in computational cost.

As we will see later, when the truncation parameter equals 32, a very tight bound is achieved.

In the analysis, we make the following assumptions:

- I/O requests arrive to the disk subsystem according to a Poisson process with rate λ. A request is a read with probability p_r, and a write with probability $p_w = 1 - p_r$. Typically, a disk subsystem receives streams of I/O requests from multiple users. Although individual streams may not be well described by a Poisson process, the process describing all of the arrivals may be well described by a Poisson process as it is a superposition of a large number of streams, provided the user population is large [57].

- The geometries of the two disks are identical.

- The service times at each disk form two mutually independent sequences of identically distributed *r.v.*'s. This independence assumption is reasonable for a mirrored disk since copies of the same data item may be stored in different cylinders, tracks, or blocks on the two disks[1].

- For a mirrored disk, each disk has its own data path so that the channel is not a bottleneck, and therefore the channel delay is ignored. This assumption is justified by the fact that a mirrored disk system usually duplicates all of its components to provide fault tolerance.

- In some systems, such as a database system, when a user issues a request to access a large amount of data, the operating system breaks the request into several small requests. Since these small requests are typically accessing the same file, we approximate them as sequential I/O's.

- Finally, for analytical tractability, we further assume that the service time is exponentially distributed with mean $1/\mu$, where the $1/\mu$ is calculated from real disk service time, i.e., $seek + rotation + transfer$ time, as shown in Section 2.3.

[1]Some mirrored disk implementations might allocate data items exactly at the same places on the two disks. This strategy may simplify data allocation algorithms but may result in the loss of some performance gains introduced by applying the *minimum seek rule* to read requests.

In fact, disk service times are not exponential in a real world. Thus, the effects of this assumption are checked by simulations which use the real disk access times.

Under these assumptions, all of the previously discussed policies can be modeled by multi-dimensional Markov chains which are solved through the use of Neut's matrix-geometric method [81]. In the next subsection, we discuss the bounding techniques developed in this study to show how these Markov chains can be manipulated in order to obtain computable bounds on the performance metrics of interest, using the DP-RJ and CP-AQ policies as examples. In the following subsection, we describe the Markov chains used to model the remaining policies.

3.2.1 Bounding Techniques – Cases for the DP-RJ and CP-AQ Policies

We focus on two policies, DP-RJ and CP-AQ, in order to illustrate the bounding techniques developed in this study.

The DP-RJ Policy

Under this policy read requests select queue i with probability α_i, $i = 1, 2$, with $\alpha_1 + \alpha_2 = 1$.

Let $W_i^{(r)}$ and $W_i^{(f)}$ denote the response times of the i-th read and write requests respectively. Let $\mathbf{T}_i = (T_{i,1}, T_{i,2})$ where $T_{i,j}$ denotes the response time of the task generated by the i-th write request that enters queue j, $j = 1, 2$. Here $T_{i,j}$ can be expressed as

$$T_{i,j} = U_{i,j} + X_{i,j} \tag{3.1}$$

where $U_{i,j}$ is the unfinished work in the queue at the time that the i-th write request arrives and $X_{i,j}$ is the service time for the task that it generates, $j = 1, 2$. Last, let $\hat{\mathbf{T}}_i = (\hat{T}_{i,1}, \hat{T}_{i,2})$ where $\hat{T}_{i,1} = \min\{T_{i,1}, T_{i,2}\}$ and $\hat{T}_{i,2} = \max\{T_{i,1}, T_{i,2}\}$. The response time of the i-th write request can be expressed as $W_i^{(f)} = \hat{T}_{i,2}$.

We are interested in the limiting random variables for the above defined random variables when they exist. We shall drop the subscript i when referring to these limiting random variables, i.e., $W^{(r)} = \lim_{i\to\infty} W_i^{(r)}$. We are also interested in the random variables $N^{(r)}$ and $N^{(f)}$ that respectively denote the stationary number of read requests and write requests in the system.

We first observe that each queue and disk can be separately modeled as an $M/M/1$ system. As a consequence, the system exhibits stationary behavior so long as $(\alpha_i p_r + p_w)\lambda < \mu$, $i = 1, 2$. Since the response time of a read request is affected only by the queue that it enters, the distribution of $W^{(r)}$ is given by a weighted sum of the distributions of two independent $M/M/1$ systems

$$P[W^{(r)} > w] = \alpha_1 e^{-(\mu - \alpha_1 p_r \lambda - p_w \lambda)w} + \alpha_2 e^{-(\mu - \alpha_2 p_r \lambda - p_w \lambda)w}$$

with mean

$$E[W^{(r)}] = \alpha_1/(\mu - \alpha_1 p_r \lambda - p_w \lambda) + \alpha_2/(\mu - \alpha_2 p_r \lambda - p_w \lambda).$$

The expected number of read requests in the system, $E[N^{(r)}]$, can be obtained through an application of Little's rule [65]. Consequently, we focus only on the behavior of write requests.

As a further consequence of the fact that each queue behaves as an $M/M/1$ system, we can write the following expressions for the *marginal* distributions of the response times of the two tasks associated with a write request

$$P[T_j > t] = e^{-(\mu - \alpha_j p_r \lambda - p_w \lambda)t}, \quad j = 1, 2$$

with means

$$E[T_j] = 1/(\mu - \alpha_j p_r \lambda - p_w \lambda), \quad j = 1, 2.$$

Let us now conduct the following experiment: select a random write request; select one of the two tasks associated with this request with equal probability; denote the response time of this task by T. Then T has the following distribution,

$$P[T > t] = (P[T_1 > t] + P[T_2 > t])/2.$$

This randomly chosen task is equally likely to be the first or the last of the tasks associated with the request to complete. Consequently, we also have the following identity,

$$P[T > t] = (F_{min}(t) + F_{max}(t))/2 \tag{3.2}$$

where $F_{min}(t) = P[\hat{T}_1 > t]$, and $F_{max}(t) = P[\hat{T}_2 > t]$. If we are able to obtain the marginal distribution for either $\hat{T}_1$ or $\hat{T}_2$, the above identity allows us to obtain the marginal distribution for the other random variable.

Equation (3.2) is also useful for obtaining bounds. For example, let us assume that we have upper bounds $F_{min}^{(ub)}(t)$ and $F_{max}^{(ub)}(t)$ to $F_{min}(t)$ and $F_{max}(t)$, i.e.,

$$\begin{aligned} F_{min}^{(ub)}(t) &\geq F_{min}(t), \quad t \geq 0, \\ F_{max}^{(ub)}(t) &\geq F_{max}(t), \quad t \geq 0. \end{aligned}$$

Then Equation (3.2) can be used to obtain the following lower bounds for $F_{max}(t)$ and $F_{min}(t)$

$$\begin{aligned} F_{max}(t) &\geq 2P[T > t] - F_{min}^{(ub)}(t), \\ F_{min}(t) &\geq 2P[T > t] - F_{max}^{(ub)}(t), \quad t \geq 0. \end{aligned} \tag{3.3}$$

In a similar manner, if we have expressions $F_{min}^{(lb)}(t)$ and $F_{max}^{(lb)}(t)$ that bound $F_{min}(t)$ and $F_{max}(t)$ from below, then Equation (3.2) allows us to obtain the following upper bounds on the last two distributions

$$\begin{aligned} F_{min}(t) &\leq 2P[T > t] - F_{max}^{(lb)}(t), \\ F_{max}(t) &\leq 2P[T > t] - F_{min}^{(lb)}(t), \quad t \geq 0. \end{aligned} \tag{3.4}$$

We shall make use of these relationships in order to bound the statistics of the response time of a write request.

The DP-RJ policy can be modeled as a Markov chain with state $\mathbf{N}(t) = (N_1(t), N_2(t))$ where $N_1(t)$ and $N_2(t)$ are the number of tasks in the queues associated with disks 1 and 2 respectively at time t. Let $q(i,j) = \lim_{t\to\infty} P[N_1(t) = i,\ N_2(t) = j]$. The stationary probabilities satisfy the following equations,

$$\begin{aligned} \lambda q(0,0) &= \mu q(1,0) + \mu q(0,1), \\ (\lambda+\mu))q(i,0) &= p_r\lambda\alpha_1 q(i-1,0) + \mu q(i+1,0) + \mu q(i,1), \qquad i = 1,\cdots, \\ (\lambda+\mu)q(0,j) &= p_r\lambda\alpha_2 q(0,j-1) + \mu q(0,j+1) + \mu q(1,j), \qquad j = 1,\cdots, \\ (\lambda+2\mu)q(i,j) &= p_w\lambda q(i-1,j-1) + p_r\lambda\alpha_1 q(i-1,j) + \\ &\quad p_r\lambda\alpha_2 q(i,j-1) + \mu q(i+1,j) + \mu q(i,j+1), \quad i = 1,\cdots;\ j = 1,\cdots. \end{aligned}$$

Unfortunately, this model is not amenable to a simple analysis. We focus instead on a modified system in which the second queue (associated with disk 2) can hold no more than B tasks. Whenever a write request arrives at the system at time t and finds $N_2(t) = B$, it generates a *single* task that enters the queue associated with disk

1. Similarly, a read request that arrives to a full queue at the second disk passes through without delay.

This modified system can be modeled as a Markov chain with the same state definition. In order to distinguish the modified system from the true system, we shall use the superscript (lb), i.e., $\mathbf{N}^{(lb)}(t)$ instead of $\mathbf{N}(t)$. We define $\mathbf{T}_i^{(lb)}$ according to Equation (3.1) even though this does not produce the correct response time at the second queue[2]. We shall describe an ordering relationship between $\mathbf{T}$ and $\mathbf{T}^{(lb)}$. We first introduce the following definition [112].

Definition 1 *Let* $\mathbf{X}, \mathbf{Y} \in \Re^n$ *be random variables. We say that* $\mathbf{X}$ *is stochastically larger than* $\mathbf{Y}$ $(\mathbf{X} \geq_{st} \mathbf{Y})$ *if for all increasing functions* f

$$E[f(\mathbf{X})] \geq E[f(\mathbf{Y})].$$

In the case that $n = 1$, this is equivalent to

$$P[X > a] \geq P[Y > a], \qquad \forall\, a.$$

We will make use of the following property of the above stochastic ordering.

Property 1 *Let* $\mathbf{X}, \mathbf{Y} \in \Re^n$, *and* $\mathbf{X} \leq_{st} \mathbf{Y}$. *Then*

$$g(\mathbf{X}) \leq_{st} g(\mathbf{Y}), \qquad \forall \ \textit{increasing} \ \ g.$$

In particular, $\max(\mathbf{X}) \leq_{st} \max(\mathbf{Y})$.

We now state and prove the following theorem.

Theorem 3.1 *The following relationships hold between the real system and the modified system.*

1. $\mathbf{N} \geq_{st} \mathbf{N}^{(lb)}$,
2. $\mathbf{T} \geq_{st} \mathbf{T}^{(lb)}$.

---

[2]This is because we have defined the response time at queue 2 to be 0 when the queue length is B at the time a request arrives. Equation (3.1) produces a non-zero response time for that event.

Proof. In order to prove 1), it is useful to study the queue lengths of the system prior to the arrival of a request of either class. Let $\mathbf{M}_i^{(lb)} = (M_{i,1}^{(lb)},\ M_{i,2}^{(lb)})$ and $\mathbf{M}_i = (M_{i,1},\ M_{i,2})$ where $M_{i,j}$ and $M_{i,j}^{(lb)}$ are the numbers in queue j $(j = 1, 2)$ prior to the arrival of request i in the real system and the modified system respectively. These r.v.'s satisfy the following recurrences,

$$\begin{aligned} M_{i+1,j} &= (M_{i,j} + I_{i,j} - D_{i,j})^+, \quad j = 1, 2, \\ M_{i+1,1}^{(lb)} &= M_{i+1,1}, \\ M_{i+1,2}^{(lb)} &= (M_{i,2}^{(lb)} + A_i I_{i,2} - D_{i,2})^+ \end{aligned}$$

where

$$I_{i,j} = \begin{cases} 0 & i\text{-th request does not generate a task for } j\text{-th queue} \\ 1 & \text{otherwise} \end{cases}$$

$$A_i = \begin{cases} 0 & \text{queue 2 is full at the time of arrival of request } i \\ 1 & \text{otherwise} \end{cases}$$

and $D_{i,j}$ is the number of departures from queue j between the arrivals of the i-th and $(i+1)$-th request.

If the initial state vectors of the two systems satisfy $\mathbf{M}_0 \geq \mathbf{M}_0^{(lb)}$, then an induction argument can be used to show $\mathbf{M}_i \geq \mathbf{M}_i^{(lb)}$ for $i = 0, 1, 2, \cdots$[3]. Consequently, we conclude that $\mathbf{M}_i \geq_{st} \mathbf{M}_i^{(lb)}$ for $i = 0, 1, 2, \cdots$. Whenever the real system is ergodic, i.e., the r.v.'s $\mathbf{M}_i$ and $\mathbf{M}_i^{(lb)}$ converge to the limiting r.v.'s $\mathbf{M}$ and $\mathbf{M}^{(lb)}$, then $\mathbf{M} \geq_{st} \mathbf{M}^{(lb)}$. Finally, since arrivals from a Poisson process see time averages[47, sec. 11-2], $\mathbf{M}$ and $\mathbf{M}^{(lb)}$ have the same joint distribution as $\mathbf{N}$ and $\mathbf{N}^{(lb)}$ and we conclude that $\mathbf{N} \geq_{st} \mathbf{N}^{(lb)}$.

The second part of the theorem is shown in a similar manner by focusing on the Lindley equations that must be satisfied by the unfinished work in the system at the time of request arrivals. ∎

Remark. It is possible to show that $\mathbf{N}(t) \geq_{st} \mathbf{N}^{(lb)}(t)$ for $t \geq 0$, where $\mathbf{N}(t)$ and $\mathbf{N}^{(lb)}(t)$ are the queue lengths of the two systems at time $t \geq 0$, and that $\mathbf{T}_i \geq_{st} \mathbf{T}_i^{(lb)}$

[3]Here $\mathbf{V}, \mathbf{V}' \in \Re^n$ satisfy the relation $\mathbf{V} \geq \mathbf{V}'$ iff $v_i \geq v_i'$, $1 \leq i \leq n$.

for $i = 1, 2, \cdots$ for any arrival process and i.i.d sequences of service times at each queue.

The modified lower bound system is solved by using the matrix-geometric method (see Appendix A), which parallels the analysis given by Rao and Posner [95]. They derive the following expression for the joint distribution of $\mathbf{T}^{(lb)}$

$$\begin{aligned} T^{(lb)}(w_1, w_2) &\equiv P[T_1^{(lb)} \leq w_1, T_2^{(lb)} \leq w_2], \\ &= \pi[I - \exp(-\mu(I - R)w_1)]B(w_2) \end{aligned}$$

where I is the identity matrix, π is a $B + 1$ element vector containing the stationary queue length distribution for the $M/M/1/B$ queue with arrival rate $(p_w + \alpha_2 p_r)\lambda$ and service rate μ, i.e., the i-th element of π is $(1 - u)u^i/(1 - u^{(B+1)})$ where $u = (\alpha_2 p_r + p_w)\lambda/\mu$, $B(w_2)$ is a $(B + 1)$ column vector with i-th component $\left[1 - \sum_{r=0}^{i}(\mu w_2)^r/r!\ \exp(-\mu w_2)\right]$, and R is the solution of a quadratic matrix equation given in Appendix A. An application of Theorem 3.1 along with Property 1 of stochastic dominance yields the following bound on $F_{max}(w)$,

$$\begin{aligned} F_{max}(w) &\geq F_{max}^{(lb)}(w), \\ &= 1 - T^{(lb)}(w, w), \\ &= 1 - \pi[I - \exp(-\mu(I - R)w))]B(w). \end{aligned}$$

Similar arguments can be used to obtain the following bound on the distribution of the time until the first of the two tasks associated with a write request complete,

$$\begin{aligned} F_{min}(w) &\geq F_{min}^{(lb)}(w), \\ &= \pi \exp(-\mu(I - R)w - \mu w)C(w) \end{aligned}$$

where $C(w)$ is a $(B+1)$ element column vector with ordered components $\sum_{j=0}^{i}(\mu w)^j/j!$, $i = 0, 1, \cdots, B$. Substitution of $F_{min}^{(lb)}(w)$ into Equation (3.4) yields the following upper bound on $F_{max}(w)$,

$$\begin{aligned} F_{max}(w) \leq\ & \exp(-(\mu - \alpha_1 p_r \lambda - p_w \lambda)w) + \exp(-(\mu - \alpha_2 p_r \lambda - p_w \lambda)w) \\ & - \pi \exp(-\mu(I - R)w - \mu w)C(w). \end{aligned}$$

These can be used to obtain bounds on the moments of response times.

Table 3.1 Bounds for DP-RJ under different values of B ($\mu = 1$)

ρ	$B = 4$		$B = 8$		$B = 16$		$B = 32$	
	l.b.	u.b.	l.b.	u.b.	l.b.	u.b.	l.b.	u.b.
.1	1.653	1.653	-	-	-	-	-	-
.2	1.843	1.844	1.844	1.844	-	-	-	-
.3	2.082	2.092	2.089	2.089	-	-	-	-
.4	2.380	2.431	2.415	2.417	2.417	2.417	-	-
.5	2.764	2.925	2.861	2.879	2.875	2.875	-	-
.6	3.287	3.709	3.494	3.586	3.560	3.563	3.562	3.562
.7	4.093	5.103	4.444	4.823	4.676	4.716	4.708	4.708
.8	5.651	8.088	6.122	7.518	6.711	7.102	6.982	7.003
.9	10.37	17.63	10.81	16.55	11.73	15.19	13.05	14.17

Table 3.1 gives bounds on the average response time of write requests for different values of B. For ease of showing the lower and upper bounds, we normalize the service rate, $\mu = 1$. In this example, no read requests enter the system. If we take the average of the bounds for an approximation, the error is less than 3% for $\rho \leq 0.8$ when $B = 16$. An error of less than 5% can be achieved for $\rho = 0.9$ by taking $B = 32$. Table 3.2 presents bounds for the average response time of a write request in a system that serves both read and write requests. Here we observe that for a fixed disk utilization, the average response time of a write request increases as the fraction of read requests increases. This is because the coupling of the arrival processes to the two queues decreases as there are fewer write requests. Again B is chosen to be 32 for all values of ρ.

The CP-AQ Policy

Let $N_1(t)$ $(0 \leq N_1(t))$ be the number of requests in the common queue, $N_2(t)$ $(0 \leq N_2(t))$ be the number of write tasks in the auxiliary queue to the disk that lags behind (including the one in service), and $N_3(t)$ $(N_3(t) = 0, 1)$ denote whether the other disk is processing a request or not at time t. The state $\mathbf{N}(t) = (N_1(t), N_2(t), N_3(t))$ forms a Markov chain. If the system is stationary, $(\mathbf{N}(t) \rightarrow \mathbf{N}$ as $t \rightarrow \infty)$ and we let $q(m, n, l) = \lim_{t\to\infty} P[\mathbf{N}(t) = (m, n, l)]$, then these probabilities satisfy the following equations,

Table 3.2 Bounds for DP-RJ under different p_r ($\mu = 1$)

ρ	$p_r = 0$	$p_r = 1/4$	$p_r = 1/2$	$p_r = 3/4$
0.1	1.65	1.65	1.66	1.66
0.2	1.84	1.85	1.85	1.86
0.3	2.08	2.10	2.11	2.12
0.4	2.42	2.43	2.45	2.47
0.5	2.88	2.89	2.92	2.95
0.6	3.56	3.59	3.63	3.68
0.7	4.71	4.76	4.82	4.89
0.8	6.99±.01	7.08±.01	7.18±.01	7.31±.01
0.9	13.61±.56	13.77±.55	13.98±.55	14.24±.55

$$
\begin{aligned}
\lambda q(0,0,0) &= \mu q(0,1,0), \\
(\lambda+\mu) q(0,1,0) &= \mu q(0,2,0) + p_r \lambda q(0,0,0) + 2\mu q(0,1,1), \\
(\lambda+\mu) q(0,n,0) &= \mu q(0,n+1,0) + \mu q(0,n,1), && n = 2, \cdots, \\
(\lambda+2\mu) q(0,1,1) &= \mu q(0,2,1) + 2p_r \mu q(1,1,1) + p_w \lambda q(0,0,0) \\
&\quad + p_r \lambda q(0,1,0), \\
(\lambda+2\mu) q(0,2,1) &= \mu q(0,3,1) + 2p_r \mu q(1,1,1) + p_r \mu q(1,2,1) \\
&\quad + p_w \lambda q(0,1,0) + p_r \lambda q(0,2,0), \\
(\lambda+2\mu) q(0,n,1) &= \mu q(0,n+1,1) + p_w \mu q(1,n-1,1) \\
&\quad + p_r \mu q(1,n,1) + p_w \lambda q(0,n-1,0) \\
&\quad + p_r \lambda q(0,n,0), && n = 3, \cdots \\
(\lambda+2\mu) q(i,1,1) &= \mu q(i,2,1) + 2p_r \mu q(i+1,1,1) \\
&\quad + \lambda q(i-1,1,1), && i = 1, \cdots \\
(\lambda+2\mu) q(i,2,1) &= \mu q(i,3,1) + 2p_w \mu q(i+1,1,1) \\
&\quad + p_r \mu q(i+1,2,1) + \lambda q(i-1,2,1), && i = 1, \cdots \\
(\lambda+2\mu) q(i,n,1) &= \mu q(i,n+1,1) + p_w \mu q(i+1,n-1,1) \\
&\quad + p_r \mu q(i+1,n,1) + \lambda q(i-1,n,1), && i = 1, \cdots; n = 3, \cdots
\end{aligned}
$$

We develop bounds on the average response times for read and write requests by truncating one of the first two state variables, suitably modifying the infinitesimal generator and applying matrix-geometric techniques to the resulting model. Our computational experience indicates that we achieve greater accuracy for the same amount of computation by truncating $N_2(t)$. Consequently, we report on this approach. Unfortunately this system lacks the symmetry required to allow us to obtain both optimistic and pessimistic bounds from a single model as we did for the DP-RJ policy. We describe and analyze separate models for each bound.

Lower Bound for CP-AQ

In order to obtain lower bounds we impose the constraint $N_2(t) \leq B$. Whenever the second auxiliary queue contains B write tasks and a new write task arrives, it passes through without incurring any queueing or service delay. Consequently, the write request associated with this task completes as soon as its other task completes at the other disk.

This modified system can be modeled as a Markov chain with the same state description, $\mathbf{N}^{(lb)}(t) = (N_1^{(lb)}(t),\ N_2^{(lb)}(t),\ N_3^{(lb)}(t))$. Let $\mathbf{N}^{(lb)} = \lim_{t\to\infty} \mathbf{N}^{(lb)}(t)$.

Lower bounds on the response times for the true system can be calculated as follows. We define the following workload vectors $\mathbf{U}_i = (U_{1,i}, U_{2,i})$ and $\mathbf{U}_i^{(lb)} = (U_{1,i}^{(lb)}, U_{2,i}^{(lb)})$ where $U_{j,i}$ and $U_{j,i}^{(lb)}$, $j = 1,2$, are the amounts of unfinished work associated with each of the two disks immediately before the arrival of the i-th request in the true system and lower bound system respectively. We order the workload measures so that $U_{1,i} \leq U_{2,i}$ and $U_{1,i}^{(lb)} \leq U_{2,i}^{(lb)}$. Last, we define an ordering function $\psi(\mathbf{V})$ that takes an arbitrary finite element vector, $\mathbf{V}$, whose elements are real numbers and returns a vector with the elements ordered in increasing value.

Let $\{\tau_i\}_{i=1,\ldots}$ be a sequence of random variables denoting the time between request arrivals, i.e., τ_i is the time between the arrival of the $(i-1)$-th and i-th requests. Let $\{R_i\}_{i=1,\ldots}$ be a sequence of binary random variables that denote whether the requests are read or write. Here $R_i = 0$ if the i-th request is read and $R_i = 1$ if it is write. Let $\{X_{1,i}, X_{2,i}\}_{i=1,\ldots}$ be a sequence of random variables that denote the service times associated with the requests. If the i-th request is a write, then its two tasks are assigned $X_{1,i}$ and $X_{2,i}$ as their service times; otherwise if it is a read it is assigned service time of $X_{1,i}$. Let $\{A_i\}_{i=1,\ldots}$ be a sequence of random variables denoting whether the auxiliary queue associated with the disk that lags behind is full at the time that the requests are scheduled for service (i.e., the queue contains B requests at the time that the i-th request is scheduled into service). Here $A_i = 0$ if that queue is full when the i-th request is scheduled for service and $A_i = 1$ otherwise. The workload vectors evolve according to the following equations,

$$\mathbf{U}_{i+1} \quad = \quad (\psi(\mathbf{U}_i + (X_{1,i}, R_i X_{2,i})) - (\tau_i, \tau_i))^+, \tag{3.5}$$

$$\mathbf{U}_{i+1}^{(lb)} = (\psi(\mathbf{U}_i^{(lb)} + (X_{1,i}, A_i R_i X_{2,i})) - (\tau_i, \tau_i))^+.$$

Here $(V_1, V_2)^+ = (\max\{V_1, 0\}, \max\{V_2, 0\})$.

Define $W_i^{(f)}$, $W_i^{(r)}$, $W_i^{(f)(lb)}$, and $W_i^{(r)(lb)}$ to be the response times of the i-th request in the true and lower bound systems respectively for the case that they are either write or read. They are calculated from the workload vectors as follows:

$$\begin{aligned} W_i^{(f)} &= \max\{U_{1,i} + X_{1,i}, U_{2,i} + X_{2,i}\}, \\ W_i^{(r)} &= U_{1,i} + X_{1,i}, \\ W_i^{(f)(lb)} &= \max\{U_{1,i}^{(lb)} + X_{1,i}, U_{2,i}^{(lb)} + A_i X_{2,i}\}, \qquad (3.6) \\ W_i^{(r)(lb)} &= U_{1,i}^{(lb)} + X_{1,i}. \qquad (3.7) \end{aligned}$$

We further define another random variable W_i', which is useful in our following calculations, as,

$$W_i' = \max\{U_{1,i}^{(lb)} + X_{1,i}, U_{2,i}^{(lb)} + X_{2,i}\}.$$

We have the following theorem. Here $W^{(r)}$, $W^{(f)}$, $W^{(r)(lb)}$, $W^{(f)(lb)}$, and W' are the stationary values of $W_i^{(r)}$, $W_i^{(f)}$, $W_i^{(r)(lb)}$, $W_i^{(f)(lb)}$, and W_i' respectively.

Theorem 3.2 *The true system and the modified system satisfy the following relationships,*

1. $W^{(f)} \geq_{st} W' \geq_{st} W^{(f)(lb)}$,
2. $W^{(r)} \geq_{st} W^{(r)(lb)}$,
3. $\mathbf{N} \geq_{st} \mathbf{N}^{(lb)}$.

Proof. If the workload vectors initially satisfy $\mathbf{U}_0 \geq \mathbf{U}_0^{(lb)}$, then an induction argument can be used to show $\mathbf{U}_i \geq \mathbf{U}_i^{(lb)}$ for $i = 0, 1, \cdots$. Whenever the real system is ergodic (i.e., the *r.v.*'s $\mathbf{U}_i$ and $\mathbf{U}_i^{(lb)}$ converge to the limiting *r.v.*'s $\mathbf{U}$ and $\mathbf{U}^{(lb)}$), $\mathbf{U} \geq_{st} \mathbf{U}^{(lb)}$. This is certainly true under the assumptions of Poisson arrivals, exponential service times, and $p_r\lambda + 2p_w\lambda < 2\mu$. It then follows from the relation $\mathbf{U}_i \geq_{st} \mathbf{U}_i^{(lb)}$ that $W_i^{(f)} \geq_{st} W_i' \geq_{st} W_i^{(f)(lb)}$ and $W_i^{(r)} \geq_{st} W_i^{(r)(lb)}$, $i = 1, \cdots$. Again, if the real system is ergodic, then $W^{(f)} \geq_{st} W' \geq_{st} W^{(f)(lb)}$ and $W^{(r)} \geq_{st} W^{(r)(lb)}$.

The proof of the third relation is similar to the proof of Theorem 3.1. ∎

Remark. The transient results hold for an arbitrary arrival process and arbitrary service times.

Corollary 3.1 *Let $N^{(r)}$, $N^{(f)}$, $N^{(r)(lb)}$, and $N^{(f)(lb)}$ denote the stationary queue lengths of read and write requests in the two systems. The following inequalities hold between their expectations, $E[N^{(r)}] \geq E[N^{(r)(lb)}]$ and $E[N^{(f)}] \geq p_w \lambda E[W'] \geq E[N^{(f)(lb)}]$.*

Proof. The proof follows from Little's result and the fact that $E[X] \geq E[Y]$ whenever X and Y are r.v.'s such that $X \geq_{st} Y$. ∎

Again, the matrix-geometric method is used to solve the modified lower bound CP-AQ system. The detailed calculations are given in Appendix B.

In the remainder of this section, we will derive expressions for the lower bounds for the expected number of read and write requests as well as the expected response time for read and write requests. In the case of write requests, we will make use of Theorem 3.2 and its Corollary.

The expected length of the common queue, $E[N_1^{(lb)}]$, is

$$E[N_1^{(lb)}] = y(1)R[I - R]^{-2}e,$$

where the vector $y(1)$ and rate matrix R are solved for in Appendix B, and e is a vector with all elements 1. The average number of read requests that are in service equals $p_r \lambda/\mu$. Consequently, by Theorem 3.2 and Corollary 3.1, a lower bound on the expected number of read requests is

$$E[N^{(r)}] \geq E[N^{(r)(lb)}] \equiv p_r E[N_1^{(lb)}] + p_r \lambda/\mu.$$

Little's result yields

$$E[W^{(r)}] \geq E[W^{(r)(lb)}] = E[N_1^{(lb)}]/\lambda + 1/\mu.$$

The expected number of write requests in the common queue of the modified system is $p_w E[N_1^{(lb)}]$. A lower bound on the expected number of write requests that

are in service is obtained by first determining the expected service delay incurred by a write request (i.e., time from beginning of processing of first task until completion of both tasks) in the modified system and then applying Little's result. Denote this expected delay by $d_1^{(lb)}$. The time required to service a write request depends on the length of the auxiliary queue and whether both disks are busy. If a write request begins service when the auxiliary queue contains $i-1$ tasks ahead of it, then the time to complete service is denoted by $h(i)$ which satisfies the recurrence

$$h(1) = 3/(2\mu),$$
$$h(i) = 1/(2\mu) + h(i-1)/2 + i/(2\mu), \quad i = 2, 3, \cdots.$$

The first term in the recurrence for $h(i)$ corresponds to the average delay until the first of the two disks completes. If the disk associated with the auxiliary queue completes, then the write request observes the system with one less task in the auxiliary queue. The average delay in this case corresponds to the second term in the above recurrence. Last, when a write request begins service, one of its tasks immediately begins service in the other disk. Consequently, if this disk completes, the average of the remaining delay of the write corresponds to the average delay of the task still present in the auxiliary queue. This is the sum of average service times of this task and of the $i-1$ tasks ahead of it in the auxiliary queue. This gives rise to the third term. This recurrence has the following solution

$$h(i) = (i + 2^{-i})/\mu, \quad i = 1, ...$$

There are three possible scenarios when a write request arrives to the system: (1) both disks may be idle; (2) one of the disks may be idle; and (3) both disks may be busy. In the first two cases, the request initiates service immediately. In the last case, the request begins service only when it is at the head of the common queue. If at this moment an auxiliary queue has not built up at either disk, the request begins service as soon as either disk completes service of its request. Otherwise, the request begins service only if the disk without the auxiliary queue completes service.

The above observations yield the following expression for $d_1^{(lb)}$,

$$d_1^{(lb)} = \frac{q(0,0,0)\lambda h(1) + \lambda y(0) V_1 + \mu y(1)[I-R]^{-1} V_2}{q(0,0,0)\lambda + \lambda y(0) + \mu y(1)[I-R]^{-1} V_3}$$

where

$$\begin{aligned} V_1 &= (h(2), h(3), \cdots, h(B), h(B+1))^T, \\ V_2 &= (2h(2), h(3), h(4), \cdots, h(B), h(B+1))^T, \\ V_3 &= (2, \underbrace{1, 1, \cdots, 1}_{B-1})^T. \end{aligned}$$

The coefficient 2 for the first element of V_2 and V_3 is a consequence of the observation that service of a new request is initiated whenever a departure occurs from either disk and there is no auxiliary queue in the case that both disks are busy.

The expected value of W', which bounds the $W^{(f)}$ from below, can now be computed by,

$$E[W'] = E[N_1^{(lb)}]/\lambda + d_1^{(lb)}.$$

This can be used to obtain the following lower bound on the average number of write requests (Corollary 3.1),

$$E[N^{(f)}] \geq p_w E[N_1^{(lb)}] + p_w \lambda d_1^{(lb)}.$$

Lower bounds on the expected response times are found in Tables 3.3 for different mixtures of read and write requests (the entries in columns $p_r = 1/2$ and $p_r = 3/4$ are both upper and lower bounds accurate to three places). These values were calculated for $B = 16$ for all values of ρ.

Upper Bound for CP-AQ

A upper bound on the performance of the CP-AQ policy is obtained in a similar manner. The auxiliary queue is allowed to have up to B requests. Whenever this queue is full and a request completes at the other disk, the completed request is required to repeat its service. This avoids the possible event of a write request at the head of the common queue placing a task in the auxiliary queue and thus increasing its length above B. The resulting Markov chain is identical to the one described for the lower bound except for the following changes in the transition rates,

1. Remove state $(0, B, 0)$ and all transitions to and from it.

2. Remove the transition from state $(i, B, 1)$ to $(i-1, B, 1)$, $i = 2, \cdots$.

Let $\mathbf{N}^{(ub)}(t) = (N_1^{(ub)}(t), N_2^{(ub)}(t), N_3^{(ub)}(t))$ be the state of this new system. Let $W^{(r)(ub)}$ and $W^{(f)(ub)}$ denote the stationary response times for a read request and write request respectively. Let $\mathbf{N}^{(ub)}$ be the stationary queue length vector for the upper bound system. We state the following theorem.

Theorem 3.3 *The true system and the modified system exhibit the following relationships,*

1. $W^{(ub)(f)} \geq_{st} W^{(f)}$,
2. $W^{(ub)(r)} \geq_{st} W^{(r)}$,
3. $\mathbf{N}^{(ub)} \geq_{st} \mathbf{N}$.

Proof. We observe that the workload vector for the upper bound system satisfies Equation (3.5) except that the service times are no longer the same. Instead the service times are $X'_{i,j} \geq X_{i,j}$, $i = 1, 2$; $j = 1, \ldots$. The inequality is due to the fact that an occasional task is required to take an additional service time in the upper bound system. Consequently a simple proof by induction yields $\mathbf{U}_i \leq_{st} \mathbf{U}_i^{(ub)}$ for $i = 1, \ldots$. The rest of the proof duplicates the arguments in the proof of Theorem 3.2. ∎

The procedure for calculating the lower bounds on the average buffer occupancies and response times in the previous section applies with no change to the computation of the upper bounds. Numerical results for these bounds can be found in Table 3.3 for different mixes of read and write requests. Again B is taken to be 16. One observes that the bounds are tight for disk utilizations less than 0.9 or when the fraction of read requests exceeds 1/4.

3.2.2 Models for Remaining Policies

In this subsection, we simply describe Markov models for all other policies discussed in Section 3.1. The basic techniques used to provide bounds for these systems are similar to those presented above.

Table 3.3 Bounds for CP-AQ under different p_r ($\mu = 1$)

	$p_r = 0$	$p_r = 1/4$		$p_r = 1/2$		$p_r = 3/4$	
ρ	$E[W_f]$	$E[W_r]$	$E[W_f]$	$E[W_r]$	$E[W_f]$	$E[W_r]$	$E[W_f]$
0.1	1.65	1.03	1.65	1.02	1.65	1.02	1.65
0.2	1.84	1.07	1.84	1.07	1.83	1.06	1.81
0.3	2.09	1.14	2.06	1.14	2.03	1.12	1.99
0.4	2.42	1.25	2.35	1.24	2.28	1.23	2.21
0.5	2.88	1.42	2.74	1.41	2.61	1.39	2.48
0.6	3.56	1.70	3.28	1.69	3.05	1.65	2.84
0.7	4.70±.02	2.20	4.12	2.19	3.73	2.12	3.41
0.8	6.92±.19	3.26	5.66	3.24	4.97	3.09	4.47
0.9	13.23±2.11	6.68±.07	9.73±.09	6.51	8.45	6.06	7.52

The DP-SQ Policy

This policy can be modeled by a Markov chain $\mathbf{N}(t) = (N_{max}(t), N_{min}(t))$, where $N_{max}(t)$ is the number of tasks in the longest queue and $N_{min}(t)$ is the number of tasks in the shortest queue at time t. We are interested in the stationary behavior of this policy. We describe how bounds are obtained for this system.

Lower Bound for the DP-SQ Policy: The optimistic bound for DP-SQ can be obtained by truncating one of the state variables, say N_{min}, to a constant B, and then applying the matrix-geometric method. The modified system behaves as follows: if a write request arrives and finds B tasks in the shortest queue, it generates only one task which enters the longest queue. In this case, the write request completes as soon as the task in the longest queue finishes. If an arriving read request finds B tasks in the shortest queue, then it exits the system immediately and incurs zero delay.

We denote the state of the modified system as $\mathbf{N}^{(lb)} = (N_{max}^{(lb)}, N_{min}^{(lb)})$, and the stationary response time as $W^{(r)(lb)}$ and $W^{(f)(lb)}$.

The stationary distribution for this modified system satisfies the following equations. The calculations leading to their solution are based on the matrix geometric approach and are omitted.

$$
\begin{aligned}
\lambda q(0,0) &= \mu q(1,0), \\
(\lambda+\mu)q(i,0) &= \mu q(i,1)+\mu q(i+1,0), && i=1,\cdots \\
(\lambda+2\mu)q(i,i) &= p_r\lambda q(i,i-1)+\mu q(i+1,i) \\
&\quad + p_w\lambda q(i-1,i-1), && i=1,\cdots,B-1. \\
(\lambda+2\mu)q(i+1,i) &= p_w\lambda q(i,i-1)+p_r\lambda q(i,i)+ \\
&\quad p_r\lambda q(i+1,i-1)+2\mu q(i+1,i+1), && i=1,\cdots,B-1. \\
(\lambda+2\mu)q(i,j) &= p_w\lambda q(i-1,j-1)+p_r\lambda q(i,j-1) \\
&\quad + \mu q(i,j+1)+\mu q(i+1,j), && j=1,\cdots,B-1; \\
& && i=j+2,\cdots \\
(\lambda+2\mu)q(B,B) &= p_w\lambda q(B-1,B-1)+p_r\lambda q(B,B-1) \\
&\quad + \mu q(B+1,B), \\
(p_w\lambda+2\mu)q(B+1,B) &= (p_r\lambda+\mu)q(B,B)+p_w\lambda q(B,B-1)+ \\
&\quad p_r\lambda q(B+1,B-1)+\mu q(B+2,B), \\
(p_w\lambda+2\mu)q(i,B) &= p_w\lambda q(i-1,B)+p_w\lambda q(i-1,B-1) \\
&\quad + p_r\lambda q(i,B-1)+\mu q(i+1,B) && i=B+2,\cdots.
\end{aligned}
$$

Note: $q(i,j)=0$ for $i<j$.

Upper Bound for the DP-SQ Policy: To obtain a upper bound, we make a slight change to the system state description. The state is now defined as $\mathbf{N}(t)=(N_{min}(t),\Delta(t))$, where $N_{min}(t)$ is the number of tasks in the shortest queue at time t, and $\Delta(t)$ is the difference between the shortest queue and the longest queue at time t, i.e., $N_{max}(t)=N_{min}(t)+\Delta(t)$. An upper bound on performance is obtained by truncating the state variable Δ to B. Whenever Δ equals B and there is a departure from the shortest queue, while the true system will transit from (N_{min},B) to $(N_{min}-1,B+1)$, the modified system generates a fictitious task occupying the disk, so that the system state is unchanged.

Let $\mathbf{N}^{(ub)}(t)=(N_{min}^{(ub)}(t),\Delta^{(ub)}(t))$, and the stationary response time be $W^{(r)(ub)}$ and $W^{(f)(ub)}$. We omit the equations that describe the behavior of the stationary distribution for this upper bound system along with the calculations leading to their solution.

We state the following theorem. Its proof is similar as that of Theorems 3.1 and 3.2, and therefore is omitted.

Theorem 3.4 *The true system and the two modified systems satisfy the following relationships,*

1. $N_{max}^{(lb)} \leq_{st} N_{max} \leq_{st} N_{max}^{(ub)}$,
2. $N_{min}^{(lb)} \leq_{st} N_{min} \leq_{st} N_{min}^{(ub)}$,
3. $W^{(r)(lb)} \leq_{st} W^{(r)} \leq_{st} W^{(r)(ub)}$,
4. $W^{(f)(lb)} \leq_{st} W^{(f)} \leq_{st} W^{(f)(ub)}$.

The DP-SQ-MS Policy

Observe that the only difference in the behavior of the DP-SQ-MS and the DP-SQ policies occurs when a read request arrives at an idle system. While under the DP-SQ policy, the read randomly selects a disk for service, under the DP-SQ-MS policy, it goes to the disk with the minimum arm movement. The Markov chain for the DP-SQ-MS policy is obtained from that of the DP-SQ policy by adding an additional dimension $N_3(t)$, that maintains information regarding the presence or absence of read requests that arrive to an idle system. Because of the introduction of the third dimension to the Markov chain, the equations describing the Markov chain are a little more complicated than those of DP-SQ. To help understand the behavior of the Markov chain, we include detailed state definitions and calculations in Appendix C.

The DP-MW Policy

Consider the behavior of the DP-MW policy. Suppose at some time t, there are n tasks waiting in the first queue, and m tasks waiting in the second queue, without loss of generality, $n \geq m$. In this case, the last m tasks in both queues are associated with the m most recently arrived requests who have not received any service yet. These m requests correspond to those requests waiting in the common queue under the CP-AQ policy. On the other hand, the first $n-m$ tasks waiting in the first queue must be write tasks, which correspond to those write tasks waiting in the auxiliary queue under the CP-AQ policy. This observation leads to the conclusion that the DP-MW policy behaves identically to the CP-AQ policy.

The CP-AQ-MS and the DP-MW-MS Policies

The same observation as that made in the last subsection leads to the equivalence of the DP-MW-MS and CP-AQ-MS policies. As was done in analyzing the DP-SQ-MS

policy, the Markov chain describing the behavior of the CP-AQ-MS policy differs from that of the CP-AQ policy only when a read request arrives at an empty system. This is accounted for by introducing a new state variable that keeps information regarding those read requests that arrive at an empty system.

The DP-MR Policy

Similar to the DP-SQ policy, this system can also be modeled by a Markov chain $\mathbf{N}(t) = (N_{max}(t), N_{min}(t))$, where $N_{max}(t)$ is the number of tasks in the longest queue at time t, and the $N_{min}(t)$ is the number of tasks in the shortest queue at time t. However, the infinitesimal generator matrix differs from that of DP-SQ.

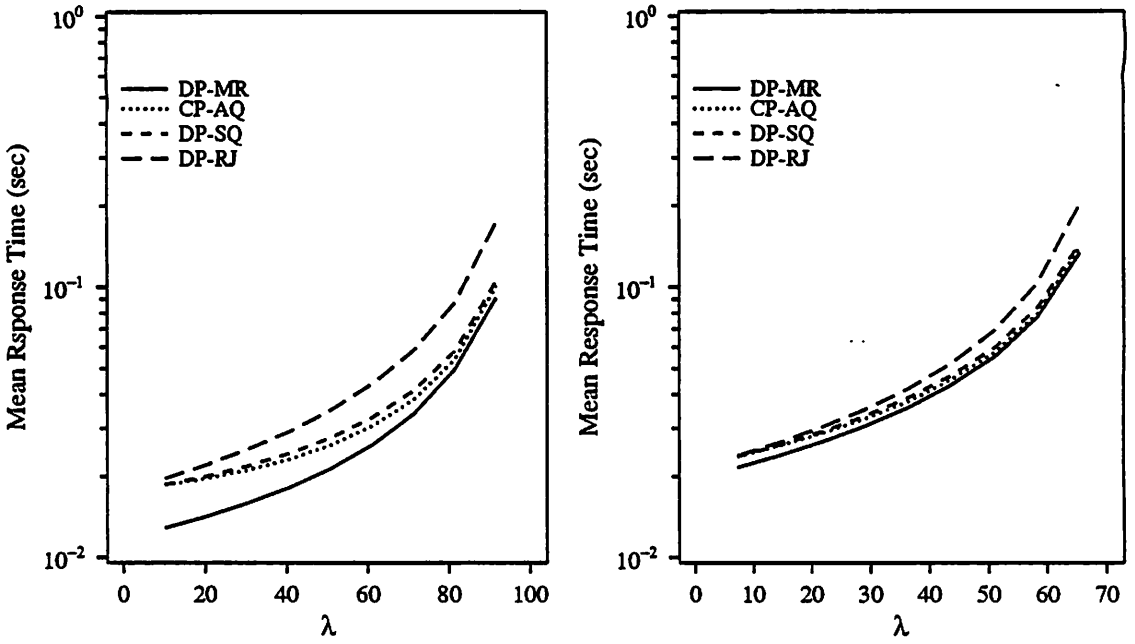
Lower bound on the stationary behavior of this policy is obtained by truncating the state N_{min} and applying the matrix-geometric method. Upper bound is obtained by truncating and solving for the stationary probabilities of the Markov chain $\mathbf{N}(t) = (N_{min}(t), \Delta(t))$ where $\Delta(t)$ denotes the difference between the longest and shortest queues. Similar relationships as stated in Theorem 3.4, i.e., the stochastic ordering between $\mathbf{N}$ and $\mathbf{N}^{(lb)}, \mathbf{N}^{(ub)}$, $W^{(r)}$ and $W^{(lb)(r)}, W^{(ub)(r)}$, and $W^{(f)}$ and $W^{(lb)(f)}, W^{(ub)(f)}$, also exist here. The detailed proof and calculations are omitted.

