INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9321159

Transaction processing in real-time database systems

Haque, Waqar ul, Ph.D.

Iowa State University, 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Transaction processing in real-time database systems

by

Waqar ul Haque

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Computer Science
Major: Computer Science

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Members of the Committee:

Signature was redacted for privacy.

Iowa State University
Ames, Iowa
1993

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

This thesis is not only a result of endless hours spent on research, but owes a lot to a number of people who directly or indirectly became a part of the whole process.

First of all, I would like to thank Dr. Johnny Wong for his patience, excellent guidance and support in academic as well as non-academic matters. We spent hours discussing various aspects of this research and his ideas were always a motivation that kept me sailing along. Dr. Wong's multiple passes through this dissertation and helpful suggestions has resulted in a high quality manuscript. In many other cases, his friendly advice during times of frustration kept me emotionally together over all these years.

I express my gratitude to other members of my POS committee, Dr. Arthur Oldehoeft, Dr. Les Miller, Dr. Glenn Luecke and Dr. Jim Davis for their helpful suggestions whenever I needed those. Working with Dr. Luecke on a variety of challenging projects resulted in enhanced research skills, the proof of which is many publications with him in international reputed journals. I would also like to thank Howard Jespersen in our Research Computing Group who passed the idea of using shell scripts to automate some simulation experiments; his suggestion certainly saved many hours of labor during thousands of simulation runs that were required.

I would also like to express my gratitude to my family, particularly my son

Mohsin and my daughter Uneza who spent countless evenings and weekends without me. My long-term commitment to this degree also kept them away from the rest of the loving family in Pakistan. My special thanks to my wife, Tauqir, whose love, understanding and endurance has been a continuing boost. She never seemed tired taking care of responsibilities which I should have shared. And last but not least, I owe so much to my caring parents: to my mother who stayed behind alone at an age when she needed me there most, and to my father who inspired me to reach this level of education, but never lived to watch it happen.

CHAPTER 1. INTRODUCTION

Real-Time Systems

The most significant difference between real-time systems and other computer systems is the importance of correct timing behavior. Real-time systems have a dual notion of correctness; that is, the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced. This implies that, unlike many systems where there is a separation between correctness and performance, in real-time systems correctness and performance are very tightly interrelated. Real-time systems span many application areas. In addition to automated factories, applications can be found in avionics, process control, robot and vision systems, as well as military systems such as command and control. The flexibility in meeting timing constraints depends on the type of application. For instance, a real-time system that controls a nuclear power plant, or one that controls a missile, cannot afford to miss timing constraints. On the other hand, in the case of a periodic task monitoring an aircraft, depending on the aircraft's trajectory, missing the processing of one or two radar readings may not cause any problems.

Based on the flexibility of timing constraints, the real-time systems are generally classified as *hard* and *soft* real-time systems. Hard real-time systems have timing constraints which should be guaranteed to be met to avoid catastrophic circumstances.

For example, a moving robot may collide with other objects if it does not stop on time or if it changes its direction too late. Therefore, guaranteeing that all timing requirements are met is one of the most important issues in the design of hard real-time systems. Flight control systems, process control systems, and automated manufacturing plants are a few examples of such systems. Most of the hard real-time computer systems are special-purpose and complex and require a high degree of fault tolerance. Also, these systems have substantial amounts of knowledge concerning the characteristics of the application and the environment built into the system. A majority of today's hard real-time systems assume that much of this knowledge is available *a priori* and, hence are based on *static* design with preallocated resources thereby making the systems very expensive and inflexible [48]. In soft real-time systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times; late results are still valuable, although the value may be reduced after a critical time. Banking and airline reservations are two traditional examples of interactive database systems with such real-time performance requirements.

Task scheduling in real-time systems can be *static* or *dynamic*. A static approach calculates schedules for tasks off-line and it requires the complete prior knowledge of tasks' characteristics. A dynamic approach determines schedules for tasks on the fly and allows tasks to be dynamically invoked. Although static approaches have low run-time cost, they are inflexible and cannot adapt to a changing environment or to an environment whose behavior is not completely predictable. When new tasks are added to a static system, the schedule for the entire system must be recalculated which can be an expensive operation. In contrast, dynamic approaches involve higher run-time costs, but their design is focused on making them flexible enough to adapt

to changes in the environment. The scheduling issues in a real-time environment are significantly different from conventional scheduling theory and those considered in areas of operation research. The goal is not only to minimize response time, but to have dynamic, on-line, adaptive scheduling algorithms which ensure that deadlines of individual tasks are met.

Many practical instances of scheduling algorithms, including the problem of determining a nonpreemptive optimal schedule, have been found to be NP-complete [19]. A majority of scheduling algorithms reported in the literature perform static scheduling and hence have limited applicability since all task requirements, particularly for the dynamically arriving aperiodic tasks, are not known *a priori.* For dynamic systems with more than one processor, an optimal scheduling algorithm does not exist [30]. These negative results point out the need for heuristic approaches to solve scheduling problems in such systems. An important metric to determine the effectiveness of heuristics-based algorithms is the measure of percent of *tardy tasks.* Tardy tasks are defined as those tasks which have missed their deadlines. In order to minimize the number of tardy tasks, efficient real-time scheduling algorithms are required.

Real-Time Database Systems

A database is a collection of information consisting of physical data on some storage media and a conceptual structure. The operations on the database are performed by *transactions* which is a collection of instructions requiring reading or updating database values. A database management system, which acts as an interface between the user and the data, is used to manage the data efficiently and to pro-

vide a logical view of the database. An important advantage of a database is to allow sharing of data thereby reducing redundancy by not requiring to have multiple copies of the same information. However, with such sharing of data, it becomes important to enforce data integrity to avoid accidental or malicious changes to the values in a database. This is usually enforced by using some form of access control to maintain data consistency. The following example shows that in the absence of concurrency control, a concurrent execution of a set of transactions can transform a consistent database state into an inconsistent state [17].

Example 1.1 Consider the two transactions T_1 and T_2 described below:

$$\begin{aligned} T_1: \quad & x \leftarrow x + 1, \\ & y \leftarrow y + 1, \\ T_2: \quad & y \leftarrow 2 * y, \\ & x \leftarrow 2 * x. \end{aligned}$$

Suppose that the integrity constraint on x and y is that $x = y$. Consider the following sequence of execution:

$$\begin{aligned} T_1: \quad & x \leftarrow x + 1 \\ T_2: \quad & y \leftarrow 2 * y \\ T_1: \quad & y \leftarrow y + 1 \\ T_2: \quad & x \leftarrow 2 * x \end{aligned}$$

It is obvious that after executing T_1 and T_2 concurrently as described above, the values of x and y will not be the same. Thus the state will be inconsistent.

Serializability is a widely accepted criterion for ensuring database consistency. Serializability requires that the combined action of a group of transactions accessing

the database is equivalent to some serial schedule, that is, the same as if all the transactions would have executed serially in some order. *Two-phase locking* is a well-known mechanism for ensuring serializability and thereby maintaining concurrency. It requires obtaining a lock before accessing a database entity, O. When a transaction T_i requests a read-lock on O, this lock will only be granted if no other transaction already holds a write-lock on O. Similarly, when T_i requests a write-lock on O, this lock will only be granted if no other transaction holds a read-lock or a write-lock on O. If a read-lock or write-lock on some entity cannot be obtained because the entity is already locked, the requesting transaction, T_i, has to wait until the lock is released. Two-phase locking guarantees serializability by preventing a transaction from obtaining a lock on any entity after releasing a lock on any other entity. Therefore, each transaction has two phases: a growing phase during which locks can only be obtained, followed by a shrinking phase during which locks can only be released. The point at which the transaction releases its first lock delimits the two phases. This point is called the locked point of the transaction. In some cases, all obtained locks are held until the transaction is ready to commit. At this point, all locks are released by a single atomic action. This form of two-phase locking is called *strict two-phase locking* because each transaction maintains locks until termination. In the rest of this dissertation, the term two-phase locking will refer to the strict two-phase locking.

A *real-time database* is a database system where transactions also have explicit timing constraints expressed in the form of a *deadline*. A deadline indicates that the transaction must be completed before a certain time. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints. This requires a preferential treatment to transactions with higher

priority values. The goal, therefore, should be to provide the best possible service to higher-priority transactions while minimizing the negative impact on lower-priority transactions. Such priority scheduling can be used as a way of minimizing the number of missed deadlines in a soft real-time transaction processing environment. Real-time database systems can be found in program trading in the stock market, radar tracking systems, battle management systems, and computer integrated manufacturing systems.

Conventional real-time systems do take into account individual transaction's timing constraints but ignore data consistency problems. On the other hand, conventional database systems focuses on query processing and database consistency, but not on meeting any time-constraints associated with transactions. When the concept of real-time is extended to database systems, it adds another dimension to the scheduling problem [47]. Like traditional databases, it is important to ensure data consistency in real-time databases. However, for timeliness of results, in contrast to traditional databases where the primary goal is to minimize the response time of user transactions and maximize throughput, the main objective in real-time databases is to ensure that transactions meet their deadlines and to minimize the percentage of transactions that miss deadlines in the system.

Priority Inversion Problem

For real-time priority-driven preemptive scheduling, each transaction is assigned a priority according to its deadline. The execution of concurrent transactions is scheduled based on their assigned priority. Ideally, a high priority transaction should never be blocked by any lower priority transaction. In particular a high priority

transaction may preempt a lower priority transaction to receive service. Under two-phase locking, on the other hand, a transaction must obtain a lock before accessing a data object and release the lock when it terminates (commits or aborts). A lock-requesting transaction will be placed in a wait queue if its lock mode is found to be incompatible with that of the lock-holding transaction(s). The queued transaction can proceed only when it is granted the lock. When the priority driven-preemptive scheduling approach and the two-phase locking protocol are simply integrated together in a real-time database system, a problem known as *priority inversion* may arise.

Priority inversion is said to occur when a high priority transaction must wait for the execution of lower priority transactions [24, 40]. This situation can usually occur when more than one transaction attempt to access shared data. The following example illustrates the priority inversion problem.

Example 1.2 Suppose that T_1 , T_2 , and T_3 be three transactions in descending order of priority. Let T_1 and T_3 access the same data object O. Suppose that at time t_1 transaction T_3 locks O. During the execution of T_3 , the higher priority transaction T_1 arrives, preempts T_3 and later tries to access O. Now T_1 will be blocked since O is already locked and T_3 will resume its execution. Note that the duration of this blocking can be unpredictable. This is because transaction T_3 can be preempted by the intermediate priority transaction T_2 that does not need to access O. The blocking of T_3 , and hence of T_1 , will continue until T_2 and any other pending transactions of intermediate priority are completed.

In order to maintain consistency, the access must be serialized. If the higher priority transaction gains access first then the proper priority order is maintained;

however, if the lower priority transaction gains access first and then the higher priority transaction requests access to the shared data, this higher priority transaction is blocked until the lower priority transaction completes its access to the shared data. Even worse, the duration of such a blocking can also become unbounded and prolonged durations of blocking may lead to the missing of deadlines even at a low level of resource utilization. In summary, blocking with priority inversion implies that higher priority transactions are waiting while lower priority transactions are in execution. This defeats the purpose behind priority assignment. A common approach to bound such arbitrary delays is to execute the transaction that holds the lock at a higher priority. A survey of some existing schemes that use this approach is given in the next chapter.

Nested Transactions

The difference between transactions and nested transactions is that nested transactions have more internal structure. A transaction is just a group of *primitive* actions (e.g., reads and writes of simple data objects) that are performed as a unit (atomically). Nested transactions have hierarchical grouping structure: each nested transaction consists of either primitive actions or some nested transactions. Traditional atomic transactions provide automatic synchronization of accesses to and updates of data. Nested transactions, an extension of traditional atomic transactions, permit safe concurrency within as well as among transactions and also enable transactions to fail partially in a graceful and controlled manner. Thus, nested transactions have at least two advantages over traditional single-level transactions. First, nested transactions have mechanisms that provide appropriate synchronization be-

tween concurrently running parts of the same transaction. This implies that more work can be processed concurrently without the danger of inconsistencies arising through improper concurrent access to data. Because of this correct synchronization provided within a transaction, a number of previously existing transactions can be combined into a new transaction without consistency problems. Second, subtransactions of a nested transaction fail independently of each other and independently of the containing transaction. This allows possibilities such as attempting a piece of a computation at one node and redoing that piece if the node fails. In the single-level transaction system, if any piece fails, the whole transaction fails [32].

Synchronization in nested transactions can be achieved by using locking or timestamp based protocols. Each scheme has drawbacks and advantages relative to the other. Locking requires that there be an instant in time at which a transaction holds all the locks it ever needs. In contrast, timestamp synchronization does not require that all of transaction's resources be held at once. These facts suggest that in some circumstances timestamp synchronization might provide more concurrency. However, locking permits transactions to serialize only as necessary and schedule itself dynamically. Timestamp ordering, on the other hand, determines the relative order of transactions in advance. This may suggest that timestamp schemes require additional waiting (to know that no more transactions with timestamps in a certain range will arrive at a given node) or will abort more transactions (a transactions with an early timestamp may be aborted at a given node because a transaction with a later timestamp has already committed).

Problem Statement

Scheduling transactions in a real-time database requires an integrated approach in which the 'schedule' does not only guarantee execution before the deadline, but also maintains data consistency (because the transactions may execute concurrently and will access a database in some unpredictable pattern). It is, therefore, necessary to study the problem of scheduling real-time transactions under a common framework which considers both concurrency control issues and the real-time constraints. Such a framework should then lead to the development of a real-time concurrency control protocol that maximizes both concurrency and resource utilization subject to three constraints at the same time: data consistency, transaction correctness and transaction deadlines.

It is clear from the literature search that there is a need for efficient scheduling algorithms which integrate the real-time and concurrency control issues for real-time transaction processing. However, there are few known algorithms which address the issue of dynamically scheduling real-time transactions in a uniprocessor environment. In fact, there are no known algorithms when the problem is extended to a distributed environment. Our goal is to develop and analyze scheduling algorithms for real-time transactions in centralized and distributed environments. This goal can be achieved by addressing the scheduling issues in stages: first, it is necessary to develop efficient algorithms for a single node (centralized environment) and evaluate performance of the proposed algorithms using simulation techniques. The performance can be further improved by fine tuning parameters that control the underlying system configuration. Based on the results, the algorithms then need to be developed for a distributed environment where the scheduling issues are significantly different from those in a

centralized environment.

Contribution of Research

We have defined a real-time transaction processing model for a centralized system. An interaction between the various components of the model has also been identified. A set of protocols have been proposed that use a unified approach to maximize concurrency together with meeting real-time constraints at the same time. In order to test the behavior of the model under the proposed protocols, we have developed a real-time transaction processing testbed using discrete event simulation techniques. Preemption, which is not available in the original language, has been implemented and incorporated in the simulation study. We have shown that different protocols work better under different load scenarios and that the overall performance can be significantly enhanced by modifying the underlying system configuration. No such study, except our recent paper [20], has addressed these issues in the past. We have also studied the effect of altering various system and transaction parameters on the overall performance of real-time transaction processing.

For the distributed environment, we have introduced a new concept, namely *real-time nested transactions*, and have proposed protocols with properties that make them very suitable for distributed transaction processing. The proposed protocols have been shown to be free from deadlocks and have tightly bounded waiting period, both of these properties being essential to real-time requirements of a transaction processing system. We have also introduced the concept of priority propagation which, in addition to resolving the priority inversion problem, addresses the issues related to transaction aborts in a nested environment. We have proposed implementation of

this concept by a dynamic status vector that contains enough information to restore priorities. A formal update protocol has been defined for this vector.

Outline of Thesis

A number of important works and developments addressing issues in the area of real-time systems and databases have been studied. We present a review of this literature search in Chapter 2. The review is divided into five significant parts: scheduling in conventional databases, scheduling in real-time databases, the integrated approach, priority inheritance and nested transactions.

We present our model of the real-time transaction processing system in Chapter 3. The various components of the model and their interaction with each other is described. The logical sequence of stages that a transaction may go through during its lifetime together with a pseudo-code is also presented. The priority assignment schemes and the concurrency control protocols are also described in this chapter.

Chapter 4 contains a detailed description of our discrete simulation based testbed that is used to analyze the behavior of various proposed protocols. The simulation parameters, assumptions and simulation results are discussed. The results studying the effect of underlying system configuration (partitioned data, buffer management, multiprocessor environment) and the effect of slack and lock modes are also provided.

In Chapter 5, we present protocols for implementation in a distributed environment. An extension of nested transactions, which are highly suited for distributed processing, is described in the real-time system environment. Several other new concepts and protocols are provided together with a formal proof of correctness of their properties.

A summary of our research together with the conclusions and directions for future research is provided in Chapter 6. Because of the encouraging results obtained, this research can be extended to explore a variety of other related issues.

CHAPTER 2. RELATED WORK

Extensive work has been done in the area of scheduling in conventional database systems (without timing constraints). Similarly, the scheduling problem in real-time systems have been addressed and various algorithms proposed (without addressing the data consistency issues). However, research in the area of real-time transaction processing (which requires a unified approach) is still in its infancy and a few algorithms recently proposed have limited applications because of the restrictive underlying assumptions. As discussed in the previous chapter, real-time systems require dynamic, on-line, adaptive scheduling algorithms which ensure that deadlines of individual tasks are met. Therefore, static scheduling algorithms are not discussed in the following sections.

Scheduling in Conventional Databases

A *transaction* is an execution of a program that accesses a shared database. In order to achieve a better response time, concurrent execution of transactions is desirable. When two or more transactions execute concurrently, their database operations execute in an *interleaved* fashion. That is, operations from one program may execute in between two operations from another program. This interleaving can cause programs to behave incorrectly, or *interfere*, thereby leading to an inconsistent database.

This interference is entirely due to interleaving. That is, it can occur even if each program is coded correctly and no component of the system fails.

A simple way to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial. More precisely, an execution is *serial* if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. From a user's perspective, in a serial execution it looks as though transactions are processed atomically. Serial executions are correct because each transaction individually is correct (by assumption), and transactions that execute serially cannot interfere with each other. However, by not allowing concurrent execution we use the resources poorly and the entire process can become very inefficient. It is, therefore, necessary to broaden the class of allowable executions to include executions that have the same effect on the database as serial ones. Such executions are called *serializable.* More precisely, an execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions. Since serial executions are correct, and since each serializable execution has the same effect as a serial execution, serializable executions are correct too [8].

Concurrency control in database systems is the activity of coordinating the actions of transactions that operate in parallel, access shared data, and therefore potentially interfere with each other. A concurrency control protocol is necessary to determine the permissible interleavings of transaction steps and to resolve conflicts. Concurrency control protocols induce a serialization order among conflicting transactions. A database system scheduler must coordinate concurrent read and write requests in such a way that the resulting sequence of read and write steps is a correct

schedule that preserves database consistency. As discussed earlier, serializability is a widely accepted notion for such correctness.

The serializability theory accepts concurrent executions (schedules) that are equivalent to some serial schedule. There are three attractive properties of serializable schedules, namely, data consistency, transaction correctness and modularity of concurrency control protocols. Under the assumption that transactions are individually consistent and correct, a serializable concurrent execution of transactions lead to correct results and leaves the database consistent. In addition, serializable schedules can be enforced by modular concurrency control protocols [48], that is, protocols can be used to schedule a transaction without reference to the semantics of other transactions. For example, the two phase lock protocol [17] is a modular concurrency control protocol. The modularity of concurrency control protocols is important for large scale transaction facilities where transactions are frequently modified. Consistency, correctness and modularity are attractive properties which account for the popularity of serializable concurrency control in most conventional database systems.

Most concurrency control algorithms fall into one of three basic classes: *locking* algorithms, *timestamp* algorithms, and *optimistic* algorithms.

Locking protocols restrict access to a database object by requiring a lock to be obtained before any read/write can take place. Transactions obtain read locks on objects that they read, and these locks are later upgraded to write locks for the objects that are updated. If a lock request is denied, the requesting transaction is blocked until the lock is released. Read locks can be shared, while write locks are exclusive. Most commercial database systems use the two-phase locking protocol [17] for concurrency control. In classical two-phase locking (2PL) no two transactions

hold conflicting locks at the same time and no transaction obtains a lock after it has released one. The two-phase locking protocol is preferred over other methods for concurrency control due to its simplicity, ease of implementation and integration with a variety of recovery mechanisms.

In the conventional *timestamp* ordering, each transaction, T_i, is assigned a unique timestamp $TS(T_i)$ which is the starting time of that transaction. All the conflicting operations are required to occur in the timestamp order [6]. The basic timestamp ordering works as follows. For each entity O, two values are recorded $TS_r(O)$ and $TS_w(O)$. These values represent the largest timestamp of any read and write operations processed on O, respectively. When a read operation $R_i(O)$ by a transaction T_i is received, the timestamp of T_i is compared with the value $TS_w(O)$. If it is smaller, the operation is rejected. Otherwise, the operation is accepted. Similarly, when a write operation $W_i(O)$ is received, the timestamp value is compared with the value $max(TS_r(O), TS_w(O))$. If it is smaller, the operation is rejected. Otherwise, it is accepted. To increase the degree of concurrency, multiple versions of item values have been used as a variation of timestamp ordering [7, 33]. A general concurrency control algorithm is described in [18] and it is argued that two-phase locking and timestamp ordering are special cases of this more general algorithm.

