SCHEDULING REAL-TIME TRANSACTIONS: A PERFORMANCE EVALUATION

Robert Kilburn Abbott

CS-TR-331-91

October 1991

Scheduling Real-Time Transactions: a Performance Evaluation

Robert Kilburn Abbott

A DISSERTATION

PRESENTED TO THE FACULTY

OF PRINCETON UNIVERSITY

IN CANDIDACY FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE

BY THE DEPARTMENT OF

COMPUTER SCIENCE

October 1991

Dedicated with love and appreciation to my parents
Royal and Mary Abbott

Scheduling Real-Time Transactions: a Performance Evaluation

Robert Kilburn Abbott

Thesis Advisor - Professor Hector Garcia-Molina

This thesis has six chapters. Chapter 1 motivates the thesis by describing the characteristics of real-time database systems and the problems of scheduling transactions with deadlines. We also present a short survey of related work and discuss how this thesis has contributed to the state of the art.

In Chapter 2 we develop a new family of algorithms for scheduling real-time transactions. Our algorithms have four components: a policy to manage overloads, a policy for scheduling the CPU, a policy for scheduling access to data, i.e., concurrency control and a policy for scheduling I/O requests on a disk device.

In Chapter 3, our scheduling algorithms are evaluated via simulation. Our chief result is that real-time scheduling algorithms can perform significantly better than a conventional non real-time algorithm. In particular, the Least Slack (static evaluation) policy for scheduling the CPU, combined with the Wait Promote policy for concurrency control, produces the best overall performance.

In Chapter 4 we develop a new set of algorithms for scheduling disk I/O requests with deadlines. Our model assumes the existence of a real-time database system which assigns deadlines to individual read and write requests. We also propose new techniques for handling requests without deadlines and requests with deadlines simultaneously.

This approach greatly improves the performance of the algorithms and their ability to minimize missed deadlines.

In Chapter 5 we evaluate the I/O scheduling algorithms using detailed simulation. Our chief result is that real-time disk scheduling algorithms can perform better than conventional algorithms. In particular, our algorithm FD-SCAN was found to be very effective across a wide range of experiments.

Finally, in Chapter 6 we summarize our conclusions and discuss how this work has contributed to the state of the art. Also, we briefly explore some interesting new directions for continuing this research.

Acknowledgments

I am truly fortunate to have received the love and support of so many friends and colleagues throughout my graduate career and it would be impossible to properly acknowledge them all. The Computer Science Department at Princeton University is an exciting, rewarding place to study. Many thanks to the faculty, graduate students and staff who make it so.

I want especially to thank my advisor Hector Garcia-Molina, an exceptional researcher and teacher, who introduced me to real-time systems as well as database systems and distributed computing. His guidance, patience and contagious enthusiasm are greatly appreciated. My readers, Kai Li and Rafael Alonso, supplied constructive criticism that greatly improved the final thesis.

My colleagues at Digital Equipment Corp., particularly Walt Kohler, Linda Wright and Phil Bernstein, deserve a special note of recognition. Their patience, encouragement and flexibility are greatly appreciated. They also make Digital a challenging and fun place to work.

Many friends enriched my stay at Princeton University: Mordecai Golin, Krishna Balasubramanian, Richard Squier, Michael Laszlo and Ray Najjar. I am fortunate to have known you.

Finally, I would like to thank Susannah Wolfson, whose unwavering love, support and friendship have sustained me these past six years.

Robert K. Abbott
Littleton, Massachusetts
May 1991

Table of Contents

Figures

Tables

Chapter 1 Real-Time Database Systems

1.1 Thesis Outline

This thesis describes a new group of algorithms for scheduling real-time transactions, or transactions with deadlines. The algorithms are evaluated via a detailed simulation study. Our results show that under a wide range of workloads, the real-time algorithms perform significantly better than a conventional transaction scheduling algorithm. This thesis also studies the problem of scheduling disk requests with deadlines. Three new algorithms are proposed and their performance is evaluated via simulation. One real-time algorithm, FD-SCAN, was found to perform better than all algorithms tested, both real-time and conventional, under a wide range of experimental conditions.

There are six chapters. Chapter 1 motivates the thesis by describing the characteristics of real-time database systems and the problems of scheduling transactions with deadlines. We also present a short survey of related work and discuss how this thesis has contributed to the state of the art.

In Chapter 2 we develop a new family of algorithms for scheduling real-time transactions. Our algorithms have four components: a policy to manage overloads, a policy for scheduling the CPU, a policy for scheduling access to data, i.e., concurrency control and a policy for scheduling I/O requests on a disk device. In this part of the thesis, a disk device is modeled using a constant access time. Chapter 4 develops a more detailed model for investigating the I/O scheduling problem and uses a non-linear function to model access time.

In Chapter 3, our scheduling algorithms are evaluated via detailed simulation. Our chief result is that real-time scheduling algorithms can perform significantly better than a conventional non real-time algorithm. In particular, the Least Slack (static evalu-

ation) policy for scheduling the CPU, combined with the Wait Promote policy for concurrency control, produces the best overall performance.

In Chapter 4 we develop a new set of algorithms for scheduling disk I/O requests with deadlines. The performance goal is to minimize the number of missed deadlines. Our model assumes the existence of a real-time database system which assigns deadlines to individual read and write requests. We also propose new techniques for handling asynchronous write requests (i.e., requests without deadlines) and synchronous read requests (i.e., requests with deadlines) simultaneously. This approach greatly improves the performance of the algorithms and their ability to minimize missed deadlines.

In Chapter 5 we evaluate the I/O scheduling algorithms using detailed simulation. Our chief result is that real-time disk scheduling algorithms can perform better than conventional algorithms. In particular, our algorithm FD-SCAN was found to be very effective across a wide range of experiments.

Finally, in Chapter 6 we summarize our conclusions and discuss how this work has contributed to the state of the art. Also, we briefly explore some interesting new directions for continuing this research.

This thesis is a combination of original work and ideas proposed by others. We believe that our proposal to study algorithms to schedule transactions with deadlines is original. We made this proposal at a workshop in 1987 [**AGM1**]. The overload management policies discussed in Chapter 2, Not Tardy and Feasible Deadlines, implement ideas that are common to hard real-time systems, i.e., discard tasks that have missed or will miss their deadlines. The application of these ideas to transaction processing is original. The three policies that we use to assign priority and schedule the CPU have been studied extensively by real-time researchers [**LL, JLT**]. The use and evaluation of

these policies in a transaction processing environment is original. Concerning the concurrency control algorithms, the use of two-phase locking is well-known. [**EGLT**] and the Wait policy is the conventional way to handle conflicts in a non real-time database system. The concept of priority inheritance that appears in our Wait Promote and Conditional Restart policies first appeared in [**SRL1**]. The High Priority policy is original as is the conditional decision clause of the Conditional Restart policy. The application and evaluation of all these policies in a real-time transaction model is original. The three conventional algorithms for scheduling disk I/O in Chapter 4 are well known [**PS**]. Their evaluation in a real-time environment is original. The D-SCAN and FD-SCAN algorithms are original.

1.2 Introduction

Transactions in a database system can have real-time constraints. Consider for example program trading, or the use of computer programs to initiate trades in a financial market with little or no human intervention [**Vo**]. A financial market (e.g., a stock market) is a complex process whose state is partially captured by variables such as current stock prices, changes in stock prices, volume of trading, trends, and composite indexes. These variables and others can be stored and organized in a database to model a financial market.

One type of process in this system is a sensor/input process which monitors the state of the physical system (i.e. the stock market) and updates the database with new information. If the database is to contain an accurate representation of the current market then this monitoring process must meet certain real-time constraints.

A second type of process is an analysis/output process. In general terms this process reads and analyzes database information in order to respond to a user query or

to initiate a trade in the stock market. An example of this is a query to discover the current bid and ask prices of a particular stock. This query may have a real-time response requirement of say 2 seconds. Another example is a program that searches the database for arbitrage opportunities. Arbitrage trading involves finding discrepancies in prices for objects, often on different markets. For example, an ounce of silver might sell for $10 in London and fetch $10.50 in Chicago. Price discrepancies are normally very short-lived and to exploit them one must trade large volumes on a moments notice. Thus the detection and exploitation of these arbitrage opportunities is certainly a real-time task.

Another kind of real-time database system involves threat analysis. For example, a system may consist of a radar to track objects and a computer to perform some image processing and control. A radar signature is collected and compared against a database of signatures of known objects. The data collection and signature look up must be done in real-time.

A *real-time database system* (RTDBS) has many similarities with conventional database management systems and with so called real-time systems. However, a RTDBS lies at the interface and is not quite the same as either type of conventional system. Like a database system, a RTDBS must process *transactions* and guarantee that the database consistency is not violated. Conventional database systems do not emphasize the notion of time constraints or deadlines for transactions. The performance goal of a system is usually expressed in terms of desired *average* response times rather than constraints for individual transactions. Thus, when the system makes scheduling decisions (e.g., which transaction gets a lock, which transaction is aborted), individual real-time constraints are ignored.

Conventional real-time systems account for individual transaction constraints but ignore data consistency problems. Real-time systems typically deal with simple transactions (called processes) that have simple and predictable data (or resource) requirements. For a RTDBS we assume that transactions make unpredictable data accesses (by far the more common situation in a database system). This makes the scheduling problem much harder and leads to another difference between a conventional real-time system and a RTDBS. The former usually attempts to ensure that *no* time constraints are violated, i.e., constraints are viewed as "hard" [**Mok1**]. In a RTDBS, on the other hand, it is difficult to guarantee all time constraints, so we strive to *minimize* the ones that are violated.

In the previous paragraphs we have "defined" what we mean by a RTDBS (our definition will be made more precise in Chapter 2). However, note that other definitions and assumptions are possible. For instance, one could decide to have hard time constraints and instead minimize the number of data consistency violations. However, we believe that the type of RTDBS that we have sketched better matches the needs of applications like the ones mentioned earlier. For instance, in the financial market example, it is probably best to miss a few good trading opportunities rather than permanently compromise the correctness of the database, or restrict the types of transactions that can be run.

We should at this point make two comments about RTDBS applications. It may be argued that real-time applications do not access databases because they are "too slow." This is a version of the "chicken and the egg" problem. Current database systems have few real-time facilities, and hence cannot provide the service needed for real-time applications. The way to break the cycle is by studying a RTDBS, designing the proper

facilities, and evaluating the performance (e.g., what is the price to be paid for serializability?).

It is also important to note that with good real-time facilities, even applications one does not typically consider "real-time" may benefit. For example, consider a banking transaction processing system. In addition to meeting average response time requirements, it may be advantageous to tell the system the urgency of each transaction so it can be processed with the corresponding priority. As a matter of fact, a "real" banking system may already have some of these facilities, but not provided in a coherent fashion by the database management system.

The design and evaluation of a RTDBS presents many new and challenging problems: What is the best data model? How can we model transaction time constraints? What mechanisms are needed for describing and evaluating triggers (a trigger is an event or a condition in the database that causes some action to occur)? How are transactions scheduled? How do the real-time constraints affect concurrency control? Should transaction time constraints be considered when scheduling I/O requests?

In this thesis we focus on the last three questions. In particular, if several transactions are ready to execute at a given time, which one runs first? If a transaction requests a lock held by another transaction, do we abort the holder if the requester has greater urgency? If transactions can provide an estimate of their running time, can we use it to tell which transaction is closest to missing a deadline and hence should be given higher priority? If we do use runtime estimates, what happens if they are incorrect? How are the various strategies affected by the load, the number of database conflicts, and the tightness of the deadlines? Finally, there is the problem of scheduling the disk head itself. Traditional algorithms perform seek optimization to meet non real-time perform-

ance goals. Will these same algorithms perform well under real-time metrics? What kinds of algorithms can be developed using deadline information? How well do they perform? Should read requests be handled differently from write requests? How should the requests be sequenced so that time constraints are met and the disk resource is used efficiently?

1.3 Related Work

In recent years, a number of papers have been published on issues in real-time database systems. [**AGM1**, **AGM2**, **AGM3**, **AGM4**, **Day**, **HSRT1**, **HSRT2**, **HSTR**, **CJL1**, **HCL1**, **HCL2**, **HS**, **LL**, **LS**, **SLJ**, **SRL2**, **SZ, JCL**]. The subjects that are addressed in these papers include the identification and description of time-constrained database systems and real-time transactions [**AGM1**, **Day**, **HSRT**, **SZ**, **LL**, **SRL2**], real-time transaction scheduling [**AGM2**, **AGM3 AGM4**, **HSTR**, **SRL2**], concurrency control and conflict resolution [**AGM3**, **AGM4**, **HCL1**, **HCL2**, **HSRT1**, **HSRT2**, **HSTR**, **LS**, **SLJ**, **SRL2**], buffer management [**CJL1**, **HS**, **JCL**] and I/O scheduling [**AGM3**, **AGM4**, **CJL1**]. A number of papers on related issues have also appeared. These include work on a protocol for timed atomic commitment [**DLW**], fast recovery protocols for real-time databases [**IYL**, **LYI**, **SLJ**] and a model for adding time to synchronous process communications [**LD**]. The feasibility of the relational languages for real-time process control is examined in [**Stok**].

Much of the research in the area of scheduling real-time transactions was pursued independently and concurrently. We published our first concept papers in 1987 [**AGM1**] and 1988 [**AGM2**] and followed with papers presenting algorithms and performance analysis in 1988 [**AGM3**], 1989 [**AGM4**] and 1990 [**AGM5**]. Both Sha et al. and Stankovic et al. published their first papers in the same years (and even the same

proceedings !) [**SRL2**] [**SZ**]. Stankovic et al. published their first performance results in 1989 [**HSTR**] and followed with papers in 1990 [**HSRT1**] and 1991 [**HSRT2**]. while the group from the University of Wisconsin published their performance results in 1990 [**HCL1**, **HCL2**].

The work on transaction scheduling and concurrency control found in [**SRL2**, **HCL1**, **HCL2**, **HSTR**, **HSRT1**, **HSRT2**, **LS**] is most similar to our own work. However, there are some important differences in the transaction models that are employed and the way that concurrency control is achieved. Our model, like those found in [**HCL1**, **HCL2**, **HSTR**, **HSRT1**, **LS**], assumes that transactions arrive sporadically with unpredictable arrival times and resource requirements. Furthermore, the data requirements of each transaction are unknown although a worst case execution time may be available to the scheduler. Our algorithms use a form of two-phase locking to enforce serializability. Other researchers have examined optimistic concurrency control methods [**HCL1**, **HCL2**, **LS**, **HSRT2**].

Sha et al. present algorithms for scheduling a fixed set of periodic transactions with hard deadlines [**SRL2**]. Their model assumes that transaction priorities and resource requirements are known *a priori*. The rate-monotonic algorithm is used for determining transaction priority and scheduling the CPU. A priority ceiling protocol based on locking is used for concurrency control. The priority ceiling algorithm appears to have promise for the hard real-time environment since it prevents deadlock formation and strictly bounds transaction blocking times. The price, however, is *a priori* knowledge of transaction priorities and resource requirements.

The use of optimistic concurrency control techniques is explored in [**HCL1**, **HCL2**]. Their simulation studies show that in a system where late transactions are dis-

carded, optimistic concurrency control can perform better than locking. The High Priority real-time concurrency control algorithm [**AGM3**] was used as the locking algorithm. This thesis shows that there are better locking -based protocols than High Priority.

A mixed concurrency control algorithm that combines locking and optimistic techniques is presented in [**LS**]. Priority-based locking is used to guide the serialization order during execution. A final validation is performed at the transaction commit point. No performance evaluation was presented.

The model and algorithms developed in [**HSTR**] are the only ones that are directly comparable to those developed in this thesis. Both models assume that transactions arrive sporadically and have unpredictable resource requirements. The scheduler is aware of a worst case execution time for each transaction but it does not know what its data requirements are. Also, both sets of algorithms use two-phase locking to enforce serializability. Their transaction model uses a value function in addition to a deadline to capture transaction time constraints. A transactions criticality is derived from the value function, i.e., the highest value. They study three priority functions for CPU scheduling: earliest deadline first, most critical first and a function that combines criticalness and deadlines. Five methods for conflict resolution (concurrency control) are considered. One protocol makes use of a virtual clock that is assigned to each transaction. This clock is used to compute a virtual deadline that is used in scheduling decisions. The other four conflict resolution protocols make decisions based on information about transaction deadlines, criticalness and estimation of remaining execution time. In Section 3.5 we will return to this discussion and examine how the results of [**HSTR**] compare with our own results.

Chapter 2 Algorithms for Scheduling Real-Time Transactions

2.1 Introduction

This chapter begins by describing our real-time transaction model and stating our assumptions, Section 2.2. In Sections 2.3 through 2.7 we develop a new family of algorithms for scheduling real-time transactions. Our scheduling algorithms have four components: a policy to manage overloads, a policy for assigning priorities to tasks and scheduling the CPU, a concurrency control mechanism, and a policy for scheduling I/O requests. In this part of the thesis a very simple model for I/O is used. Chapter 4 develops a more detailed model for investigating the I/O scheduling problem.

2.2 Model and Assumptions

The system consists of a *single processor*, a disk-based database, and a main memory buffer pool. (The multiple processor case is also of interest, but we have not addressed it in this thesis.)

The unit of database granularity we consider is a *page*. Transactions access a sequence of pages. If a page is not found in the buffer pool, a disk read is initiated to transfer the page to the pool. Modified pages are held in the pool until the transaction completes. At that time, the log is flushed and the transaction commits. Finally, the modified pages are written back to disk to free space in the buffer pool. (This buffer management strategy can be characterized as ¬ATOMIC, ¬STEAL and FORCE [**HR**].) We assume that the buffer pool is large enough so that a transaction never has to write modified pages to disk until after commit. Thus, aborting a transaction involves no disk writes. We assume that the log is kept on a disk (or tape) separate from the database disks (the most common scenario in practice).

Each arriving transaction has a release time r, a deadline d, and a runtime estimate E. The release time is the earliest time the transaction can be started and is usually the arrival time. The deadline is the desired maximum commit time. Estimate E approximates the duration of the transaction on an unloaded system. It takes into account both the CPU and disk access time involved. Parameters r, d, and E are known to the system as soon as the transaction arrives. However, the access pattern of the transaction is not known in advance. As the transaction executes, it asks to read or write one page at a time. Our decision to assume knowledge of a runtime estimate but no knowledge of data requirements is justified because it is easier to estimate the execution time of a transaction than to predict its data access pattern. In any case, E is simply an estimate that could be wrong or not given at all.

The RTDB system schedules transactions with the objective of minimizing the number of missed deadlines. If transactions can miss their deadlines, one must address the question of what happens to transactions that have already missed their deadlines but have not finished yet. There are at least two alternatives. One is to assume that a transaction that has missed its deadline, i.e., is tardy, is worthless and can be aborted. This may be reasonable in our arbitrage example. Suppose that a transaction is submitted to buy and sell silver by 11:00am. If the deadline is missed, it may be best not to perform the operation at all; after all, the conditions that triggered the decision to go ahead may have changed. The user who submitted the transaction may wish to reconsider the operation.

A second option is to assume that all transactions must be completed eventually, regardless of whether they are tardy or not. This may be the correct mode of operation in, say, a banking system where customers would rather do the transaction late than not at all. (Of course, the user may on his own decide to abort his transaction, but this is

another matter.) If tardy transactions must be executed, there is still the question of their priority. Tardy transactions could receive higher and higher urgency as their tardiness increases. On the other hand, since they already missed the deadline anyway, they may simply be postponed to a later, more convenient time (e.g., execute at night).

In this thesis we will study both cases, when tardy transactions must be completed and when they can be aborted. If they must complete, we will assume that their priority increases as the tardiness increases (they are not put off). (Incidentally, [**AGM1**] discusses a more detailed deadline model where users can specify how the "value" of a transaction changes over time, both as the deadline approaches and passes.)

We assume that transaction executions must be serializable [**EGLT**]. For most applications we believe that it is desirable to maintain database consistency. It is possible to maintain consistency without serializable schedules but this requires more specific information about the kinds of transactions being executed [**GM**]. Since we have assumed very little knowledge about transactions, serializability is the best way to achieve consistency.

Finally, we assume that serializability is enforced by using a locking protocol. Our purpose is not to do a comparative study of concurrency mechanisms. Instead we have chosen a well-understood and widely-used mechanism and explored the different ways that transactions can be scheduled using this mechanism. Of course, it is conceivable that some other algorithm, like an optimistic protocol, may be better for a RTDBS, but this will have to be addressed by further research.

2.3 Algorithms

Our scheduling algorithms have four components: a policy to manage overloads, a policy for assigning priorities to tasks, a concurrency control mechanism, and a policy

for scheduling I/O requests. In a real-time system, we say that an overload occurs whenever transaction timing constraints are violated. (Note that this definition has no relation to resource utilization metrics. It is possible that a lightly loaded system, measured by resource utilizations, misses many deadlines and a highly loaded system can miss few. By real-time standards, it is the lightly utilized system that is overloaded.) The overload management policy is used to detect when overloads occur and initiate actions to handle the overload. The priority assignment policy controls how transaction time constraints are used to assign a priority to a transaction. The concurrency control mechanism can be thought of as a policy for resolving conflicts between two (or more) transactions that want to lock the same data object. Some concurrency control mechanisms permit deadlocks to occur. For these a deadlock detection and resolution mechanism is needed. The fourth component controls how scheduling of the I/O queue is done, i.e., whether a transaction's real-time constraints are used to decide which I/O request is serviced next.

Each component may use only some of the available information about a transaction. In particular we distinguish between policies which do not make use of E, the runtime estimate, and those that do. A goal of our research is to understand how the accuracy of the runtime estimate affects the algorithms that use it.

2.4 Managing Overloads

There are a number of ways both to detect and to handle overloads. A detection method can be *observant* or *predictive*. An observant method simply examines all unfinished transactions and determines if any has missed its deadline. A predictive technique would build a candidate schedule and then determine if a transaction will miss its deadline if executed under that schedule.

Another issue concerns what actions to take when an overload is detected. Possibilities include aborting transactions which are thought to "cause" the overload, and/or executing alternative transactions. This thesis does not examine the issue of executing alternative transactions.

Finally, there is the question of how often the overload management module is invoked. Our solution is to call the overload detector whenever the scheduler is invoked. The detector runs first and if an overload is detected, the overload management routine is called. When this finishes, control passes to the scheduler which chooses a new job for the CPU.

We consider three different policies for managing overloads.

2.4.1 All Eligible

Under this policy no overload detection is performed. This means that no job is unilaterally aborted and all jobs are eventually executed.

2.4.2 Not Tardy

An overload is detected if an unfinished transaction has missed its deadline. Transactions which have missed their deadlines are aborted. Jobs which currently are not tardy remain eligible for service. Note that this detection method is observant.

2.4.3 Feasible Deadlines

An overload is detected if an unfinished transaction has an infeasible deadline. A transaction T has a infeasible deadline at time t if $t + E - P > d$ where P is the amount of service time that T has received. In other words, based on the runtime estimate there is not enough time to complete the transaction before its deadline. Jobs with infeasible deadlines are aborted. Transactions with feasible deadlines remain eligible for service. Note that this policy is predictive and uses E, the runtime estimate.

2.5 Assigning Priorities

There are many ways to assign priorities to real-time tasks [**LW**][**JLT**]. We have studied three.

2.5.1 First Come First Serve

This policy assigns the highest priority to the transaction with the earliest release time. If release times equal arrival times then we have the traditional version of FCFS. The primary weakness of FCFS is that it does not make use of deadline information. FCFS will discriminate against a newly arrived task with an urgent deadline in favor of an older task which may not have such an urgent deadline. This is not desirable for real-time systems.

2.5.2 Earliest Deadline

The transaction with the earliest deadline has the highest priority. A major weakness of this policy is that it can assign the highest priority to a task that already has missed or is about to miss its deadline. By assigning a high priority and system resources to a transaction that will miss its deadline anyway, we deny resources to transactions that still have a chance to meet their deadlines and cause them to be late as well. One way to solve this problem is to use the overload management policy Not Tardy or Feasible Deadlines to screen out transactions that have missed or are about to miss their deadlines.

2.5.3 Least Slack

For a transaction T we define a slack time $S = d - (t + E - P)$. (Recall that P is the amount of service time received by T so far.) The slack time is an estimate of how long we can delay the execution of T and still meet its deadline. If $S \geq 0$ then we expect that if T is executed without interruption then it will finish at or before its

deadline. A negative slack time is an estimate that it is impossible to make the deadline. A negative slack time results either when a transaction has already missed its deadline or when we estimate that it cannot meet its deadline.

Note that Least Slack is very different from Earliest Deadline in that the priority of a task depends on how much service time it has received. The slack of a transaction which is being executed does not change. (Its service time and the clock time increase equally.) The slack time of a transaction which is not executing *decreases*. Hence the priority of that transaction *increases*.

A natural question to consider is how often to evaluate a transaction's slack. We consider two methods. With the first, *static evaluation*, the slack of a transaction is evaluated once when the transaction arrives. This value is the transaction's priority for as long as the transaction is in the system. (If a transaction is rolled back and restarted, the slack must be recalculated. In effect, the transaction is re-entering the system as a new arrival. The ramifications of this priority adjustment are discussed in Section 2.6.3.) Under the second method, *continuous evaluation*, the slack is recalculated whenever we wish to know the transaction's priority. This method yields more up-to-date information but also incurs more overhead.

Our performance studies have shown that sometimes it is better to use static evaluation and sometimes it is better to use continuous evaluation. (See Section 3.3.1.2.) The majority of our experimental results use static evaluation. We chose this because static evaluation performed better than continuous at higher load settings, which is where we performed many of our experiments.

2.6 Concurrency Control

If transactions are executed concurrently then we need a mechanism to order the updates to the database so that the final schedule is serializable. Our mechanisms allow shared and exclusive locks. Shared locks permit multiple concurrent readers. Before presenting the algorithms we introduce some terminology and explain the conventions we use to implement two-phase locking with shared and exclusive locks.

The priority of a data object O is defined to be the maximum priority of all transactions which hold a lock on object O. If O is not locked then its priority is undefined.

Let T be a transaction requesting a shared lock on object O which is already locked in shared mode by one or more transactions. Transaction T is allowed to join the read group only if the priority of T is greater than the maximum priority of all transactions, if any, which are waiting to lock O in exclusive mode. In other words, a reader can join a read group only if it has a higher priority than all waiting writers. Otherwise the reader must wait. Conflicts arise from incompatible locking modes in the usual way. That is, an exclusive lock request conflicts with both shared and exclusive lock modes, and a shared lock request conflicts with an exclusive lock mode.

We are particularly interested in conflicts that can lead to priority inversions. A *priority inversion* occurs when a transaction T of high priority requests and blocks on a lock for object O which has a lesser priority than T. This means that T has a higher priority then the transaction(s) which holds the lock on O. Transaction T must wait until the lock holder(s) releases its lock on O, either voluntarily or involuntarily. Conflicts which cannot lead to priority inversion, i.e., the priority of the requester is less than the priority of the object, are handled by having the requester wait. Of course a deadlock detection method must be employed to detect cycles of waiting transactions.

We now discuss four techniques to resolve conflicts that may lead to a priority inversion. In the following discussion let T_R be a transaction that is requesting a lock on a data object O that is already locked by transaction T_H. Furthermore, the lock modes are incompatible and T_R has a higher priority than the priority of O. Thus the priority of T_R is greater than T_H. Namely we have a priority inversion.

2.6.1 Wait

Under the *Wait* policy, priority inverting conflicts are handled exactly as non-priority inverting conflicts. That is, the requesting transaction always blocks and waits for the data object to become free. This is the standard method for most DBMS which do not execute real-time transactions. All conflicts are handled identically and the concurrency control mechanism makes no effective use of transaction priorities. The Wait policy implements FCFS scheduling for access to data items. The algorithm is shown in Figure 2-1.

Figure 2-1 Wait Conflict Resolution Policy

IF	T_R conflicts with T_H
THEN	T_R blocks

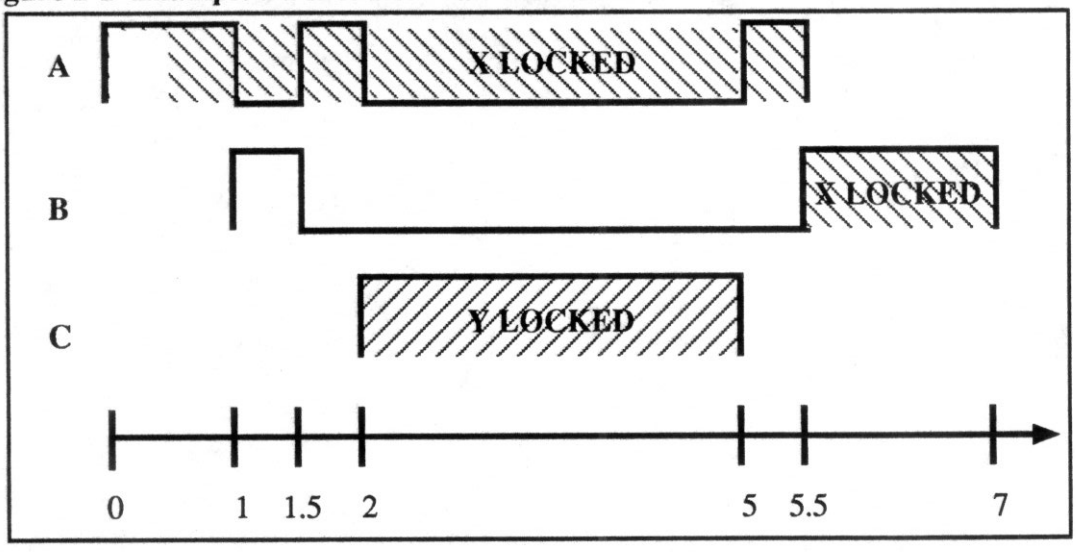
To illustrate the Wait policy, consider the set of transactions with release time r, deadline d, runtime estimate E and data requirements as shown in Table 2-1.

Table 2-1 Example 1: Transaction Table

Transaction	*r*	*E*	*d*	Updates
A	0	2	7.5	X
B	1	2	4	X
C	2	3	7	Y

Note that transactions *A* and *B* both update item *X*. Therefore these transactions must be serialized. If we use Earliest Deadline to assign priority and Wait to resolve conflicts then the schedule shown in Figure 2-2 is produced. A time line is shown at the bottom of the figure. A scheduling profile is shown for each transaction. An elevated line means that the transaction is executing on the CPU. A lowered line means the transaction is not executing. The cross hatching shows when a transaction has a lock on a data object. The cross hatching begins when the lock is granted and ends when the lock is released. Finally, the schedules assume that estimates are perfect and ignore the time required to make scheduling decisions or rollback transactions.

Figure 2-2 Example 1 Schedule: Wait with Earliest Deadline



Transaction *A* is the only job in the system at time 0, so it gains the processor and executes until time 1 when transaction *B* arrives. During this time it requests and gains an exclusive lock on data object *X*. Since *B* has an earlier deadline than *A*, *B*

preempts *A* and begins to execute. At time 1.5, *B* attempts to lock data object *X* which is already locked by *A*. Under the Wait strategy *B* must wait until *A* finishes and releases the lock on *X*. Thus *B* loses the processor and *A* resumes execution. At time 2, transaction *C* arrives and preempts *A* because *C* has an earlier deadline than *A*. Transaction *C* executes to completion and finishes at time 5. Then *A* resumes execution and completes its remaining 0.5 units of computation at time 5.5. When it commits, it releases the lock on item *X*, thus *B* is unblocked and resumes execution. It finishes its remaining 1.5 units of computation at time 7. Under this schedule, *B* misses its deadline by 3 units and *A* and *C* both meet their deadlines. The overall schedule length is 7.