3.3 Performance Comparisons

In this section, we compare the performance of the policies developed and analyzed in the previous two sections. In our experiments, we assume that the arrival rate of I/O requests to the subsystem is λ, the probability that a request being a read is p_r, and the probability that a request being a write is $p_w = 1 - p_r$. In particular, we examine two cases, in which most of requests are reads ($p_r = 0.75$) and most of requests are writes ($p_r = 0.25$). Typically, if there is no cache, it is believed that disk accesses are dominated by reads. However, in the presence of a cache, most of the reads can be satisfied by the cache, and therefore the disk access patterns are expected to consist mostly of writes. In this case, the results obtained by setting $p_r = 0.25$ are more interesting.

As indicated in the analytic models above, the disk service times are assumed to be exponentially distributed with rate μ, where $1/\mu$ is obtained from real disk

access times (Equation (2.8)). The disk service times for those reads which take advantage of *minimum seek* are assumed to be exponentially distributed with rate μ', where $1/\mu'$ is calculated from Equation (2.10) with sequential probability $p_s = 0.5$. We further assume that under the DP-RJ policy, reads choose either disk with the same probability, $\alpha_1 = \alpha_2 = 1/2$. We estimate the mean I/O response times under different policies by averaging the lower and the upper bounds obtained from the models introduced in the previous section. In fact, as illustrated in previous tables, the bounds are very tight. Hence, the estimates we obtained are good approximations.

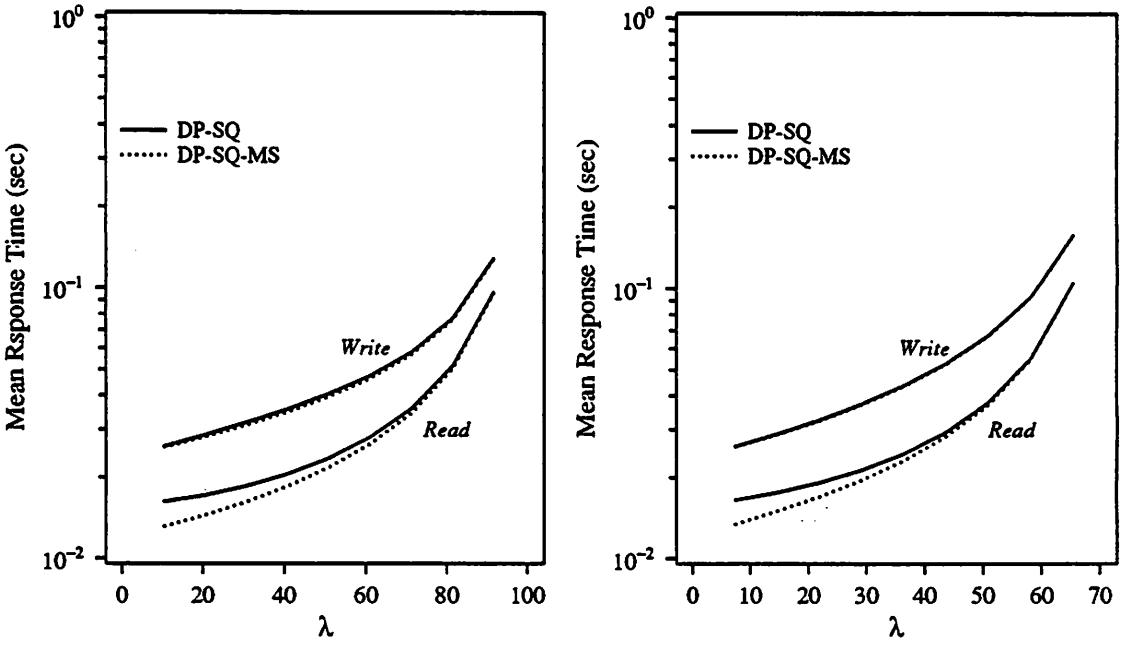


(a) p_r= 0.75 (b) p_r= 0.25

Figure 3.3 Performance of Different Policies.

In Figure 3.3, we show the mean response times for read and write requests as a function of arrival rate. From these results, we observe that the DP-SQ policy performs better than the DP-RJ policy. This is because the DP-SQ policy can achieve a better balance among the queue lengths. The CP-AQ policy shows a higher performance than the DP-SQ policy because, instead of balancing the queue lengths, the CP-AQ policy tries to balance the unfinished work among the two disks. It is interesting to note, however, that the DP-MR policy provides the best performance among all the policies. Under the DP-MR policy, although the disks are facing higher

loads, i.e., both disks may simultaneously serve the same read request, the benefit is probably explained by the fact that each disk will not be required to service the read for more than the minimum of two independently distributed disk access times. In addition, the DP-MR policy accrues a definite advantage when the system is empty at the arrival of a read request.



(a) p_r= 0.75 (b) p_r= 0.25

Figure 3.4 The Impact of Minimum Seek Discipline.

Figure 3.4 shows the impact of routing a read request to the disk requiring the least arm movement when both disks are idle at the time of its arrival. The results are obtained by comparing the DP-SQ-MS policy against the DP-SQ policy. As we expected, the performance improvement is noticeable when the workload decreases, since at high loads, it is less likely that an arriving request finds an empty system. From the Figure, we observe that, by using the minimum seek mechanism, the mean response time for reads improve at low loads, but that the mean response times for writes are basically the same under the two policies. The improvement of DP-MW-MS(CP-AQ-MS) over DP-MW(CP-AQ) is similar and is omitted here.

3.4 Simulation Experiments

In our analytical models, disk service times are assumed to be exponentially distributed for tractability. However, in reality, disk service time are not exponential. In order to validate our performance results on different policies, we have developed a simulator for a mirrored disk that simulates the seek, rotational latency, and transfer times on an individual disk properly. The seek distance is generated according to the distribution described by Equation (2.1). The rotational latency of each disk is uniformly distributed in $[0, 16.7]$ and the transfer time is determined by the disk transfer rate and the I/O size.

The simulation results are obtained by averaging over 40 runs, each run including 50,000 I/O requests. We obtained 95% confidence intervals whose widths are less than 2% of the estimated point values.

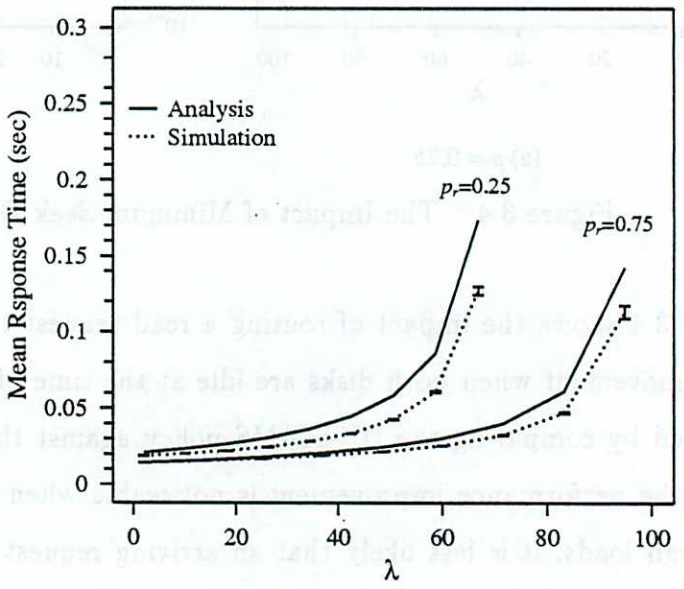


Figure 3.5 Model Validation via Simulations (DP-SQ-MS).

Figure 3.5 illustrates the analytic and simulation results for the shortest queue policy DP-SQ-MS. Although the analytic results are slightly higher than those obtai-

ned through simulations due to the unrealistic exponential service time assumption made in our analysis, the qualitative behavior of these two curves remains the same. We have observed the same behavior in a number of other experiments. From these simulation experiments, we observe that our analytic models may slightly overestimate the mean I/O response times, especially at high loads. However, the performance ordering of different policies predicted by our analytic models is the same as that obtained by simulations. Therefore, our conclusions, such as "the policy DP-SQ-MS is better than DP-RJ", remain valid.

3.5 Summary

In this chapter, we have described several I/O scheduling policies suitable for a mirrored disk subsystem which serves both read and write requests. We developed analytic models that can be used to bound the performance of these policies. These bounds are obtained by appropriately approximating a Markov chain with two unbounded state variables by a chain with only one unbounded state variable. These chains are solved using the Matrix geometric methodology and lead to either upper or lower bounds on the performance metrics of interest. Numerical results show that these bounds can be quite accurate. Consequently, the performance of the different policies can be studied under a wide variety of workloads.

Using these models, we observed that the performance of *centralized* policies is better than that of most of the *distributed* policies, since the former try to balance the unfinished workload between the two disks. Allowing a read which arrives to an empty system to start service at the disk requiring the least arm movement may lead to a slight performance improvement, but only when the system workload is light. The policy that always assigns a read to the shortest queue performs reasonably well. Finally, the policy that assigns each read request to both disks and as soon as one completes, aborts the other leads yet to an additional performance improvement. This policy, however, requires the hardware support of aborting a request in service.

While in our models, all of the queues are served in a first-come-first-serve fashion, it is easy to employ other algorithms such as SCAN or SSTF (*shortest seek time first*) within each queue. Our simulation results show that all policies benefit from these

arm scheduling algorithms when the workload is high. We expect, however, those policies that maintain longer queues, such as DP-MR and DP-MW, to benefit more than other polices.

Last, we indicate that all of these policies studied in this chapter apply immediately to the mirrored disk array (RAID 1) to be examined in the next chapter.

CHAPTER 4

DISK ARRAYS FOR HIGH PERFORMANCE COMPUTING SYSTEMS

The main purpose of this chapter is to study the design issues and performance evaluation of disk arrays. A disk array typically operates in a *normal* mode, i.e., all disks in the array are operational. However, since there are many disks in the array, occasionally a disk failure may occur. In this case, the disk array transits to a *failure* mode, and if there is a hot spare disk in the array, it immediately moves to a *rebuild* mode. In this chapter, we will focus on the performance of disk arrays operating in a normal mode.

The performance gains achievable by using disk arrays are two-fold. On one hand, for applications in which each I/O request typically accesses a small amount of data, such as transaction processing systems and workstation environments[1] [75, 44, 76, 88], disk arrays can support multiple I/O requests concurrently. On the other hand, for applications in which each request typically accesses a large amount of data, such as supercomputing and image processing systems, disk arrays can support the data transfer from multiple disks simultaneously. We first describe several disk array architectures of most interest to us. We then propose scheduling policies suitable for different architectures and applications, and develop analytic models for studying these systems. Last, the performance of these disk arrays is studied and compared with each other by using these analytic models and simulations.

4.1 Disk Array Architectures

As mentioned in Section 1.4, five levels of RAIDs have been identified. Among the five, we are most interested in RAID 1 and RAID 5, as well as their variants.

[1] In these application areas, there are some users demanding a large amount of data, such as a scan query encountered in a database system. However, many systems, such as UNIX, divide such a large request into many small requests and serve them individually. In this case, the I/O subsystem sees a sequence of small reads or writes arriving to it.

We consider disk arrays containing N identical disks. In the architectures of interest to us, data is *block interleaved* across W disks, where W is the *stripe width*, $W \leq N$. For example, a file f is logically divided into blocks $f_1, f_2, ..., f_n$, which are stored on contiguous disks in the stripe (n might be larger than the stripe width W). If a request accesses exactly W blocks of data in a stripe (i.e., aligned with the stripe boundaries), it is called a "full stripe I/O". Otherwise, it is called a "partial stripe I/O"[2]. The main advantage of block interleaving is that it supports the concurrent execution of multiple small I/O requests. Other alternatives are *bit* or *byte interleaving*, as exemplified by RAID 2 and RAID 3, respectively. However, both of these require that all disks in the array be involved in servicing each I/O request, and therefore at most one I/O request can be executed at a time. The most significant benefit achievable from these architectures is the fast transfer rate. While these architectures may be suitable for applications in which each request transfers a large amount of data, they are not appropriate for applications such as transaction processing and workstation environments. In the later case, since each request typically transfers a small amount of data, the advantage of a fast transfer rate diminishes. On the other hand, since all disks are needed to serve a request, the disk array cannot support multiple small I/O's concurrently. A common advantage of interleaving data across stripes in an array is the load sharing of heavily used files among multiple disks.

4.1.1 RAID 1 and Its Variants

In this subsection, we describe the RAID 1 architecture and its variants. Specifically, we focus on the existing architecture RAID 1, which is also called *mirrored array* or *mirrored declustering* [91, 34], and a variant *chained declustering* that has been proposed in the literature [51]. We then propose a new architecture, called *group-rotate declustering*, that combines the advantages of both mirrored and chained declustering.

---

[2]If a request accesses multiple blocks of data which cover several stripes plus a partial stripe, we still consider it to be a "partial stripe I/O".

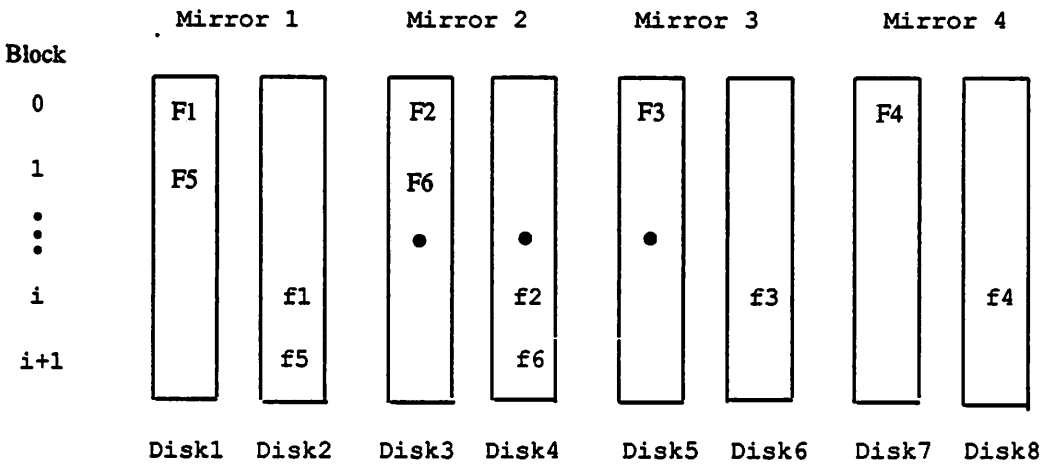


Figure 4.1 The RAID 1 (Mirrored Declustering) Architecture ($N = 8$).

Mirrored Declustering: In mirrored declustering, the N disks are configured as $N/2$ pairs of mirrored disks, with the i-th pair termed *mirror i*. Two copies of data are striped over these $N/2$ pairs. Each pair contains the same data, which is not necessarily stored at the same location on the two disks. However, contiguous logical blocks of both copies are allocated sequentially on the two $N/2$–striped disk pairs. Figure 4.1 shows a scenario where file f, consisting of 6 blocks, is allocated on an array of 8 disks. In the Figure, we use F_i to denote the first copy and f_i to denote the second copy of the i-th file block, $i = 1, 2, \cdots$. A read request can be satisfied by either copy, but a write request requires an update of both copies. One of the drawbacks of this architecture is that when a disk fails and the disk array operates in a *failure* or *rebuild* mode, all of the traffic originally directed to the failed disk is now redirected to its mirror, which may become a bottleneck.

Chained Declustering: To overcome the above bottleneck during recovery period, Hsiao and DeWitt [51] proposed the *chained declustering* architecture, where two physical copies of data, termed the *primary copy* and the *backup copy*, are maintained. As shown in Figure 4.2, if a primary data block is allocated on disk i, its duplicate backup block is allocated on disk $(i+1)$ mod N. While in [51] a read is only allowed to access the primary copy in normal mode, we will attempt to improve its performance

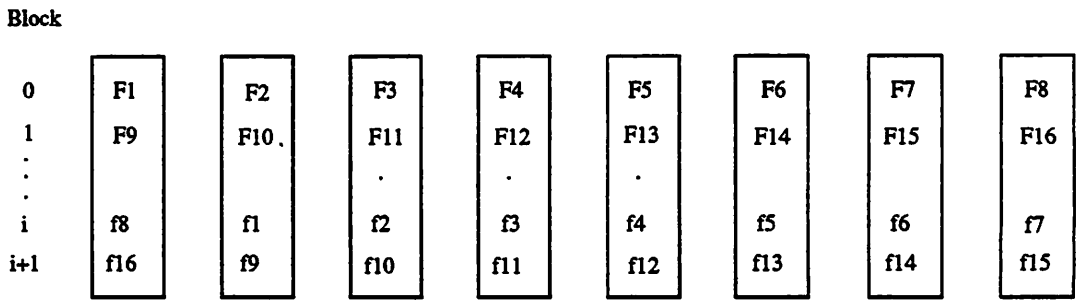


Figure 4.2 The Chained Declustering Architecture ($N = 8$).

by allowing a read to be executed on either copy.

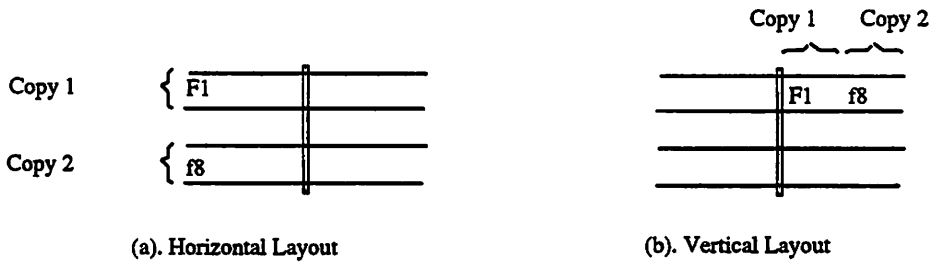


Figure 4.3 Data Layout Approaches for Chained Declustering.

There are many variants on this architecture according to how the two data copies are laid out on the disks. We study two of these data layouts in this chapter, as shown in Figure 4.3. Typically, a disk spindle consists of a collection of platters. Each platter contains a number of tracks. A cylinder consists of tracks from the same position on each platter. For workstation disks, usually read/write heads are attached to an actuator which positions the heads to an appropriate track, and only one head transfers data at a time. The two approaches are described as follows:

- **Horizontal Layout (HL)**: By using this approach, half of the platters on each spindle is used to store the first copy, and the remaining platters are used to store the second copy;
- **Vertical Layout (VL)**: By using this approach, the first copy occupies half of the cylinders, and the second copy occupies the other half.

For example, consider a disk spindle consisting of 4 platters labeled 0 to 3 and 1000 cylinders. Then the *horizontal* approach will allocate copy 1 on platter 0 and 1, occupying all tracks on these two platters, and copy 2 on platter 2 and 3. On the other hand, the *vertical* approach will allocate the copy 1 on cylinder 0 to 499, including all the tracks on each cylinder, and the copy 2 on cylinder 500 to 999[3]. As we will see later in Section 4.3.1, the HL approach is favored for writes whereas the VL approach is favored for reads in large I/O environments.

Notice that a main difference between the mirrored declustering and chained declustering is that when a read request needs to access a large amount of data, the mirrored declustering can support two such "large" reads concurrently, but chained declustering can only support one at a time. However, when a disk fails, chained declustering can evenly distribute the burden of the failed disk to all of the other $N-1$ operational disks.

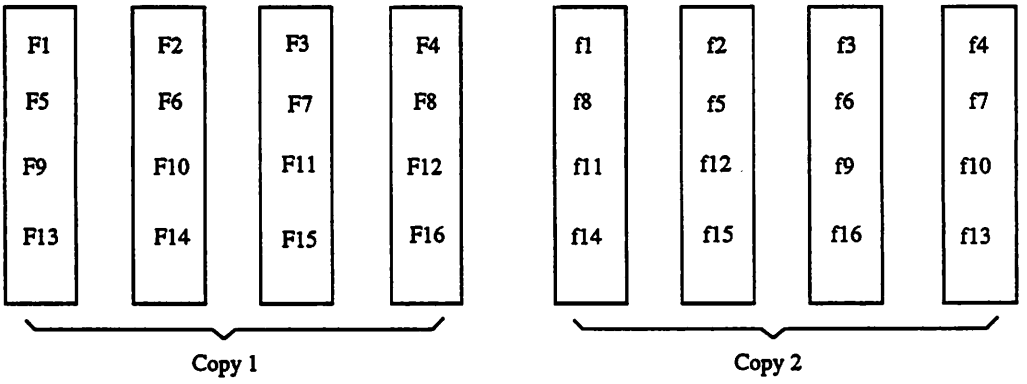


Figure 4.4 The Group-Rotate Declustering Architecture ($N = 8$).

Group-Rotate Declustering: By taking advantage of both mirrored and chained declustering, we propose a third architecture, called *group-rotate declustering*, as shown in Figure 4.4. In this architecture, the first copy is stored in the same way as mirrored declustering, but the second copy is rotated among the remaining $N/2$ disks. Clearly, group-rotate declustering can still support two "large" reads concurrently as

[3]We assume that each track has the same capacity.

mirrored declustering above, but during failure it distributes the burden of the failed disk evenly among $N/2$ disks.

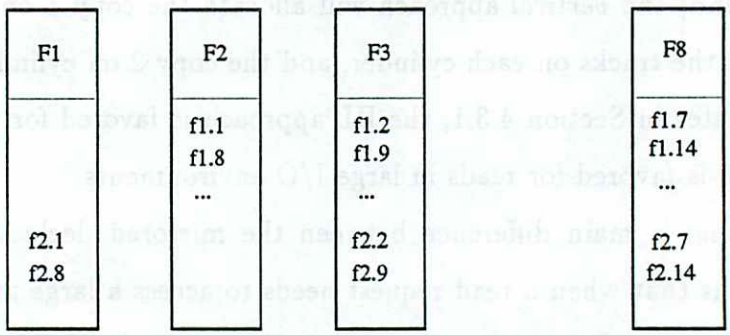


Figure 4.5 The Interleaved Declustering Architecture ($N = 8$).

Interleaved Declustering: There is another architecture proposed in the literature, called *interleaved declustering* [115], as shown in Figure 4.5. As in [76], the Figure shows that the primary copy is divided into N large blocks, denoted by F_i, $i = 1, \cdots, N$, one for each disk. The backup copy of F_i is subdivided into small blocks $f_{i.1}, f_{i.2}, \cdots, i = 1, \cdots, N$, and placed on disks other than disk i. This architecture was originally proposed for database systems, in which reads are only allowed to access the primary copy. This architecture has been studied extensively in the literature [51, 52, 76, 34]. An obvious drawback of this architecture is that it does not support applications in which I/O requests may update a large amount of data, since although all disks are involved in serving a write, the write has to wait until the large block (the primary copy) is updated. As indicated in [76], since this architecture maintains a large block of data on a disk, it may suffer from skewed accesses when the write ratio is high. Hence, we exclude it from our discussions.

4.1.2 RAID 5 and Parity Striping

Unlike RAID 1 and its variants which duplicate each data item, the RAID 5 and Parity Striping architectures provide parity information to achieve reliable service. This is attractive because only 1 block of redundant information is required for every

$N-1$ data blocks. This is in contrast to RAID 1 and its variants which double the number of disks.

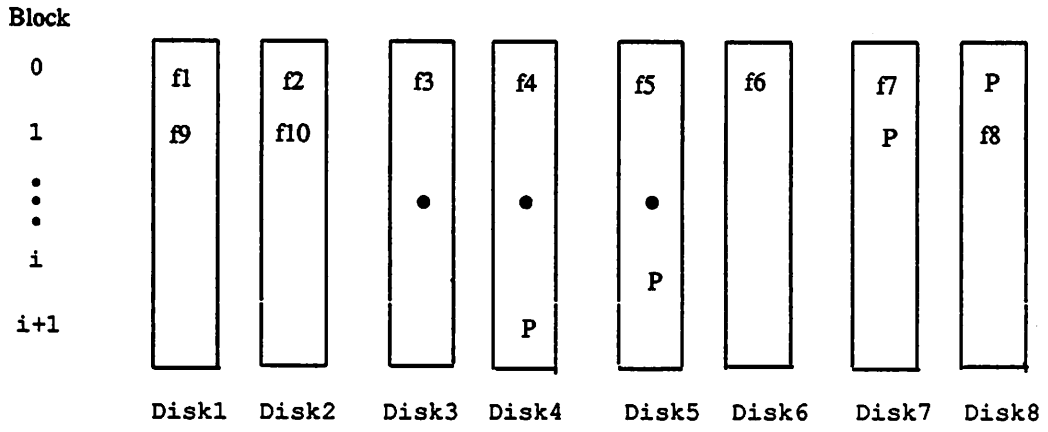


Figure 4.6 The RAID 5 Architecture ($N = 8$).

RAID 5: The RAID 5 architecture is shown in Figure 4.6. For reliability purposes, each stripe contains a parity block, which is the XOR of all of the other data blocks in the same stripe. If any one disk fails, the array is still operational, since the failed data can be restored by XORing data from all of the other disks. The price paid for this reliability is that each write operation is required to update the parity block(s) as well as the data block(s). This creates additional workload to the array. If a write request desires to update a partial stripe of data, which we refer to as a "partial stripe write", the new parity block for that stripe is calculated by

$$new\_parity = new\_data \oplus old\_data \oplus old\_parity. \tag{4.1}$$

where $X \oplus Y$ corresponds to the XOR of X and Y[4]. Thus, prior to updating the new data and parity, the old data and parity have to be read out first in order to calculate

[4]Another way to calculate the new parity is to read out those blocks not being updated in a stripe and XOR them with the new data blocks. This approach is beneficial only when most of the requests access more than half disks in a stripe, since it creates extra workloads to those disks not being updated. Our simulation results show that when the request size (in term of number of blocks) is uniformly distributed within a stripe, this approach performs worse than reading from these disks to be updated.

the new parity. This is referred to as the additional "read-update procedure" for a write request. On the other hand, if a request updates a complete stripe of data, which we call a "full stripe write", the new parity block for that stripe can be calculated by XORing all of the new data blocks. Thus it is not necessary in this case to read the old data and old parity blocks. In order to avoid the updating the parity blocks from becoming a bottleneck, parity blocks are rotated among the N disks in RAID 5.

Notice that for a *partial stripe write*, the parity update operation cannot proceed before the corresponding data block(s) has (have) been read out, since the calculation of a new parity block is based on the old data information (see Equation (4.1)). We call this the *write synchronization problem*, which will be addressed further later in Section 4.2.1.

One of the advantages of RAID 5 is its inherent load balancing property. Since all data files are spread over multiple disks, each disk faces the same statistical workload.

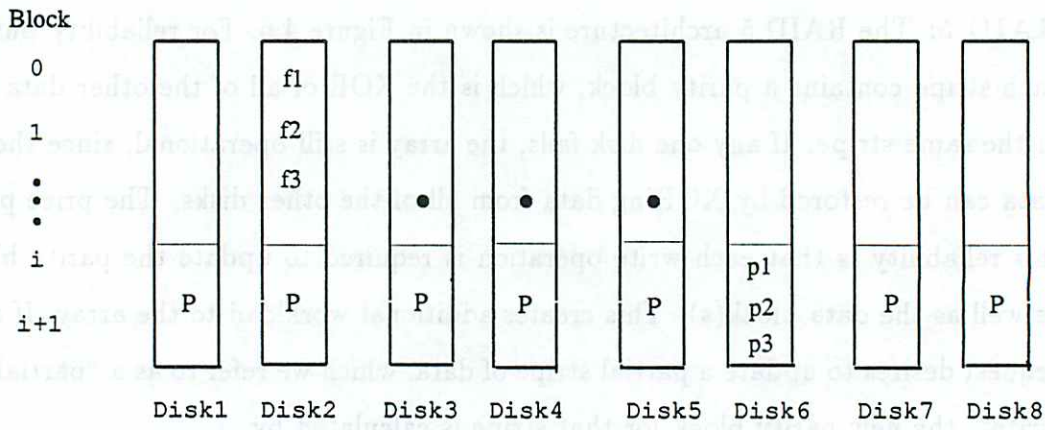


Figure 4.7 The Parity Striping Architecture ($N = 8$).

Parity Striping: Gray *et al.* [44] argued that RAID 5 may suffer a performance loss in small I/O environments (such as transaction processing systems), because multiple seeks and rotations are needed to serve a multiple block request. Instead, they recommended the *Parity Striping* architecture, which can be considered a variation of RAID 5 (Figure 4.7). Parity Striping only stripes parity blocks across the disks,

but not data[5]. In addition, parity blocks may occupy a contiguous area on each disk, called parity area. For instance, in Figure 4.7, the parity block of f_1 is p_1 and is stored in the parity area of disk 6. By using this architecture, a read request can usually be satisfied by one disk. For most transaction systems, since a request size is typically small (less than 10K bytes) compared to the parity area on each disk (more than 10^5K bytes[6]), Gray *et al.* [44] claimed that 99.9% of the writes can be satisfied by accessing a data disk and a disk containing its parity. While uniform access to each disk is assumed in [44], we will show that the Parity Striping architecture may suffer when the access pattern is nonuniform (skewed). Last, the *write synchronization problem* also exists with the Parity Striping architecture.

In the above, we have described several disk array architectures of interest to us. In the rest of this chapter, we will focus on the performance study of these disk array architectures.

4.2 Disk Arrays for Transaction Processing and Workstation Environments

In this section, we study disk arrays for applications such as transaction processing and workstation environments. We first describe scheduling policies suitable for these applications. We then provide analytic models and present the performance results for these disk array architectures coupled with different policies. Since it has been observed in the literature that for applications such as transaction processing systems and workstation/engineering environments, I/O requests typically access a small amount of data [75, 44, 76, 88], we assume that each request accesses a single block of data (4096 bytes [85]) in most cases. We, however, do extend our discussions to multiple block accesses in some cases.

---

[5]In fact, as indicated in [44], when RAID 5 is configured with a very large striping unit, it will have the same throughput characteristics as Parity Striping. In this case, RAID 5 almost completely abandons its advantages of high transfer rate and load balancing. While Gray *et al.* used a striping unit of 1K bytes in [44] for RAID 5, in our study, we select a striping unit of 4K bytes [85, 97]. However, it is our interest to examine the performance impact of increasing striping unit and its associated negative effect on the load balancing property of RAID 5 in a future research.

[6]Note: the size of parity area on each disk depends on the capacity of each disk and the total number of disks in the array.

4.2.1 Scheduling Policies

Policies for RAID 1 and its Variants

The scheduling policies we studied in the previous chapter for a mirrored disk apply for RAID 1. Recall that the best policy is DP-MR, which sends a read to the two disks and, whenever the first completes, aborts the other one. This policy, however, requires hardware support for aborting a request in service. The centralized policies also exhibit good performance on a mirrored disk but they are not appropriate for some disk array architectures. For example, consider the *group-rotate declustering* architecture (Figure 4.4), in which for each disk, the copies of all of its blocks are spread over $N/2$ disks. Thus $N/2$ central queues are required in this case. Summing over all disks in a group yields the requirement of $N^2/4$ central queues. Hence it is impractical to implement such a policy when the number of disks N is large.

The shortest queue policy DP-SQ-MS has been observed also to yield good performance on a mirrored disk. This policy is also easy to apply to disk arrays and is the one we will study in this section. Specifically, one queue is maintained in front of each disk. When a read arrive to the subsystem, the controller routes it to the shortest queue among the two target disks.

For comparison purposes, we will also study the DP-RJ policy, which randomly sends a read to one of the two target disks.

Two Synchronized Scheduling Policies for RAID 5 and Parity Striping

We now propose two synchronized I/O scheduling policies for RAID 5 and Parity Striping, which account for the *write synchronization problem* described in Section 4.1.2. In all cases, a write request completes only when both the data and the parity have been updated.

- **The Before Service (BS) Policy**

  Under this policy, two queues are maintained for each disk in the array, one for arriving read and write data requests (*D queue*) and the other for parity update requests (*P queue*), as illustrated in Figure 4.8. When a read or write request arrives at the disk subsystem, the dispatcher sends it to the D queue of the target disk. When it reaches the head of the queue and is scheduled for service, if it is

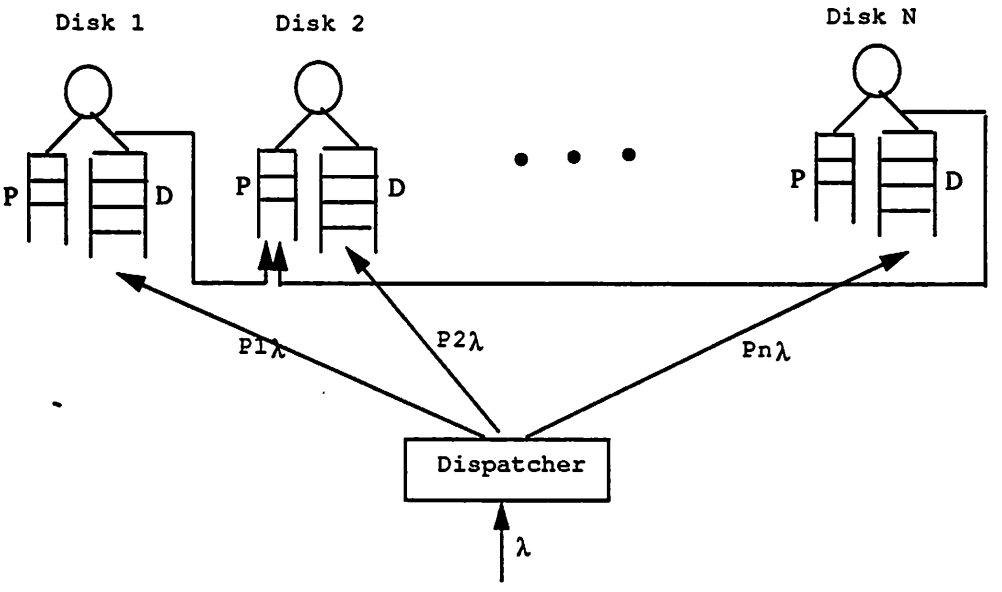


Figure 4.8 A Queueing Model for the PS Policy.

a write, it generates a parity update request to the P queue of the corresponding disk containing its parity block(s). Since the parity request is generated prior to the beginning of service, we call it the *before service* policy. The requests in the P queue are given higher priority than requests in the D queue to ensure that an outstanding parity request begins service as soon as possible since its corresponding data update operation has already started. The disk operation is assumed to be non-preemptive, therefore a new parity request cannot commence service until the current I/O finishes.

Under the BS policy, if the disk containing the parity block(s) is idle, the parity update request can start service immediately. To serve a write request, both the data and parity update requests will experience the following 5 service stages: seek, rotation, read (for the calculation of the new parity), full rotation, and write. Hence, if the parity update request finishes the 4-th stage prior to the completion of the 3-rd stage by the data update request, then one or more additional full rotations are necessary. Figure 4.9 shows the scenario for serving a write request. However, the probability that the sum of the seek (S') and rotational delay (R') on the data disk exceeds the sum of queueing delay in the

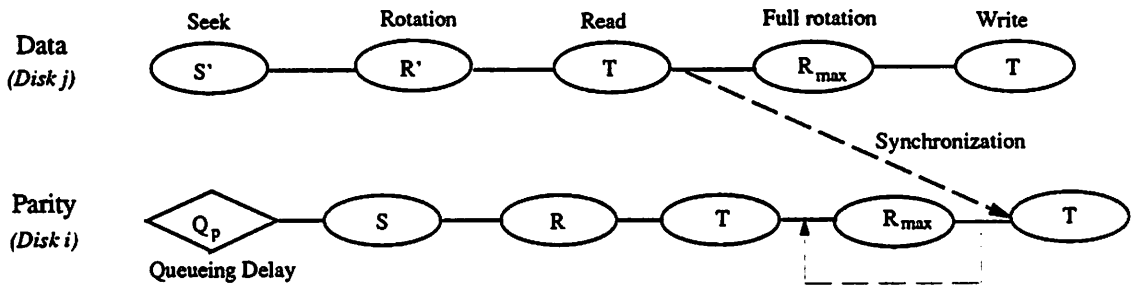


Figure 4.9 Scenario for Serving a Write Request.

P queue (Q_p), seek (S), rotation (R), and a full rotation (R_{max}) on the parity disk is very small (the transfer times (T) on both disks are the same) as we will observe later from the analysis.

- **The After Read (AR) Policy**

  The AR policy is the same as the BS policy except that the parity request is generated after the data required to calculate the new parity has been read out. This guarantees that the data required to calculate the new parity is available whenever needed, and therefore the loop in Figure 4.9 can never occur.

Remark: The performance of the AR policy is expected to be worse than that of the BS policy since it delays the generation of parity update requests. However, it may be useful in a network environment or a distributed file system in which the disk containing the parity block is located on another site, since it may reduce the network traffic and simplify the file system.

4.2.2 Variations of the RAID 1 Architecture

In this subsection, we study and compare RAID 1 and its variants, *mirrored declustering, chained declustering*, and *group-rotate declustering*, coupled with the DP-RJ and DP-SQ-MS policies. In the following, we present analytical models for the three architectures coupled with the DP-RJ policy. For DP-SQ-MS, we are only able to provide analytic models for RAID 1 (mirrored declustering). Therefore we

will resort to simulation in examining the performance of RAID 1 and its variants coupled with DP-SQ-MS.

Analysis

In our analytic models, data is assumed to be block interleaved with a block size of 4K bytes [85]. All of these variations are assumed to use the same random join scheduling policy DP-RJ. I/O requests arrive to the disk array according to a Poisson process with rate λ. After arrival, a write request is sent to the two disks containing its target data, and a read is randomly directed to one of them. Since data is striped over multiple disks, we assume that each disk faces the same workload in a statistical sense. As in the previous chapter, in order to make the analysis tractable, we assume that the disk service time is exponentially distributed. These analytic models will be validated by simulation results later in Section 4.4.

Consider the two disks on which a request may find its target data. We model the two disks as a two server fork/join queueing system, in which fork/join customers correspond to write requests that perform update operations on the two disks, and regular customers correspond to read requests and those write requests that have only one target data copy stored on one of these two disks. In the following, we discuss the arrival rates of fork/join and regular customers for the three architectures.

Mirrored Declustering: In mirrored declustering, the two disks constitute a mirrored pair. Since we assume that I/O requests are routed to each mirrored pair with the same probability, each pair sees an arrival rate of $\lambda' = \lambda/(N/2)$. The arrival rate for fork/join customers is $\lambda' p_w$, and for regular customers is $0.5\lambda' p_r$ per disk.

Chained Declustering: In this architecture, each disk is divided into two parts, one for the first copy, and the other for the second copy. Each I/O request can always find two adjacent disks containing its target data[7]. Consider the two disks, say D_i and D_{i+1}. Since there is only a fraction η (0.5 in this case) of their capacity being mirrored, the arrival rate for fork/join customers is $\eta\lambda' p_w$, and for regular customers is $0.5\lambda' p_r + (1-\eta)\lambda' p_w$ per disk.

[7]The two ways of data layout for chained declustering have no effect on the performance when each request only accesses one block of data.

In this way, the mirrored declustering is only a special case with $\eta = 1$.

Group-Rotate Declustering: Notice that with this architecture, there are two groups of disks for keeping the two data copies. If we select one disk from each group, then the mirrored fraction on the two disks is $2/N$ of the disk capacity. Hence, the same arrival rate formulas for chained declustering above can be used here by setting $\eta = 2/N$.

Finally, the two server fork/join queueing system can be solved in the same way as described in Section 3.2.1 and Appendix A.

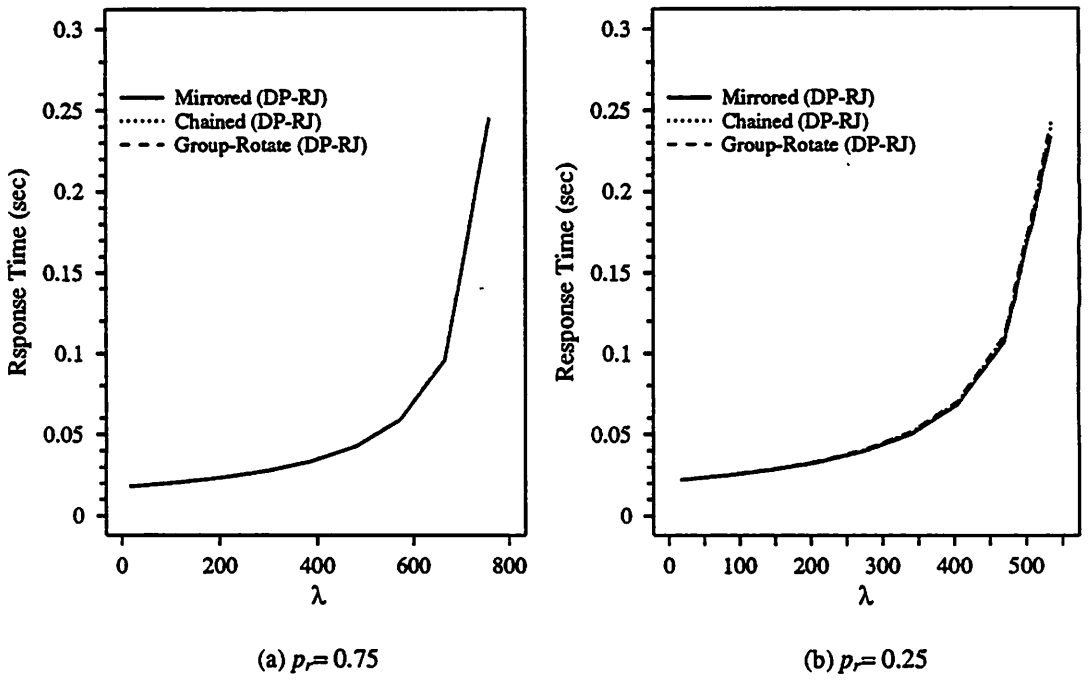
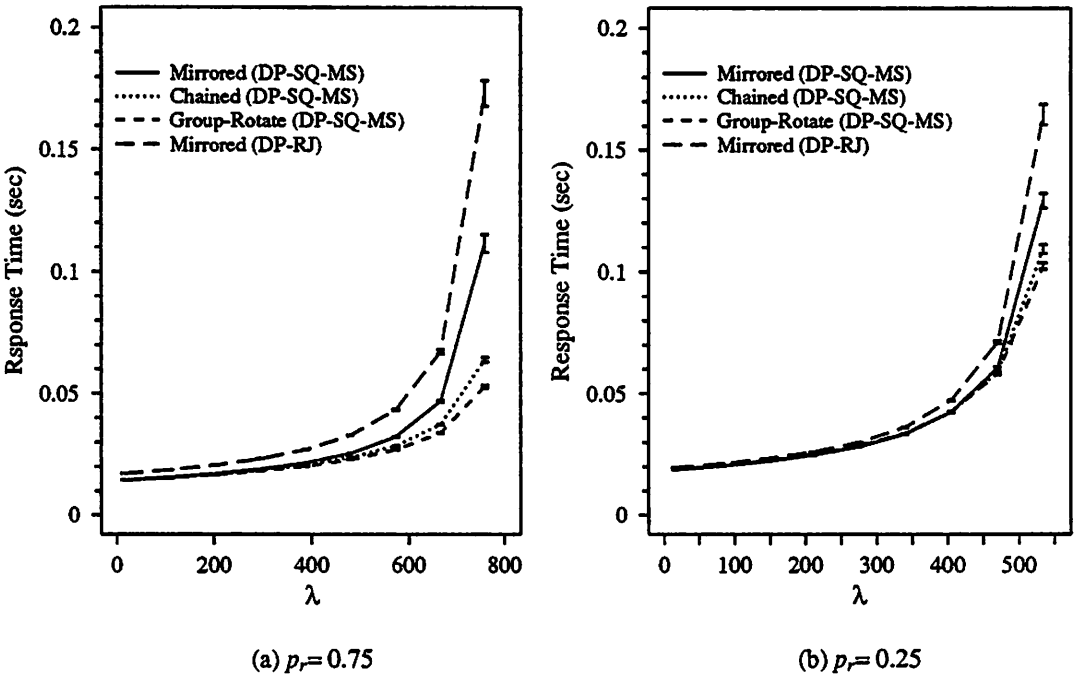


Figure 4.10 RAID 1 and Its Variants Coupled with DP-RJ in Small I/O Environments.

Performance Results

The performance of RAID 1 and its variants coupled with the DP-RJ policy is illustrated in Figure 4.10 for an array of 16 disks. As expected, in small I/O environments where each request accesses one block of data, the three architectures basically perform the same. There is only a slight difference when the disk array is highly loaded and most of the requests are writes, with mirrored declustering providing

slightly better performance than the other two architectures. This observation is also supported by our simulation results. Therefore, we conclude that the three variants under the DP-RJ policy are equivalent in such "small" I/O environments.



(a) p_r= 0.75 (b) p_r= 0.25

Figure 4.11 RAID 1 and Its Variants Coupled with DP-SQ-MS in Small I/O Environments.

When the shortest queue policy DP-SQ-MS is used, however, the three variants exhibit significantly different performance, especially at high loads. The results shown in Figure 4.11 are obtained from simulation experiments. These results are obtained by averaging over 40 runs, 50,000 requests executed in each run. 95% confidence intervals are illustrated in the Figure. We see that group-rotate declustering is the best, and mirrored declustering is the worst. The reason for this is that under DP-SQ-MS, mirrored declustering only provides load balancing of reads between two queues, since each mirrored pair is basically independent of other pairs. In the group-rotate architecture, data on each disk is replicated over $N/2$ disks. Therefore, the shortest queue policy has the ability to balance load among $N/2$ queues. The chained declustering falls in between these two architectures. The performance of mirrored

declustering coupled with DP-RJ is presented as a baseline for comparison.