In classical *optimistic* concurrency control [26], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. A transaction is restarted at its commit point if it fails its validation test. This test checks that there is no conflict of the validating transaction with transactions that committed since it began execution. The advantage of optimistic concurrency control is that it can increase concurrency and throughput by eliminating bottlenecks and

needless serialization. The potential disadvantage of the optimistic method is more work to redo, because conflicts are not discovered until a transaction has finished processing and attempts to commit.

Scheduling in Real-Time Systems

A brief survey of the static and dynamic scheduling algorithms proposed for hard real-time systems is provided in [14]. The survey shows that, for both centralized and distributed systems, the static scheduling approaches are inflexible and cannot be efficiently applied to the dynamic scheduling problem. Most of the algorithms which are optimal for static scheduling are not optimal for dynamic scheduling. In particular, for multiprocessor systems, there can be no optimal algorithm for scheduling preemptable tasks if the arrival times of tasks are not known *a priori* [30]. Furthermore, because run-time cost is an important factor for dynamic scheduling, most sophisticated static algorithms are not appropriate for dynamic scheduling. Because of these reasons, heuristic algorithms become important to dynamic scheduling problems.

A heuristic function and a backtracking scheme for scheduling nonpreemptable tasks with resource constraints is presented in [54]. A set of simple and integrated heuristic functions using multiple resources is provided in [53]. It is found that none of the simple heuristics work well. However, using an integrated heuristic together with a limited number of backtracking, the success ratio of their search algorithm for scheduling tasks is shown to be as high as 99.5% of that of an exhaustive search algorithm.

Stankovic and Ramamritham have proposed several dynamic algorithms for scheduling tasks in a real-time distributed system [35, 36, 37]. Their work is fo-

cused on obtaining best schedules for a number of nonpreemptable tasks that may arrive at different intervals. Since the algorithms consider tasks as the scheduling entity without considerations for the data access patterns, data consistency problems have not been addressed. Their goal is to generate schedules which maximize the number of tasks that can be guaranteed to complete before their deadlines. A local scheduler determines whether or not a newly arrived task can be scheduled locally. If a task cannot be *guaranteed* locally, the global scheduler is invoked. A *bidding* algorithm and a *focused addressing* algorithm is proposed for global scheduling. In bidding, the task is sent to a remote node which is selected based on the bids received for the task. In focused addressing, the task is sent to a remote node that is estimated to have a high surplus processing time. Scheduling decision made in focused addressing is based on inaccurate state information, but it entails low communication delay. The communication delay involved in bidding is high, but the selection is based on relatively accurate state information of nodes. A combined algorithm that uses both schemes has also been proposed. Simulation results of this algorithm are reported in [36, 46]. A major restriction in their work is that all research is focused on non-preemptive scheduling and is applicable to hard real-time systems only. Non-preemptive scheduling is not appropriate in a database setting since long transactions and I/O cause unnecessary blocking to other transactions thus preventing them from meeting their deadlines. However, it is indicated in [14] that their approach is being extended to preemptable tasks.

An algorithm to consider precedence constraints in dynamic scheduling is proposed in [13]. However, tasks are considered to be arriving in *groups* with each group having the same deadline. For a group that must be distributed, their approach at-

tempts to partition tasks in the group into subgroups and distribute the subgroups in the network to be scheduled in parallel. Tasks in a group are scheduled to run either completely or not at all. The algorithm that combines focused addressing and bidding is used to determine how to distribute the subgroups in the network.

Real-Time Database Scheduling

In 1989, Buchmann [9] formally presented the idea of integrating real-time scheduling and concurrency control for time-critical database scheduling. The incompatibility of assumptions in real-time scheduling and database scheduling has been identified and a framework is proposed to combine the two approaches. The framework consists of two major components: the task model and the scheduling behavior model. The task model describes timing information about transactions and defines resource requirements. The scheduling behavior model addresses metrics, correctness criteria, static vs dynamic scheduling, conflict-management policy and overload-management policy. Some existing algorithms for time-critical scheduling and concurrency control are then mapped on to the proposed framework thereby identifying the shortcomings of the two approaches. The existing combined algorithms for time-critical database scheduling are also discussed. No new algorithm or performance comparison of existing algorithms is given.

Huang [23] also emphasizes on the necessity for an integrated approach and identifies several shortcomings in the current work. A centralized testbed is used for evaluating a set of integrated protocols that support real-time transactions. Using a two-phase locking protocol for concurrency control, several algorithms for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup,

and transaction restart are developed and evaluated. Criticalness and deadline are treated as two independent characteristics of real-time transactions, but combined into a value function, $v(t)$, for a transaction. The idea behind value function is to determine the importance of the completion of a transaction which has missed its 'deadline'. In case of hard deadlines, there may be no value in completing a transaction that has missed its deadline and such a transaction should be aborted. However, for soft deadlines the value of completing a transaction may reduce because of the penalties for missing deadline. The performance results illustrate the importance of CPU scheduling and the use of deadline and criticalness information in conflict resolution protocols. It is also shown that overheads such as locking and message communication are non-negligible and can not be ignored in real-time transaction analysis. The results are claimed to be the first experimental results for real-time transactions on a testbed system.

Locking protocols have also been used in real-time applications despite the fact that the blocking behavior of these protocols can greatly degrade the performance of real-time database systems. Sha et. al. [41] proposes a locking-based protocol that avoids the blocking of high priority transactions for at most the duration of a single embedded transaction. An optimistic priority-based locking mechanism that dynamically adjusts the serialization order of active transactions in order to reduce blocking has been proposed by Lin [27]. Agrawal et. al. [5] proposes a new variant of the locking approach for real-time databases with firm deadlines. Their approach is to exploit any available slack in a transaction to improve the overall performance of the system by decreasing the number of transactions that miss their deadlines. A new relationship between locks called ordered sharing is used to eliminate blocking.

Ordered sharing has the desirable property of eliminating blocking of read and write operations at the expense of a possible delay at transaction commitment. The proposed protocol works well for medium loads, but does not exhibit any advantage at very high load. Further, the protocol does suffer from the possibility of cascading aborts unless versions of the objects are maintained. The authors claim that this does not result in a large overhead because such versions are maintained anyway for recovery purposes.

Comparative studies of locking and optimistic algorithms for a conventional database system [4, 10] have generally shown that locking provides significantly better performance than optimistic concurrency control. However, in some recent performance studies [21, 22] it is shown that some variants of the optimistic protocol [26] outperform two-phase locking in real-time databases. The authors point out that transaction blocking in the two-phase locking protocol results in unpredictable delays causing transactions to miss their deadlines. In contrast, transactions in the proposed optimistic protocols neither block nor suffer from wasted restarts. The simulation results presented in [22] indicate that under conditions of low data contention, delaying the validation of lower priority transactions result in improved performance. However, the priority wait mechanism result in significant performance degradation at high contention levels because of generating a high number of data conflicts. A simple *wait control* mechanism consisting of a 50 percent rule is proposed to address this problem. According to this rule, if half or more of the transactions conflicting with a transaction are of higher priority, the transaction is made to wait; otherwise, it is allowed to commit. Huang [23] also developed locking variants of the concurrency control protocol and compared its performance with the class of two-phase locking

protocols for real-time databases. Some simulation results are provided in [21, 22].

Abbott and Garcia-Molina have proposed several algorithms for scheduling real-time transactions. Their model is based on dynamic scheduling in a uniprocessor environment and is perhaps the most complete model presented in the literature so far. In their earlier work [1, 2], the database is assumed to be memory-resident. This condition is relaxed in [3] and a disk-resident database is modeled. However, a new concept of main-memory buffer pool has been introduced. The buffer pool is assumed to be large enough to hold all the modified pages and the changes are written back to the disk only when the transaction commits. This alleviates any clean-up or roll back that may otherwise be required in case the transaction aborts.

Both shared and exclusive locks on data objects are considered in [3] as opposed to using only exclusive locks in the earlier work. This requires protocols to resolve conflicts among two or more transactions. Concurrency control is achieved by ensuring serializability and is enforced via locking protocols that allow for shared and exclusive locks on the data objects. The data object assumes the same priority as that of the transaction holding the lock on it. When several transactions hold the lock, the priority of data object is the highest of all priorities of the lock holders. Simulation results are provided for the various schemes proposed to resolve conflicts and to avoid priority inversion.

In [1], two components of the scheduling algorithms have been proposed: assigning priorities to the incoming transactions and a concurrency control mechanism for conflict resolution. In [2], another component is added to screen out the transactions which are not eligible at the time scheduler is invoked. However, this component is dropped in [3] and all transactions are considered eligible with priority of tardy

transaction increasing with time, that is, all transactions must be executed eventually. Because of the disk-resident data, it is also suggested that the scheduling of I/O requests should consider transaction priorities (as opposed to the disk-scheduler policies which are based on minimization of disk-head seek time). Based on the schemes for assigning priorities, managing concurrency, and I/O scheduling, 24 different combinations have been studied in the simulation representing a high-load scenario. Although some policies are better than others under different load scenarios, it is concluded that there is no significant difference between the various scheduling options.

Son et. al. [16, 27, 44, 45] have also addressed the issues of scheduling and consistency for *hard* real-time database systems. The authors argue that, unlike other current work, the integration of the two issues should not be based on existing concurrency control methods because these methods synchronize concurrent data access by the combination of two measures: blocking and roll back of transactions, both of which are barriers for time-critical scheduling. A priority-based concurrency control method which employs a priority-dependent locking protocol has been presented for a uniprocessor environment [27]. This method has a flavor of both locking and optimistic approach. By dynamically adjusting the serialization order, it is ensured that high priority transactions are never blocked by an uncommitted lower priority transaction while low priority transactions may not have to be aborted even in face of conflict with high priority transactions. It is claimed that the proposed protocol incurs less blocking and aborts as compared to the conventional two phase locking. No performance results are provided.

Priority Inheritance

Priority inversion is said to occur when a high priority transaction must wait for the execution of lower priority transactions [24, 40]. As illustrated by Example 1.2 in Chapter 1, this waiting period can become very large and ultimately result in missed deadlines.

A common approach to bound such arbitrary delays is to use *priority inheritance* protocols [24, 39, 40, 41]. The basic idea of priority inheritance is that when a transaction T blocks higher priority transactions, the transaction T is executed at the highest priority of all transactions blocked by T. When the blocking is over (say, by releasing locks), T returns to its original priority level. A real-time concurrency control protocol which addresses the issue of priority inversion in a distributed real-time database is described in [39]. A more detailed study of two priority inheritance protocols for real-time synchronization is provided in [40]. The transactions are assumed to consist of critical sections with preemption not allowed when a transaction is executing within its critical section. The critical sections can be properly nested. A basic protocol is proposed and it is shown that this protocol bounds the blocking period to at most the duration of one critical section for each lower priority job involved in the blocking. However, the basic protocol has two problems: it does not prevent deadlocks and that blocking duration for a job, though bounded, can still be substantial because a chain of blocking can be formed. To overcome these problems, a *priority ceiling protocol*, another type of a priority inheritance protocol, is proposed. In this protocol, each object is assigned a priority ceiling which is the priority of the highest priority job that will access this object. Transactions are then allowed access to the object only if their priority is higher than the priority ceiling of all the locked

objects. In case of blocking, the priority is inherited just as in the basic protocol. It is shown that this protocol does not suffer from the problems of the basic protocol, that is, the priority ceiling protocol prevents deadlocks and reduces the blocking to at most one critical section. However, note that assigning a priority ceiling to each object requires a static analysis of the data access patterns of all transactions. In other words, this scheme requires prior knowledge about the data objects to be accessed by each transaction.

In [41], the simple priority ceiling protocol is extended to a read/write priority ceiling protocol. This protocol integrates the two-phase locking protocol with priority-driven real-time scheduling and allows for both read and write locks. The objects are now assigned three types of priority ceilings: *write-priority ceiling* which is the priority of the highest priority task that may write the object, *absolute priority ceiling* which is the priority of the highest priority task that may read or write the object, and *r/w priority ceiling* which is a dynamically assigned priority depending upon the locking mode of the transaction. Under the r/w priority ceiling protocol, a transaction cannot acquire a lock unless its priority is higher than all the r/w priority ceilings of the data object locked by other transactions. The protocol is shown to be free of deadlocks and bounds the blocking period to at most the duration of a single embedded transaction of a lower priority task.

A variation of the simple priority inheritance protocol, namely *conditional priority inheritance* is provided in [24]. The basic idea behind this scheme is the following. When priority inversion is detected, if the low priority transaction is near completion, it inherits the priority of high priority transaction, thus avoiding an abort with its waste of resources; otherwise, the low priority transaction is aborted, thereby avoid-

ing the long blocking time for the high priority transaction, and also resulting in reduced waste of resources used thus far by the low priority transaction. Simulation results show that this scheme works better than the simple priority inheritance or the low priority abort protocols.

Some problems resulting from priority inversion in real-time communication are addressed in [50] and solutions developed for a distributed real-time operating system are presented.

Nested Transactions

Nested transactions are an extension of traditional atomic transactions and have a hierarchical grouping structure. They permit safe concurrency within as well as among transactions and also enable transactions to fail partially in a graceful and controlled manner. These two properties make nested transactions very suitable for distributed environment.

A lock-based concurrency control and recovery mechanism for nested transactions is provided in [31]. This is the first design of nested transactions that uses locking for synchronization. Algorithms are presented for locking, state restoration, distributed transaction management and distributed deadlock detection. Each transaction is required to acquire a lock on the object that it intends to access. This ensures that the object becomes inaccessible to other conflicting transactions while the transaction holding the lock is accessing it. Read locks are allowed to be shared whereas write locks can only be acquired in an exclusive mode. Locks are allowed to be shared (in any mode) with the ancestors. A parent transaction inherits the locks of a committing child transaction. This ensures that the effects of the child trans-

action are not seen by other transactions until the root transaction commits. When a transaction holds a write lock, state restoration information is created and later used to restore the objects in case of transaction aborts. Like lock inheritance, when a transaction commits, each of its associated states is also offered to the committing transaction's parent which may accept the new state if it does not already have one for each of the accessed objects. If the parent already has an associated state for the object, then it takes precedence because it is earlier than the child's state. The deadlock detection algorithm is based on tracing wait-for graphs for cycles. Transactions are assumed to have permanently assigned priorities and when a cycle is found, the lowest priority member of the cycle is aborted to break the deadlock.

The first comprehensive design of a nested transaction system based on timestamp ordering and multiple object versions is presented in [38]. It is the combination of pseudo-times and dependent commit records that implement nested transactions. The design uses pseudo-times (which are essentially timestamps) for synchronization. These timestamps readily resolve (avoid) deadlocks though it may eventually result in many needless transaction aborts. The notion of multiple versions of objects is also introduced. The basic scheme proposed permits transaction starvation because a transaction request that performs updates can be aborted every time it is run. If a transaction reads any objects written later by another transaction with an earlier pseudo-time (i.e., the read and write are attempted in an order opposite to that of the pseudo-times for the transactions), the transaction with the earlier pseudo-time will abort because it cannot acquire needed objects at its particular pseudo-time. To solve this problem, preallocations of resources in the form of token reservations (a deadlock prevention technique) is proposed. Note that token reservations require

a certain amount of predictability, and can reduce concurrency. This model is further augmented by [49] to provide for resilient distributed computing. Like many timestamp based schemes, this scheme also relies heavily on stable storage.

Summary

In this chapter, we have provided a review of the related work in scheduling, priority inheritance and nested transactions. It has been shown that the scheduling issues in conventional databases and traditional real-time systems are treated differently and have been studied separately. For real-time database systems, an integrated approach is required which should address both database consistency and timing constraints at the same time. A need for such integration and an outline of a common framework has been emphasized in recent research. Several algorithms with severe limitations have been proposed in the past and further research is being actively pursued. In Chapter 3, we present our proposed model that uses a unified approach for scheduling real-time transactions.

A survey of work in the area of priority inversion problem and nested transactions is also presented. The priority inversion problem is now well understood, but there are not many solutions available. The idea of nested transactions have been around for quite some time, but there has been no research done in the real-time transaction processing environment. We have addressed this issue in Chapter 5.

CHAPTER 3. THE REAL-TIME TRANSACTION PROCESSING (RTP) MODEL

Our real-time transaction processing model is a priority-driven preemptive scheduling model. In this chapter, we first define the various priority assignment schemes that are selected, followed by the concurrency control protocols and then describe the model in detail.

Priority Assignment Schemes

Each transaction that enters the system is assigned a priority to identify its importance in scheduling considerations. A transaction executes at its assigned priority as long as it stays in the system. The priority of a transaction is used for resolving data conflicts and for implementing the concurrency control schemes. When a transaction waits, this priority is used to maintain the priority queue. The priorities can be assigned in a number of ways, but we selected the following four traditional schemes for our study [2, 3].

First-Come-First-Served (FCFS)

This is the simplest priority assignment scheme in which the priorities are assigned according to the arrival time of the transactions. The highest priority is

assigned to the transaction that arrived first. The main disadvantage of this priority assignment scheme is that it does not take advantage of deadline information. FCFS will discriminate against a newly arrived transaction with an urgent deadline in favor of an older transaction which may either have a much later deadline or has already missed its deadline. This is not very desirable for real-time systems, but we study FCFS for comparison with other schemes.

Earliest Deadline First (EDF)

In EDF, a transaction whose deadline is the closest is always assigned the highest priority and the one with the farthest deadline is assigned the lowest priority. This priority assignment scheme is known to result in a better performance in real-time systems under low and moderate load levels. However, a disadvantage of EDF is that it can assign the highest priority to a task that is about to miss its deadline. When this is done, the system allocates resources to a transaction which cannot meet its deadline in favor of a transaction which could meet its deadline. Thus EDF may not be desirable for heavily loaded systems. An overload management policy as described in [2] can be used to solve this problem. This policy screens out transactions that have missed or are about to miss their deadlines.

Minimum Slack Time First (MSTF)

The slack time of a transaction is the cushion that can be absorbed by unpredictable delays (waiting in queues, etc) without missing the deadlines. More formally, if t_{arr} is the arrival time, E is the estimated execution time and d is the deadline, then the slack time at the time of arrival of a transaction is $S = d-(t_{arr}+E)$. If the

transaction has already received some service, say R, then $S = d - (t_{curr} + E - R)$, where t_{curr} is the current time. A negative slack time implies that the transaction has either already missed its deadline or when it has been estimated that it cannot meet its deadline. In MSTF, the transaction which has the least positive slack is assigned the highest priority because such a transaction will have a high probability of missing its deadline.

Note that the slack time of a transaction changes as the time passes. In other words, the priority of a transaction increases as the slack decreases. Therefore, with MSTF, the static evaluation of priorities may not be very accurate. It may therefore be necessary to recalculate slack and reevaluate the priority from time to time. However, this reevaluation can incur significant overhead if done too frequently.

Shortest Job First (SJF)

In SJF, the shortest job is assigned the highest priority because it will have a high probability of meeting its deadline. This scheme has not been used in real-time scheduling in the past, but in CPU scheduling it has been shown to be optimal in that it gives the minimal average waiting time for a given set of jobs [34]. We study this scheme for comparison purposes.

Concurrency Control Protocols

Since we allow concurrent execution of transactions we need a concurrency control mechanism to order the updates to the database in such a way that the resulting schedule is serializable. Shared locks permit multiple concurrent readers. Three concurrency control protocols (blocking, high-priority preemption and conditional

preemption) have been selected for this study. In the following, let T_r be a transaction requesting a lock on a data object O, that is already locked by a transaction T_h. The examples in this section are due to [2] and illustrate the requirement for real-time concurrency control protocols. In these examples, it is assumed that the estimates are exact and the time required to make scheduling decisions or roll back transactions is ignored.

Blocking (BLOCK)

In *blocking*, T_r always blocks and waits for the lock to be released by T_h. This is the standard method for most database management systems which do not execute real-time transactions. The approach is simple but contrary to real-time requirements and also results in priority inversion. All conflicts are handled identically and the concurrency control mechanism makes no effective use of transaction priorities. It has been implemented for use as a baseline scheme for comparison with other algorithms.

High Priority Preemption (HIPRTY)

In *high-priority* preemption, if T_r's priority, $P(T_r)$ is higher than the priority of T_h, $P(T_h)$, then T_h is preempted. That is, conflicts are always resolved in favor of the higher priority transaction. The only exception is in the case when priorities are assigned using minimum slack because a preempted transaction can assume a much higher priority immediately after preemption because of the lost service time and a reduced slack. We have avoided this problem by considering the priority after a transaction aborts, $P(T_h^a)$. Note that for FCFS, EDF and SJF schemes, $P(T_h) = P(T_h^a)$, so it does not matter if we use this modified scheme or do not consider the priority

after abort. This conflict resolution policy can be written as follows:

if for any T_h

$$P(T_r) > P(T_h) \textbf{ and } P(T_r) > P(T_h^a)$$

then preempt and roll back each T_h

else T_r blocks.

endif

Example 3.1 Consider the set of transactions given in Table 3.1. In this table, t_{arr} is the arrival time of the transaction, E is the estimated execution time, d is the deadline. The variable(s) updated by the transaction are also listed.

Table 3.1: Parameters for Example 3.1

Transaction	t_{arr}	**E**	**d**	**Updates**
A	0	2.6	5	X
B	1	2.0	4	X
C	2	2.4	8	Y

Note that transaction A and B both update item X. Therefore, these transactions must be serialized. The schedule produced by using EDF to assign priority and HIPRTY to resolve conflicts is given in Figure 3.1a. In this schedule, A runs in the first time unit during which it acquires a lock on item X. Transaction B preempts A at time 1 (because of an earlier deadline) and requests a lock on item X. Thus a conflict is created which is resolved by rolling back A thereby freeing the lock on X. Transaction B continues processing and completes before its deadline. After B

completes at time 3, A is restarted. Transactions B and C meet their deadlines but A is tardy.