Note that transactions can wait for locks; thus deadlock is a possibility. Deadlock detection can be done using one of the standard algorithms [**IM**]. Victim selection, however, should be done with consideration of the time constraints of the tasks involved in the deadlock. In our simulations, deadlocks are detected by maintaining a wait-for graph and searching for cycles whenever a new arc is added to the graph. When a deadlock is detected a victim is selected by choosing the transaction with lesser priority of the two transactions that completed the cycle in the graph. Other methods for selecting a victim are possible, e.g., select the lowest priority transaction in the entire cycle. We do not consider these methods here.

2.6.2 Wait Promote

The previous example exposed an obvious fault with the Wait policy, namely that transaction *B* had to wait for both *C* and *A* to complete before it could finish. This happened because *B* blocked on *A* and then *A* was preempted by *C*. However *B* has an earlier deadline than *C* and should be scheduled before *C*. Our second concurrency control policy, *Wait Promote*, handles this problem by increasing the priority of the lock

holder T_H to be as high as the lock requester T_R whenever a priority inverting conflict occurs. (Since locks are retained until commit time, T_H will keep its inherited priority until it commits or is restarted. In the event that T_H is restarted, e.g., because of deadlock, it assumes its normal priority. A pure implementation of priority inheritance would demote the priority of T_H if T_R were aborted before T_H finished. We chose not to implement demotion. Our tests showed that it occurs so seldom that any difference in overall performance is not measurable.) This method for handling priority inversions was proposed in [**SRL1**].

The reason for promoting T_H is that it is blocking the execution of T_R, a higher priority transaction. Thus T_H should execute at an elevated priority in order to get it done and removed so that T_R can execute. Priority inheritance ensures that only a transaction with priority greater than T_R will be able to preempt T_H from the CPU. A transaction T_I of intermediate priority, a priority greater than T_H and less than T_R, would normally be able to preempt T_H. But with priority inheritance, T_I has a lesser priority than T_H which is now executing on behalf of T_R.

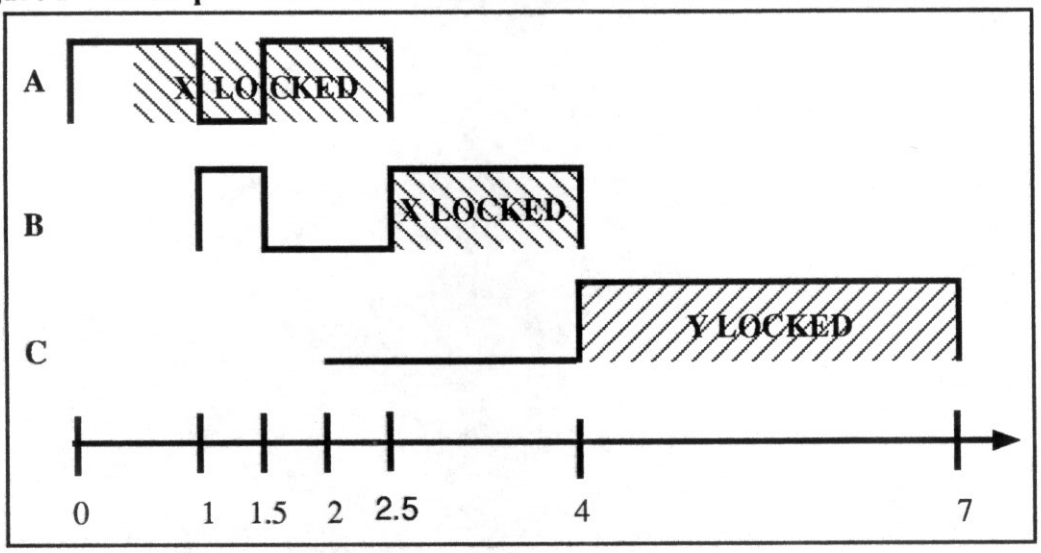
Figure 2-3 shows the Wait Promote algorithm.

Figure 2-3 Wait Promote Conflict Resolution Policy

IF	$P(T_R) > P(T_H)$
THEN	T_R blocks
	T_H inherits the priority of T_R
ELSE	T_R blocks

Figure 2-4 shows the schedule that is produced when Wait Promote is used to schedule the transactions of Table 2-1.

Figure 2-4 Example 1 Schedule: Wait Promote with Earliest Deadline



As before, transaction *A* gains a lock on object *X* and computes and then is preempted by transaction *B* at time 1. A conflict arises when *B* requests a lock on *X* at time 1.5. Transaction *B* waits for the lock to be released and *A* inherits the deadline of *B*. Thus when *C* enters the system it does not preempt *A* because *A* has the same deadline as *B*, namely 4, and *C* has a deadline of 7. Transaction *A* commits at time 2.5 and releases its locks. Transaction *B* is unblocked and resumes execution to finish at time 4. Then *C* can execute and finish at time 7. In this schedule all transactions meet their deadlines. The overall schedule length is also 7.

What if the object is locked by more than one transaction? In this event all transactions in the read group will inherit the priority of T_R. Note that a priority inversion can affect only some of the transactions in a read group. For example, the requesting transaction may have a priority that is greater than only some of the transactions in the read group. These transactions will inherit the greater priority of the requester. The priority of the other transactions in the read group remains unchanged. Thus every transac-

tion holding a lock on object O has a priority that is at least as high as the highest priority transaction which is waiting for the lock.

Finally, the property of priority inheritance is transitive. If, for example, T_H is blocked by transaction T_{HH}, and the priority of T_{HH} is less than T_R then T_{HH} will inherit the priority of T_R.

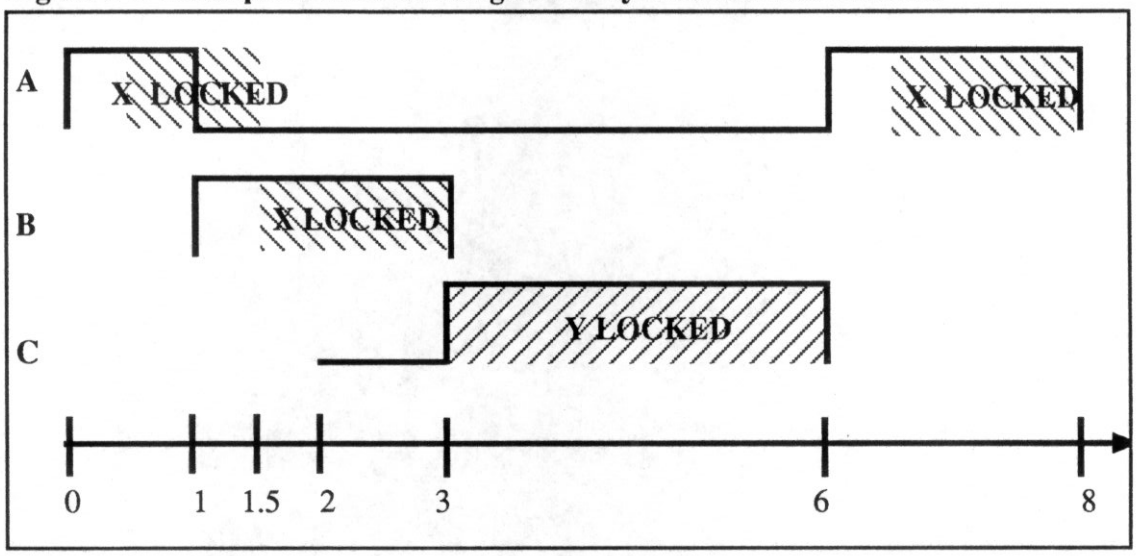
Note that when priority inheritance is combined with Least Slack, continuously evaluated, T_H inherits not a static priority but a priority function which evaluates the slack of T_R.

2.6.3 High Priority

Under the first two policies, the requesting transaction always waits for the lock holding transaction to finish and release its locks. This is true even when the requesting transaction has a very high priority. An alternative approach, one taken by the *High Priority* policy, is to resolve a conflict in favor of the transaction with the higher priority. The favored transaction, the winner of the conflict, is allowed to lock the contested object. We implement this policy by comparing transaction priorities at the time of the conflict. If the priority of T_R is greater than the priority of object O, and thus greater than every transaction holding a lock on O, then we abort the lock holders thereby freeing the object for T_R. T_R can resume processing; the lock holders are rolled back and scheduled for restart. If the priority of T_R is less than or equal to the priority of O then T_R blocks to wait for O to become free.

Figure 2-5 shows the schedule produced when High Priority is used to schedule the transactions in Table 2-1.

Figure 2-5 Example 1 Schedule: High Priority with Earliest Deadline



As before, *A* runs in the first time unit during which it acquires a lock on item *X*. Transaction *B* gains the processor at time 1 and causes a conflict at time 1.5 when it requests a lock on item *X*. Since *B* has an earlier deadline than *A* and thus a higher priority, the conflict is resolved by rolling back *A* thereby freeing the lock on *X*. Transaction *B* continues processing and completes at time 3. Transaction *C*, with an earlier deadline than *A*, gains the processor and completes at time 6. Finally, *A* regains the processor and starting from the beginning, executes for 2 units and finishes at time 8. In this schedule, *A* misses its deadline by 0.5 units and *B* and *C* meet their deadlines. The overall schedule length is 8 because a portion of transaction *A* is executed twice.

An interesting problem arises when we use Least Slack to prioritize transactions. Recall that under this policy, a transaction's priority depends on the amount of service time that it has received. Rolling back a transaction to its beginning reduces its effective service time to 0 and raises its priority under the Least Slack policy. Thus a transaction

T_H, which loses a conflict and is aborted to allow a higher priority transaction T_R to proceed, can have a higher priority than T_R immediately after the abort. The next time the scheduler is invoked, T_R will be preempted by T_H. T_H may again conflict with T_R initiating another abort and rollback.

Our solution to this problem is to compare the priority of T_R against that of each lock holder assuming that the lock holder were aborted. Using the notation $P(T_H)$ to denote the priority of T_H and $P(T_H^A)$ to denote the priority of T_H were it to be aborted, we can write this algorithm as follows:

Figure 2-6 High Priority Conflict Resolution Policy

IF	For all T_H holding a lock on O
	$P(T_R) > P(T_H)$ **AND** $P(T_R) > P(T_H^A)$
THEN	Abort each lock holder
ELSE	T_R blocks

For FCFS and Earliest Deadline policies, $P(T_H) = P(T_H^A)$, so it does not matter if we use the original High Priority resolution rule or the modified one above. Since the modified rule is clearly superior for Least Slack priority assignment, we will use it for our performance evaluations.

2.6.4 Conditional Restart

Sometimes High Priority may be too conservative. Let us assume that we have chosen the first branch of the algorithm, i.e., T_R has a greater priority than T_H and T_H^A. We would like to avoid aborting T_H because we lose all the service time that it has already consumed. We can be a little cleverer by using the *Conditional Restart* policy to resolve conflicts. The idea here is to estimate if T_H, the transaction holding the lock, can be finished within the amount of time that T_R, the lock requester, can afford to wait. Let

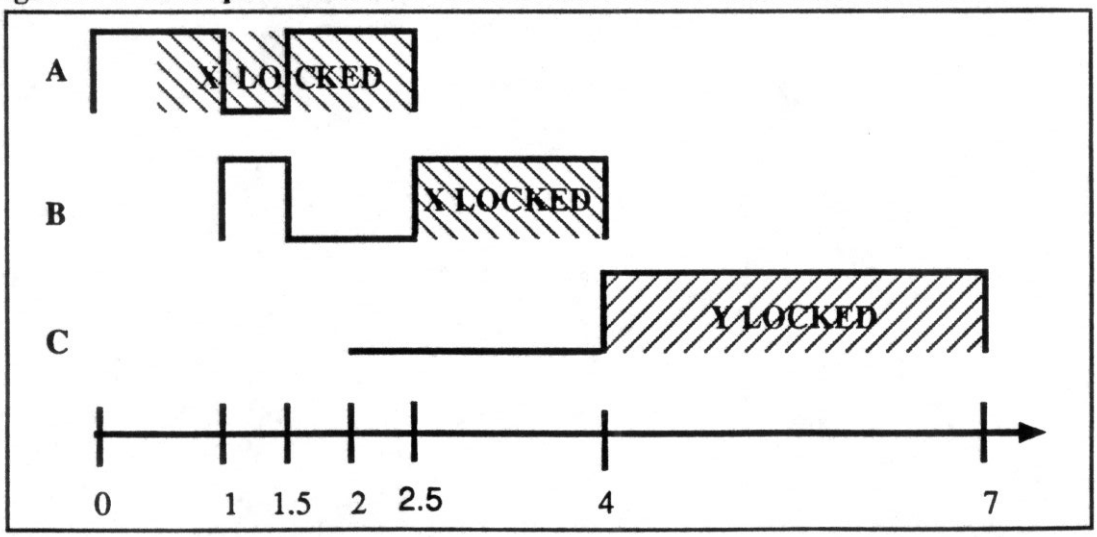
S_R be the slack of T_R and let E_H - P_H be the estimated remaining time of T_H. If $S_R \geq E_H$ - P_H then we estimate that T_H can finish within the slack of T_R. If so, we let T_H proceed to completion, release its locks and then let T_R execute. This saves us from re-starting T_H. If T_H cannot be finished in the slack time of T_R then we restart T_H (as in the previous algorithm). This modification yields the following algorithm:

Figure 2-7 Conditional Restart Conflict Resolution Policy

IF	$P(T_R) > P(T_H)$ **AND** $P(T_R) < P(T_H^A)$	
THEN	**IF**	$S_R \geq E_H$ - P_H
	THEN	T_R blocks
		T_H inherits the priority of T_R
	ELSE	Abort T_H
ELSE	T_R blocks	

Note that if T_R blocks in the inner branch, then T_H inherits the priority of T_R. This inheritance is exactly the same as described in the Wait Promote algorithm. Figure 2-8 shows the schedule that is produced when Conditional Restart is used to schedule the transactions of Table 2-1.

Figure 2-8 Example 1 Schedule: Conditional Restart with Earliest Deadline



As before, a conflict occurs when *B* requests a lock on *X* at time 1.5. At this time the algorithm calculates the slack time for *B* as $S = 4 - 1.5 - 1.5 = 1$. This equals exactly the remaining run time for *A*. Therefore *B* waits and *A* inherits the priority of *B*. *A* regains the processor and executes without preemption until it finishes at time 2.5. (Transaction *C* does not preempt *A* because *A* has inherited the deadline of *B*, namely 4.) Transaction *B* is unblocked and resumes execution to finish at time 4. Then *C* executes to finish at time 7. All transactions meet their deadlines.

Note that this schedule is exactly the same as that produced by Wait Promote. In fact it is easy to see that Conditional Restart behaves like Wait Promote when the first branch of the inner condition is taken, and like High Priority when the second branch is taken.

We only implement Conditional Restart if the conflict is one-on-one, i.e., there is no read group involved. Furthermore we do not consider chained blockings. That is, we only make the special Conditional Restart decision if the requester conflicts with exactly one lock holder and the lock holder is not blocked waiting for some other lock. Experience with our simulations has indicated that chained blockings are rare, so that the payoff for handling them in a clever way is limited.

Finally, we caution that the examples we have used to illustrate the different algorithms are greatly simplified. They are presented to motivate the algorithms, not to prove that one algorithm is better than another. For instance, in reality transactions may update several items or none at all (i.e., read-only), and this will obviously affect the performance of the algorithms. In Chapter <evaluation by simulation> we discuss a detailed simulation model that can help us to compare the various scheduling and concurrency control options.

2.7 I/O Scheduling

In a non memory resident database system, the disk is an important resource which can be managed to optimize various performance criteria. In conventional systems the usual goal is to maximize the throughput of the I/O system. One way that this is accomplished is by using a disk scheduling algorithm (e.g., SCAN [**PS**]) to order the sequence of I/O requests so that the mean seek time is minimized. While this may be good for maximizing throughput, it may be bad for a real-time system which is trying to meet transaction deadlines. For example, SCAN may order a batch of requests so that an I/O request from a transaction with an early deadline is serviced last.

This section looks at two simple ways to schedule I/O requests. One method uses transaction timing constraints for ordering requests while the other does not. The simulation model for evaluating these two methods makes the simplistic assumption that disk access times are constant (Section 3.2). Based on the knowledge gained when the simple I/O model was simulated (Section 3.4), a more detailed I/O model and several more algorithms are developed in Chapter 4.

2.7.1 FIFO

When FIFO is used to schedule the I/O queue, requests are serviced in the order in which they are generated. This service order is somewhat related to transaction priorities because I/O requests are generated by the CPU, which is scheduled by priority. The ordering is essentially random with respect to cylinder position on the disk.

2.7.2 Priority

Under this policy each I/O request has a priority which is equal to the priority of the transaction which issued the request. The next I/O request to service is the one with the highest priority. Thus a newly arrived request from a transaction with a high priority

can leapfrog over other requests which have been waiting longer in the I/O queue. We also expect this ordering to be random with respect to cylinder positions on the disk.

In our model there are two types of I/O requests: reads, that are issued by unfinished transactions, and writes that are generated by committed transactions that are flushing their updates back to disk. (The log resides on a separate device, so it receives only log writes which are serviced FCFS. Log writes are sequential and ordered by cylinder position.) Giving higher priority to reads over writes is desirable because it will speed the completion of transactions which are trying to meet their deadlines. Giving high priority to writes does not enhance performance directly because the transactions which issued the writes have already committed. In fact, as our studies have shown, giving high priority to writes can decrease performance if it excessively delays the servicing of read requests. The priority of writes cannot be too low, however, as writes must be completed in order to free space in the memory buffer pool.

In our first set of experiments, writes have the same priority as the transaction that issued them. (The second set of experiments, performed under a different model, are presented in Chapter 5.) For the priorities FCFS and Earliest Deadline, this means that writes are given a relatively high priority. (The arrival times and deadlines of committed transactions are usually earlier than those of uncommitted transactions.) If we use static evaluation to implement Least Slack, the slack times of committed transactions are not necessarily larger or smaller than those of uncommitted transactions. However if we use continuous evaluation, then the slack times of committed transactions will usually be smaller than the slack times of uncommitted transactions.

Chapter 3 Evaluation by Simulation

3.1 Introduction

This chapter describes the simulation model and the metrics that we use to evaluate the scheduling algorithms, Section 3.2. We present the experimental results for a memory resident database, Section 3.3, and for a disk resident database, Section 3.4. Finally we present our conclusions and discuss how our findings compare with results of other published research, Section 3.5

3.2 Simulation Model

Our program to simulate a RTDB system was built using SIMPAS, an event-oriented discrete system simulation language [**Br**]. The names and meanings of the four parameters that control the configuration of the system resources are given in Table 3-1. The database log is maintained on a separate device which is of equal speed as the database disks. Each disk has its own queue of service requests.

Table 3-1 System Resource Parameters

Parameter	Meaning
DBsize	Number of pages in database
MemSize	Number of pages in memory buffer pool
NumDisks	Number of disks
IOtime	Time to perform a disk access(read or write)

The database buffer pool is modeled as a set of pages each of which can contain a single database object. We do not model each buffer page individually, that is, we do not maintain a free list of pages, nor do we keep track of which pages have been modified. Instead we model the buffer pool as a collective set. When a transaction attempts to read an object, the system generates a random boolean variable which has the value true

with probability $\frac{MemSize}{DBsize}$. If the value is true then the page is in memory and the transaction can continue processing. If the value is false then an I/O service request is created and placed in the input queue of the appropriate disk. The database is partitioned equally over the disks and we use the function $D = \left\lceil \frac{i \times NumDisks}{DBsize} \right\rceil$ to map an object i to the disk where it is stored.

Transaction characteristics are controlled by the parameters listed in Table 3-2. Transactions enter the system with exponentially distributed inter-arrival times and they are ready to execute when they enter the system (i.e., release time equals arrival time). The number of objects accessed by a transaction is chosen from a normal distribution with mean *Pages* and the actual database items are chosen uniformly from the database. Each page is updated with probability *Update*. Pages which are updated are locked exclusively, other pages are locked in shared mode. Updated pages are stored in the buffer pool until a transaction commits and then they are flushed out to disk.

Table 3-2 Transaction Parameters

Parameter	Meaning
ArrRate	Mean arrival rate of transactions
Pages	Mean number of pages accessed per transaction
CompFactor	CPU computation per page accessed
Update	Probability that a page is updated
MinSlack	Minimum slack
MaxSlack	Maximum slack
EstErr	Error in runtime estimate as a fraction of total runtime
Rebort	Time needed to rollback and abort a transaction

A transaction has an execution profile which alternates lock requests with equal size chunks of computation, one for each page accessed. Thus the total computation

time is directly related to the number of items accessed. Let C denote the CPU requirement for a transaction; then $C = Pages' \times CompFactor$. (We use *Pages'* to denote the actual number of pages for a specific transaction rather than the mean.) The I/O service requirement for a transaction has three components: the first is the time needed to read pages from the disk into memory; the second is the time needed to write a log record; and the third is the time needed to write the modified pages back to the disk. Since the writing of modified pages back to disk occurs after a transaction commits, this I/O time is not included in the runtime estimate. Assuming that logging can be done with one disk access, the expected amount of pre-commit I/O service time needed is $I = IOtime \times Pages' \times \left[1 - \frac{MemSize}{DBsize}\right] + IOtime$. Thus the total expected runtime service needed by a transaction executing in an unloaded system is $R = C + I$. The accuracy of a transaction's runtime estimate E with respect to R is controlled by the parameter *EstErr*, where $E = R \times (1 + EstErr)$. How we choose the value of *EstErr* is explained later when we discuss the experimental results.

The assignment of a deadline is controlled by two parameters *MinSlack* and *MaxSlack* which set a lower and upper bound respectively on a transaction's slack time. A deadline is assigned by choosing a slack time uniformly from the range specified by the bounds. *Rebort* controls the amount of CPU time needed to abort or restart a transaction. Aborting a transaction consists of rolling it back and removing it from the system. The transaction is not executed. When a transaction is restarted it is rolled back and placed again in the ready queue. (Recall that updates are flushed after transaction commit, so an abort does not generate an I/O service request.) Note that aborts are generated by the overload management policy, restarts result from lock conflicts.

The simulator does not explicitly account for time needed to execute the lock manager, conflict manager, and deadlock detection manager. These routines are executed on a per data object basis and we assume that the costs of these calls are included in the variable that states how much CPU time is needed per object that a transaction accesses. Context switching and the time to execute the scheduler is also ignored.

As mentioned earlier, deadlocks are detected by maintaining a wait-for graph and searching for cycles whenever a new arc is added to the graph. When a deadlock is detected a victim is selected by choosing the transaction with lesser priority of the two transactions that completed the cycle in the graph. The victim is rolled back and placed in a special queue until the transaction with which it deadlocked exits the system, either because it commits or because it is aborted. When this happens the victim is placed in the ready queue and allowed again to enter the system. This system of enforced waiting is necessary to prevent the excessive formation of deadlocks.

We use several metrics to evaluate the algorithms. In particular we measured the percentage of transactions which missed their deadlines, the average amount of time by which transactions missed their deadlines, and the total number of restarts caused by lock conflicts or deadlocks. We also study how the algorithms perform when the system experiences a sudden load increase. How we simulate this input step function will be explained in Section 3.4.3.

The percentage of missed deadlines is calculated with the following equation: $\%Missed = \frac{Tardy\ jobs + Aborts}{Jobs\ processed} \times 100$. A job is processed if either it executes completely or it is aborted. The mean tardy time is simply the average of the tardy times of

all committed transactions. A transaction that commits before or on its deadline has a tardy time of zero. Aborted transactions do not contribute to this metric.

In this study we have included tardy jobs and aborted jobs together in the *%Missed* metric. For some applications it may be useful to describe a separate metric for aborted jobs as they represent tasks which were never completed and as such may be more serious than simply tardy jobs. For reasons of space we do not study these other metrics here. Unlike conventional performance evaluations of concurrency control mechanisms, this study does not focus on transaction response times. The reason is that response time is not critical as long as a transaction meets its deadline.

All of our experiments were conducted in one of two cases: the main memory case and the disk resident case. Section 3.3 presents the results for the main memory case and Section 3.4 presents the results for the disk resident case. Section 3.5 summarizes our conclusions.

Section 2.3 proposed three different methods for managing overloads and four methods each for assigning priority and managing concurrency. Also I/O scheduling can be done in two different ways. Taking the cross product yields 96 different algorithms. Table 3-3 summarizes the methods of Section 2.3 and provides the abbreviations that we will use when referring to them. We use the format NT/LS/WP to denote the algorithm formed by combining the polices Not Tardy, Least Slack and Wait Promote. Other algorithms are denoted similarly.

Table 3-3 Summary of Scheduling Policies

Component	Policy
Overloads	AE - All Eligible NT - Not Tardy FD - Feasible Deadlines
Priority	FCFS - First Come First Serve ED - Earliest Deadline LS - Least Slack (Static evaluation) LSC - Least Slack (Continuous evaluation)
Concurrency	W - Wait WP - Wait Promote HP - High Priority CR - Conditional Restart
I/O Scheduling	FIFO Priority

In Sections 3.3 and 3.4 we discuss some of the results of the many different experiments that we performed. We have selected the graphs which best illustrate the differences and performance of the algorithms. For each experiment we ran the simulation with the same parameters for 20 different random number seeds. Each run, except for the input step function experiment, continued until at least 700 transactions were executed. For each algorithm tested, numerous performance statistics were collected and averaged over the 20 runs. It is these averages and 90% confidence intervals (shown as vertical bars) which are plotted in the graphs.

3.3 Experimental Results: Memory Resident Database

We begin our performance analysis by studying the algorithms in experiments where the database is memory resident. This assumption simplifies our model somewhat and makes it easier to understand the scheduling options and their impact on performance. In Section 3.4 we allow the database to be disk resident and we study how I/O

scheduling impacts performance. In any case, studying a memory resident system is important since many existing real-time systems currently hold all their data in memory. Furthermore, since memory prices are steadily dropping, memory sizes are growing and memory residence becomes less of a restriction.

The base parameters for the memory resident case are shown in Table 3-4.

Table 3-4 Base Parameters: Memory Resident Database

Parameter	Value	Units	Parameter	Value	Units
DBsize	400	Pages	*MemSize*	400	Pages
ArrRate	7	Jobs/sec.	*Pages*	12	Pages
CompFactor	10	ms.	*Update*	1	
MinSlack	0.1	sec.	*MaxSlack*	1	sec.
EstErr	0		*Rebort*	5	ms

These values are not meant to model a specific real-time application but were chosen as reasonable values within a wide range of possible values. We chose the arrival rate so that the corresponding CPU utilization (an average 0.84 seconds of computation arrive per second) is high enough to test the algorithms. It is more interesting to test the algorithms in a heavily loaded rather than lightly loaded system. (We return to this issue in the Section 3.5.) Also note that the probability that a page that is accessed is updated is 1. This simplifying assumption means that all locks are exclusive locks and all lock conflicts are between exactly two transactions. We will first study the algorithms under this simplifying assumption and then relax it to allow read locks.

The I/O system processes two types of requests: the first type consists of log records that are written sequentially to the separate log device. This activity induces a relatively low load on the I/O system. The second type of request are writes of modified pages back to the disk resident copy of the database. Updates are flushed to maintain an up-to-date copy of the database on disk. Since writes occur after a transaction commits,

they have no affect on transaction tardiness. Also since no transaction reads from the disk, the writes cannot interfere with other transactions.

3.3.1 Effect of Increasing Load

In this experiment we varied the arrival rate from 6 jobs/sec to 8 jobs/sec in increments of 0.5. The other parameters had the base values given in Table 3-4. Under these settings the CPU utilization ranges from 0.72 to 0.96 seconds of computation arriving per second.

3.3.1.1 Overload Management

Our simulation experiments show that the overload management policies NT and FD substantially reduce the number of deadlines missed compared with the AE policy. Aborting a few late transactions helps all other jobs meet their deadlines. This is illustrated in Figure 3-1 which shows the three overload management policies for FCFS scheduling. The three algorithms perform comparably when the arrival rate is lowest. As the load increases, NT and FD perform significantly better than AE, and at the highest load setting, NT and FD yield an approximate 50 percent decrease in missed deadlines over AE. This same behavior holds true for the other priority assignment policies as well.

Unless noted otherwise, the remaining graphs show algorithms which schedule all transactions, clearly the more difficult case. If the policy is AE the code will be omitted in the legend for the graph.

3.3.1.2 Priority Assignment

To eliminate concurrency control as a factor in the performance of the algorithms we performed an experiment with concurrency control turned off i.e., all lock requests were granted immediately and the resulting schedule was non-serializable. (A

similar effect can be achieved by setting all transactions to be read-only, thus there will be no lock conflicts. However, this will slightly alter the runtime and deadline characteristics of transactions since no logging would be necessary.) Figure 3-2 shows the performance for each of the priority algorithms. There are four graphs because LS appears once with static evaluation and once with continuous evaluation. As expected, all the algorithms miss a greater number of deadlines as the load increases. Algorithm FCFS misses the most deadlines for all load settings. This is not surprising since it does not use transaction time constraints when assigning priority. At lower load settings ED performs best. As the load increases, the performance margin of ED and LSC over FCFS narrows. As mentioned earlier (Section 2.5.2), ED performs poorly at higher load settings because it assigns high priorities to transactions which have missed or are about to miss their deadlines. This causes other transactions which could meet their deadlines to be tardy. The same is true for LSC and its performance curve follows that of ED very closely. We will see repeatedly that ED (or LSC) is usually a good performing algorithm when the load is low but that it loses its performance margin over other algorithms when the load increases.

At higher load settings LS is clearly the superior policy. Because the slack time is evaluated only once when the transaction enters the system, this algorithm avoids the weakness that is common to both ED and LSC. Since LS (static evaluation) is dramatically better than LSC we will use it as the preferred version of LS for the remainder of the experiments unless noted otherwise.

Ideally we want our algorithms to schedule all transactions such that all deadlines are met. However, if this is not possible, then we would want to minimize the amount by which tardy transactions miss their deadlines. Figure 3-3 graphs the mean

tardy time in seconds against arrival rate. (Concurrency control is still turned off.) It is interesting to note that ED has the least mean tardy time, then LSC, then FCFS and finally LS. These results are not surprising for it is known that ED minimizes the maximum task tardiness and LS maximizes the minimum task tardiness [**CD**].

3.3.1.3 Concurrency Control

We now examine the performance of the four concurrency control mechanisms when they are paired with the three priority policies. (We do not consider LSC.) Since we are in the main memory database case, using FCFS to schedule transactions results in a serial execution of transactions. The currently executing transaction can never be preempted by an arriving transaction. Thus there is no difference in performance when FCFS is paired with the different concurrency control mechanisms. Figure 3-4 graphs the Wait and Wait Promote concurrency control strategies for each of the three priority policies. For reasons of clarity, only one curve is shown for FCFS. At lower load settings ED/W and ED/WP perform better than both FCFS and LS. As the load increases, the performance margin of ED over FCFS narrows. Again we see the the problem with ED at higher load settings. Although LS/WP is not as good as either ED/W or ED/WP at the lowest settings, it is clearly the superior policy at higher loads.

Figure 3-5 shows the results for the High Priority and Conditional Restart algorithms for each of the three priority policies. (Again we show only one graph for FCFS.) The results are similar to Figure 3-4 except that ED/HP and ED/CR lose their performance margin over FCFS even sooner. This occurs because both HP and CR will abort transactions when conflicts occur. When the load is high and conflicts are frequent, these aborts effectively increase the transaction arrival rate. Under the increased load, the performance of ED degrades as explained earlier.