Based on the above observations, we suggest a variant of chained declustering which has the potential of improving performance further. This variant, instead of placing the second copy of disk D_i on disk D_{i+1}, rotates the second copy of D_i among the disks other than D_i in a similar way as the group-rotate declustering architecture. For example, consider disk 1 shown in Figure 4.2, on which F_1 and F_9 are allocated. Under this new variant, while f_1 remains on disk 2, f_9 is now placed on disk 3. This variant is expected to achieve a better load balance when the DP-SQ-MS policy is employed.

4.2.3 RAID 5 vs. Parity Striping

The focus of this subsection is on the study of RAID 5 and Parity Striping for small I/O environments. Our work differs from previous work [75, 44] by explicitly modeling *write synchronization*. In particular, we will study the two synchronized I/O scheduling policies described in Section 4.2.1 for both RAID 5 and Parity Striping. We provide accurate mathematical models for estimating the mean I/O response times and the maximum throughput of RAID 5 and Parity Striping coupled with these two synchronized scheduling policies. Using these models, we compare the performance of RAID 5 and Parity Striping with each other.

Analytic Models

We again consider a disk array of N disks. The disk model has been described in Section 2.3. Based on that model, the disk service time for a read request is (Equation (2.7))

$$Y_r = X + T \tag{4.2}$$

and for a write request is

$$Y_w = X + T + R_{max} + T \tag{4.3}$$

where X is the sum of seek and rotational latency, T is the transfer time, and R_{max} is the full rotation time.

In the following, we only describe the basic idea for the analysis of single block accesses. Since the only difference between the AR and BS policies is the time at

which parity update requests are generated, we conclude that the analytic model for the BS policy is also applicable to the AR policy, except for a slight difference in the calculation of mean write response time. Interested readers are referred to [27] for the analysis of multiple block accesses and the calculations of the maximum throughput in terms of requests per second.

For ease of reference, we introduce the following additional notation in our analysis:

λ: the arrival rate to the disk I/O subsystem;

λ_p, λ_d: the arrival rates for the high priority parity update requests and the low priority read, write requests to a disk, respectively;

Q_p, Q_d: *r.v.*'s for queueing times of high and low priority requests;

Y_p, Y_d: *r.v.*'s for service times of high and low priority requests;

In addition, when it is necessary to distinguish between the disks, we will use the notation $Q_p(i), Q_d(i), \cdots$, and $Z(i)$ for the queueing delays, service times, and response times on disk i, $i = 1, 2, \cdots, N$. This is convenient when we consider the case that the access patterns are nonuniform.

A Queueing Analysis of RAID 5: In our model, I/O requests are assumed to arrive to the disk array according to a Poisson process with rate λ. As shown in Figure 4.8, two queues are maintained in front of each disk, one for read and write data requests (D queue), and the other for parity requests (P queue). Since in this study RAID 5 uses a small striping unit (4K bytes) and spreads data across multiple disks, it is reasonable to assume that requests go to each disk with same probability, i.e., $p_i = 1/N$, $i = 1, 2, \cdots, N$. Therefore the arrivals to the D queue of disk i is described by a Poisson process with parameter $\lambda_d = \lambda/N$, where a request is a read with probability p_r and a write with probability $p_w = 1 - p_r$. The arrivals to the P queue of disk i is a superposition of $N - 1$ parity request generating processes which direct requests to disk i with probability $1/(N-1)$. When N is large, it can be approximated by a Poisson process with parameter $\lambda_p = p_w\lambda/N$ [57]. Strictly

speaking, the $N-1$ parity generating processes are not independent of each other. However, when N is large, the correlations between these processes become small, and therefore we conjecture that they become independent in the limit as $N \rightarrow \infty$. In fact, when $N = 16$, our simulation results yield a close match to those of analysis. Last, the FCFS discipline is used within both the D and P queues.

Thus, disk i ($i = 1, \cdots, N$) can be modeled as a two priority class non-preemptive priority queue, where the service time for low priority customers (read and write requests in the D queue) is obtained by Equations (4.2) and (4.3). Suppose a write request is directed to disk j and its corresponding parity block is located on disk i. The service time of a high priority customer (parity update request in the P queue) depends on the service time of its corresponding data update request on disk j and the time that it waits in the P queue of disk i. Specifically, let X and X' be the sum of seek and rotation times on disks i and j (which have the same distribution), and Q_p be the queueing time of a parity update request in queue P of disk i, then the service time for this parity update request[8] is

$$Y_p = \begin{cases} X + 2\tau + R_{max}, & X' < Q_p + X + R_{max}, \\ X + 2\tau + 2R_{max}, & X' \geq Q_p + X + R_{max}. \end{cases} \tag{4.4}$$

As shown in Figure 4.9, the first term on the right hand side corresponds to the case where the old data required to calculate the new parity block has become available prior to disk i becoming ready to update the parity block. The second term corresponds to the opposite situation, i.e., the old data has not been read out by disk j by the time disk i completes a seek, a rotation, a read of the old parity, and a full rotation. Therefore, one more full rotation is needed before the old data becomes available.

Since the service time for high priority requests, Y_p, depends on the queueing time Q_p, there is no easy way to solve this model. As an alternative, we develop tight bounds on the service time Y_p and then obtain estimates of the mean I/O response times based on these bounds.

[8]Note: since the probability $P\{X' \geq Q_p + X + 2R_{max}\}$ is extremely small (less than 0.00015, as obtained from a simulator by taking $Q_p = 0$), we ignore the cases where two or more additional full rotations are required in serving a parity update request in order to achieve the write synchronization.

The mean read response time is given by

$$\overline{Z_r} = \overline{Q_d} + \overline{X} + \alpha \tag{4.5}$$

where $\overline{Q_d}$ is the mean queueing delay at D queue and $\overline{X} = \overline{S} + \overline{R}$ is the mean for the sum of seek and rotational latency and can be derived directly from Equations (2.2) and (2.4).

From Figure 4.8 and the scenarios shown in Figure 4.9, it is clear that the mean write response time, $\overline{Z_w}$, is calculated by

$$\overline{Z_w} = \overline{Q_d} + E\left[\max\{X', Q_p + X\}\right] + 2\alpha + R_{max}. \tag{4.6}$$

Notice that the Q_d above is the queueing time that a write request waits in the D queue of disk j which contains its target data, whereas the Q_p is the queueing time that its corresponding parity request waits in the P queue of disk i which contains its parity block. Since all of the disks are statistically identical, we use the notation Q_d and Q_p instead of $Q_d(j)$ and $Q_p(i)$ in (4.6).

Finally, the mean I/O response time is

$$\overline{Z} = p_r \overline{Z_r} + p_w \overline{Z_w}$$

The detailed derivations can be found in Appendix D. We have also run simulations as an intermediate validation of our analytic models. As a consequence, the simulation results fall between the results obtained from lower and upper bound on service time Y_p for all workloads, but are very close to the lower bound results. This suggests that the model based on the lower bound of Y_p is a very good approximation to the true system. The simulation results can be found in Tables D.1 and D.2 in Appendix D.

An Analysis of Parity Striping: For a single block access, if each I/O request is assumed to be directed to any disk with equal probability, then there is no difference between the RAID 5 and the Parity Striping architectures. However, if the I/O access pattern is skewed, i.e., some files are "hotspots", then the Parity Striping architecture will suffer a performance degradation, since files are allocated on a single disk. Under skewed accesses, a large fraction of I/O requests is directed to a small fraction of disks.

Hence, these disks are likely to become a bottleneck which will limit the throughput of the I/O subsystem.

In order to examine the effect of skewed accesses, we use the "s-th degree of skew" model which is a common mathematical way to characterize skewing [83][9]. In this model, if the least busy disk has an arrival rate of λ, then the k-th least busy disk has an arrival rate of $k^s\lambda$. A 0-degree skew produces a uniform distribution. If the skewed access is of s-degree, and disk i is the i-th least busy device in the array, then a request goes to disk i with probability

$$p_i = \frac{i^s}{\sum_{j=1}^{N} j^s}, \qquad i = 1, 2, \cdots, N \tag{4.7}$$

and the arrival rates of data and parity requests to disk i are

$$\begin{aligned} \lambda_d(i) &= p_i\lambda, \\ \lambda_p(i) &= \frac{p_w(1-p_i)\lambda}{N-1}, \qquad i = 1, 2, \cdots, N. \end{aligned} \tag{4.8}$$

Here, we assume that for any write request to disk i, its corresponding parity block resides on any of the other $N-1$ disks with the equal probability, i.e., its parity update request goes to disk j with probability $1/(N-1)$, $j \neq i$.

Using Equation (4.8) and then following the same steps as in Appendix D yield the mean queueing delay $\overline{Q_d(i)}$. The mean read response time on disk i, $\overline{Z_r(i)}$, can be obtained immediately from Equation (4.5). In a similar way, by using $\lambda_d(j)$ and $\lambda_p(j)$ we can calculate the mean $E[\max\{X', Q_p(j) + X\}]$. Thus the mean write response time on disk i, $\overline{Z_w(i)}$, is

$$\overline{Z_w(i)} = \overline{Q_d(i)} + \frac{1}{N-1} \sum_{j=1; j\neq i}^{N} E[\max\{X', Q_p(j) + X\}] + 2\alpha + R_{max}$$

Finally, the mean response times are averaged over all disks

$$\begin{aligned} \overline{Z_r} &= \sum_{i=1}^{N} p_i \overline{Z_r(i)}, \\ \overline{Z_w} &= \sum_{i=1}^{N} p_i \overline{Z_w(i)}, \end{aligned}$$

---

[9] Another typical model for skewed accesses is the $\alpha - \beta$ model, where a fraction α of workloads goes to a fraction β of disks. We choose the "s-degree of skew" model because there is only one parameter, whereas the $\alpha - \beta$ model requires studying the effects of varying two parameters.

$$\overline{Z} = p_r \overline{Z_r} + p_w \overline{Z_w}. \quad (4.9)$$

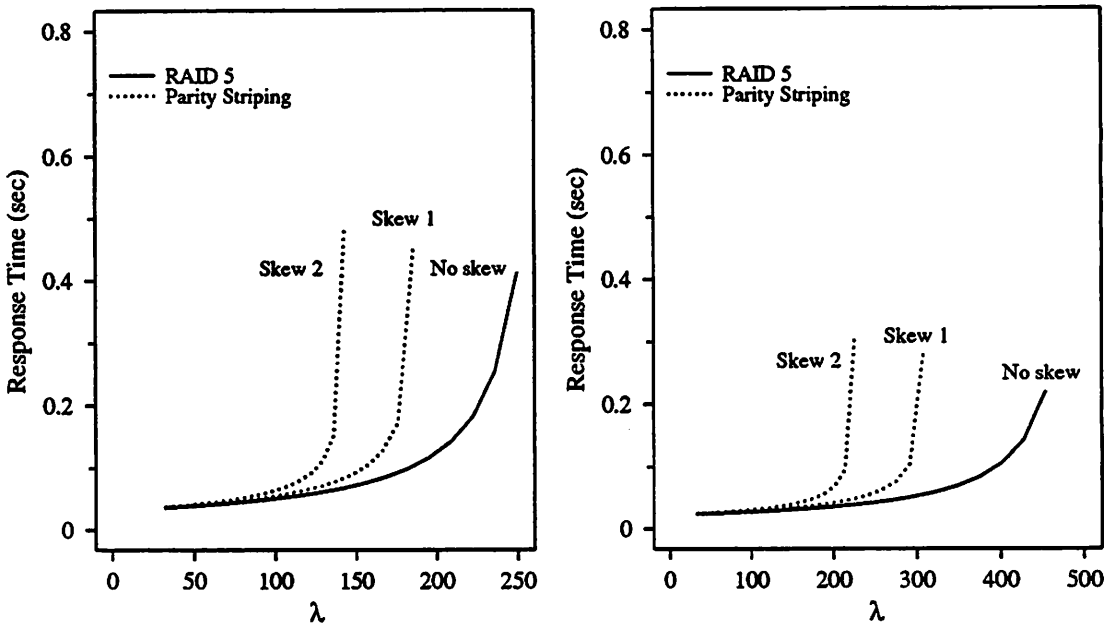
Performance Results

In this section, we compare the performance of the RAID 5 and Parity Striping architectures by using the analytic models developed above. In doing so, we will focus on the BS policy. Readers can easily generate results for the AR policy by using the same approach as shown in Appendix D. The number of disks, N, in the array is 16. Two kinds of workloads are reported here for the performance measurements, single block accesses and multiple block accesses. In the case of multiple block accesses, the request size is assumed to have either a quasi-geometric distribution or a uniform distribution. The response time for a multiple block request is the maximum delay until all the blocks have been read out or updated. In the case of writes, response times also account for the delay incurred by updating the parity block(s).

In addition, we also report the performance degradation of the Parity Striping architecture under skewed accesses. Finally, the maximum throughput supported by the RAID 5 and the Parity Striping architectures are examined. In all of our experiments, the sequential access probability, p_s, is set to 0.3.

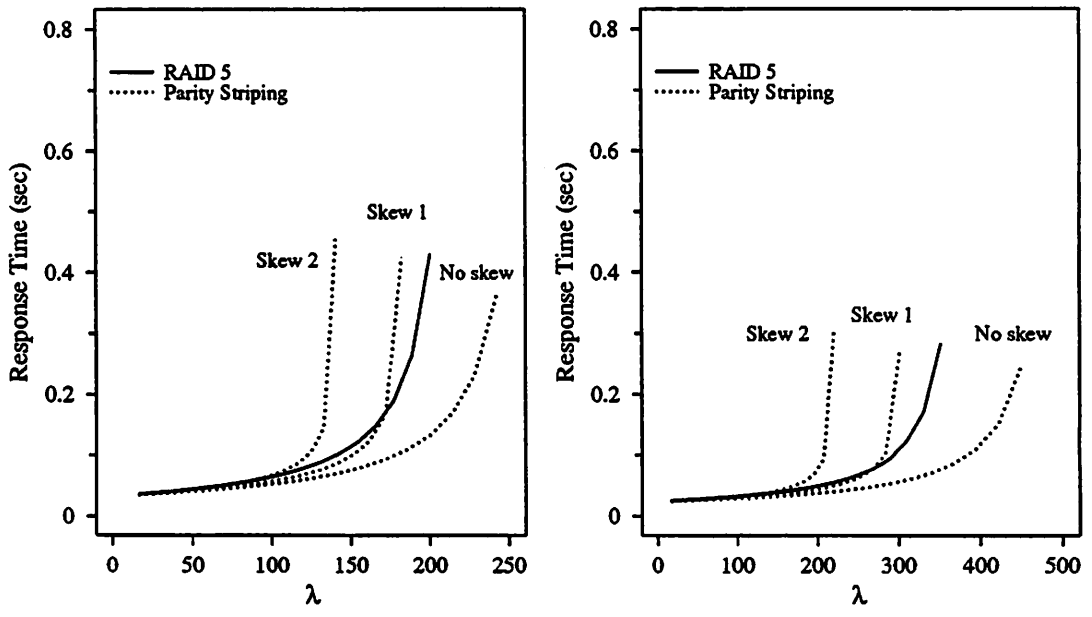
Single Block Access: Figure 4.12 plots the mean I/O response time as a function of the request arrival rate λ, where each request accesses one block of data (4K bytes). As discussed in the last paragraph, when each request is directed to any disk with the same probability, RAID 5 and Parity Striping are the same and therefore have the same mean I/O response times. However, when the access pattern is skewed, even for the first degree, the performance of Parity Striping degrades quickly and exhibits worse performance than RAID 5.

Multiple Block Accesses: From previous discussions, we learned that, by using a small block size (the unit of interleaving), RAID 5 can fairly tolerate skewed accesses. However, it suffers from multiple block accesses as the mean number of blocks in a request increases. This is because multiple disks are required to serve a request, each paying a seek and rotational latency. In contrast, under Parity Striping, only one disk is needed to serve a read, and typically two disks are needed to serve a write.



(a) Mean I/O Response Time.(p_r=0.25) (b) Mean I/O Response Time.(p_r=0.75)

Figure 4.12 Performance of RAID 5 and Parity Striping with Single Block Accesses.



(a) Mean I/O Response Time. (p_r=0.25) (b) Mean I/O Response Time. (p_r=0.75)

Figure 4.13 Multiple Block Accesses with Request Size Quasi-Geometrically Distributed ($\sigma = p = 0.7$).

Hence, the disk utilization under RAID 5 is higher than that under Parity Striping, which may results in an early saturation of RAID 5. On the other hand, since data is not striped under Parity Striping, if some files are hotspots, then disks containing these files will receive more requests than other disks. The experiments here and in the next paragraph are designed to study the sensitivity of RAID 5 to the increase of request size, and of Parity Striping to the skewed accesses.

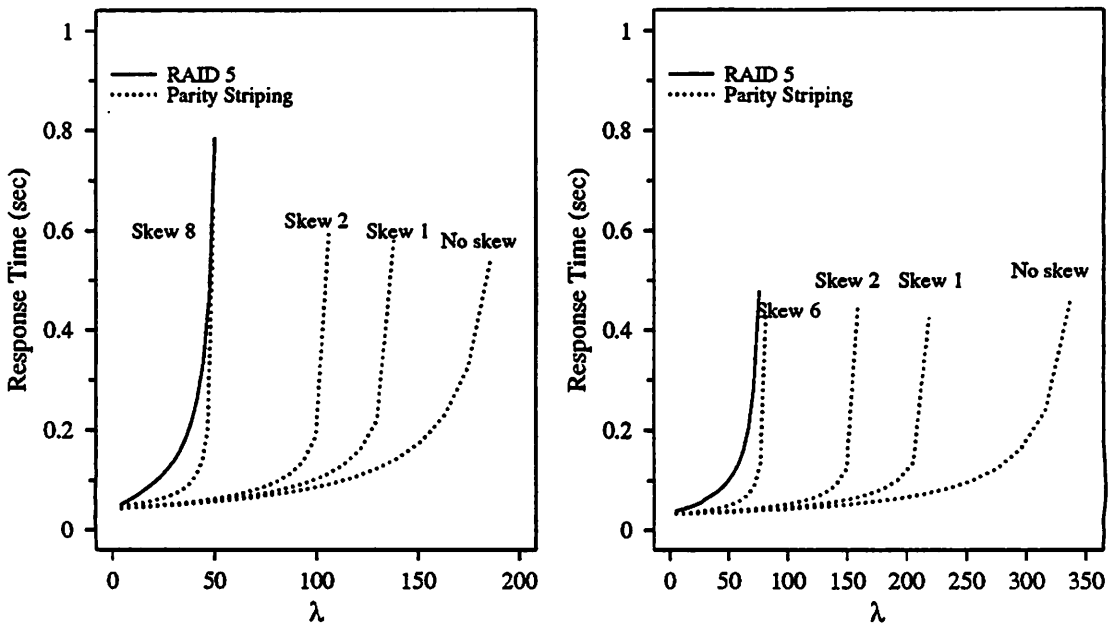
Since it has been observed that in most database systems I/O requests predominately access only one block of data (4KB) [83, 75], we first assume that the request size B has the following quasi-geometric distribution,

$$P\{B = i\} = \begin{cases} \sigma, & i = 1, \\ (1-\sigma)\frac{p(1-p)^{i-1}}{(1-p)-(1-p)^{N-1}}, & i = 2, \cdots, N-1, \end{cases}$$

where σ is the probability that a request accesses a single block of data, and p is a parameter for the geometric distribution.

Figure 4.13 illustrates the situation where 70% of requests access one block of data ($\sigma = p = 0.7$). In this case, the mean request size is about 1.5 blocks. The results show that Parity Striping outperforms RAID 5 when there is no access skew, but is outperformed by RAID 5 even for the accesses of 1-degree of skew, which is likely to occur in most applications. In another experiment where the request size follows a uniform distribution with a mean of 8 blocks, RAID 5 is observed much worse than Parity Striping, even with extremely skewed accesses (Figure 4.14).

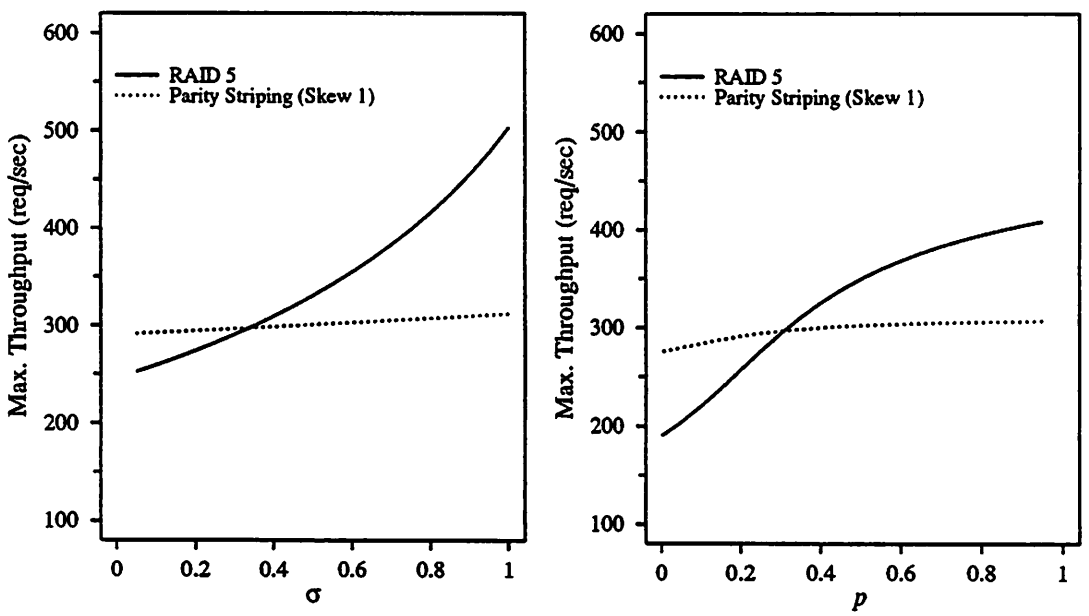
The Maximum Throughput: Here, we examine the maximum throughput supported by RAID 5 and Parity Striping in terms of the number of I/O requests per second. As discussed earlier, the performance of RAID 5 degrades quickly as the mean request size increases. On the other hand, the Parity Striping architecture is sensitive to the skewed access pattern. Figure 4.15 explores the sensitivity of the two architectures to the I/O request size. Here the request size is assumed to be quasi-geometrically distributed. In Figure 4.15 (a), we fix the parameter $p = 0.7$ and vary the single block probability σ, and in (b), we fix $\sigma = 0.7$ and vary p. Only the case $p_r = 0.75$ is plotted. The same behavior has been observed for the case $p_r = 0.25$. Figure 4.16 illustrates the sensitivity of Parity Striping to the skew in access patterns.



(a) Mean I/O Response Time.(p_r= 0.25)

(b) Mean I/O Response Time.(p_r=0.75)

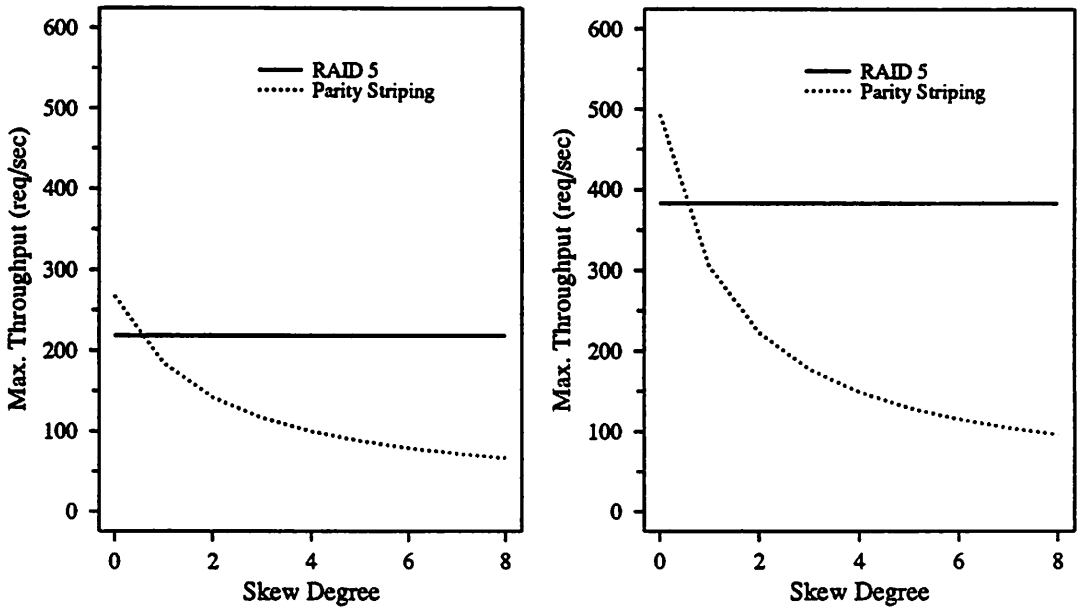
Figure 4.14 Multiple Block Accesses with Request Size Uniform in $[1, N-1]$.



(a) Effect of Varying Req. Size.
(p= 0.7)

(b) Effect of Varying Req. Size.
(σ=0.7)

Figure 4.15 Sensitivity of RAID 5 and Parity Striping to the Request Size (p_r = 0.75, Request Size: Quasi-Geometrically Distributed).



(a) Effect of Skewing on Parity Striping. (p_r=0.25)

(b) Effect of Skewing on Parity Striping. (p_r=0.75)

Figure 4.16 Sensitivity of Parity Striping to the Skew in Access Patterns (Request Size: Quasi-Geometrically Distributed, $\sigma = p = 0.7$).

From the above discussions, it is clear that Parity Striping is advantageous only for a certain range of parameter values. Figure 4.17 shows the parameter ranges where one architecture may provide a higher throughput than the other. For example, if an application has a mean request size of 2 blocks and has a skew degree of one, then Figure 4.17 suggests that Parity Striping is better than RAID 5. On the other hand, if the access pattern has a skew degree of one and a mean request size less than 1.7, then RAID 5 may support a higher throughput than Parity Striping. Again, the request size is assumed to be quasi-geometrically distributed; the parameter p is fixed to 0.7; and the various values for the mean request size are obtained by varying the single block probability, σ.

4.2.4 Comparison of RAID 1 and RAID 5

The most significant advantage of RAID 5 and Parity Striping over RAID 1 is the lower cost, since RAID 1 requires nearly double the number of disks in order to achieve the same capacity, or by using the same number of disks, RAID 1 only

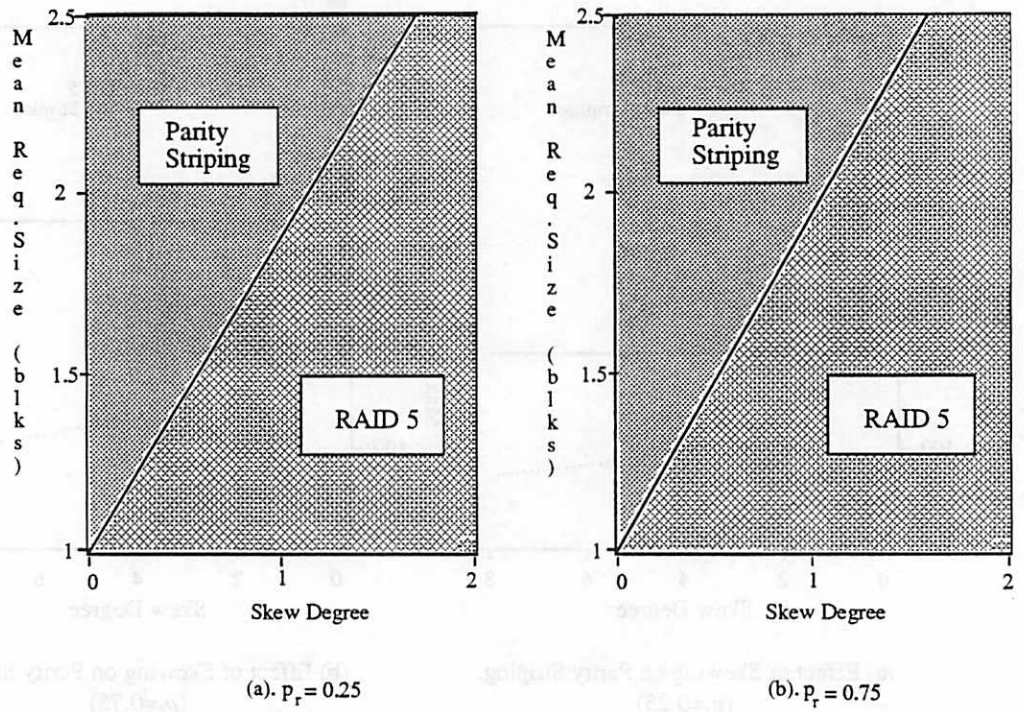


Figure 4.17 Favored Areas for the RAID 5 and Parity Striping Architectures (Request Size: Quasi-Geometric with $p = 0.7$).

achieves half capacity as that of RAID 5. Given the high cost paid by RAID 1, in this subsection, we are interested in the performance gain achievable by RAID 1 over RAID 5.

For RAID 1, we will use the DP-SQ-MS policy and the model as in Appendix C. Although we have developed a quite accurate model for RAID 5 which uses real disk seek and rotational latency distributions in the last subsection, here we will use a simple model which assumes the basic disk service time to be an exponential random variable. This is because RAID 1 has been modeled under the assumption of exponential service times. In order to make the two architectures comparable, the same approximation of the service time distribution is necessary. Again, these models are validated via simulation experiments, which will be discussed later in Section 4.4.

A Simplified Model for RAID 5

As in Section 4.2.3, we assume that I/O requests arrive to the disk subsystem according to a Poisson process with rate λ. After its arrival, a request is directed to

the disk containing its target data block by the I/O subsystem dispatcher. As before, each disk in the array is assumed to be accessed with equal probability $1/N$, where N is the number of disks in the array. If the request is a write, then it is also directed to the disk containing the appropriate parity block in order to update the parity block.

We approximate the behavior of each disk as an $M/G/1$ queue and assume that a write request completes when the block has been written. This ignores the effect of the synchronization required to update the parity block, namely that the new parity block can not be written prior to the old value of the data block being read out. Therefore, this model should provide an optimistic estimate for the RAID 5 architecture in a small I/O environment when $p_w > 0$.

As stated above, the service time for a read is assumed to be exponentially distributed with mean $1/\mu$, which is obtained directly from Equation (2.8).

The service time for a write request consists of a seek, rotational latency, block read followed by a full rotation, and the time required to write the block back to the disk. Consequently, the service time for a write request can be expressed as,

$$Y_w = Y_r + \Theta$$

where Θ is a constant consisting of the full rotation time and the second transfer time (τ).

Because of the special *read-update* procedure required for write operations in the RAID 5 architecture, each write request will also bring extra workload to the disk on which the corresponding parity block is allocated. Let p_r be the probability that a request is a read and p_w be the probability that a request is a write. Then, each disk will face an arrival rate of $\lambda_i = \lambda(1 + p_w)/N$. These requests to update parity blocks behave in the same way as writes, i.e., they need to follow the *read-update* procedure. Therefore, the actual fractions of reads and writes are $\tilde{p}_r = p_r/(1 + p_w)$ and $\tilde{p}_w = 2p_w/(1 + p_w)$ respectively.

Let Y be the *r.v.* for disk service time; it has the following density function,

$$f_Y(x) = \begin{cases} \tilde{p}_r \mu e^{-\mu x} & x < \Theta, \\ \tilde{p}_r \mu e^{-\mu x} + \tilde{p}_w e^{\mu\Theta} \mu e^{-\mu x} & x \geq \Theta \end{cases}$$

and first and second moments,

$$\overline{Y} = \tilde{p}_r \frac{1}{\mu} + \tilde{p}_w (\frac{1}{\mu} + \Theta)$$

$$\overline{Y^2} = \frac{2}{\mu^2} + \tilde{p}_w(\frac{2\Theta}{\mu} + \Theta^2).$$

By the Pollaczek-Khinchin formula [61], the mean waiting time in the queue is

$$\overline{Q} = \frac{\lambda_i \overline{Y^2}}{2(1-\rho)}$$

where $\rho = \lambda_i \overline{Y}$ is the device utilization.

Finally, the mean I/O response time, $\overline{Z}$, is given by

$$\begin{aligned} \overline{Z_r} &= \overline{Q} + \frac{1}{\mu} \\ \overline{Z_w} &= \overline{Q} + \frac{1}{\mu} + \Theta \\ \overline{Z} &= \tilde{p}_r \overline{Z_r} + \tilde{p}_w \overline{Z_w} \end{aligned}$$

where $\overline{Z_r}$ and $\overline{Z_w}$ are the mean response times for read and write requests, respectively.

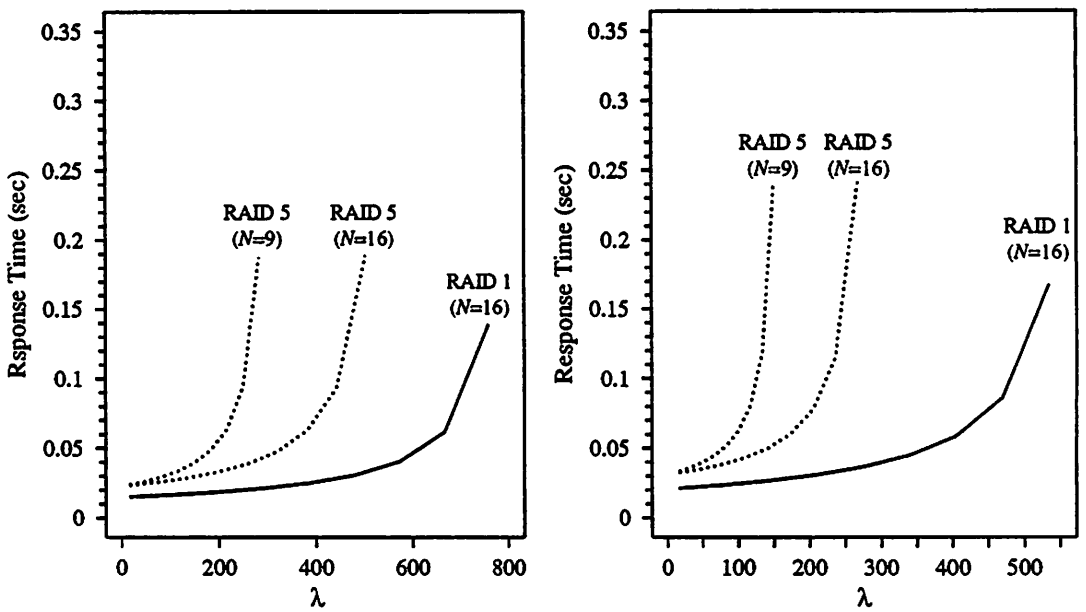


Figure 4.18 Compare RAID 1 and RAID 5 in Small I/O Environments.

Performance Comparisons

The performance comparisons are conducted in two different ways. First, we let both RAID 1 and RAID 5 have the same capacity, i.e., the number of disks in RAID 1

is $2n$, and in RAID 5 is $n+1$, where n is selected to be 8 in our experiment. Second, we compare the two architectures with the same number of disks, where RAID 1 loses half of its useful capacity. The results are shown in Figure 4.18 for the cases of $p_r = 0.75$ and $p_r = 0.25$. From the Figure, we observe that when both RAID 1 and RAID 5 have the same capacity, RAID 1 can support more than thrice the arrival rate of I/O requests as that of RAID 5. When both arrays contain the same number of disks, RAID 1 can still support more than double I/O's per second as compared to RAID 5. Therefore, we conclude that RAID 1 performs significantly better than RAID 5 in small I/O environments at the cost of doubling the number of disks or losing half capacity.

4.3 Disk Arrays for Supercomputing and Image Processing Systems

Applications such as supercomputing and image processing usually carry lower multiprogramming levels and, consequently, fewer I/O requests are issued per unit time. However, each I/O request typically accesses a large amount of data. Generally, computation parameters are moved in bulk from disks to memory resident data structures, and results are periodically written back to disks [55]. In this case, multiple disks work together as a single logical device providing a faster transfer rate. In this section, we assume that all of the disks that work together to serve a request are synchronized in terms of rotation and arm movement [59]. For example, for the mirrored declustering and group-rotate declustering architectures, disks are divided into two groups, one for each data copy. Thus each group is considered to be synchronized, but the two groups may work independently. For chained declustering and RAID 5, since all disks in the array may be involved in serving a request, they are assumed to be all synchronized. Some discussions on asynchronous RAID 1 and RAID 5 can be found in an our previous study [29]. Since Parity Striping is primarily proposed for transaction processing systems and is obviously not suitable for large data transfers, we exclude it from our discussions in this section.

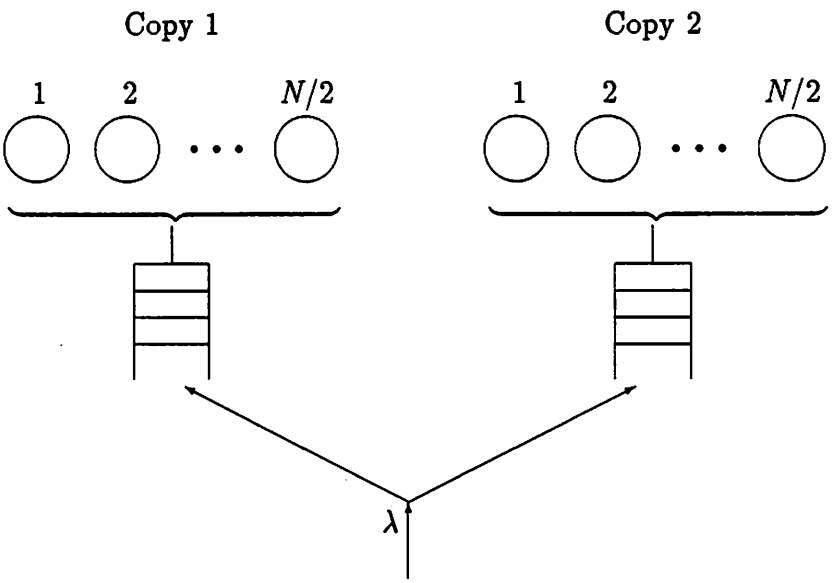


Figure 4.19 RAID 1 Queueing Model for Large I/O Environments.

4.3.1 Variations of the RAID 1 Architecture

Analysis

In the following, we assume that the I/O request size is a *r.v.* having an arbitrary distribution with a mean of $\overline{B}$ blocks.

Mirrored Declustering and Group-Rotate Declustering: As stated above, for these two architectures, the N disks are divided into two groups, each with $N/2$ disks. The queueing model is shown in Figure 4.19, where two queues are maintained, one for each group. Again, the DP-SQ-MS policy is adopted here. In large I/O environments, these two architectures are equivalent. Disks in each group are synchronized. When a request at head of a queue is scheduled for service, accesses at all $N/2$ disks are initiated simultaneously. Thus, the mean disk service time is

$$E[Y] = E[S] + E[R] + E[T] \tag{4.10}$$

where the mean seek time $E[S]$ and mean rotational latency $E[R]$ are obtained from Equations (2.3) and (2.5), and the mean transfer time $E[T] = \tau\overline{B}/(N/2)$, where τ is the transfer time for a single block. When a read arrives and finds all disks idle, it starts at the group requiring the least arm movement. Thus the mean seek time

is obtained from Equation (2.9). Therefore, the analytic model developed in Section 3.2.2 and Appendix C is also applicable here by using the mean disk service time $E[Y]$ from Equation (4.10).

Chained Declustering: As described in Section 4.1, there are two ways to layout data under chained declustering, *horizontal layout (HL)* and *vertical layout (VL)*. The *horizontal layout* is favored for write operations, since the corresponding data belonging to the same stripe in the two copies (e.g., F_1 and f_8 in Figure 4.2 and 4.3) are allocated at the same cylinder. Therefore a *large* write can update the two copies by moving the arm to the destination cylinder, updating the first copy, followed by a rotational delay and then updating the second copy. Hence, only one seek is needed. The mean disk service time for a read, $E[Y_r]$, is the same as in (4.10) with $E[T] = \tau\overline{B}/N$, and for a write is

$$E[Y_w] = E[Y_r] + E[R] + E[T].$$

While this can be modeled by a $M/G/1$ queue with an accurate disk service time distribution as described in Section 2.3, in order to make it qualitatively comparable to RAID 1 (for which we have made an exponential service time assumption), we assume the read service time to be an exponential *r.v.* with mean $1/\mu_r = E[Y_r]$ and the write service time to be an exponential *r.v.* with mean $1/\mu_w = E[Y_w]$. Then the chained declustering with horizontal data layout is modeled by a $M/G/1$ queue with disk service time distribution

$$f_Y(x) = p_r\mu_r e^{-\mu_r x} + p_w\mu_w e^{-\mu_w x}.$$

The $M/G/1$ queue is solved in a similar way as for the simplified model for RAID 5 in the last section.

The *vertical layout (VL)* strategy is designed to favor read requests, since the two copies of data are allocated on different cylinders with a distance of $C/2 - 1$ tracks, where C is the total number of cylinders on each device. Thus, a read can go to the copy closest to the current arm position. In any case, a read never seeks more than $C/2$ tracks. The mean service time for reads is

$$E[Y_r] = E[S'] + E[R] + E[T]$$

where the calculation for the mean seek time $E[S']$ is given in Appendix F,

While the VL strategy reduces the read service time, it increases the overhead for write operations. When a write operation starts, it always seeks to the copy closest to the current arm position, updates the copy, and then moves $C/2 - 1$ tracks to update the other copy. Hence, the mean service time for writes is

$$E[Y_w] = E[S'] + E[R] + E[T] + S_c + E[R] + E[T]$$

where S_c is the seek time of $C/2 - 1$ tracks.

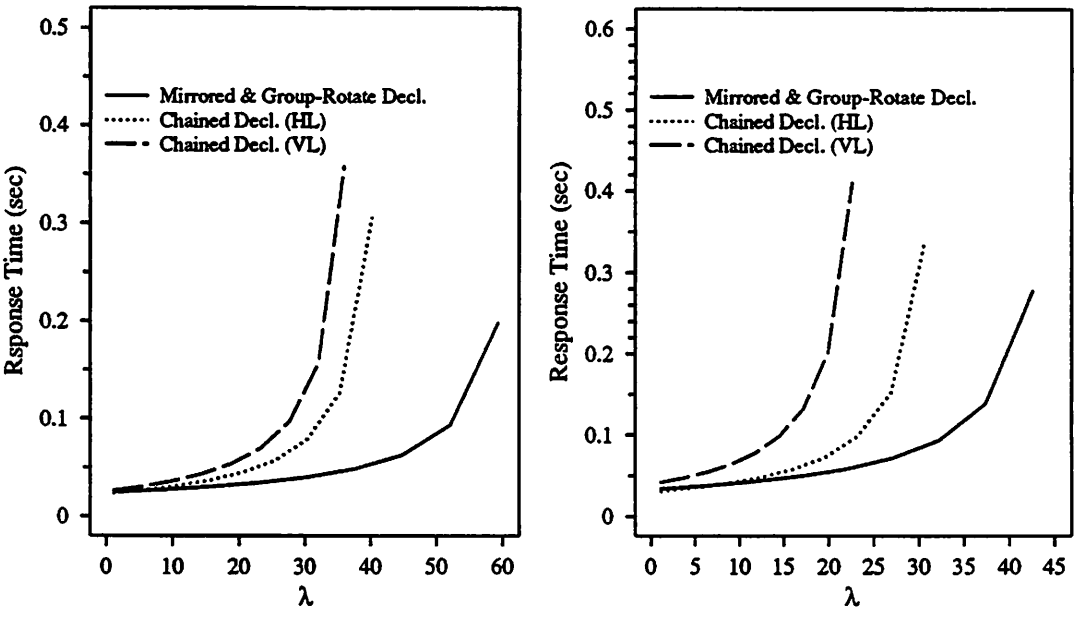
Consequently, the chained declustering with VL strategy is modeled by a $M/G/1$ queue with exponential service times for reads and writes in a similar way as the HL strategy.

Performance Results

The performance of the three variations of RAID 1 is reported here by using the models described above. The mean request size $\overline{B}$ is assumed to be 256 blocks (1M bytes). The number of disks in the array is assumed to be $N = 64$. If a workstation disk has a capacity of 500M bytes, a disk array consisting of 64 such disks can yield 32G bytes of raw capacity, which provides ample capacity to replace mainframe disks.

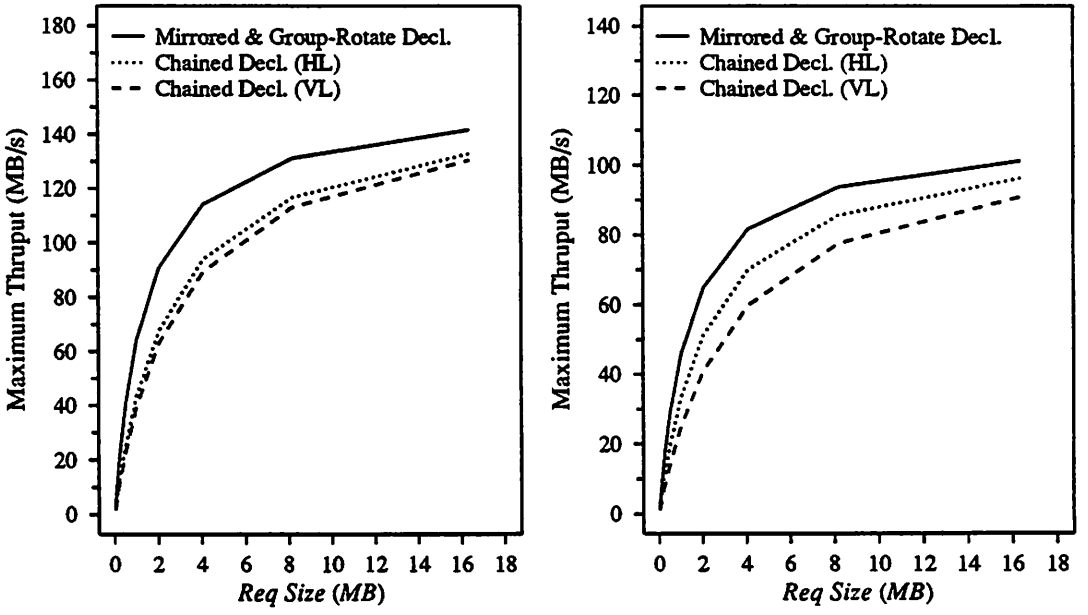
Figure 4.20 plots the mean I/O response time as a function of workload for the two cases of read probabilities $p_r = 0.75$ and $p_r = 0.25$. From the Figure, we observe that mirrored declustering and group-rotate declustering perform much better than chained declustering, especially when most of the requests are reads. This is because they can serve two read operations concurrently, whereas chained declustering can only serve one request at a time. Between the two data layout strategies of chained declustering, HL is observed to be better than VL because of the high cost for writes under VL. However, as read probability increases, VL is close to HL, and we have observed that when $p_r > 0.9$, VL outperforms HL.