However, if the MSTF scheme was used to assign priorities then, after preemption, A could have immediately assumed a higher priority because of the reduced slack (Figure 3.1b). Upon arrival, slack of A, $S_A = 2.4$, and slack of B, $S_B = 2.0$. At any time t_{curr}, if U is the amount of service already received by a transaction, the slack can be calculated as $S = d - (t_{curr} + E - U)$. Now when a conflict occurs at time 1.5, B has a slack of 1.0 but A, if it were aborted, has a slack of 0.9. So, A is not aborted but assigned to the processor while B waits for A to finish. Transaction B is unblocked when A finishes at time 3.1. Transactions A and C meet their deadlines but B is tardy.

Conditional Priority Preemption (CPR)

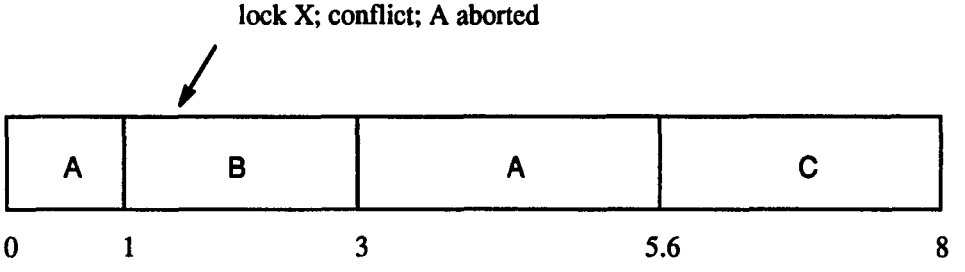
The concurrency control using HIPRTY results in better performance, but can be too conservative at times. This is because, depending upon the allowable slack, a newly arrived high priority transaction may be able to wait while the lower priority transaction completes. In *conditional priority* preemption, if T_r's priority is higher than the priority of T_h, then it is first evaluated whether or not the requesting transaction can wait without missing its deadline. If yes, T_h is allowed to complete; otherwise, T_h is preempted. This modification yields the following algorithm:

if $P(T_r) > P(T_h)$ **and** $P(T_r) > P(T_h^a)$

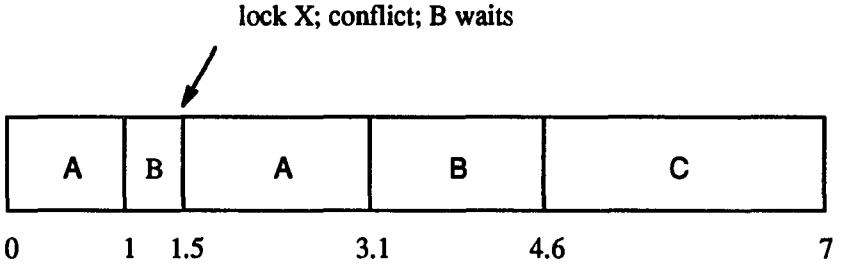
then

if T_r can wait

then T_r blocks



(a) HIPRTY with EDF



(b) HIPRTY with MSTF

Figure 3.1: Concurrency Control using HIPRTY

else preempt and roll back T_h

endif

else

T_r blocks

endif

We only implement CPR if the conflict is one-to-one, that is, there are no multiple readers involved in the conflict. If there are multiple readers involved then to determine a maximal subset of readers all of which can finish within the slack of the T_r is an NP-complete problem [3]. Further, we use CPR if T_r conflicts with exactly one transaction. In other words, if the locks required by T_r are held by multiple other transactions, then T_r always blocks.

The following example illustrates the idea of this policy (CPR). In this example also, it is assumed that EDF is used to assign the priorities, estimates are exact and scheduling decisions and rollbacks are done instantly.

Example 3.2 Consider the set of transactions given in Table 3.2. The resulting schedule is given in Figure 3.2. A conflict occurs when B requests a lock on X at time 1.5. The slack time for B is calculated as $S_B = 4 - (1.5 + 2.0 - 0.5) = 1.0$. This equals exactly the remaining run time for A. Therefore, B waits for A to finish and release its locks. A finishes at time 2.5 and B, with 1.5 time units left to compute, regains the processor and completes at time 4. All transactions meet their deadlines.

Note that the above example is a simple case illustrating the idea of the protocol. In case of chained blocking or incorrect estimates, the protocol may behave differently. As described in the next chapter, we have considered issues related to this problem

Table 3.2: Parameters for Example 3.2

Transaction	t_{arr}	E	d	Updates
A	0	2	5	X
B	1	2	4	X
C	2	3	8	Y

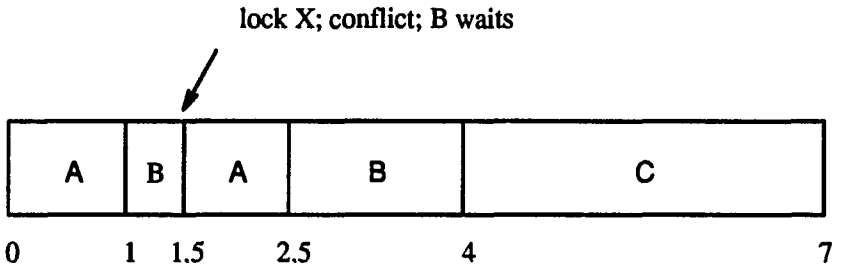


Figure 3.2: Concurrency Control using CPR

during the implementation.

The RTP Model

There are a number of considerations and choices in selecting an underlying configuration for a real-time transaction model. These include number of processors, location of database, distribution of data and size of main memory. Our real-time transaction processing (RTP) model consists of one or more processors, a disk-resident database and a main memory buffer pool. The first part of this study is based on a uniprocessor environment, but we also study the effects of having multiple processors in this centralized environment. A centralized system is one in which the processors are located at a single point in the system and the inter-processor communication cost is negligible compared to the processor execution cost. A multiprocessor system with shared memory is an example of such system. In contrast, a distributed system is one in which the processors are distributed at different points in the system and the inter-processor communication cost is not negligible compared to the processor execution cost. A local area computer network is an example of such system. In distributed systems, interprocessor communication cost is an important factor which must be explicitly taken into account in scheduling. For our study of the effect of multiple processors, we implemented a centralized system with a shared memory organization.

The database can be assumed to reside in the main memory or on secondary storage. By assuming the entire database to be memory-resident as in [3], the study is simplified because of the elimination of the necessity for modeling of disks or I/O scheduling. However, for our study, we assume a disk-resident database because it

represents a more realistic view of the data. Furthermore, the database may either reside on a single disk or may be partitioned on two or more disks where each disk has its own service queue. The performance is studied in both of these cases. The unit of database granularity is considered to be a *page* and the transactions access a sequence of pages to read/write data objects. The pages may be locked in a shared or an exclusive mode. We assume that the buffer pool is large enough so that a modified page need not be written to the disk until after the transaction commits. Thus, aborting a transaction involves no disk writes.

The transactions are characterized by timing constraints, computation requirements, and data requirements. The *timing constraints* can either be modeled by a value function, $v(t)$, or by simply considering a single deadline. Currently, we assume that all transactions maintain a priority which is not a function of a transaction being tardy and, therefore, the 'value' of a transaction to the system is not considered as a parameter. The *computation requirements* of a task are characterized by an estimated execution time, after the task becomes ready for execution. The *data requirements* are not known in advance and, therefore, it becomes necessary to have some dynamic concurrency control mechanism to resolve conflicts based on the task's data access pattern during the runtime.

The RTP model consists of four basic components: the source module, transaction manager, concurrency control manager, and the resource manager. Figure 3.3 shows the block diagram illustrating the interaction between various components of the model. Each of these components is explained in more detail in the following sections. All queues are maintained as priority queues.

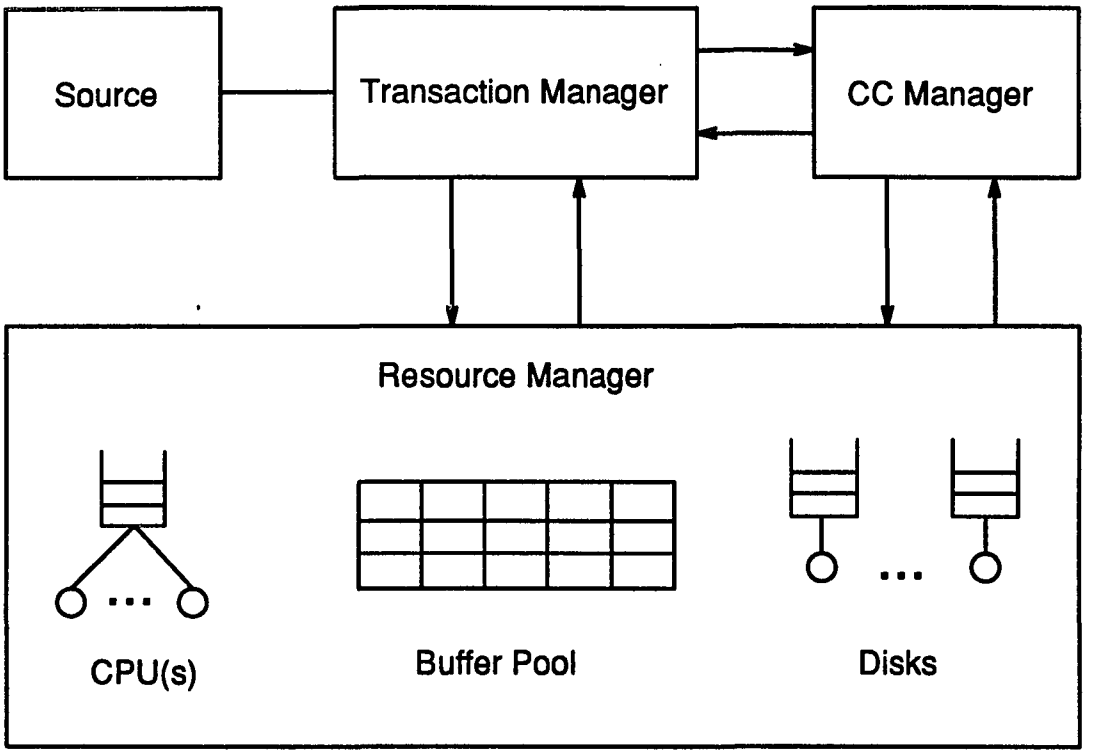


Figure 3.3: Module Interaction

Source Module

The *source module*, responsible for modeling the external workload, generates external tasks with certain known characteristics and sends these tasks to the Transaction Manager. The transactions are assumed to arrive to the transaction manager with a *ready time* (usually the same as the arrival time), an estimated *execution time* and a *deadline*. The deadline is actually computed on the basis of slack time, as

$$deadline = ready\_time + execution\_time + slack\_time$$

The slack time determines the tightness or looseness of deadline because smaller the slack time, closer is the deadline. The data access pattern of the transaction may not

be known in advance.

Transaction Manager

The *transaction manager* is responsible for accepting transactions from the source module and modeling their execution. Based on the arrival time, execution time and deadline information, the *transaction manager* assigns a priority to each arriving transaction according to a previously selected protocol (FCFS, EDF, MSTF, or SJF). After arrival in the system, the transaction waits for all resources to become available. It first requests the locks on pages that may be accessed (read or updated) when the transaction executes.

An I/O request by a transaction begins with a concurrency control request to get access permission. Once the request is granted, a read request for the page is sent to the resource manager. The buffer control agent of the resource manager then determines whether or not the page is in the main memory. If it is, no I/O is required; otherwise, a disk I/O to read the page is scheduled. This might also require flushing of a dirty page. Once the requested pages are available in the buffer pool, CPU is requested for processing. After processing, another I/O request may be made for writing the modified pages to the disk.

The transaction manager also controls the load of the system by limiting the number of concurrently active transactions. If a new transaction is received by the transaction manager when there are already maximum allowed transactions present in the system, the new transaction is queued in a system queue until one of the active transactions completes execution. When a transaction completes, one of the waiting transactions in this system waiting queue is activated.

Concurrency Control Manager

The *scheduler* component of the *concurrency control (CC) manager* uses the transaction's priority in conjunction with the concurrency control protocol to determine whether the transaction can be immediately scheduled or needs to wait. The CC manager also determines whether an active transaction needs to be preempted in favor of the newly arrived transaction; in the latter case, the preempted transaction releases the resources, rolls back and joins the waiting transactions. Note that this preemptive-restart scheme is not a 'total loss' because the transaction being preempted has already transferred the requested page into the buffer thereby reducing the I/O service time for the transaction initiating the preemption.

The scheduler is invoked whenever a transaction commits or a new transaction arrives; the concurrency control mechanism is invoked whenever a conflict arises (Figure 3.3). Our primary goal is to minimize the number of transactions that miss their deadlines. For transactions which do miss the deadlines, the goal is to minimize the mean tardy time. We assume that once a transaction enters the system, it will complete its execution even after it becomes tardy, that is, no transaction is aborted because of missing its deadline. This implies that we assume soft deadlines in contrast to hard deadlines where there is no value completing a transaction that has missed its deadline and such transactions should be aborted.

The data consistency is maintained by ensuring that the execution schedule is serializable. Serializability has been chosen as the way to achieve consistency because we assume very little *a priori* knowledge about the transactions. In our model, serializability is enforced by using the two-phase locking protocol. During its lifetime, a transaction accesses pages for read and (possible) write after some

computation. In order to access a page, the transaction acquires a lock on the page; the lock is released when either the transaction is preempted or when it commits. The read locks can be shared but the write locks can only be obtained in an exclusive mode. If the requested page exists in the buffer, it is locked and accessed from there; otherwise, an I/O request is initiated to transfer the page to the buffer.

Resource Manager

The *resource manager* controls the physical resources such as the CPU, disks and buffer pool. It provides service to both the transaction manager and the concurrency control manager. The entire database can reside on a single disk or it may be partitioned on multiple disks. In the latter case, the disks may either be serviced by a single queue (similar to the tellers and customer queue in a bank) or each disk may have its own queue. For partitioned data, it is more suitable to have a service queue for each of the disks. We have studied the performance of the system using both a single disk and multiple disks (partitioned data). In the case of multiple disks, we assumed that each disk has its own service queue.

The buffer pool consists of a set of page frames. A dirty flag is associated with each page frame. This flag is used to determine which pages need to be flushed in case a new page has to be brought in. When a request to access a page is received from the transaction manager, the buffer manager checks whether the requested page is in the buffer pool. If it is present, no disk I/O is required. If the page is not found, the buffer manager searches for a non-dirty page using a *least recently used (LRU)* policy. If such a clean page is available, a disk I/O is scheduled to read the requested page into the buffer frame occupied by the replacement victim. If no clean page is

found, a disk write is scheduled to write back the least-recently-used dirty page in addition to the read that is scheduled for the requested page.

Logical Structure of the RTP Model

Figure 3.4 shows a logical sequence of stages that a transaction may go through after entering the system. All queues indicated on this diagram are priority queues. An arriving transaction may have to wait in a system queue if the number of active transactions has already reached a predefined maximum. After a transaction enters the system, it is considered active and can take one of the two paths: if the transaction is unable to acquire all resources (locks), it is transferred to *lock queue* for later consideration when some locks are released; if all the resources are available or if the transaction is able to preempt another transaction (on the basis of a concurrency control protocol), the transaction manager interacts with the concurrency control manager and the resource manager to bring in the pages requested by the transaction and preempt another transaction, if necessary (*read phase*). The preempted transactions release the resources (locks), roll back and return to the lock queue. Since none of the pages have been modified yet, there is no need to undo any work. After the I/O is scheduled and the pages are brought into the buffer pool, the transaction moves to the CPU queue. There may be one or more CPUs servicing this queue and depending on the transaction's priority, it gets the service when the CPU(s) become available (*processing phase*). After completion of processing, another I/O request is made for writing back the updated pages to the disk (*write phase*). The updated pages are written back to the disk but not flushed out of the buffer pool which results in an improved performance using the buffer management. This

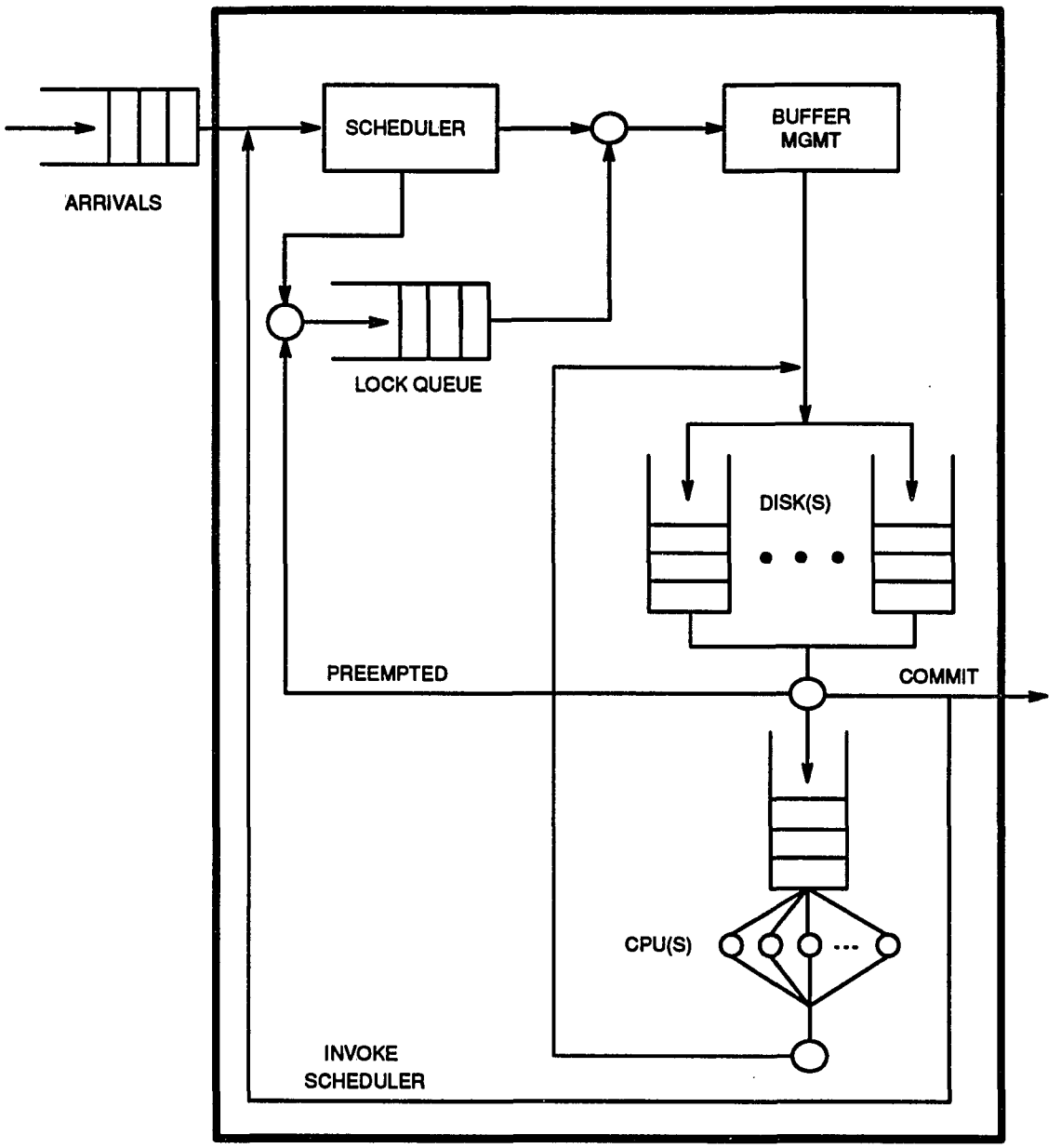


Figure 3.4: Logical Sequence of Transaction's Lifetime

improved performance comes from the fact that when another transaction wants to access the same page, it will find the updated page in the buffer and does not have to initiate an I/O request. Also note that we do not allow preemption in the processing or the write phase because of a possible waste of resources since the transaction has already received much service time. Further, this also implies that once the pages have been processed and modified, a transaction will not be preempted thereby eliminating the necessity of undoing a transaction's actions if it were allowed to be preempted at this stage. Upon completion, the transaction commits and exits the system. At the same time, the scheduler is invoked so that any waiting transactions may be scheduled.

Figure 3.5 illustrates the model using a pseudo code for the high priority preemption (HIPRTY) concurrency control scheme. A newly arrived transaction, T_{new}, is assigned a priority after which it is determined whether T_{new}'s execution will conflict with other transactions. If there are no conflicts, the transaction requests resources and begins execution by initiating an I/O request for pages that are not already in the buffer. Once all the required pages are brought into the buffer, processing is initiated followed by another I/O request for writing updated pages to the disk. The transaction then exits the system. In case of conflicts, however, it is examined whether the priority, P_{new}, of this newly arrived transaction, T_{new} is higher than the priority of all conflicting transactions. If P_{new} is the highest priority, then it is determined whether the conflicting transactions can be preempted or not. Recall that a transaction past its read phase is not eligible for preemption regardless of its priority. If both of these conditions are satisfied, then all conflicting transactions are preempted and T_{new} begins execution. Otherwise, T_{new} waits in the lock queue

```
        P_new = assign_priority(arrival time, execution time, deadline)
'start': if no conflict exists
                request resources;
                begin_execution;
          else
                for all conflicting transactions, T_i
                      check if P_new > P_i  ∀i, AND
                          T_i can be preempted;
                endfor;
                if P_new is the highest priority and
                  preemption is possible,
                      preempt all conflicting transactions;
                      begin_execution;
                  else
                      wait in the queue with priority P_new until
                                invoked by the scheduler;
                      goto 'start';
                endif;
        endif;


        begin_execution
                check if the needed pages are in buffer;
                read missing pages using LRU policy for
                    buffer management;
                do processing;
                write modified pages to the disk;
        end begin_execution;
```

Figure 3.5: Structure of the Transaction

with priority P_{new} until invoked by the scheduler. When a transaction is invoked by the scheduler, it once again checks for conflicts and proceeds as described above.