In Figure 3-6 we plot only ED and LS with each concurrency control policy. No algorithm is best at all load settings. Algorithms ED/WP and ED/CR are best at the lowest load settings while LS/WP and LS/CR perform better at higher loads. The ED algorithms are bunched closely with ED/WP and ED/CR performing slightly better than ED/W and ED/HP. Finally the worst combinations are LS/W and LS/HP.

An obvious question raised by Figure 3-4 is why LS/WP is so much better than LS/W, particularly under high loads. By contrast the performance gap between ED/W and ED/WP is small. The reason is that LS scheduling permits a greater degree of concurrency (average number of active jobs in the system at any time) than does ED scheduling. (Remember that the database is memory resident. Preemptions occur only when a higher priority transaction joins the ready queue. The current job never gives up the processor to wait for I/O.)

To confirm this, Figure 3-7 graphs the average number of active jobs for the LS and ED algorithms in the arrival rate experiment. We see that both LS algorithms produce higher levels of concurrency than the FCFS and ED algorithms. We also see that the average number of active jobs for LS/WP is less than that of LS/W for nearly all load settings.

To explain the results of Figure 3-7 we note that with LS priority assignment there is no correlation between a transaction's arrival time and its priority, or slack. By contrast, with Earliest Deadline there is a direct correlation between arrival time and priority. Namely, a transaction T_B which arrives much later then transaction T_A is more likely to have a deadline which is later than T_A. Thus it is less likely that transaction T_B will preempt transaction T_A.

For example, transaction T_A arrives at time 4, has a slack time of 3, and a deadline of 10. Transaction T_B arrives at time 8, has a slack of 2 and a deadline of 16. If we use LS to determine priority then T_B will preempt T_A when it arrives. However, if we use ED to determine priority then T_B will not preempt T_A.

One consequence of this higher level of concurrency is that there are more lock conflicts. Furthermore, there are more conflicts where a high priority transaction blocks on a lock held by a lower priority transaction i.e., a priority inversion. When a priority inversion occurs, algorithm LS/WP takes the right action by promoting the priority of the lock holder to be as high as the priority of the lock requester. This shortens the waiting time for the high priority transaction and increases the chances that it will meet its deadline. The Wait strategy does not do this. This demonstrates the importance of handling priority inverting conflicts correctly.

This same characteristic of LS, namely a higher average level of concurrency, also makes it more likely that a deadlocks will occur. Although we do not include the graphs, our experiments show that LS algorithms do produce more deadlocks than ED algorithms. Furthermore LS/W suffers significantly more deadlocks than LS/WP.

3.3.2 Biasing the Runtime Estimate

The runtime estimate E is used by three of the scheduling policies that we presented in Section 2.3. Of the three overload management policies, Feasible Deadlines is the only one which makes use of the runtime estimate E. The priority policy Least Slack also uses E as does the concurrency control policy Conditional Restart. For the policies FD and CR it is relatively easy to predict how they will respond to error in the runtime estimate (see below). The case for LS, however, is not as simple.

To study how error in the runtime estimate E affects the different components of scheduling algorithms we devised three different experiments, each of which biased the runtime estimate in a different way. The first experiment was designed to introduce a random amount of error (within a certain range) into the estimate E. The second experiment was designed to bias all the runtime estimates in the same direction and by proportionally equal amounts. In both experiments the priority policy LS performed nearly equally well under both high error and low error parameter settings. This finding motivated a final third experiment where half the transactions were made to overestimate their runtime estimate and the other half underestimated. This technique for biasing E did yield changes in performance for the LS policy. Thus it is this technique that is used in the experimental results reported below.

For half of the transactions $E = R \times (1 + EstErr)$; for the other half $E = R \times (1 - EstErr)$. The value of *EstErr* was varied from 0 to 4 in increments of 1. Thus when $EstErr = 0$, $E = R$ and there is no deliberate bias in the runtime estimate. When $EstErr = 1$, half the transactions have $E = 0$ and half have $E = 2 \times R$. (For values of *EstErr* greater than 1, negative runtime estimates are converted to 0.)

3.3.2.1 Overload Management

We would expect the accuracy of the runtime estimate to have a large effect on the policy FD. The overload management policy is responsible for aborting transactions and since aborted transactions are counted as having missed their deadlines, the policy directly affects the performance. It is easy to see how this policy is affected by error in the runtime estimate. When the runtime estimate for jobs is zero, FD behaves like NT,

aborting jobs only if they have missed their deadlines. When the estimate is high, FD thinks that jobs are much longer than they are and will judge incorrectly, that they have infeasible deadlines. Thus jobs with feasible deadlines are unnecessarily aborted. The predicted behavior is confirmed in Figure 3-8. Note that algorithms using AE and NT are not affected by changes in *EstErr*.

3.3.2.2 Priority Assignment

Figure 3-9 graphs the results for LS (static evaluation) when used with each of the concurrency control policies. The difference in performance between when the error is low (*EstErr* = 0) and when the error is high (*EstErr* = 4) is only a few percentage points.

When continuous evaluation is used (graph is not shown), the performance of LS, is more sensitive to error in the runtime estimate. This is understandable since continuous evaluation means that the inaccurate runtime estimate will be used many more times to make scheduling decisions.

3.3.2.3 Concurrency Control

Only the concurrency control strategy Conditional Restart uses the runtime estimate. Algorithm CR uses the runtime estimate to decide if a low priority transaction holding a lock can finish within the slack time of the higher priority transaction requesting the lock. We can easily describe the behavior of CR for the extreme values of E. When $E = 0$, CR will always judge (assuming that the lock requester has a positive slack, this is true if the transaction has not missed its deadline) that the lock holder can finish within the slack of the lock requester. Thus the lock requester will always wait for the lock holder. Since promotion is used by CR, this behavior is exactly the same as WP.

At the other extreme, when E is much larger than R, the algorithm will nearly always judge that the lock holder cannot finish within the slack time of the lock requester. Thus the lock holder will be rolled back and restarted. This behavior is the same as the concurrency control strategy HP. When the value of E is not so extreme CR will behave somewhere between WP and HP.

3.3.3 Cost of Serializability

We can use the results of the experiment where concurrency control was turned off to understand how the enforcement of serializable schedules affects performance in terms of missed deadlines. Figure 3-10 shows the performance of the serialized and unserialized versions for one of the better versions of ED and LS, namely ED/WP and LS/WP. The unserialized versions perform better than the serialized version for each algorithm. Thus serializability does cause the algorithms to miss more deadlines. However, missed deadlines is only one cost metric. Database inconsistency occurs as a result of unserialized schedules. For some applications the cost of database inconsistency may far outweigh the performance benefit in terms of missed deadlines gained by ignoring concurrency control.

3.3.4 Increasing Conflicts

In this experiment we varied the value of *DBsize* from 200 to 400 in increments of 50. The parameter *MemSize* was varied in the same way so that we remained in the main memory case. The other parameters had the values shown in Table 3-4. In this kind of experiment, the overall load and transaction characteristics remain constant. However, since transactions access the same number of objects, the probability of conflict is higher when the size of the database is small. The probability of conflict de-

creases as the number of objects in the database increases. Thus we can compare how the various concurrency control strategies perform as the number of conflicts changes.

Figure 3-11 shows the results for all four concurrency control strategies each paired with ED and LS. It confirms our expectation that, as *DBsize* increases, all scheduling algorithms will perform better. One noteworthy observation is that the curves for the ED algorithms are remarkably flat, i.e., there is only a small difference between the small database and the large database performance values. Recall from Figure 3-7 that ED scheduling in a main memory database results in a very low level of concurrency. If the average number of active jobs is very low (less than two) then it doesn't matter how small the database is because there are not enough active jobs requesting locks to conflict.

However, LS scheduling results in higher levels of concurrency (Figure 3-7) and algorithms using LS are more sensitive to changes in database size (Figure 3-11). In fact LS/W and LS/HP perform very poorly when the database is small. By contrast, LS/WP and LS/CR, which use priority inheritance to manage priority inversions, maintain good performance even when the database is small. Again this demonstrates the importance of managing priority inversions correctly.

3.3.5 Increasing Slack

The average amount of slack in job deadlines affects all scheduling algorithms. If all deadlines are extremely tight then most algorithms will perform poorly. Similarly, if all deadlines are very loose than most algorithms will perform well. In this experiment the parameter *MaxSlack* was varied from 0.2 sec. to 1.8 sec in steps of 0.4 sec. Recall that this parameter governs the maximum possible slack time for a transaction. The

minimum slack time was 0.1 sec and the rest of the parameters had the values in Table 3-4.

3.3.5.1 Priority Assignment

Figure 3-12 shows the performance of the three priority assignment polices when paired with the concurrency control policy WP. When the maximum slack is small (and deadlines are tight), LS is clearly the better policy then ED which is slightly better than FCFS. As the slack increases, the performance of all three policies improves. However, the relative performance of ED and LS changes so that ED is the better algorithm when the slack is large. At this range both algorithms are clearly superior to FCFS. If the slack were very much greater (not shown) then the three curves would converge to zero because no deadlines would be missed.

3.3.5.2 Concurrency Control

The relative performance of the concurrency control policies is not affected as strongly as the priority policies. Figure 3-13 shows the four concurrency control strategies paired with LS. Strategies WP and CR have nearly identical performance so their graphs appear as one. Both WP and CR perform significantly better than W and HP which have similar performance. We can also see that the performance margin between W and WP (HP and CR) increases slightly as the slack increases. In other words, WP and CR perform even better as the slack increases. The results for the policies when paired with ED are similar.

3.3.6 Increasing Cost of Restart

In this experiment the cost of restarting or aborting a transaction ranges from 0 ms (i.e. no cost) to 50 ms in increments of 10. For reference, when the cost is 40 it is equal to one-third the average transaction computation time. This is a very high cost,

given that we have a memory resident database. Our objective is not to simulate a particular application but rather to see how the algorithms perform under even extreme parameter settings.

The cost of restarting or aborting a transaction affects the eligibility polices NT and FD. All the concurrency control polices are affected because the restart cost controls the amount of time needed to recover from a deadlock. The policies HP and CR also perform restarts when lock conflicts are encountered.

It follows that as the cost of restarting or aborting increases, the performance of algorithms which use any of the above policies will deteriorate. This is indeed the case as shown by Figure 3-14 which plots the results for the three different priority policies paired with HP. When the restart/abort cost is zero, ED is best followed by FCFS and and LS. As the time required to restart a transaction increases the performance of ED and LS degrades. The performance of FCFS does not change since it effectively executes transactions serially. (The entire database is main memory, thus there is no chance for concurrency caused by waiting for I/O completions.) Thus there are no conflicts and no restarts. When the time to restart a transaction is very high, 50 ms, we see that ED and FCFS perform the same. However at this setting, the time to restart a transaction is nearly half the average running time of a transaction.

The priority policy LS is most sensitive to increases in the time needed to restart a transaction. It performs the most restarts and thus suffers most when the cost is increased.

Figure 3-1 Main Memory; Overload Management.

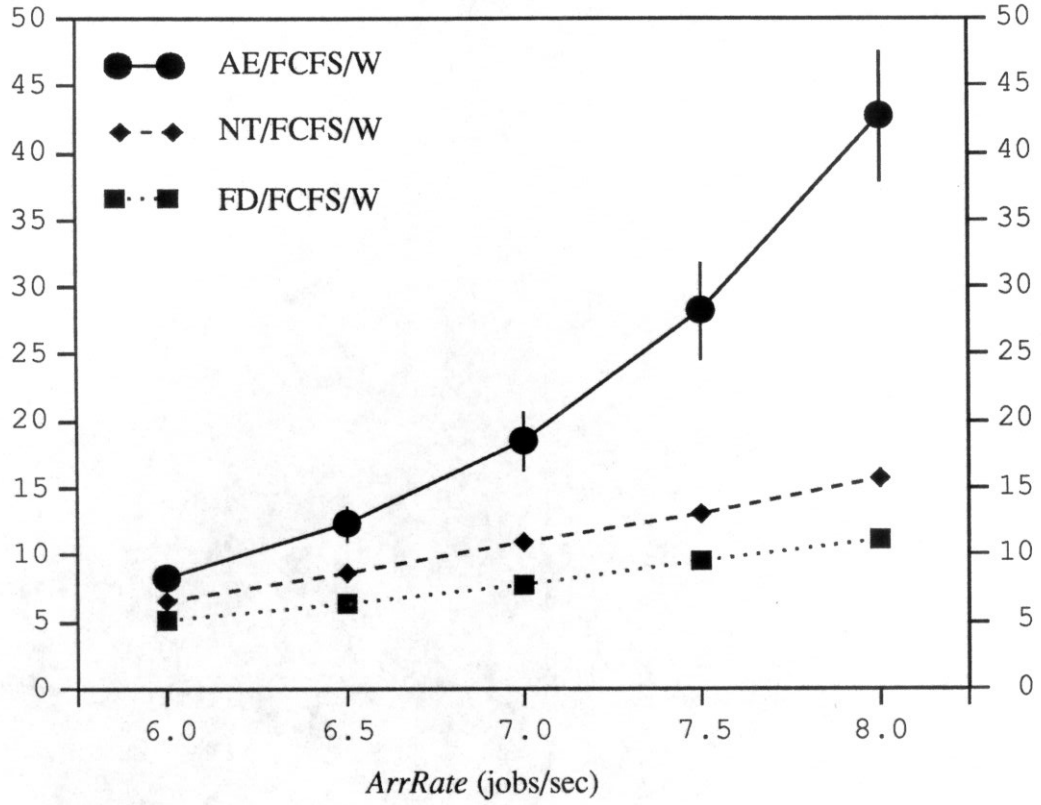


Figure 3-2 Main Memory; Priority Assignment (No CC).

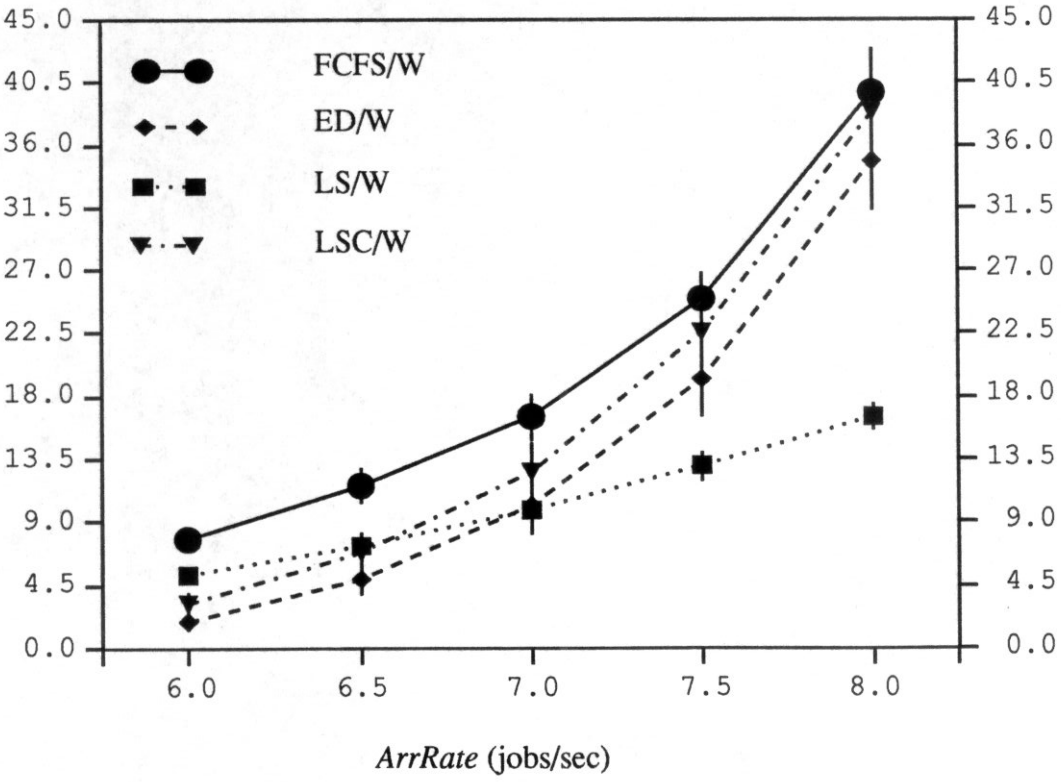


Figure 3-3 Main Memory; Priority Assignment (No CC).

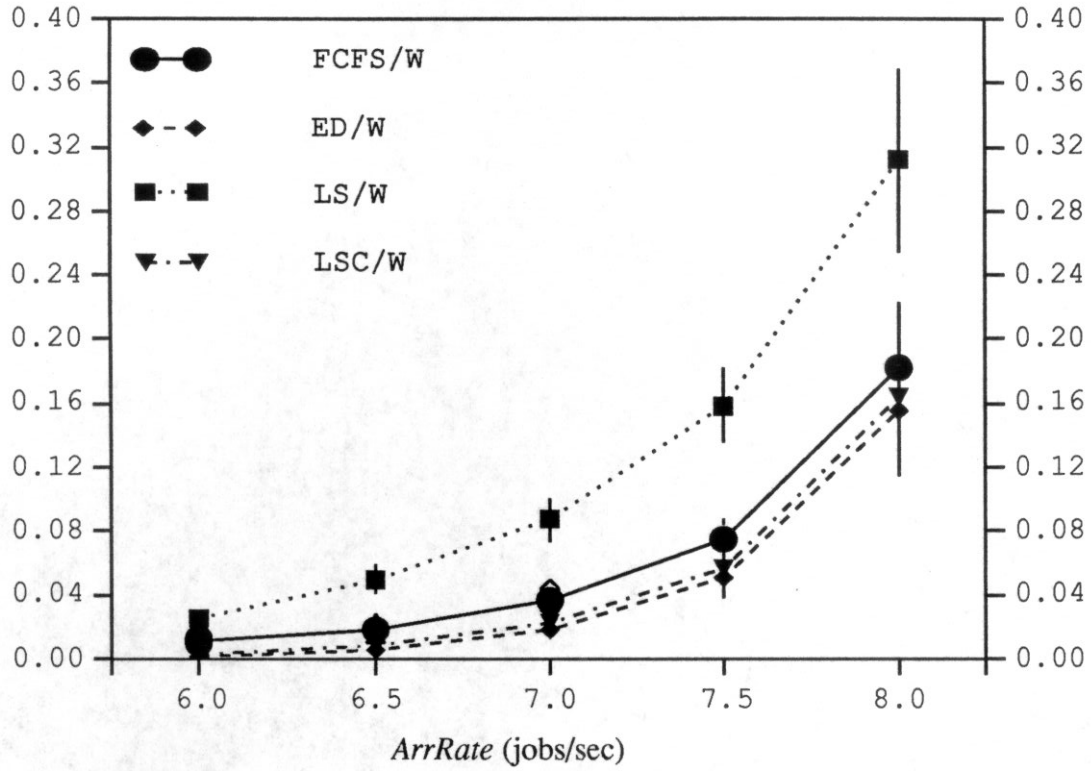


Figure 3-4 Main Memory; Concurrency Control.

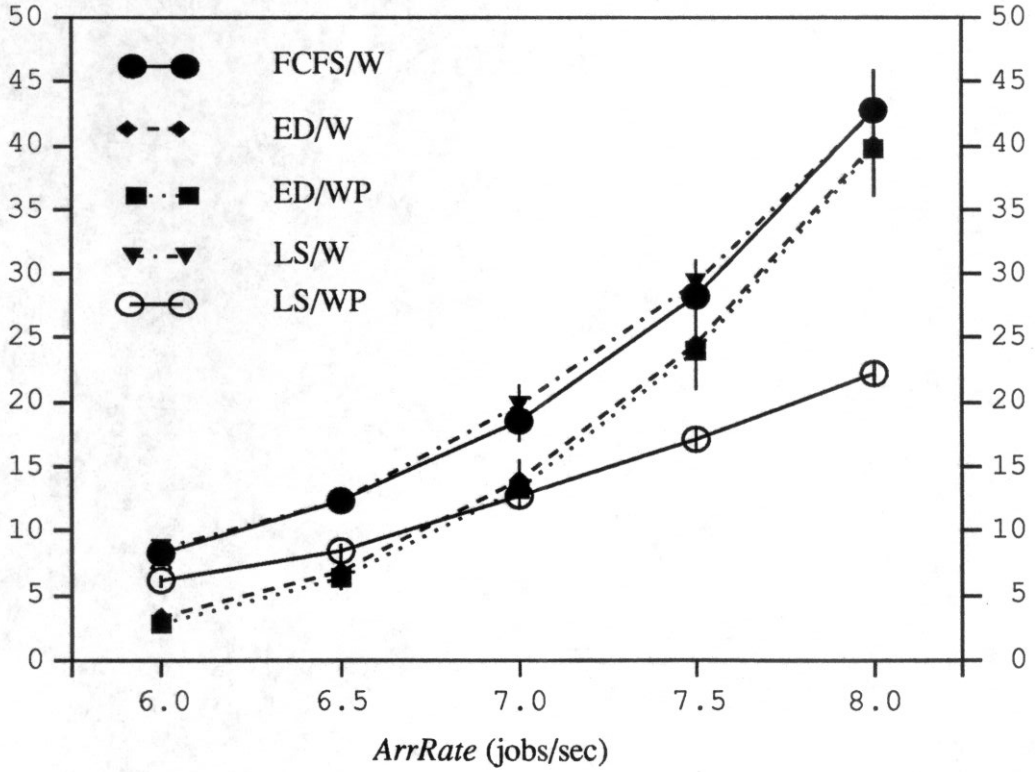


Figure 3-5 Main Memory; Concurrency Control

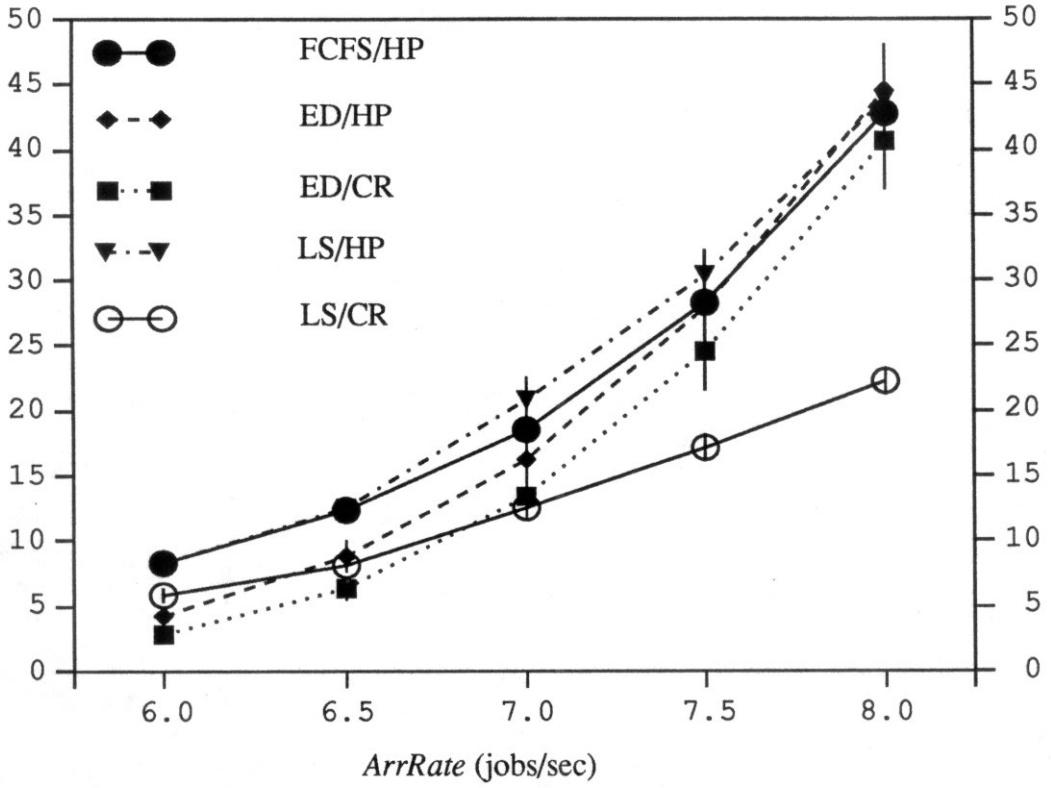


Figure 3-6 Main Memory; Priority Assignment and Concurrency Control

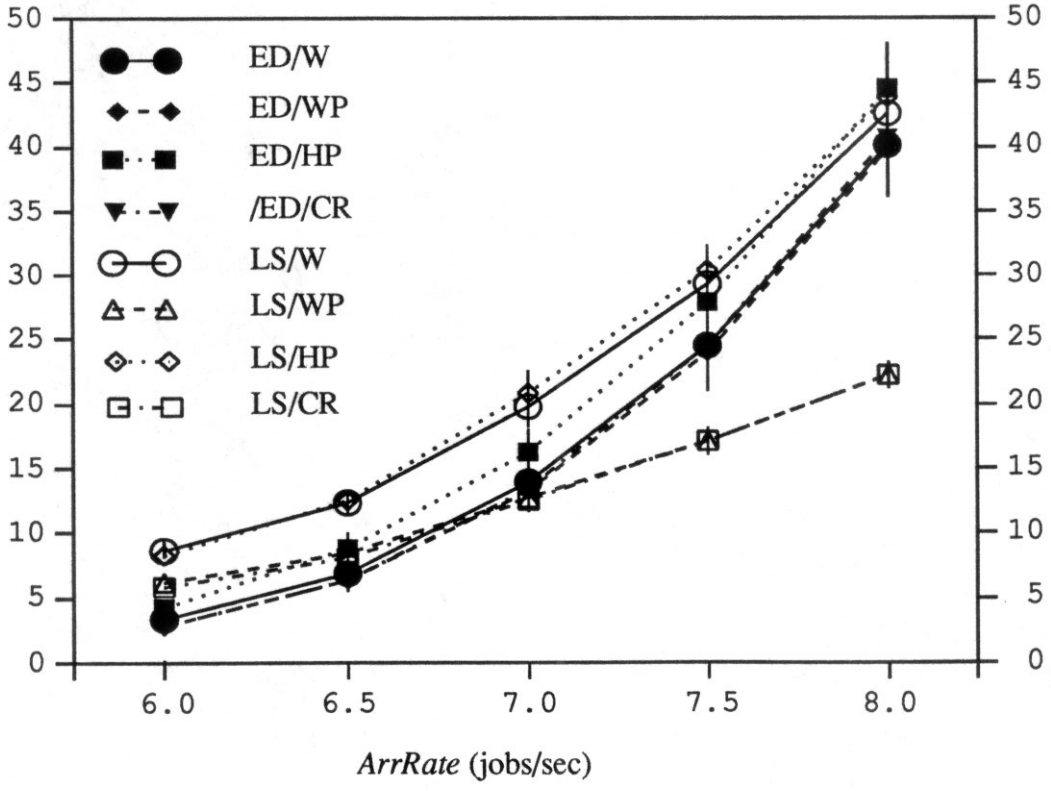


Figure 3-7 Main Memory; Priority Assignment.

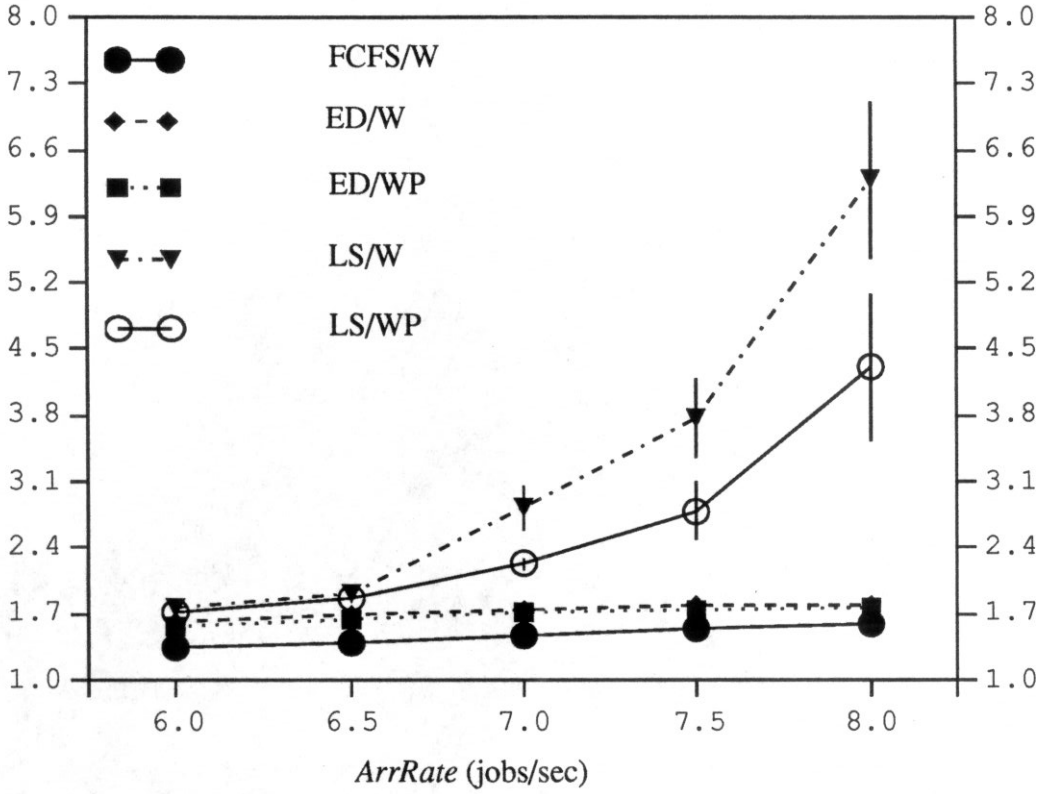


Figure 3-8 Main Memory; Overload Management.

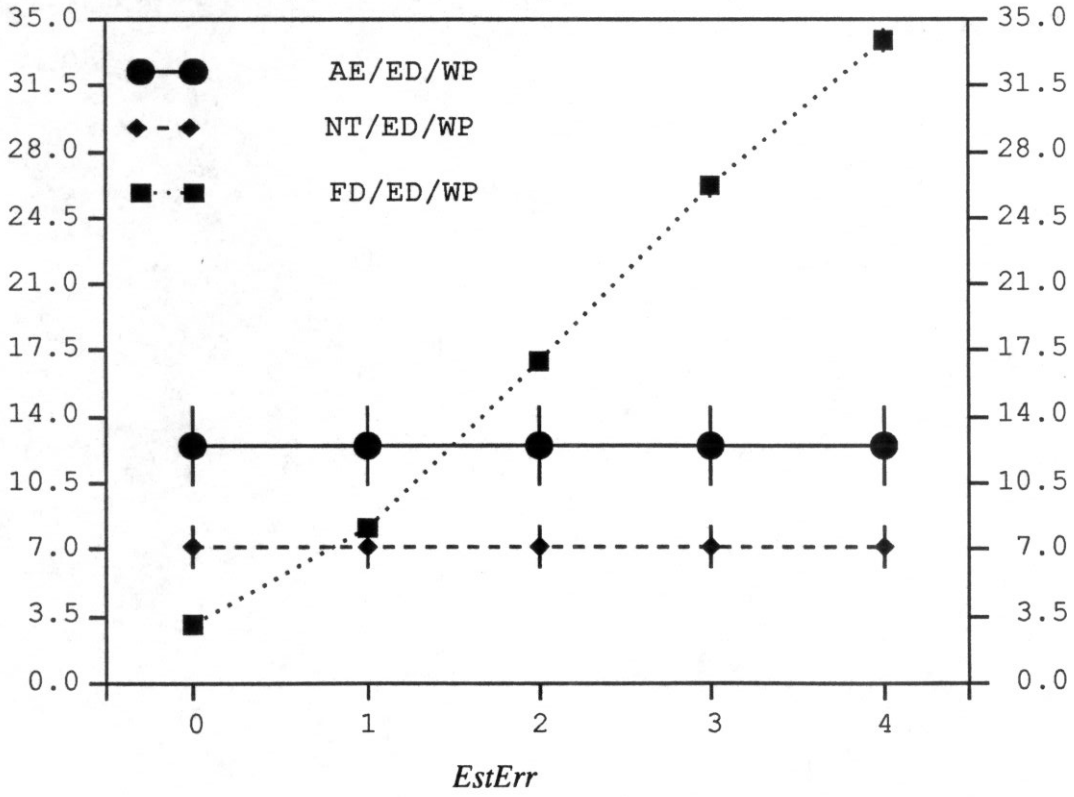


Figure 3-9 Main Memory; Priority Assignment.