In Figure 4.21 and 4.22, we illustrate the maximum throughput supported by the three architectures as a function of the mean request size and the number of disks in the array, respectively. Again, the mirrored declustering and group-rotate declustering provide higher throughput than chained declustering.



(a) p_r= 0.75 (b) p_r= 0.25

Figure 4.20 RAID 1 and Its Variants in Large I/O Environments.



(a) p_r= 0.75 (b) p_r= 0.25

Figure 4.21 Maximum Throughput of RAID 1 on Request Size.

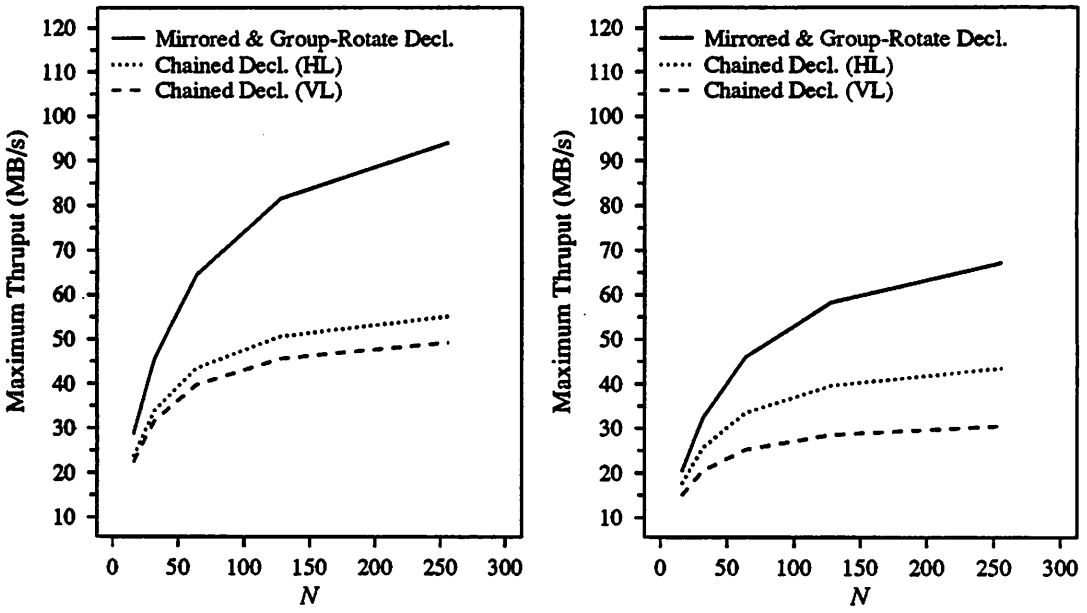


Figure 4.22 Maximum throughput of RAID 1 on Number of Disks.

4.3.2 RAID 1 vs. RAID 5 in Large I/O Environments

The RAID 1 model for large I/O's has been investigated in the last subsection. In the following, we briefly present an analytic model for RAID 5 and then compare it with RAID 1 for the cases of: (1) the two arrays have the same capacity; and (2) they include the same number of disks, as we did in the small I/O context.

A RAID 5 Model for Large I/O Environments

Since we have assumed that all disks in the array are synchronized, the mean service time for a read request, $1/\mu_r = E[Y_r]$, can be obtained immediately from Equation (2.8), where the mean transfer time $E[T] = \tau\overline{B}/(N-1)$.

For write requests, since a large I/O operation may involve several stripes of data, we distinguish between two cases. In the case of a *full stripe write*, where the data blocks to be updated start and end at stripe boundaries, the write service time is the same as that of a read because the parity block of each stripe can be calculated from the new data blocks. In this case, no extra read operation is needed. However, if either the starting block or the end block is not aligned with the stripe boundary,

a *partial stripe write* occurs and a *read-update* procedure has to be followed. For simplicity, we assume that a partial stripe write is only required for the last stripe of each I/O, i.e., the starting block is always aligned with the stripe boundary [24]. This may yield an optimistic estimate of the performance of RAID 5.

Let p_f be the probability that a write request is a full stripe write. For a partial stripe write, since it accesses multiple blocks from each disk, after reading out a block from the last stripe (for the purpose of a new parity block calculation), the disk will have rotated part way to the beginning of the $B' = \overline{B}/(N-1)$ data blocks. Therefore, instead of a full rotation, it only needs to rotate the remaining distance prior to writing back the new data. We approximate this latency by a mean rotational latency[10]. Thus the mean service time for a partial stripe write is

$$1/\mu'_w = E[Y'_w] = E[X] + \tau + E[R] + E[T]$$

and the density function for the disk service time, Y, is given by

$$f_Y(x) = p_r \mu_r e^{-\mu_r x} + p_w p_f \mu_r e^{-\mu_r x} + p_w (1 - p_f) \mu'_w e^{-\mu'_w x}.$$

RAID 5 can now be modeled as a $M/G/1$ queue.

Comparison of RAID 1 and RAID 5

First we compare RAID 1 and RAID 5 with the same capacity, i.e., the RAID 1 system contains $2n$ disks, whereas the RAID 5 system contains $n+1$ disks (with $n = 32$). The results are shown in Figure 4.23 for different values of the *full stripe probability* p_f, and RAID 1 is observed to be much better than RAID 5 in all of the cases. Figure 4.24 illustrates the situation in which both RAID 1 and RAID 5 contain the same number of disks. We observe from Figure 4.24 (a) that RAID 1 outperforms RAID 5 when most of the requests are reads. This is because RAID 1 can serve two reads simultaneously, while RAID 5 has to serve read requests sequentially. Although the transfer time under RAID 5 is only half of that under RAID 1, the reduction in queueing delay caused by concurrently serving multiple reads under RAID 1 outweighs the disadvantage of the longer transfer time. When most requests are writes, RAID

[10]If the distribution of the request size is given, then this latency can be precisely calculated.

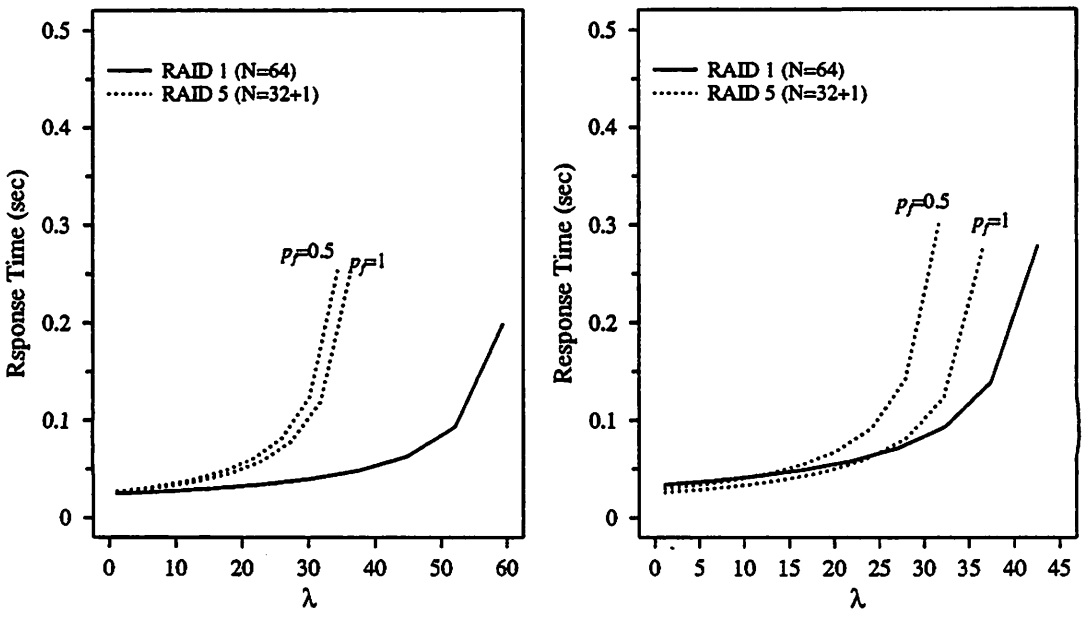


Figure 4.23 Compare RAID 1 and RAID 5 in Large I/O Environments (*Same Capacity*).

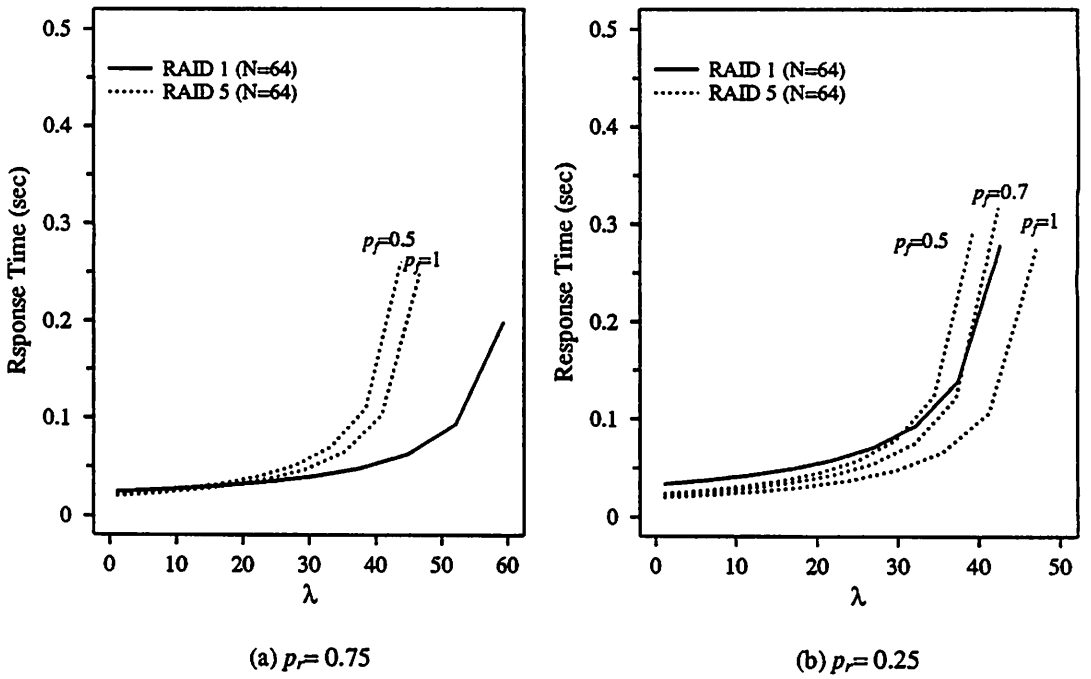


Figure 4.24 Compare RAID 1 and RAID 5 in Large I/O Environments (*Same No. of Disks*).

1 loses its advantage of being able to perform concurrent reads. Hence RAID 5 may exhibit better performance because of its shorter transfer time. However, as shown in Figure 4.24 (b), the performance of RAID 5 degrades rapidly as the full stripe probability p_f decreases, because of the additional disk service time required for the read-update procedure of partial stripe writes. When 30% of writes are to a partial stripe, RAID 1 is essentially equivalent to RAID 5 at high loads, and when half of the writes are to a partial stripe, RAID 1 is observed to outperform RAID 5.

In the above discussions, we assumed that most of the I/O requests to RAID 5 perform full stripe I/O operations ($p_f > 0.5$). This requires the file system or cache manager to be aware of the stripe width of the underling disk arrays. Whenever the number of disks in the array increases or decreases, the file systems or cache manager has to adapt to the change in order to maintain high performance.

4.4 Simulation Validations

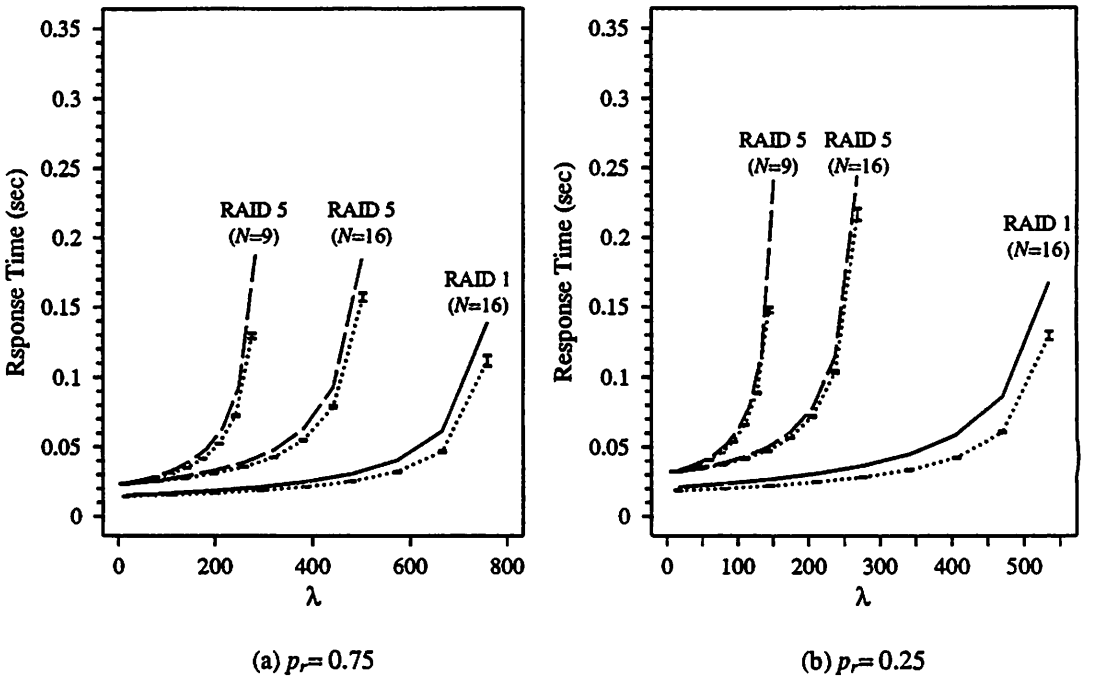
In several of our analytic models described in the previous sections, disk service times were assumed to be independent and exponentially distributed random variables for tractability. However, in reality, disk service times are not exponential by distribution. In order to validate the performance predicted by these models, we developed simulators for these disk array architectures that simulate the seek, rotational latency, and transfer times on an individual disk properly. The distributions of seek times and rotational latencies are taken from Section 2.3.

The simulation results are obtained by averaging over 40 runs, each run including 50,000 I/O requests. We obtained 95% confidence intervals whose widths are less than 2.5% of the estimated point values.

In the following, we report the results from three of our simulation experiments. Similar behavior has been observed in a number of other experiments which are omitted from our discussions.

4.4.1 Case 1: RAID 1 and RAID 5 in Small I/O Environments

Figure 4.25 illustrates the analytic and simulation results which compare the performance of RAID 1 and RAID 5 in a small I/O environment (see Section 4.2.4).



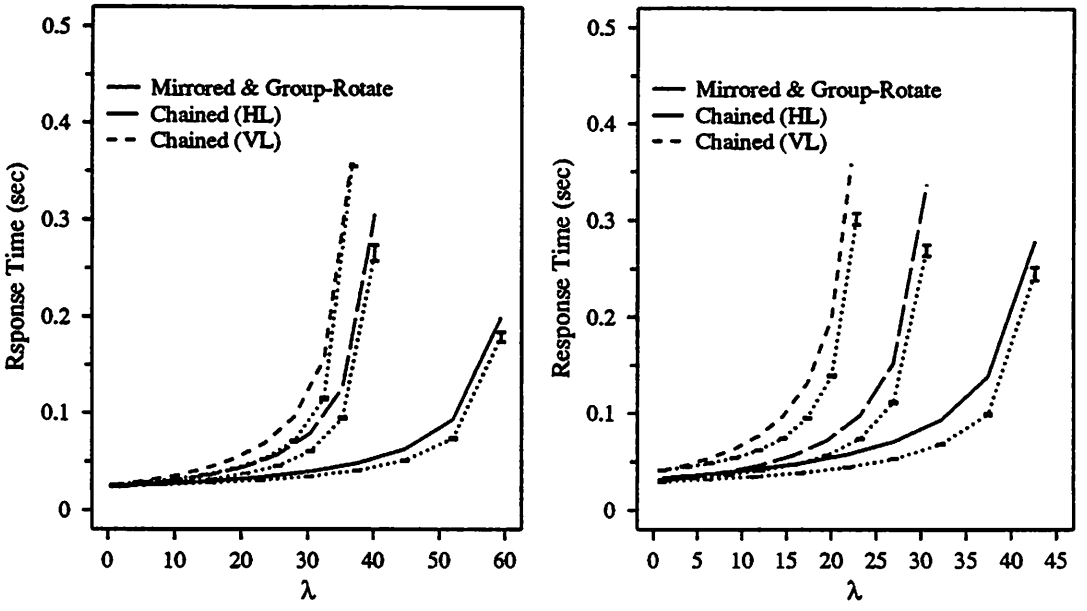
(a) p_r= 0.75 (b) p_r= 0.25

Figure 4.25 Model Validation: RAID 1 vs. RAID 5 in Small I/O Environments.

As in the analysis, the simulation of RAID 5 does not account for the synchronization delay that is incurred while updating the parity block during the execution of a write request in a small I/O scenario. Hence, the predictions made for RAID 5 remain optimistic. As shown in the Figure, the analytic and simulation results have a slight difference (with the analytic model overestimating the mean response time by up to 20% at high loads for RAID 1), because of the unrealistic exponential service time assumption made in our analysis. However, the qualitative differences between RAID 1 and RAID 5 are the same in all cases, and it is safe to say that RAID 1 can provide a much higher performance than RAID 5 in small I/O environments.

4.4.2 Case 2: Variations of RAID 1 in Large I/O Environments

Figure 4.26 plots the results of simulating RAID 1 (mirrored declustering) and its variants, chained declustering and group-rotate declustering, in a large I/O environment (see Section 4.3.1). As in the previous subsection, a slight difference between the simulation and analytic results are observed. Once again, the performance ordering of all these curves remains the same for both simulation and analysis. Furthermore,



(a) Analysis vs. Simulations (Pr=0.75)

(a) Analysis vs. Simulations (Pr=0.25)

Figure 4.26 Model Validation: RAID 1 and Its Variants in Large I/O Environments.

the quantitative differences between different architectures from both simulation and analysis remain basically the same. Therefore, our conclusion that mirrored declustering and group-rotate declustering are better than chained declustering in large I/O environments is validated.

4.4.3 Case 3: RAID 5 vs. Parity Striping

Last, we indicate that while the assumption of exponential service time distribution is made in other analyses, in Section 4.2.3, we model the RAID 5 and Parity Striping by using real disk seek and rotational latency distributions. We have run several simulations to validate this model. As a consequence, in most of the cases, the simulation results match the analytic results very well, with the simulation curves overlapping the analysis curves. The only discrepancy shows up for the RAID 5 model when the mean request size is large. This is because the queueing times and service times of all requests (Data and Parity) are assumed to be independent in the approximate analysis. While this independence assumption is valid for single block accesses, it is violated in the case that requests are allowed to access multiple blocks

of data. First, arrivals to the different disks are not independent since all of the data tasks associated with a read or write request arrive simultaneously at different disks. Second, the tasks corresponding to a single request leave the disk arms on the same cylinder. Hence a subsequent request may generate data tasks that will exhibit the same seek times at different disks. Last, in the case that a write request is to separate stripes (i.e., across stripe boundary), although the data tasks are routed to separate disks, it is possible that the parity task associated with one stripe will be routed to one of the disks that is executing a data task associated with the other stripe.

As a consequence, we have observed that the analytic results match those of simulation very well when the request size follows a quasi-geometric distribution (see Section 4.2.3) with mean 1.5 blocks. When the request sizes are uniformly distributed in $[1, N-1]$ (with $N = 16$), the approximate analysis produces estimates of the mean response times that are as much as 12% higher than those obtained from the simulation. The largest discrepancy occurs at high loads when most requests are write requests.

4.5 Summary

The main contributions of this chapter are the study of the design issues, development of mathematical models, and examination of the performance of different disk array architectures, both in small I/O and large I/O environments.

First, we considered RAID 1, *mirrored declustering*, and its two variants, *chained declustering* and *group-rotate declustering*. We observed that all of the three architectures coupled with the random join policy DP-RJ perform essentially the same for transaction processing and workstation/engineering environments. However, when coupled with the shortest queue policy DP-SQ-MS, group-rotate declustering was observed to be the best, and mirrored declustering to be the worst. For supercomputing and image processing systems, mirrored declustering and group-rotate declustering provide much better performance than chained declustering. Under chained declustering, the *Horizontal* data layout strategy was observed to be better than the *Vertical* layout strategy except when over 90% of requests are reads. While group-rotate declustering has exhibited an equivalent or better performance than

mirrored declustering during normal mode, it is also expected to perform better than mirrored declustering when disk failure occurs. Since the main focus of this chapter is to study the performance of disk arrays in the normal mode, we leave the performance evaluation of these architectures during failure, especially for group-rotate declustering, in a future study.

Second, we compared RAID 1 and RAID 5 in two different ways: (1) where the two systems have the same capacity, i.e., allowing RAID 1 containing $2n$ disks and RAID 5 containing $n + 1$ disks; and (2) where the two systems contain the same number of disks. Consequently, RAID 1 was observed to perform much better than RAID 5 in terms of minimizing the mean I/O response times. The only exception exists when the I/O size is very large, most of the requests are writes, and most of the writes perform a *full stripe write*. Obviously, RAID 1 gains a performance benefit at the cost of doubling the number of disks or losing half of its capacity, as compared to only a $1/N$ loss under RAID 5.

Third, we studied RAID 5 vs. Parity Striping for applications with typically small reads and writes. In this context, we first identified the "write synchronization problem" that exists in these two architectures and has been ignored in previous performance studies. We then proposed two synchronized scheduling policies suitable for both RAID 5 and Parity Striping. These two policies, we believe, provide a practical way for the implementation of RAID 5 and Parity Striping. A priority queueing model was developed for the analyses of RAID 5 and Parity Striping coupled with the two scheduling policies. The results show that RAID 5 is sensitive to the increase of mean request size and Parity Striping is sensitive to the skewed accesses. Hence, we conclude that RAID 5 is most useful for those applications in which most of the requests only access one block of data, and it can better tolerate skewed accesses. On the other hand, Parity Striping is suitable for those applications where most of the requests access more than one block of data and each data file is likely to be accessed with same probability. These results also suggest that if an array is implemented as a RAID 5, the block size, which is the unit of interleaving, should be large enough to cover the size of a large fraction of requests for the application. If the access pattern is skewed, the increase of block size should be limited. The best performance can

only be achieved by carefully tuning the block size.

Last, from the above studies, we can see that the main impact on the performance of RAID 5 is the high cost of the *partial stripe write*. Researchers have looked at different ways to reduce this cost. The most attractive way is the *Log-Structured File System*, as described in Section 2.1.2, in which small writes are accumulated in cache and converted into a large full stripe write. Another way is called *floating parity* [72], where parity blocks are clustered into cylinders each containing a spare track. A new parity block, instead of being written in place, is written on the nearest unallocated block following the old parity block. The cost paid is that the controller has to keep a map of the locations of all parity blocks.

CHAPTER 5

DISK SUBSYSTEMS FOR REAL-TIME COMPUTING SYSTEMS

In this chapter, we study disk I/O subsystems for real-time systems. Real-time systems typically divide into two categories, *hard real-time* systems in which missing a deadline may cause catastrophe, and *soft real-time* systems in which missing a deadline only reduces the 'value' of the system. Since hard real-time systems involve few runtime disk I/O activities, we will focus on disk I/O for soft real-time systems. Soft real-time systems can be further divided into two classes: *removal* systems, in which a computational entity that misses deadline is removed from the system without being served, and *non-removal* systems, in which all the entities are to be served eventually even though some of them have missed their deadlines. Here, we will study disk I/O architectures for both removal and non-removal real-time systems. The main focus of this chapter is on the development and evaluation of real-time disk scheduling algorithms. In doing so, we first propose two new real-time disk scheduling algorithms and then evaluate them along with other known algorithms in an integrated removal real-time transaction system model. We also study the behavior of mirrored disks and disk arrays in a real-time environment by combining these real-time algorithms with the architectures and policies studied in the previous two chapters. Last, we examine disk I/O for non-removal real-time systems in an open system model.

5.1 Description of Various Disk Scheduling Algorithms

In this section, we describe all nine disk scheduling algorithms examined in this chapter. We first introduce the two new real-time disk scheduling algorithms, SSEDO and SSEDV. We then describe three other real-time disk scheduling algorithms, ED (*the earliest deadline strategy*), P-SCAN, and FD-SCAN. It should be noted that ED

is a special case of each of the two new real-time algorithms. Finally, we very briefly describe four traditional disk scheduling algorithms.

5.1.1 Two New Real-Time Disk Scheduling Algorithms

Motivation

In a real-time system, timing constraints are important to consider when making scheduling decisions. For real-time task scheduling, Towsley *et al.*[118] have proved that the *earliest deadline* policy is optimal in the sense of minimizing the loss probability among all policies which are independent of task service times, given that these task service times are *i.i.d.* and exponentially distributed. For disk scheduling, however, the independence and exponential assumptions are no longer valid.

For example, disk service times depend on the scheduling discipline employed. A simple example can be found in considering the *First Come First Serve* (FCFS) and the *Shortest Seek Time First* (SSTF) disciplines, where the mean disk service time under FCFS is shown to be longer than that under SSTF [48]. Second, successive service times are not independent since a request's service time depends on the current disk arm position (the cylinder address of the previously served request). Third, disk service times are not exponentially distributed, in general. From these observations, we can expect that a proper disk scheduling algorithm for a real-time system should take into account not only the time constraint but also the disk service time.

The SSEDO and SSEDV Algorithms

Based on the above considerations, we propose two new real-time disk scheduling algorithms, SSEDO (for *Shortest Seek and Earliest Deadline by Ordering*) and SSEDV (for *Shortest Seek and Earliest Deadline by Value*), for a single disk subsystem [26]. We introduce the following notation:

n: the number of requests in the queue waiting for service;

r_i: the I/O request with the i-th smallest deadline at a scheduling instance, $1 \leq i \leq n$;

d_i: the distance between the current arm position and request r_i's position, $1 \leq i \leq n$;

L_i: the absolute deadline of r_i, $1 \leq i \leq n$.

Note that an I/O request's deadline can be the same as that of the computational entity (e.g., a transaction in our model) issuing the request, or can be calculated by using the entity's deadline and other information, as will be shown for the step deadline in Section 5.2.1.

The two algorithms maintain a queue sorted according to the (absolute) deadline, L_i, of each request. A window of size m is defined as the first m requests in the queue, i.e., the window consists of m requests with smallest deadlines. Hence we shall refer to these two algorithms, SSEDO and SSEDV, as window algorithms. The advantage of defining a window can be seen later in this section.

The SSEDO Algorithm: At a scheduling instance, the scheduler selects one of the requests from the window for service. The scheduling rule is to assign each request a weight, say w_i for request r_i, where $w_1 = 1 \leq w_2 \leq ... \leq w_m$ and m is the window size, and to choose the one with the minimum value of $w_i d_i$. We shall refer to this quantity $w_i d_i$ as the *priority value* associated with request r_i. If there is more than one request with the same priority value, the one with the earliest deadline is selected. It should be clear that the priority value for any specific request may vary at each scheduling instance, since d_i, r_i's distance to the disk arm, changes as the disk arm moves.

The idea behind the above algorithm is to give requests with smaller deadlines higher priorities so that they can receive service earlier. This can be accomplished by assigning smaller values to their weights. On the other hand, when a request with a large deadline is "very" close to the current arm position (which means less service time), it should get higher priority. This is especially true when a request is to access the cylinder where the arm is currently positioned. Therefore these requests should be given the highest priority. There are various ways to assign these weights, w_i. In our experiments, the weights are simply set to

$$w_i = \beta^{i-1} \quad (\beta \geq 1) \qquad\qquad i = 1, 2, ..., m$$

where β is an adjustable scheduling parameter. Note that SSEDO assigns priority only on the basis of the *relative order* of deadlines, i.e., the value of w_i is independent

of the absolute and relative deadlines of r_i. Observe that, when all weights are equal ($\beta = 1$), the algorithm serves requests in the window according to the SSTF discipline. As the window size becomes large, the algorithm converges to pure SSTF. When the window size is equal to one, the algorithm is the same as the ED algorithm. In the following section, we will see that performance of the system is improved dramatically by choosing a window size of three or four.

The SSEDV Algorithm: In the SSEDO algorithm described above, the scheduler uses only the ordering information of requests' deadlines and does not use the differences between deadlines of successive requests in the window. For example, suppose there are two requests in the window, with r_1's deadline very small and r_2's deadline very large. If r_2's position is "very" close to the current arm position, then the SSEDO algorithm might schedule r_2 first, which may result in the loss of r_1. However, if r_1 is scheduled first, then both r_1 and r_2 might get served before their deadlines. In the other extreme, if r_2's deadline is almost the same as r_1's, and the distance d_2 is less than d_1 but greater than d_1/β, then SSEDO will schedule r_1 for service and r_2 may be lost. In this case, since there could be a loss anyway, it seems reasonable to serve the closer one (r_2), for its service time is smaller. Based on these considerations, we expect that a more intelligent scheduler should use not only the deadline *ordering* information but also the deadlines themselves for decision making. This leads to the following algorithm: associate a priority value of $\alpha d_i + (1 - \alpha) l_i$ to request r_i and choose the request with the minimum value for service, where l_i is the *remaining life time* of request r_i, defined as the length of time between the current time and r_i's deadline L_i, and α ($0 \leq \alpha \leq 1$) is a scheduling parameter.

Again when $\alpha = 1$, this approximates the SSTF algorithm, and when $\alpha = 0$, this corresponding to the ED algorithm.

A common characteristic of the SSEDV and SSEDO algorithms is that both consider *time constraints* and *disk service times*. The relative contributions of these components in decision making can be adjusted by tuning the scheduling parameters α or β, depending on the algorithm.

Scheduling Costs of the SSEDO and SSEDV Algorithms

From an implementation point of view, the ED queue required for the SSEDO and SSEDV algorithms can be maintained by the operating system or the device controller when requests arrive at a cost $O(\log n)$ in the number of requests, n, in the queue. For a fixed window size m (usually $m = 3$ is sufficient as shown in the next section), the priority value calculation cost is $O(1)$ in n. That is why we define a window, rather than compute priority values for all of the requests in the queue which would result in a cost of $O(n)$. This is motivated by the approximate calculations used in [49], and this idea is further explored in [43]. Since the $O(\log n)$ queue maintenance cost at arrival instances can be performed in parallel with disk operations, and since there is only a $O(1)$ cost at scheduling instances, the scheduling cost can be kept low. In addition, the $O(1)$ priority value computation cost can be further reduced by scheduling one request while serving another. This is feasible since the cylinder address of the request under service is known. Hence, the seek distances and remaining lifetimes of all the requests waiting in the window can be calculated in advance.

5.1.2 Other Real-Time Disk Scheduling Algorithms

Three other real-time disk scheduling algorithms, ED, P-SCAN and FD-SCAN, have been suggested in the literature.

The ED Algorithm: In the ED algorithm, I/O requests are served based on their deadlines. The request with the earliest deadline has the highest priority.

The P-SCAN Algorithm: The priority SCAN (P-SCAN) strategy is based on an idea suggested by Carey [23]. Specifically, all requests in the I/O queue are divided into multiple priority levels. The SCAN algorithm is used within each level, i.e., the disk arm serves any requests that it passes in the priority level currently being served until there is no more request in that direction. On the completion of each disk service, the scheduler checks to see whether a disk request of a higher priority is waiting for service. If a higher priority request is found, the scheduler switches to that higher level. In this case, the request with the shortest seek distance from the current arm position is used to determine the scan direction.

It is observed in [23] that a small number of priority levels is preferred in order to provide reasonable I/O performance. How to assign priority to an I/O request is an interesting question, which is not treated in [23]. In our experiments, all I/O requests are mapped into three priority levels according to their deadline information, as will be described in Section 5.2.2.

The FD-SCAN Algorithm: Recently, Abbott and Garcia-Molina [3] proposed another variant of the SCAN algorithm, called FD-SCAN. In FD-SCAN, the track location of the request with the earliest feasible deadline is used to determine the scan direction. A deadline is *feasible* if the scheduler estimates that it can be met. Determining the feasibility of a request's deadline is simple. Once the current arm position and the request's track location are known, its service time can be estimated. A request's deadline is feasible if it is greater than the current time plus the request's service time. At each scheduling point, all requests are examined to determine which has the earliest feasible deadline. After selecting the scan direction, the arm moves toward that direction and serves all requests along the way.

5.1.3 Four Classical Disk Scheduling Algorithms

The four classical scheduling algorithms described below are well-known. They have been discussed extensively in the literature. Here we simply list them for ease of reference.

FCFS: This is the simplest strategy in which each request is served in a first-come-first-serve fashion.

SCAN: This is also known as the *elevator* algorithm in which the arm moves in one direction and serves all of the requests in that direction until there is no further request in that direction. The arm then changes its scan direction and repeats this operation.

C-SCAN: The circular SCAN algorithm works in the same way as SCAN except that it always scans in one direction. After serving the last request in the scan direction,

the arm returns to the start position (typically an edge of the disk) without servicing requests and then begins scanning again.

SSTF: The SSTF (*shortest seek time first*) algorithm simply selects the request closest to the current arm position for service.

A common feature of all these classical scheduling algorithms is that none of them takes the time constraints of requests into account. As we shall see later that this results in their poor performance in real-time systems.

5.2 A Single Disk I/O Subsystem

In this section, we first describe a real-time transaction system model, and then compare the performance of all nine algorithms described in the last section by using a simulator of the real-time transaction system model.

5.2.1 A Real-Time Transaction System Model

Model Description

We first describe a system model which takes an integrated view of real-time transaction performance. While it is our intent to specifically study real-time disk I/O scheduling algorithms, we do so in a complete system setting. This allows us to include system overhead, such as waiting for CPU, for each I/O request. The overall metric of interest is the transaction loss probability, i.e., the fraction of transactions that do not meet their deadlines; note that this is only partially affected by the disk I/O policy. Consequently, we study nine disk I/O scheduling algorithms in a system with fixed algorithms for concurrency control, lock conflict resolution, commit processing, CPU scheduling, deadlock detection, deadlock resolution, transaction restart, and transaction wakeup. We now describe the system model in detail.

Since we are interested in the disk I/O scheduling problem for real-time transaction systems, the database is assumed to be very large and disk resident. Specifically, the transaction system is modeled as a closed system which consists of multiple users, a CPU, and a disk for storing the database (Figure 5.1). All log information is placed

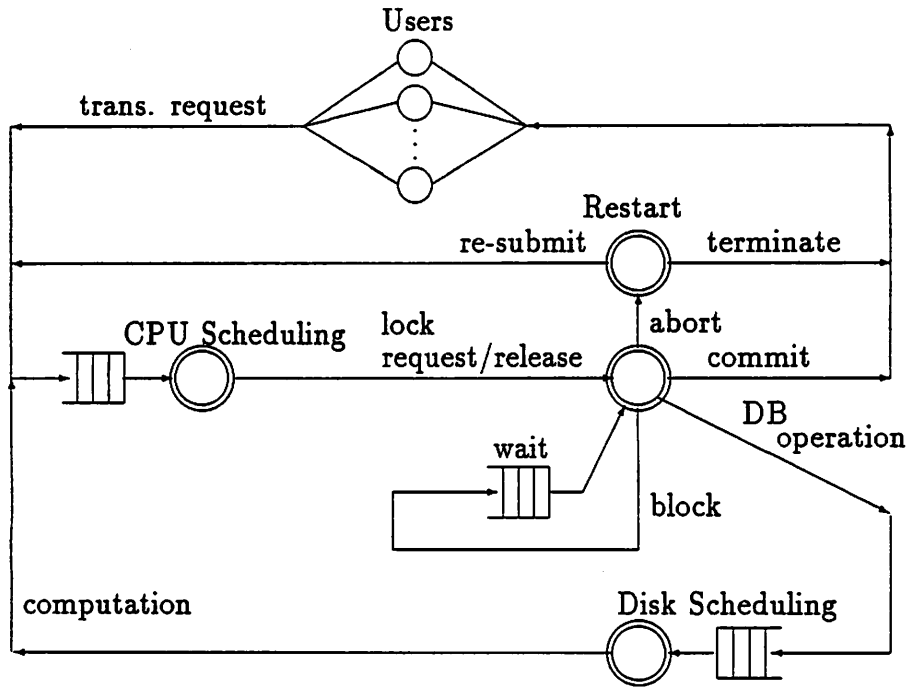


Figure 5.1 A Real-Time Transaction System Model.

on a separate log disk. The log disk does not show up in the figure because its cost is accounted for and subsumed as part of the CPU operation.

In this system model, each user spends a random amount of time in a *think* state before generating a transaction. Each transaction is considered to have the same importance and is assigned a deadline when submitted to the system. If a transaction cannot commit before its deadline, it is said to be lost and is removed from the system immediately. The user who submitted the lost transaction returns to the *think* state. A transaction consists of a random number of operational steps. Each step performs an access to the disk, (only if the desired page is not in the main memory buffer), and manipulates the data items retrieved. In some of our experiments, we assume that the number of steps for a transaction is known to the system as soon as the transaction arrives. This assumption is often reasonable since the knowledge of a transaction's length can sometimes be obtained from the compiler or some other transaction preprocessor by counting the transaction's I/O requests.

Generally, we assume that transactions randomly access records and that multiple records are stored on each page of data. We also assume that the size of a page is the

same as the size of a disk block. The random access assumption will be relaxed when we examine the effect of locality of transactions' accesses. In the case of transactions randomly accessing the database, since the database is assumed large, the memory buffer hit ratio is low and therefore disk I/O is required most of time. This better enables us to investigate the impact of I/O scheduling algorithms when the system is I/O bound.

To maintain the consistency of the database, serializability is enforced by using the *two-phase locking (2PL)* protocol. Specifically, each page is associated with a lock. Each transaction step needs to request and hold a lock before being allowed to access the database. For any transaction, all locks obtained at each step will be held until it commits or aborts. Two types of locks are identified, shared locks and exclusive locks. A shared lock can be granted to multiple users for query (read) operations, and an exclusive lock is granted to only one user for update (write) operations. A lock conflict may occur when a requested lock has been granted to other transactions, and either the requesting or the holding transaction is to perform an update operation. In order to handle this situation, a lock conflict resolution policy must be defined.

In the following, we briefly describe the set of policies and protocols used in our model, which co-operatively support a real-time transaction processing system.

Lock Conflict Resolution: When there is a lock conflict, the requesting transaction always suspends, i.e., joins a queue waiting for the lock. There is no lock preemption. This is based on the assumption that all transactions are of the equal importance so that preemption is not warranted. When each transaction carries, in addition to a deadline, a distinct importance value, preemption has been shown to be a good strategy [53].

Deadlock Resolution: The locking scheme used may cause deadlocks. A simple cycle detection deadlock detection algorithm is employed in our model. When a deadlock is detected, the requesting transaction is aborted. In this case, the transaction releases all resources held and proceeds according to the restart strategy.

Restart Policy: A transaction aborted from a deadlock is restarted as long as its deadline has not yet expired. Transactions aborted for other reasons, such as user

termination, arithmetic overflow, and execution exceptions, are removed from the system and returned to the *think* state. In the former case, a smarter policy might restart a transaction only if its estimated minimum processing time is less than the remaining time before missing its deadline. In the latter case, a more sophisticated strategy might restart a transaction if it is aborted due to arithmetic overflow and/or execution exception, after doing some exception handling.

Wakeup Strategy: When a transaction commits, aborts, or misses its deadline, it releases all of its resources. If some of these resources (pages) are needed by more than one other transactions, then the one with the earliest deadline among all waiting transactions is selected to wakeup and granted the associated lock.

Transaction Commit: When a transaction commits, it releases all of its resources after completion of logging and returns to the *think* state. If there are any dirty pages in the memory buffer, they are flushed to the disk. Obviously, these flushing operations do not deserve a high priority since the transaction issuing these operations has already committed. On the other hand, from the buffer manager's standpoint, flushing operations should complete as soon as possible to free up buffer space. This means that they should not have a low priority. In our experiments, we choose to not explicitly degrade the priority of flushing write operations.

Buffer Manager: The buffer manager is not explicitly modeled in order to simplify the simulation programs. Instead, a *buffer hit probability*, which is assumed to be 0.1, is used to determine whether a disk access is required.

CPU Scheduling: The *earliest deadline* (ED) algorithm is used for CPU scheduling. As mentioned in the literature [2], a major drawback of this algorithm is that it may assign the highest priority to a transaction that has already missed or is about to miss its deadline. To partially overcome this weakness, we propose a modified ED algorithm which, instead of scheduling according to a transaction's deadline, schedules according to the *step deadline* of a transaction that is a function of the number of steps in that transaction, given the number of steps is known to the system at the time of submission. Specifically, let L, a, and n be a transaction's (absolute) deadline, arrival

time, and the number of operational steps. A simple way to assign a step deadline for step i is to set

$$step\_deadline(i) = a + \frac{i}{n}(L - a). \tag{5.1}$$

Note that under this policy, the earlier the step, the smaller the step deadline, and therefore, the higher the priority for the transaction when executing its earlier steps. Also note that the deadline for the last step is always equal to the transaction's deadline. One consideration for using the step deadline is that if two transactions have the same deadline, but one executes more steps than the other, then the longer transaction will have a higher priority for its earlier steps. We tested the effect of using step deadlines on both CPU and I/O scheduling. The results show that this strategy does indeed improve the system performance (see Section 5.2.3).

In summary, in order to achieve good performance in the sense of minimizing the transaction loss probability, there is a collection of algorithms that must be developed, including CPU scheduling, commit processing, I/O disk scheduling, concurrency control, lock conflict resolution, deadlock resolution, restart, and wakeup. Since our main goal is to examine the impact of I/O scheduling in a real-time transaction system, we have selected a simple, but suitable set of algorithms for the system.

Model Validation

In order to validate our basic model of an integrated set of protocols required to perform real-time transaction processing, we conducted a series of experiments on an actual real-time database testbed, called RT-CARAT [53]. Given the the same workloads and algorithm settings (including CPU scheduling, lock conflict resolution, deadlock resolution, restart strategy, and commit processing), our simulation results concerning the basic simulation model (which uses the FCFS disk service strategy) match quite well with that of the testbed. On the other hand, it was not possible for us to validate the simulation results comparing all nine I/O disk scheduling algorithms because the disk controller in the testbed could not be modified.

RT-CARAT is currently running on a VAX[1] Station II/GPX under the VMS operating system. The database consists of 3000 pages, with each page containing 6 records. The number of users is fixed at 8. Each user continuously generates transactions with a fixed number of operational steps. When a transaction commits or aborts, the corresponding user submits his next transaction immediately. There was no *think time* between transaction submissions during the validation runs. On RT-CARAT, each operational step accesses 4 records, which are randomly located in the database, and therefore up to 4 I/O requests may be generated at each step. In our model, since we assume each operational step only needs to access the database once, we set the total number of steps for each transaction to four times that in RT-CARAT so that a transaction will generate the same number of I/O requests. Although RT-CARAT implements various real-time CPU scheduling algorithms, conflict resolution protocols, deadlock resolutions, and wakeup strategies, it uses the basic disk I/O scheduling strategy employed by the underlying VMS operating system and its disk controller. In fact, since the VAX Station II/GPX uses a RQDX3 controller and a RD54 disk drive [119], both of which are very early models, I/O requests are served in an approximate first-come-first-serve fashion.

In setting deadlines for transactions we follow two steps: first, a selected workload is placed on RT-CARAT in a non-real-time database setting, and the mean transaction response time and its standard deviation are measured. In other words, we make a preliminary run to determine a transaction's mean response time (*avg_resp*) and its standard deviation (*std_devi*), under the assumption that transactions have no deadlines. This is done with a basic set of protocols that do not use deadlines to make decisions. Then a base deadline value (*base_line*) is calculated by

$$base\_line = avg\_resp - std\_devi$$

and each transaction's deadline is randomly selected from the range [*base_line*, $f *$ *base_line*], where f is a factor that enables us to test a wide range of deadlines. We set f to 3 in the validation experiments.

[1]VAX and VMS are trademarks of Digital Equipment Corporation.

In order to compare the performances of the simulation model and the testbed, the database configuration and the collection of algorithms supporting a real-time transaction system are configured to be the same on both. The system overhead, such as context switching, interprocess communication, locking mechanism, and logging, are considered and assimilated into the computation time for each step in our model. In order to compare their performances, the step computation time in our model is tuned so that both systems have approximately the same disk utilization in the non-real-time cases.

The comparison of the performance of RT-CARAT and our simulation model is shown in Figure 5.2. The x-axis gives the number of steps for each transaction in our simulation model. In Figure 5.2 (a) we see that transaction loss probability in our simulation model is slightly less than that in RT-CARAT when the system is lightly loaded, and slightly higher than that of RT-CARAT when the workload increases. One reason for this is that the variance in the transaction response time predicted by our model is less than that measured in RT-CARAT when the load is low, and it increases faster than its counterpart in RT-CARAT as the load increases. In Figure 5.2 (b) we observe that the average transaction response times in both models are the same. In Figure 5.2 (c), we observe that the average disk utilizations are nearly the same on both systems. On the other hand, CPU utilization drops faster in our simulation model than in the testbed.

5.2.2 System Parameter Settings

In the following, we describe the system parameter settings for our experiments, including: system configuration, transaction characteristics, deadline settings, and the parameters for the two new algorithms, SSEDO and SSEDV. Our experimental results are presented in the next subsection.