Summary

We have proposed a priority-driven preemptive scheduling model to be used as a testbed for evaluating the performance in a real-time transaction processing system. The transactions arriving in the system can be assigned priorities using various schemes and the consistency can be maintained by a variety of concurrency control protocols. We have identified four priority assignment schemes (FCFS, EDF, MSTF and SJF) and three concurrency control protocols (BLOCK, HIPRTY and CPR) for study. A description of various components of the model and their interaction has also been provided. The logical structure of the transaction processing system outlines the stages that a transaction may go through after entering the system. The read, processing and write phases of the transaction have also been identified. The actions taken for the transaction are further explained using pseudo code for the HIPRTY concurrency control scheme. In Chapter 4, we provide the implementation details of our RTP model and the results obtained from this testbed.

CHAPTER 4. SIMULATION MODELING

In order to evaluate the performance of the proposed protocols under a variety of load scenarios and system parameters, it is necessary to develop a real-time transaction processing testbed. SIMSCRIPT II.5, a general purpose discrete event simulation language, is used to develop a comprehensive simulation model so that the proposed protocols can be tested and analyzed for performance comparison. SIMSCRIPT has been selected because of its ease of use, reliability, portability and flexibility that allows a great control over the processes and events. The language also supports graphics interface to monitor the progress of the simulation. When required, user-written C and Fortran routines can also be linked to the simulation program. The SIMSCRIPT language is well-suited for modeling centralized systems, but may not be ideal for modeling a distributed environment.

The results provide a performance comparison among the algorithms under different system loads and configuration. This chapter outlines the underlying assumptions, describes the simulation model and presents the observations made during the simulation experiments.

Assumptions

The following assumptions are inherent in the study of our proposed RTP model.

- We assume that the estimates of execution time, E, are exact and scheduling decisions and rollbacks are done instantly.

- We require that a transaction locks all pages before it starts any computation. This assumption enforces an *all or nothing* scheme thereby avoiding any deadlocks. If a deadlock detection module is used, this assumption can be relaxed. This module should be invoked periodically and a 'resource preemption' policy used to break the deadlock cycles. Under this scheme, resources will be preempted from some transactions until the deadlock cycle is broken. The most important consideration is the overhead which should be kept at an absolute minimum to justify an efficient real-time transaction processing system.

- In our model, once a transaction has locked all pages and completed the read stage, it is no longer considered to be a candidate for preemption. This assumption has two main advantages. First, since a transaction spends a significant amount of time in the read phase, it is not feasible to preempt it after it has acquired all pages needed for further processing. Second, it ensures that a modified page is always written back to the disk thereby eliminating the need for housecleaning. Note that the possibility of a 'dangling' modified page left behind by a preempted transaction does not exist.

- For the multiprocessor study, we assume that a transaction can execute on any processor and that the cost of executing a transaction is independent of the

processor used.

Simulation Model

As a first phase of the development of simulation model, a simple real-time transaction processing system is developed and tested for various algorithms. The CPU(s) and the system queue are modeled as resources whereas the transactions, source module and disks are modeled as processes. Note that disks are a resource, but they have been modeled as a process because we require a separate service queue for each disk. Further, it allows greater control over the I/O scheduling. When a transaction requests I/O, we initiate subtransactions, one for each disk and each of these subtransactions wait in a separate service queue. However, a transaction may need a very few pages from one disk, but may require many pages from the other disk(s). This results in different I/O completion times on each disk and requires synchronization of the subtransactions. This synchronization is achieved by requiring the transaction to proceed (to the processing stage in case of read, and to the commit stage in case of write) only after all requested I/O is complete.

Since preemption is not available as a part of the SIMSCRIPT language, we implemented it using other data structures. The preempted processes (transactions) release the resources (locks) and return to a ready queue for a later restart. Note that we do not abort a preempted transaction, but use a roll back scheme to restart it later. Further, as discussed in the previous chapter, it is not necessary to undo the effects of a preempted transaction. The possibility of starvation can be eliminated by periodically increasing the priority of tardy transactions.

Serializability is enforced via a two-phase locking protocol which allows read-

locks in a shared mode and write-locks in an exclusive mode. Conflicts are resolved using one of the concurrency control schemes described earlier. All queues in the system are priority queues.

We have implemented a disk-resident database partitioned on one or two disks with the resource parameters as given in Table 4.1. When one disk is used, the entire database is assumed to reside on that disk. Each disk has its own queue of service requests.

Table 4.1: System Resource Parameters

Parameter	**Base Value**
Database Size	400 pages
Buffer Size	200 pages
Number of Disks	One or more
Disk access time per page	25 ms
Number of CPUs	One or more

We assume a large buffer pool relative to the database size in order to allow all pages of active transactions to fit in main memory. The buffer pool is modeled as a set of pages each of which can contain a single database object. Each buffer page is modeled individually, that is, we maintain a list of free pages and keep track of pages which are in the buffer whether or not they are currently locked by a transaction. Since we do not allow preemption after the read phase has been completed, any modified pages are guaranteed to be written back to the disk before the locks are released. Therefore, we do not need to explicitly keep track of the pages which are in the buffer and have been modified.

Transaction characteristics are defined by the parameters listed in Table 4.2. Transactions arrive according to a Poisson distribution and are scheduled dynamically. They are ready to execute when they enter the system, that is, the ready time equals the arrival time, t_{arr}. The execution time, E, depends on the number of pages that the transaction will access/update during its lifetime. The number of pages accessed is chosen from a normal distribution based on a predefined *mean*:

$$Pages\_accessed = NORMAL.F(mean, std\_dev, seed)$$

Table 4.2: Transaction Parameters

Parameter	**Base Value**
Mean Arrival Rate	4 - 8 jobs/sec[a]
Mean # of pages accessed per job	8 pages
CPU Service time per page	15 ms
Probability page is updated	0.5[b]
Slack Factor	2-8 times estimated execution time[c]
Max # of active jobs	25 jobs

[a] A higher job arrival rate is studied for the multiprocessor environment.
[b] A range of 0 to 1.0 is also studied.
[c] Fixed slack in the range of 0 to 8 is also studied.

A *mean* of 8 pages is selected for this study. For the first part of this study, it is assumed that an accessed page may be updated with a probability of 0.5. To implement this, a random number is generated between 0 and 1 and the page is marked as a possible update candidate if the random number is less than the update proba-

bility. The actual database items (pages) are selected randomly from the database. The execution time, E, is then calculated as

$$E = pages_{accessed} * (P + I) + pages_{updated} * I$$

where P is CPU service time per page and I is the I/O time per page.

The slack time, t_{slack}, is evaluated as a uniform distribution between the bounds of the slack factor as specified in Table 4.2, that is,

$$t_{slack} = UNIFORM.F(min\_slk * E, max\_slk * E, seed2)$$

The deadline, d, is the sum of arrival time, execution time and the slack time for a transaction and is calculated as

$$d = t_{arr} + E + t_{slack}$$

The tardy time, t_{tardy}, of a transaction is the time that a transaction spends for completion after its deadline has passed. If t_{comp} is the completion time of a transaction, then its tardy time is defined as:

$$t_{tardy} = t_{comp} - d$$

Pages can be locked in a shared or an exclusive mode. The modified pages are written back to the disk only after the transaction has committed. By limiting the total number of transactions that can be active at any given instance of time and by assuming a large enough buffer pool, we ensure that a *write* does not become necessary until after the transaction commits.

When the buffer is full, one or more pages may have to be swapped out to bring in the pages requested by an active transaction. We selected the LRU page

replacement policy because it falls in the class of *stack algorithms* and is generally found to be very efficient [34]. A *buffer manager* uses the LRU policy to select the victim page to be replaced in the buffer pool. This is obviously the page that is not currently locked by any active transaction. By restricting the maximum number of active transactions and assuming a large buffer ensures that such a 'victim' page always exists.

During the simulation we recorded a number of parameters. In order to present the results, we have selected three important performance metrics for the real-time environment: % missed deadline, mean tardy time, and average response time. The primary goal in any real-time system is to meet the deadlines as much as possible. In a soft real-time environment, we can allow transactions to miss their deadlines. However, we want to keep this number to an absolute minimum and a measure of this parameter (% missed deadlines) essentially determines the effectiveness of a protocol. For the transactions that miss their deadline, our goal is to minimize tardy time. Therefore, measuring the mean tardy time in a system where deadlines are missed is also important. In real-time database systems, the average response time is less important than the other two metrics just defined. However, the average response time of the system should not become large for the system to be realistic. The other parameters that are recorded include system throughput, number of preemptions, average and maximum queue lengths, and CPU utilization.

A series of simulation experiments reflect the performance of various algorithms under different system load and configuration scenarios. The algorithms represent various combinations of the policies used for priority assignment and concurrency control. The four priority assignment schemes (FCFS, EDF, MSTF and SJF) and

three concurrency control protocols (blocking, high-priority preemption and conditional preemption) as described earlier have been implemented. We have studied the effect of tuning the underlying system when locks are held in exclusive mode only. The model is then extended to allow shared locks and all other studies are done using an enhanced system configuration and the shared locking mode. To illustrate the performance improvement obtained by allowing shared locks, we also compare the performance of the two locking modes.

Simulation Results

In this section we present results obtained from our experiments performed. Each run is continued until at least 700 transactions are executed. SIMSCRIPT allows using up to 20 different random number seeds. For each algorithm tested, numerous performance statistics have been collected and averaged over runs for different random number seeds. As mentioned earlier, the two most important metrics which measure the performance in a real-time transaction processing system are the percentage of the transactions that miss their deadlines and the average tardy time of all committed transactions. Other experiments are performed for the percentage of transactions processed, average response time and the system throughput. The system load is varied from 1 to 8 transactions arriving per second. It is expected that a real-time transaction processing system should not only perform well under a lightly loaded situation but also for periods when the system is heavily loaded. Further, when the system is lightly loaded, most transactions meet their deadlines for the chosen parameters. Therefore, we only present results for the load ranging from 4 to 8 transactions per second with an interval of 0.5.

For the priority assignment and concurrency control protocols described earlier, we have studied the effect of partitioning the data on multiple disks, effect of buffer management, effect of allowing preemption, effect of update probability, effect of locking modes, effect of slack factor and effect of multiprocessing.

Effect of Partitioning Data, Buffer Management and Preemption

In Figures 4.1 to 4.20, we use a 3-character notation to represent the underlying features of our simulation model. The first character indicates whether or not buffer management is used, the second character indicates whether or not preemption[1] is allowed, and the third character indicates the number of disks used. With this notation, NN1 represents the most basic system configuration with data residing on a single disk, no buffer management and no provision for preemption. NN2 represents data partitioned on two disks; YN2 represents the case when an LRU buffer management policy is incorporated and YY2 represents the case when preemption is also allowed. The remaining results (Figures 4.21 to 4.24) are for the enhanced (YY2) configuration. In the results shown, a significant performance improvement is observed when data is partitioned on two disks because of the smaller queues and a reduced waiting time. Similarly, buffer management and introduction of preemption further reduces the number of tardy transactions and the mean tardy time. For this part of the study, all locks are assumed to be held in an exclusive mode.

Figures 4.1 to 4.4 shows the percent of transaction missing deadlines under different system configurations and priority assignment schemes described above. Note that even for a high load scenario (8 jobs/second) the number of tardy transactions

[1]For this part of the study, we only allow high priority preemption.

is reduced to less than half for most cases when data is partitioned on two disks, buffer management is used and preemption is allowed (the YY2 configuration). For medium loads (4-6 jobs/second), the number of transactions that miss the deadlines is reduced to zero from a range of 60-99 percent for the NN1 configuration. The best performance is shown by the SJF and the MSTF priority assignment schemes. EDF and FCFS both result in a higher number of tardy transactions under a high load scenario.

Figures 4.5 to 4.20 show similar results for ratio of the number of transactions processed to the total number of transactions arriving in the system, mean tardy time and average response time. The percentage of transactions that are processed decrease with an increasing load for the NN1 and NN2 configurations (Figures 4.5-4.8). However, for YN2 and YY2 both, this ratio is very high and fairly steady over the entire range of loads. This implies that buffer management greatly assists in improving the throughput of the system. The mean tardy time (Figures 4.9 to 4.11) and the mean response time (Figures 4.13 to 4.16) are improved by over a factor of 16 for the EDF and FCFS schemes between the NN1 and YY2 configurations at the arrival rate of 8 jobs/sec. By simply partitioning the data on two disks (NN2), the results show a four-fold improvement as compared with the NN1 configuration. This improvement comes from the fact that now the transactions not only spend less time waiting in the queues, but also that the disks can be accessed concurrently. However, as expected, preemption does not affect performance when the transactions are assigned priorities using the FCFS scheme because with this scheme no later transaction can have a higher priority than an earlier transaction, thereby eliminating any possibility of preemption.

Figures 4.21 to 4.24 compare various priority assignment schemes for the YY2 configuration. Figure 4.21 shows the percentage of transactions missing the deadlines under various load conditions. Since EDF gives the highest priority to transactions that have the least remaining time to complete, it performs the best under low and moderate load levels. However, the performance of EDF steeply degrades in an overloaded system. This is because, under higher loads, transactions gain high priority only when they are close to their deadlines. Gaining high priority at this late stage may not leave sufficient time for a transaction to complete before its deadline. Therefore, as illustrated by Figure 4.21, a fundamental weakness of EDF under heavy loads is that this policy tends to assign high priorities to transactions which will miss their deadlines anyway. Further, the high-priority conflict resolution can produce a lot of restarts which effectively increase the arrival rate of new transactions into the system, thus increasing the load and making EDF undesirable at high load. This can be improved when other concurrency control protocols are used. The MSTF and SJF priority assignment schemes exhibit a superior performance for overloaded systems. With the same concurrency control mechanism, MSTF produces fewer restarts than EDF because of an additional test (checking for priority after abort) that can prevent transactions from being restarted. This results in MSTF performing better at higher load levels.

Effect of Update Probability

As shown in the previous section, the use of buffers, preemption and data partitioning significantly improves the performance of a real-time transaction processing system. In this section, we show how the data contention resulting from high update

probability can affect the performance of the system. For this and remainder part of the study, we use our improved system configuration (YY2) in which the data is partitioned on two disks, buffer management is used and preemption is allowed. We have studied this effect for the four priority assignment schemes (EDF, SJF, FCFS, and MSTF) and the three concurrency control protocols (BLOCK, HIPRTY, and CPR) as described earlier.

The transactions access a certain number of pages (mean 8) during their lifetime and may or may not update one or more of these pages. In case the pages are updated, it is necessary that these pages are written back to the nonvolatile storage (disks) before the transaction can commit. Depending on the number of pages updated, this can result in a moderate to a very high disk contention and can effectively delay the transactions eventually resulting in missed deadlines. In [20] we assume that an accessed page may be updated with a probability of 0.5. Keeping other variables fixed, we have now studied the effect when the update probability is varied from 0 to 1.0 with a step of 0.1. The effect of this variation is studied on two important performance metrics: the percentage of transactions missing the deadline and the mean tardy time. An update probability of 0 represents read-only transactions because it implies that none of the pages accessed is updated. On the other hand, the update probability of 1 represents the other extreme where every page accessed is assumed to be updated resulting in a high disk contention during the write phase.

Figures 4.25 to 4.44 show that, under any of the priority assignment schemes and any concurrency control method, the performance of a real-time transaction processing system is significantly affected by the transaction's I/O requirements. It is observed that transactions with high update probabilities can result in poor per-

formance even at very low loads compared to the transactions that do not update the data. For example, for EDF with blocking (Figures 4.25 to 4.28), 65% of the transactions miss their deadlines at an arrival rate of 6 jobs/sec whereas such performance is not observed even at an arrival rate of 8 jobs/sec in case of the read-only transactions. In fact, for an arrival rate of 6 jobs/sec, the transactions that miss the deadline increase from under 5% to almost 60% when the update probability changes from 0.5 to 1 (Figure 4.27). Similarly, at an arrival rate of 8 jobs/sec, the mean tardy time increases from under 2 seconds to about 16 seconds when the update probability changes from 0.5 to 1 (Figure 4.28).

A similar behavior is observed for all other combinations of priority assignment schemes and the concurrency control policies. Since the general trend is the same, we present results for high-priority preemption (HIPRTY) only. For this concurrency control protocol, Figures 4.29-4.32 illustrate the performance for EDF, Figures 4.33-4.36 for SJF, Figures 4.37-4.40 for FCFS, and Figures 4.41-4.44 for MSTF, priority assignment schemes. In all these cases, the overall trend is the same for a fixed update probability. However, the performance gets worse sooner, that is, at lower loads, as the update probability becomes higher. The results for BLOCK and CPR concurrency control protocols also exhibit the same trend and are, therefore, not presented here.

Effect of Locking Mode

When locks are shared, the waiting time of transactions decrease significantly and less preemptions are required. These factors, in turn, result in an improved performance compared to the case when the locks are always held in an exclusive mode.

Note that with exclusive locks only, conflict always involve a pair of transactions. With shared locks, the lock may actually be held by a set of concurrently reading transactions, each with a different deadline. In our study, we allow preemption only if the conflict is one-to-one. To implement this, we maintain a *conflict set* associated with each transaction, T. This set contains a list of all transactions that T is conflicting with.

Figures 4.45 to 4.48 illustrate the effect of allowing shared locks under the EDF priority assignment scheme using high priority preemption (HIPRTY) and conditional preemption protocols (CPR). The results are obtained for the system configuration described earlier except that the update probability is once again fixed at 0.5 (YY2). At a job arrival rate of 8 jobs/sec the number of preemptions drop from 78 to 30. This results in number of transactions missing the deadlines to improve by about 16% with the shared read-locks. The mean tardy time of the transactions and the average response time also improve by 9% and 11% respectively. Note that all this improvement is a result of allowing readers to share locks. The results for other priority assignment schemes are similar and, therefore, not presented here.

Effect of Slack Time

It is not feasible to consider a transaction's deadline to be equal to the arrival time plus the execution time since the transaction may spend time waiting in queues because of unavailability of resources. To account for such unpredictable delays, it is important to allow some slack for completion of a transaction. In this study, we evaluate the effects of varying the slack factor on the performance of various combinations of priority assignment schemes and concurrency control protocols under

different load scenarios. Recall that we define slack time as a function of the total execution time of a transaction calculated from a uniform distribution as

$$Slack\_time = UNIFORM.F(min\_slack * E, max\_slack * E)$$

where E is the estimated execution time of the transaction. In this simulation experiment, however, we use the same value for the minimum and maximum slack so that the slack factor becomes a ratio of the slack time to the execution time, that is

$$SlackFactor = \frac{Slack\_time}{Execution\_Time}$$

The slack factor is varied from 0 to 8. A slack factor of 0 (or close to 0) represents the case of very tight deadlines. All other parameters are kept constant, preemption is allowed, buffer management is in place and the data is assumed to be partitioned on two disks. To represent light, medium and heavy load scenarios we select arrival rates of 4.0, 6.0 and 8.0 jobs per second. The percentage of transactions missing the deadlines is used as the performance metric for this study.

Figures 4.49 to 4.52 show the percent of transactions that missed the deadlines for the three load scenarios for the four priority assignment schemes (EDF, SJF, FCFS and MSTF) and when simple blocking is used as a concurrency control mechanism. In all the cases, a very high percentage of transactions miss their deadlines when the deadlines are tight (smaller slack). In fact, for a job arrival rate of 8.0 jobs per second (a high load scenario), none of the transactions could meet its deadline when EDF, FCFS or MSTF is used for priority assignment. For low and medium load levels, all the transactions meet their deadlines when the slack factor is high (4 for EDF and 7 for the other schemes). However, for higher loads, even at a slack factor of 8, a significant number (22-33%) of transactions still miss their deadlines. A general

observation from all these figures is that a higher slack factor does result in reduced miss rate.

Figures 4.53 to 4.60 show similar results for the HIPRTY and CPR concurrency control protocols. Note that the general trend for these protocols is the same as for EDF discussed above. Any difference in the percentage of tardy transactions is not because of the slack factor but is a result of a different concurrency control protocol. HIPRTY and CPR concurrency control protocols seem to perform better than the simple BLOCK where no preemptions are allowed.

Effect of Multiprocessing

The basic characteristic of a multiprocessing system is the existence of several processors which can operate independently. The efficient utilization of such a system can be very effective in decreasing the response times of many programs. This is particularly important for real-time systems where the results are needed more quickly than they can be provided by a single processor. Transaction scheduling on multiprocessors is difficult if an optimal schedule is desired under timing and resource constraints. In our study, we illustrate that performance can be enhanced to a certain extent by providing more resources. In this section, we present our observations based on our previous model extended to allow multiple processors keeping all other parameters constant.