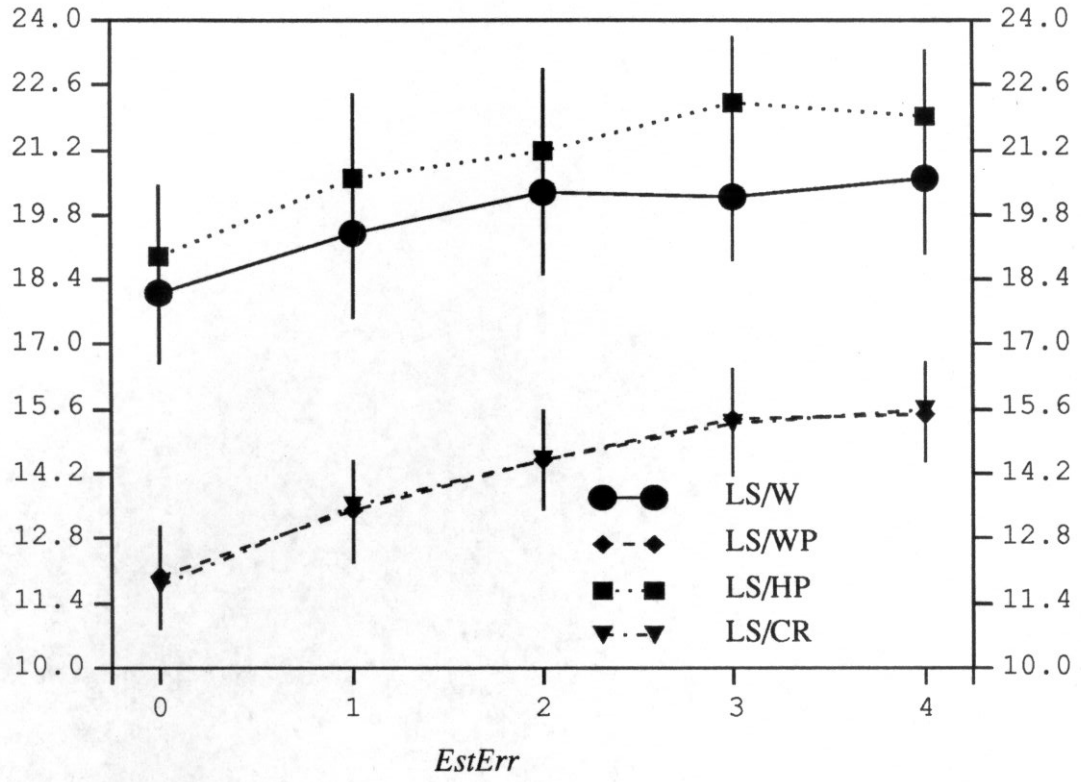


Figure 3-10 Main Memory; Serialized vs. Unserialized.

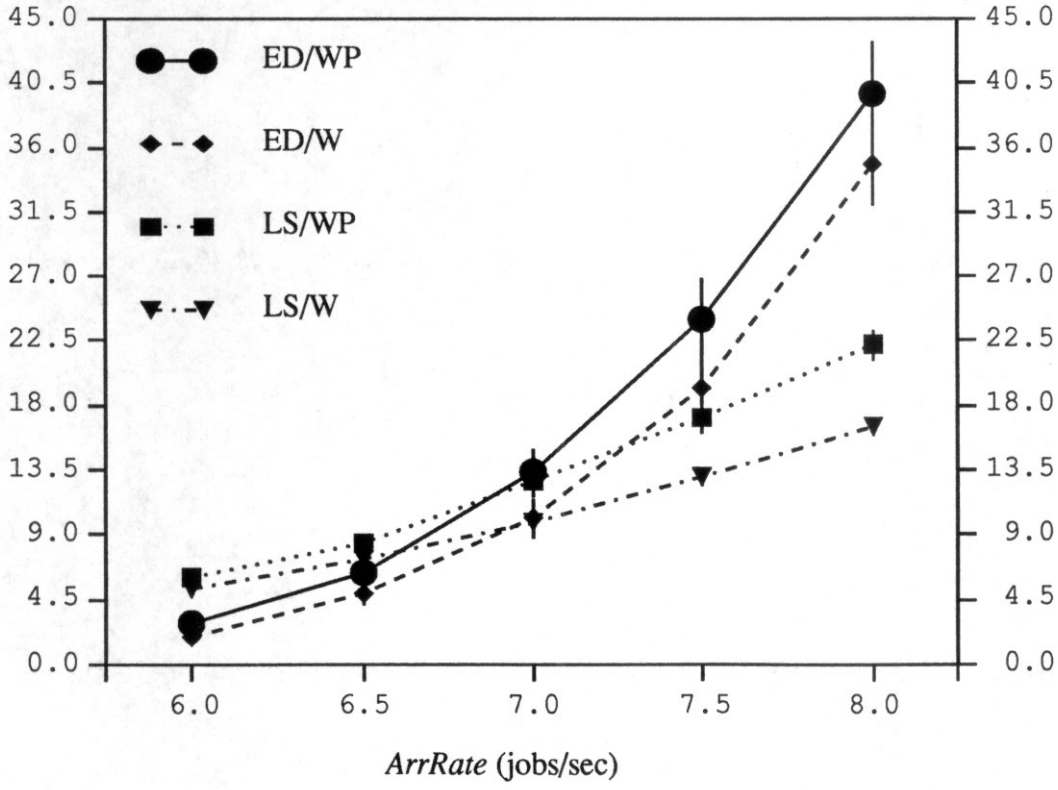


Figure 3-11 Main Memory; Priority Assignment and Concurrency Control.

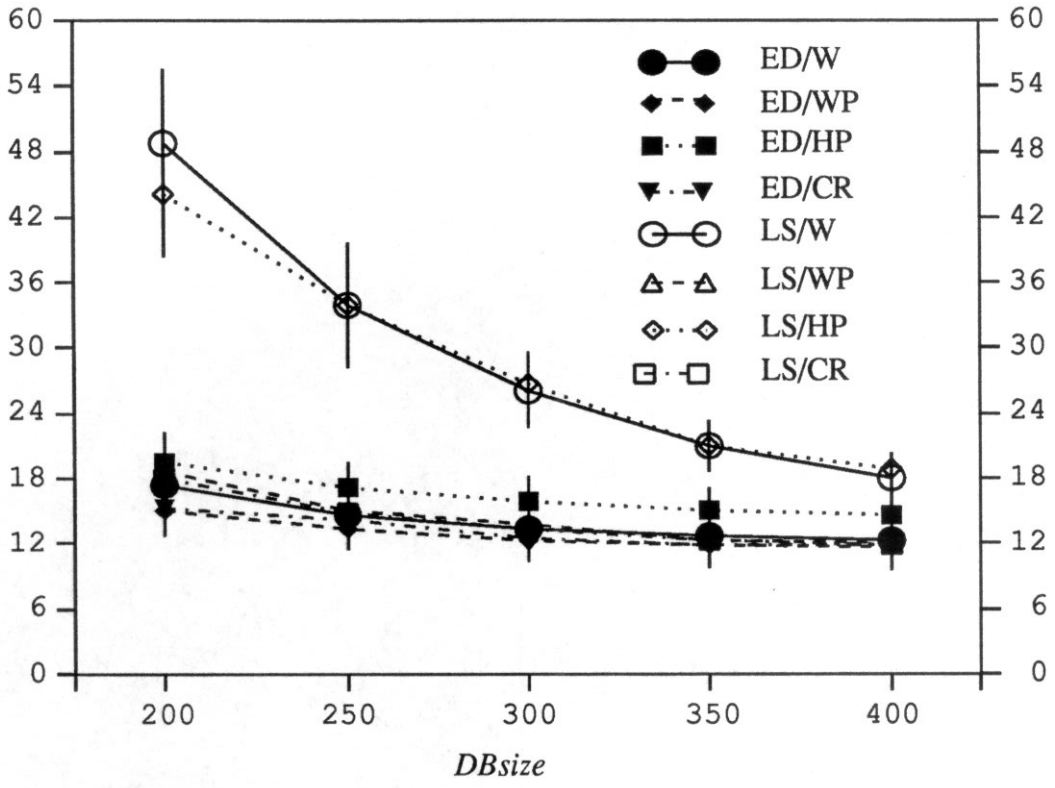


Figure 3-12 Main Memory; Priority Assignment.

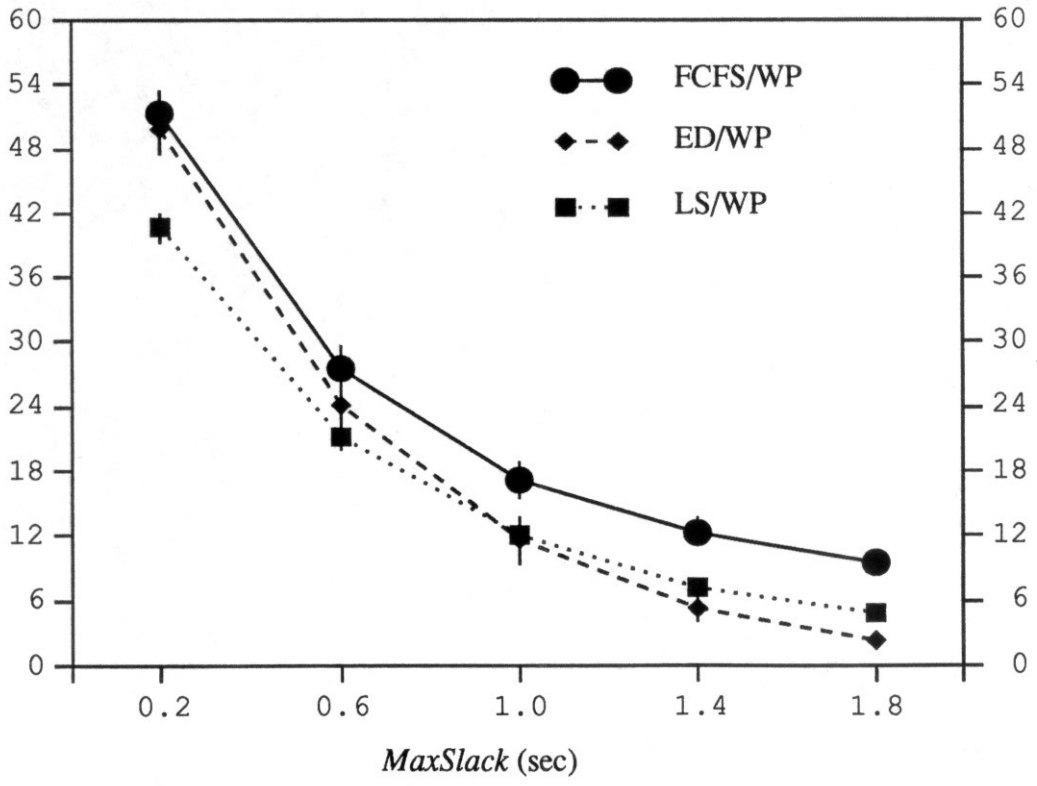


Figure 3-13 Main Memory; Concurrency Control.

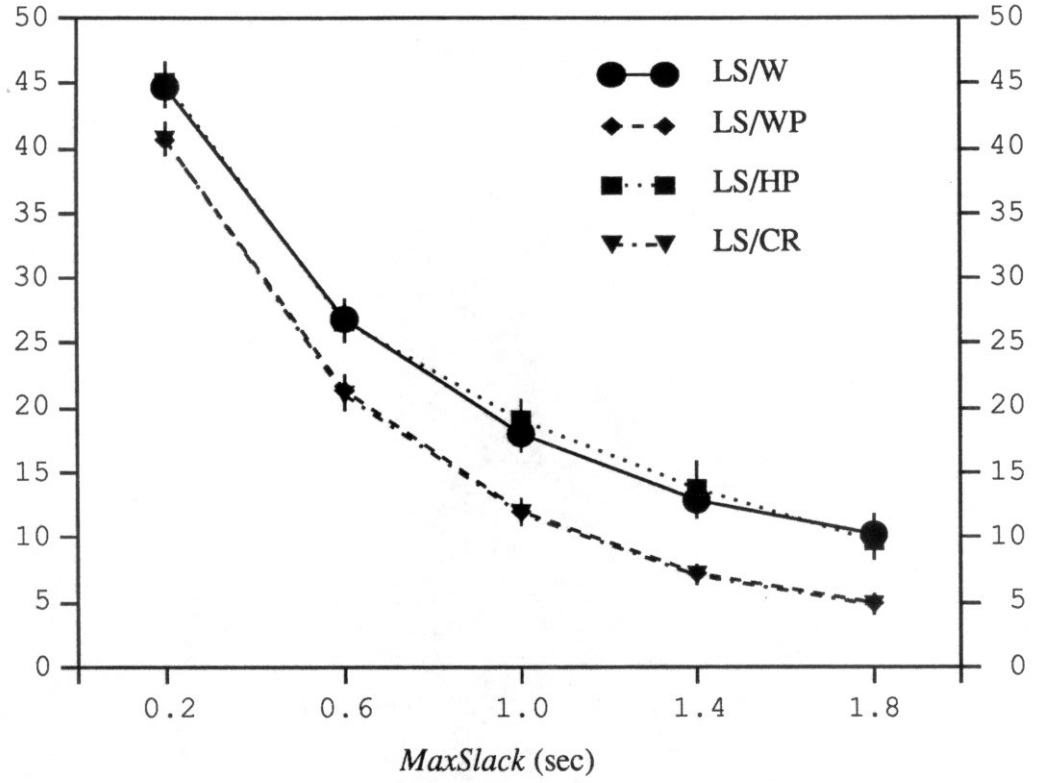
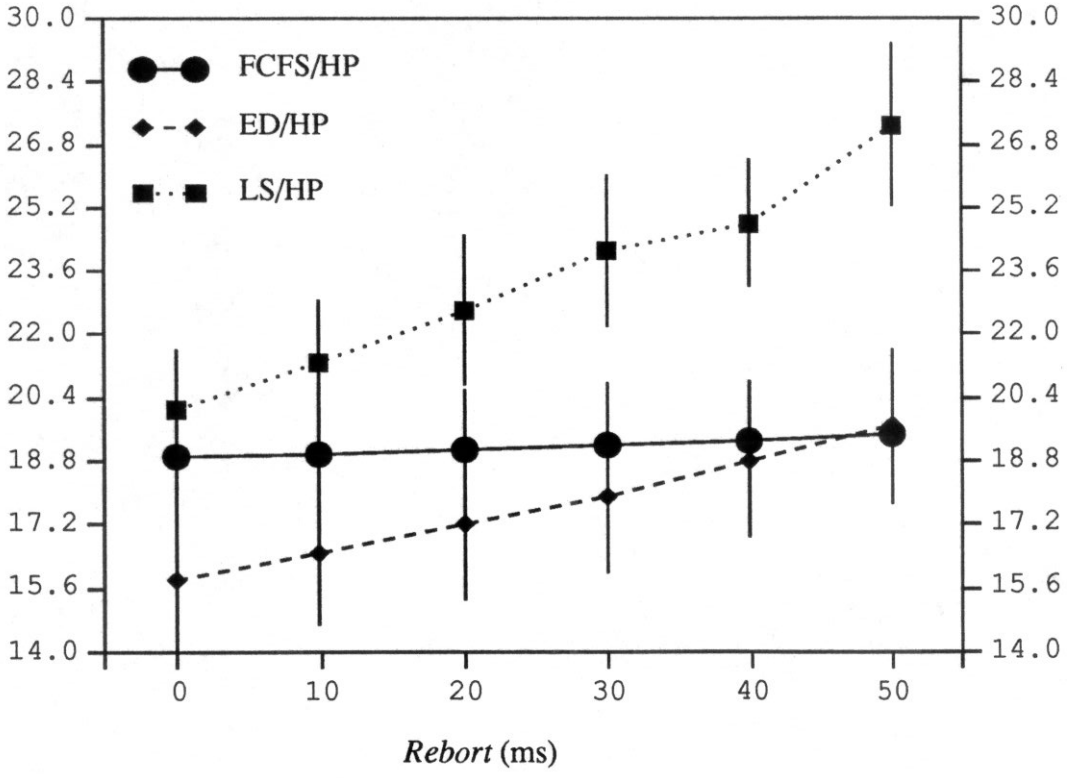


Figure 3-14 Main Memory; Priority Assignment.



3.4 Experimental Results: Disk Resident Database

We continue our performance analysis by studying the algorithms in experiments where the database is disk resident. In this section, emphasis is placed on how I/O scheduling impacts performance. We also allow transactions to obtain both shared and exclusive locks.

The base parameters for the disk resident case are shown in Table 3-5.

Table 3-5 Base Parameters, disk resident database

Parameter	Value	Units	Parameter	Value	Units
DBsize	400	Pages	*MemSize*	200	Pages
Disks	2		*IOtime*	25	ms
ArrRate	7	Jobs/sec.	*Pages*	13	Pages
CompFactor	8	ms.	*Update*	0.25	
MinSlack	0.5	sec.	*MaxSlack*	5	sec.
EstErr	0		*Rebort*	5	ms

Again, we chose these values not to model a specific real-time application but as reasonable values under which to test the algorithms. The arrival rate of 7 jobs per second yields a corresponding CPU utilization of 0.73. The I/O utilization is 0.85. Overall, the parameters have been changed from the settings of Table 3-4 to create a job mix that is I/O intensive rather than CPU intensive. The "average" transaction now consists of 104 ms of computation and 162.5 ms of I/O reads. It also generates 81.25 ms of I/O writes. Under these parameter settings we expect that algorithms which schedule I/O requests according to transaction time constraints will enjoy a performance advantage over algorithms that do not.

3.4.1 Effect of Increasing Load

In this experiment we varied the arrival rate from 6 jobs/sec to 8 jobs/sec in increments of 0.5. The other parameters had the base values given in Table 3-5. The CPU

utilization ranges from 0.62 to 0.83 seconds of computation arriving per second. The I/O system experiences a range in utilization of 0.73 to 0.98 seconds of I/O service requests arriving per second. For many algorithms, especially those which do not use Priority I/O scheduling, the system is unstable at the highest arrival rate, 8 jobs/sec. For this reason, we computed another data point at the arrival rate 7.75 jobs/sec. At this setting the CPU utilization is 0.80 and the I/O utilization is 0.95.

3.4.1.1 I/O Scheduling

The first question which we address is whether using priority to schedule I/O requests helps the system meet transaction deadlines. Figure 3-15 graphs %Missed Deadlines for ED/WP/FIFO and ED/WP/Priority. The results show clearly that the Priority scheduling discipline is superior to FIFO at low load levels and especially at higher loads. If we examine the throughput curves for these two algorithms, Figure 3-16, we see that ED/WP/FIFO becomes saturated earlier than ED/WP/Priority, namely at 7.5 jobs/sec versus 7.75 jobs/sec. Thus we conclude that Priority I/O scheduling enables our algorithms to achieve better performance and higher rates of utilization.

These observations are true for ED when combined with the other three concurrency control polices. They are also true when LS or FCFS is used to assign priority. Since it is always better to use Priority I/O scheduling (at least under these parameter settings) all algorithms in the ensuing discussions will use Priority I/O scheduling unless otherwise noted. Section 3.4.5 will discuss a situation where Priority I/O scheduling, as we have implemented it, is detrimental to performance.

3.4.1.2 Priority Assignment

To eliminate concurrency control as a factor in the performance of the priority assignment policies we performed an experiment with concurrency control turned off,

i.e., all lock requests were granted immediately and the resulting schedule was non-serializable. Figure 3-17 graphs %Missed Deadlines for each of the three priority policies. Both ED and LS are much better than FCFS. Up to *ArrRate* = 7.5, both ED and LS are able to meet all deadlines. After this point both algorithms begin to miss deadlines and by *ArrRate* = 7.75, LS is performing better than ED. Again, this is because ED assigns a high priority to transactions which will miss their deadlines anyway. This prevents other transactions from meeting their deadlines.

3.4.1.3 Concurrency Control

When we pair the four different concurrency control techniques with the priority policies we learn that the performance results, to a great extent, confirm what we observed in the main memory experiments (Section 3.3). Figure 3-18 graphs %Missed Deadlines for all four concurrency control policies with ED. For the two lowest load settings, ED/WP, ED/HP and ED/CR have similar performance and all are better than ED/W. As the load increases, ED/W and ED/HP begin to perform badly while ED/WP and ED/CR are still similar and performing better than W and HP. Finally, at the highest load settings, ED/WP is clearly the best policy. Algorithm ED/CR is second best, then ED/W. For *ArrRate* = 7.5 and above, the system executing with ED/HP is unstable and the data points are not shown. Under high load settings, and high conflict rates, HP produces too many aborts and restarts thereby increasing the effective load and causing the system to become unstable. For *ArrRate* = 7.75 the system is unstable for the ED/W and ED/CR algorithms.

Figure 3-19 graphs %Missed Deadlines for all four concurrency control policies with LS. The general behavior of the four concurrency control policies is the same as observed with ED. A notable exception is that the LS/W and LS/HP algorithms cause

the system to become unstable even earlier than their ED counterparts. (This can be seen from examination of the throughput curves for the different algorithms. We do not show these graphs here.)

Both WP and CR use priority inheritance which is especially important when the database is disk resident and the job mix is I/O intensive. To meet transaction deadlines it is essential for the system to minimize I/O queue waiting time for the most important jobs. Priority inheritance allows the Priority I/O scheduler to accomplish just that. Neither W nor HP uses priority inheritance and they perform poorly with ED and LS. Similar observations apply to FCFS.

3.4.2 Changing Disk Access Time

In the preceding section we saw that the Priority I/O scheduling discipline, when used with the correct priority and concurrency control polices (e.g., LS and WP), increases greatly the percentage of transactions which meet their deadlines versus the same algorithm which does FIFO I/O scheduling. However both methods ignore disk head position and do not minimize the average seek time. Would a disk scheduling algorithm which minimizes average seek time, (e.g., SCAN) but is ignorant of transaction time constraints, yield better performance in a real-time system than our algorithm which considers transaction time requirements? We cannot answer this question directly but we have performed an experiment which indicates how much faster the average seek time of a SCAN type algorithm would have to be for it to match the real-time performance of our Priority algorithm. In this experiment we fixed the parameter settings at the values shown in Table 3-5. Then we varied the speed of the disks by increasing the value of *IOtime* from the initial 21 ms. to 25 ms. in steps of 1. Figure 3-20 plots the performance of LS/WP/FIFO, which, like a SCAN type algorithm, does not use transac-

tion priorities for I/O scheduling. The horizontal line is the performance of LS/WP/Priority when *IOtime* = 25 ms. The two curves intersect at *IOtime* = 22.5 ms indicating that a SCAN type algorithm must have an average seek time roughly 10% less than Priority in order for it to yield comparable real-time performance. Of course it is possible to design a SCAN type algorithm which also uses transaction priorities, but we have not investigated this possibility.

3.4.3 Performance Under a Sudden Load Increase

In the previous experiments, we studied the performance of the various algorithms under an increasing but steady load. In this experiment we study the algorithms under a different type of load increase namely a batch of arrivals in an otherwise idle system. To simulate this input step function we programmed the simulator so that a set of transactions arrived all at the same time. The system then executed all the jobs in the set. The number of jobs in the set varied from 20 to 40 in steps of 5. The parameters controlling transaction characteristics were changed to reflect the fact that jobs arrive all at once: parameter *Pages* = 10, *CompFactor* = 10, *MinSlack* = 0.75, and *MaxSlack* = 7.5. The remaining parameters had the values shown in Table 3-5.

In reality, we would not expect all jobs in an overload to arrive at the same instant, nor would we expect the system to be completely idle at this time. However, our idealized step function model lets us study the impact of an overload without distracting second order effects. If an algorithm performs well under a step function, we could also expect it to do well in a system that has plenty of spare capacity under normal circumstances but is suddenly faced with a flurry of jobs (e.g., a radar system facing a sudden all-out enemy attack).

A good strategy for operation in this situation might be to employ an overload management policy to abort transactions which have missed their deadlines. If these transactions must be executed anyway then they should be restarted only after the spike has passed. We may also want to limit priority inversions from occurring. The reasoning is that if a high priority job blocks, it will almost certainly miss its deadline if forced to wait on a low priority job. Instead we may want to abort the low priority job and restart it later when the load has decreased.

Figure 3-21 shows the four different concurrency control policies for the ED priority assignment policy. I/O scheduling is done according to job priority. In this experiment ED/HP is clearly the best algorithm. It performs better than both ED/WP and ED/CR which were the two best versions of ED under a steady load. Although for reasons of space, we do not show the curves for the LS algorithms, the results are similar. Namely, LS/HP misses fewer deadlines than the other versions. Also, the performance for ED/WP ED/HP and ED/CR is somewhat better than the corresponding versions of LS.

3.4.4 Effect of Increasing Memory

In this experiment we varied the value of *MemSize* from 180 to 240 in increments of 20. The other parameters had the values shown in Table 3-5. Thus the size of the memory resident fraction of the database varied from 0.45 to 0.6 of the total database. As memory size changes the load on the CPU remains unchanged, namely 0.73 seconds of computation arriving per second. The load on the I/O system varies from 0.91 to 0.74 seconds of I/O requests arriving per second.

Note that the proportion of memory resident database influences the assignment of deadlines to transactions. This happens because deadlines are assigned with respect to

the total service time, both CPU and I/O, required by a transaction. As the memory increases, the I/O service requirement decreases (pages are more likely in memory), and the deadlines are shortened proportionately. In this way we can compare the scheduling of sets of transactions with similar task urgencies in system configurations of various memory sizes.

Our expectation, as memory size increases, is that all scheduling algorithms will perform better. The reason is that transactions will be doing much less I/O and the time spent waiting for service in I/O queues decreases. (CPU utilization and waiting time are unchanged.) Thus transactions will receive quicker service and are more likely to meet their deadlines.

Figure 3-22 graphs %Missed Deadlines for LS paired with each concurrency control method. It confirms our expectation that algorithms will perform better as the memory size increases. In particular LS/HP improves dramatically as the memory size increases. The large number of aborts and restarts has a smaller effect on the I/O system. The results for ED are similar.

3.4.5 Changing the Number of Updates

The value of *Update*, by controlling the size of the read and write sets, impacts the probability of conflict between two concurrent transactions. The value of *Update*, by determining the size of the writeset, also contributes to the I/O load of the system. This is because updates are written to disk after a transaction commits.

In this experiment we varied the value of *Update* from 0.15 to 0.3 in increments of 0.05. There is also a data point at *Update* = 0.275. When *Update* = 0.15, only 15% of a transaction's pages are locked exclusively. When *Update* = 0.3, twice as many pages

are updated. The other parameters had the values shown in Table 3-5. In this experiment the I/O load varies from 0.74 to 0.91. The CPU load remains constant at 0.73.

Figure 3-23 shows the four different types of concurrency control with the ED priority policy and Priority I/O scheduling. The general shape of the curves is as expected: when *Update* is large, the performance is poor because the rate of conflict is high and because the overall I/O load is high. When *Update* is small, the conflict rate is lower and ED/WP, ED/HP and ED/CR perform equally well and all are better than ED/W. As *Update* increases, ED/W and ED/HP perform worse than ED/CR and ED/WP. Algorithm ED/WP is best at the highest conflict rate.

Another experiment shows how giving high priority to disk writes can lead to decreased performance. For this experiment *Pages* = 8, *CompFactor* = 15, and *MaxSlack* = 2. The parameter *Update* is varied from 0.2 to 0.8 in steps of 0.2 The other parameters had the values shown in Table 3-5.

Figure 3-24 plots ED/W with priority I/O scheduling against ED/W with FIFO I/O scheduling. Somewhat surprising is the result that the version with I/O scheduling perform worse than the version without I/O scheduling in the high update (right side) and thus high I/O load part of the scale. The result is explainable, however, when we note that at this end of the scale most of the I/O, roughly 62%, consists of writing modified pages back to disk. As we discussed in Section 2.7.2, giving high priority to writes can delay the service of read requests with lower priorities. One conclusion that we can draw from this is that it may be advisable to give I/O writes lower priorities than reads. We investigate this issue in depth in Chapters 4 and 5.

Figure 3-15 Disk Resident; I/O Scheduling.

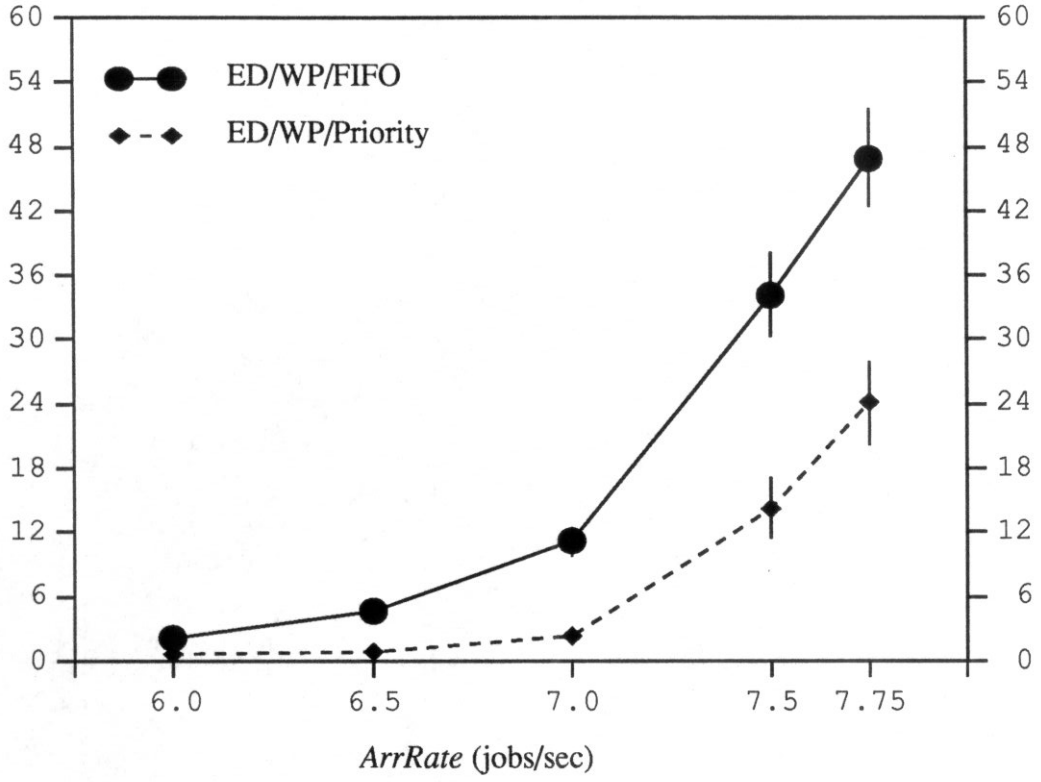


Figure 3-16 Disk Resident; I/O Scheduling.