System Configuration: The database consists of 6000 pages, which are uniformly distributed on the disk with each page corresponding to a disk block.

The disk parameters are summarized in Table 5.1. These parameters differ from those used in previous chapters, since here we try to match the disk parameters used in our simulator to that of the RT-CARAT testbed.

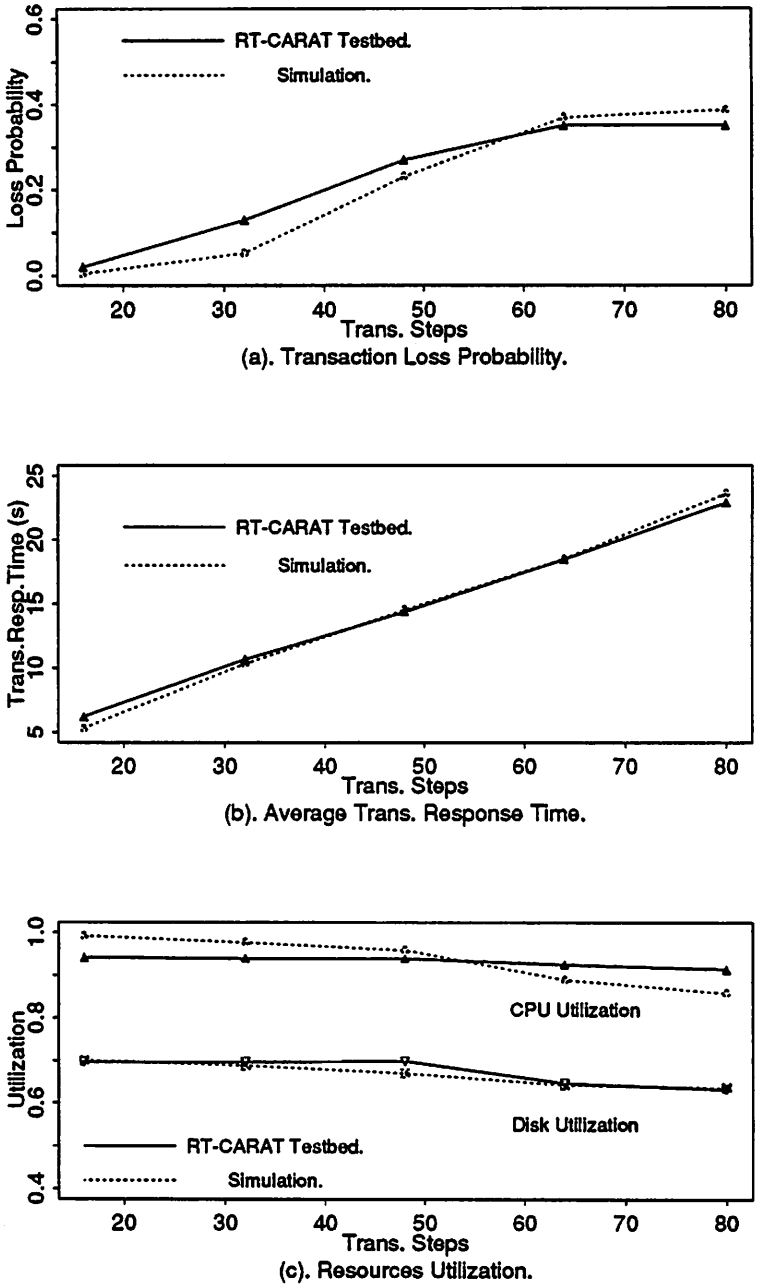


Figure 5.2 Results Compared with RT-CARAT Testbed ($Users = 8$, $p_r = 0.8$, $p_s = 0$).

Table 5.1 Disk Parameters

No. Tracks (C)	1000
Seek Factor (b)	0.5
Arm Acce. Time (a)	8 ms
Latency (R)	Uniform in [0, 16.7]
Transfer Time (T)	0.8 ms

Transaction Characteristics: Transaction characteristics and system workloads are defined as follows: the number of users, which limits the maximum number of transactions in the closed system, ranges from 4 to 20. Each user may think a random amount of time, which is exponentially distributed with mean 1 second, before generating a transaction. A transaction consists of a random number of operational steps which is uniformly distributed between 1 and 20. Each step needs to access the database once and is followed by a manipulation of those data items fetched. The computation time of each step is assumed to be 15 ms, and the time needed to abort a transaction is 5 ms. With these parameter settings, the simulated system is I/O bound, which simplifies our task of examining the disk scheduling algorithms.

Further, each transaction step may perform a read or an update operation to the database. While a transaction is progressing, it brings those pages desired into a main memory buffer and does a corresponding query or update there. In the case of update operations, the updated pages are also written on a separate log disk. When a transaction commits, all of its dirty pages are flushed to disk. In our model, the main memory buffer is assumed large enough to accommodate all of the pages desired by currently active transactions. In most of our experiments, the read probability, p_r, is set to 0.8, since most of the time users are querying the database. We also investigate the effect of varying the read probability from 0 to 1. To examine the effect of locality of a transaction's accesses, we define the *probability of sequential access*, p_s, to be the probability that a transaction step accesses the same cylinder as the previous step. There is no locality assumed among different transactions. Usually, p_s is set to 0

except when we study the locality effect in Section 5.2.3, where p_s is varied from 0 to 0.8.

Deadline Settings: The deadline setting for each transaction in our model depends on the system load and the transaction length. The system load can be characterized by the number of users in the closed system (with the mean think time fixed), and the transaction length corresponds to the number of steps. Specifically, we estimate a transaction's minimum system time by

$$T_{min} = (CPU\_Time + I/O\_Time) * Num\_Steps$$

where CPU_Time and I/O_Time are the estimated average CPU and disk service time for each step under the FCFS strategy. Then the transaction deadline is set by,

$$Trans\_Deadline = T_{min} * \eta$$

where η is a $r.v.$ drawn from a uniform distribution with lower and upper limits $[LOW\_DL, UP\_DL]$. Here LOW_DL is a deadline lower bound selected to be proportional to the number of users in the system, $LOW\_DL = k{*}Num\_Users$, and UP_DL is a deadline upper bound $UP\_DL = f * LOW\_DL$. In most of our experiments, k is equal to 0.25 and f is 4. However, we also examine the case in which transactions' deadlines are very tight and/or very loose by varying LOW_DL.

The deadline setting for each I/O request is inherited from that of the transaction issuing the request in most of the experiments. However, several experiments are performed in which each I/O request issued by a transaction is given a separate deadline as will be discussed in Section 5.2.3.

Parameters for SSEDV and SSEDO: The scheduling parameters, α and β, for SSEDV and SSEDO are set to 0.8 and 2 in most cases. One of our experiments examines the performance of SSEDV and SSEDO to changes in α and β under two different workloads. Another experiment studies the effect of varying the window size while other parameters remain unchanged. The results show that a window size of 3 or 4 is satisfactory, and therefore usually the window size, m, is set to 3.

Priority Levels for the P-SCAN Algorithm: In our experiments, all I/O requests are mapped into three priority levels according to their deadline information.

Specifically, we assume that transactions' relative deadlines are uniformly distributed between LOW_DL and UP_DL, where LOW_DL and UP_DL are lower and upper bounds for transaction deadline settings. If a transaction's relative deadline is greater than $(LOW\_DL + UP\_DL)/2$, then it is assigned the lowest priority. If the (relative) deadline is less than $(LOW\_DL+UP\_DL)/4$, then the transaction receives the highest priority. Otherwise the transaction is assigned a middle priority. This strategy is observed to be better than the *one-third* strategy which evenly divides the deadline range into 3 intervals and maps each interval into a priority level.

5.2.3 Simulation Results

In this subsection, we report our experimental results on the performance of the previously described nine algorithms. In particular, we study the performance over a wide range of workloads, assess the two new algorithms' sensitivity to window size and algorithm parameters, determine the impact of scheduling via step deadlines, examine the system behavior by varying transaction deadline settings and read probability, investigate the locality of transaction accesses, and report on the importance of real-time scheduling by comparing the performances of real-time and non-real-time scheduling algorithms.

Results of each experiment are averaged over 20 runs. In each run 1050 transactions are executed and 95% confidence intervals are obtained by using the method of independent replications. Confidence interval widths are less than 11% of the point estimates of the loss probability in all cases (these intervals are not shown in our figures in order to clearly show the results). For each run, the execution of the first 50 transactions is considered to belong to the transient phase and is excluded from our statistics. In order to avoid congested plots, the nine algorithms are divided into two groups, classical algorithms and real-time algorithms, and plotted separately. The curve for the FD-SCAN algorithm is shown in both plots to provide a basis of comparison. In the following, all transactions and I/O requests are scheduled according to the transaction deadline except where explicitly indicated.

By using those parameter settings specified in the last subsection, the mean I/O queue lengths are provided in Table 5.2 for some workloads. Other algorithms fall

Table 5.2 Average I/O Queue Length

Users	FCFS	SCAN	SSTF	FD-SCAN	ED	SSEDV
4	0.735	0.706	0.699	0.728	0.738	0.701
8	3.70	3.41	3.366	3.658	3.587	3.472
20	16.22	15.79	15.62	15.63	15.11	15.02

between those shown in this table. From these results, we can see that there are only slight differences in mean I/O queue lengths due to different algorithms. However, we will observe later that if we increase the step computation time, the mean I/O queue length becomes more sensitive to the CPU and disk scheduling algorithms.

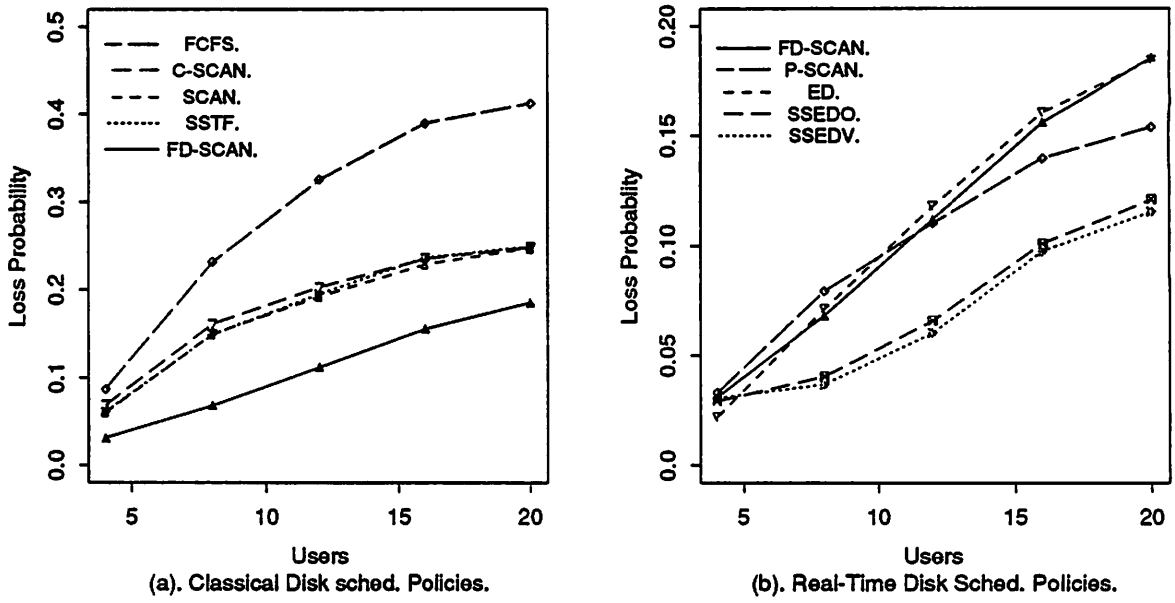


Figure 5.3 Performance of Various Disk Scheduling Algorithms.

Performance of Various Disk Scheduling Algorithms: In this experiment, we explore the transaction loss probabilities of these nine algorithms under different system workloads. The results are shown in Figure 5.3, from which we observe that the SSEDO and SSEDV algorithms significantly reduce the transaction loss probability compared to other SCAN-based real-time algorithms (up to a 38% improvement)

or conventional algorithms (up to a 53% improvement). Of these two algorithms, SSEDV is better than SSEDO, since SSEDV uses more timing information than SSEDO for decision making. All of the real-time algorithms perform better than the non-real-time ones. P-SCAN and FD-SCAN exhibit similar performance, with one slightly better at high loads, but slightly worse at low loads. The ED algorithm is good when the system is lightly loaded, but degenerates quickly as load increases. The conventional algorithms SSTF, SCAN and C-SCAN exhibit basically the same performance, and FCFS exhibit the worst performance.

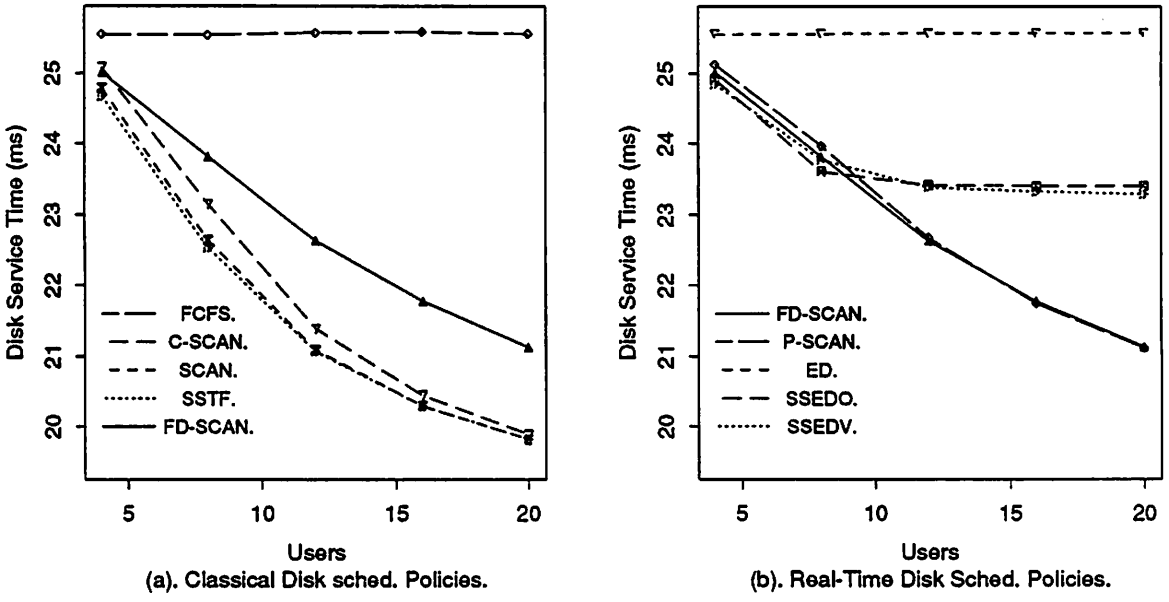


Figure 5.4 Mean Disk Service Time for Different Algorithms.

The mean disk service times under different algorithms are plotted in Figure 5.4. As expected, FCFS and ED both yield the highest mean disk service time which is independent of the system load, since the requests selected for service under either algorithm are randomly located on the disk independent of all other requests and system workload. The smallest mean disk service time belongs to SSTF, as its name suggests. SCAN shows almost the same performance as SSTF, and C-SCAN is a little bit higher [114]. Disk service times of the two variants of the SCAN algorithm, P-SCAN and FD-SCAN, are similar. The reason why they give higher mean service

times than SCAN is clear. A common characteristic of SSTF and various SCAN algorithms is that their disk access times decrease as the system load increases, due to their use of seek optimizations. For SSEDO and SSEDV, their mean disk access times decrease as the load increases, but after a point they become constant. This elbow depends on the window size, since the candidate for next service can only be selected from the window. From Table 5.2 we observe that when the number of users exceeds 8, the mean queue length typically exceeds 3. As mentioned above, the window size is set to 3 in this experiment. Consequently, further increasing the system workload has no effect on the mean disk access time.

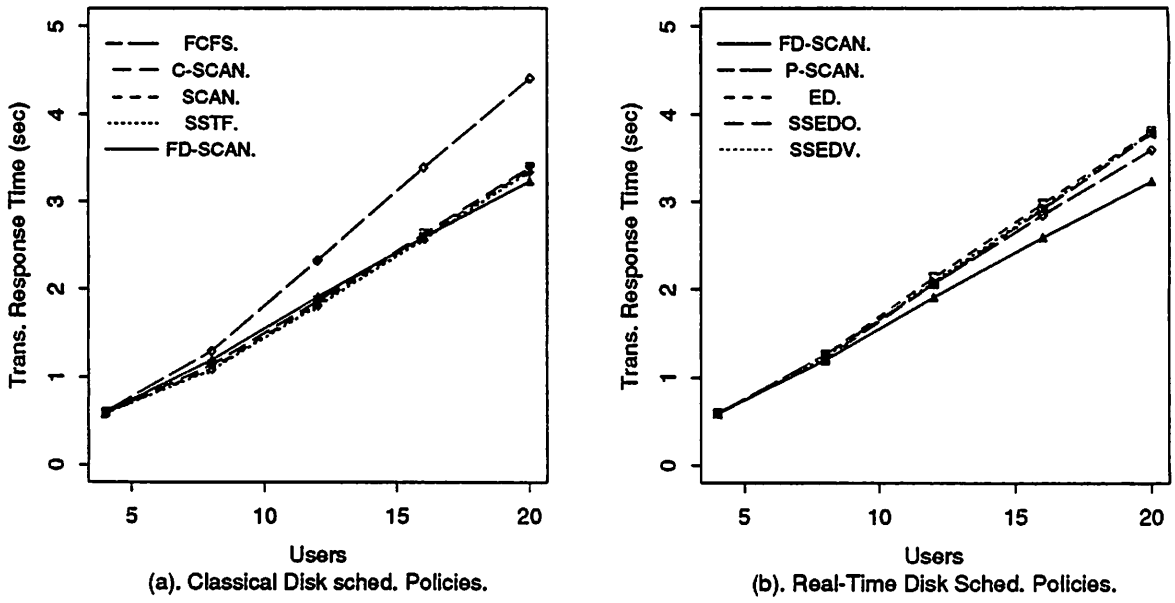


Figure 5.5 Mean Response Time for Committed Transactions.

Figure 5.5 shows the average response time for committed transactions. Given the mean disk service times shown in Figure 5.4, it is not hard to understand why transactions under SSEDO and SSEDV require more time to complete than those under SSTF and the various SCAN algorithms but less than under FCFS and ED. FD-SCAN has the same mean response time as SSTF, SCAN, and C-SCAN. This is because, although FD-SCAN exhibits a longer mean disk access time, its transaction loss probability is less than that of SSTF and SCAN. For those lost transactions, by

the times they miss their deadlines, they have already completed part of their steps. These steps consume resources which may result in the blocking of other transactions and therefore increase the response times of transactions which do commit.

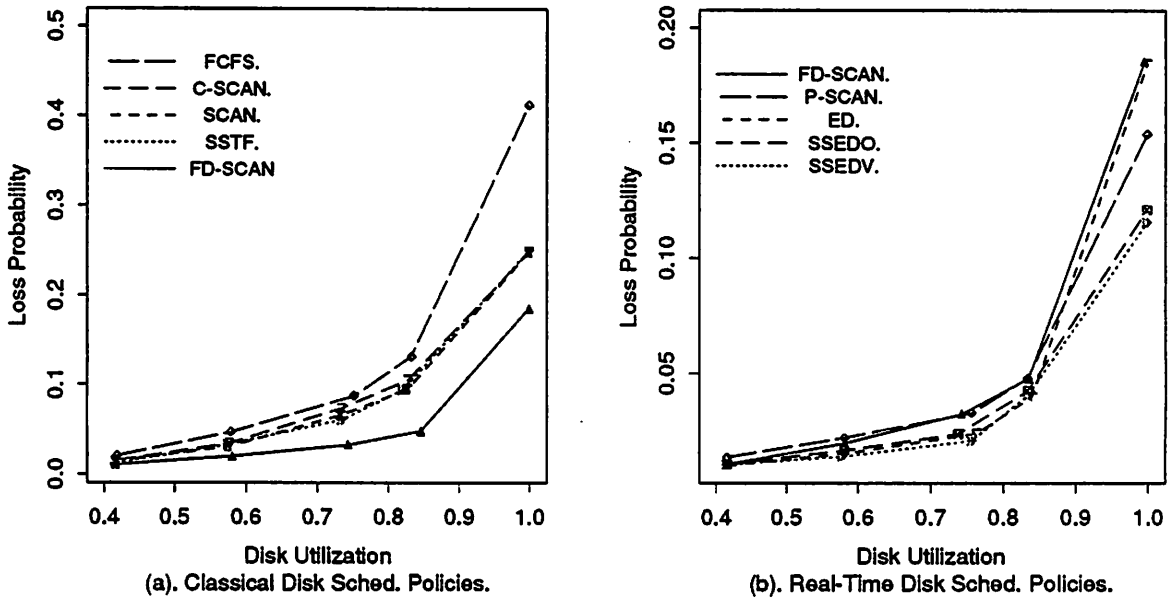


Figure 5.6 Performance with Fixed Disk Utilizations.

Figure 5.6 depicts the performance of all nine algorithms as a function of the disk utilization. The advantage of real-time algorithms over non-real-time ones is obvious. For real-time algorithms, the window algorithms SSEDO and SSEDV begin to show better performance than P-SCAN and FD-SCAN when disk utilization is over 55%. Their advantage becomes clearer when the disk is kept busy over 90% of the time. For example, at 95% disk utilization, SSEDV is observed to outperform FCFS by 71% and FD-SCAN by 36%.

Sensitivity of Changing Window Size and Scheduling Parameters in SSEDV and SSEDO: With this experiment, we are interested in how the system performs when the window size and/or scheduling parameters α and β are changed in the two window algorithms SSEDV and SSEDO. Note that the leftmost points in Figure 5.7 correspond to a window size of one, which is equivalent to the ED algorithm. By increasing the window size, we observe an improvement in the performance, especially

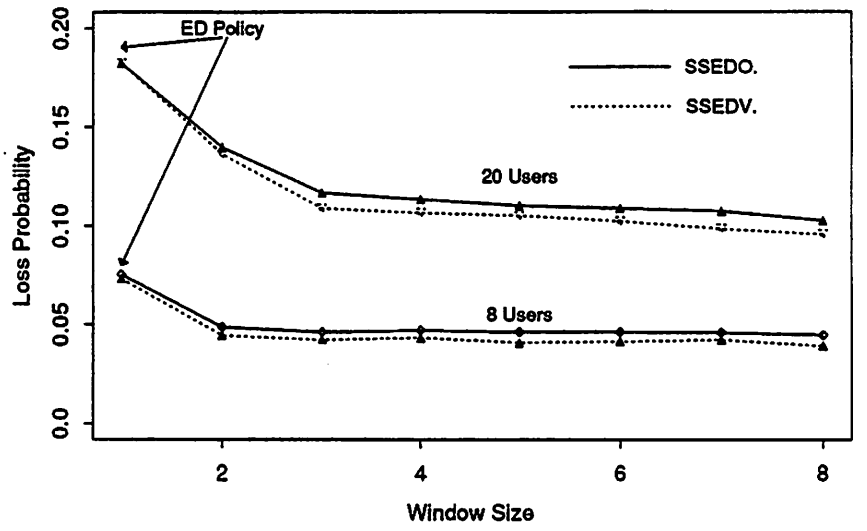


Figure 5.7 Window Size for SSEDV and SSEDO.

in the case of high loads. If the system is extremely lightly loaded, e.g., the average CPU and I/O queue lengths are less than one, we expect all algorithms to perform the same. As workload increases, the benefit of increasing window sizes becomes more pronounced. This is because at a higher load, the improvement caused by the *shortest seek time* component of these two window algorithms increases. On the other hand, we also observe that a window size of three or four is good enough in most cases. Increasing the window size further contributes very little to the system performance, but increases the scheduling cost. We also performed experiments for the case of tight deadline settings, and observed similar results.

Figure 5.8 shows the sensitivity performance under SSEDV and SSEDO to changes in α and β, respectively. Intuitively, a larger (smaller) value of α (β) results in a greater bias towards the *shortest seek time* component, and a smaller α (larger β) results in a greater bias towards the *earliest deadline* component of SSEDV (SSEDO). From Figure 5.8 (a), we can see that a large value of α is preferred in a highly loaded system, and a moderate value of α is appropriate in a lightly loaded system. In either case, a range of 0.7 to 0.8 for α is acceptable. A similar observation can be made from Figure 5.8 (b), in which $\beta = 2$ is found appropriate in either high or low workload cases.

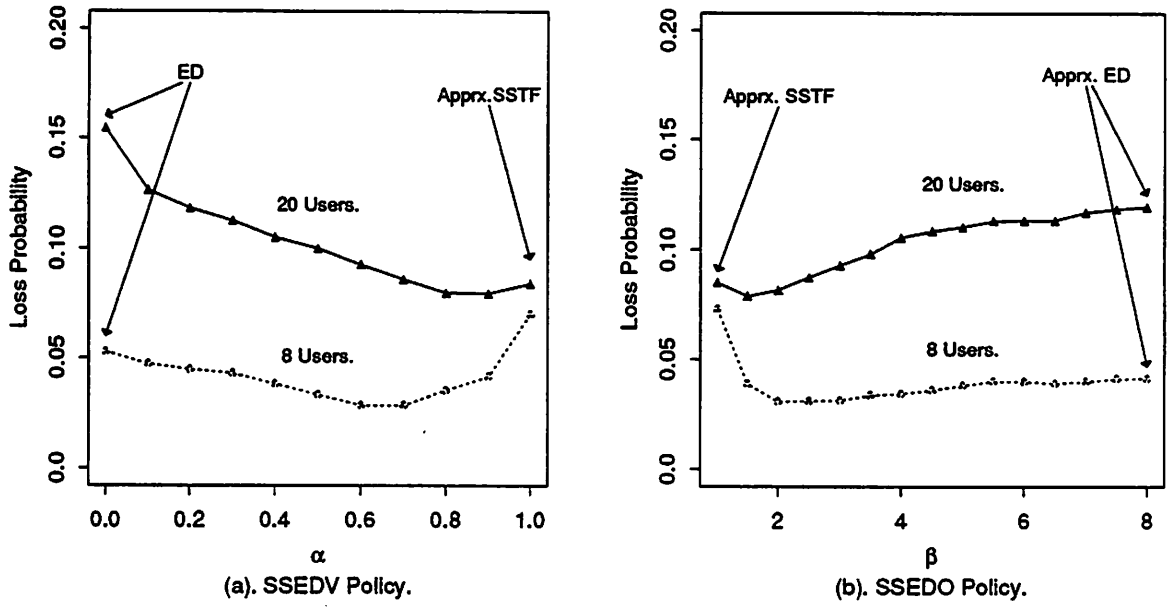
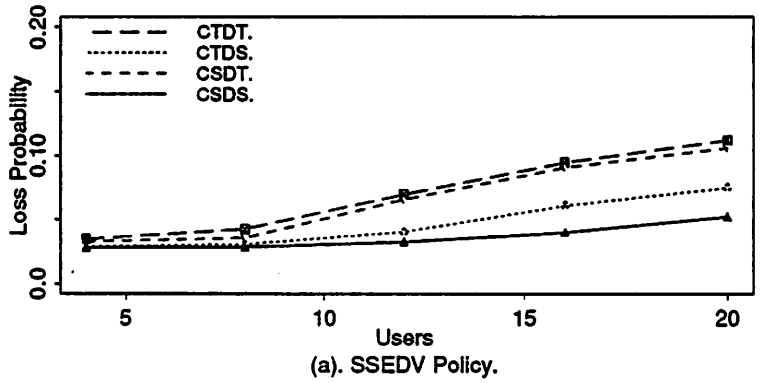


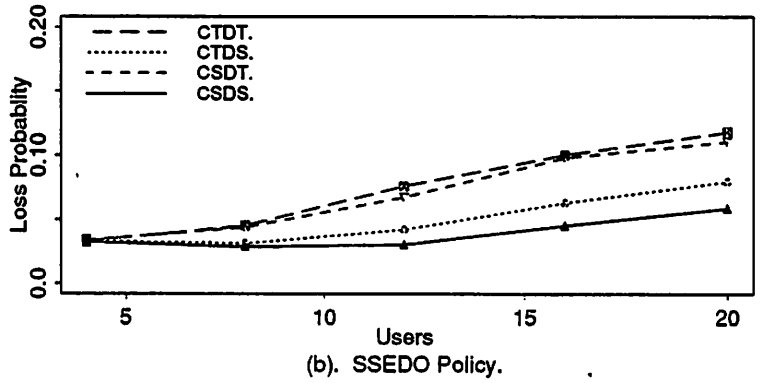
Figure 5.8 Sensitivity of Scheduling Parameters of SSEDV and SSEDO.

Impact of Scheduling by Step Deadlines: As mentioned in Section 5.2.1, if the system has knowledge of the number of steps required by each transaction, it can use this knowledge for CPU and/or I/O scheduling. That is, instead of using transaction deadlines, the scheduler can make its decision by using step deadlines. Recall that the step deadlines are only used to provide priority information for scheduling and that missing a step deadline, except during the last step, does not abort a transaction. Two experiments were conducted to show the effect of scheduling by step deadlines under the SSEDV, SSEDO, and ED algorithms.

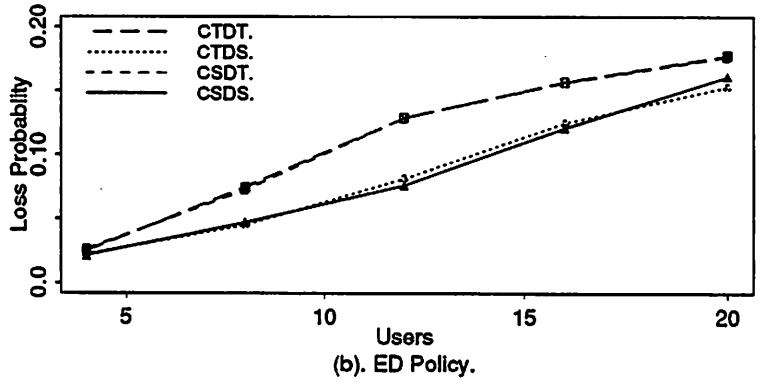
In the first experiment, in which step deadlines are incremented evenly, i.e., calculated by Equation (5.1), we show results for four combinations of scheduling by step deadlines. From Figure 5.9, we observe that the strategy of both CPU and I/O scheduled by step deadlines (CSDS) performs consistently the best under all of the three disk scheduling algorithms. The worst case is when both CPU and disk are scheduled by transaction deadlines (CTDT). The other two alternatives, i.e., CPU scheduled by transaction deadlines and disk by step deadlines (CTDS), and vice versa (CSDT), fall in between. For instance, in the case of high loads, scheduling by



(a). SSEDV Policy.



(b). SSEDO Policy.



(b). ED Policy.

Figure 5.9 Scheduling by Step Deadlines ($Users = 20$).

step deadlines can achieve up to a 53% improvement over scheduling by transaction deadlines under SSEDV. Also, we observe that using step deadlines in I/O scheduling results in proportionately more improvement than using them in CPU scheduling and that in general, the larger the workload, the greater the improvement.

In the second experiment, we fix CPU scheduling by transaction deadlines, but perform I/O scheduling by step deadlines (CTDS). The objective is to examine the effects of different step deadline assignments on performance. One way, as defined in Equation (5.1), is to evenly divide the transaction's deadline among the steps so that each step has the same relative deadline (EVEN). Another way is to let early steps have looser and later steps have relatively tighter step deadlines (ELLT). The motivation for this is that we may expect an almost finished transaction to have a relatively higher priority. A third alternative is to allow early steps to have tighter and later steps to have looser step deadlines (ETLL). In this experiment, we use the following method to assign step deadlines with ELLT and ETLL: let L, a, and n be a transaction's (absolute) deadline, arrival time, and the number of operational steps, as before. In addition, let b denote the *base step length*, and Δ the step increment. For ETLL, the deadline for step i is

$$step\_deadline(i) = step\_deadline(i-1) + b + i\Delta, \qquad i = 1, 2, \cdots, n. \tag{5.2}$$

where $step\_deadline(0) = a$, or

$$step\_deadline(i) = a + ib + \frac{i(i+1)}{2}\Delta, \qquad i = 1, 2, \cdots, n. \tag{5.3}$$

Since the last step's deadline is the same as the transaction's deadline, we have

$$\begin{aligned} step\_deadline(n) &= a + nb + \frac{n(n+1)}{2}\Delta, & (5.4)\\ &= L & (5.5) \end{aligned}$$

or

$$\Delta = \frac{2(L - a - nb)}{n(n+1)}. \tag{5.6}$$

When $b = (L - a)/n$, $\Delta = 0$. Substituting them into Equation (5.3) reproduces Equation (5.1). By changing the value of b, we can adjust the looseness of the step

increment. In this experiment, $b = 2(L - a)/3n$, and Δ is obtained from (5.6). For ELLT, in a similar manner, we have

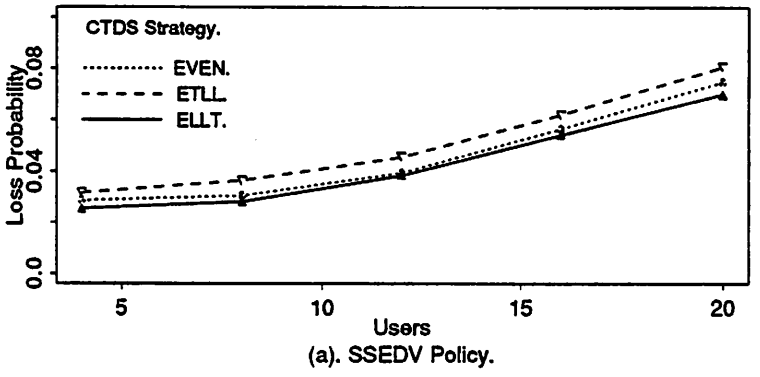
$$\begin{aligned} step\_deadline(i) &= step\_deadline(i-1) + b + (n-i+1)\Delta, && (5.7) \\ &= a + ib + \frac{(n+n-i+1)i}{2}\Delta, \qquad i = 1, 2, \cdots, n && (5.8) \end{aligned}$$

where Δ is given by Equation (5.6).

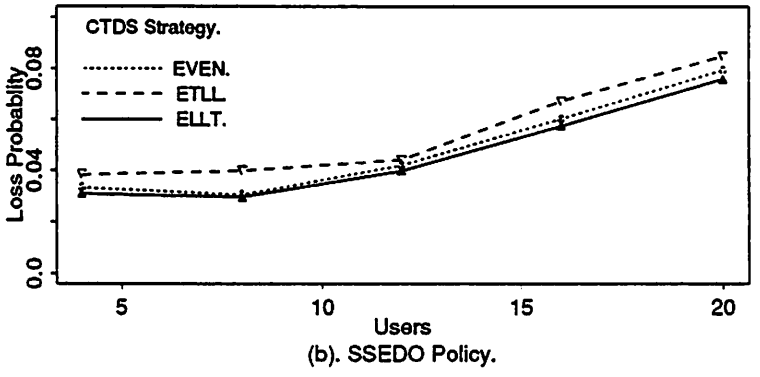
The results are illustrated in Figure 5.10, and we observe that ELLT performs slightly better than the other two deadline assignments under both SSEDV and SSEDO for a variety of workloads. However, in either case, step deadlines should not be too 'loose', otherwise the policies will degenerate to the case in which requests are scheduled according to transaction deadlines which has been shown to be the worst in Figure 5.9. For the ED algorithm, there is no significant differences between various ways of assigning the step deadlines, especially when the I/O load is high.

Varying Transaction's Deadline Settings: This experiment examines the performance of all nine disk scheduling algorithms under various deadline settings, from very loose to extremely tight. At one extreme, every transaction is successfully served and none are lost. At the other extreme, almost every transaction misses its deadline. From the results shown in Figure 5.11, we observe that SSEDV and SSEDO perform the best over the entire range of deadline settings, and that FCFS is the worst.

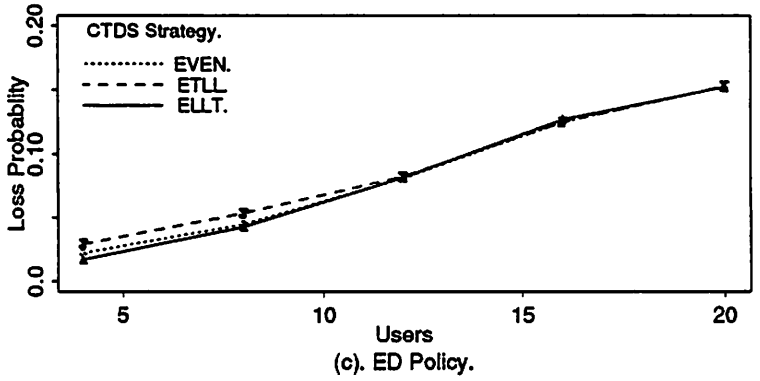
Varying Read Probability: The purpose of this experiment is to study the sensitivity of the performance of various disk scheduling algorithms to changes in the read probability p_r. In our database model, an increase in update probability (i.e., a decrease in read probability) results in more lock conflicts since update requests require exclusive locks. It also results in an increase in I/O load because more flushing of dirty pages is required. The experimental results are shown in Figure 5.12. As expected, the transaction loss probability decreases for all nine I/O scheduling algorithms as the read probability increases. However, the performance ordering of these algorithms remains the same as in previous discussions except for the ED algorithm which has a proportionately greater improvement as the read probability approaches one.



(a). SSEDV Policy.



(b). SSEDO Policy.



(c). ED Policy.

Figure 5.10 Step Deadline Assignment ($Users = 20$).

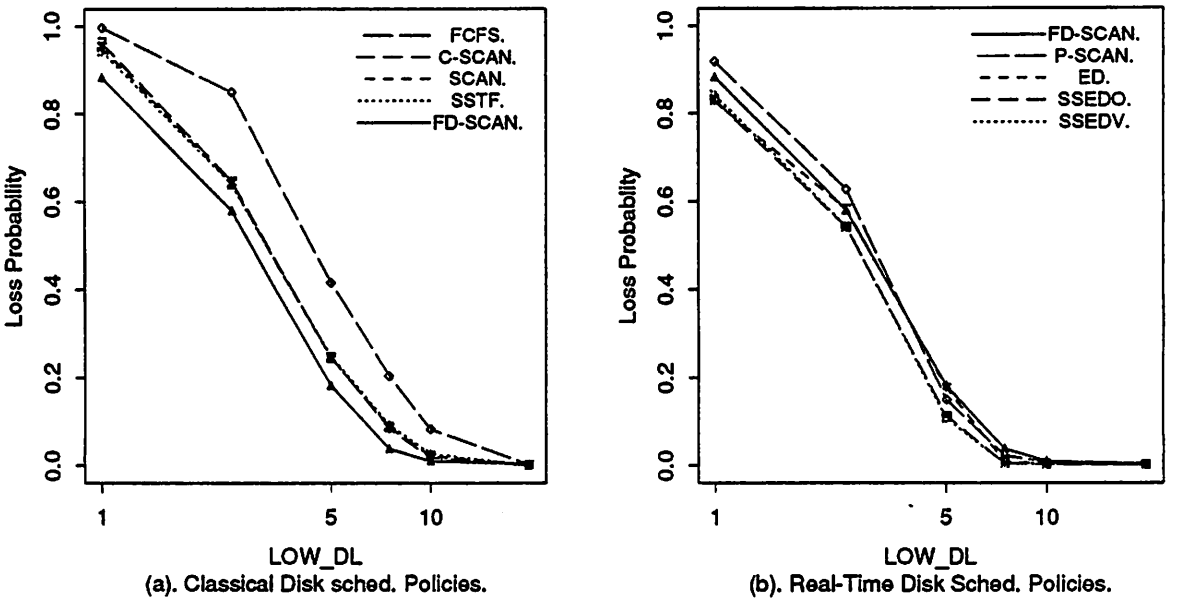


Figure 5.11 Performance with Various Deadline Settings ($Users = 20$).

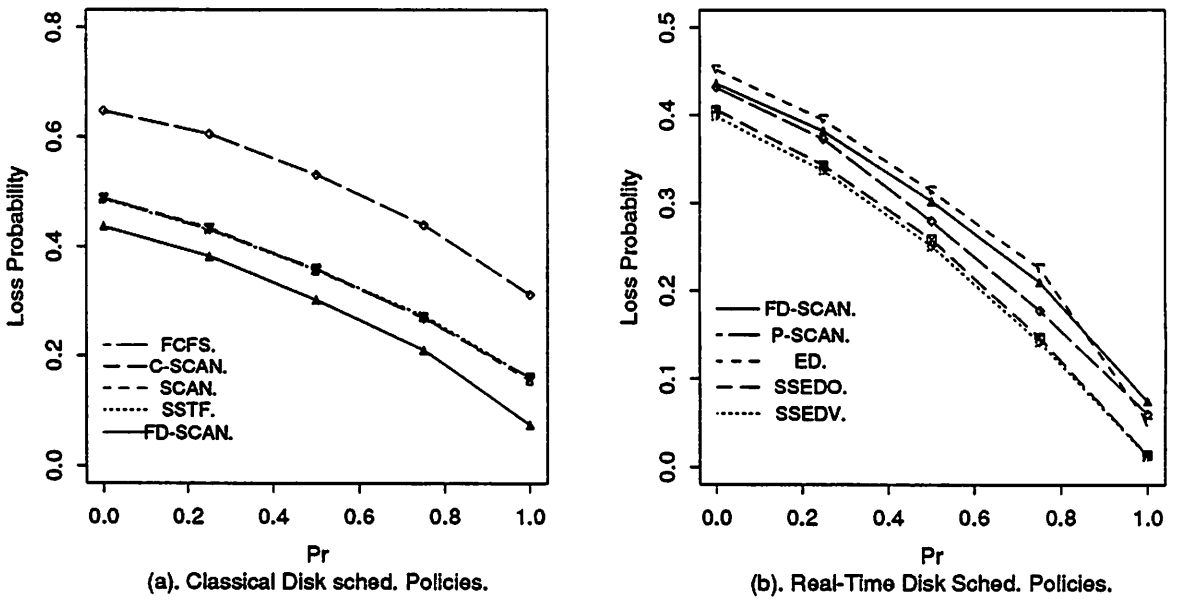


Figure 5.12 Performance under Different Read Probability ($Users = 20$).

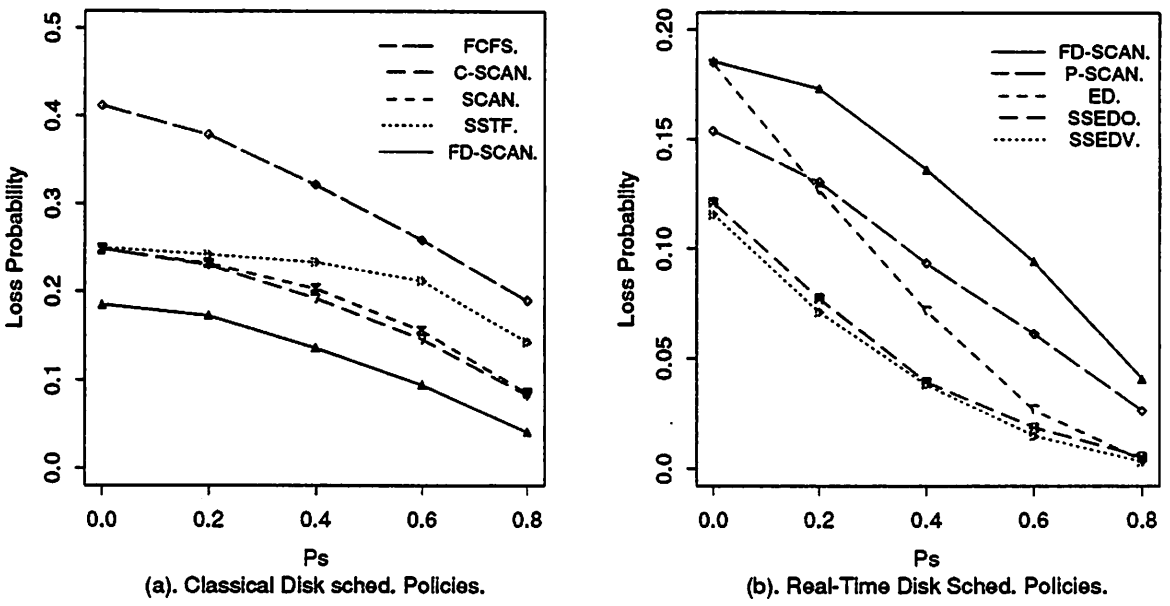


Figure 5.13 Performance under Different Sequential Access Probability ($Users = 20$).

Locality of Transaction Accesses: The purpose of this experiment is to examine the effect of varying access locality on performance. We first investigate the system behavior when the transaction sequential access probability, p_s, varies from 0 to 0.8. In this experiment, we choose a high workload case (20 users), and a read probability of 0.8. From Figure 5.13, we again observe that the performance of all algorithms improve as p_s increases, and that SSEDV and SSEDO are consistently better than the others. Interestingly, SSTF benefits less from an increases in p_s than SCAN and C-SCAN. On the other hand, ED benefits the most when sequential accesses occur. A partial reason for this is that the main benefit gained by increasing p_s is the reduction in seek time, which SSTF has already taken advantage of, but ED has not.