We have modeled a multiprocessor system by allowing multiple instances of the resource, CPU, in our simulation model. We assume that all processors are identical and any transaction can execute on any processor. Further, the cost of executing a transaction on one processor is the same as the cost of executing it on any other

processor. No transaction requests more than one processor. When a transaction T completes its read phase and requests a processor, the first available processor will service this request. If a processor is not available, T joins the queue of transactions waiting for the processor. All processors service a single queue which is maintained by priority of the transactions. Keeping other transaction and system parameters fixed and assuming the data to be residing on one disk, we increase the number of processors from 1 to 10. The experiment has been conducted for a number of load scenarios (varying job arrival rate from 4.0 to 24.0 jobs/second). The slack factor is fixed at 2, that is, each transaction had a slack time equal to twice its estimated execution time. Recall that in our previous experiments, this was the lower bound on the slack factor. We have chosen percent of transactions that miss the deadline as a performance metric for representing results from this study. The effect on the mean tardy time of transactions and the average transaction time is similar. The results presented here illustrate interesting observations and are representative of all experiments.

Figures 4.61-4.72 present results from some combinations of the priority assignment and concurrency control schemes under different load scenarios. It is generally observed that a significant performance improvement results when the number of processors is increased from 1 to 2, but the improvement gradually becomes less and less significant thereafter. In fact, as illustrated by Figures 4.61-4.64, for concurrency control schemes that do not allow preemption (such as BLOCK), there is no improvement at all when the number of processors is increased. The reason for this is that the disk is already the bottleneck even with one processor. Therefore, the first improvement can only come from removing this bottleneck device. Figures 4.65-4.68

show that when high priority preemption (HIPRTY) is used for resolving conflicts, performance improves significantly at first but the curve soon becomes flat indicating no further performance gain no matter how many additional processors are provided. A similar behavior is observed when conditional priority preemption (CPR) is used for concurrency control. This behavior is not surprising because the other resources in the system presumably become the bottleneck device limiting the system performance. At this point, too much improvement of one resource can be wasteful until the bottleneck device is improved [29]. Also note that for FCFS, no difference is observed (Figures 4.62 and 4.66) because under this scheme all transactions have the same priority and are serviced from a FIFO queue.

It was suspected that the disk, which is another resource in the system, could be a potential bottleneck device. This is confirmed when we perform the simulation experiment with exactly the same parameters except that now the data is partitioned on two disks. We observe that under this configuration, increasing the number of processors resulted in a better performance over a wider range. Note that now even the BLOCK concurrency control scheme benefits from an increased number of processors (Figures 4.61-4.64) because the data is partitioned over two disks and perhaps the processor is the bottleneck device to start with. The new bottleneck device (disk) once again takes over, but this time at a later point, after which no further improvement could be seen. At this point, adding more disks to the system could further improve the performance as long as another resource does not become a bottleneck device.

Figures 4.69-4.72 show results when the CPR scheme is used and the transactions arrive at a rate of 12.0 jobs/second. At this higher arrival rate, we observe that

increasing the number of processors gradually result in performance gain but the curve becomes flat after the maximum improvement is obtained from adding more processors. The trend is the same for one or multiple disks with very little difference when more disks are added. For the FCFS scheme and at this arrival rate, no performance gain is obtained by adding more resources (Figure 4.70).

For a very high job arrival rate (24.0 jobs/second), increasing the number of processors do not improve the performance of the system at all even when the data is partitioned on two disks. This is because even when the data is partitioned, the disk(s) continue to be the bottleneck device. In such circumstances, increasing the number of processors only reduces the processor utilization without resulting in any performance gain. The curves for this arrival rate are straight horizontal lines and are not selected for inclusion in the figures presented in this chapter.

Summary

In this chapter, we have described our simulation model and the underlying assumptions. The model has been implemented using discrete-event simulation techniques to provide a testbed for studying various proposed algorithms for priority assignment and concurrency control. A set of system and transaction parameters selected on the basis of literature review is also presented. We have studied the effect of varying these parameters on the overall performance of the transaction processing system.

The results obtained from the simulation experiments illustrate the effect of partitioning data, buffer management, preemption, update probability, locking mode, slack time and multiprocessing. It is observed that significant performance gain can

be achieved by simple partitioning of data on multiple disks and/or using buffer management. Performance is further enhanced by using preemptive scheduling algorithms and by allowing shared locks. Since I/O accounts for a significant portion of the transaction's execution time, the overall performance is better for read-only transactions as compared to those which require many updates. Furthermore, as expected, tighter slack times result in more tardy transactions. By running the simulation over a range of values for the slack factor, an acceptable slack time can be determined for the entire set of transactions. The experiments using multiple processors illustrate the impact of bottleneck devices in a system. It has been observed that too much correction at one resource cannot indefinitely improve the overall performance because another resource can then become the bottleneck device.

In the next chapter, we focus our attention on a distributed environment. We define a model that uses the notion of nested transaction in the real-time database environment. The proposed model is shown to exhibit properties that make it very desirable for distributed real-time transaction processing.

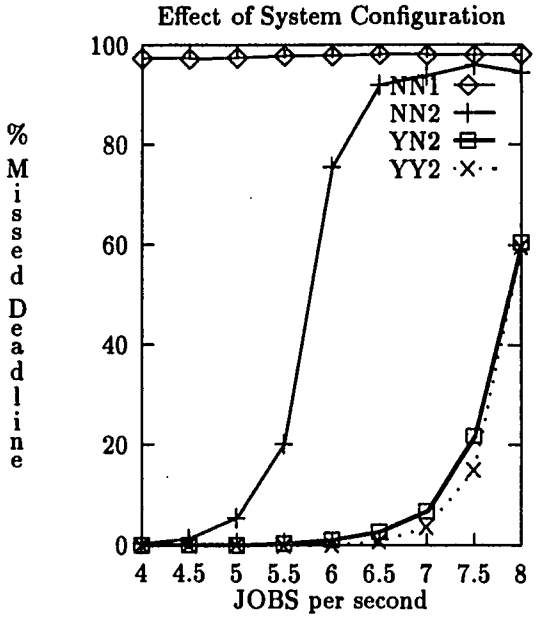


Figure 4.1: Earliest Deadline First.

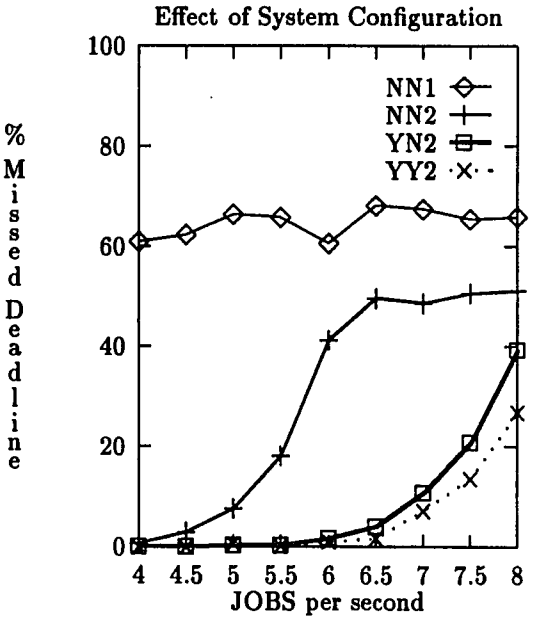


Figure 4.2: Shortest Job First.

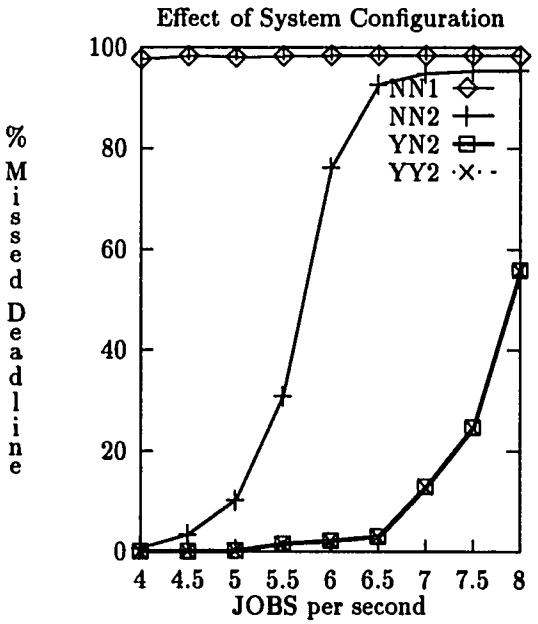


Figure 4.3: First-Come-First-Served.

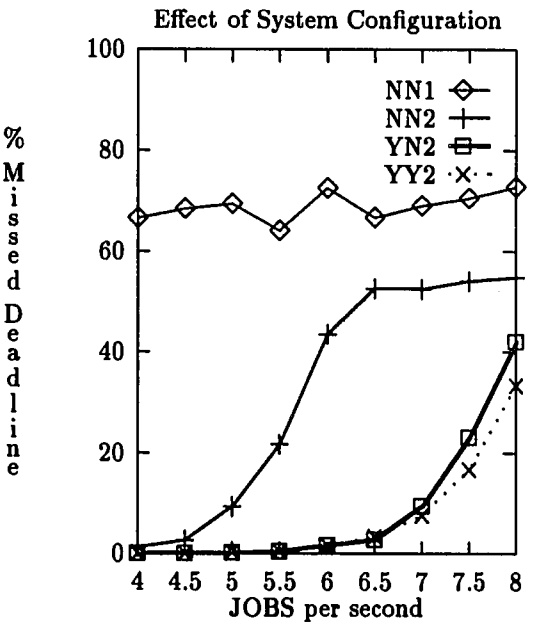


Figure 4.4: Minimum Slack First.

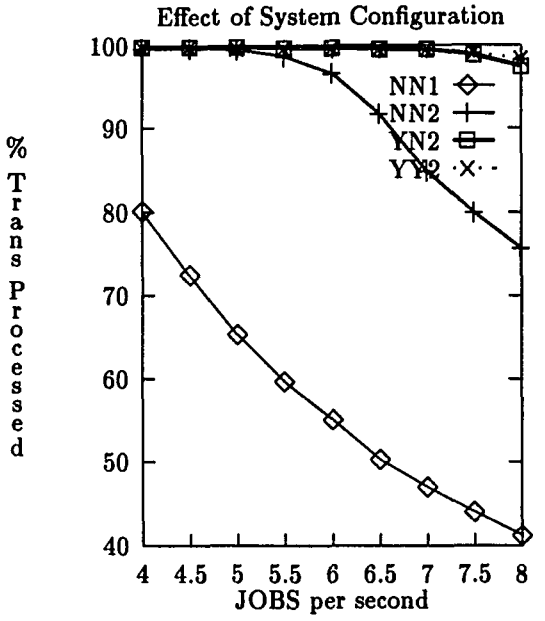


Figure 4.5: Earliest Deadline First.

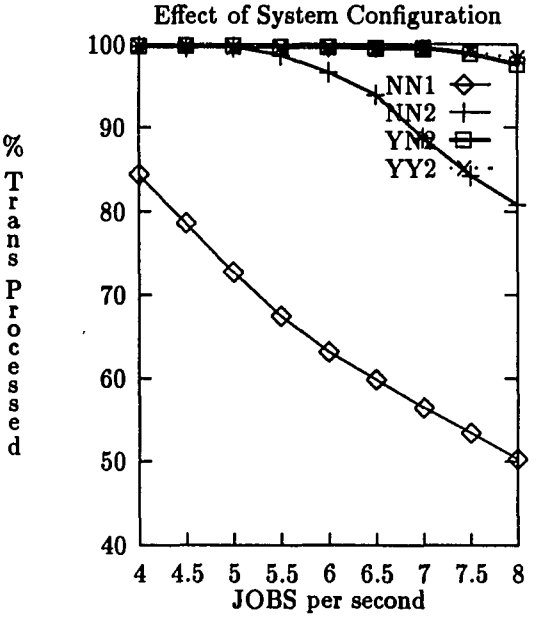


Figure 4.6: Shortest Job First.

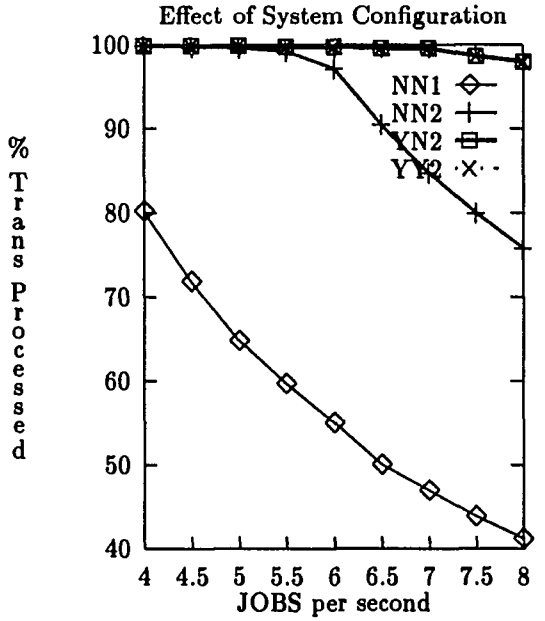


Figure 4.7: First-Come-First-Served.

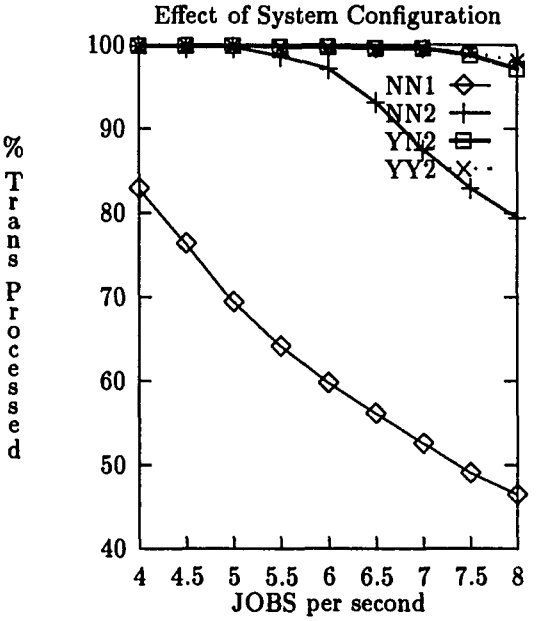


Figure 4.8: Minimum Slack First.

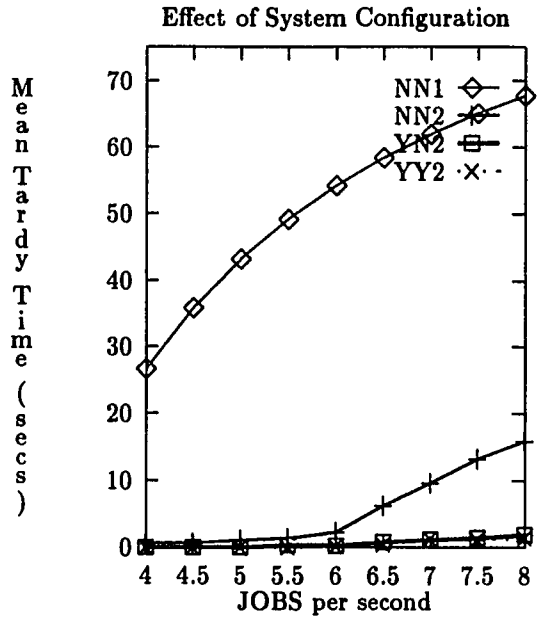


Figure 4.9: Earliest Deadline First.

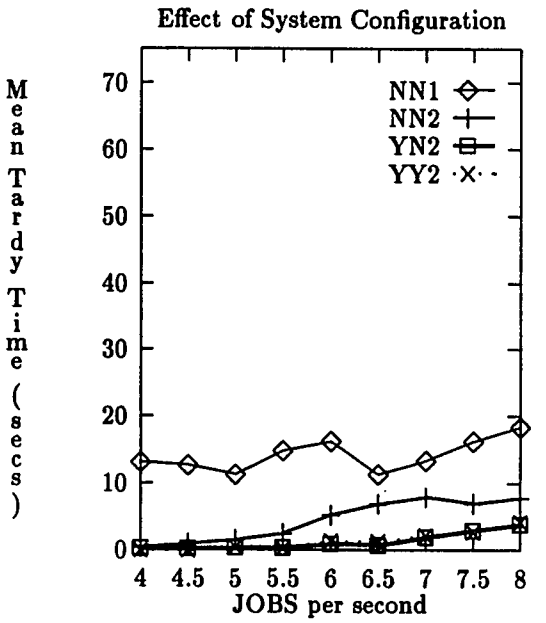


Figure 4.10: Shortest Job First.

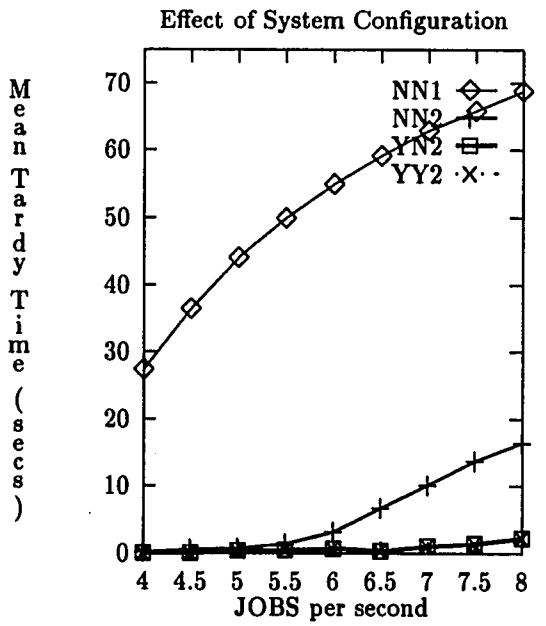


Figure 4.11: First-Come-First-Served.

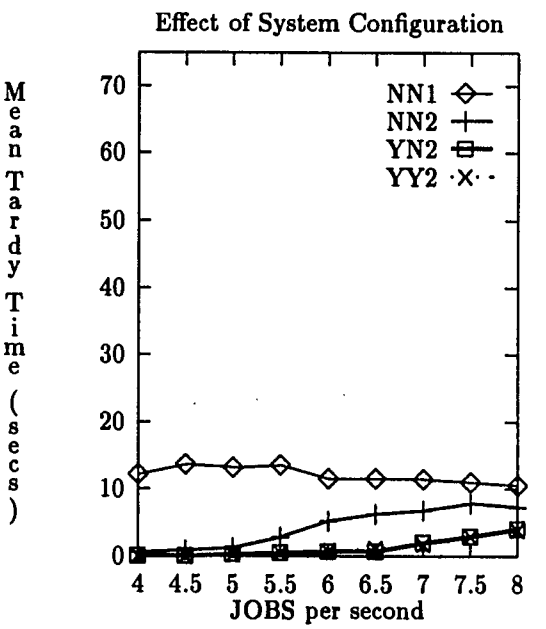


Figure 4.12: Minimum Slack First.

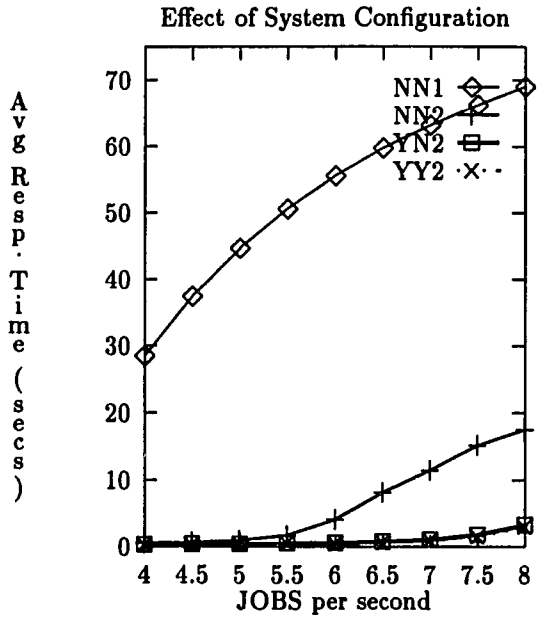


Figure 4.13: Earliest Deadline First.

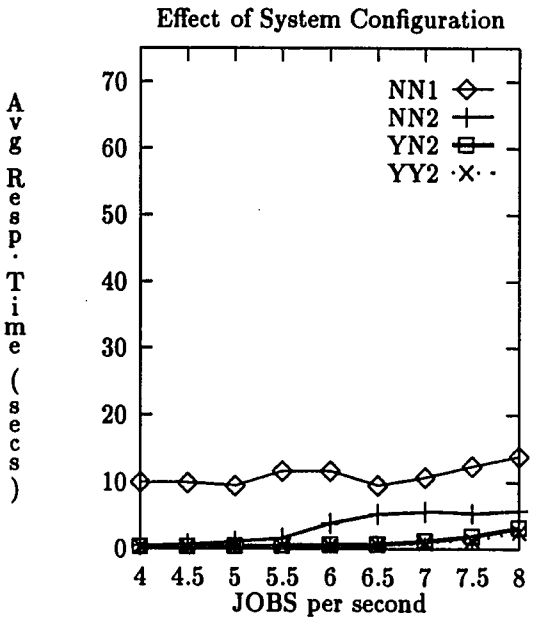


Figure 4.14: Shortest Job First.