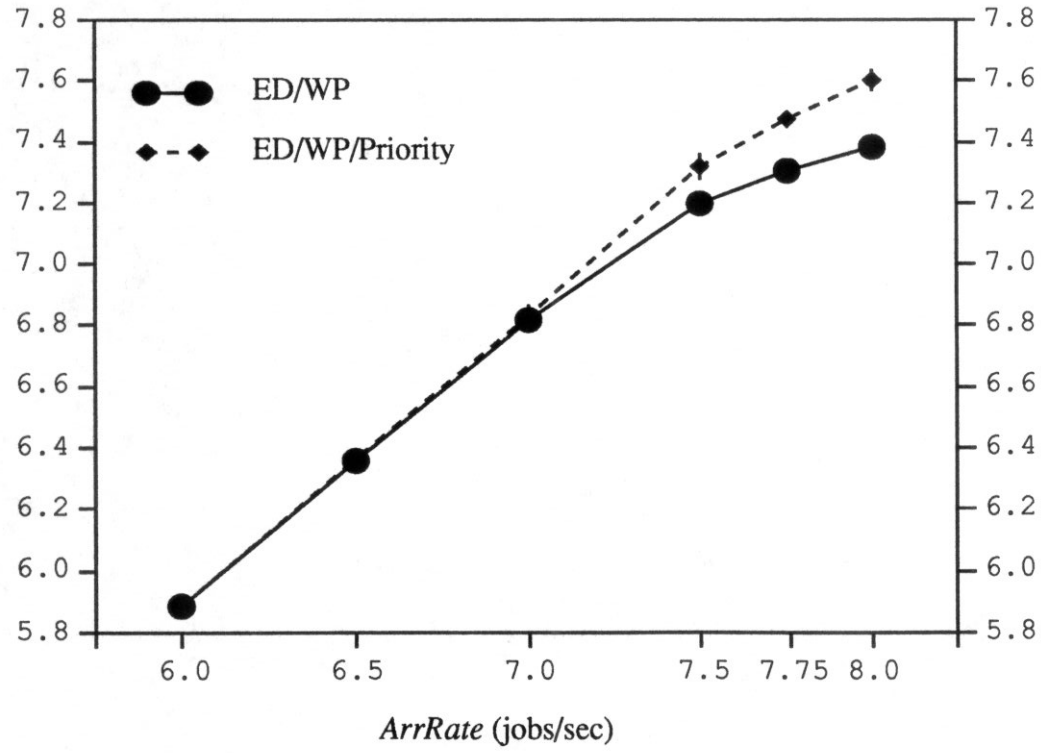


Figure 3-17 Disk Resident; Priority Assignment (No CC).

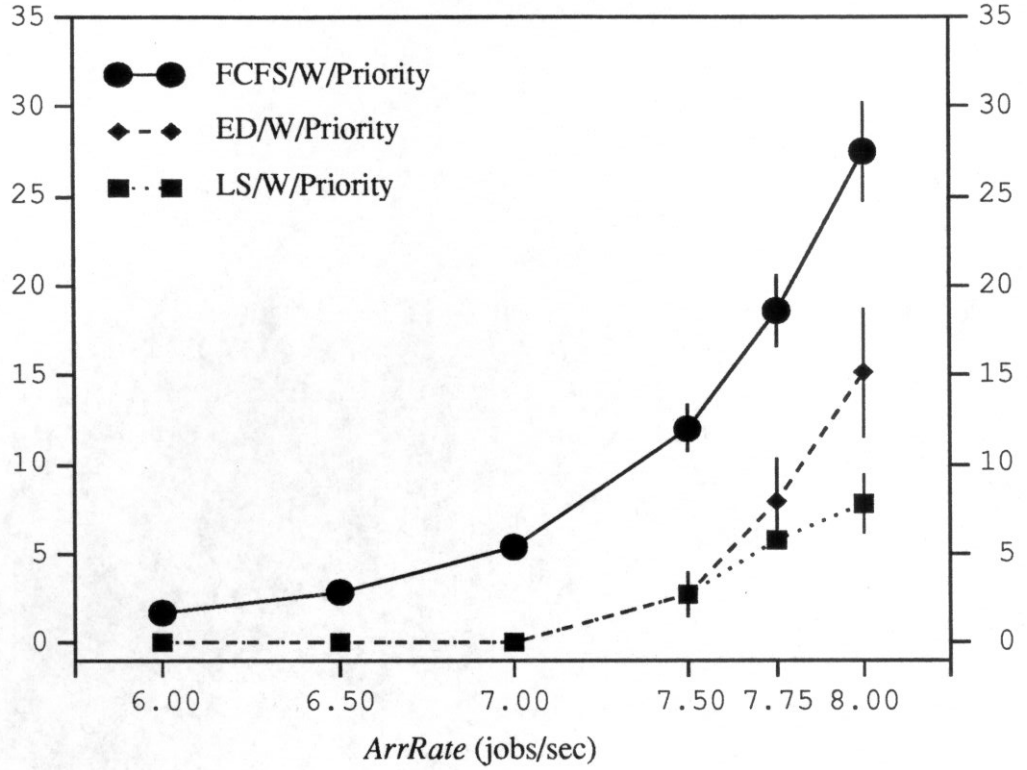


Figure 3-18 Disk Resident; Concurrency Control.

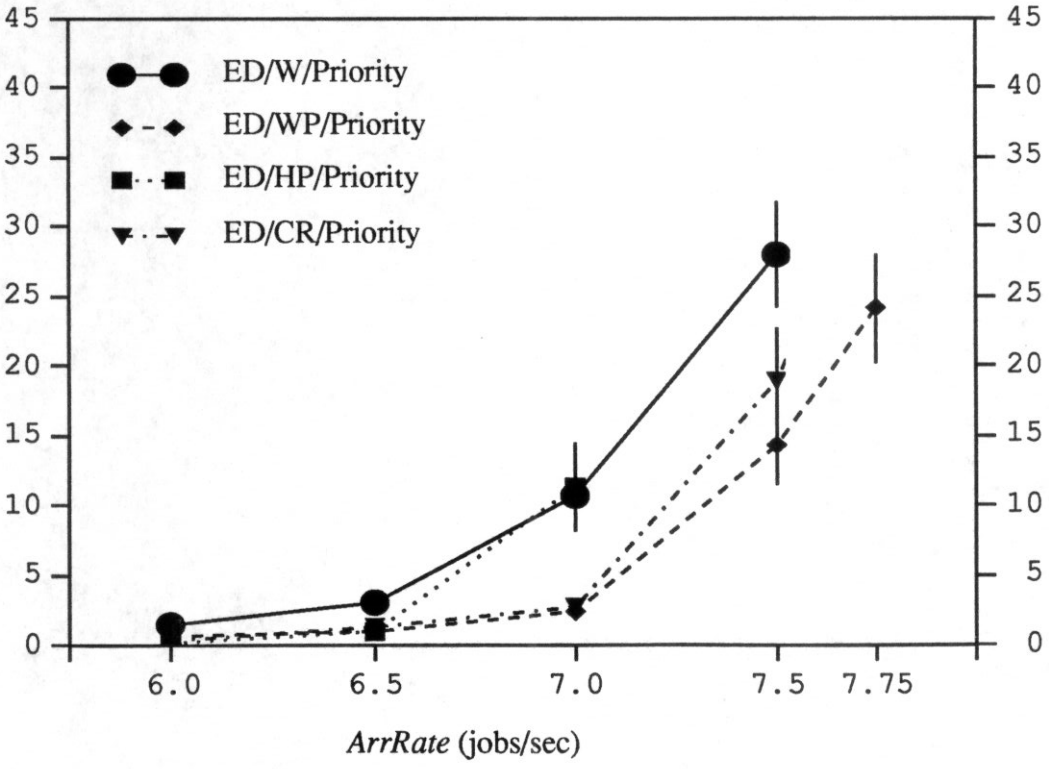


Figure 3-19 Disk Resident; Concurrency Control.

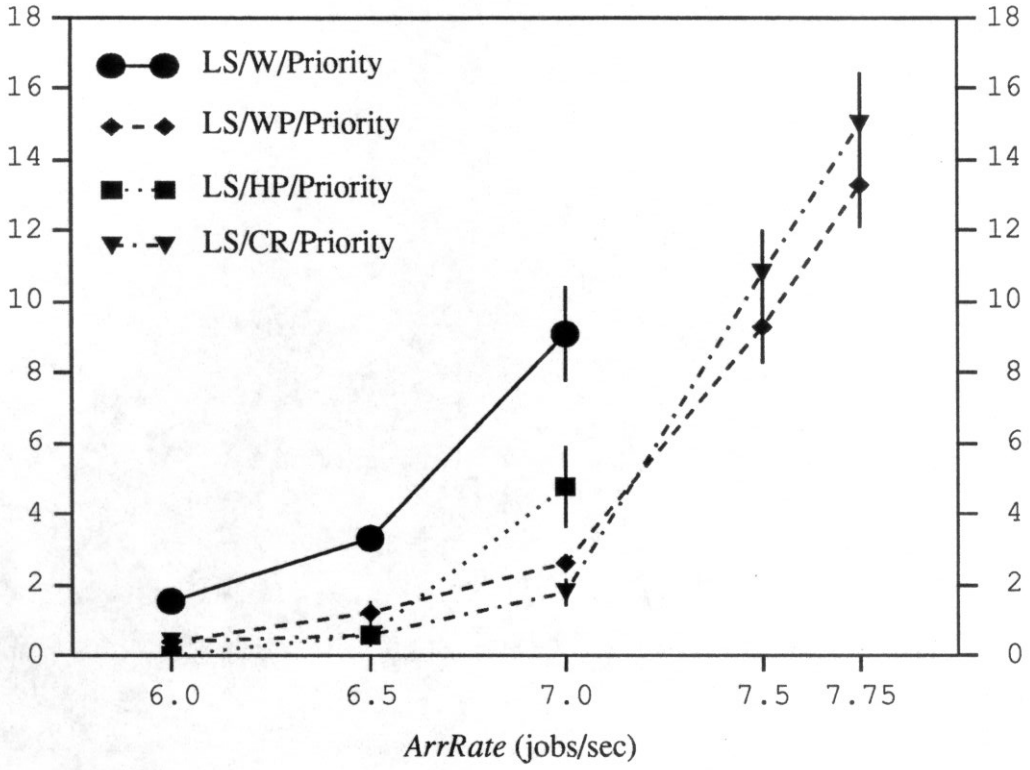


Figure 3-20 Disk Resident; I/O Scheduling.

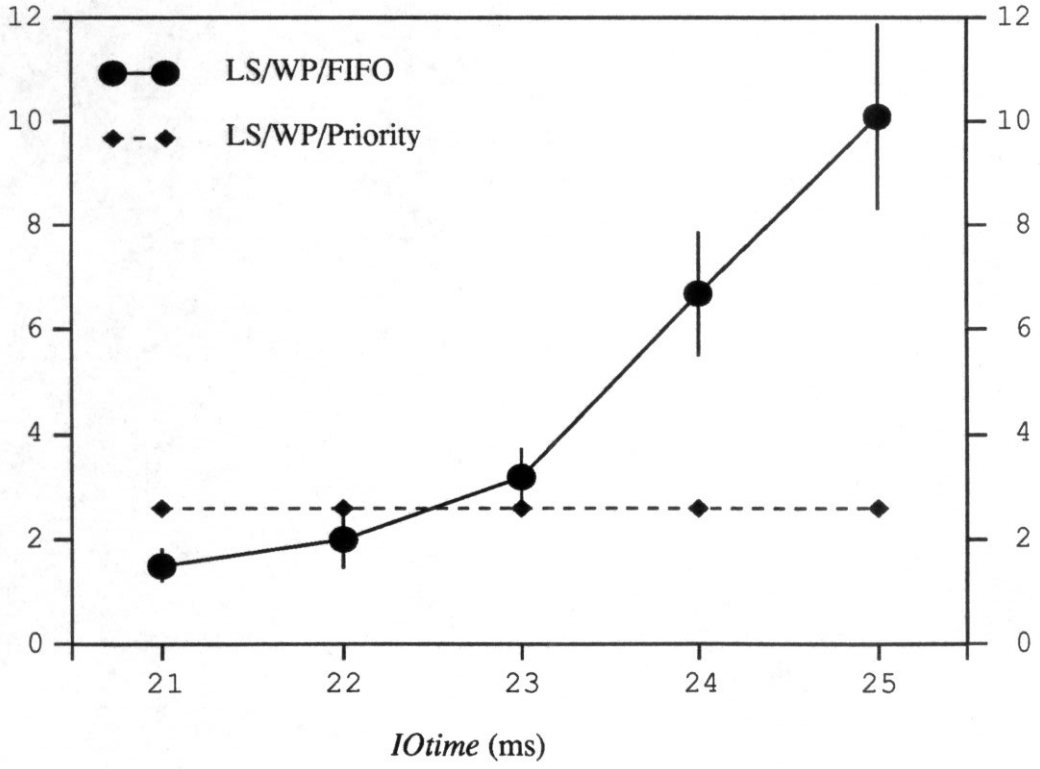


Figure 3-21 Disk Resident; Input Step Function.

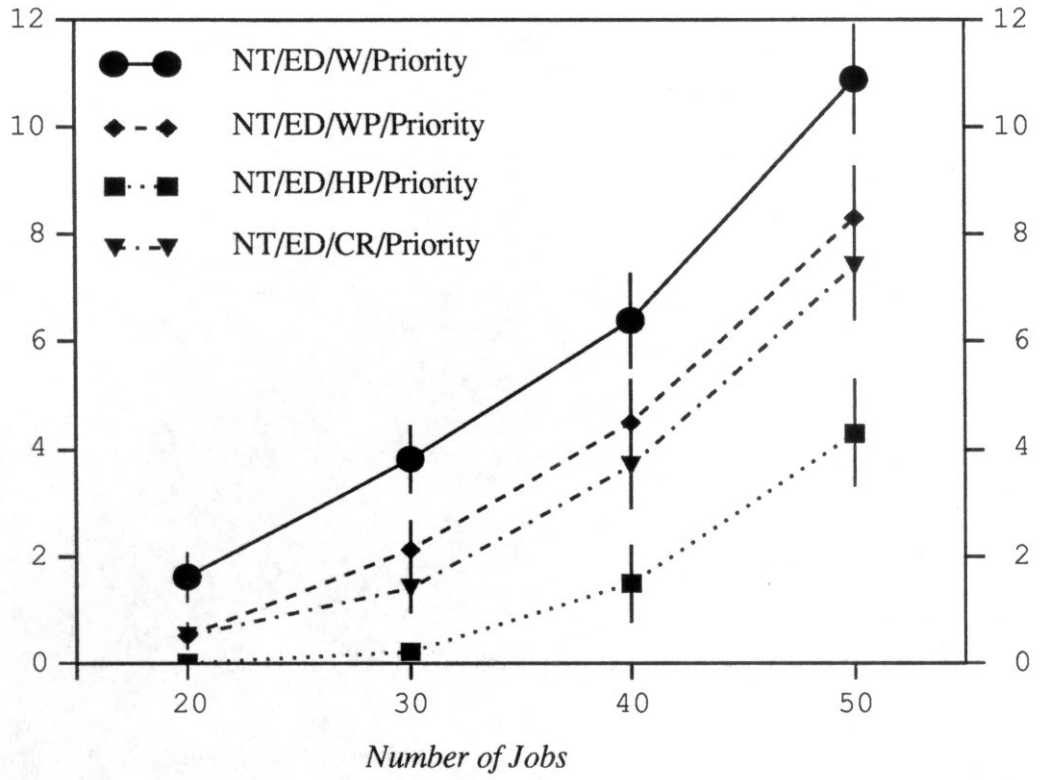


Figure 3-22 Disk Resident; Concurrency Control.

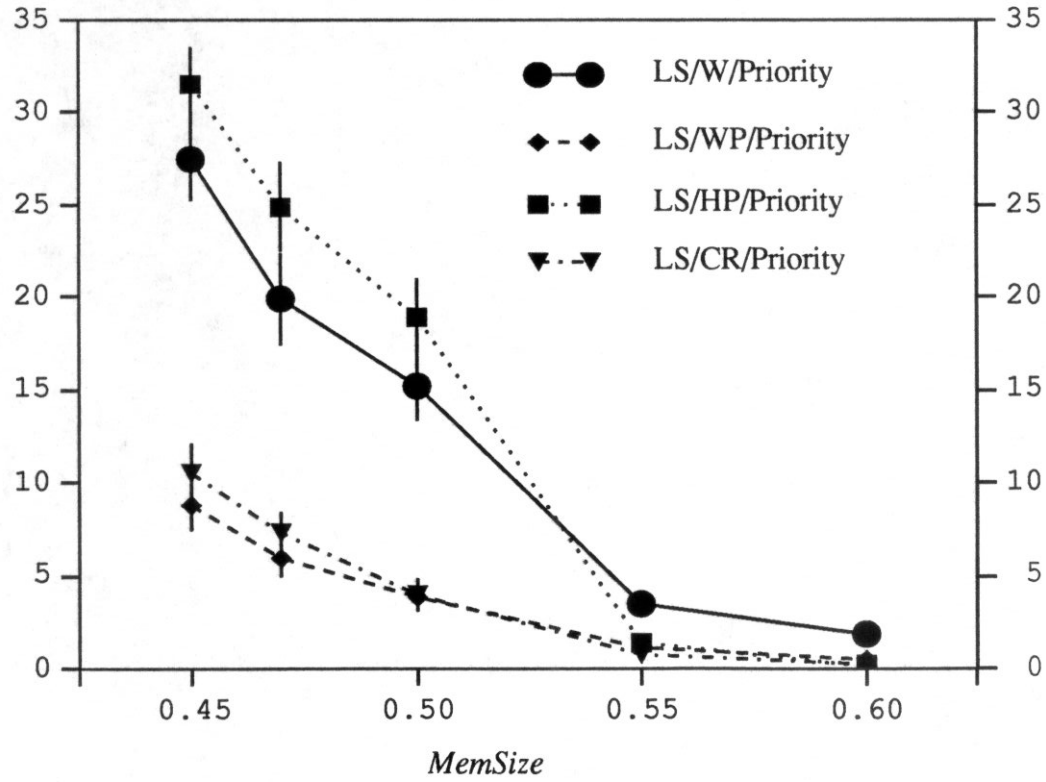


Figure 3-23 Disk Resident; Concurrency Control.

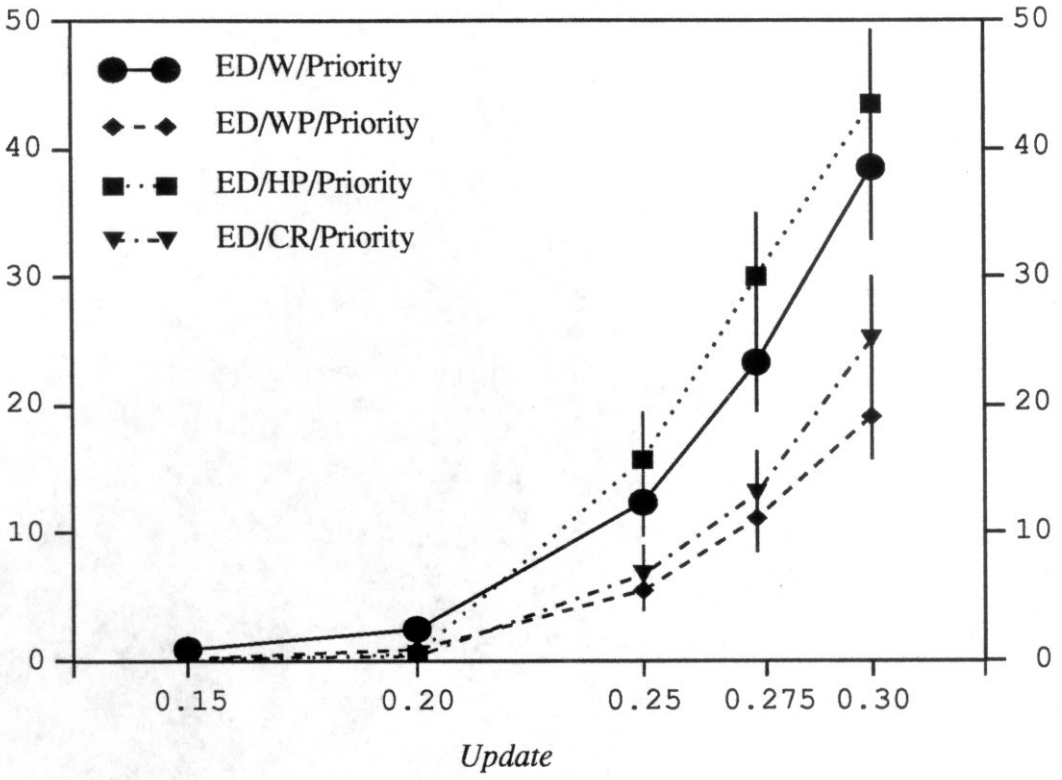
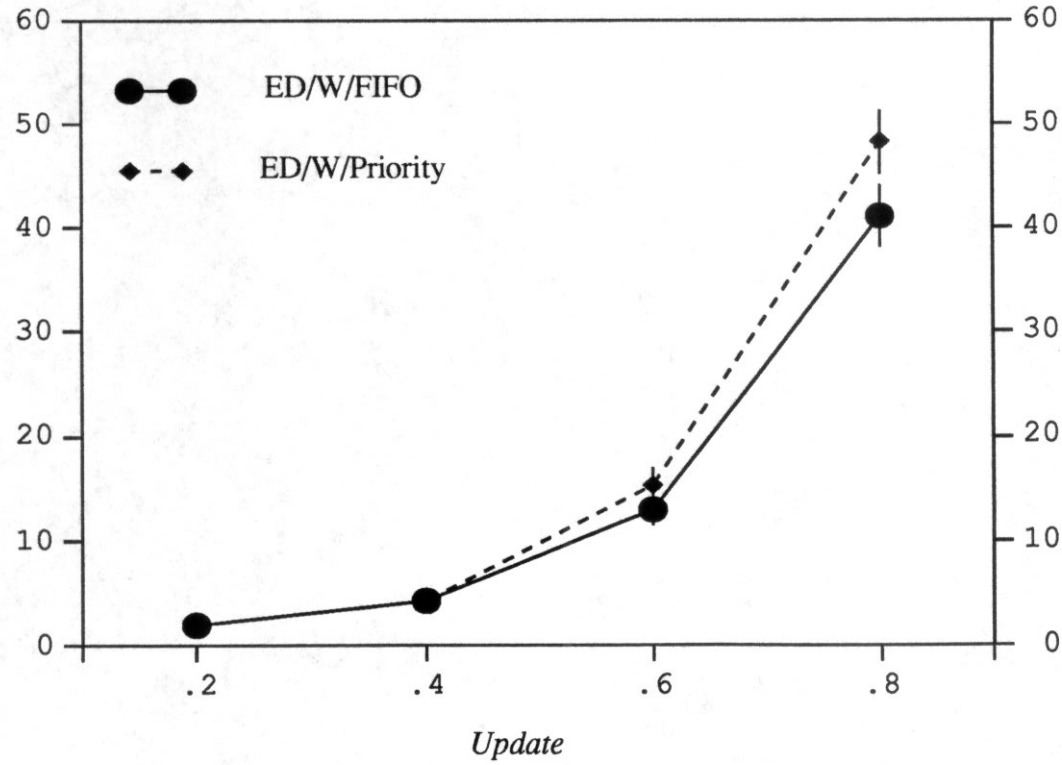


Figure 3-24 Disk Resident; I/O Scheduling.



3.5 Conclusions

Our simulation results have illustrated the tradeoffs involved, at least under one representative database and transaction model. Before reaching some general conclusions, we would like to make two observations. The first observation is that our base parameters represent a high load scenario. One could argue that such a scenario is "unrealistic." However, we believe that for designing real-time schedulers, one must look at precisely these high load situations. Even though they may not arise frequently, one would like to have a system that misses as few deadlines as possible when these peaks occur. In other words, when a "crisis" hits and the database system is under pressure is precisely when making a few extra deadlines could be most important.

It could also be argued that some of the differences between the various scheduling options is not striking. In many cases, the difference between one option and another one is a few percentage points. If we were discussing transaction response times, then a say 10 percent improvement would not be considered impressive by some. However, our graphs show missed deadlines (in most cases) and we believe that this is a very different situation. Again, the difference between missing even one deadline and not missing it could be significant. Thus, if we do know that some scheduling options reduce the number of missed deadlines, why not go with the best one?

And which are the best options? It is difficult to make any absolute statements, but we believe the following statements hold under most of the parameter ranges we tested.

- (a) Using an overload management policy test to screen out transactions that have missed (NT) or are about to miss their deadlines (FD) greatly improves system performance. Aborting a few late transactions helps all remaining jobs to meet their

deadlines. Of course, in some applications we may be forced to use the less efficient All Eligible (AE) policy because transactions must be executed even if they miss their deadlines.

- When the database is memory resident:

  (a) Of the tested priority policies, when missed deadlines is the metric, Earliest Deadline (ED) performs best at lower load settings while Least Slack (LS), statically evaluated, performs best at higher load settings. Policy LSC, (Least Slack, continuous evaluation) performs slightly worse than ED but better than First Come First Serve (FCFS) which is the poorest policy overall. When Mean Tardy Time is the metric, LS, statically evaluated, is the poorest performing policy, at all load settings. The best two are ED and LSC while FCFS is slightly worse.

  (b) Of the tested concurrency control policies, Wait Promote (WP) and Conditional Restart (CR) are clearly the preferred policies to pair with LS. Policy LS performs poorly when paired with either Wait (W) or High Priority (HP). For ED, policies W, WP and CR all have similar performance. Policy HP is the worst choice for ED. It makes no difference which policy is used with FCFS because the database is memory resident.

  (c) Bias in the runtime estimate E has a large effect on the overload management policy FD. The effect on LS is relatively small but the effect on LSC is larger. Also the effect on the concurrency control policy CR is small. As expected, all policies perform worse when the error in E increases.

  (d) The priority assignment policy LS permits a greater level of concurrency than both ED and FCFS. Greater levels of concurrency increases the probability of conflicts and priority inversions. Priority inversions can be handled effectively

by using priority inheritance. Thus LS/W and LS/HP are more sensitive to experiments which change the rate of conflict than are LS/WP and LS/CR.

- When the database is disk resident:

  (a) Of the tested priority policies for real-time database systems, Least Slack (LS) is the best overall. It always performed better than FCFS, and was better than ED under the higher load ranges. Earliest deadline (ED) is the second choice for assigning priorities.

  (b) Of the concurrency control policies we tested, Wait Promote (WP) is the best overall. It performs very well when combined with either LS or ED. Conditional Restart (CR) is the second choice. High Priority (HP) does not do well with either LS or ED.

  (c) When the I/O system is heavily loaded, using real-time transaction priorities to schedule I/O requests yields significant performance gains over scheduling I/O requests in a FIFO manner.

- When the load is an "input step function":

  (a) ED is the best priority policy to use. It performs better than LS at the higher load settings for three of the four concurrency control policies.

  (b) HP is the best concurrency control policy. It is the best of the four when combined with either ED or LS. The second and third choices for concurrency are CR and WP.

Differences in the transaction model make it difficult to compare our results directly to those reported in **[HSTR]**. However we can report broad agreement on some results. The method of CPU scheduling is the most important policy. This is sensible as

transactions issue requests for other resources only after they have gained the CPU. Using real-time transaction information in concurrency control policies yields better performance over policies that do not use this information. Finally, for CPU scheduling, they reported that ED is most sensitive to deadline distributions and is useful only when deadlines are not too tight. This agrees with our finding that ED performs well only under the lower load settings.

Chapter 4 Scheduling Disk Requests with Deadlines

4.1 Introduction

So far, we have used a simple model of an I/O system was used to study the effect of using real-time priorities to schedule disk accesses. Our results indicated that priority based disk scheduling could be beneficial to overall performance. However the interaction between reads and writes needed to be carefully controlled. This chapter examines in more detail the problem of scheduling I/O requests with deadlines.

Consider a common method for modeling a transaction in centralized database system: a transaction is an alternating sequence of data actions (reads and updates) and compute actions. The sequence terminates with a COMMIT action where log processing is done and locks are released. Finally, the modified pages produced by the transaction are written to the disk resident database. If we make the assumption that memory is large enough to hold all uncommitted updates, i.e., a ¬STEAL buffer management strategy, then the following observations are true:

- Requests to read pages from the disk resident database are always performed *before* the requesting transaction is committed.
- Requests to write modified pages back to the disk are always performed *after* the transaction which created the modified pages has committed.

These observations are particularly important in the context of a system which executes real-time transactions. It means that the I/O system will service two types of requests: reads and writes. Read requests are issued by uncommitted real-time transactions. These requests should receive service in accordance with the time constraints of the tasks that issued them. In general, this means that read requests issued by tasks with immediate deadlines are serviced before read requests issued by tasks with later dead-

lines. How the time constraints of the issuing tasks can be inherited by the read requests themselves is but one question. How should we evaluate system performance in meeting these time constraints is another.

Read requests have explicit time constraints that they inherit from the tasks that issued them. Write requests do not have explicit time constraints because they are not issued by real-time tasks. For example, a buffer manager regularly writes modified pages to disk in order to maintain a suitable amount of free memory. Typically, the performance of a buffer manager is not measured with real-time metrics. A write request must be serviced but *when* it should be serviced is not clear. On the one hand, write requests should be serviced at an average rate that is equal to their average arrival rate. On the other hand, servicing write requests can interfere with the timely servicing of read requests. This interference should be minimized. How can we service write requests and still meet the deadlines of read requests? How can we ensure that write requests are serviced "often enough" even though they lack real-time constraints?

It is certainly conceivable that some write requests in a real-time transaction system will have explicit hard deadlines. For example, a distributed real-time system may implement a timed atomic commit protocol must complete before the next can begin. The completion of a phase is recorded by writing a record to stable storage, i.e., the disk. However we believe that the above observations are true for most I/O requests. Therefore, throughout Chapters 4 and 5 we will refer to I/O requests with deadlines as reads, and requests without deadlines as writes. However our model and scheduling algorithms can work for write requests with deadlines as well.

Finally, there is the problem of scheduling the disk head itself. Traditional algorithms perform seek optimization to meet non real-time performance goals. Will these

same algorithms perform well under real-time metrics? What kinds of algorithms can be developed using deadline information? How well do they perform? Should read requests be handled differently from write requests? How should the requests be sequenced so that time constraints are met and the disk resource is used efficiently?

These questions are addressed in the following sections. Section 4.2 presents our basic model and assumptions. The model contains our methods for creating streams of I/O requests as well as the metrics we will use to measure the performance of the scheduling algorithms. Section 4.3 addresses the problem of guaranteeing a suitable service rate for write requests, and Section 4.4 presents a number of algorithms for scheduling the disk head. Section <IO simulation model> describes the detailed simulation model used to evaluate the performance of the various scheduling techniques and our experimental results are presented in Section 5.2.

4.2 Model and Assumptions

The basic model that we use to study the problem of scheduling I/O requests with deadlines is shown in Figure 4-1. Our model has two halves: memory, which is depicted on the left and the I/O subsystem which is shown on the right. The boundary between the two serves as a deadline checkpoint, a place where we evaluate if time constraints have been met.