Viewing the locality effect from another perspective, we rearrange the layout of the database on the disk so that the database only occupies a portion of the disk (this situation is found in the RT-CARAT testbed), say the central 100 tracks, rather than randomly scattered over 1000 tracks. The results are shown in Figure 5.14. If we compare these results with Figure 5.3, we can see that the SSEDV, SSEDO, and

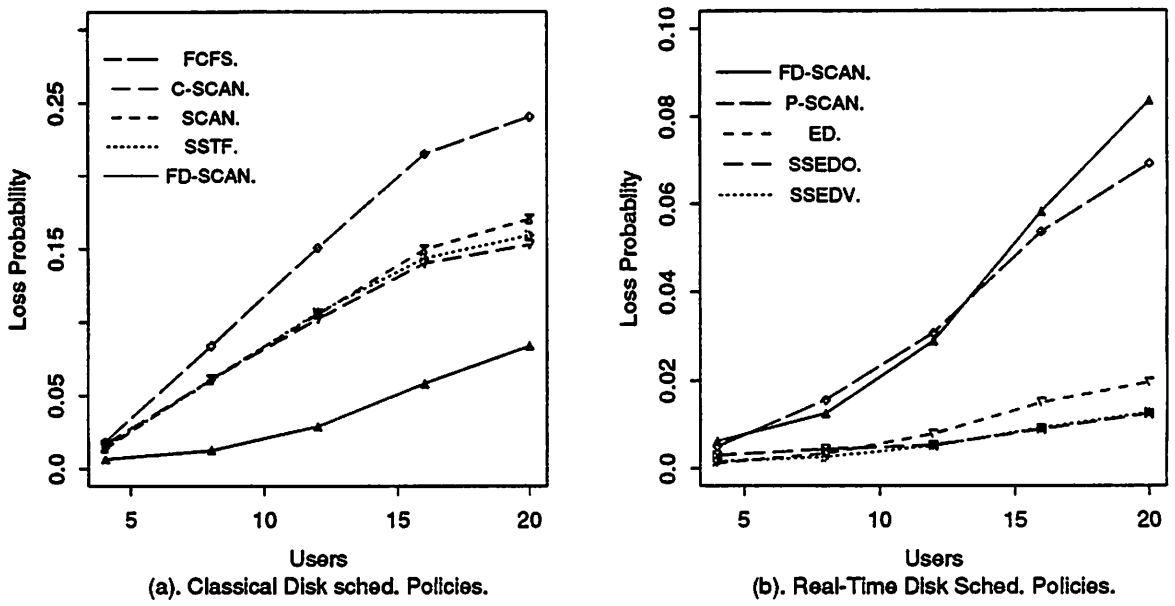


Figure 5.14 Performance under Different Database Layout.

ED algorithms gain more than the others by this arrangement. Among these, the ED algorithm sees tremendous improvement. The reason for this is that both SSEDV, SSEDO and ED are *time-constraint-oriented* algorithms, which means they place more weight on the *time constraint* component rather than on the *disk service time* component as do the various SCAN algorithms. For example, we have observed that SSEDV and SSEDO choose the request with the earliest deadline for next service over 80% of the time, whereas ED chooses that request 100% of the time. The new layout reduces the disk service time. This helps the SSEDV, SSEDO and ED algorithms to make up their deficiency with respect to the *disk service time* component, and therefore improves their relative performance.

Importance of Real-Time Scheduling: This final experiment is designed to see how the performance varies when the system employs real-time or non-real-time scheduling algorithms in CPU and/or disk scheduling. To examine the effect of CPU scheduling, we increase the computation time for each step to 25 msec (which is approximately the same as the mean disk service time). Four combinations, CPU scheduled by FCFS and I/O by SCAN, CPU by FCFS and I/O by SSEDO, CPU by ED and I/O by SCAN, and CPU by ED and I/O by SSEDO, are investigated.

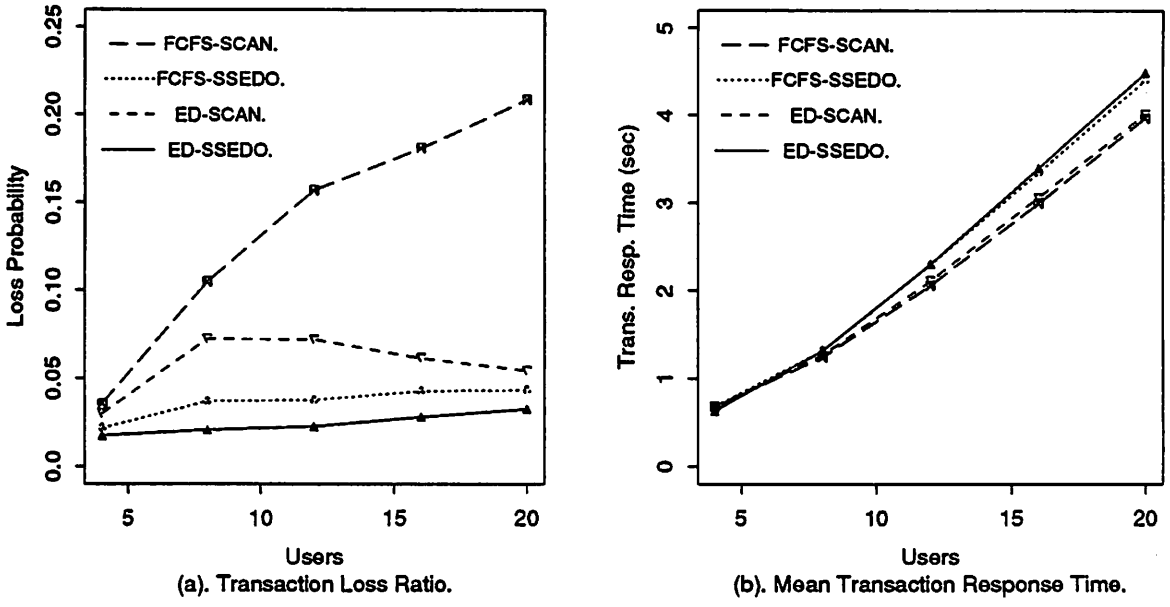


Figure 5.15 Importance of Real-Time Scheduling ($Comp\_Time = 25ms$).

Figure 5.15(a) gives the transaction loss probability in each case. Obviously, when both CPU and I/O scheduled by real-time strategies, the performance is much better than when no real-time algorithms are used for CPU and disk scheduling (with up to a 84% performance improvement in high load cases). We also observe that the use of a real-time disk scheduling algorithm is more beneficial than the use of a real-time CPU scheduling algorithm. The transaction response times for the cases of I/O scheduled by real-time algorithms are shown to be higher than that of I/O scheduled by non-real-time algorithms in Figure 5.15(b). This is because the disk service times for real-time algorithms (SSEDO) are higher than non-real-time algorithms (SCAN). Therefore, we conclude that real-time disk scheduling algorithms do reduce the transaction loss probability at the cost of higher mean transaction response times.

Interestingly, for the two cases of I/O scheduled by non-real-time algorithms, the CPU tends to be a bottleneck as the system load increases (Figure 5.16) under the parameter settings above. In contrast, most transactions will be queued up at the I/O device if SSEDO is employed as the I/O scheduling algorithm. The shorter seek time of the SCAN algorithm partially explains why this occurs.

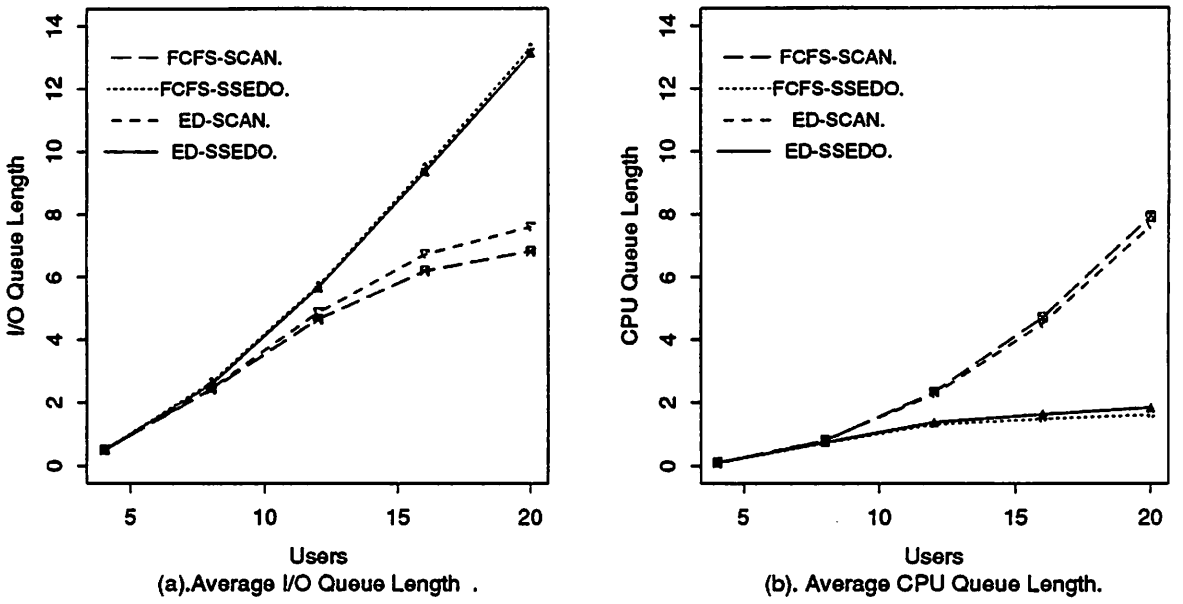


Figure 5.16 Average CPU and I/O Queue Length ($Comp\_Time = 25ms$).

5.3 A Mirrored Disk I/O Subsystem

In this section, we study the behavior of a mirrored disk in a real-time environment. The main purpose is to explore the performance improvement achievable by introducing disk mirroring, as well as by applying real-time algorithms over non-real-time algorithms to a mirrored disk. Readers are referred to [28] for more detailed discussions.

5.3.1 Scheduling Policies for a Mirrored Disk

In Section 3.1, we presented several *policies* for a mirrored disk, which differ from each other in the way that read requests are routed to one of the two disks. On the other hand, in the last section, we examined different *algorithms* for selecting requests from a queue for service in a real-time environment. Combining these *policies* and *algorithms* yields different new scheduling policies for a mirrored disk. Take the *shortest queue* policy as an example. Whenever a read arrives at the mirrored disk, it is directed to the shortest queue, and whenever a disk becomes idle, it selects a request from the corresponding queue according to the algorithm employed, such as SSEDO,

FD-SCAN, or SSTF. Since it is impractical to study all of these combinations, we, instead, select several of them of interest to us and examine their performance.

Non-Real-Time Policies: All of the policies described in Section 3.1 apply here. However, instead of using FIFO, we use SSTF within each queue in our experiments. Among the various distributed policies, since the performance of DP-SQ-MS is close to that of DP-MW-MS and DP-MR and the former is easier to implement, we use DP-SQ-MS in most of our experiments. In the remainder of this subsection, we simply use DP to refer to the DP-SQ-MS policy. The other alternatives will be indicated explicitly when we examine their performance. Similarly, we use CP to refer to CP-AQ-MS to simplify notation.

Real-Time Policies: Corresponding to the above non-real-time policies, we obtain real-time policies RT-DP and RT-CP by applying SSEDO within each queue. We have observed that the auxiliary queue length is typically short under the *centralized policies*. Hence there is no significant difference between different algorithms applied to the auxiliary queue. Other alternatives, such as the FD-SCAN algorithm, will be indicated explicitly whenever used.

Finally, we simply use the names of algorithms, such as SSEDO, to refer to a single disk system which employs the algorithm.

5.3.2 Performance Results

In this subsection, we illustrate the performance benefits gained by applying the previous described real-time scheduling policies on a mirrored disk. In particular, we are interested in the improvement of a mirrored disk over a single disk and of real-time algorithms over non-real-time algorithms. All of these issues are examined in the context of the previously mentioned real-time transaction system model, where we replace the single disk subsystem with a mirrored disk. The system parameter settings for the experiments are the same as those of Section 5.2.2 except for two changes. First, we tightened the deadline settings by reducing the coefficient of deadline upper bound, f, from 4 to 3. Second, we reset the read probability p_r from 0.8 to 0.6 in most of our experiments. This may result in more lock conflict since update requests

require exclusive locks. As a consequence, the transaction loss probabilities fall into a range that allows us to more readily compare the different policies.

In the first experiment, we explore the system performance under different I/O configurations and disk scheduling policies.

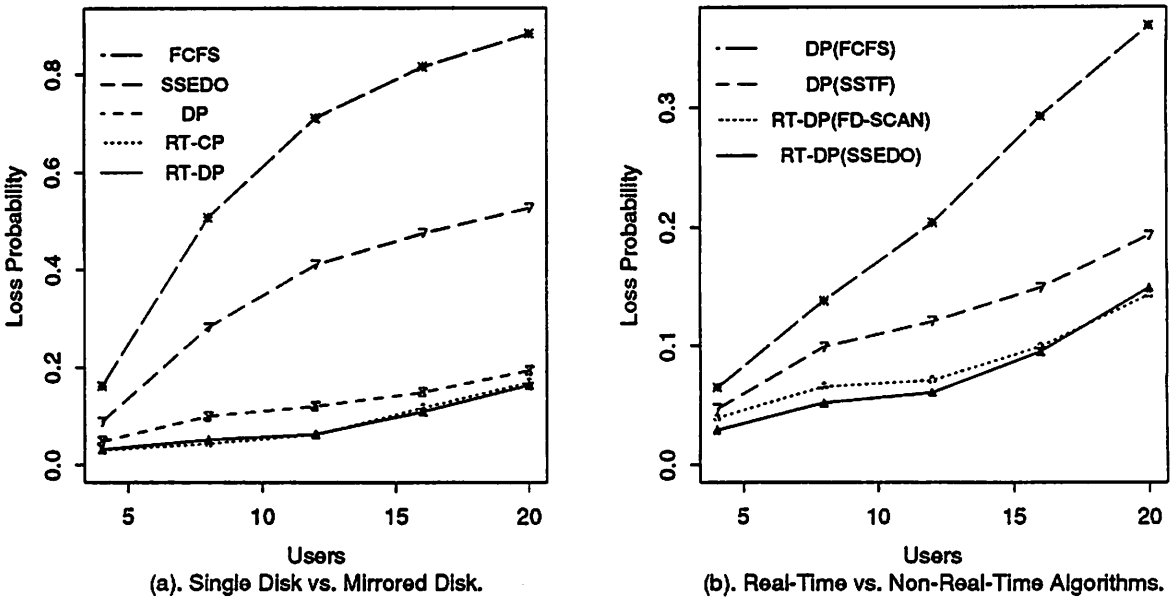


Figure 5.17 Performance of a Mirrored Disk for Real-Time Systems ($p_r = 0.6$).

Single Disk vs. Mirrored Disk: Figure 5.17 (a) illustrates the performance improvement by introducing a mirrored disk. This improvement is due to the fact that the mirrored disk can support concurrent reads which has the potential of increasing the disk bandwidth and decreasing the I/O response times. The improvement is significant; The RT-DP and RT-CP policies can yield up to a 68% improvement over the real-time SSEDO policy and up to a 81% improvement over the FCFS policy for a single disk I/O subsystem. The two real-time policies, RT-DP and RT-CP, provide similar performance.

Real-Time Policies vs. Non-Real-Time Policies: In Figure 5.17 (b), we present the performance of different real-time and non-real-time algorithms in combination with the DP policy. The two non-real-time DP policies apply FCFS and SSTF to each queue, whereas the two real-time policies adopt SSEDO and FD-SCAN, respectively.

From this Figure, we observe the importance of using algorithms that account for real-time constraints on a mirrored disk subsystem. As we can see, DP coupled with FCFS is the worst, since it ignores both time constraints and seek optimization. DP coupled with SSTF is better because it accounts for the seek distances when scheduling requests. RT-DP coupled with SSEDO and FD-SCAN outperform the non-real-time policies, since they consider not only seek distances but also the deadline information of I/O requests in making their decisions. These RT-DP policies differ in the weight placed on the two factors. FD-SCAN places more weight on reducing the service time, while SSEDO places more weight on the time constraints (e.g., it will schedule the request with the earliest deadline most of the time). The curves show that the SSEDO variant performs slightly better than the FD-SCAN variant with the only exception being at an extremely high workload. Henceforth, RT-DP will refer to the RT-DP coupled with SSEDO.

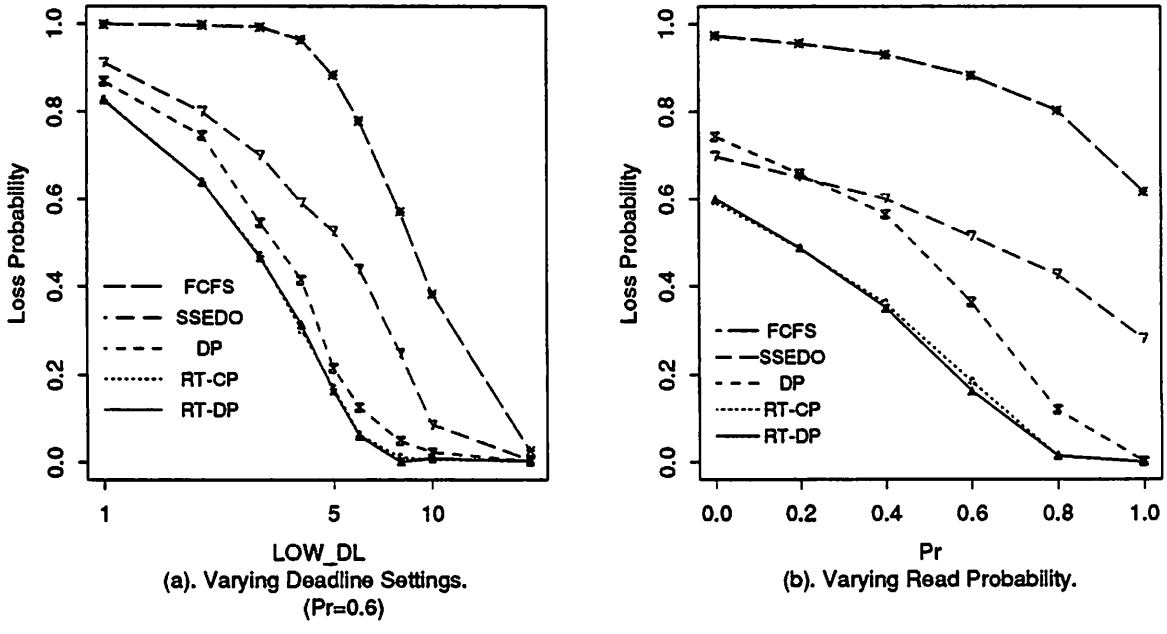


Figure 5.18 Alternate Parameter Settings for a Mirrored Disk ($Users = 20$).

Varying Transaction Deadline Settings: This experiment is designed to examine the effects of the deadline settings on the performance of different disk scheduling policies. At one extreme the deadline is loose so that every transaction is successfully

served and none are lost. At the other extreme, the deadlines are tight so that most transactions miss their deadlines. From the results shown in Figure 5.18 (a), we observe that a mirrored disk subsystem outperforms a single disk subsystem, and that the RT-DP and RT-CP policies perform the best over the entire range of deadline settings.

Varying the Read Probability: In this experiment, we study how the various disk scheduling policies behave as a function of the read probability, p_r. As indicated before, in our database model, an increase in update probability (i.e., a decrease in read probability) results in more lock conflicts since update requests require exclusive locks. This will result in more transactions being blocked while waiting for locks, and thus more deadlocks, which in turn causes more transactions to be aborted. The increase in update probability also results in an increase in I/O load because more writes occur at the end of transaction execution. Both of these phenomena result in a decrease in the loss probability as p_r increases. The experimental results are shown in Figure 5.18 (b). The transaction loss probability decreases for all of the I/O scheduling policies as p_r increases. However, the mirrored disk subsystems remain more beneficial than single disk subsystems even as p_r approaches 1. This is again due to the presence of concurrent reads provided by the mirrored disk subsystem.

5.4 Disk Array I/O Subsystems

In Chapter 4, we studied disk arrays for conventional systems. Here, we briefly examine disk arrays in a real-time environment. The main purpose is to compare different disk array architectures coupled with real-time as well as non-real-time algorithms. Unlike the previous two sections, in which a single disk and a mirrored disk are examined in a closed real-time transaction system model, in this section, we study disk arrays in an open system model, in which I/O requests arrive to the disk array subsystem according to a Poisson process and each request carries a time constraint. The reasons for doing so are two-fold. First, we are trying to avoid the high computational cost required to generate enough I/O requests to saturate the disk array by using the integrated transaction system model. Second, the open system model is of independent interest, and is simple to examine. The open system

model will also be used in the next section when we study disk I/O for non-removal real-time systems.

5.4.1 Disk Array Architectures

The disk arrays considered in this subsection are RAID 1 (*mirrored array*) and RAID 5 (*parity rotate array*), where the DP-SQ-MS policy (see Section 3.1) is used for RAID 1. Two versions are examined for each of the RAID 1 and RAID 5 architectures. In the real-time version, the SSEDO algorithm is applied to each queue, whereas in the non-real-time version, the SSTF algorithm is applied to each queue. To simplify notation, we use RAID 1 to refer to the real-time version of RAID 1, and RAID 1 (Non-RT) to the non-real-time version. The same conventions hold for RAID 5.

5.4.2 Workloads and Deadline Settings

The I/O requests are assumed to arrive to the subsystems according to a Poisson process with rate λ. Each request accesses a single block of data (4096 bytes). The time constraint carried by each request is obtained as follows: let $Mean\_Access\_Time$ denote the mean disk access time for a request in a non-real-time RAID architecture which serves I/O requests in a FCFS manner (see Equation (2.8) for the calculation of $Mean\_Access\_Time$); then the deadline for each request is

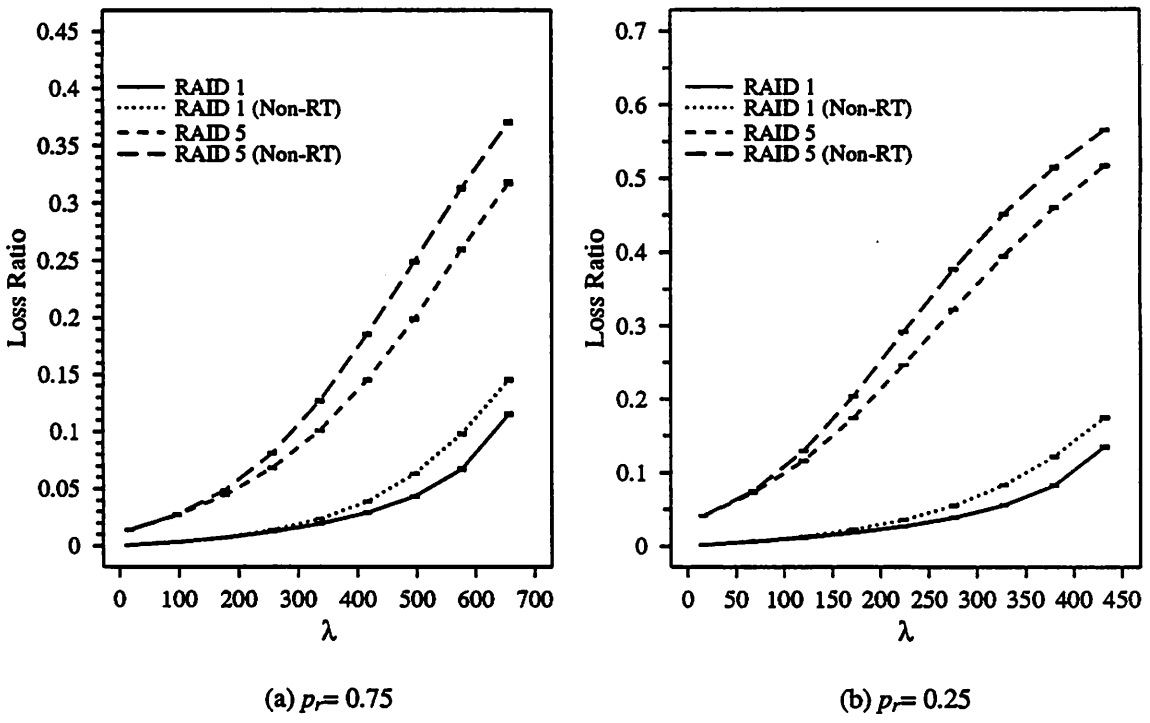
$$Deadline = Arrival\_Time + Mean\_Access\_Time + Slack\_Time$$

where $Slack\_Time$ is chosen uniformly from the range $[Min\_Slack, Max\_Slack]$. Here Min_Slack is fixed at 10 milliseconds. While most of the time Max_Slack is set to be 300 milliseconds, it will also be varied in one of our experiments.

5.4.3 Performance Results

All of the results reported here are obtained via simulation. Each experimental result is averaged over 40 runs, and within each run, 50,000 I/O requests are served. In some runs where the deadline setting is loose, each run includes 60,000 I/O requests. We obtain 95% confidence intervals which are less than 1% of the values of the point estimates. There is no overlap in the confidence intervals except at low loads.

In the first experiment, illustrated in Figure 5.19, we study the behavior of both the non-real-time and real-time RAID architectures in the case that all requests are for a single block. The disk arrays contain 16 disks. The probability of requests that miss their deadlines is given as a function of the request arrival rate. Two sets of curves are given; one for a workload consisting primarily of read requests ($p_r = 0.75$), and the other for a workload consisting primarily of write requests ($p_r = 0.25$). We observe that there is considerable difference in the performance of the different RAID architectures with RAID 1 producing the best performance. Furthermore, the addition of the SSEDO real-time algorithm provides an additional improvement (up to 20%) over the performance of non-real-time RAID architectures.

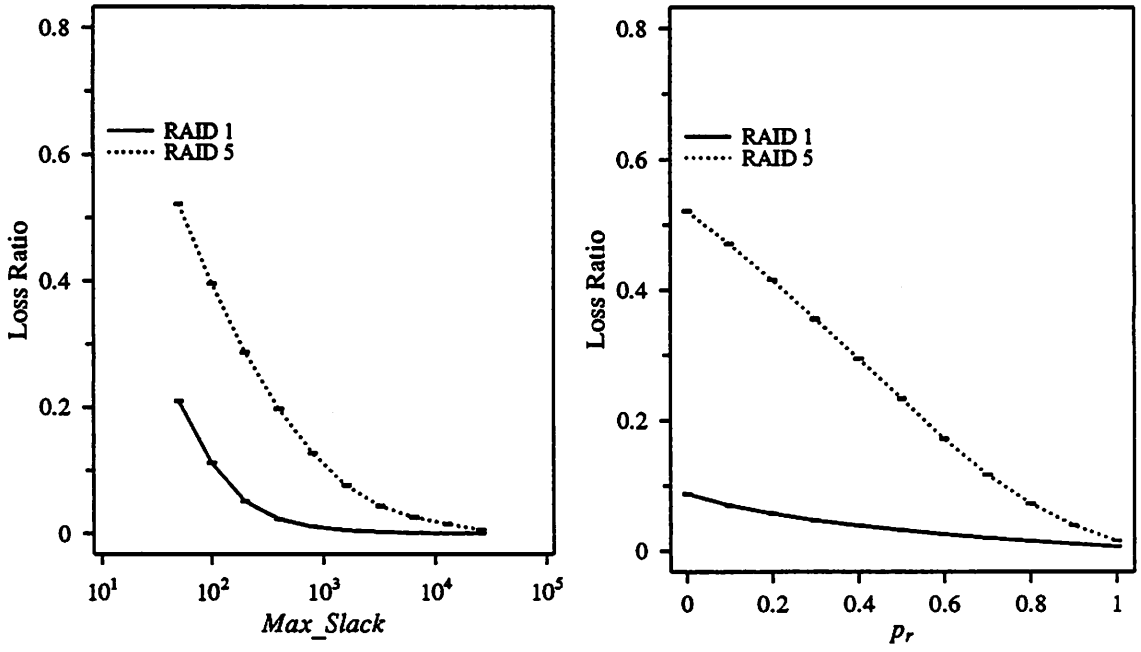


(a) p_r= 0.75 (b) p_r= 0.25

Figure 5.19 Performance of RAID 1 and RAID 5 for Real-Time Disk I/O ($N = 16$).

The second experiment examines the sensitivity of our results to the deadline distribution. Recall that deadline slack times are uniformly distributed in the interval $[Min\_Slack, MaxSlack]$ where $Min\_Slack = 10$, and $Max\_Slack = 300$. In this experiment, we fix the arrival rate $\lambda = 320$ (a median workload) and vary Max_Slack

from 50 to 25,600 (msec) so that the deadline settings change from very tight to very loose. From Figure 5.20 (a), RAID 1 is observed to perform consistently better than RAID 5 for all deadline settings.



(a) Varying Deadline Settings (p_r=0.5)

(b) Varying Read Probability

Figure 5.20 Different Parameter Settings for Disk Arrays ($\lambda = 320$, $N = 16$).

Finally, Figure 5.20 (b) shows the results for different read probabilities. From this Figure, we observe that when read probability decreases, RAID 5 performs significantly worse than RAID 1. This is, again, due to the high cost of "small" writes inherent in the RAID 5 architecture.

5.5 Disk I/O for Non-Removal Real-Time Systems

In this section, we study the behavior of disk I/O for non-removal real-time systems, and explore the relationship between removal and non-removal systems. In a non-removal real-time system, all customers have to be served to completion, even though some of them have already missed their deadlines. Unlike previous studies on non-removal real-time systems, in which the main goal was to reduce the customer

lateness and tardiness [113, 6, 18, 93], we are interested in scheduling policies which minimize the fraction of customers (I/O requests) which are lost.

In particular, we propose a useful paradigm for transferring policies or algorithms from removal systems to non-removal systems. By using this paradigm, in addition to allowing us to build policies for non-removal systems, we further conjecture that the quality of different policies shown in a removal system is preserved when they are ported to a non-removal system. This is of interest because considerable effort has been spent on removal real-time systems during past several years [58, 118, 53, 90]. Consequently, if the conjecture is true, then results obtained for removal real-time systems should also hold for non-removal systems. Therefore the simple paradigm allows us to study the behavior of non-removal real-time systems with less effort. There is yet another simple way to transfer policies from removal systems to non-removal systems, i.e., each policy serves I/O requests according to the same scheduling rule but ceases to throw away customers. We refer to this approach as *direct mapping.* We expect that using the paradigm above may result in a performance improvement over the direct mapping.

In this section, we first define the paradigm more precisely, and then apply it to several real-time disk scheduling policies[2] which have been previously studied in a removal system environment. We then examine the performance of the resulting policies in a non-removal system environment. As a result, we are able to show empirically that: (1) the performance ordering of different policies is the same as that of in a removal real-time system; and (2) policies obtained by adopting the paradigm outperform those obtained by direct mapping.

5.5.1 The Model and the Paradigm

The Non-Removal System Model: The non-removal real-time system model we are interested in is illustrated in Figure 5.21. In this model, customers arrive to the system according to a Poisson process. Associated with each customer is a relative deadline, which is drawn from an arbitrary distribution and is independent

[2]Since we are only considering a single disk system in this section, we will use the term scheduling policy or algorithm interchangeably.

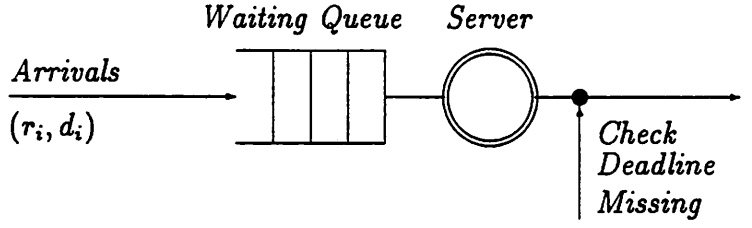


Figure 5.21 The Model for Non-Removal Real-Time Systems.

of the arrival process. The absolute deadline of a customer is the sum of its relative deadline and its arrival time and is the time by which the customer is expected to complete service. In the following, deadline will be used to refer to absolute deadline except when otherwise indicated. After arrival, a customer joins a queue waiting to be scheduled according to the scheduling policy employed. Service times are assumed to form a sequence of identically distributed *r.v.*'s independent of the arrival process and deadline distribution. In this model, all customers have to be served eventually, even though some of them have already missed their deadlines. When a customer finishes service and leaves the system, its completion time is checked against its deadline to see if it has been lost. Our goal is to minimize the fraction of customers which have missed their deadlines when they leave the system.

The Paradigm: In this paragraph, we describe a simple but useful paradigm for transferring policies used in removal systems to non-removal systems. Formally, let Σ be the class of scheduling rules by which customers are selected for service, and Γ be the class of removal rules by which customers are removed from the system before being served. Then, a policy π for a removal system can be defined as a couple,

$$\pi = (\sigma, \gamma)$$

where $\sigma \in \Sigma$ and $\gamma \in \Gamma$. Since there are no customers being thrown away in a non-removal system, policies there consist of only the scheduling rule component. The paradigm is then defined as a function,

$$\Phi : \Sigma \times \Gamma \rightarrow \Sigma.$$

For any policy π in a removal system, we can obtain a policy $\Phi(\pi)$ for non-removal systems. $\Phi(\pi)$ schedules the same as π except that whenever π throws away a customer, $\Phi(\pi)$ moves the customer into a second low priority queue. When π faces a empty system, $\Phi(\pi)$ switches to the secondary queue and schedules customers there in an arbitrary order (e.g., FCFS). Observe that most policies for removal systems are directly applicable to non-removal systems by following the same scheduling rule but without removing customers. We identify policies obtained in this way as results of a direct mapping Λ,

$$\Lambda : \Sigma \times \Gamma \rightarrow \Sigma.$$

A particular removal rule of interest is the *deadline-miss rule* (*d-m* rule), which removes a customer at the time that misses its deadline. There are two variations of the *d-m* rule, depending on whether the system allows a customer to be aborted during service or not,

m : which removes a customer anytime that it misses its deadline, regardless of whether it is in service or not, given the system allows abortion;

m' : which removes a customer anytime that it misses its deadline and it is waiting in the queue. Customers always complete service once they start it.

For example, there are two variations of the ED policy, $(\underline{ed}, m)$ and $(\underline{ed}, m')$, which schedule the customer with the earliest deadline but differ from each other by the removal rule. By applying direct mapping, the resulting policy for a non-removal system, Λ(ED) will not throw away customers but will still follow the ED rule for scheduling. However, Φ(ED) moves all missed customers to a secondary queue, and schedules customers in the primary queue according to the ED rule. Only when the primary queue becomes empty are customers from the secondary queue allowed to receive service.

We conjecture that this paradigm exhibits the following properties:

1. for any policies π and π' for a removal system such that $\pi \leq \pi'$, then $\Phi(\pi) \leq \Phi(\pi')$;

2. for any policy π for a removal system, $\Lambda(\pi) \leq \Phi(\pi)$;

where by $\pi \leq \pi'$, we mean policy π' performs better than policy π in terms of minimizing the loss probability. The first property is interesting, since if it is true, then previous valuable results obtained for removal systems will also hold for non-removal systems. This can help us to understand the behavior of non-removal real-time systems with little effort. The second property provides us a mechanism to obtain good policies for non-removal systems. In [30], we have tried to theoretically justify our conjectures, where, parallel to the optimality of ED in removal systems under certain conditions, we were able to show that, for a non-removal system,

- if the deadline is to the beginning of service, service times are a sequence of *i.i.d.* *r.v.*'s independent of the arrival times and deadlines, and there is no service time information available to the scheduler for decision making, then Φ(ED) stochastically minimizes the customer loss ratio;

  Remark: The counterpart of this results for removal systems requires the exponential service times condition. Here, the service times can be arbitrarily distributed.

- if the deadline is to the end of service, service times are *i.i.d.* exponential *r.v.*'s independent of arrival times and deadlines, the scheduler has no any service time information, and abortion of a customer during service is allowed, then Φ(ED) is optimal in terms of minimizing the customer loss ratio.

In this section, we study the paradigm in a real-time disk I/O context, in which the above mentioned conditions do not hold. In the following discussions, we will omit the notation Λ for direct mapping whenever there is no confusion. For example, when we discuss the ED policy in a non-removal environment, we mean the Λ(ED) policy.

5.5.2 Policies for Non-Removal Real-Time Disk I/O Subsystems

In this subsection, we study several policies for non-removal real-time disk I/O subsystems. The non-removal system model is shown in Figure 5.21, which was first used by Abbott and Garcia-Molina [3] in examining real-time disk scheduling. In this context, we assume that each I/O request arriving to the disk I/O subsystem carries

a real-time constraint. If a request cannot finish the disk access before missing its deadline, it is said to be lost. The I/O subsystem will serve every request even though its deadline has expired. The same goal as above is to be achieved, i.e., minimizing the fraction of requests which miss their deadlines. In the following, we first describe the concept of a *feasibility check*, and then obtain policies by using the paradigm described above.

Feasibility Check: Because of the availability of disk service time information, we can use this information to avoid scheduling a request which may be lost during service. In particular, we will use the definition introduced in [3], i.e., a deadline is *feasible* if it is greater than the current time plus the potential service time which consists of seek time, rotational latency, and the transfer time[3]. While Abbott and Garcia-Molina [3] assumed the latency and transfer time to be a constant, we assume that the latency is uniformly distributed in the range of $[0, 16.7]$ milliseconds. Consequently, we use the average latency to estimate the feasibility of a deadline. Other more conservative or radical strategies can be obtained by using the maximum or minimum latency to estimate the feasibility.

An even more sophisticated strategy is also possible given the availability of hardware similar to RPS (*rotational position sensing*) to sense the arm position under rotation. The seek time is first calculated followed by the calculation of the rotational latency to the destination block. Observe that it is hard to obtain a precise latency because of the time needed for the calculations. However, since the calculations are performed more quickly than the mechanical actions of the disk, the resulting latency can be quite accurate. This strategy is useful if the rotational latency is dominate the disk access time as suggested in [83], but requires a more intelligent disk controller and driver. We will further investigate these alternatives later in this section.

It is interesting to observe that the inclusion of the feasibility check (which is specific to the disk scheduling context) introduces a new removal rule, called the *infeasible rule* $f \in \Sigma$, which throws away a request if its deadline is found to be infeasible even though it is still valid. Finally, in most cases, we assume that disk I/O

[3]We assume that each disk has its own data path and therefore the channel delay is ignored.

service is non-interruptable, i.e., once a request starts a disk access, it will go through to the end. In this case, the *deadline-miss* removal rule is m'. We will also consider the case in which preemption or abortion of a request which misses its deadline during service is allowed, which implies the m as the *deadline-miss* rule.

Policies for Non-Removal Real-Time Disk I/O Subsystems: In the previous sections, we studied several real-time disk scheduling policies for removal systems. Here we obtain the corresponding policies for non-removal systems by applying the paradigm Φ, or simply by applying the direct mapping Λ. The removal rules of interest are the *deadline-miss* rule m' and the *infeasible* rule f. Thus, for each scheduling rule σ, we have two policies for the removal model, (σ, m') and (σ, f). In order to simplify notation, in the following we will express policy (σ, m') by σ and (σ, f) by σ-F. The policies for non-removal systems are then immediately obtained by applying the paradigm Φ and mapping Λ to σ and σ-F.

Take the earliest deadline scheduling rule as an example, the policy ED = $(\underline{ed}, m')$ schedules a request with the smallest deadline and throws away a request if its deadline has expired. The policy ED-F = $(\underline{ed}, f)$ also schedules requests according to the ED rule but throws away a request if its deadline is found to be infeasible. The corresponding policies for non-removal systems are obtained as,

Λ(ED) (or simply ED): which always schedules the request with the earliest deadline for service even though its deadline has expired;

Φ(ED) : which selects a request with the smallest deadline from a primary queue. If its deadline has expired, the request is transferred to a secondary low priority queue;

Φ(ED-F) : which behaves the same as Φ(ED) except that it transfers a request to the secondary low priority queue if its deadline is infeasible. In other word, it adds a feasibility check to Φ(ED).

While a request scheduled by Φ(ED) may or may not be lost during service, it is most unlikely to be lost under Φ(ED-F), i.e., Φ(ED-F) uses the service time information to

predict the possibility of losing in service. The secondary low priority queue required by the paradigm can be scheduled by SSTF or SCAN to reduce service times.

Last, the policies we are going to examine in this section include ED, SSEDV, FD-SCAN, and SSTF.

5.5.3 Performance Comparisons

The performance results of those policies discussed above are reported here. Since the *infeasible* removal rule f uses more knowledge than the *deadline-miss* rule m' for decision making, the f-rule is expected to perform better than the m'-rule, and therefore we are more interested in the four policies, Φ(ED-F), Φ(SSEDV-F), Φ(FD-SCAN-F), and Φ(SSTF-F). The FD-SCAN policy is provided for the purpose of comparison. All of the results are obtained via simulation experiments as described in Section 5.4.3. The results show that using the paradigm Φ and the deadline feasibility check can significantly reduce the request loss probability. Among all of these policies, Φ(SSEDV-F) is observed to be the best, and Φ(ED-F) performs reasonably well. When preemptions are allowed, we observe that the strategy which allows a new arrival to preempt the request in service only when it has been lost can lead to some performance improvement. Finally, as a complement to our previous work, we revisit some of these policies in a removal real-time system model.

Simulation Parameter Settings: In order to provide comparisons, most of the parameters used in the simulation model are taken from [3]. These parameters are summarized in Table 5.3. The arrival process is assumed to be Poisson with rate λ, where λ varies from 22 to 40 to represent different workloads. The deadline of each request is obtained in the same way as described in Section 5.4.2, i.e.,

$$Deadline = Arrival\_Time + Average\_Access\_Time + Slack\_Time$$

where $Slack\_Time$ is chosen uniformly from the range $[Min\_Slack, Max\_Slack]$. While most of time the Max_Slack is set to 100 ms, we also vary it from 50 ms to 500 ms. Under these parameter settings, the disk utilization varies from 55% to 95%. For Φ(SSEDV-F), the default window size and scheduling parameter α are set to 3 and 0.1, respectively.

Table 5.3 Simulation Parameters for the Non-Removal System Model

Num_Tracks (C)	1000
Acce_Time (a)	5 ms
Seek_Factor (b)	0.6
Latency (R)	Uniform in $[0, 16.7]$
Transfer (T)	1.5 ms
Avg_Access_Time	25 ms
Min_Slack	10 ms
Max_Slack	100 ms

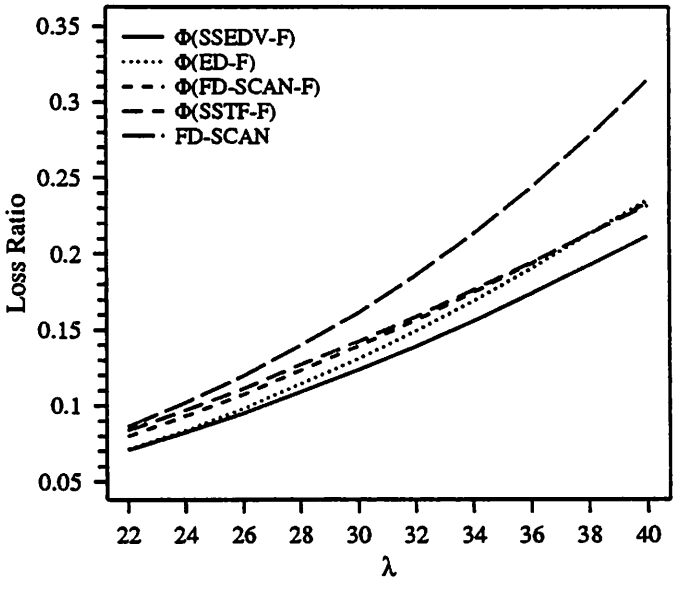
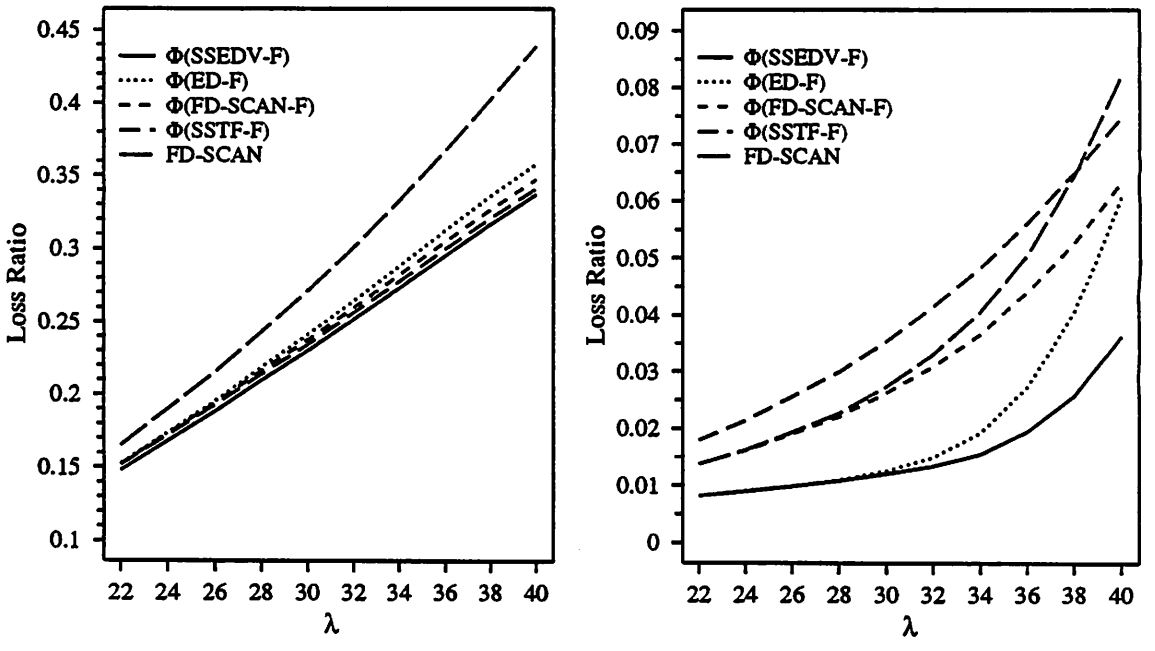


Figure 5.22 Performance for Non-Removal Real-Time System Model.

In the following, we report the results of our simulation experiments.

Loss Ratio under Different Polices: Figure 5.22 shows the performance of various scheduling policies in our real-time disk I/O model. We observe that Φ(SSEDV-F) is the best, outperforming the second best policy, Φ(ED-F), by up to 10%. This is because Φ(SSEDV-F) combines the time constraint factor and the reduction of disk access time while Φ(ED-F) only does the former. It is interesting to observe that Φ(FD-SCAN-F), which avoids serving infeasible requests along the way of scanning, achieves up to a 25% improvement over the original FD-SCAN. For this set of deadline settings (Max_Slack = 100 ms), Φ(FD-SCAN-F) and Φ(SSTF-F) are observed to perform slightly worse than Φ(ED-F). Both of Φ(FD-SCAN-F) and Φ(SSTF-F) are characterized by placing more weight on reducing the service time than on accounting for time constraints.