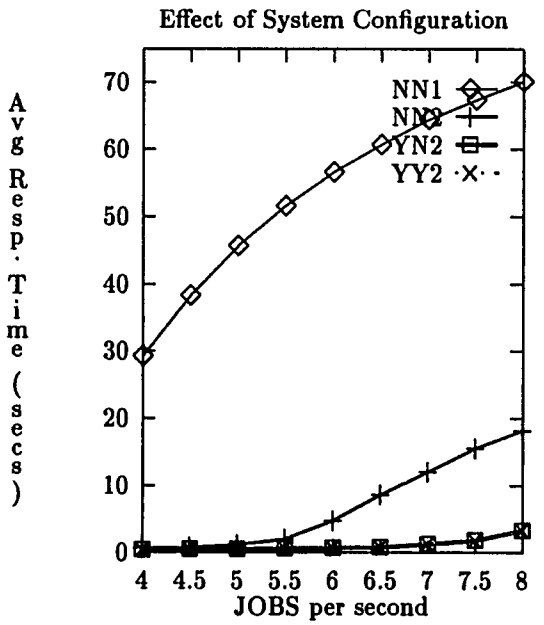


Figure 4.15: First-Come-First-Served.

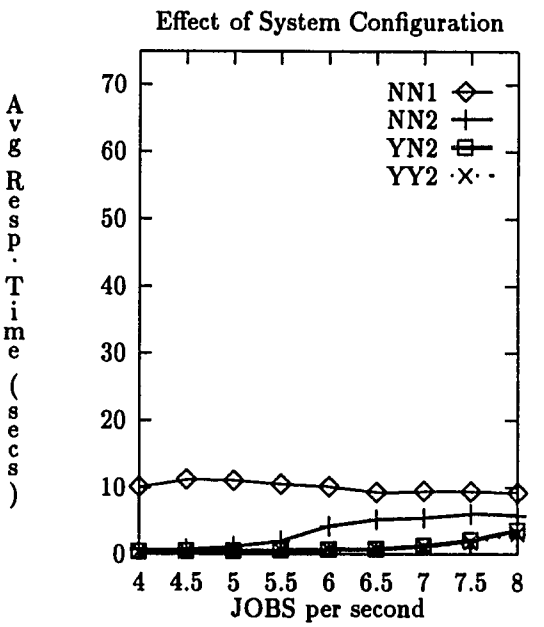


Figure 4.16: Minimum Slack First.

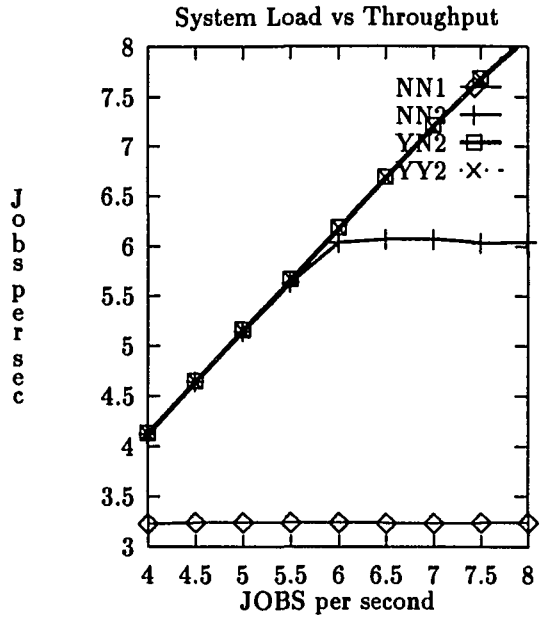


Figure 4.17: Earliest Deadline First.

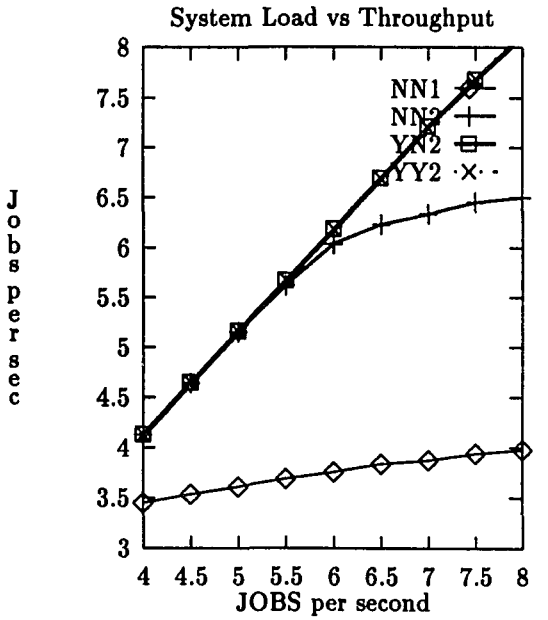


Figure 4.18: Shortest Job First.

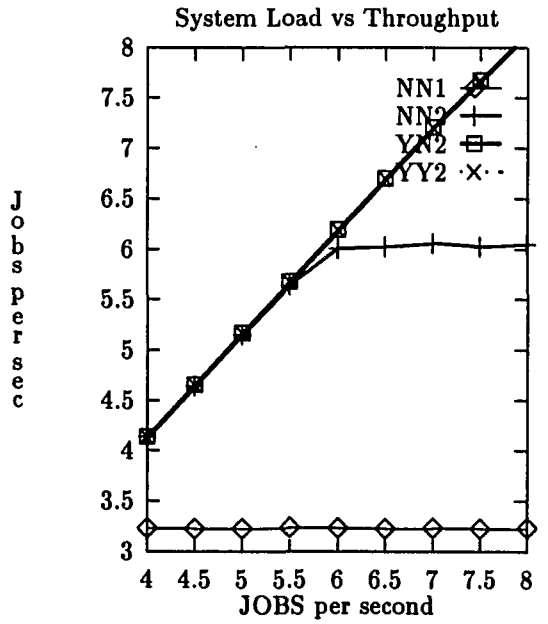


Figure 4.19: First-Come-First-Served.

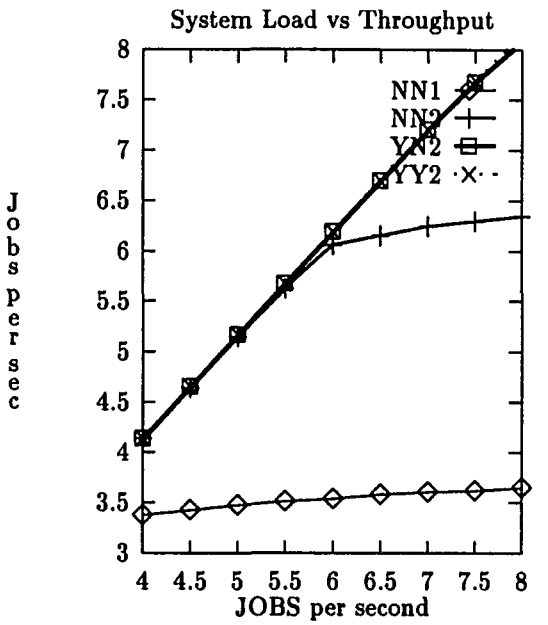


Figure 4.20: Minimum Slack First.

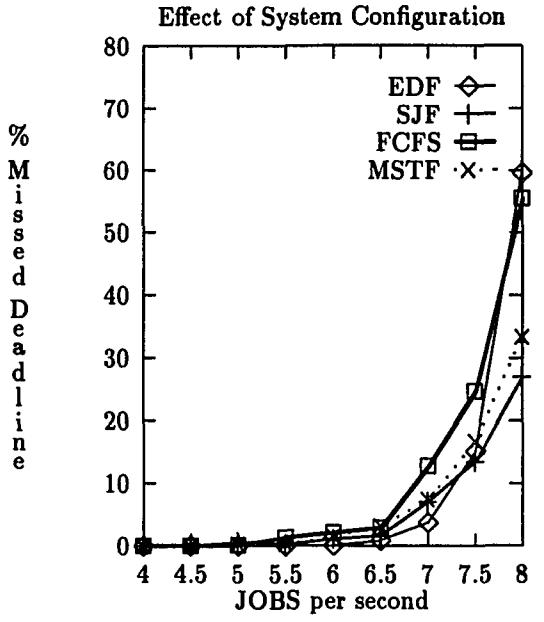


Figure 4.21: Deadlines Missed.

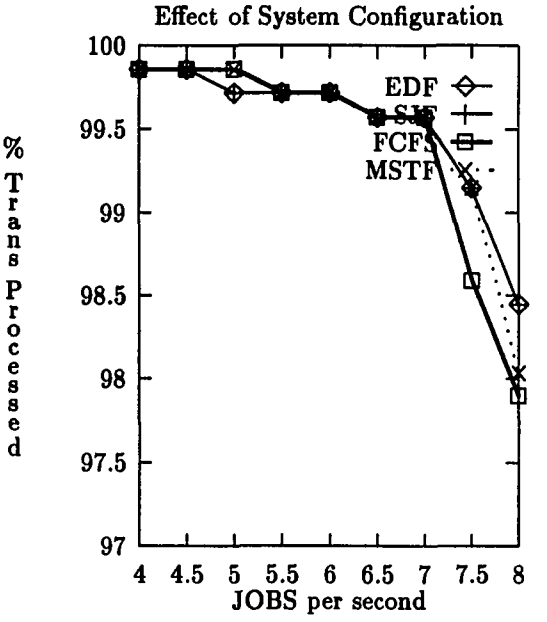


Figure 4.22: Tardy Transactions.

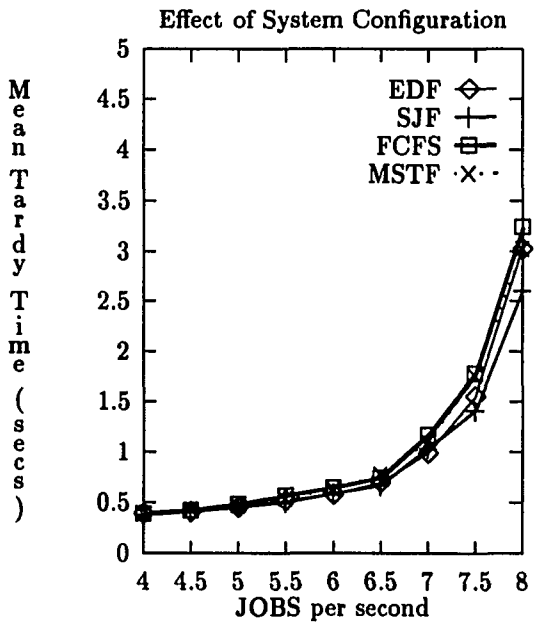


Figure 4.23: Transactions Processed.

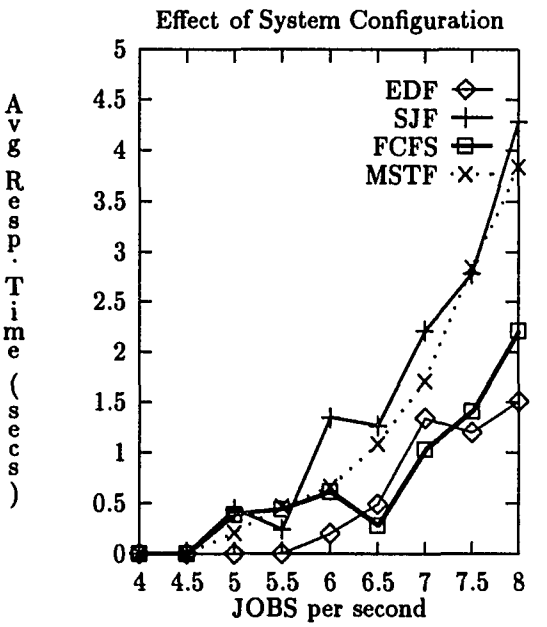


Figure 4.24: Response Time.

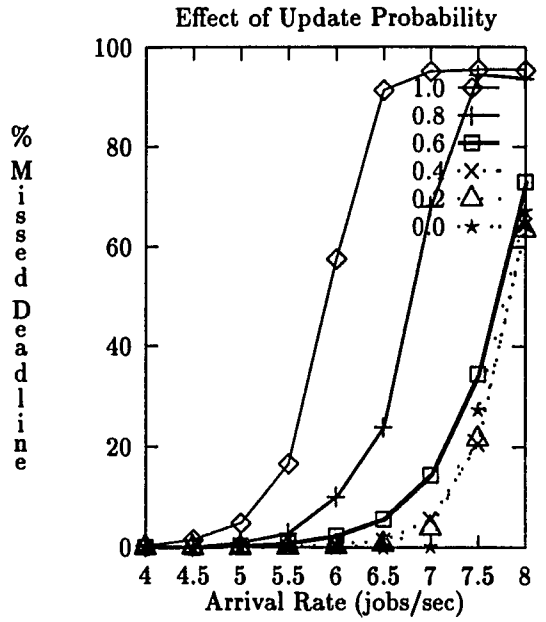


Figure 4.25: EDF(BLOCK).

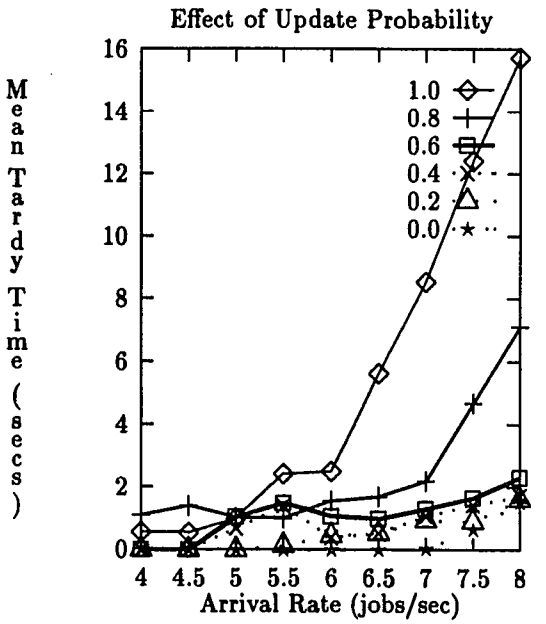


Figure 4.26: EDF(BLOCK).

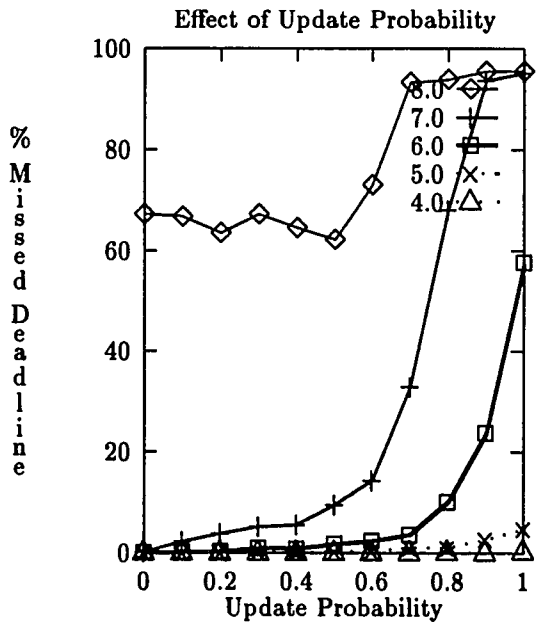


Figure 4.27: EDF(BLOCK).

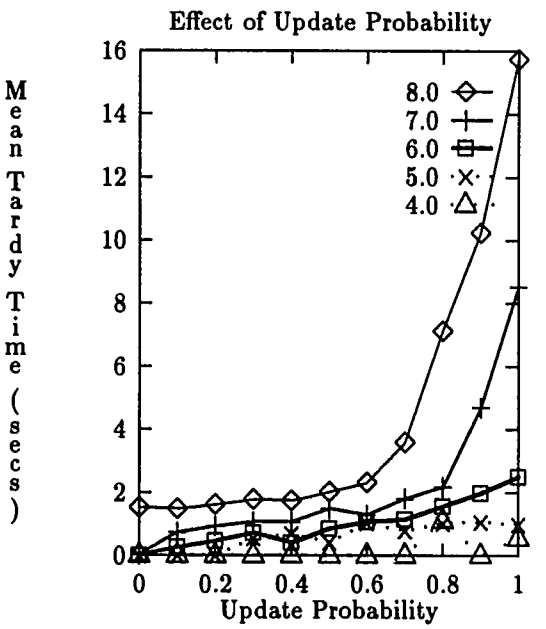


Figure 4.28: EDF(BLOCK).

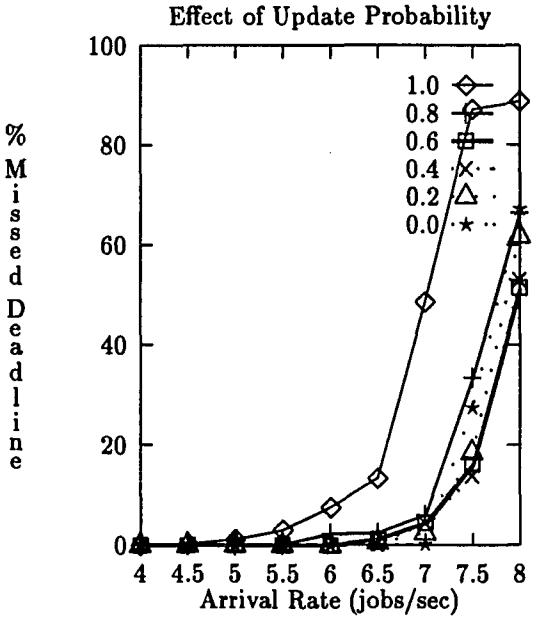


Figure 4.29: EDF(HIPRTY).

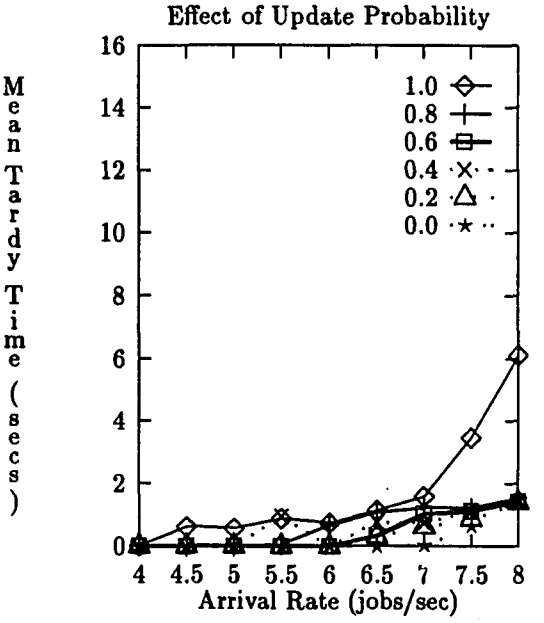


Figure 4.30: EDF(HIPRTY).

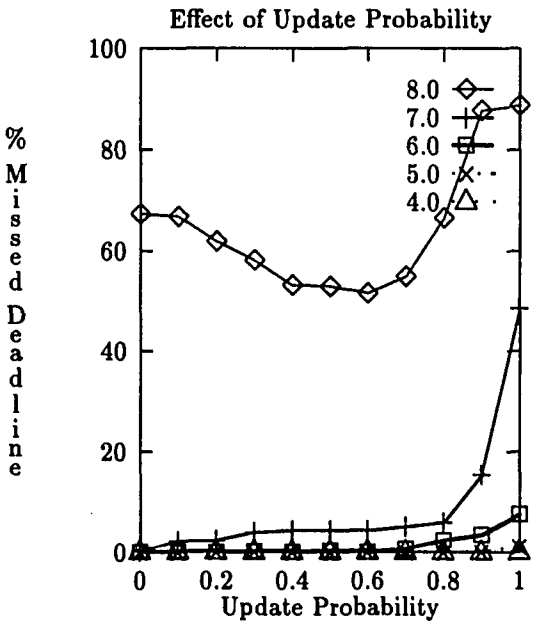


Figure 4.31: EDF(HIPRTY).

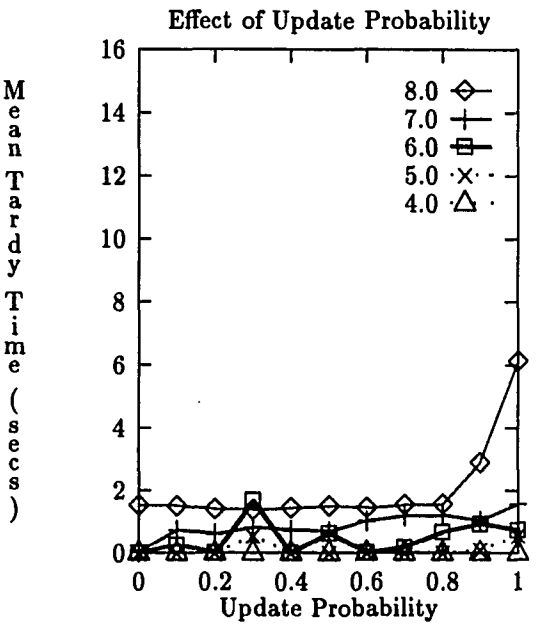


Figure 4.32: EDF(HIPRTY).

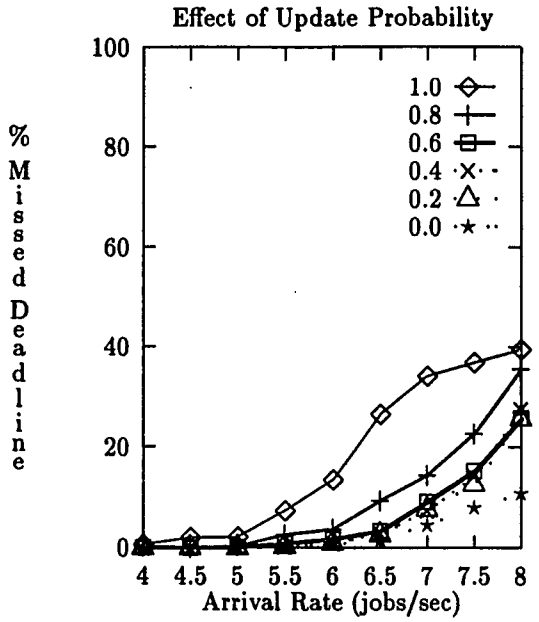


Figure 4.33: SJF(HIPRTY).