Read and write requests are generated not by transactions directly but by a *buffer manager* that lies between the running transactions and the disk handler. This buffer manager receives read and write requests from the transactions, and in turn generates read and write requests for the disk handler.

If a transaction read request cannot be satisfied by a page in the buffer, then the request is forwarded to the disk handler. We model each read request by the pair (p, d),

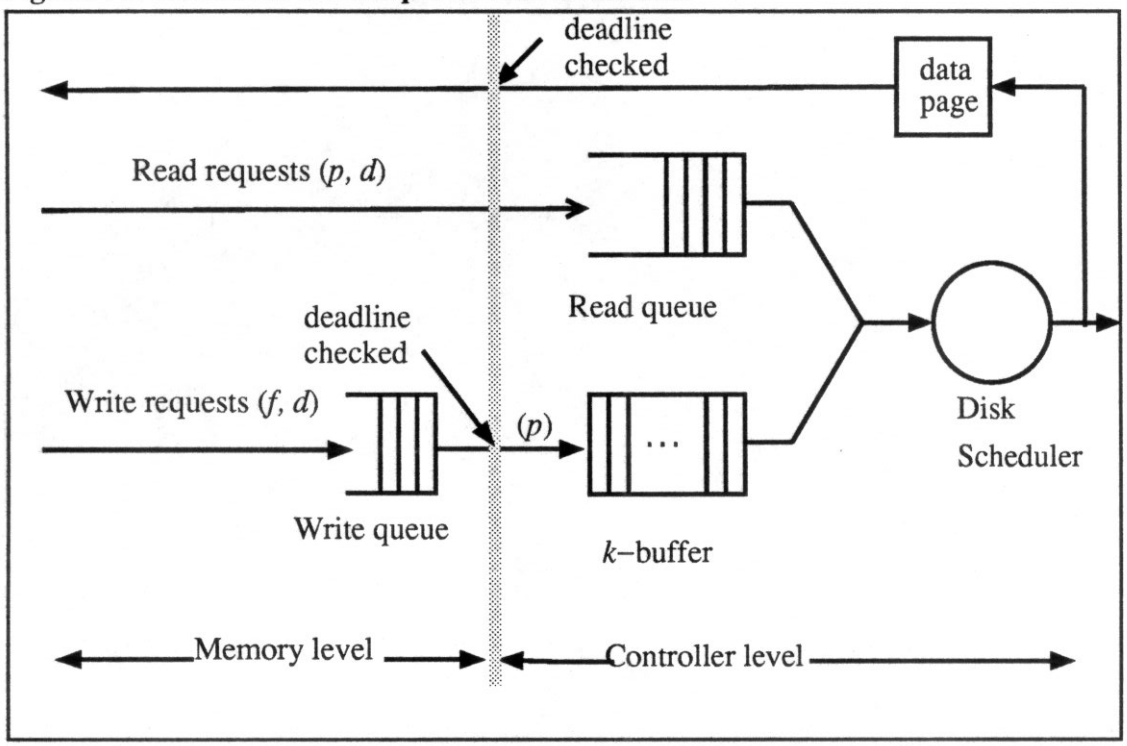
where p is the required page, and d is the deadline. The deadline d is determined by the buffer manager based on the deadline of the transaction that issued the read. Each request also identifies the page in the buffer that is to hold the newly read page.

Thus read requests are issued by the buffer manager, serviced by the disk handler, and processed by the disk. The data becomes available to the requesting transaction only when the page has been read into the memory buffer. So the deadline for a read request sets a time limit for the round trip from issuance, through the controller level read queue and back to main memory. The path of read requests is shown in Figure 4-1. Note that the deadline is checked at the border between memory and the I/O system (disk handler) on the way back from the disk.

When the buffer manager needs to clear buffer space (or needs to force writes to disk), it flushes dirty buffer pages. This generates write requests for the disk handler. We do not model the particular page replacement or force policy used by the buffer manager itself. Rather, we assume that write requests are generated at an average rate

λ_w. A write request (f, d) means the buffer manager needs to free frame f by time d. The contents of frame f is a modified data page p.

Figure 4-1 Model for I/O Requests with Deadlines



There are a several ways to model how a modified page is written out to disk. For example, page p can remain in memory while a write request is issued to the I/O system. The write request would eventually be scheduled at the disk and the data page is pulled from main memory and written on the disk. This model is an analog of the read request model. The buffer frame is not actually emptied until the write is performed at the disk. The deadline is a time constraint on the movement of data from the memory to this disk.

A serious flaw with this method is that a deadline is associated with an individual write request from the time it is issued until it is serviced at the disk. Unlike read requests, there is no natural way to assign these individual deadlines since write requests

are not issued by real-time transactions. The buffer manager is concerned only with freeing buffer frames; it does not care about when specific pages are written to disk.

A second way to model how a buffer page makes its way to disk is shown in Figure 4-1. Under this model the modified page p is first copied into a frame in a fixed size buffer pool. We call this buffer the *k-buffer* where k is the number of available frames. This buffer is managed by the disk handler and is separate from the buffer used by the buffer manager. The k-buffer can be thought of as a buffer at the disk controller, or as a buffer at a lower level of the operating system than the buffer directly seen by transactions. (The latter is a common arrangement, where the database management system manages its own buffer for transactions, and the operating system underneath manages the k-buffer.)

Once a page is copied into the k-buffer, the frame f is free. Thus, the deadline is satisfied if this copy is done before time d. The copy can be done immediately so long as there is an empty place in the k-buffer. If there is not, then the page must wait in the frame f until a place becomes available. Note that in this case the deadline applies only to the buffer to buffer copy operation, and not to the eventual disk write. Section 4.3 discusses how we create time constraints for emptying the k-buffer so that copy operation deadlines are met.

The model we have presented is relatively simple; yet we think it captures the essential aspects of the disk handler and buffer manager interactions. A more complete model could be postulated, one that included the details of the buffer manager, CPU scheduling, and the transactions themselves. Such a model would let us study the performance of the overall system. However, since our model uncouples the disk handler

from the rest of the system and reduces the number of irrelevant parameters, we believe it is better suited to studying the finer points of disk scheduling.

One last observation we make concerns the the use of deadlines at the level of individual I/O requests. Clearly, the top level transactions have deadlines. But how are these deadlines translated into deadlines for the individual read requests made by the buffer manager? Would it be better to use fixed priorities for the read requests, as suggested by [**CJL1**]?

Regarding the first question, there are many ways to translate deadlines. The simplest is to give the I/O request the same deadline as the issuing transaction. A more sophisticated approach would account for future requests to be issued by the transaction as well. For example, if it is known that a transaction typically issues 10 read requests, the first one would get a tenth of the remaining time to the final deadline, and so on. But if no access pattern information is available, the simple approach is probably the best. That is, in this case the I/O requests should be prioritized by the deadline of the issuing transaction. Although the issue of assigning deadlines is important, we do not discuss it further in this thesis.

Regarding the second question, a fixed priority scheme is an alternative to deadlines. Fixed priorities are natural in some applications, but in applications where deadlines exist, we believe it is more natural to work with deadlines and to carry them through to lower level tasks like I/O requests. Furthermore, deadlines give us more flexibility: 1) we are not constrained to a fixed set of priorities, 2) we can tell when a request is infeasible and can be ignored, and 3) they may be more useful in a distributed system where the priorities of one computer may be different from those of others.

4.3 Managing the *k*-Buffer

Our model contains two types of I/O requests: those with deadlines and those without. The normal mode of operation is to give priority to servicing the requests with deadlines. The requests without deadlines are serviced either when there are no read requests or when there is little space left to buffer the write requests and the buffer is in danger of overflowing. This section discusses two heuristic techniques that can be used to trigger service for write requests. These policies provide a way to decide if the next I/O request to service will be a read or a write. They do not choose which particular request to service. This decision is left to the scheduling policies in Section 4.4.

4.3.1 Space Threshold

The motivation of the Space Threshold policy is to maintain a minimum amount of free space in the write buffer at all times, so that each arriving write request will find an open slot in the buffer. The amount of free space that is maintained is a parameter that can be tuned to accommodate the expected "burstiness" of the write arrival pattern. A space threshold heuristic is applied as follows:

Figure 4-2 Space Threshold

IF Free Space < Threshold **OR** there are no read requests

THEN Service write requests.

ELSE Service read requests

Preferential service is granted to read requests so long as the space threshold has not been exceeded. Otherwise, write requests are serviced. Write requests are also serviced when there are no read requests in the system.

4.3.2 Time Threshold

The motivation of the Time Threshold technique is to create an artificial deadline for a write event. We use the term "write event" to denote the action of writing the contents of a buffer slot to disk. Note that a write event, and hence the write event deadline, is not associated with a particular write request. It is unimportant which request is serviced as long as the buffer as a whole does not overflow. The closeness of the write event deadline reflects the urgency of emptying a buffer slot. If the buffer is relatively empty then the deadline would be far off. If the buffer is nearly full then the deadline would be much sooner. Let D_W be the deadline for the write event. The Time Threshold approach is applied as follows:

Figure 4-3 Time Threshold

IF D_W < The earliest read deadline **OR** there are no read requests

THEN Service a write request

ELSE Service read requests

Again, the method for choosing which read or write request to service is decided by one of the algorithms in Section 4.4.

In contrast to the Space Threshold approach, Time Threshold always gives consideration to the timing requirements of read requests, regardless of how full the write buffer is. However, when the buffer is almost full, it is unlikely that any read request will have an earlier deadline than D_W. The problem now is to assign appropriate deadlines for write events. We chose to adopt a simple linear function which uses the state of the write buffer (amount of free space) and an estimate of write arrival patterns in order to create write event deadlines.

4.3.2.1 Linear Function for Setting D_W

Using this approach, the closeness of the deadline varies linearly with the amount of free space in the write buffer. The average inter-arrival time of write requests is the slope. Thus $D_W = t + (1 / Write\_Rate) * (\text{free space} + 1)$. *Write_Rate* is the average arrival rate of write requests and t is the current time. Thus if a write request arrives and fills the last slot in the buffer, i.e., free space = 0, then D_W is set to (1 / *Write_Rate*) seconds from the current time. If a buffer can be emptied before the next write arrival is expected then the buffer will not overflow.

Other functions to set D_W are possible. For example, one could choose a function that varies D_W quadratically with the amount of available free space. Another option is to set an acceptably small probability for not overflowing the write buffer and work backward from the Poisson probability functions to solve for the expected time at which the overflow would occur. However these methods seem unnecessarily complicated and we found (see Section 5.2) that the Space Threshold and simple linear function for the Time Threshold technique worked well in practice.

4.4 Disk Scheduling Algorithms

This section presents a number of algorithms for scheduling I/O requests. The first three are traditional disk scheduling algorithms that have been studied before, although not in the context of real-time performance metrics. This group of scheduling policies does not use deadline information to make scheduling decisions. Seek optimization, if done at all, is done with the use of track information. The algorithms that we have chosen are generally acknowledged to perform well under one or more of the traditional non real-time performance metrics: average response time, throughput, fairness to requests. Thus their performance can be used as a reference against which the real-time

scheduling algorithms can be compared. The second group of three are new algorithms that are specifically designed for the real-time environment. These algorithms do make use of deadline information.

In the following subsections we explain each scheduling policy and illustrate how it can be combined with the policies for managing the k-buffer (Section 4.3) in order to create a complete algorithm. The set of requests which the algorithm is trying to schedule is composed of the read requests, each a pair (p, d), and the writes, each simply (p). We use p to denote the track number where the page is located on disk, and d is the deadline for read requests. If the Space Threshold technique is used for managing the k-buffer then F denotes the free space threshold. If the Time Threshold technique is employed, then D_w denotes the write event deadline.

4.5 Three Traditional Algorithms

We chose to study three traditional disk scheduling algorithms under real-time performance metrics. They are First-Come-First-Served (FCFS), Shortest-Seek-Time-First (SSTF), and the elevator algorithm SCAN. The performance of these algorithms under non real-time metrics has been studied extensively [**CH**, **CKR**, **GD**, **TP**, **Wi**].

The traditional algorithms schedule from a single pool of requests. There is no distinction between read and write requests. Also these scheduling models do not use a buffer pool to buffer write requests. Applied as is to our model, the algorithms do not use a special policy for managing the k-buffer.

Although these algorithms do not require the use of a buffer management technique they can be modified to make better use of the k-buffer by combining them with

the Space Threshold technique. For example, combining FCFS with Space Threshold produces the following algorithm:

Figure 4-4 FCFS Scheduling with Space Threshold

> **IF** Buffer free space $< F$ **OR** there are no read requests
>
> **THEN** Service the closest write request.
>
> **ELSE** Service the read request with the earliest arrival time.

The FCFS scheduling policy is applied only to read requests. When the k-buffer is nearly full, and in danger of overflowing, the algorithm will service the closest write request thereby emptying the k-buffer as quickly as possible. Notice that write requests are only serviced when the free space threshold has been exceeded or when there are no read requests to service.

4.6 Three Real-Time Scheduling Algorithms

4.6.1 Earliest Deadline First (ED)

The Earliest Deadline algorithm is an analog of FCFS. read requests are ordered according to deadline and the request with the earliest deadline is serviced first. Since no positional information is used to make scheduling decisions, ED will have the same expected seek time profile as FCFS. Unlike the first three algorithms, it is necessary to use ED with one of the k-buffer management techniques. This is because ED uses only deadline information to make scheduling decisions. Since write requests do not have deadlines, one of the k-buffer management techniques must be employed to guarantee that the k-buffer is emptied.

Combining ED with the Space Threshold technique for managing the k-buffer produces the following algorithm:

Figure 4-5 Earliest Deadline Scheduling with Space Threshold

IF Buffer free space $< F$ **OR** there are no read requests

THEN Service the closest write request

ELSE Service the read request with the earliest deadline.

Combining ED with a Time Threshold technique for managing the k-buffer produces the following algorithm: (Recall that D_W denotes the write event deadline.)

Figure 4-6 Earliest Deadline Scheduling with Time Threshold

IF $D_W <$ the earliest read deadline **OR** there are no read requests

THEN Service the closest write request

ELSE Service the read request with the earliest deadline

4.6.2 Earliest Deadline SCAN (D-SCAN)

This algorithm is a modification of the traditional SCAN algorithm. In D-SCAN the track location of the read request with the earliest deadline is used to determine the scan direction. The head seeks in the direction of the read request with the earliest deadline servicing all read requests along the way (these will be for requests with later deadlines) until it reaches the target track. After the target request is serviced, the scan direction is updated towards the direction of the read request with the next earliest deadline. We have chosen to use earliest deadline to select the scan direction, however any real-time priority scheme could be used, e.g., least slack.

Unlike the traditional SCAN algorithm the new scan direction may be the same as the previous scan direction. Also, the scan direction can change before the target

track is reached. This will happen if a request with an earlier deadline than the target request, and a track location behind the current head position arrives after the scan direction is chosen. Also D-SCAN does not scan to the last request in a particular direction. If the requests with the earliest deadlines are grouped within a small region, then D-SCAN will scan only in that region. However as time progresses, the deadlines of requests at other portions of the disk will be the earliest and the head will scan over those portions of the disk.

Like ED, D-SCAN must be combined with one of the *k*-buffer management techniques to ensure that write requests are serviced. Using the Time Threshold technique produces the following algorithm:

Figure 4-7 D-SCAN Scheduling with Time Threshold

IF D_W < the earliest read deadline **OR** there are no read requests

THEN Service the closest write request

ELSE Service the closest read request in the scan direction.

4.6.3 Feasible Deadline Scan (FD-SCAN)

The FD-SCAN algorithm is similar to D-SCAN except that only read requests with feasible deadlines are chosen as targets that determine the scanning direction. A deadline is feasible if we estimate that it can be met. More specifically, a request that is n tracks away from the current head position has a feasible deadline d if $d \geq t + Access(n)$ where t is the current time and $Access(n)$ is a function that yields the expected time needed to service a request n tracks away.

Each time that a scheduling decision is made, the read requests are examined to determine which have feasible deadlines given the current head position. The request

with the earliest feasible deadline is the target and determines the scanning direction. The head scans toward the target servicing read requests along the way. These requests either have deadlines later than the target request or have infeasible deadlines, ones that cannot be met. If there is no read request with a feasible deadline, then FD-SCAN simply services the closest read request. Since all request deadlines have been (or will be) missed, the order of service is no longer important for meeting deadlines and SSTF is used to efficiently service the outstanding requests as quickly as possible.

Like ED, and D-SCAN, FD-SCAN must be combined with a buffer management policy to ensure that write requests are serviced. Combining FD-SCAN with the Time Threshold technique produces an algorithm similar to that produced by D-SCAN.

4.7 A Brief Note on Complexity

The algorithms we have proposed are heuristics and we do not claim that they are optimal. An interesting question is what is an optimal algorithm for scheduling I/O requests with deadlines? For the on-line scheduling problem, clearly no optimal algorithm exists since we can use an adversarial strategy to issue requests that would defeat any particular algorithm.

Consider an off-line version of the problem: at time 0 there are N I/O requests. Each needs to access a particular track before a deadline d. Let $C(i)$ be a cost function that expresses the amount of time needed to access a request i tracks away from the current head position. The question is: can the requests be sequenced so that each request meets its deadline? Not surprisingly, the difficulty in answering this question depends on the cost function $C()$. If the cost function is linear in i then a dynamic programming solution is optimal. For non-linear cost functions (the more realistic case for rotational

disk devices) the existence of an optimal algorithm for the off-line case is an open problem [**PSMK**].

We believe this justifies the use of simulation for evaluating the performance of the proposed algorithms.

Chapter 5 Evaluation by Simulation

5.1 Simulation Model

To test the algorithms, we built a program to model real-time I/O requests in a system with a single data disk. We make the realistic assumption that log writes are directed toward a separate device. Our program was built using CSIM, a process-oriented simulation language [**Sc**].

5.1.1 Device Model.

The program models a single-head disk device at the track level; we do not model sectors within tracks. This is reasonable since none of the algorithms under study performs rotational optimization. The names and meaning of the four parameters that control the I/O system configuration are shown in Table 5-1.

Table 5-1 Device Parameters

Parameter	Meaning	Base Value
Tracks	# of tracks on disk	1000
BufferSize	# of pages in *k*-buffer	10
SeekFactor	Seek time scaling factor	0.6 ms
DiskConstant	Rotational latency plus transfer	15 ms.

The base values are not meant to represent a particular workload but were selected as reasonable values within a range. Also our experiments vary the values of the parameters to learn how the algorithms perform under different workload characteristics.

The access time for an I/O request n tracks away from the current head position is expressed by the equation:

$$Access(n) = Seek(n) + Rotational\ latency + Transer\ time.$$

In our model, rotational latency and transfer time are grouped together in the single parameter *DiskConstant*. In today's disk technology, seek times are non-linear with seek distance[**BG, SCO**]. Accordingly, we use the function

$$Access(n) = Seek(n) + Rotational\ latenc$$

to model the time to seek n tracks. Thus the complete access time is given by the equation:

$$Access(n) = DiskFactor \times \sqrt{n} + DiskConstant$$

Using the values from Table 5-1 and assuming an average seek distance of 333 tracks, the average access time for our modeled device is 26 ms. The average access time will be used in the construction of individual request deadlines.

5.1.2 Workload Model

The names and meanings of the parameters that control the workload characteristics are shown in Table 5-2.

Table 5-2 Workload Parameters

Parameter	Meaning	Base Value
Read_Rate	Read arrival rate	
Write_Rate	Write arrival rate	
Min_Slack	Minimum slack time (for reads)	10 ms
Max_Slack	Maximum slack time (for reads)	100 ms.

Requests for I/O service arrive from an open source with exponentially distributed inter-arrival times with mean arrival rates denoted by *Read_Rate* and *Write_Rate* for read and write requests respectively. Each request needs to access a single track which is chosen uniformly from the range request is placed in the *k*-buffer. The recording of write missed deadlines (buffer overflows) and the setting of the write event deadline are done as explained in Sections 4.2 and 4.3. If the request is a read, then it is placed in a queue and its deadline is chosen as follows. A slack time is chosen uniformly from the range [*Min_Slack*, *Max_Slack*]. The equation for computing the deadline is:

$$\textit{Deadline} = \textit{Arrival time} + \textit{Average access time} + \textit{Slack time}$$

In reality, deadlines can be unreasonable, or impossible to meet. In our experiments we want to avoid scenarios where deadlines are either, (1) so slack that any scheduling algorithm will meet them, or (2) so tight that no algorithm could meet them. Thus we tried to choose deadlines that leave some room for "intelligent" scheduling. Our experiments will then show which algorithms have this "intelligence."

5.1.3 Data Generation and Metrics

In the following sections we discuss some of the results from the experiments that we performed. Due to space considerations we cannot present all of our results but have selected the graphs which best illustrate the differences and performance of the al-

gorithms. For each experiment we ran the simulation using 40 different random number seeds. Each run continued until 3000 I/O requests were processed. Numerous performance statistics were collected and averaged over the 40 runs. It is not necessary to throw away the early statistics during a "warm-up" period. The initial state of each run - empty queues and a randomly chosen head position - is equivalent to the system state at any other time that the system is idle. Since our parameters exercise the system within its throughput capacity for most of the algorithms, idle periods during a run do occur.

The primary metric that we use to measure performance is percentage of missed deadlines. Recall that our goal is to schedule I/O requests so that they meet their *individual* response time goals, or deadlines. Measuring the percentage of requests that miss their deadlines is a good performance metric for this goal. Since our model contains two types of deadlines, we measure the performance of each separately using the following equations:

$$\%\textit{Missed Read Deadlines} = \frac{\textit{Missed Read deadlines}}{\textit{Number of Reads serviced}} \times 100$$

$$\%\textit{Missed Write Deadlines} = \frac{\textit{Missed Write deadlines}}{\textit{Number of Write arrivals}} \times 100$$

We also collected statistics on response times, seek times, seek distances and queue lengths for both read and write requests. The mean values of these metrics are presented in the graphs that follow. The graphs also plot 95% confidence intervals as error bars around each data point. These intervals are so small however, that they are rarely visible outside the bullet that is used to mark a data point.

5.2 Simulation Results

This section presents results for three different sets of experiments. In the first set, only requests with explicit deadlines are scheduled. We will continue to refer to requests with deadlines as reads but they could in fact be writes with explicit deadlines. In the second set of experiments we add the *k*-buffer and examine scheduling of both read requests with explicit deadlines and write requests without. In the last, we simulate a periodic checkpointing task that issues a fixed number of write requests all having the same deadline. The checkpoint generated requests are scheduled concurrently with the regular stream of read requests. This experiment is done without the *k*-buffer.

5.3 Read Requests Only

In this set of experiments we studied performance behavior under a workload that contained only I/O requests with deadlines. Deadlines were chosen according to the method described in Section <IO simulation model>.

5.3.1 Experiment 1: *Min_Slack* = 50, *Max_Slack* = 50

In the first experiment, the *Min_Slack* and *Max_Slack* parameters were both set to 50 ms. Thus each request had a deadline that was approximately 76 ms from its arrival time (average access plus 50 ms). This also guarantees that arriving requests have later deadlines than requests already in the queue. The parameter *Read_Rate* was varied from 22 requests per second to 40 requests per seconds in increments of 2. Figure 5-1 graphs %Missed Read Deadlines for all six scheduling algorithms. Because of the way that deadlines are assigned, FCFS and ED have exactly the same behavior. The data for these two algorithms is omitted for arrival rates greater than 36 since these two algorithms are unable to meet the throughput demands due to inefficient use of the disk.

These parameter settings describe a very difficult workload where many deadlines are missed. One could argue that such a scenario is unrealistic. However, we believe that for designing real-time schedulers, once must look at precisely these high-load situations. Even though they may arise infrequently, one would like to have a system that misses as few deadlines as possible when these peaks occur.

The results show that FD-SCAN consistently has the best performance across all load settings. At the highest setting, FD-SCAN misses approximately 6.5% fewer deadlines that SSTF, the second best algorithm. This represents a performance improvement of about 15%. The SCAN and D-SCAN algorithms are the next best with SCAN being slightly better.

The SSTF and SCAN algorithms work well because they move the disk head efficiently and thus use the disk resource efficiently. The mean seek distance, and thus mean response time, is significantly lower for these two algorithms, Figure 5-2. Importantly, the mean seek distance decreases significantly as the load increases. This is not surprising since it is exactly how these algorithms were designed to work. The SSTF and SCAN algorithms are excellent baseline algorithms to try to beat precisely because they use the disk resource efficiently. However, they schedule the requests randomly with respect to deadlines. Our goal is to learn which algorithms can beat SSTF and SCAN by doing intelligent deadline scheduling, and yet still use the disk efficiently

In contrast to SSTF and SCAN, FCFS and ED move the disk arm very inefficiently, performing a random seek for every request. As the load increases, these two algorithms are unable to maintain throughput.

The two real-time algorithms D-SCAN and FD-SCAN try to do intelligent deadline scheduling and move the disk arm efficiently. In this experiment, Figure 5-1, we see

that D-SCAN performs similarly to SCAN. Because of the way deadlines are assigned, D-SCAN scans the disk in much the same way as SCAN. First, recall that ordering requests by deadline is equivalent to ordering by arrival time. Consider the disk head at some point during a scan. The requests behind the head (in the wake of the scan) will be recent arrivals. The requests lying in front of the head will contain some recent arrivals but more older requests as well. The scan direction will not change until these older requests are serviced. Once they are, the oldest requests are now at the other end of the disk and the scan reverses direction. Because a newly arrived request cannot change the direction of scan, D-SCAN operates much like SCAN.

The FD-SCAN algorithm has the best performance because it gives high priority to requests with feasible deadlines. Thus FD-SCAN may change direction to service a recent arrival if it determines that the older requests lying before it have infeasible deadlines. When all requests have infeasible deadlines, FD-SCAN defaults to SSTF which is an efficient way to service the requests and empty the queue as rapidly as possible.

5.3.2 Experiment 2: *Min_Slack* = 10, *Max_Slack* = 50

In the second experiment we set *Min_Slack* = 10 ms. and *Max_Slack* = 50 ms. This allows for a greater range of deadlines than the first experiment, but deadlines are still relatively tight. As before *Read_Rate* was varied from 22 to 40 requests per second in increments of 2. Figure 5-3 graphs %Missed Read Deadlines for the six algorithms. As before, FD-SCAN consistently has the best performance, especially at the higher load settings.

Now that deadlines are not assigned uniformly, ED and FCFS will have different behavior. In fact, ED performs slightly better than FCFS at the lowest loads. However, ED rapidly loses its performance edge as the load increases. This occurs because ED

seeks inefficiently and also because it always gives the highest priority to the request with the earliest deadline. When the load is high, and the throughput rate is slipping, it is unlikely that this deadline can be met. By servicing it with high priority, ED delays the completion of requests with possibly feasible deadlines.

5.3.3 Experiment 3: *Min_Slack* = 10, *Max_Slack* = 100

In a third experiment we set *Min_Slack* = 10 ms and *Max_Slack* = 100 ms. This allows an even greater range of deadlines. The load was varied as it was in the first two experiments. Figure 5-4 graphs %Missed Read Deadlines for the six algorithms. This time we see that ED misses the fewest deadlines when the arrival rate is low. At this rate, all algorithms will have high mean seek distances because mean queue length is small. Therefore ED is not handicapped, relative to the other algorithms, by its high mean seek time. However, as the load exceeds 32 requests per second, performance deteriorates swiftly due to inefficient use of the disk.

Another interesting observation is that D-SCAN performs better than SSTF and SCAN when the load is low and just as well when the load is high. The variability in deadlines allows D-SCAN to make intelligent choices and still use use the disk efficiently. At the higher load settings, FD-SCAN performs better than all other algorithms.

In Figure 5-5 we graph the Mean Tardy Time for all six algorithms. (Only requests that miss their deadlines contribute to this metric.) The performance of ED is interesting since it has the lowest Mean Tardy time when the load is low, but a very high Mean Tardy time when the load is high. This is characteristic for ED; it can meet most deadlines when the load is low but it rapidly saturates and misses most deadlines, by a lot, when the load is high. Similar remarks apply to FCFS, although it does not perform as well as ED.

Algorithms SSTF, SCAN and D-SCAN have similar performance across the range of loads. These algorithms have low Mean Tardy times because 1) they do not miss many deadlines, (see Figure 5-4), and 2) they have low response time variances, Figure 5-6. These three policies tend to limit the maximum response time experienced by a request. The SCAN policy does this because it always sweeps to the edge of the disk (or as far as the outermost request). Thus requests may miss their deadlines but they are guaranteed to be serviced during the next scanning sweep. The D-SCAN algorithm also bounds response time with its scanning behavior. Perhaps more important is the fact that eventually, a request that remains unserviced will become the highest priority request and thus become the target of the scan. Finally, SSTF bounds response times because of the uniform distribution of requests along the disk surface. Because requests are not clustered, the head does not get "stuck" in one area of the disk.

Interestingly, FD-SCAN has the highest Mean Tardy time, Figure 5-5. Although FD-SCAN misses the fewest deadlines (see Figure 5-4), it misses them by a greater amount. The Mean Tardy time is greater because requests with infeasible deadlines can wait for a long time before they are serviced. In fact, requests with infeasible deadlines are only serviced when they lie between the disk head and the current scanning target, whenever a request with a feasible deadline exists. Thus a request, A could lie only one track from the current head position but not be serviced because there is no request B with a feasible deadline to "pull" the head over request A. Contrast this to SSTF which would service A because it is so close.

The preceding discussion suggests that we examine how the algorithms perform for certain areas of the disk. Figure 5-7 graphs the distribution of missed deadlines for 10 disk areas at *Read_Rate* = 36 requests per second. Disk area 1 contains tracks 1-100,

area 2, 101-200, and so on. The FCFS and ED algorithms are very fair, each area accounts for 10% of the total number of missed deadlines. The SCAN algorithm is also relatively fair. The FD-SCAN algorithm is remarkably unfair to requests in the two outermost disk areas. A center area of the disk accounts for only 6% of the missed deadlines while each outermost area accounts for more than 16%.

Since requests on the outer tracks require longer seeks, they are more likely to miss their deadlines. The SSTF, SCAN, D-SCAN and FD-SCAN policies all favor the center tracks. However, SCAN regularly services the outer tracks. Requests that arrive while the head is there will be serviced quickly and meet their deadlines. The D-SCAN policy only services the outer areas when the earliest deadline request is located there. The FD-SCAN policy discriminates against the outermost areas severely, scanning them only when a request with a *feasible* deadline is there. However it is very bad only to the outermost areas. The missed deadline distribution for the next to outermost areas is comparable to SSTF, SCAN and D-SCAN.

Finally, we plot the Mean Queue Length for the algorithms in Figure 5-8. Both ED and FCFS suffer throughput problems near an arrival rate of 36 requests per second and this shows in the rapid increase in average queue length. The other two real-time algorithms, FD-SCAN and D-SCAN, also have longer average queues than SSTF and SCAN. However, the mean queue lengths are nearly always less than four requests. Thus the load settings are high but not unreasonably so.

5.4 Read and Write Requests

In this set of experiments, we studied performance behavior under a workload that contained both read and write I/O requests. We are interested in comparing the per-

formance of the algorithms that do not manage the *k*-buffer to those that do manage the buffer. Also we want to learn which algorithms are best overall.