In the following, we will study the performance of these policies under different deadline settings.



(a). Tight Deadline Setting. [10,50]

(b). Loose Deadline Setting. [10,500]

Figure 5.23 Varying Deadline Settings for Non-Removal Systems.

Varying Deadline Settings: In this experiment, we vary deadline settings from extremely tight to quite loose. In particular, we maintain *Min_Slack* at 10 ms while varying *Max_Slack* from 50 ms to 500 ms. Figure 5.23 shows the results for the two extreme cases. When *Max_Slack* equals 200 ms and 300 ms, the behavior of these policies differs little from the case *Max_Slack* = 500 ms (Figure 5.23 (b)). When deadlines are extremely tight, all of these policies exhibit almost identical performance, with Φ(SSEDV-F) observed outperforming Φ(SSTF-F) and Φ(FD-SCAN-F) only by 1% and 3% respectively (Figure 5.23 (a)). However, when deadlines become loose, Φ(SSEDV-F) may achieve up to a 40% performance improvement over all other policies (Figure 5.23 (b)). One lesson we learn from this experiment is that in an environment where deadlines are extremely tight, reducing disk access time is more important than accounting for I/O request's time constraint. But the reverse is true when deadlines are not very tight. Since rarely is a system designed to work in an environment with extremely tight deadlines, we conclude that the design of good policies should place more weight on the time constraint factor.

Alternate Ways for Feasibility Check: From discussions above, we see that using disk service time information to perform a feasibility check when scheduling may result in a considerable performance improvement. While our previous discussions use the average latency (8.3 ms) for feasibility check, here we consider other alternatives, maximum latency (16.7 ms), minimum latency (0 ms), as well as the ideal situation – knowing the exact latency to serve a request. The results are illustrated in Figure 5.24. The *maximum latency strategy* is not good because it overestimates the potential disk access time. This may result in some requests being rejected for service which in fact can make their deadlines. On the other hand, the *minimum latency strategy* underestimates the potential access time and results in more requests being lost during service. In the ideal case, up to a 10% improvement can be achieved over the *average strategy*, which provides us an upper bound for the potential improvement. Since the precise latency calculation requires more complicated hardware support, there is a performance/cost trade-off.

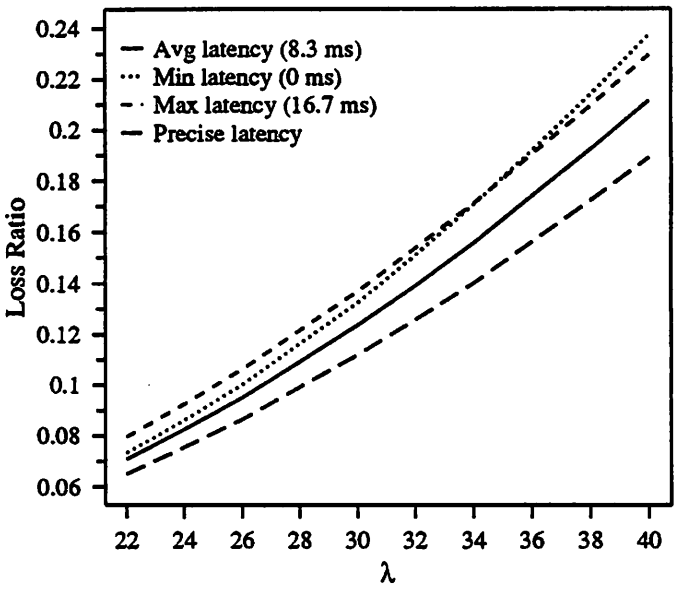
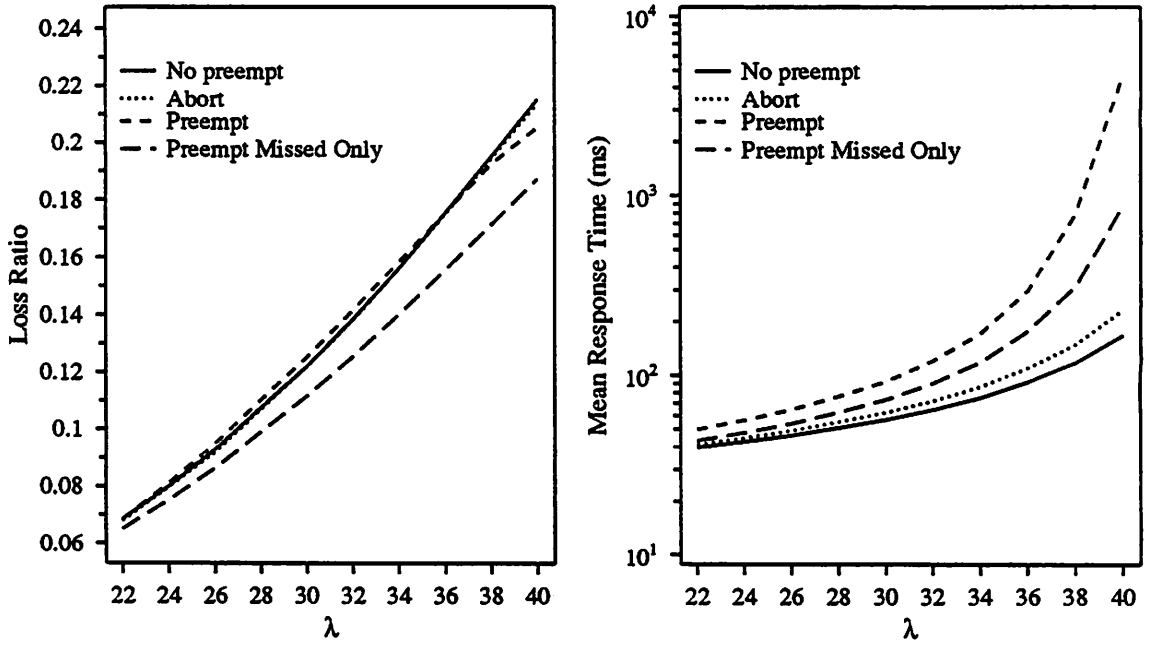


Figure 5.24 Alternate Strategies for Feasibility Checking under Φ(SSEDV-F).



(a). Request Loss Ratio.

(b). Mean I/O Response Time.

Figure 5.25 Preemption vs. Non-Preemption (Φ(SSEDV-F)).

Preemptions vs. Non-Preemptions: While our discussions above assume disk service to be non-interruptable, here we relax the restriction and allow preemptions and/or abortions to occur during service. Notice that in a non-removal model, if a request is preempted (aborted) while in service, it must be rescheduled later on, and there is no way for it to resume from the point of preemption. Hence, in addition to requiring a more sophisticated device controller and driver, preemptions also create more workloads to the system. Therefore, there is a trade-off between the cost paid and the benefit gained (if any) by allowing preemptions. In the following, we identify three strategies for allowing preemptions (abortions), and compare their performance against non-preemptive policies. In doing so, we ignore the extra overhead on device controller and driver.

- The first strategy, called A, is in fact non-preemptive, but the controller is intelligent enough to abort a request in service if its deadline expires and at the same time there are some requests with valid deadlines waiting to be served.
- The second strategy, in addition to abortions as in A, allows a new arrival to preempt the one in service if its deadline has expired. Under this strategy, when a preemption occurs, all of the requests waiting to be served must have missed their deadlines. We refer to this strategy as *preempt missed only* (PMO).
- The third strategy, called P, differs from strategy PMO by allowing a new arrival to preempt the one in service if it has been lost or the new arrival has a smaller deadline.

The effects of these strategies are shown in Figure 5.25 (a) for the Φ(SSEDV-F) policy. Similar behavior has been observed for other policies. Strategy A achieves no gains because Φ(SSEDV-F) effectively avoids the occurrence of requests missing deadlines during service. Strategy P performs basically the same as the non-preemptive strategy, but produces incredibly higher mean response times (Figure 5.25 (b)). This is because strategy P will create too much extra burden on the system. It is, therefore, certainly not a good strategy to pursue. The only strategy which noticeably improves

performance is PMO, which may lead up to a 13% improvement in loss ratio at the cost of higher mean response times (especially for those missed requests) and of more complicated hardware and software support.

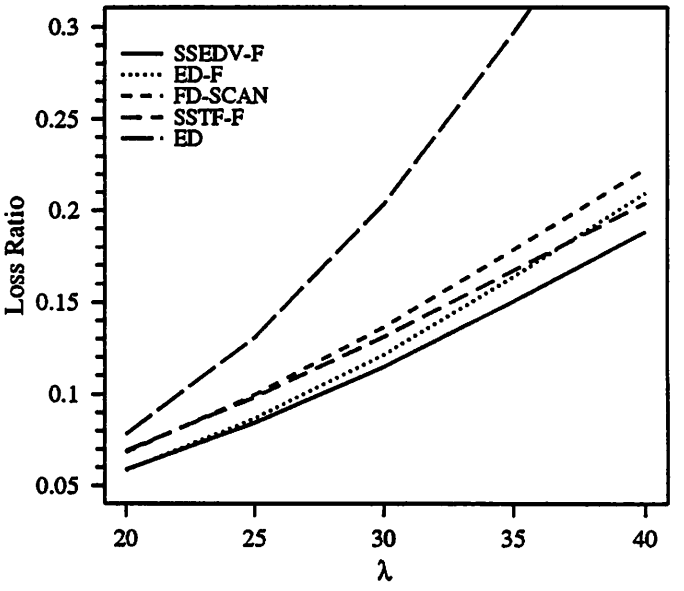


Figure 5.26 Revisit of Removal Real-Time System Model.

Revisiting Removal Real-Time Disk I/O Subsystems: Finally, we revisit a real-time disk I/O subsystem in which missed requests are removed without being scheduled. Although some of these policies have been reported in our previous studies [26], the purpose here is to verify our expectations on the relationships between removal and non-removal real-time system models. From Figure 5.26 and 5.22 we observe that although the degree of improvement of one policy over another may differ from that of non-removal system, the performance ordering of these policies remains the same for both system models. In addition, the ED-F and SSTF-F policies, which are simple and show good performance, were not included in our previous studies.

5.6 Summary

This chapter was dedicated to examining disk I/O subsystems suitable for real-time systems. In particular, we proposed two new real-time disk I/O scheduling algorithms, SSEDO and SSEDV, for a single disk subsystems. Their performance was first studied and compared with other known real-time as well as non-real-time algorithms in an integrated real-time transaction system model, where each transaction which has missed its deadline is removed from the system immediately. The transaction system model has been validated on an actual real-time transaction system testbed, called RT-CARAT. Then we extended our study to mirrored disks and disk arrays by applying the proposed real-time algorithms to various disk mirroring strategies and disk array architectures.

The main performance results are as follows:

- SSEDV and SSEDO can improve the performance up to 38% over the three previously suggested real-time disk scheduling algorithms, or up to 53% over the four conventional algorithms in terms of minimizing the transaction loss ratio. These two algorithms are easily implemented;

- We also showed that SSEDV performs better than SSEDO, since SSEDV uses more knowledge concerning the time constraint and the cost of using that knowledge is obtained for free since it can be used in parallel with disk seeks;

- Our window algorithms SSEDV and SSEDO take into account two factors, *the time constraint* and *the disk service time*, for their decision making, but place more weight on the time constraint component. This is quite different from the two SCAN based algorithms, P-SCAN and FD-SCAN, which place more weight on the disk service time factor. The results show that our algorithms perform better;

- The SSEDV and SSEDO algorithms are robust with respect to window size. A small window works well under a wide variety of workloads. Careful adjustment of algorithm parameters α and β may improve system performance, but a small range of these parameters appears to be suitable for various workloads;

- In addition to providing fault tolerance, disk mirroring can also improve the system performance. We have shown that, by adopting an appropriate scheduling policy, a mirrored disk can decrease the fraction of transactions that miss their deadlines over a single disk by up to 68%. Our results also reveal the importance of real-time policies, which can lead up to a 17% performance improvement over non-real-time policies for a mirrored disk;

- Disk arrays can support concurrent I/O simultaneously. However, employing a real-time algorithm can result in up to a 20% additional performance improvement. Among different disk array architectures, RAID 1 is shown to be significantly better than RAID 5 in terms of reducing the loss ratio, since RAID 5 suffers from the high cost of "small" writes.

In addition, with today's technology, the disk controller can be implemented to monitor the I/O load dynamically, and to select a proper scheduling algorithm accordingly [13]. This technique can be used here with our SSEDV and SSEDO algorithms in a soft real-time environment. For example, when the I/O queue length is less than a threshold, the ED algorithm (window size 1 in SSEDV or SSEDO) might be used for scheduling, otherwise the window size would be set to 3 or 4. Alternatively, we might dynamically update the scheduling parameter α or β according to a queue length threshold.

Last, we studied the disk I/O characteristics for non-removal real-time systems. Unlike the usual way of looking directly into the behavior of such a system, we do so by studying the relationships between removal and non-removal system models. A simple but useful paradigm, expressed as a function Φ, for transferring policies for removal systems to policies for non-removal systems has been proposed. By using this paradigm, we easily obtain various disk scheduling policies suitable for a non-removal real-time system from those previously studied in a removal system environment. An important observation is that, by applying the paradigm, the performance ordering of different policies is preserved among both removal and non-removal real-time system models. Therefore we can easily locate a good policy for a non-removal real-time disk I/O subsystem based on the results achieved from removal systems.

Finally, while our performance metric of most interest is the customer loss ratio, it is reasonable to consider reducing the lateness (or tardiness) of missed customers as well for a non-removal real-time system. The problem of how to find a 'good' cost function which combines these metrics together is still open, and will be examined in the future.

CHAPTER 6

SUMMARY AND FUTURE WORK

6.1 Summary of the Dissertation

In this dissertation, we studied the design and performance issues of high performance disk I/O subsystems for a variety of computing systems.

In this thesis work, we first studied a mirrored disk which can be found in various fault tolerant systems, in which each data item is duplicated. If one device fails, data is still available from the other device. As long as the second failure does not occur before the failed one has recovered, the system is *non-stop*. For such a mirrored disk, writes must be performed on both disks but reads can be satisfied by either disk. Given this fault tolerance, we are interested in how to achieve the performance gain by taking advantage of the duplicated data copies. In particular, we proposed and examined several policies which differ in the manner that a read request is routed to one of the two disks. Analytic models have been developed to model the behaviors of these policies.

The results show that the best policy is the distributed DP-MR policy which maintains one queue in front of each disk and routes a read to join both queues. Whenever one of the two reads finishes, the other one is aborted. Therefore this policy requires the controller to support abortion of a request during service. If disk accesses are assumed to be non-preemptive, i.e., once a request starts, it will be served to complete, then the shortest queue policy DP-SQ-MS and the central queue policy CP-AQ-MS perform reasonably well, with CP-AQ-MS slightly better than DP-SQ-MS when requests are served in a first-come-first-serve fashion. When the two disks are physically located on two different sites connected through a local area network, then the distributed DP-RJ policy is the best choice.

While our main focus is on the study of mirrored disks, the modeling techniques developed in this study is of independent interest. In particular, we modeled different

policies by multiple dimensional Markov chains. Because of the complexity of the system, there is no easy way to solve these Markov chains. Instead, we developed bounding techniques which provide tight bounds on the performance metrics of interest. These modeling techniques can be used or extended to other system studies. An immediate extension of these modeling techniques is to the model of RAID 1 mirrored disk array.

Second, we investigated various disk array architectures proposed in the literature. We also proposed a new architecture which has been observed to provide comparable or higher performance than existing ones. Our main contributions are the design of scheduling policies suitable for each of these architectures, the development of analytic models for these architectures coupled with the proposed policies, and the performance evaluation of these architectures. The disk array architectures of interest to us are RAID 1 (*mirrored declustering*) and its variants (*chained declustering* and *group-rotate declustering*), RAID 5, and Parity Striping. Multiple dimensional Markovian chains were used to approximately model RAID 1 and its variants. For RAID 5 and Parity Striping, we developed a priority queueing model which accurately models the behavior of these two architectures. Our work differs from previous performance studies by explicitly modeling the *write synchronization problem*, which exists in these two architectures, in our performance study. Consequently, two synchronized scheduling policies for RAID 5 and Parity Striping were proposed and examined.

The performance of these disk array architectures and policies were examined in different computing system environments. On one hand, for transaction processing and workstation/engineering applications, I/O access patterns are typically characterized by large arrival rate but each request accesses a small amount of data. On the other hand, for supercomputing and image processing systems, fewer I/O's are issued per unit time but each request transfers a large amount of data. For each case, we considered the workloads where most of the requests are reads and the workloads where most of the requests are writes.

As a consequence, among the three variations of RAID 1, our proposed *group-rotate declustering* provides the best performance for applications with predominantly small transfers. In a large I/O environment, mirrored declustering and group-rotate

declustering provide similar performance, and both of them perform much better than chained declustering.

The performances of RAID 5 and Parity Striping, which have the same cost and achieve the same capacity, were compared only in a small I/O environment, since Parity Striping was proposed for transaction processing systems and it certainly is not good for large I/O transfers. Our results show that RAID 5 is sensitive to the increase of mean request size, and Parity Striping is sensitive to skewed accesses. Thus, RAID 5 may outperform Parity Striping for some applications, but the reverse is true for some other applications. A favored area where one is preferred over the other is provided as a function of mean request size and skew degree. This will help designers and users to choose which architecture for their applications.

We also compared RAID 1 and RAID 5 under two cases: (1) both arrays have the same user capacity; and (2) both arrays contain the same number of disks. Notice that RAID 1 is more expensive than RAID 5, with either doubling the number of disks or losing half of its capacity. Given the high cost paid by RAID 1, the goal is to examine what performance gain can be achieved by constructing a disk array as RAID 1. As a consequence, RAID 1 does provide much higher performance than RAID 5, especially in small I/O environments. The only exception observed is for the case of both arrays having the same number of disks, most of the requests being writes, each request accessing a large amount of data, and more than 70% of these writes performing a *full stripe write.*

Third, we studied disk I/O subsystems for real-time systems, in which each I/O is expected to finish before a deadline, and the goal is to minimize the loss ratio. Specifically, we first developed two new real-time disk scheduling algorithms, SSEDO and SSEDV. We then compared their performance against other known real-time algorithms FD-SCAN, P-SCAN, and ED, as well as conventional algorithms SSTF, SCAN, C-SCAN, and SSTF. The performance study of these algorithms was conducted in an integrated removal real-time transaction system model, which contains a whole set of protocols supporting real-time transaction processing, such as CPU scheduling, 2PL locking mechanism and lock conflict resolution, deadlock detection and resolution, transaction wakeup and restart strategies, etc. In such

a removal system, if a transaction cannot commit before hitting its deadline, it is removed from the system without receiving service. The real-time transaction system model, in which I/O requests are served in a FCFS manner, was validated on an actual real-time transaction system testbed, called RT-CARAT.

The results show that the two new algorithms can improve performance up to 38% over previously known real-time disk scheduling algorithms, and up to 53% over conventional algorithms. While the two new algorithms were first developed for a single disk system, we extended our study to mirrored disks and disk arrays by combining the real-time algorithms with the architectures and policies studied for non-real-time systems. The results show that a mirrored disk can decrease loss ratio up to 68% over a single disk subsystem, and real-time algorithms can lead up to a 17% performance improvement over non-real-time algorithms for such a mirrored disk. For disk array subsystems, a similar performance improvement can be achieved by employing real-time algorithms. In addition, RAID 1 is observed to be significantly better than RAID 5 in terms of reducing the loss ratio.

Finally, we studied disk I/O subsystems for non-removal real-time systems, in which all of the computational entities will be served eventually, even though some of them have already missed their deadlines. Unlike the usual way of looking directly into the behavior of such a non-removal system, we did so by studying the relationships between removal and non-removal system models. A simple but useful paradigm to transfer policies developed or studied in removal systems to non-removal systems was proposed. By using this paradigm, we can easily locate good disk scheduling policies for a non-removal real-time system.

6.2 Future Extensions

In this dissertation, we have partially addressed some design issues and examined the performance of I/O subsystems for different computing systems. While the results, we believe, are valuable to the system designers for their decision making, there are some limitations on our models. For example, our models do not include a cache and only account for traffic passed through the cache and actually performed on disks. An integrated model combining the disk and cache activities will better serve system

designers. In addition, there are several issues which we think are interesting but have not been addressed in this dissertation. Hence, the work reported in this dissertation can be extended in several directions:

- **Disk Arrays in a Distributed Environment:**

  When a disk array is implemented in a distributed environment, there are a number of challenging issues that need to be addressed. Consider a high speed local area network connecting multiple workstations which are supported by a file server based on a disk array. The I/O subsystem is reliable since the disk array contains redundant information to tolerate a single disk failure. However, if the file server crashes, the whole system fails. A distributed disk array places one disk at each site so that it can tolerate a single site failure. The cost paid for obtaining this fault tolerance is expected to be high, since much more messages are transferred across the network. In this case, questions such as how to select the unit size for data interleaving, what kind of network protocol is appropriate, whether it is worthwhile to provide duplicate data other than parity, are open problems. For example, if the distributed disk array is implemented as RAID 5 or Parity Striping, then, as indicated in this dissertation, the AR policy might be a better choice than the PS policy (see Section 4.2.1). An accompanying question is whether the network protocol should give the parity update request a high priority.

- **Disk Array Cache Modeling and Log-Structured File System:**

  As mentioned above, an integrated model of cache and disk arrays will provide a better view on the behavior of I/O subsystems. As we have seen in previous chapters, RAID 5 suffers when write requests perform *partial stripe writes*, and yields high performance if most of the writes are *full stripe writes*. The Log-Structured File System tries to achieve this full stripe write by accumulating small writes in the cache and writing a whole stripe to the disk. Research in this direction is still in its infancy, and there are many interesting questions remaining to be examined. Moreover, when the disk array is on a remote site,

many systems also maintain a local cache. Therefore, modeling such a cache hierarchy is even more complicated.

- **Disk Array Support of Time Constrained Multimedia Service:**

  Currently, as network speed goes to gigabits per second, multimedia service has raised more and more interest among researchers, where a large amount of digitized video and voice data are transferred across a high speed network and shown up on remote video or audio devices. Because of the high capacity and high bandwidth properties of disk arrays, they can be used as storage devices for these video and voice data. Since video and voice data, unlike regular files, typically carry a time constraint, they should be transferred from the disk array before missing its deadline. A typical application is the transfer of a stream of pictures every Δ time units from a disk array to a remote screen. Thus, the question is how the disk array serves this kind of time constrained stream. Since typically a disk array's bandwidth is higher than that required by a customer and there may be more than one such customer requiring services, the disk array should be properly scheduled to serve multiple streams, providing them with their respective quality of service. An even more complicated situation exists when a disk array needs to serve a mixture of such time constrained data as well as regular files.

- **Disk Arrays During Failure:**

  As indicated before, the focus of this dissertation is on the study of disk arrays under normal operational mode. Since there may be hundreds of disks in the array, occasionally a device failure may occur. In this case, the disk array enters a failure mode and then a rebuild mode. In these modes, while the disk array continues serving customers, a background process is automatically started to rebuild the failed disk on a hot spare disk. There is a trade-off between the quality of service during failure and the rebuild time. If one wants to maintain a certain level of service to customers, then the rebuild period is enlarged, and therefore the probability that a second device fails during this rebuild period increases. On the other hand, if one wants to have a short rebuild time, then

customers will suffer much longer delays. Many studies on the behavior and performance of disk arrays during failure have been reported in the literature [34, 77, 76, 75, 51, 52, 42]. However, there are many new disk array architectures and techniques have been developed, and therefore there are many new issues to be examined.

In addition, more work needs to be done on implementation issues of different disk subsystem architectures and scheduling policies. It would also be interesting to conduct trace-driven performance studies of different disk array architectures based on real disk access traces rather than assuming Poisson arrivals. Other work includes examining the effect of disk array pre-fetching and access localities, as well as quantitatively examining the improvement of applying *shortest seek time first* or *elevator* algorithm to each queue of disk arrays.

APPENDIX A

CALCULATIONS FOR THE DP-RJ POLICY

In this Appendix, we obtain the stationary probabilities for the model that produces a lower bound on the performance of the DP-RJ policy, $P[\mathbf{N}^{(lb)} = (i,j)] = q(i,j)$, $i = 0,1,\cdots;\ j = 0,1,\cdots,B$. We define the steady state probability vector

$$Y = (y(0), y(1), y(2), \cdots)$$

where $y(i)$ is a $(B+1)$ element vector,

$$y(i) = (q(i,0), q(i,1), \cdots, q(i,B)), \qquad i = 0,1,\cdots.$$

The infinitesimal generator Q, satisfying $YQ = 0$, is listed below

$$Q = \begin{bmatrix} B_1 & A_0 & 0 & 0 & \cdots \\ B_2 & A_1 & A_0 & 0 & \cdots \\ 0 & A_2 & A_1 & A_0 & \cdots \\ 0 & 0 & A_2 & A_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

where the matrices B_1, B_2, A_0, A_1, and A_2 are defined as $(B+1) \times (B+1)$ matrices,

$$A_0 = \begin{bmatrix} \alpha_1 p_r \lambda & p_w \lambda & & & \\ 0 & \alpha_1 p_r \lambda & p_w \lambda & & \\ & & \ddots & & \\ & & & & \alpha_1 p_r \lambda + p_w \lambda \end{bmatrix}$$

$$B_1 = \begin{bmatrix} -\lambda & \alpha_2 p_r \lambda & 0 & & & & \\ \mu & -(\lambda+\mu) & \alpha_2 p_r \lambda & & & & \\ 0 & \mu & -(\lambda+\mu) & & & & \\ & & \ddots & \ddots & & & \\ & & & & \mu & -(\lambda+\mu) & \alpha_2 p_r \lambda \\ & & & & & \mu & -(p_w \lambda + \alpha_1 p_r \lambda + \mu) \end{bmatrix}$$

$$B_2 = A_2 = \mu I_{(B+1)\times(B+1)}$$

$$A_1 = \begin{bmatrix} c_0 & \alpha_2 p_r \lambda & & & & & \\ \mu & c_1 & \alpha_2 p_r \lambda & & & & \\ 0 & \mu & c_1 & \alpha_2 p_r \lambda & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & & \mu & c_1 & \alpha_2 p_r \lambda \\ & & & & & \mu & c_2 \end{bmatrix}$$

$$\begin{aligned} c_0 &= -(\lambda + \mu) \\ c_1 &= -(\lambda + 2\mu) \\ c_2 &= -(p_w \lambda + 2\mu + \alpha_1 p_r \lambda) \end{aligned}$$

Define the matrix $A = A_0 + A_1 + A_2$. Neuts [81] showed that Q is positive recurrent if $\pi A_0 e < \pi A_2 e$, where π is the unique solution to $\pi A = 0$, $\pi e = 1$. Here e is a column vector containing $B+1$ ones and π is a $B+1$ element vector containing the stationary queue length distribution for the M/M/1/B queue with arrival rate $(\alpha_2 p_r + p_w)\lambda$ and service rate μ, i.e., the i-th element of π is $(1-u)u^i/(1-u^{(B+1)})$ where $u = (\alpha_2 p_r + p_w)\lambda/\mu$. In the modified model, the condition for positive recurrence is $(\alpha_1 p_r + p_w)\lambda < \mu$. Whenever Q is positive recurrent, the stationary probability vector Y can be expressed as

$$y(i) = y(0)R^i, \quad i = 0, 1, \cdots.$$

where R is the minimal solution of $A_0 + RA_1 + R^2 A_2 = 0$. The matrix R can be obtained iteratively by the following approach. Let $R(n)$ denote the value of R after n iterations.

$$\begin{aligned} R(0) &= 0, \\ R(n+1) &= -A_0 A_1^{-1} - R^2(n) A_2 A_1^{-1}, \quad n > 0. \end{aligned}$$

Rao and Posner [95] have shown that the vector $y(0)$ takes the form

$$y(0) = \pi(I - R).$$

The preceding analysis differs from that presented by Rao and Posner only in the definition of the submatrices B_1, B_2, A_0, A_1, A_2.

APPENDIX B

CALCULATIONS FOR THE CP-AQ POLICY

The stationary probability distribution for the model that provides a lower bound on the performance of the CP-AQ policy, $P[\mathbf{N}^{(lb)} = (m,n,l)] = q(m,n,l)$, $m = 0,1,\cdots$; $n = 0,1,\cdots,B$; $l = 0,1$, is obtained here. We define the steady state probability vector $Y = (x, y(0), y(1), y(2), \cdots)$ where $x = [q(0,0,0)]$, $y(0)$ is the B element vector $[q(0,1,0), q(0,2,0), \cdots, q(0,B,0)]$, and $y(i)$ is a B element vector $y(i) = [q(i-1,1,1), \cdots, q(i-1,B,1)]$, $i = 1,2,\cdots$. The infinitesimal generator Q satisfying $YQ = 0$ is listed below,

$$Q = \begin{bmatrix} D_2 & C_1 & B_0 & 0 & 0 & \cdots \\ D_3 & C_2 & B_1 & 0 & 0 & \cdots \\ 0 & C_3 & A_2 & A_1 & 0 & \cdots \\ 0 & 0 & A_3 & A_2 & A_1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

where D_2 is a one element vector, $D_2 = [-\lambda]$, D_3 is a B element column vector $D_3 = [\mu, 0, \cdots, 0]^T$, B_0 and C_1 are B element vectors $B_0 = [\mu, 0, \cdots, 0]$, $C_1 = [p_r\lambda, 0, \cdots, 0]$, and C_2, C_3, B_1, A_1, A_2, A_3 are $B \times B$ matrices

$$C_2 = \begin{bmatrix} -(\lambda+\mu) & 0 & 0 & & \\ \mu & -(\lambda+\mu) & 0 & & \\ 0 & \mu & -(\lambda+\mu) & & \\ & & \ddots & \ddots & \\ & & & \mu & -(\lambda+\mu) \end{bmatrix}$$

$$C_3 = \begin{bmatrix} 2\mu & 0 & & \\ 0 & \mu & & \\ & \ddots & \ddots & \\ & & 0 & \mu \end{bmatrix}$$

$$B_1 = \begin{bmatrix} p_r\lambda & p_w\lambda & & & \\ & p_r\lambda & p_w\lambda & & \\ & & \ddots & \ddots & \\ & & & p_r\lambda & p_w\lambda \\ & & & & \lambda \end{bmatrix}$$

$$A_1 = \lambda I_{B\times B},$$

$$A_2 = \begin{bmatrix} -(\lambda+2\mu) & 0 & & & \\ \mu & -(\lambda+2\mu) & & & \\ & \ddots & \ddots & & \\ & & & \mu & -(\lambda+2\mu) \end{bmatrix}$$

$$A_3 = \begin{bmatrix} 2\mu p_r & 2\mu p_w & 0 & & \\ 0 & \mu p_r & \mu p_w & & \\ & \ddots & \ddots & & \\ & & & \mu p_r & \mu p_w \\ & & & & \mu \end{bmatrix}$$

Define the matrix $A = A_1 + A_2 + A_3$. The infinitesimal generator Q is positive recurrent if $\pi A_1 e < \pi A_3 e$, where π is the unique solution to $\pi A = 0$, $\pi e = 1$. The stationary probability vector Y satisfies the matrix-geometric form

$$y(i) = y(1)R^{i-1},\ i = 1, 2, \cdots$$

where R is the minimal solution of $A_1 + RA_2 + R^2A_3 = 0$. The matrix R can be obtained in a similar manner as for the model of the DP-RJ policy. Finally, the vectors x, $y(0)$, and $y(1)$ are obtained by solving the following set of equations,

$$\begin{aligned} xD_2 + y(0)D_3 &= 0, \\ xC_1 + y(0)C_2 + y(1)C_3 &= 0, \\ xB_0 + y(0)B_1 + y(1)[A_2 + RA_3] &= 0, \\ x + [y(0) + y(1)[I - R]^{-1}]e &= 1. \end{aligned}$$

APPENDIX C

CALCULATIONS FOR THE DP-SQ-MS POLICY

In this Appendix, we present a solution to the Markov chain $\mathbf{N}(t) = (N_{max}(t), N_{min}(t), N_3(t))$, which provides a lower bound for the DP-SQ-MS policy that directs a read request to join the shortest queue at the moment of its arrival and remains there until it is served. If a read request, say R^*, arrives and finds both disks idle, it is assigned to the one that requires the least arm movement (see Section 3.1 and 3.2.2). We further identify two variations of the DP-SQ-MS scheduler, which differ slightly in their treatment of those read requests that arrive prior to the completion of R^* and find the two queue lengths to be equal. The first variation, termed *intelligent scheduler (IS)*, requires that the scheduler maintain a one bit flag to keep track of the disk currently serving R^*. Whenever the two queue lengths are the same, a new read is always assigned to the queue associated with the server currently occupied by R^*. Since R^* arrived earlier and was assigned to the disk providing the smallest seek time, it is expected to complete before the request being served on the other disk.

The second variation, called *simple scheduler (SS)*, does not keep track of which disk is currently serving R^*. Instead, a new read is randomly assigned to either of the two queues with equal probability whenever their lengths are the same.

In the following, we first present an analytic model for the *intelligent scheduler (IS)*, and then simply describe modifications required to model the *simple scheduler (SS)*. The numerical results for the two variations, as expected, are very close to each other such that the performance differences can be perceived by only tens of nanoseconds on the mean I/O response times, with IS slightly better than SS. Hence, the *simple scheduler* is preferred.

Model for the Intelligent Scheduler:

In this model, the state variables $N_{max}(t)$ and $N_{min}(t)$ denote the longest and the shortest queue lengths at time t, respectively. $N_3(t)$ indicates whether the most recent

busy period was initiated by a read and this read is still in service ($N_3(t) = 1$), or this read has already finished or the busy period was initiated by a write ($N_3(t) = 0$). Observe that when $N_3(t) = 1$, the read that initiated the most recent busy period (which took a fast service rate μ') is always in the longest queue at time t because of the behavior of the *intelligent scheduler*.

To solve this model, we first calculate the stationary probability distribution

$$P\{\mathbf{N} = (m,n,l)\} = q(m,n,l), \quad \begin{array}{l} m = 0,1,\cdots; \\ n = 0,1,\cdots,B; \\ l = 0,1, \end{array}$$

and then obtain the mean I/O response times. We define the steady state probability vector

$$Y = (y(0), y(1), y(2), \cdots)$$

where $y(0)$ is a $2B+1$ element vector

$$y(0) = [q(0,0,0), q(1,1,0), q(1,1,1), \cdots, q(B,B,0), q(B,B,1)]$$

and $y(i)$ is a $2B+2$ element vector

$$y(i) = [q(i,0,0), q(i,0,1), \cdots, q(i+B,B,0), q(i+B,B,1)], \qquad i = 1,2,\cdots$$

These probabilities satisfy the following equations,

$$\begin{aligned}
\lambda q(0,0,0) &= \mu q(1,0,0) + \mu' q(1,0,1), \\
(\lambda+\mu) q(1,0,0) &= \mu q(2,0,0) + 2\mu q(1,1,0) \\
&\quad +\mu'[q(2,0,1) + q(1,1,1)], \\
(\lambda+\mu') q(1,0,1) &= p_r \lambda q(0,0,0) + \mu q(1,1,1), \\
(\lambda+\mu) q(i,0,0) &= \mu[q(i+1,0,0) + q(i,1,0)] + \mu' q(i+1,0,1), \\
(\lambda+\mu') q(i,0,1) &= \mu q(i,1,1), \qquad i = 2,3\cdots \\
\\
(\lambda+2\mu) q(i,i,0) &= p_r \lambda q(i,i-1,0) + p_w \lambda q(i-1,i-1,0) \\
&\quad \mu q(i+1,i,0) + \mu' q(i+1,i,1), \qquad i = 1,\cdots,B. \\
(\lambda+\mu+\mu') q(1,1,1) &= p_r \lambda q(1,0,1), \\
(\lambda+\mu+\mu') q(i,i,1) &= p_r \lambda q(i,i-1,1) + p_w \lambda q(i-1,i-1,1), \\
&\quad i = 2,\cdots,B.
\end{aligned}$$

$$
\begin{aligned}
(\lambda+2\mu)q(i+1,i,0) &= p_r\lambda[q(i+1,i-1,0)+q(i,i,0)] \\
&\quad +p_w\lambda q(i,i-1,0) \\
&\quad +2\mu q(i+1,i+1,0)+\mu q(i+2,i,0) \\
&\quad +\mu'[q(i+1,i+1,1)+q(i+2,i,1)], \\
(\lambda+\mu+\mu')q(i+1,i,1) &= p_r\lambda[q(i+1,i-1,1)+q(i,i,1)] \\
&\quad +p_w\lambda q(i,i-1,1)+\mu q(i+1,i+1,1), \\
&\quad i=1,\cdots,B-1. \\
(p_w\lambda+2\mu)q(B+1,B,0) &= p_r\lambda q(B+1,B-1,0)+p_w\lambda q(B,B-1,0) \\
&\quad +\lambda q(B,B,0)+\mu q(B+2,B,0) \\
&\quad +\mu' q(B+2,B,1), \\
(p_w\lambda+\mu+\mu')q(B+1,B,1) &= p_r\lambda q(B+1,B-1,1)+p_w\lambda q(B,B-1,1) \\
&\quad +\lambda q(B,B,1), \\
\\
(\lambda+2\mu)q(i,j,0) &= p_r\lambda q(i,j-1,0)+p_w\lambda q(i-1,j-1,0) \\
&\quad +\mu[q(i+1,j,0)+q(i,j+1,0)] \\
&\quad +\mu' q(i+1,j,1), \\
(\lambda+\mu+\mu')q(i,j,1) &= p_r\lambda q(i,j-1,1)+p_w\lambda q(i-1,j-1,1) \\
&\quad +\mu q(i,j+1,1), \\
&\quad i=3,4,\cdots;j=1,\cdots,\min(i-2,B-1) \\
(p_w\lambda+2\mu)q(i,B,0) &= p_w\lambda[q(i-1,B-1,0)+q(i-1,B,0)] \\
&\quad +p_r\lambda q(i,B-1,0)+\mu q(i+1,B,0) \\
&\quad +\mu' q(i+1,B,1), \\
(p_w\lambda+\mu+\mu')q(i,B,1) &= p_w\lambda[q(i-1,B-1,1)+q(i-1,B,1)] \\
&\quad +p_r\lambda q(i,B-1,1), \qquad i=B+2,B+3,\cdots.
\end{aligned}
$$

The infinitesimal generator Q satisfying $YQ=0$ is listed below,

$$
Q=\begin{bmatrix}
B_1 & B_0 & 0 & 0 & 0 & \cdots \\
B_2 & A_1 & A_0 & 0 & 0 & \cdots \\
0 & A_2 & A_1 & A_0 & 0 & \cdots \\
0 & 0 & A_2 & A_1 & A_0 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{bmatrix}
$$

where B_1 is a $2B+1$ dimensional matrix,

$$
B_1=\begin{bmatrix}
-\lambda & \lambda p_w & 0 & & & & \\
 & -(\lambda+2\mu) & 0 & \lambda p_w & & & \\
 & & -(\lambda+\mu+\mu') & 0 & \ddots & & \\
 & & & -(\lambda+2\mu) & \ddots & \lambda p_w & \\
 & & & & \ddots & 0 & \\
 & & & & & -(\lambda+\mu+\mu')
\end{bmatrix}
$$

B_2 is a $(2B+2)\times(2B+1)$ matrix,

$$B_2 = \begin{bmatrix}
\mu & \lambda p_r & & & & & & \\
\mu' & 0 & \lambda p_r & & & & & \\
0 & \mu & 0 & \lambda p_r & & & & \\
 & \mu' & 0 & 0 & \lambda p_r & & & \\
 & & 0 & \mu & 0 & \ddots & \lambda p_r & \\
 & & & \mu' & 0 & \ddots & 0 & \lambda p_r \\
 & & & & 0 & \ddots & \mu & 0 \\
 & & & & & & \mu' & 0
\end{bmatrix}$$

B_0 is a $(2B+1)\times(2B+2)$ matrix,

$$B_0 = \begin{bmatrix}
0 & \lambda p_r & & & & & & & \\
2\mu & 0 & \lambda p_r & & & & & & \\
\mu' & \mu & 0 & \lambda p_r & & & & & \\
 & 0 & 2\mu & 0 & \lambda p_r & & & & \\
 & & \mu' & \mu & 0 & \lambda p_r & & & \\
 & & & 0 & 2\mu & 0 & \ddots & & \\
 & & & & \mu' & \mu & \ddots & \lambda p_r & \\
 & & & & & 0 & \ddots & 0 & \lambda p_r \\
 & & & & & & \ddots & 2\mu & 0 & \lambda \\
 & & & & & & & \mu' & \mu & 0 & \lambda
\end{bmatrix}$$

and A_0, A_1, A_2 are $2B+2$ dimensional matrices,

$$A_0 = \begin{bmatrix}
0 & & & & & \\
0 & \ddots & & & & \\
\mu & \ddots & 0 & & & \\
 & \ddots & 0 & 0 & & \\
 & & \mu & 0 & \lambda p_w & \\
 & & & \mu & 0 & \lambda p_w
\end{bmatrix}$$

$$A_1 = \begin{bmatrix}
-(\lambda+\mu) & 0 & \lambda p_w & & & & \\
 & -(\lambda+\mu') & 0 & \lambda p_w & & & \\
 & & c_1 & 0 & \ddots & & \\
 & & & c_2 & \ddots & & \\
 & & & & \ddots & 0 & \lambda p_w \\
 & & & & & -(\lambda p_w + 2\mu) & 0 \\
 & & & & & & -(\lambda p_w + \mu + \mu')
\end{bmatrix}$$

where $c_1 = -(\lambda + 2\mu)$ and $c_2 = -(\lambda + \mu + \mu')$.

$$A_2 = \begin{bmatrix} \mu & 0 & \lambda p_r & & & & & & \\ \mu' & 0 & 0 & \lambda p_r & & & & & \\ & 0 & \mu & 0 & \lambda p_r & & & & \\ & & \mu' & 0 & 0 & \lambda p_r & & & \\ & & & 0 & \mu & 0 & \ddots & & \\ & & & & \mu' & 0 & \ddots & \lambda p_r & \\ & & & & & 0 & \ddots & 0 & \lambda p_r \\ & & & & & & \ddots & \mu & 0 \\ & & & & & & & \mu' & 0 \end{bmatrix}.$$

The parameter λ is the request arrival rate; μ is the disk service rate obtained from Equation (2.8); and μ' is the service rate for those reads that find both data copies available at the moment of their arrival (Equation (2.10)).

Define the matrix $A = A_0 + A_1 + A_2$. Neuts [81] showed that Q is positive recurrent if $\pi A_0 e < \pi A_2 e$, where π is the unique solution to $\pi A = 0$, $\pi e = 1$. Here e is a column vector containing $2B + 2$ ones. Whenever Q is positive recurrent, the stationary probability vector Y satisfies the matrix-geometric form

$$y(i) = y(1)R^i, \quad i = 0, 1, \cdots$$

where R is the minimal solution of $A_0 + RA_1 + R^2A_2 = 0$ (see Appendix A).

Finally, the vectors $y(0)$ and $y(1)$ are obtained by solving the following set of equations,

$$\begin{aligned} y(0)B_1 + y(1)B_2 &= 0 \\ y(0)B_0 + y(1)(A_1 + RA_2) &= 0 \\ y(0)e' + y(1)[I - R]^{-1}e &= 1 \end{aligned}$$

where e' is a column vector with $2B + 1$ ones.

The mean I/O response times for read and write requests are obtained in a following way. Let $y'(0)$ be a $2B + 2$ element vector obtained by inserting a zero at the second position of $y(0)$, $y'(0) = (q(0,0,0), 0, q(1,1,0), q(1,1,1), \cdots, q(B,B,1))$. The mean queue length for the shortest queue is

$$\overline{N_{min}} = \sum_{k=0}^{B} k[y'(0) + \sum_{i=1}^{\infty} y(i)]e_k$$

$$= [y'(0) + y(1)(I - R)^{-1}] \sum_{k=0}^{B} k e_k$$

where e_k is a $2B+2$ element vector with the $(2k+1)$-th and $(2k+2)$-th elements set to one, and all other elements set to zero.

The mean read response time is

$$\begin{aligned} \overline{Z_r} &= q(0,0,0)\frac{1}{\mu'} + \sum_{j=0}^{B} \sum_{i=j, i\neq 0}^{\infty} [q(i,j,0) + q(i,j,1)](j+1)\frac{1}{\mu}, \\ &= (\overline{N_{min}} + 1)\frac{1}{\mu} + q(0,0,0)(\frac{1}{\mu'} - \frac{1}{\mu}). \end{aligned}$$

To obtain the mean write response time $\overline{Z_w}$, we first define $T_{i,j,k}$ to be the mean response time for a write request which finds the two queue lengths to be $i-1$ and $j-1$ at the moment of its arrival. The subscript k is defined in a similar way as the state variable $N_3(t)$, i.e., it is used to identify whether the most recent busy period was initiated by a read and this read is still in service ($k=1$), or this read has already finished or the busy period was initiated by a write ($k=0$). Then $T_{i,j,k}$ can be obtained by solving the following recurrences:

$$\begin{aligned} T_{i,j,0} &= \frac{1}{2\mu} + \frac{1}{2}T_{i-1,j,0} + \frac{1}{2}T_{i,j-1,0}, \\ T_{i,j,1} &= \begin{cases} \frac{1}{\mu'+\mu} + \frac{\mu'}{\mu'+\mu}T_{i-1,j,0} + \frac{\mu}{\mu'+\mu}T_{i,j-1,1}, & i \geq j, \\ \frac{1}{\mu'+\mu} + \frac{\mu}{\mu'+\mu}T_{i-1,j,1} + \frac{\mu'}{\mu'+\mu}T_{i,j-1,0}, & i < j, \end{cases} \\ & \qquad\qquad i,j = 0,1,2,\cdots \end{aligned}$$

with the boundary conditions

$$\begin{cases} T_{0,j,0} = \frac{j}{\mu}; & T_{0,j,1} = \frac{1}{\mu'} + \frac{j-1}{\mu}, \quad j = 0,1,\cdots; \\ T_{i,0,0} = \frac{i}{\mu}; & T_{i,0,1} = \frac{1}{\mu'} + \frac{i-1}{\mu}, \quad i = 0,1,\cdots. \end{cases}$$

In case of $k=0$, the first term in the recurrence corresponds to the average delay until the first of the two disks completes. Since the disk that finishes first can be either one in a mirrored pair with same probability, the write request observes the system with one less task in that queue. Hence, its average delay in this case corresponds to the second and the third terms in the recurrence. Similar explanations apply for the case of $k=1$.