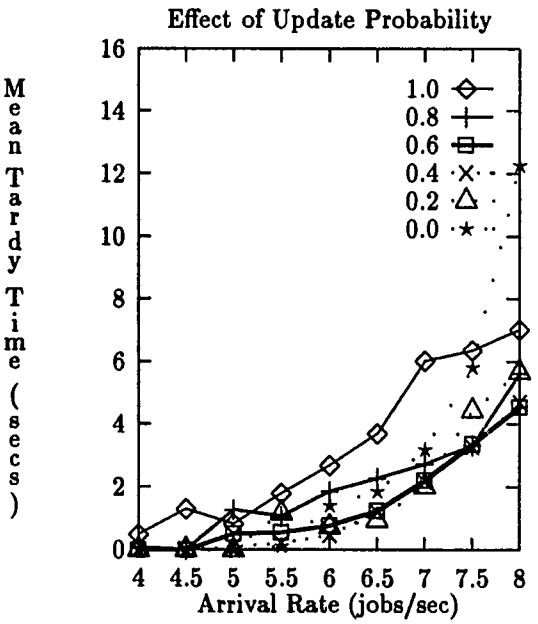


Figure 4.34: SJF(HIPRTY).

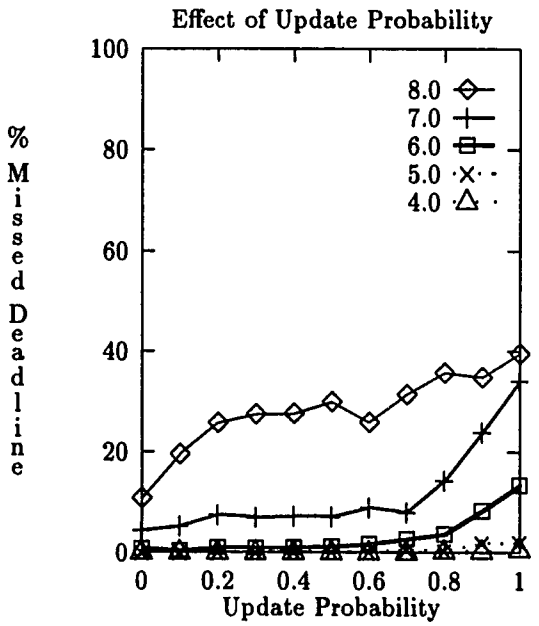


Figure 4.35: SJF(HIPRTY).

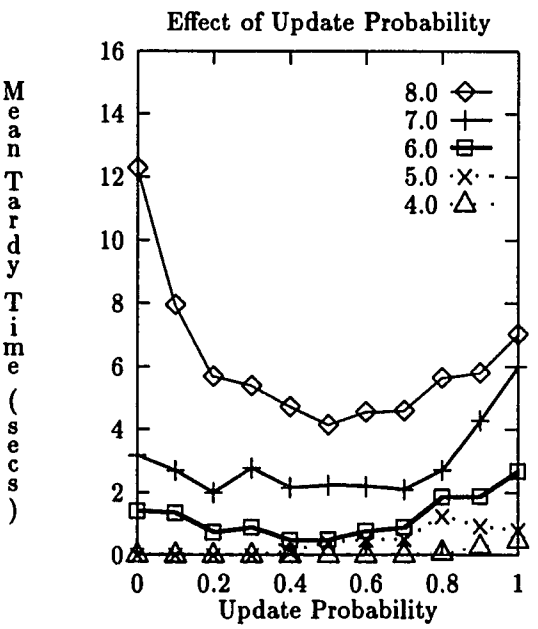


Figure 4.36: SJF(HIPRTY).

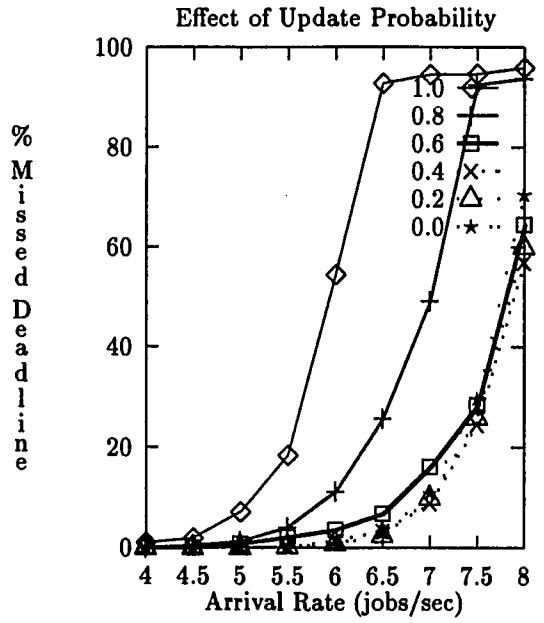


Figure 4.37: FCFS(HIPRTY).

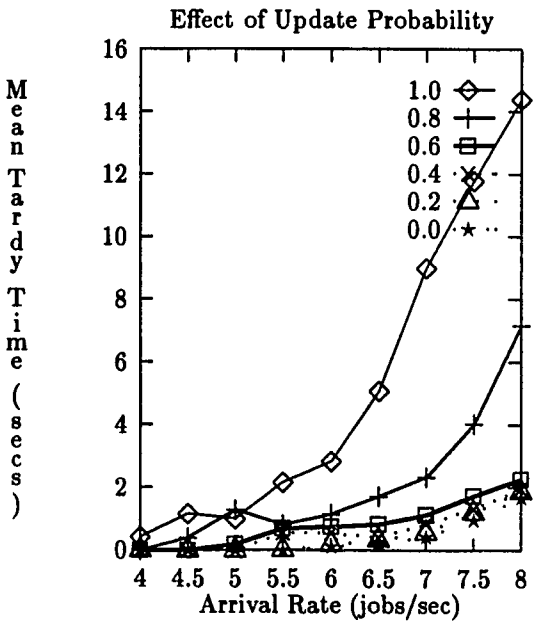


Figure 4.38: FCFS(HIPRTY).

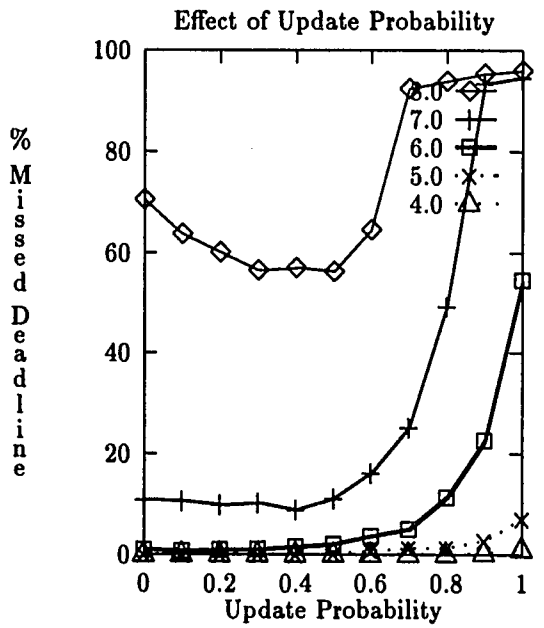


Figure 4.39: FCFS(HIPRTY).

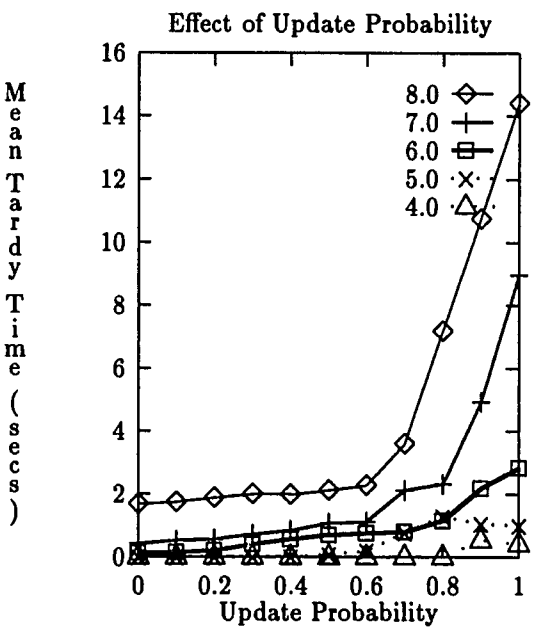


Figure 4.40: FCFS(HIPRTY).

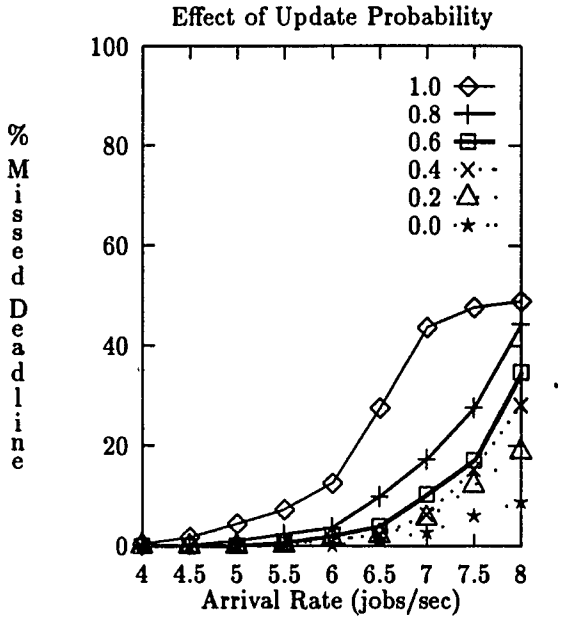


Figure 4.41: MSTF(HIPRTY).

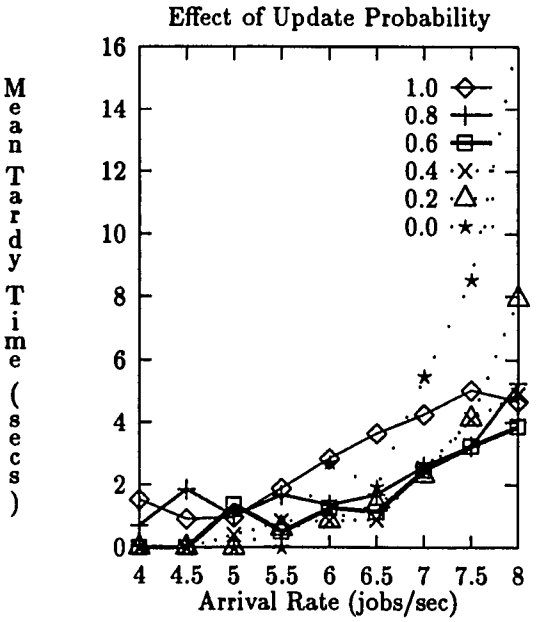


Figure 4.42: MSTF(HIPRTY).

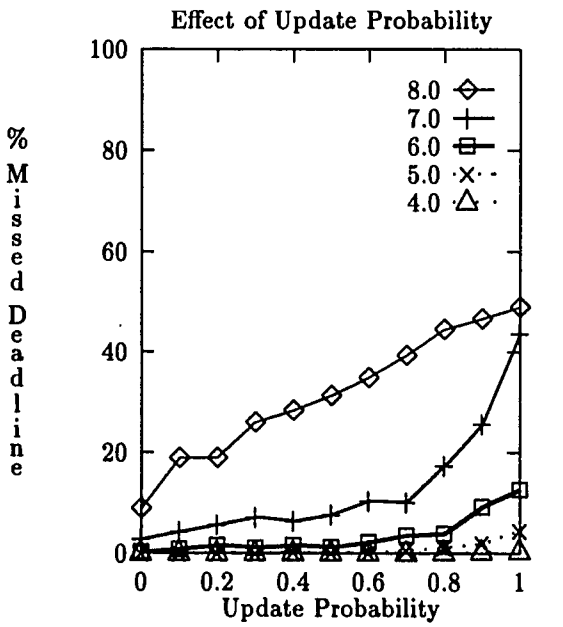


Figure 4.43: MSTF(HIPRTY).

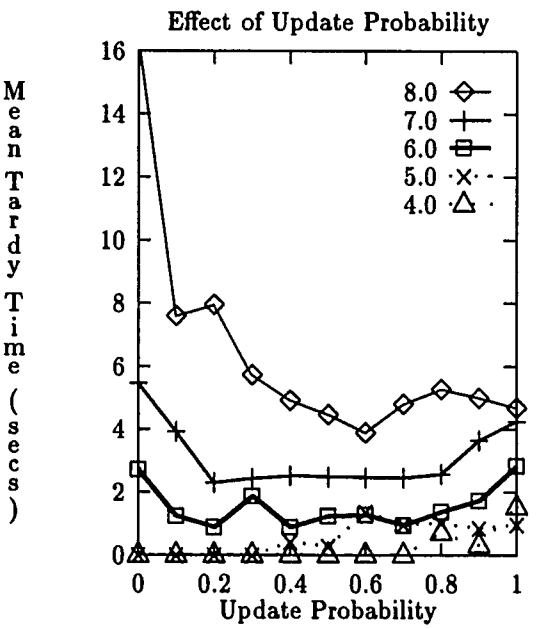


Figure 4.44: MSTF(HIPRTY).

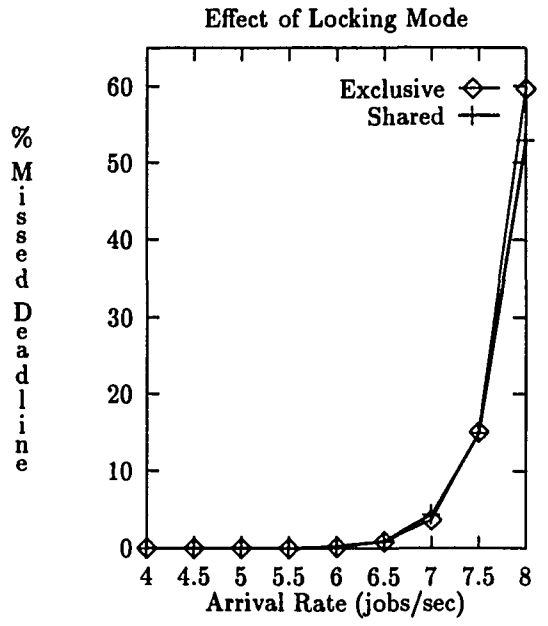


Figure 4.45: EDF(HIPRTY).

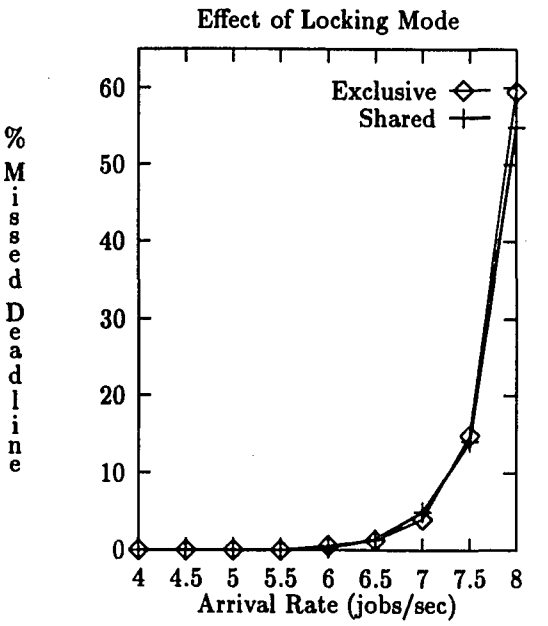


Figure 4.46: EDF(CPR).

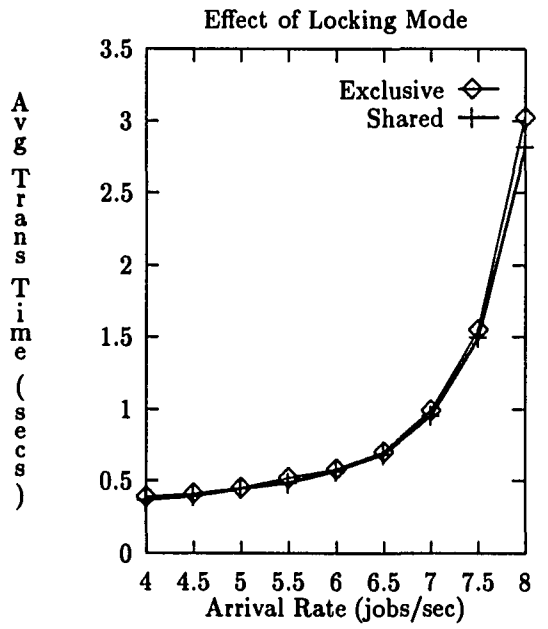


Figure 4.47: EDF(HIPRTY).

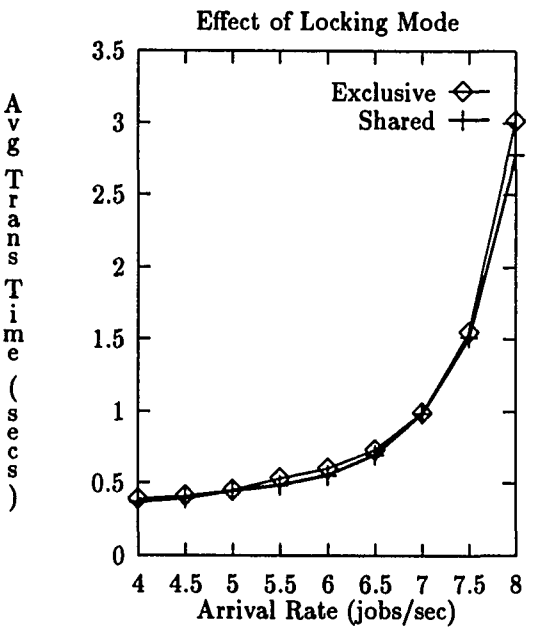


Figure 4.48: EDF(CPR).

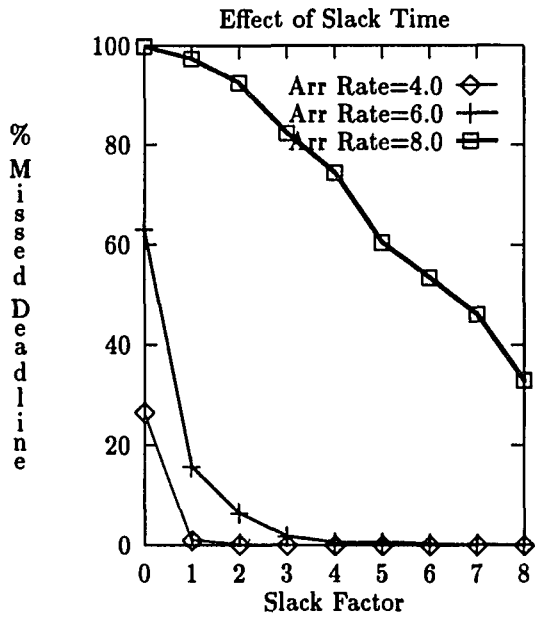


Figure 4.49: EDF with BLOCK.

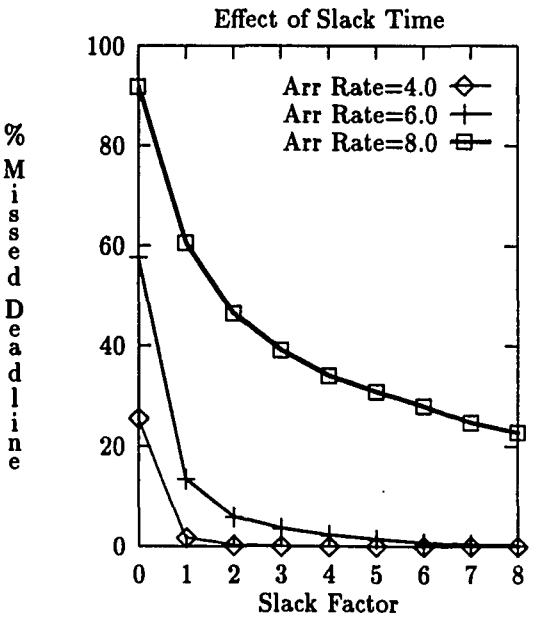


Figure 4.50: SJF with BLOCK.

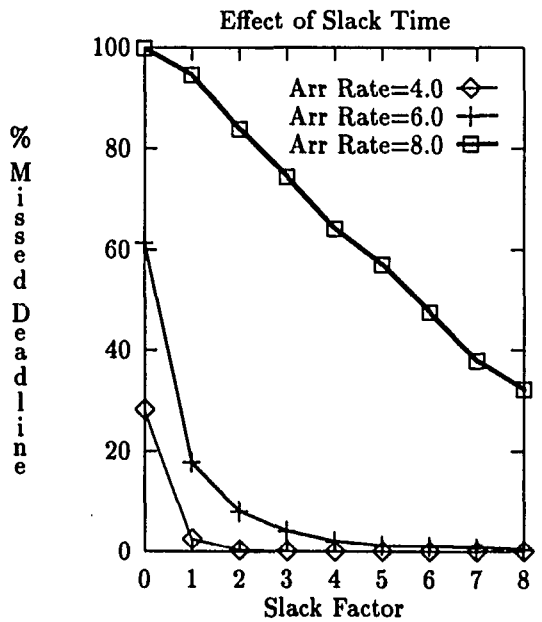


Figure 4.51: FCFS with BLOCK.