5.4.1 Experiment 1: Vary *Read_Rate*

In the first experiment *Write_Rate* was set at 10 requests per second while *Read_Rate* was varied from 12 to 30 requests per second in increments of 2. Note that the cumulative arrival rate is the same as in the first set of experiments. The slack parameters had the base values shown in Table 5-2. Figure 5-9 graphs %Missed Read Deadlines for both the traditional and the buffer adapted versions of SSTF and SCAN. The buffer adapted versions use the Space Threshold technique with the threshold set to 1. It is obvious that the buffer adapted versions miss far fewer read deadlines than the traditional versions. Figure 5-10 graphs %Missed Write Deadlines for the same experiment. Note that although the buffer adapted versions do miss some write deadlines, it is less than 2.5% of the total write arrivals. The traditional versions never let the buffer overflow.

Figure 5-11 shows %Missed Read Deadlines for the three real-time algorithms ED, D-SCAN and FD-SCAN, and the three buffer adapted traditional algorithms. The real-time algorithms use the Time Threshold technique for buffer management. The others use the Space Threshold technique. (We do not include the Space Threshold versions of the real-time algorithms because the resulting graph would be too crowded. Also, we found that, for this experiment, the Time Threshold version performed slightly better than the Space Threshold version.) Although ED performs well at lower load settings, its performance quickly deteriorates once *Read_Rate* exceeds 22 requests per second. At the highest rate, ED is no better than FCFS(B). The D-SCAN and SCAN(B) algorithms have nearly identical performance. The second best performer is SSTF(B) and

the best is FD-SCAN. At *Read_Rate* = 30, FD-SCAN misses approximately 3% fewer read deadlines than SSTF(B). This represents an improvement of 12%.

Figure 5-12 shows %Missed Write Deadlines for the same six algorithms. The buffer adapted algorithms SSTF(B) and SCAN(B) permit significantly buffer overflows than D-SCAN or FD-SCAN at the two highest load settings. This happens because D-SCAN and FD-SCAN use the Time Threshold technique for managing the *k*-buffer. (We are not saying that all Time Threshold techniques would behave like this, perhaps some would be better. However, the one that we tested, namely the linear technique, exhibits this performance.) When Read_Rate is high there is a greater chance that a read request will have an earlier deadline than D_w, the write event deadline. Thus reads receive priority and more write deadlines are missed. Recall that FD-SCAN missed the fewest read deadlines, Figure 5-11. When the two measures (missed reads and missed writes) are combined in a weighted average, SSTF(B) and FD-SCAN have nearly identical performance.

5.4.2 Experiment 2: Vary *Write_Rate*

In a second experiment, *Read_Rate* was fixed at 12 requests per second. and *Write_Rate* was varied from 10 to 28 requests per second. The other parameters were unchanged. Figure 5-13 graphs the weighted average of %Missed Deadlines for both reads and writes. Note that all the algorithms that make effective use of the write buffer performs better than the traditional versions of SSTF and SCAN. One reason for the lack of differentiation among the better performing algorithms is that they all are missing very few deadlines anyway. It is easy to meet deadlines because the request stream consists mostly of write requests, and meeting write deadlines (keeping the buffer from overflowing) is much easier to do than meeting the individual read deadlines. However,

when *Write_Rate* is large enough, buffer overflows become more of a problem. In this experiment, %Missed Write Deadlines increases from less than 1% at 22 writes per second to roughly 6% at 28 requests per second for the algorithms that manage the *k*-buffer. This accounts for most of the increase in %Missed Deadlines in Figure 5-13.

5.4.3 Experiment 3: Vary *Buffer_Size*

In a third experiment, *Read_Rate* was fixed at 20 requests per second, *Write_Rate* at 16, and *Buffer_Size* was varied from 5 to 14. %Missed Read Deadlines versus *Buffer_Size* is shown in Figure 5-14. First, note that SSTF and SCAN do not change. These algorithms do not manage the *k*-buffer, thus changing the buffer size will not affect %Missed Read Deadlines. Second, all of the algorithms that manage the *k*-buffer perform much better than SSTF and SCAN. These algorithms can delay servicing writes in order to meet the deadlines for reads. The amount that they can delay, and thus the effective advantage to read scheduling, increases significantly as the buffer size increases. Finally, we note that FD-SCAN performs the best with SSTF(B) and D-SCAN tied for second. Also, no algorithms missed any write deadlines for any of the buffer size settings.

5.4.4 Experiment 4: Vary *Space_Threshold*

In a fourth experiment, *Read_Rate* was fixed at 16 requests per second, *Write_Rate* at 20, *Buffer_Size* at 10, and the *Space_Threshold* parameter was varied from 1 to 4. (Recall that *Space_Threshold* controls when the emptying of the *k*-buffer is triggered for those algorithms that use the Space Threshold technique for managing the *k*-buffer.) Thus, when *Space_Threshold* = 1, the emptying of the *k*-buffer is triggered only when the buffer is full (free space < 1). When *Space_Threshold* = 4, the emptying of the *k*-buffer is triggered when there are only three empty spaces left.

Figure 5-15 graphs %Missed Read Deadlines for the three traditional algorithms which use Space_Threshold. The FD-SCAN algorithm, which uses Time_Threshold, is also shown. Note that the graph for FD-SCAN is flat since it is not affected by the *Space_Threshold* parameter. The other three algorithms miss more read deadlines as the *Space_Threshold* parameter increases. Although we do not show the graph, these algorithms also miss *fewer* write deadlines. However when *Space_Threshold* = 3, these three algorithms miss less than 1 percent of write deadlines and increasing *Space_Threshold* further does not improve write performance any more. Moreover, the ability to make the best use of all of the *k*-buffer is hampered. Similar behavior was observed for algorithms using Time_Threshold when the function that sets the write deadline was varied from being optimistic to pessimistic.

5.5 A Checkpointing Experiment

So far we have assumed that pages modified by a transaction can be written to disk at any time after its commit. In some systems, however, it is required to periodically force out to disk all dirty pages in order to achieve a *checkpoint* [**Gr**]. At the checkpoint, the system writes in the log a special record. This record, together with the flushed pages, contain enough information so that a consistent database state (as of this point in time) can be reconstructed in case of failure.

We may model a checkpointing system by assuming that the time of the next checkpoint is known in advance. The writes issued by the buffer manager have this time as their deadline. Thus, writes issued long before the checkpoint time will not be urgent; writes issued close to the checkpoint will be more urgent. Read requests are serviced concurrently; their deadlines are independent of the checkpointing. When the checkpoint time arrives, the special log record is written, a new checkpoint time is selected

and the process repeats itself. What we have described here is a rough approximation to what really takes place, but we believe it is realistic enough to let us study our disk scheduling algorithms.

In a final experiment we implemented such a checkpointing model. The Poisson stream of read arrivals governed by *Read_Rate* and the two slack parameters, is unchanged. Two new parameters *Check_Period* and *Check_Size* were introduced to describe a periodic checkpointing process that issues *Check_Size* requests every *Check_Period* seconds. At the beginning of a checkpoint, *Check_Size* unique I/O requests are generated. Each of these requests has a deadline equal to the start of the next checkpoint period. The checkpoint requests "arrive" in the system at equally spaced intervals, one every *(Check_Period/Check_Size)* * 0.75 seconds. Thus all the checkpoint requests for a particular period are issued in the first 3/4 of that period. The process repeats when the next checkpoint time is reached. Note that the requests in this system all have explicit deadlines. No *k*-buffer is used; all requests are directed to a single queue.

We conducted an experiment where *Check_Period* = 30 sec. and *Check_Size* = 100 and *Read_Rate* varied from 26 to 40 requests per second. The slack parameters were as in Table 5-2. Figure 5-16 graphs %Missed Deadlines (both checkpoint and non-checkpoint requests) for the top four algorithms. The FD-SCAN algorithm clearly has the best performance. It is able to delay servicing the checkpoint requests in favor of read requests with more urgent deadlines. However, as the checkpoint deadline approaches, the checkpoint requests have the highest priority and they are serviced ahead of reads.