Let T_i be a $2B+2$ element vector, $T_i = (T_{i,1,0}, T_{i,1,1}, T_{i+1,2,0}, T_{i+1,2,1}, \cdots, T_{i+B,B+1,0}, T_{i+B,B+1,1})$. The mean write response time is given by

$$\begin{aligned}\overline{Z_w} &= q(0,0,0)T_{1,1,0} + \sum_{j=0}^{B}\sum_{i=j,i\neq 0}^{\infty}[q(i,j,0)T_{i+1,j+1,0} + q(i,j,1)T_{i+1,j+1,1}] \\ &= y'(0)T_1 + y(1)\sum_{i=1}^{\infty} R^{i-1}T_{i+1}.\end{aligned}$$

In practice, the infinite summation can be truncated to a finite number which is large enough, say 500, since the I/O queue length can rarely exceed 500. Finally, the overall mean I/O response time is

$$\overline{Z} = p_r\overline{Z_r} + p_w\overline{Z_w}.$$

Model for the Simple Scheduler:

To model the *simple scheduler*, it is necessary for $N_3(t)$ to take an additional value $N_3(t) = 2$, which carries the semantics that the most recent busy period prior to t was initiated by a read, this read is still in service, and its associated queue is the shortest one at time t. $N_3(t) = 1$, as described above, has the same meaning except that the read that initiated the most recent busy period is now in the longest queue at t. $N_3(t) = 0$ has the same meaning as before. Thus, if the system is stationary and in a state $(k, k, 1)$, $k = 1, 2, \cdots, B$, i.e., the system was started by a read which is still in service and both queue lengths are equal, it transits to the states $(k+1, k, 1)$ or $(k+1, k, 2)$ with a equal rate $(0.5\lambda p_r)$ on the event of an arrival of a new read. Transitions to and from other states remain unchanged as in the *intelligent scheduler* model.

With these modifications, we can construct a new Markov chain, $\mathbf{N}(t) = (N_{max}(t), N_{min}(t), N_3(t))$, in a similar manner to the *intelligent scheduler*. The modified Markov chain may still be solved by using the matrix-geometric method in a similar way as above, and the details are omitted here.

Last, the upper bound for DP-SQ-MS can be modeled by a Markov chain $\mathbf{N}(t) = (N_{min}(t), \Delta(t), N_3(t))$, and be solved in a similar manner.

APPENDIX D

AN ANALYSIS OF RAID 5 WITH SYNCHRONIZED POLICIES

In this Appendix, we present a detailed analysis of RAID 5 coupled with the two synchronized policies, BS and AR. In the following, we will use superscripts l and u to denote quantities obtained based on the upper and lower bounds of Y_p, the service time for high priority parity update requests.

The BS policy

As described before, under the BS policy, a parity request is generated whenever a write request is scheduled for service.

Lower Bound on the Service Time Y_p: The following is a lower bound on Y_p,

$$Y_p \geq L_p \equiv X + 2\tau + R_{max} \tag{D.1}$$

having mean

$$\overline{L_p} = \overline{X} + 2\tau + R_{max}. \tag{D.2}$$

Compared to Equation (4.4), the second full rotation never occurs. L_p matches well to Y_p when the system is highly loaded. This is because when workload increases, the queueing delay Q_p increases, and the probability $P\{X' \geq Q_p + X + R_{max}\}$ approaches zero.

The density function for L_p is

$$f_{L_p}(x) = f_X(x - 2\tau - R_{max}), \qquad 2\tau + R_{max} \leq x \leq 2\tau + 2R_{max} + S_{max} \tag{D.3}$$

where the density function $f_X(\cdot)$ is given by Equation (2.6).

Let Y_d be a $r.v.$ for the service time of low priority read and write requests,

$$Y_d = \theta Y_r + (1-\theta)Y_w$$

where $P\{\theta = 1\} = p_r$ and $P\{\theta = 0\} = p_w$. The density function for Y_d is given by

$$f_{Y_d}(y) = \begin{cases} p_r f_X(y-\tau), & \tau \le y \le 2\tau + R_{max}, \\ p_r f_X(y-\tau) + & \\ \quad p_w f_X(y-2\tau-R_{max}), & 2\tau + R_{max} < y \le \tau + R_{max} + S_{max}, \\ p_w f_X(y-2\tau-R_{max}), & \tau + R_{max} + S_{max} < y \le 2\tau + 2R_{max} + S_{max}, \end{cases} \tag{D.4}$$

and its mean is

$$\overline{Y_d} = p_r\left(\overline{X} + \tau\right) + p_w\left(\overline{X} + 2\tau + R_{max}\right). \tag{D.5}$$

We then obtain the Laplace transforms of the service times of high and low priority requests by

$$B_p^*(s) = \int_{2\tau+R_{max}}^{2\tau+3R_{max}+S_{max}} f_{L_p}(y)e^{-sy}\,dy \tag{D.6}$$

and

$$B_d^*(s) = \int_{\tau}^{2\tau+2R_{max}+S_{max}} f_{Y_d}(y)e^{-sy}\,dy. \tag{D.7}$$

We now use these service time distributions to develop an approximate analysis of the system under the assumption of Poisson arrivals. Based on results on non-preemptive priority queues, the Laplace transforms for the queueing delays of the P and D queues are ([62] pp. 121)

$$W_p^*(s) = \frac{(1-\rho)s + \lambda_d[1 - B_d^*(s)]}{s - \lambda_p + \lambda_p B_p^*(s)} \tag{D.8}$$

where ρ is the device utilization, $\rho = \lambda_p\overline{L_p} + \lambda_d\overline{Y_d}$, and

$$W_d^*(s) = \frac{(1-\rho)[s + \lambda_p - \lambda_p G_p^*(s)]}{s - \lambda_d + \lambda_d B_d^*\left(s + \lambda_p - \lambda_p G_p^*(s)\right)} \tag{D.9}$$

where $G_p^*(s) = B_p^*\left(s + \lambda_p - \lambda_p G_p^*(s)\right)$.

The mean read response time is given by

$$\overline{Z_r^l} = \overline{Q_d^l} + \overline{X} + \tau \tag{D.10}$$

where $\overline{Q_d^l} = -W_d^{*\prime}(0)$.

As in Equation (4.6), the mean write response time, $\overline{Z_w^l}$, is calculated by

$$\overline{Z_w^l} = \overline{Q_d^l} + E\left[\max\{X', Q_p^l + X\}\right] + 2\tau + R_{max}. \tag{D.11}$$

Since we only have the Laplace transform for the distribution of Q_p^l and in our case, there is no easy way to obtain a closed form for the distribution of Q_p^l by inverting the Laplace transform, we approximate the distribution of Q_p^l by a Coxian distribution [35] with identical first and second moments.

The first and second moments of Q_p^l are derived from Equation (D.8),

$$E[Q_p^l] = \frac{\lambda_d \overline{Y_d^2} + \lambda_p \overline{L_p^2}}{2(1 - \lambda_p \overline{L_p})}, \tag{D.12}$$

$$E\left[(Q_p^l)^2\right] = \frac{\lambda_d \overline{Y_d^3} + \lambda_p \overline{L_p^3}}{3(1 - \lambda_p \overline{L_p})} + \frac{(\lambda_d \overline{Y_d^2} + \lambda_p \overline{L_p^2})\lambda_p \overline{L_p^2}}{2(1 - \lambda_p \overline{L_p})^2} \tag{D.13}$$

where the moments of L_p and Y_d are given in Appendix E.

Let ξ_k be a K-stage Coxian *r.v.* with parameters q and μ, and *pdf*

$$g(x) = (1-q)u_0(x) + (1-q)\sum_{i=1}^{K-1} \frac{q^i \mu(\mu x)^{i-1} e^{-\mu x}}{(i-1)!} + \frac{q^K \mu(\mu x)^{K-1} e^{-\mu x}}{(K-1)!} \tag{D.14}$$

where $u_0(x)$ is the unit impulse function (see Section 2.3).

The first and second moments of ξ_k are

$$\overline{\xi_k} = \frac{\sum_{i=1}^{K} q^i}{\mu},$$

$$\overline{\xi_k^2} = \frac{2\sum_{i=1}^{K} i q^i}{\mu^2}.$$

The parameters q, μ, and K are calculated in such a way that the above two moments match exactly to those of Q_p^l. Let C_x be the coefficient of variance of Q_p^l, $C_x^2 = E\left[(Q_p^l)^2\right] / E\left[Q_p^l\right]^2 - 1$, then the number of stages satisfies the equation

$$\frac{1}{C_x^2} \leq K < \frac{1}{C_x^2} + 1$$

and q and μ are obtained by solving the following equations

$$E[Q_p^l] = \frac{\sum_{i=1}^{K} q^i}{\mu},$$

$$E\left[(Q_p^l)^2\right] = \frac{2\sum_{i=1}^{K} iq^i}{\mu^2}.$$

Define *r.v.* $V = Q_p^l + X$, having *pdf*

$$f_V(x) = \int_0^x g(x-t) f_X(t)\, dt.$$

Since X' is independent of Q_p^l and X, the cumulative distribution function (CDF) of $\max\{X', Q_p^l + X\}$ is $F_{X'}(x)F_V(x)$, where $F_{X'}(x)$ and $F_V(x)$ are the CDF's of X' and V respectively. The mean is given by

$$E\left[\max\{X', Q_p^l + X\}\right] = E[X'] + E[Q_p^l + X] - \int_0^{R_{max}+S_{max}} [1 - F_{X'}(x)]\,[1 - F_V(x)]\, dx.$$

Last, the mean I/O response time is given by

$$\overline{Z^l} = p_r \overline{Z_r^l} + p_w \overline{Z_w^l}.$$

We shall refer to this as the *Lower Bound (LB) Approximation.*

Upper Bound on the Service Time Y_p: The upper bound on Y_p is obtained by

$$Y_p \le U_p \equiv X + 2\tau + R_{max} + \mathbf{1}(X' \ge X + R_{max}) R_{max} \tag{D.15}$$

where $\mathbf{1}(A)$ is the indicator function defined by

$$\mathbf{1}(A) = \begin{cases} 1, & \text{if A is true,} \\ 0, & \text{if A is not true.} \end{cases}$$

We expect that U_p better matches Y_p when the system is lightly loaded since, as workload decreases, the queueing delay Q_p approaches zero.

The density function for U_p is

$$f_{U_p}(y) = \begin{cases} f_X(y - 2\tau - R_{max}) F_{X'}(y - 2\tau), & 2\tau + R_{max} \le y \le 2\tau + 2R_{max}, \\ f_X(y - 2\tau - 2R_{max})[1 - F_{X'}(y - 2\tau - R_{max})] + \\ \quad f_X(y - 2\tau - R_{max}) F_{X'}(y - 2\tau), & 2\tau + 2R_{max} < y \le 2\tau + R_{max} + S_{max}, \\ f_X(y - 2\tau - 2R_{max})[1 - F_{X'}(y - 2\tau - R_{max})] + \\ \quad f_X(y - 2\tau - R_{max}), & 2\tau + R_{max} + S_{max} < y \le 2\tau + 2R_{max} + S_{max}, \end{cases} \tag{D.16}$$

where $F_{X'}(y)$ is the CDF of X' and $X' =_{st} X$, i.e., X' and X have the same CDF.

Table D.1 Compare the Results from Upper and Lower Bound on Service Time Y_p, and Simulations ($p_r = 0.25$)

λ	Read Resp. Time (sec)			Write Resp. Time (sec)		
	UB Approx.	LB Approx.	Sim.	UB Approx.	LB Approx.	Sim.
32.0	0.02090	0.02081	0.02087	0.04537	0.04523	0.04530
59.2	0.02412	0.02391	0.02399	0.05041	0.05007	0.05016
86.4	0.02868	0.02826	0.02847	0.05703	0.05639	0.05665
113.6	0.03538	0.03457	0.03469	0.06607	0.06495	0.06516
140.8	0.04569	0.04418	0.04453	0.07909	0.07712	0.07737
168.0	0.06279	0.05985	0.06022	0.09932	0.09574	0.09595
195.2	0.09467	0.08831	0.08837	0.13484	0.12761	0.12763
222.4	0.16880	0.15128	0.15192	0.21324	0.19456	0.19526
249.6	0.48015	0.37764	0.37772	0.53010	0.42600	0.42694

The mean read, write and overall response times can be obtained in terms of the upper bound on Y_p by using the same methodology as described in obtaining the LB Approximation. We shall refer to this as the *Upper Bound (UB) Approximation.*

Simulation Validations: To verify our models above, we have run several simulation experiments on a disk array of 16 disks. The simulation results are obtained by averaging over 40 runs, each run including 50,000 I/O requests. We obtain 95% confidence intervals whose widths are less than 2.5% of the estimated point values. The results obtained from the upper and lower bound on the service time Y_p, as well as simulations are shown in Table D.1 and D.2 for read probabilities $p_r = 0.25$ and $p_r = 0.75$, respectively. The sequential access probability p_s is set to 0.3 in these experiments. The device utilization varies from 0.11 to 0.92 in these tables.

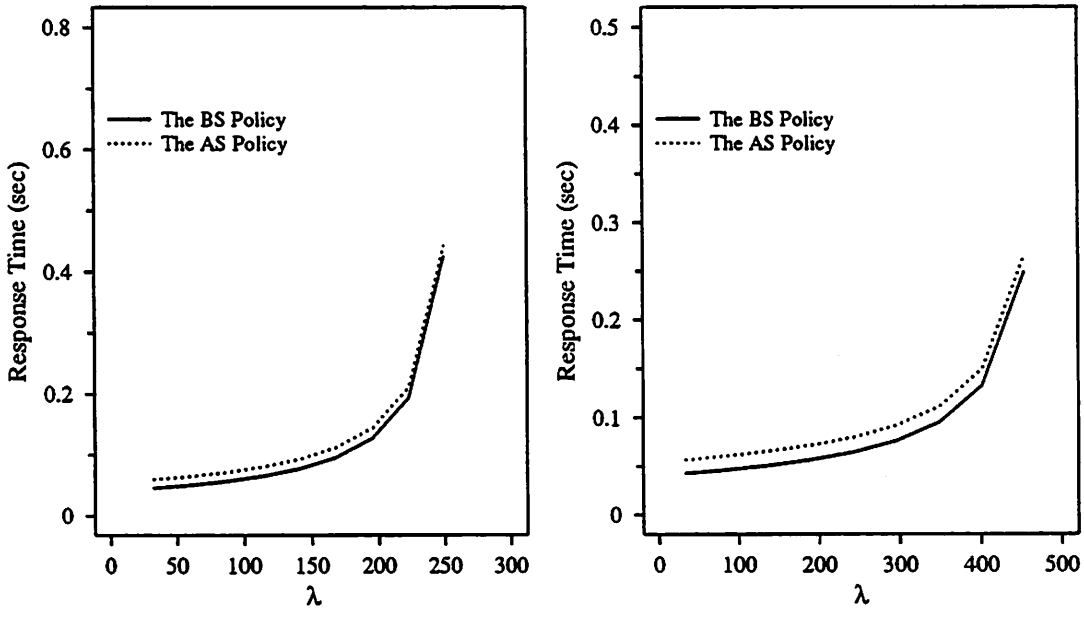
From these tables, we observe that the LB approximation, which is based on the lower bound of Y_p, estimates the performance of the true system very well.

The AR Policy

As described in Section 4.2.3, the AR policy delays generating a parity update request until the corresponding old data, which is required for the calculation of new parity, has been read out. In this case, whenever the disk containing the parity is

Table D.2 Compare the Results from Upper and Lower Bound on Service Time Y_p, and Simulations ($p_r = 0.75$)

λ	Read Resp. Time (sec)			Write Resp. Time (sec)		
	UB Approx.	LB Approx.	Sim.	UB Approx.	LB Approx.	Sim.
32.0	0.01931	0.01929	0.01931	0.04272	0.04267	0.04272
84.8	0.02160	0.02152	0.02158	0.04639	0.04625	0.04632
137.6	0.02472	0.02454	0.02466	0.05101	0.05074	0.05089
190.4	0.02912	0.02879	0.02893	0.05705	0.05657	0.05678
243.2	0.03563	0.03502	0.03515	0.06534	0.06452	0.06464
296.0	0.04593	0.04479	0.04505	0.07758	0.07616	0.07644
348.8	0.06408	0.06177	0.06208	0.09786	0.09518	0.09535
401.6	0.10302	0.09726	0.09818	0.13913	0.13290	0.13368
454.4	0.23732	0.21107	0.21049	0.27614	0.24929	0.24806



(a) Write Response Time. (p_r= 0.25)

(b) Write Response Time. (p_r= 0.75)

Figure D.1 Compare the BS and AR policies.

ready to update the parity block, the information for the calculation of new parity block is always available.

Since the only difference between the AR and the BS policies is the time at which parity update requests are generated, we conclude that the analytic model for the BS policy is also applicable to the AR policy, except that the write response time is now calculated by

$$\overline{Z_w} = \overline{Q_d} + \overline{X'} + \overline{Q_p} + \overline{X} + R_{max} + 3\tau. \tag{D.17}$$

In Figure D.1, we show the performance degradation of the AR policy compared to the BS policy, in which only the curves obtained from the lower bound of Y_p are illustrated. The mean response times for read requests are the same under both policies.

APPENDIX E

MOMENTS CALCULATIONS FOR THE BS POLICY

In this Appendix, we present the moments for service times of low and high priority requests, Y_d and Y_p, which are used in calculating the moments of queueing delays Q_d and Q_p in Appendix D.

$$\begin{aligned}
\overline{Y_d} &= p_r\left(\overline{X}+\tau\right)+p_w\left(\overline{X}+2\tau+R_{max}\right)\\
\overline{Y_d^2} &= p_r\left(\overline{X^2}+2\tau\overline{X}+\tau^2\right)+p_w\left(\overline{X^2}+2(2\tau+R_{max})\overline{X}+(2\tau+R_{max})^2\right)\\
\overline{Y_d^3} &= p_r\left(\overline{X^3}+3\tau\overline{X^2}+3\tau^2\overline{X}+\tau^3\right)+\\
&\quad p_w\left(\overline{X^3}+3(2\tau+R_{max})\overline{X^2}+3(2\tau+R_{mx})^2\overline{X}+(2\tau+R_{max})^3\right).
\end{aligned}$$

where

$$\begin{aligned}
\overline{X} &= \overline{S}+\overline{R}\\
\overline{X^2} &= \overline{S^2}+2\overline{S}\,\overline{R}+\overline{R^2}\\
\overline{X^3} &= \overline{S^3}+3\overline{S^2}\,\overline{R}+3\overline{S}\,\overline{R^2}+\overline{R^3}
\end{aligned}$$

and the moments for seek time S and rotational latency R can be easily calculated from their distributions provided in Equation (2.2) and (2.4)

For Y_p, we distinguish two cases:

Lower Bound on Y_p:

$$\begin{aligned}
\overline{L_p} &= \overline{X}+2\tau+R_{max}\\
\overline{L_p^2} &= \overline{X^2}+2(2\tau+R_{max})\overline{X}+(2\tau+R_{max})^2\\
\overline{L_p^3} &= \overline{X^3}+3(2\tau+R_{max})\overline{X^2}+3(2\tau+R_{mx})^2\overline{X}+(2\tau+R_{max})^3.
\end{aligned}$$

Upper Bound on Y_p:

$$\overline{U_p} = \overline{X}+2\tau+R_{max}+R_{max}\int_0^{S_{max}} f_X(x)\left[1-F_{X'}(x+R_{max})\right]\,dx$$

$$
\begin{aligned}
\overline{U_p^2} &= \overline{X^2} + 2(2\tau + R_{max})\overline{X} + (2\tau + R_{max})^2 + 2R_{max}\int_0^{S_{max}} x f_X(x)\,dx \\
&\quad + \left(4\tau R_{max} + 3R_{max}^2\right) F_{X'}(S_{max}) - 2R_{max}\int_0^{S_{max}} x f_X(x) F_{X'}(x + R_{max})\,dx \\
&\quad - \left((2\tau + 2R_{max})^2 - (2\tau + R_{max})^2\right)\int_0^{S_{max}} f_X(x) F_{X'}(x + R_{max})\,dx \\
\overline{U_p^3} &= \overline{X^3} + 3(2\tau + R_{max})\overline{X^2} + 3(2\tau + R_{max})^2\overline{X} + (2\tau + R_{max})^3 \\
&\quad + 3R_{max}\int_0^{S_{max}} x^2 f_X(x)\,dx + 3\left(4\tau R_{max} + 3R_{max}^2\right)\int_0^{S_{max}} x f_X(x)\,dx \\
&\quad + \left((2\tau + 2R_{max})^3 - (2\tau + R_{mx})^3\right) F_{X'}(S_{max}) \\
&\quad - 3R_{max}\int_0^{S_{max}} x^2 f_X(x) F_{X'}(x + R_{max})\,dx \\
&\quad - 3R_{max}\,(4\tau + 3R_{max})\int_0^{S_{max}} x f_X(x) F_{X'}(x + R_{max})\,dx \\
&\quad - \left((2\tau + 2R_{max})^3 - (2\tau + R_{max})^3\right)\int_0^{S_{max}} f_X(x) F_{X'}(x + R_{max})\,dx.
\end{aligned}
$$

As indicated by Equation (2.6), the *pdf* of X, $f_X(x)$, is expressed by 4th-order polynomial functions. Thus, since $X =_{st} X'$, the CDF $F_{X'}(x)$ is expressed by 5th-order polynomials, and therefore all of the above integrals can be expressed in closed form.

APPENDIX F

SEEK TIME FOR THE VERTICAL DATA LAYOUT IN CHAINED DECLUSTERING

In this Appendix, we obtain the seek distance distribution and the mean seek time for the *vertical data layout* (VL) strategy in chained declustering (see Section 4.1.1 and 4.3.1).

Under the VL strategy, one copy of data is allocated to cylinders 0 to $C/2-1$, and the other copy to cylinders $C/2$ to $C-1$. Thus if a stripe is located on cylinder j, then its duplicate is on cylinder $(j+C/2)$ mod C. The arm always moves to the copy closest to the current arm position, and the seek distance never exceeds $C/2$. Let A be a *r.v.* denoting the current arm position, and I be a *r.v.* denoting the target cylinder for a request (which corresponds to two physical cylinders), $I = 0, 1, \cdots, C/2-1$. We first obtain the distribution of A, $P\{A = i\} = a_i$. The status of the arm position can be modeled by a Markov chain $n(t)$, $n(t) = 0, 1, \cdots, C-1$. Following the assumptions made in Section 2.3 that the arm needs to move to serve a request with probability $1 - p_s$, and the arm moves to any other $C/2 - 1$ cylinders with equal probability, we have the following equations describing the stationary distribution of the Markov chain,

$$a_i = \begin{cases} \frac{2}{C-2} \sum_{j=0, j\neq i}^{\lceil C/4 \rceil - 1 + i} a_j & i = 0, \cdots, \lceil \frac{C}{4} \rceil - 1; \\ \frac{2}{C-2} \sum_{j=0, j\neq i}^{\lfloor C/4 \rfloor + i} a_j & i = \lceil \frac{C}{4} \rceil, \cdots, \frac{C}{2} - 1; \\ \frac{2}{C-2} \sum_{j=i-\lfloor C/4 \rfloor, j\neq i}^{C-1} a_j & i = \frac{C}{2}, \cdots, \frac{C}{2} + \lceil \frac{C}{4} \rceil - 1; \\ \frac{2}{C-2} \sum_{j=i-\lceil C/4 \rceil + 1, j\neq i}^{C-1} a_j & i = \frac{C}{2} + \lceil \frac{C}{4} \rceil, \cdots, C-1; \end{cases}$$

$$\sum_{i=0}^{C-1} a_i = 1$$

Solving these linear equations yields the distribution of A.

We now obtain the distribution of the seek distance. For ease of discussion, we will assume that C is a multiple of 4. Again, following the notation introduced in Section 2.3, the distribution for seek distance is $P\{D = 0\} = p_s$, and for $i > 1$

$$P\{D = i\} = P\{D = i|V = 1\}(1 - p_s) + P\{D = i|V = 0\}p_s$$

$$= \begin{cases} (1-p_s)\sum_{j=0}^{C-1} a_j \Big[P\{I = (j-i) \bmod \frac{C}{2} | A = j, V = 1\} \\ \quad + P\{I = (j+i) \bmod \frac{C}{2} | A = j, V = 1\}\Big] & i = 1, \cdots, \frac{C}{4} - 1; \\ (1-p_s)\sum_{j=0}^{C-1} a_j P\{I = (j+i) \bmod \frac{C}{2} | A = j, V = 1\} & i = \frac{C}{4}; \\ (1-p_s)\Big[\sum_{j=0}^{C/2-i-1} a_j P\{I = j+i | A = j, V = 1\} + \\ \quad \sum_{C/2+i}^{C-1} a_j P\{I = (j-i) \bmod \frac{C}{2} | A = j, V = 1\}\Big] & i = \frac{C}{4}+1, \cdots, \frac{C}{2}-1, \end{cases}$$

where V is a *r.v.* denoting whether an access is sequential ($V = 0$) or not ($V = 1$). Since we assumed that the target cylinder can be one of the $C/2 - 1$ cylinders with same probability, and is independent of the arm position, we have

$$P\{D = i\} = \begin{cases} (1-p_s)\frac{2}{C-2}\Big[\sum_{j=i}^{C-1} a_j + \sum_{j=0}^{C-i-1} a_j\Big] & i = 1, \cdots, \frac{C}{4} - 1; \\ (1-p_s)\frac{2}{C-2}\sum_{j=0}^{C-1} a_j & i = \frac{C}{4}; \\ (1-p_s)\frac{2}{C-2}\Big[\sum_{j=0}^{C/2-i-1} a_j + \sum_{j=C/2+i}^{C-1} a_j\Big] & i = \frac{C}{4}+1, \cdots, \frac{C}{2}-1. \end{cases} \tag{F.1}$$

Finally, in a similar way as in Equation (2.3), the mean seek time is

$$E[S] = \sum_{i=1}^{\frac{C}{2}-1} P\{D = i\}(a + b\sqrt{i}). \tag{F.2}$$

BIBLIOGRAPHY

[1] Abbott, R. and Garcia-Molina, H. Scheduling real-time transactions. In *Proc. ACM SIGMOD Record*, March 1988.

[2] Abbott, R. and Garcia-Molina, H. Scheduling real-time transactions with disk resident data. Technical Report CS-TR-207-89, Princeton University, Feb. 1989.

[3] Abbott, R. and Garcia-Molina, H. Scheduling I/O requests with deadlines: A performance evaluation. In *Proc. of Real-Time Systems Symposium*, Dec. 1990.

[4] Amdahl, G. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. AFIPS 1967 Spring Joint Comp. Conf.*, Atlantic City, NJ, April 1967.

[5] Avi-Itzhak, B. and Halfin, S. Non-preemptive priorities in simple fork-join queue. Technical Report 41-90, RUTCOR, Rutgers U., August 1990.

[6] Baccelli, F., Boyer, P., and Hebuterne, G. Single server queue with impatient customers. *Adv. Appl. Prob.*, 16:887–905, 1984.

[7] Baccelli, F. and Coffman Jr., E. A data base replication analysis using an M/M/m queue with service interruptions. *Performance Evaluation Review*, 11:102–107, April 1982-1983.

[8] Baccelli, F., Makowski, A., and Shwartz, A. Fork-join queues and related systems with synchronization constraints: Stochastic ordering, approximations and computable bounds. *Adv. Appl. Prob.*, 21:629–660, 1989.

[9] Baccelli, F. and Trivedi, K. S. A single server queue in a hard real-time environment. *Oper. Res. Lett.*, 4(4):161–168, 1985.

[10] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed file system. In *Proc. of the 13th Symposium on Operating Systems Principles*, Oct. 1991.

[11] Bartlett, J. A nonstop kernel. In *Proc. Eighth Symp. on Operating System Principles*, pages 22–29, 1981.

[12] Bate, G. Alternative storage technologies. In *Proc. IEEE Spring ComCon*, pages 151–157, February 1989.

[13] Bates, K. H. Performance aspects of the HSC controller. *Digital Technical Journal*, (8), Feb. 1989.

[14] Bell, C. G. The mini and micro industries. *IEEE Comp. Mag.*, 17:14–30, October 1984.

[15] Bhattacharya, P. P. and Ephremides, A. Optimal scheduling with strict deadlines. *IEEE Trans. on Automatic Control*, 34(7), July 1989.

[16] Bitton, D. and Gray, J. Disk shadowing. In *Proc. 14th VLDB Conf.*, Los Angeles, CA, August 1988.

[17] Bitton, D. Arm scheduling in shadowed disks. In *Proc. IEEE Spring ComCon*, pages 132–136, February 1989.

[18] Blanc, J. P. C., de Waal, P. R., Nain, P., and Towsley, D. Optimal control of admission to a multiserver queue with two arrival streams. *IEEE Trans. on Automatic Control*, 37(6):785–797, June 1992.

[19] Bucher, I. Y. and Hayes, A. H. I/O performance measurement on Cray-1 and CDC 7600 computers. In *Proc. Cray Users Group Conf.*, Oct. 1980.

[20] Buchmann, A. P. et al. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proc. Data Engineering Conference*, Feb. 1989.

[21] Bultman, D. L. High performance SCSI using parallel drive technology. In *Proc. BUSCON Conf.*, Anaheim, CA, February 1988.

[22] Bursky, D. Optical disk drives tackle all needs. *Electronic Design*, 37(26):57–74, October 1989.

[23] Carey, M. J., Jauhari, R., and Livny, M. Priority in DBMS resource scheduling. In *Proc. of the 15th VLDB Conf.*, 1989.

[24] Chen, P., Gibson, G., Katz, R. H., and Patterson, D. A. An evaluation of redundant arrays of disks using an Amdahl 5890. *Perf. Eval. Rev.*, 18(1):74–85, May 1990.

[25] Chen, P. and Patterson, D. A. Maximize performance in a striped disk array. In *Proc. Computer Architecture*, pages 322–331, June 1990.

[26] Chen, S.-Z., Stankovic, J. A., Kurose, J. F., and Towsley, D. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time System Journal*, 3(3):307–336, Sept. 1991.

[27] Chen, S.-Z. and Towsley, D. Raid 5 vs. parity striping: Their design and evaluation. Technical Report 92-31, COINS Dept. U. Massachusetts, Feb. 1992. To appear in Journal of Parallel and Distributed Computing.

[28] Chen, S.-Z. and Towsley, D. Performance of a mirrored disk in a real-time transaction system. In *Proc. of SIGMETRICS'91*, San Diego, CA, May 1991.

[29] Chen, S.-Z. and Towsley, D. A queueing analysis of RAID architectures. Technical Report 91-71, COINS Dept. U. Massachusetts, May 1991.

[30] Chen, S.-Z. and Towsley, D. Scheduling time constrained customers in a non-removal real-time system: with applications to disk scheduling. A forthcoming paper, 1992.

[31] Cheng, S. *Dynamic Scheduling Algorithms for Distributed Hard Real-Time Systems.* PhD thesis, Univ. of Massachusetts, 1987.

[32] Coffman Jr., E., Gelenbe, E., and Plateau, B. Optimization of the number of copies in a distributed data base. *IEEE Trans. on Software Engi.*, 7(1):78–84, Junuary 1981.

[33] Coffman Jr., E., Pollak, H., Gelenbe, E., and Wood, R. An analysis of parallel-read sequential-write systems. *Performance Evaluation*, 1:62–69, 1981.

[34] Copeland, G. and Keller, T. A comparison of high-availability media algorithms. In *Proc. ACM SIGMOD Conf.*, Portland, OR, May 1989.

[35] Cox, D. R. A use of complex probabilities in theory of stochastic processes. In *Proc. Combridge Phil. Soc., 51*, pages 313–319, 1955.

[36] Dertouzos, M. Control robotics: The procedural control of physical processes. In *Proc. of the IFIP Congress*, pages 807–813, 1974.

[37] Doshi, B. T. An M/G/1 queue with a hybrid discipline. *Bell Syst. Tech. J.*, 62(5):1251–1271, 1983.

[38] Douglis, F. and Ousterhout, J. Log-structured file systems. In *Proc. IEEE Spring ComCon.*, pages 124–129, February 1989.

[39] El Abbadi, A., Skeen, D., and Cristian, F. An efficient fault-tolerant protocol for replicated data management. In *Proc. ACM-SIGACT SIGMOD Conf. on Principles of Database Systems*, Waterloo, Canada, March 1985.

[40] Flatto, L. and Hahn, S. Two parallel queues created by arrivals with two demands I. *SIAM J. Appl. Math.*, 44:1041–1053, 1984.

[41] Garcia-Molina, H. and Lipton, R. J. A massive memory machine. *IEEE Transactions on Computers*, C-33 (5):391–399, May 1984.

[42] Gibson, G. A. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage.* PhD thesis, University of California, Berkeley, April 1991.

[43] Goli, P., Kurose, J., and Towsley, D. Approximate minimum laxity scheduling algorithms for real-time systems. Technical Report 90-88, COINS Dept. U. Massachusetts, Sept. 1990.

[44] Gray, J., Host, B., and Walker, M. Parity striping of disk arrays: Low-cost reliable storage with acceptable throughput. In *Proc. 16th VLDB Conf.*, pages 148–161, Brisbane, Australia, 1990.

[45] Grossman, C. Evolution of the DASD control. *IBM Systems Journal*, 28(2), 1989.

[46] Harker, J. M. et al. A quater century of disk file innovation. *IBM J. Res. Dev.*, 25:677–689, September 1981.

[47] Heyman, D. P. and Sobel, M. J. *Stochastic Models in Operations Research*, volume I. McGraw Hill, 1982.

[48] Hofri, M. Disk scheduling: FCFS vs. SSTF revisited. *ACM Communications*, 23(11), Nov. 1980.

[49] Hong, J., Tan, X., and Towsley, D. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Trans. on Computer*, 38(12):1736–1744, Dec. 1989.

[50] Howard, J. et al. Scale and performance in a distributed file system. *ACM Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[51] Hsiao, H. and DeWitt, D. Chained declustering: A new availibility strategy for multiprocessor database machines. In *Proc. 6th Int'l Conf. on Data Engineering*, Los Angeles, CA, Feb. 1990.

[52] Hsiao, H. and DeWitt, D. A performance study of three high availability data replication strategies. Technical Report RC 16690, IBM, Feb. 1991.

[53] Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K. Experimental evaluation of real-time transaction processing. In *Proc. Real-Time System Symposium*, pages 144–153, Dec. 1989.

[54] Joy, W. Presentation at ISSCC'85 Panel Session, February 1985.

[55] Katz, R. H., Gibson, G., and Patterson, D. A. Disk system architectures for high performance computing. *Proc. of the IEEE*, 77(12):1842–1858, December 1989.

[56] Katzman, J. A fault-tolerant computing system. In Siewiorek, D. and Swarz, R., editors, *The Theory and Practice of Reliable System Design*. Digital Press, 1982.

[57] Khinchine, A. Y. *Mathematical Methods in the Theory of Queueing*. Hafner Publishing Co., New York, 1960.

[58] Kim, C. and Agrawala, A. Analysis of a fork-join queue. *IEEE Trans. on Comp.*, C-38 (2):250–255, February 1989.

[59] Kim, M. Y. Synchronized disk interleaving. *IEEE Trans. on Comp.*, C-35 (11):978–988, Nov. 1986.

[60] Kim, M. Y. and Tantawi, A. N. Asynchronized disk interleaving: Approximating access delays. *IEEE Trans. on Comp.*, C-40 (7):801–810, July 1991.

[61] Kleinrock, L. *Queueing Systems*, volume 1. John Wiley, New York, 1975.

[62] Kleinrock, L. *Queueing Systems*, volume 2. John Wiley, New York, 1976.

[63] Lazowska, E. D., Zahorjan, J., Cheriton, D. R., and Zwaenepoel, W. File access performance of diskless workstations. *Transaction on Computer Systems*, 4(3):238–268, Aug. 1986.

[64] Lee, E. K. and Katz, R. H. Performance consequences of parity placement in disk arrays. In *Proc. 4th Int'l Conf. on Archi. Sup. for Prog. Lang. and OS*, pages 190–199, April 1991.

[65] Little, J. D. C. A proof of the queueing formula $l = \lambda w$. *Operations Research*, 9:383–387, 1961.

[66] Livny, M., Khoshafian, S., and Boral, H. Multi-disk management algorithm. In *Proc. SIGMETRICS Conf.*, pages 69–77, May 1987.

[67] Locke, C. D., Tokuda, H., and Jensen, E. D. A time-driven scheduling model for real-time operating system. Technical report, CMU, 1985.

[68] Lynch, W. C. Do disk arms move? *Perform. Eval. Rev.*, 1:3–16, December 1972.

[69] Matloff, N. S. A multiple-disk system for both fault tolerance and improved performance. *IEEE Trans. on Reliability*, 36(2):199–201, June 1987.

[70] Matloff, N. S. and Lo, R. W. A 'Greedy' approach to the write problem in shadowed disk systems. In *Proc. 6th Int'l. Conf. on Data Engineering*, February 1990.

[71] McKusick, M. et al. A fast file system for UNIX. *ACM Trans. on Computer Systems*, 2(3):181–197, Aug. 1984.

[72] Menon, J. and Kasson, J. Methods for improved update performance of disk arrays. Technical Report RJ 6928 (66034), IBM Almaden Research Center, Nov. 1990.

[73] Menon, J. and Mattson, D. Comparison of sparing alternatives for disk arrays. Technical Report RJ 8431 (76439), IBM Almaden Research Center, October 1991.

[74] Menon, J., Mattson, D., and Ng, S. Distributed sparing for improved performance for disk arrays. Technical Report RJ 7943 (72926), IBM Almaden Research Center, Jan. 1991.

[75] Menon, J., Mattson, D., and Ng, S. Performance of disk arrays in transaction processing environments. Technical Report RJ 8230 (75424), IBM Almaden Research Center, July 1991.

[76] Merchant, A. and Yu, P. S. Modeling and comparisons of striping strategies in data replication architectures. Technical report, IBM Thomas J. Watson Research Center, Sept. 1991.

[77] Muntz, R. R. and Lui, J. C. Performance analysis of disk arrays under failure. In *Proc. 16th VLDB Conf.*, pages 162–173, Brisbane, Australia, 1990.

[78] Nelson, M., Welch, B., and Ousterhout, J. Caching in the Sprite network file system. *ACM Trans. on Comp. Sys.*, 6(1):134–154, February 1988.

[79] Nelson, R. D. and Iyer, B. R. Analysis of a replicated data base. *Performance Evaluation*, 5:133–148, 1985.

[80] Nelson, R. D. and Tantawi, A. N. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Trans. Computers*, 37:739–743, 1988.

[81] Neuts, M. F. *Matrix-Geometric Solutions in Stochastic Models - An Algorithmic Approach.* John Hopkins University Press, 1981.

[82] Ng, S. Some design issues of disk array. In *Proc. IEEE Spring ComCon*, pages 137–142, 1989.

[83] Ng, S. Improving disk performance via latency reduction. *IEEE Trans. Computers*, 40(1):22–30, 1991.

[84] Ng, S., Lang, D., and Selinger, R. Trade-offs between devices and paths in achieving disk interleaving. In *Proc. 15th Intern. Sym. on Comp. Archit.*, Hawaii, May 1988.

[85] Ogata, M. and Flynn, M. A queueing analysis for disk array systems. Technical Report CSL-TR-90-443, Stanford Univ., August 1990.

[86] Olson, T. M. Disk array performance in a random I/O environment. *Comput. Archit. News*, 17(5):71–77, September 1989.

[87] Ousterhout, J. K. and Douglis, F. Beating the I/O bottleneck: A case for log-structured file systems. *Operating System Rev.*, 23(1):11–28, Jan. 1989.

[88] Ousterhout, J. K. et al. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. 10th Sym. on Operating Systems Principles*, pages 15–24, Dec. 1985.

[89] Panwar, S. S. and Towsley, D. On the optimality of the STE rule for multiple server queue that serve customers with deadlines. Technical Report 88-81, COINS Dept. U. Massachusetts, July 1988.

[90] Panwar, S. S., Towsley, D., and Wolf, J. K. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. *J. ACM*, 35(4):832–844, Oct. 1988.

[91] Patterson, D. A., Chen, P., Gibson, G., and Katz, R. H. Introduction to redundant arrays of inexpensive disks (RAID). In *Proc. IEEE Spring ComCon.*, pages 112–117, 1989.

[92] Patterson, D. A., Gibson, G., and Katz, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD Conf.*, pages 109–116, July 1988.

[93] Pinedo, M. Stochastic scheduling with release dates and due dates. *Operations Research*, 31:559–572, 1983.

[94] Poston, A. A high performance file system for UNIX. In *Proc. USENIX*, pages 215–226, September 1988.

[95] Rao, B. M. and Posner, M. J. M. Algorithmic and approximation analyses of the split and match queue. *Stochastic Models*, 1:433–456, 1985.

[96] Rao, B. M. and Posner, M. J. M. Algorithmic and approximation analyses of the shortest queue model. *Naval Research Logistic*, 24:381–398, 1987.

[97] Reddy, A. L. N. and Banerjee, P. An evaluation of multiple-disk I/O systems. *IEEE Trans. Comp.*, 38(12), December 1989.

[98] Reddy, A. L. N. and Banerjee, P. Gracefully degradable disk arrays. In *Proc. of FTCS-21*, June 1991.

[99] Rosenblum, M. and Ousterhout, J. K. The LFS storage manager. In *Proc. Summer '90 USENIX Tech. Conf.*, pages 315–324, Anaheim, CA, June 1990.

[100] Rosenblum, M. and Ousterhout, J. K. The design and implementation of a log-structured fils system. Draft, March 1991.

[101] Salem, K. and Garcia-Molina, H. Disk striping. In *Proc. IEEE Data Engineering Conf.*, Los Angeles, CA, February 1986.

[102] Satyanarayanan, M. A study of file sizes and functional lifetimes. In *Proc. 8th Symposium of Operating Systems Principles*, pages 96–108, 1986.

[103] Schroeder, M., Gifford, D., and Needham, R. A caching file system for a programmer's workstation. In *Proc. 10th Sym. on Operating System Principles*, pages 25–34, December 1985.

[104] Schulze, M., Gibson, G., Katz, R., and Patterson, D. How reliable is a RAID? In *Proc. IEEE Spring ComCon Conf.*, San Francisco, CA, February 1989.

[105] Scranton, R. A. and Thompson, D. A. The access time myth. Technical Report RC 10197, IBM, Sept. 1983.

[106] Smith, A. J. Optimization of I/O systems by cache disks and file migration: A summary. *Performance Evaluation*, 1:249–262, 1981.

[107] Smith, A. J. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, Aug. 1985.

[108] Son, S. H. and Chang, C. H. Priority-based scheduling in real-time database systems. In *Proc. of the 15th VLDB Conference*, 1989.

[109] Srinivasan, V. and Mogul, J. C. Spritely NFS: Experiments with cache-consistency protocols. In *Proc. 12th Symposium on Operating Systems Principles*, pages 45–57, Litchfield Park, Arizona, Dec. 1989.

[110] Stankovic, J. A. and Ramamritham, K. *Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[111] Stonebraker, M. and Schloss, G. A. Distributed RAID – a new multiple copy algorithm. In *Proc. 6th Int'l. Conf. on Data Engineering*, February 1990.

[112] Stoyan, D. *Comparison methods for queues and other stochastic models.* John Wiley & Sons, Chichester England, 1983.

[113] Su, Z.-S. and Sevcik, K. C. A combinatorial approach to scheduling problems. *Operations Research*, 26:836–844, 1978.

[114] Teorey, T. J. and Pinkerton, T. B. A comparative analysis of disk scheduling policies. *ACM Communications*, 5(3):177–184, March 1972.

[115] Teradata Corp. *DBC/1012 Database computer system manual release 2.0*, Nov. 1985.

[116] Towsley, D. and Chen, S.-Z. Design and modeling scheduling policies for two server fork/join queueing systems. submitted to Performance Evaluation J., March 1991.

[117] Towsley, D., Chen, S.-Z., and Yu, S.-P. Performance analysis of a fault tolerant mirrored disk system. In *Proc. PERFORMANCE'90*, pages 239–253, Edinburgh, Scotland, September 1990.

[118] Towsley, D. and Panwar, S. S. On the optimality of minimum laxity and earliest deadline scheduling for real-time multiprocessors. In *Proc. Euromicro'90 Workshop on Real-Time Sys.*, pages 17–24, June 1990.

[119] Warchol, N. A. and Shirron, S. F. The RQDX3 design project. *Digital Technical Journal*, (2), March 1986.

[120] Welch, T. A. Analysis of memory hierarchies for sequential data access. *IEEE Computer*, 12(5):19–26, 1979.

[121] Wu, Z. J., Luh, P. B., Chang, S. C., and Castanon, D. A. Optimal control of a queueing system with two interacting service stations and three classes of impatient tasks. *IEEE Trans. on Automat. Contr.*, 33:42–49, 1988.

[122] Zhao, W., Ramamritham, K., and Stankovic, J. A. Preemptive scheduling under time and resource constraints. *IEEE Trans. on Computers*, pages 949–960, Aug. 1987.

[123] Zhao, W., Ramamritham, K., and Stankovic, J. A. Scheduling tasks with resource requirements in hard real-time systems. *IEEE Trans. on Software Engineering*, SE-12(5), 1987.