Figure 5-1 ***Min_Slack*** **= 50,** ***Max_Slack*** **= 50**

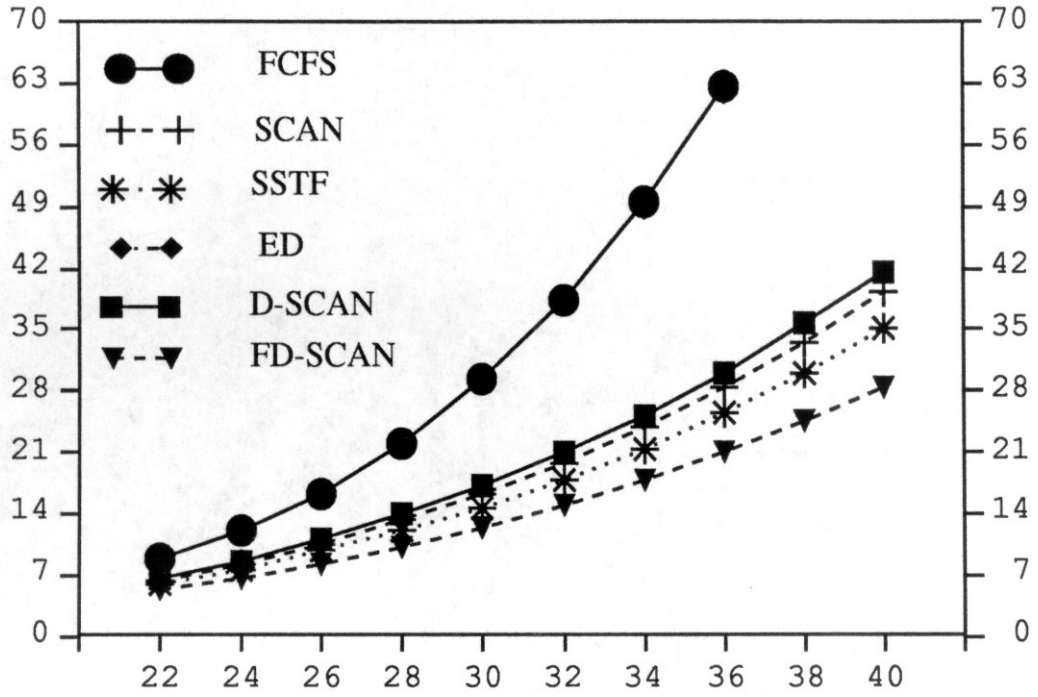


Figure 5-2 ***Min_Slack*** **= 50,** ***Max_Slack*** **= 50**

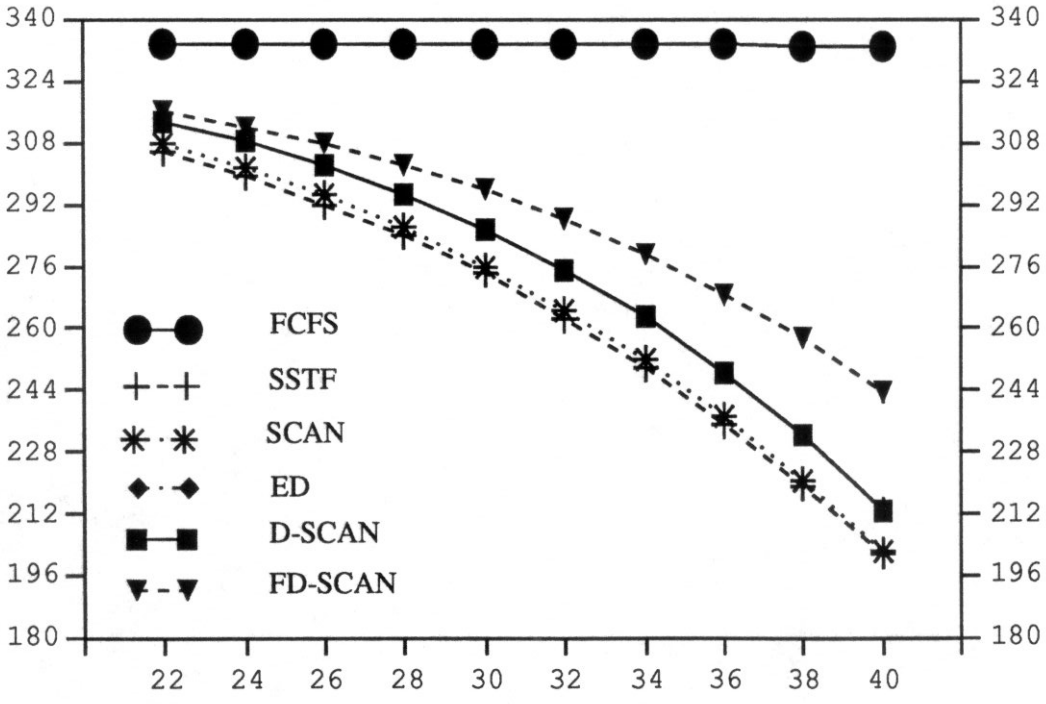


Figure 5-3 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 50**

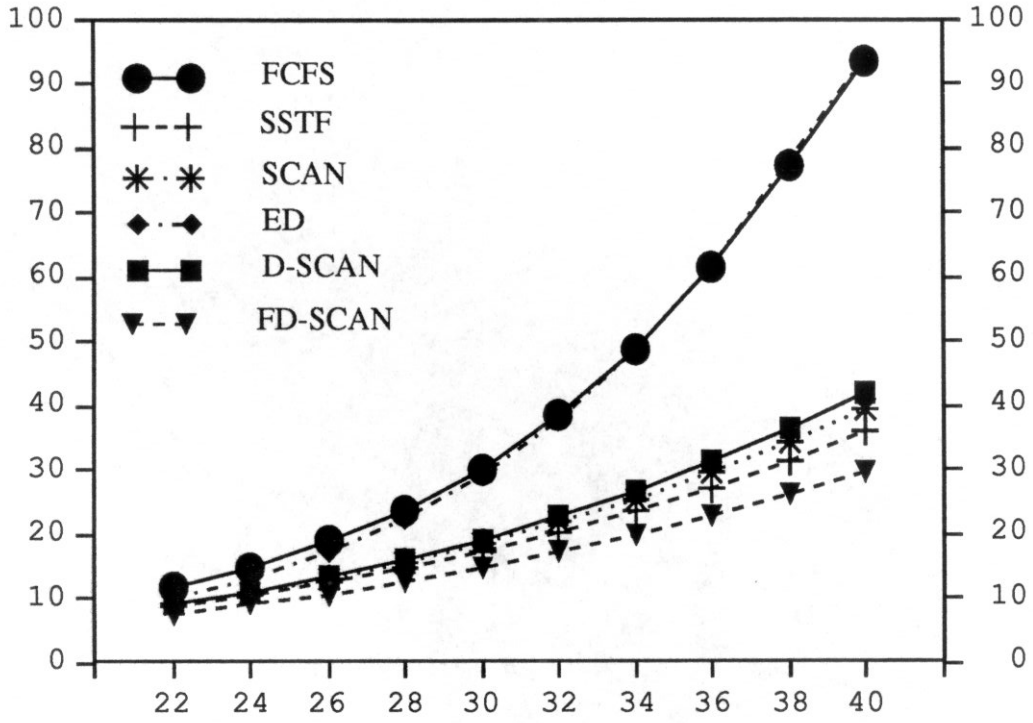


Figure 5-4 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 100**

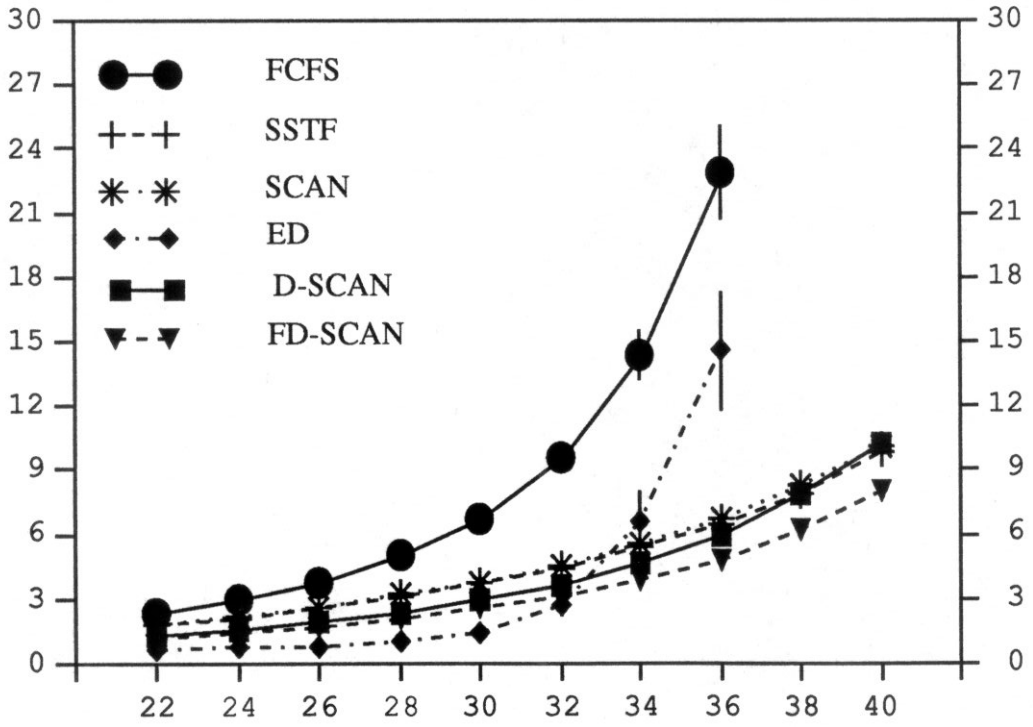


Figure 5-5 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 100**

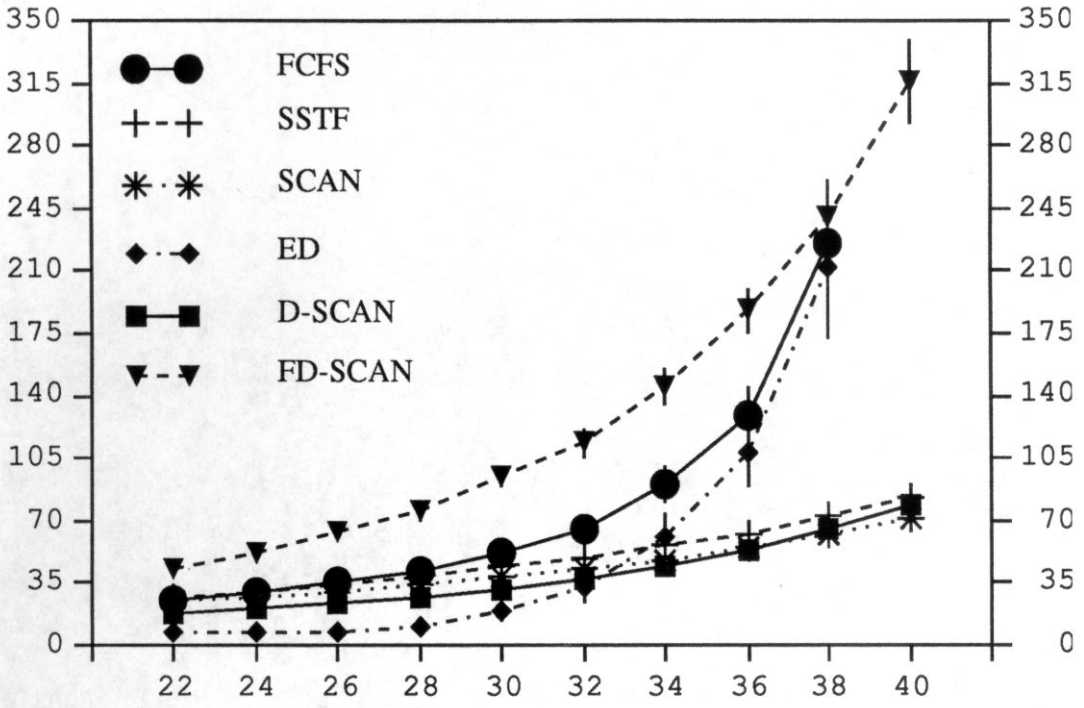


Figure 5-6 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 100**

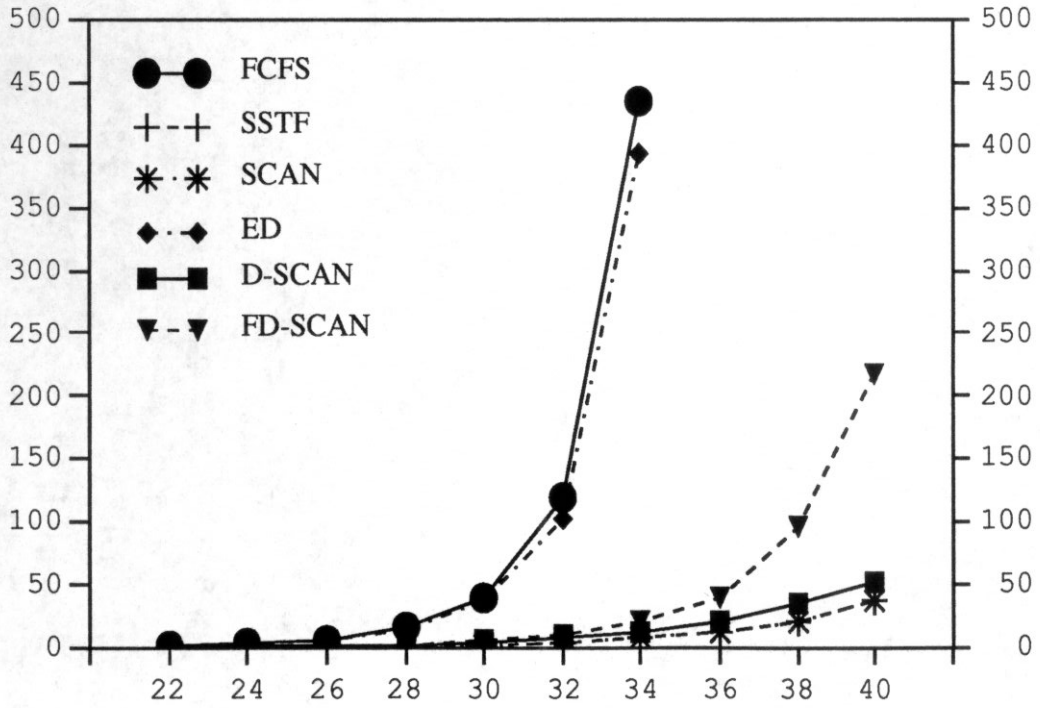


Figure 5-7 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 100**

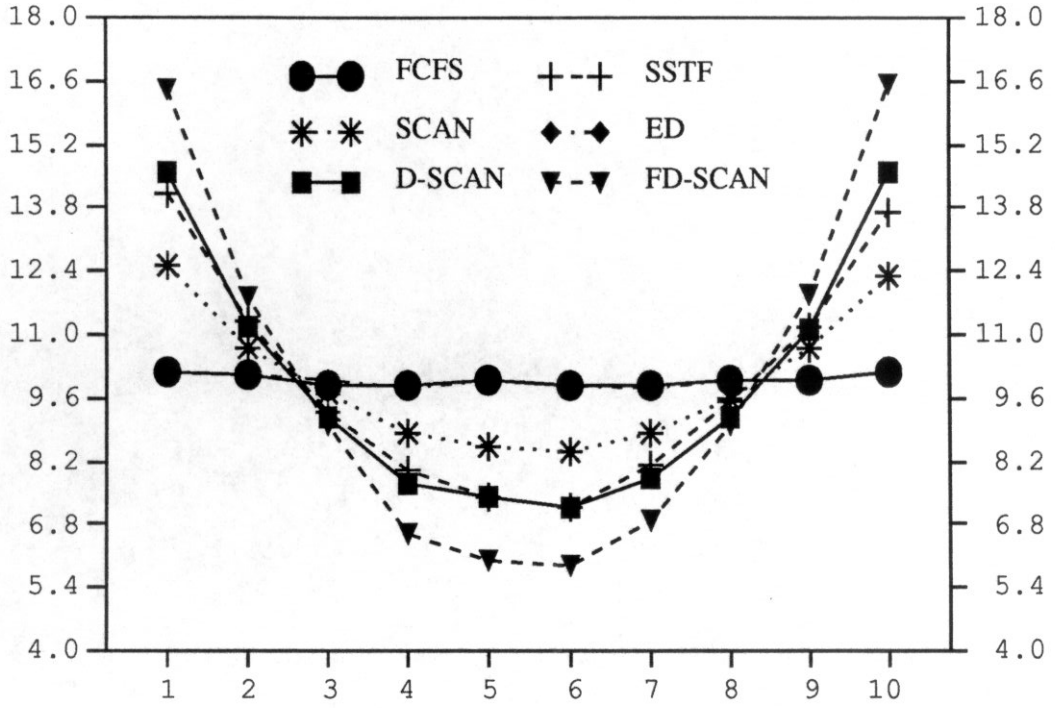


Figure 5-8 ***Min_Slack*** **= 10,** ***Max_Slack*** **= 100**

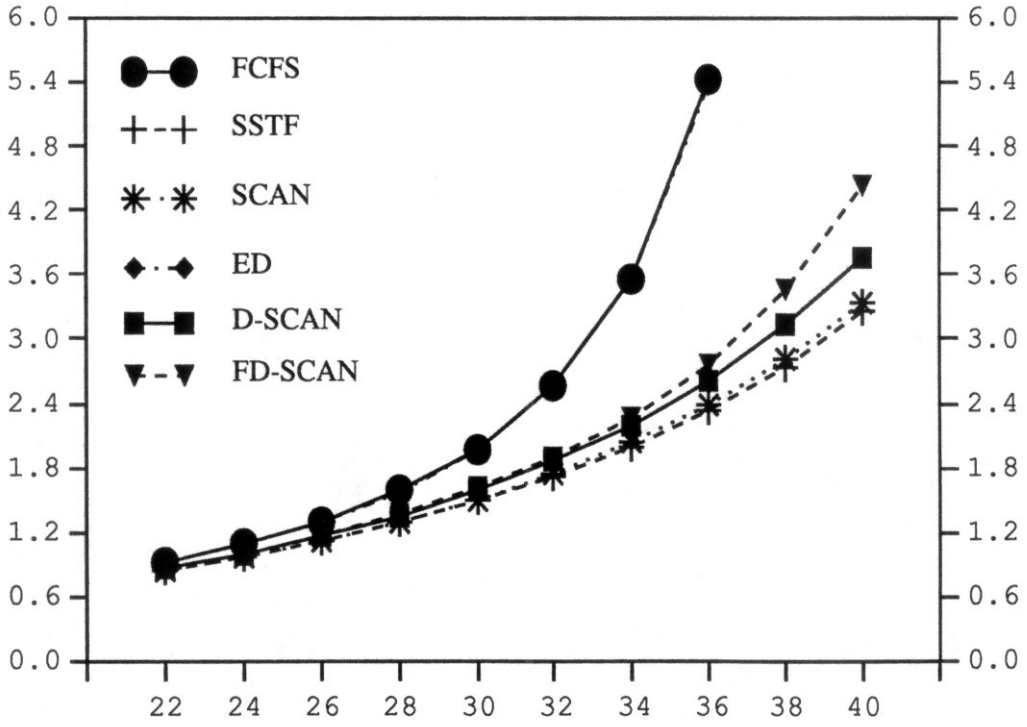


Figure 5-9 Vary *Read_Rate*

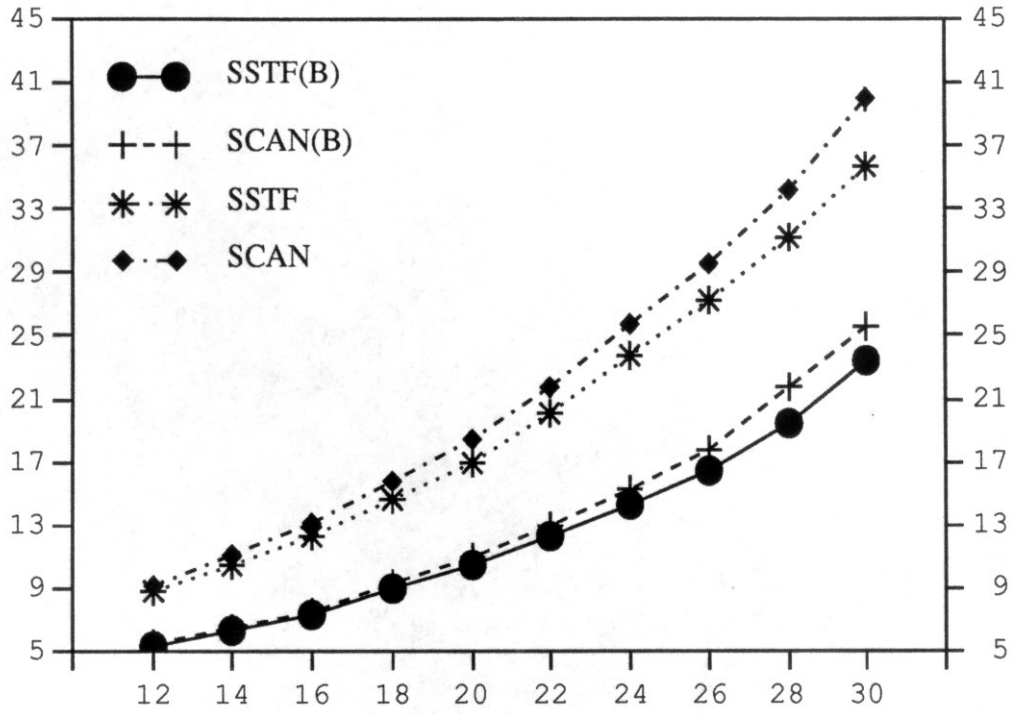


Figure 5-10 Vary *Read_Rate*

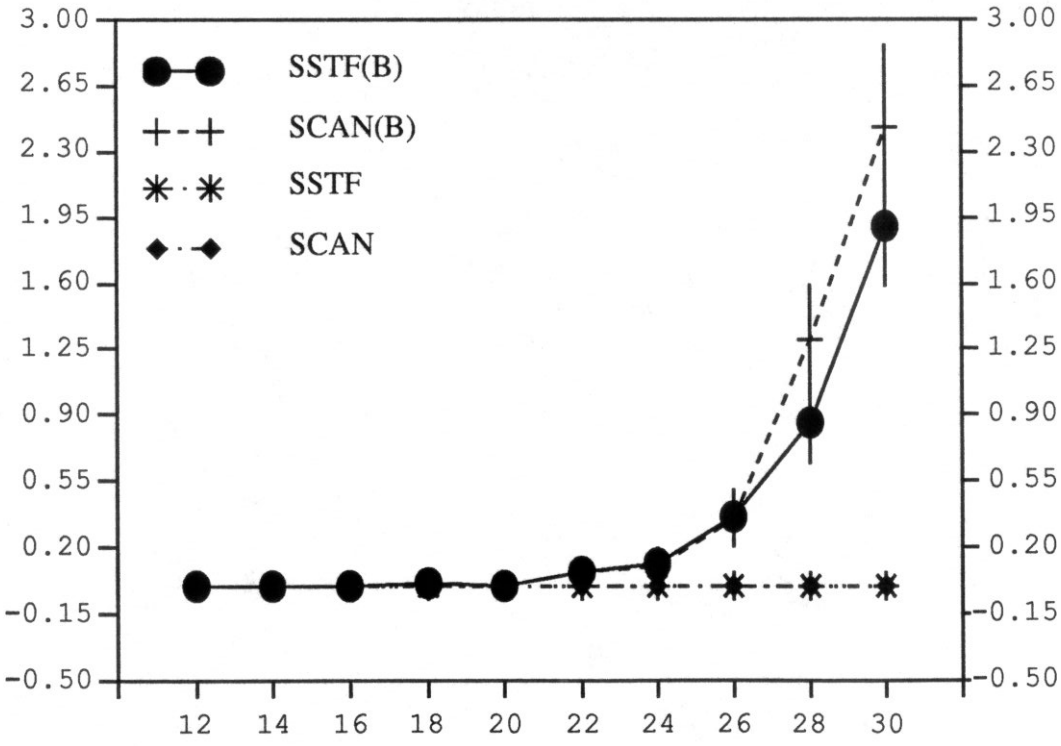


Figure 5-11 Vary *Read_Rate*

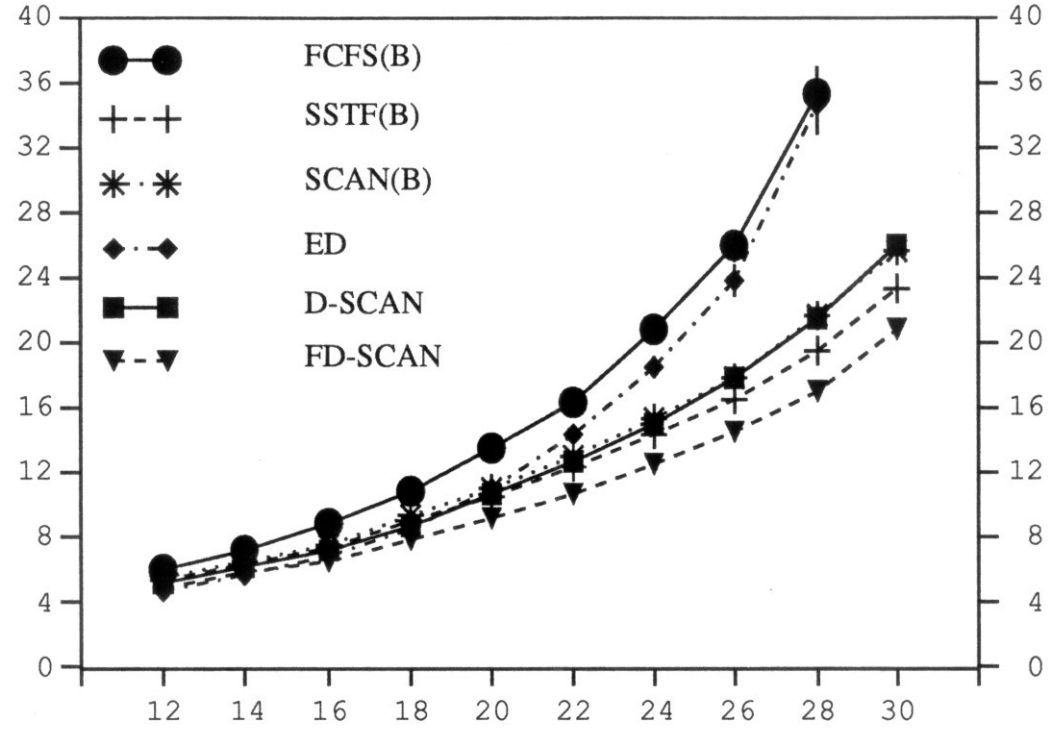


Figure 5-12 Vary *Read_Rate*

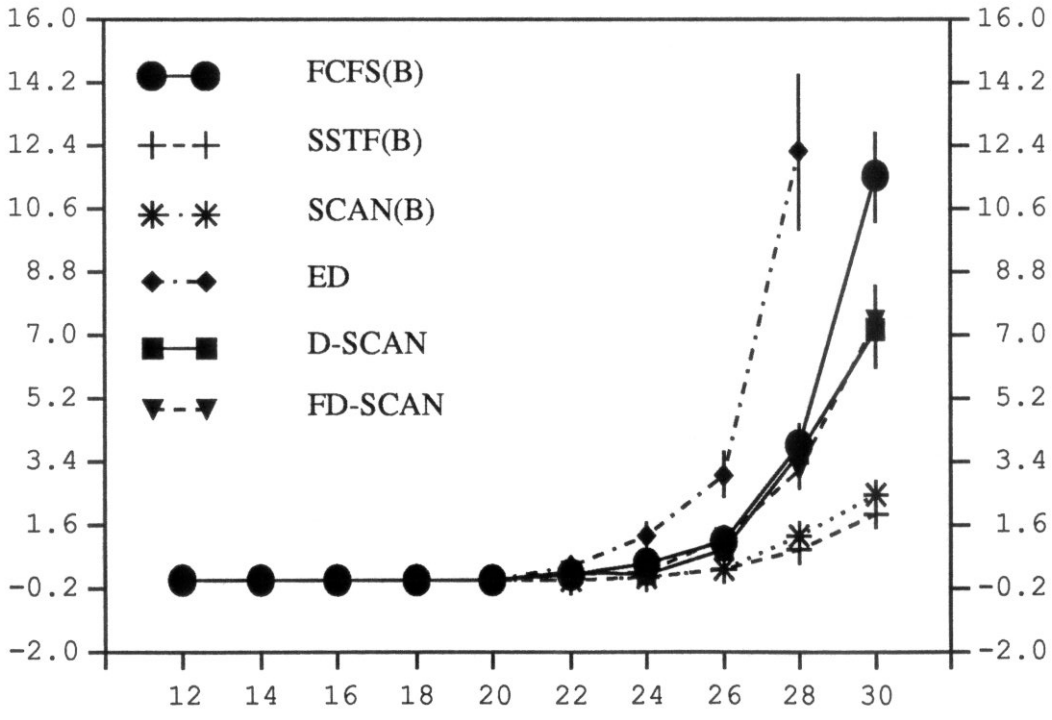


Figure 5-13 Vary *Write_Rate*

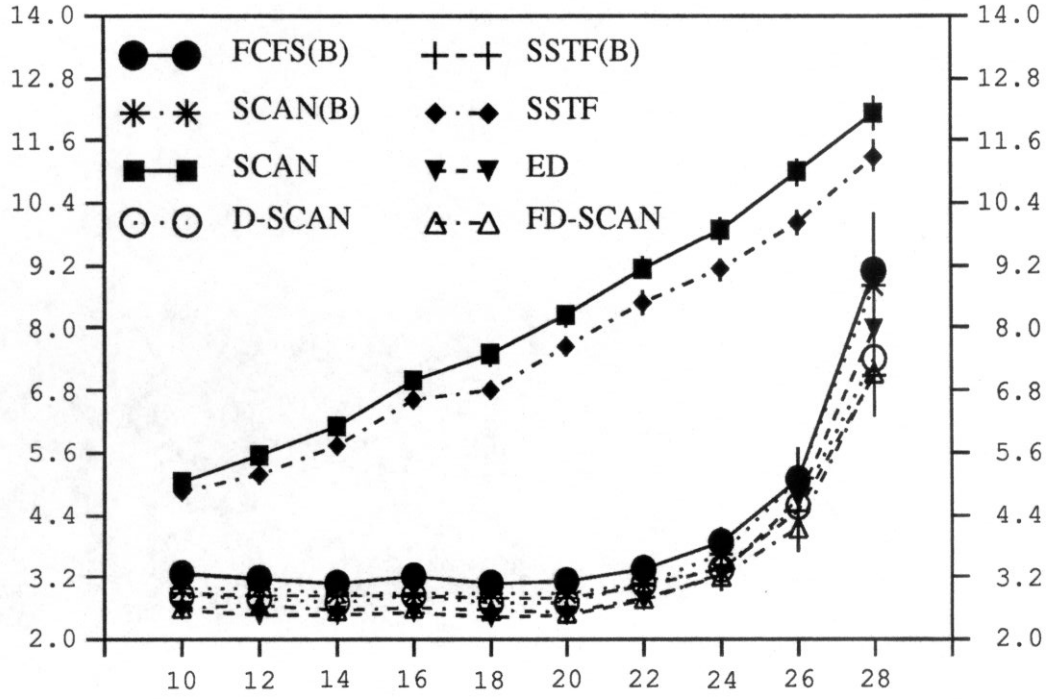


Figure 5-14 Vary *Buffer_Size*

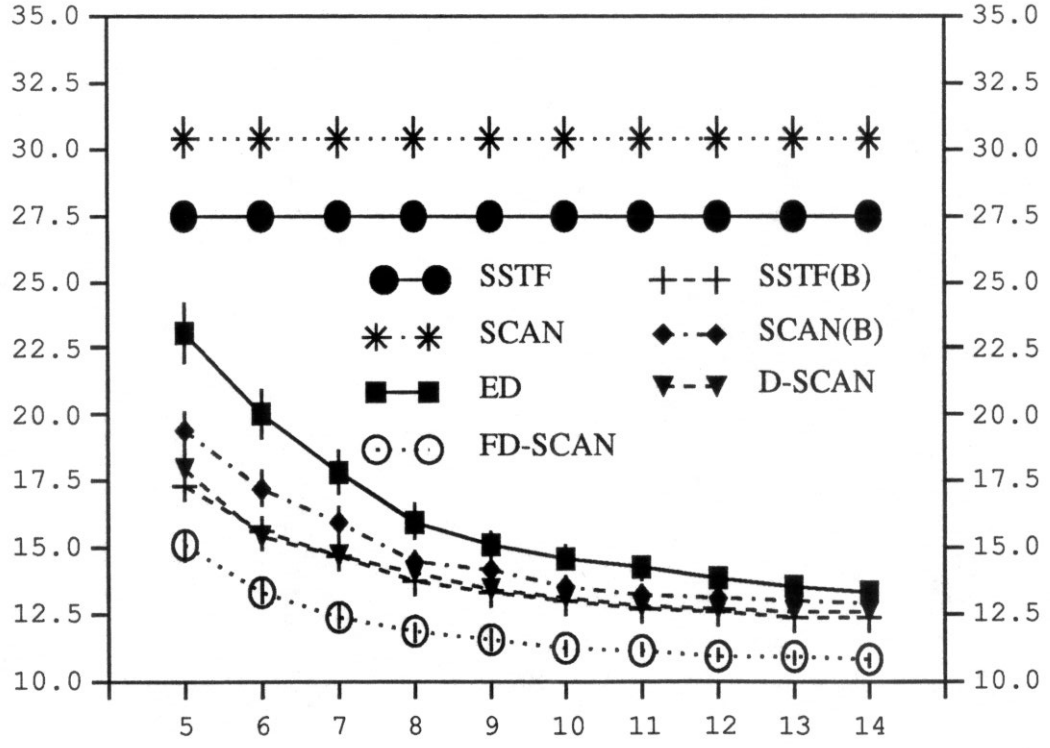


Figure 5-15 Vary *Space_Threshold*

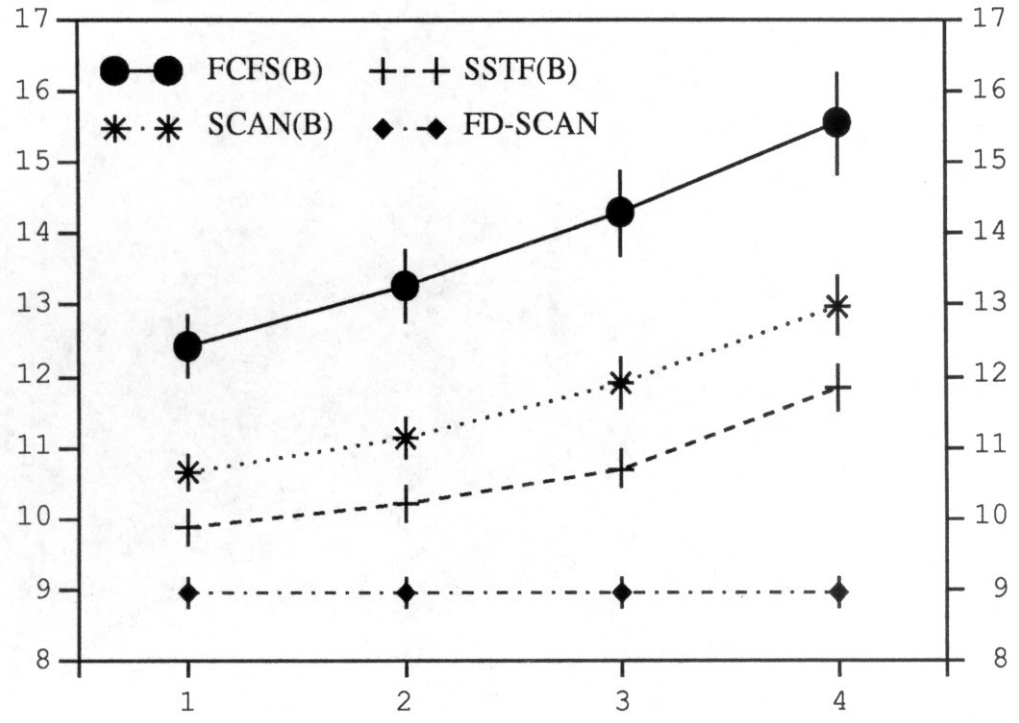
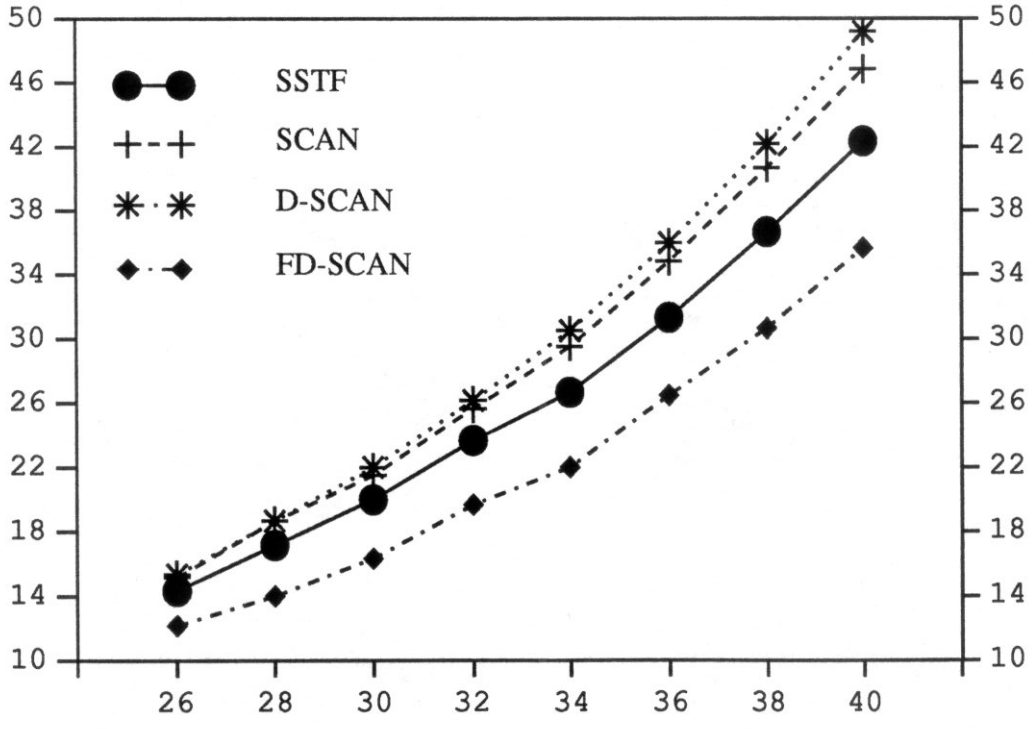


Figure 5-16 Checkpoint Experiment



5.6 Conclusions

In Chapters 4 and 5 we have presented a number of new algorithms for scheduling I/O requests with deadlines. The algorithms were evaluated via detailed simulation and compared with traditional scheduling algorithms. One new algorithm, FD-SCAN, had consistently the best performance in a wide variety of experiments.

We also investigated a model for handling read requests differently from write requests. This model buffers write requests in a separate queue from read requests. Two techniques for managing the buffer were examined and both were found to be effective. The overall approach of buffering writes was especially helpful for meeting read deadlines. We believe that systems. e.g., real-time information systems, that produce read requests with deadlines and write requests without, can use this approach to great benefit.

Chapter 6 Future Directions

The research area known as real-time databases or real-time transaction scheduling is only four years old. This thesis presents some of the earliest work done in this area and represents a significant contribution to computer science. Indeed, much of the work by other researchers has either built upon our work or compared itself to ours. **[HCL1, HSRT1]**.

As might be expected of such a young field, some of the results presented so far are contradictory and by no means is there a consensus on which are the best algorithms. For example, we have reported that concurrency control policies that combine blocking with priority inheritance (Wait Promote, Conditional Restart) generally perform better than a policy that aborts lower priority transactions in order to favor higher priority ones (High Priority). The opposite of this result is reported in **[HSRT1]**. The reason for this may lie in the fact that they used an experimental testbed for their performance results and were unable to program the disk device driver to implement priority scheduling of I/O requests. Our performance evaluation was done via simulation and did incorporate priority based disk scheduling. Furthermore, our results show that priority based disk scheduling significantly enhances the performance of the priority inheritance algorithms (see Figure 3-15). The beneficial aspect of priority inheritance in real-time database systems is also reported in **[SL]**.

The well-known debate regarding the performance of locking based concurrency control protocols versus optimistic ones has surfaced in the field of real-time database systems. One group of researchers has reported an optimistic concurrency control algorithm that can generally perform better than a locking based protocol (our High Priority

algorithm) [**HCL1**]. A different group of researchers has reported the opposite [**HSRT2**].

The small differences in transaction models and the large differences in evaluation techniques (e.g., simulation versus experimental testbed) make it difficult to resolve these competing claims. Only after many more researchers examine these problems will a clearer understanding of these problems begin to develop.

Although it would certainly be fruitful to develop more algorithms and analysis for real-time transaction scheduling, we feel that a stronger, more integrated system and transaction model is needed to support advancement in this field. We make that observation that traditional task models in the field of "hard" real-time scheduling generally assume that the CPU is the only resource to be scheduled. If the model includes shared access to data resources then the scheduler usually knows all task resource requirements *a priori*. These types of models lead to the development of monolithic schedulers, or a single scheduling algorithm that has global knowledge and makes global decisions.

By contrast, real-time database systems have many resources that are shared among tasks and thus require scheduling. These resources include the CPU, memory, data objects, communication links, and storage devices. In conventional systems each of these resources is scheduled and controlled by an independent component. These are the CPU scheduler, the buffer manager, the concurrency control manager, the ethernet controller, and device drivers. Each of these schedulers acts independently from the others and each employs its own policy to optimize performance. Each component has local knowledge and makes local decisions.

This same approach that has been adopted in this thesis as well as all other work on real-time transaction scheduling. We, (and others) have proposed and examined a

number of choices for each scheduling component. The choices are combined to achieve a total scheduling solution. A problem with this approach is that it leads to a multitude of scheduling algorithms, far too many to analyze or understand completely. An alternate approach would propose a stronger, more satisfactory model for transactions and the transaction processing system. Components for scheduling system resources would cooperate more closely in meeting transaction time-constraints. Instead of purely local knowledge, a component might have regional or perhaps global knowledge. Local decisions made by the individual scheduling component would then be reflected in this global knowledge pool so that other components could make use of it.

References

[**AGM1**] Abbott, Robert and Hector Garcia-Molina, "What is a Real-Time Database System?," *Abstracts of the Fourth Workshop on Real-Time Operating Systems,* IEEE Computer Society, (July 1987) 134-138.

[**AGM2**] Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, (March 1988) 71-81.

[**AGM3**] Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-Time Transactions: a Performance Evaluation," *Proceedings of the 14th VLDB Conference*,(August 1988) 1-12.

[**AGM4**] Abbott, Robert and Hector Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*,(August 1989).

[**AGM5**] Abbott, Robert and Hector Garcia-Molina, "Scheduling I/O Requests with Deadlines: a Performance Evaluation," *IEEE Real-Time Systems Symposium,* (December 1990) 113-124.

[**BHG**] Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Mass. (1987).

[**BG**] Bitton, D. and J. Gray., "Disk Shadowing," *Proceedings of the 14th VLDB Conference,* (August 1988) 331-338.

[**BGMP**] Blasgen, M. J. Gray, M. Mitoma, and T. Price, "The Convoy Phenomenon," *Operating Systems Review*, **13**(2) (April 1979) 20-25.

[**Br**] Bryant, R. M., "SIMPAS 5.0 User Manual," Computer Sciences Dept. **TR-146**, Univ. of Wisconsin-Madison, (November 1981).,

[**CJL1**] Carey, M., R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference,* (August 1989) 397-410.

[**CGL2**] Carey, M. R. Jauhari, and M. Livny, "On Transaction Boundaries in Active Databases: A Performance Perspective," Computer Sciences Dept. **TR-796**, Univ. of Wisconsin-Madison, (November 1988).

[**CSR**] Cheng, S., J. Stankovic and K. Ramaritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *IEEE Real-Time Systems Symposium,* (December 1986) 166-174.

[**CDW**] Chou, H. and D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the 11th VLDB Conference,* (August 1985).

[**Cl**] Clark, Raymond, "Scheduling Dependent Real-Time Activities," Thesis Proposal, Dept. of Computer Science, Carnegie-Mellon Univ., (October 1988).

[**CD**] Coffman, E. and P. Denning, *Operating Systems Theory,* Prentice Hall, Englewood Cliffs, NJ (1973).

[**CH**] Coffman, E.G., and M. Hofri, "On the Expected Performance of Scanning Disks," *SIAM Journal of Computing*, **11**(1) (February 1982) 60-70.

[**CKR**] Coffman, E.G., L. Klimko and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times," *SIAM Journal of Computing*, (September 1972) 269-279.

[**CMM**] Conway, R, W. Maxwell, and L. Miller, *Theory of Scheduling,* Addison Wesley, Reading, MA (1967).

[**Das**] Dasarthy, B., "Timing Constraints of Real-Time Systems: Constructs for Expressing them, Methods of Evaluating them," *IEEE Real-Time Systems Symposium*, (December 1982) 197-204.

[Da] Date, C. J., *An Introduction to Database Systems (4th ed),* **1,2** Addison Wesley Reading, MA (1986).

[DLW] Davidson, S., I. Lee, and V. Wolfe, "A Protocol for Timed Atomic Commitment," *IEEE ICDCS* (July 1989) 199-206.

[Day] Dayal, U. et al. "The HiPAC Project: Combining Active Databases and Timing Constraints," *ACM SIGMOD Record,* (March 1988) 51-70.

[De] Denning, Peter, "The Working Set Model for Program Behavior," *Communications of the ACM*, **11**(5) (1968) 323-333.

[Deu] Deutsch, Michael S., "Focusing Real-Time Systems Analysis on User Operations," *IEEE Software*, **5** (September 1988) 39-50.

[EH] Effelsberg, W. and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, **9**(4) (December 1984) 560-595.

[EGLT] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, **19**(11) (November 1976) 624-633.

[Fo] Ford, Ray, "Concurrent Algorithms for Real-Time Memory Management," *IEEE Software*, **5** (September 1988) 10-23.

[GM] Garcia-Molina, Hector, "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems,* **8** (June 1983) 186-213.

[GD] Geist, Robert and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems*, **5**(1) (February 1987) 77-92.

[**GL**] Gligor, V. and G. Luckenbaugh, "An Assessment of the Real-Time Requirements for Programming Environments and Languages," *IEEE Real-Time Systems Symposium,* (December 1983) 3-19.

[**Gr**] Gray, Jim, "Notes on Database Operating Systems," *Operating Systems: An Advanced Course*, G. Seegmuller, Editor, Springer-Verlag, (1978) 393-481.

[**HR**] Haerder, T. and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys*, **15**(4) (1983) 287-317.

[**HCL1**] Haritsa, J., M. Carey and M. Livny, "On Being Optimistic About Real-Time Constraints," *Proc. ACM Symposium on Principles of Database Systems,* (April 1990).

[**HCL2**] Haritsa, J., M. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *IEEE Real-Time Systems Symposium,* (December 1990) 94-103.

[**HS**] Huang, J. and J. Stankovic, "Buffer Management in Real-Time Databases," Univ. of Massachusetts, COINS TR 90-65, (July 1990).

[**HSRT1**] Huang, J., J. Stankovic, K. Ramaritham, and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," Univ. of Massachussetts, COINS TR 90-121, (November 1990).

[**HSRT2**] Huang, J., J. Stankovic, K. Ramaritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," Univ. of Massachussetts, COINS TR 91-16, (January 1991).

[**HSTR**] Huang, J., J. Stankovic, D. Towsley, and K. Ramaritham, "Experimental Evaluation of Real-Time Transaction Processing," *IEEE Real-Time Systems Symposium*, (December 1989) 144-153.

[**IM**] Isloor, S., and T. Marsland, "The Deadlock Problem: An Overview," *IEEE Computer*, (September 1980) 55-78.

[**IYL**] Iyer, B., P. Yu and Y. Lee, "Analysis of Recovery Protocols in Distributed On-line Transaction Processing Systems," *IEEE Real-Time Systems Symposium,* (December 1986) 226-233.

[**Ja**] Jaffay, Kevin, "Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints," *IEEE Real-Time Systems Symposium,* (December 1989) 295-305.

[**JCL**] Jauhari, R., M. Carey and M. Livny, "Priority Hints: An Algorithm for Priority Based Buffer Management," Computer Sciences Dept., Univ. of Wisconsin-Madison, TR 911, (February 1990).

[**JLT**] Jensen, E., C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *IEEE Real-Time Systems Symposium*, (December 1985) 112-122.

[**Je**] Jensen, E. Douglas, "The Future Reality of Real-Time," *Unix Review*, (January 1989).

[**KSB**] Krishna, C. M., K. G. Shin, and I. S. Bhandari, "Processor Trade-offs in Distributed Real-Time Systems," *IEEE Transactions on Computers* **c-36**(9) (September 1987) 1030-1040.

[**LD**] Lee, I. and S. Davidson, "Adding Time to Synchronous Process Communications," *IEEE Transactions on Computers* **c-36**(8) (August 1987) 941-948.

[**LDW**] Lee, I. S. Davidson and V. Wolfe, "Motivating Time as a First Class Entity," *Abstracts of the Fourth Workshop on Real-Time Operating Systems*, (July 1987) 165-169.

[**LYI**] Lee, Y. H., P. S. Yu, and B. R. Iyer, "Progressive Transaction Recovery in Distributed DB/DC Systems," *IEEE Transactions on Computers* **c-36**(8) (August 1987) 976-987.

[**Le**] Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *IEEE Real -Time Systems Symposium,* (December 1990) 201-209.

[**LSD**] Lehoczky, J. P., L. Sha, and Y. Ding, "The rate monotonic scheudling algorithm: exact characterization and average case behavior," *IEEE Real-Time Systems Symposium*, (December 1989) 166-171.

[**LAT**] Levi, S. A. Agrawala, and S. Tripathi, "Introducing the MARUTI Hard Real-Time Operating System," Dept. of Computer Science, Univ. Maryland, CS-TR-2010 (April 1988).

[**LL**] Lin, K. and M. Lin, "Enhancing Availability in Distributed Real-Time Databases," *ACM SIGMOD Record*, (March 1988) 34-43.

[**LS**] Lin, Y. and S. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *IEEE Real-Time Systems Symposium,* (December 1990) 104-112.

[**LL**] Liu, C. L. and J. W. Layland, "Scheduling Algorithms for Multiprogramming in Hard Real-Time Environment," *Journal of the ACM,* **20** (January 1973) 46-61.

[**LB**] Luqui, and V. Berzins, "Rapidly Prototyping Real-Time Systems," *IEEE Software*, **5** (September 1988) 25-36.

[**Ma**] Marzullo, Keith, "Concurrency Control for Transactions with Priorities," Dept. of Computer Science, Cornell Univ. TR 89-996, (May 1989).

[**Mok1**] Mok, Aloysius, "Fundamental Design Problems of Distributed Systems for the Real-Time Environment," Ph.D Thesis, MIT Dept. of Electrical Engineering and Computer Science (May 1983).

[**Mok2**] Mok, A., "The Design of Real-Time Programming Systems Based on Process Models," *IEEE Real-Time Systems Symposium,* (December 1984) 5-17.

[**Mok3**] Mok, A., "The Decomposition of Real-Time System Requirements into Process Models," *IEEE Real-Time Systems Sympoisum*, (1984) 125-134.

[**Mo**] Moore, J. M. An N Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs," *Management Science*, **15** (1968) 102-109.

[**PS**] Peterson, J and A. Silberschatz, *Operating System Concepts*, Addison Wesley, Reading, Mass. (1986).

[**PSMK**] Psaraftis, H. N., M. Solomon, T. Magnanti, and T. Kim., "Routing and Scheduling on a Shoreline with Release Times," MIT Tech. Report, OR 152-86, (September 1986)

[**Re**] Reuter, Andreas, "Performance Analysis of Recovery Techniques," *ACM Transactions on Database Systems*, **9**(4) (December 1984) 526-559.

[**Ri**] Risch, Tore, "Monitoring Database Objects," *Proceedings of the 15th VLDB Conference,* (August 1980) 445-453.

[**RCB**] Rosenthal, A. S. Chakravarthy, and B. Blaustein, "Situation Monitoring for Active Databases," *Proceedings of the 15th VLDB Conference*, (August 1989) 455-464.

[**SS**] Sacco, G. and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM Transactions on Database Systems*, **11**(4) (December 1986) 473-498.

[Sc] Schwetman, Herb, "CSIM Reference Manual," MCC.

[SCO] Seltzer, Margo, P. Chen and J. Ousterhout, "Disk Scheduling Revisited," *Proceedings of USENIX, Winter '90,* pp. 313-323.

[SLJ] Sha, L., J. P. Lehoczky, and E. D. Jensen, "Modular Concurrency Control and Failure Recovery," *IEEE Transactions on Computers,* **37** (February 1988) 146-159.

[SLR] Sha, L., J. P. Lehoczky, and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium,* (December 1986) 181-191.

[SRL1] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," Dept. of Computer Science, Carnegie-Mellon Univ., CMU-CS-87-181 (December 1987).

[SRL2] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, **17** (March 1988) 82-98.

[Sh] Shih, et al., "Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error," *ACM Operating Systems Review*, **23**(3) (July 1989) 14-28.

[SL] Song, X. and J. Liu, "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, (February 1990).

[St1] Stankovic, John, "Real-Time Computing Systems: The Next Generation," Dept. of Computer and Information Science, Univ. of Mass.-Amherst, COINS TR 88-06 (January 1988).

[St2] Stankovic, John, "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, **21** (October 1988) 10-19.

[SZ] Stankovic, J. and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, **17** (March 1988) 4-18.

[**Stok**] Stok, P. van der, "The Feasibility of a Relational Database Programming Language in Process Control," *IEEE Real-Time Systems Symposium*, (December 1984) 105-113.

[**Sto**] Stonebraker, Michael, "Operating System Support for Database Management," *ACM Communications*, **24**(7) (July 1981) 412-418.

[**TP**] Teorey, Toby J. and Tad B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, **15**(3) 177-184.

[**TM**] Tokuda, H. and C. Mercer, "ARTS: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, **23**(3) (July 1989) 29-53.

[**TWW**] Tokuda, H., J. Wendorf and H. Wang, "Implementation of a Time-Driven Scheduler for Real-Time Operating Systems," *IEEE Real-Time Systems Symposium*, (December 1987) 271-280.

[**Vo**] Voelcker, John, "How Computers Helped Stampede the Stock Market," *IEEE Spectrum*, **24** (December 1987) 30-33.

[**Wi**] Wilhelm, Neil C., "An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empircal Model for Disk Accesses," *ACM Communications*, **19**(1) 13-17.

[**ZRS1**] Zhao, W., K. Ramamritham, and J. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints," *IEEE Transactions on Computers*, **c-36** (August 1987) 949-960.

[**ZRS2**] Zhao, W., K. Ramamritham, and J. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, **SE-13** (May 1987) 564-577.