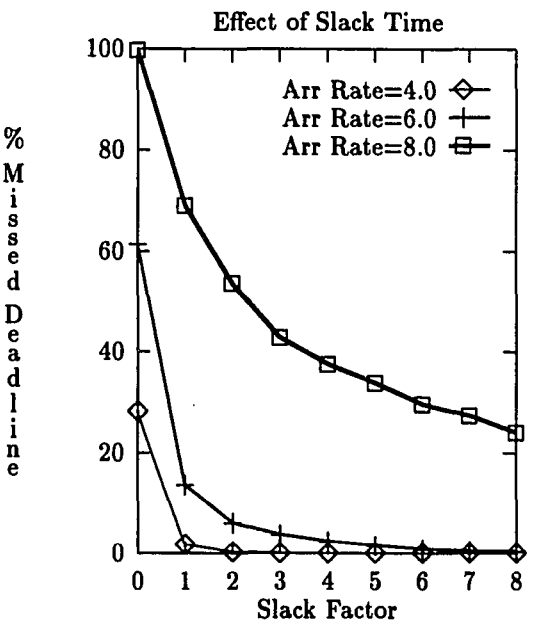


Figure 4.52: MSTF with BLOCK.

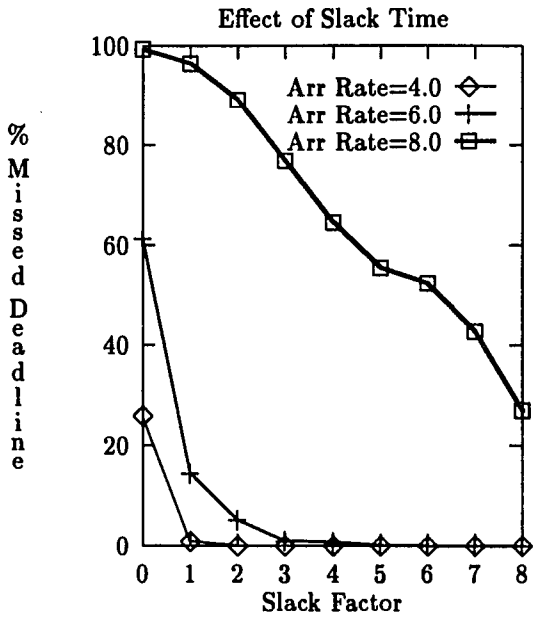


Figure 4.53: EDF with HIPRTY.

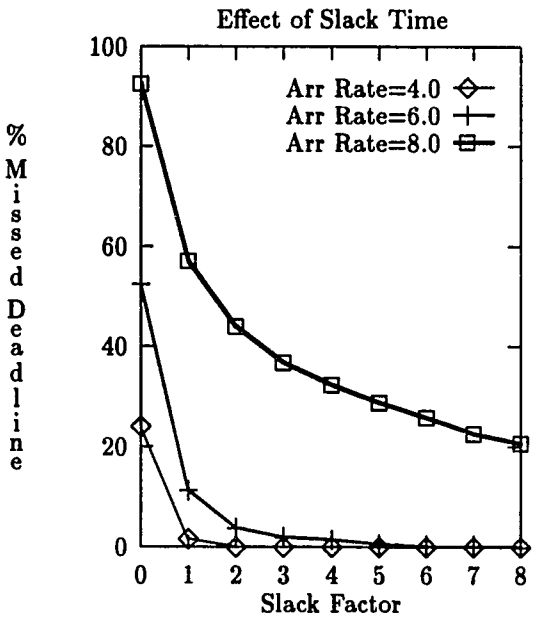


Figure 4.54: SJF with HIPRTY.

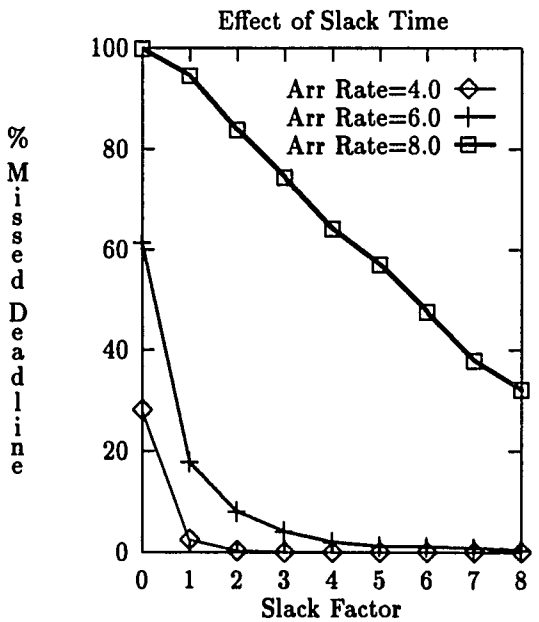


Figure 4.55: FCFS with HIPRTY.

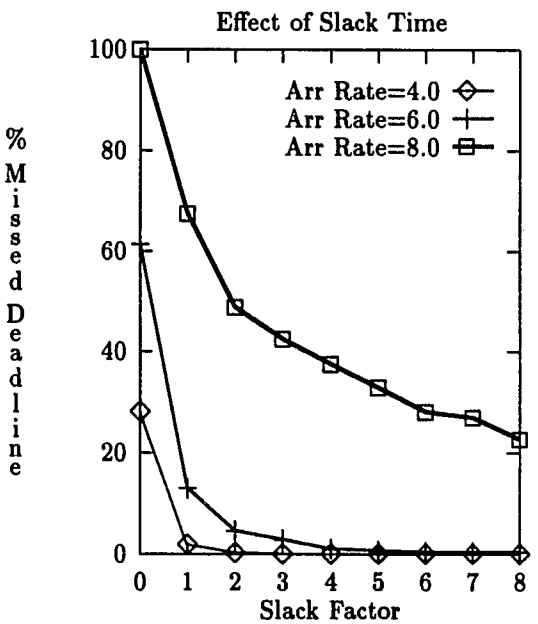


Figure 4.56: MSTF with HIPRTY.

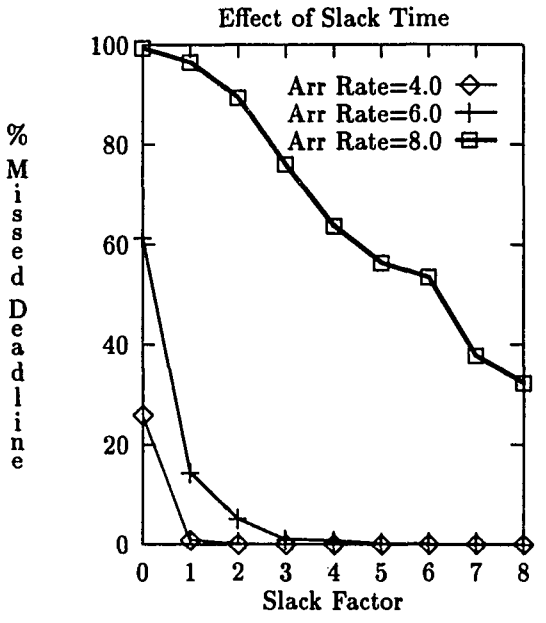


Figure 4.57: EDF with CPR.

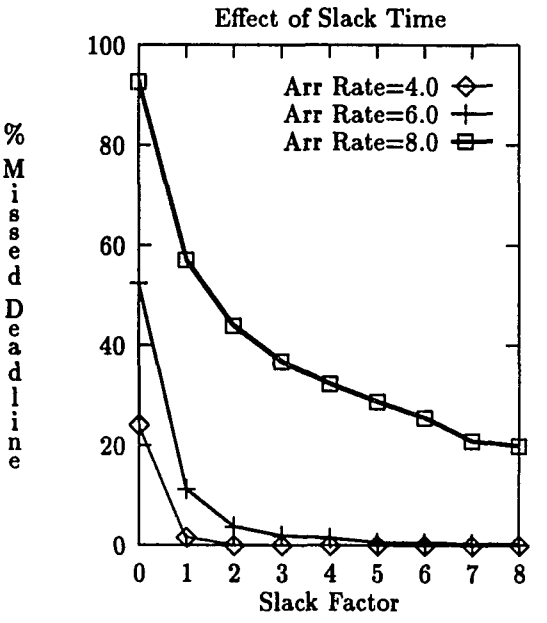


Figure 4.58: SJF with CPR.

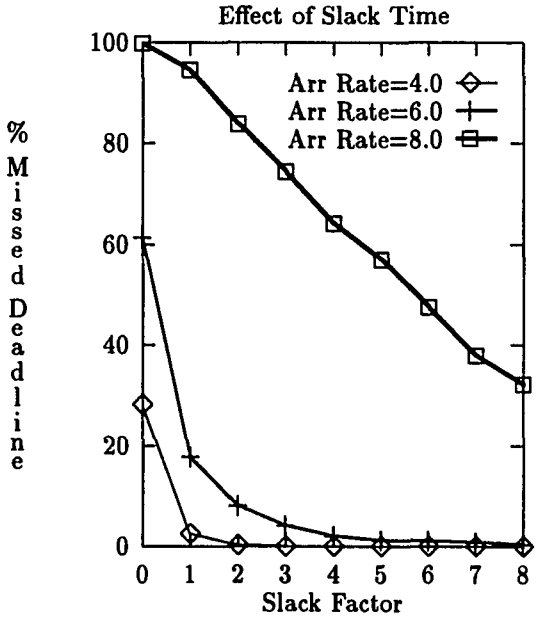


Figure 4.59: FCFS with CPR.

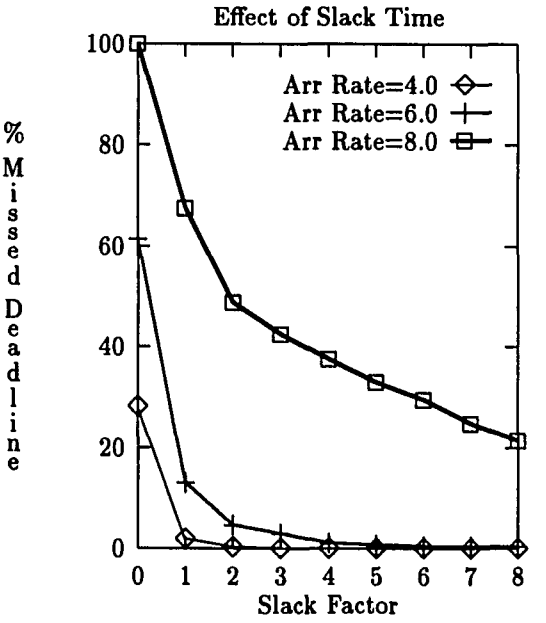


Figure 4.60: MSTF with CPR.

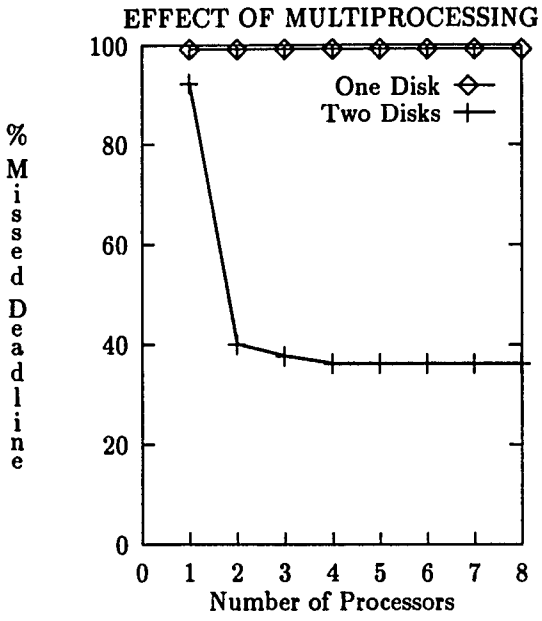


Figure 4.61: EDF (BLOCK) - 8 jobs/s

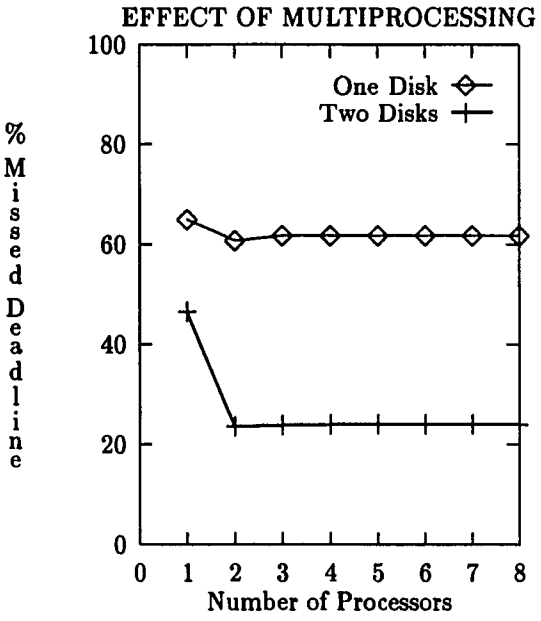


Figure 4.62: SJF (BLOCK) - 8 jobs/s

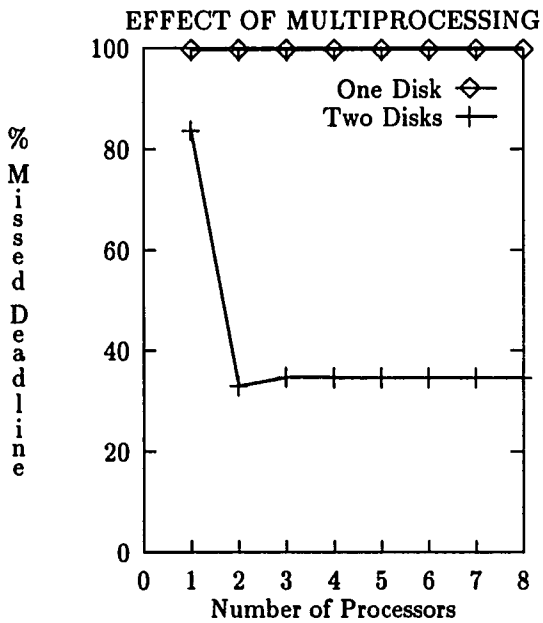


Figure 4.63: FCFS (BLOCK) - 8 jobs/s

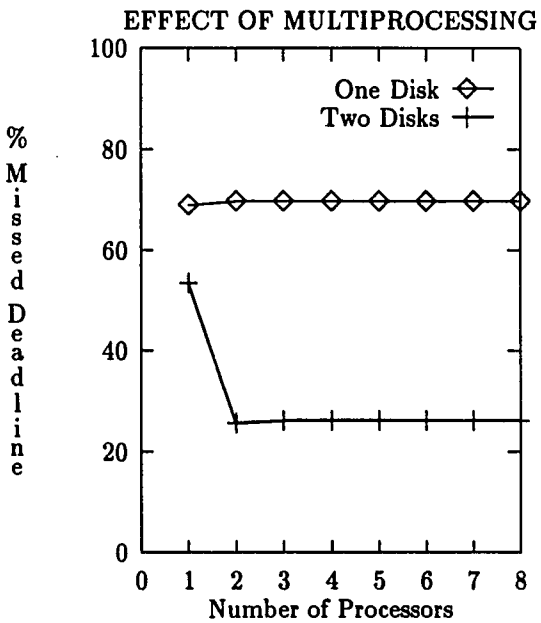


Figure 4.64: MSTF (BLOCK) - 8 jobs/s

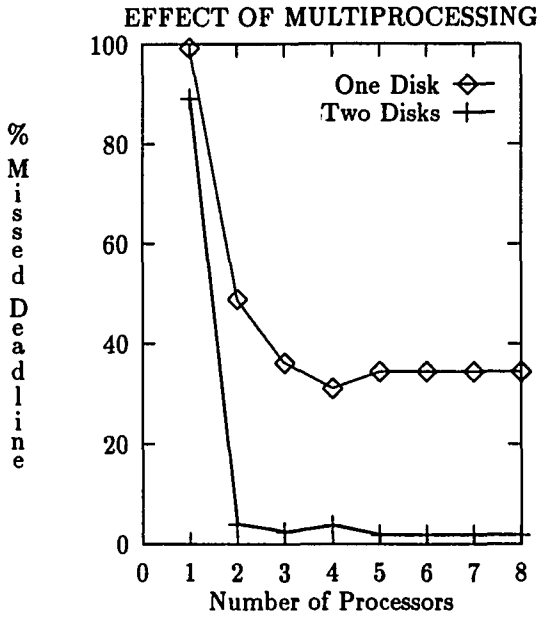


Figure 4.65: EDF (HIPRTY) - 8 jobs/s

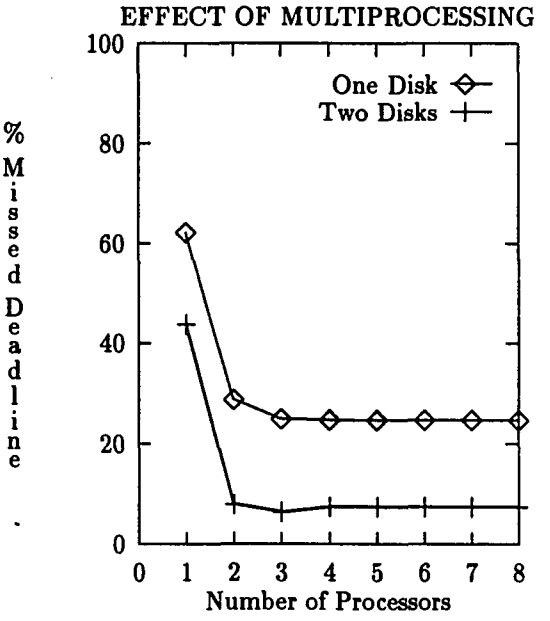


Figure 4.66: SJF (HIPRTY) - 8 jobs/s

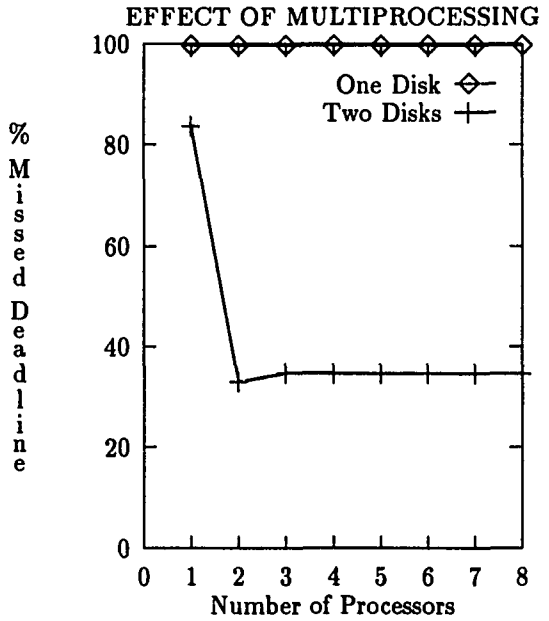


Figure 4.67: FCFS (HIPRTY) - 8 jobs/s

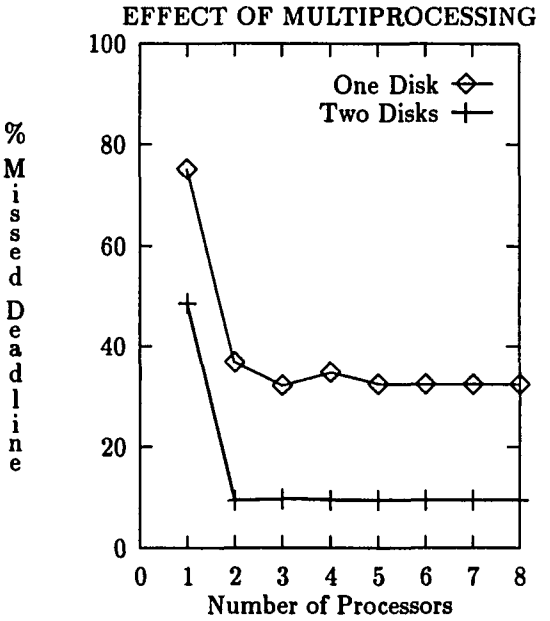


Figure 4.68: MSTF (HIPRTY) - 8 jobs/s

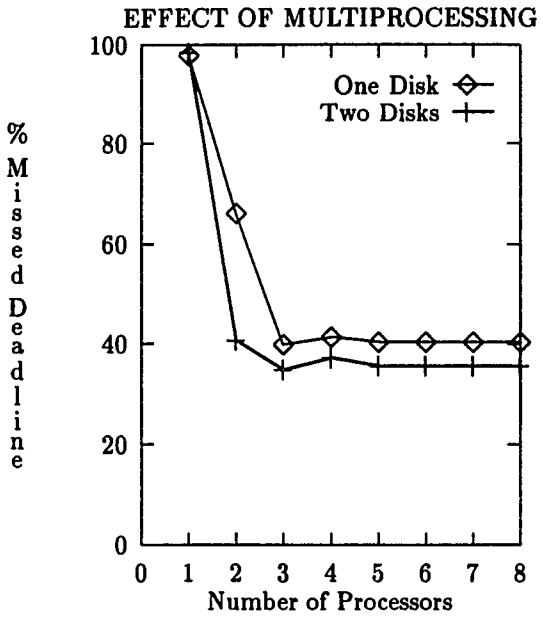


Figure 4.69: EDF (CPR) - 12 jobs/s

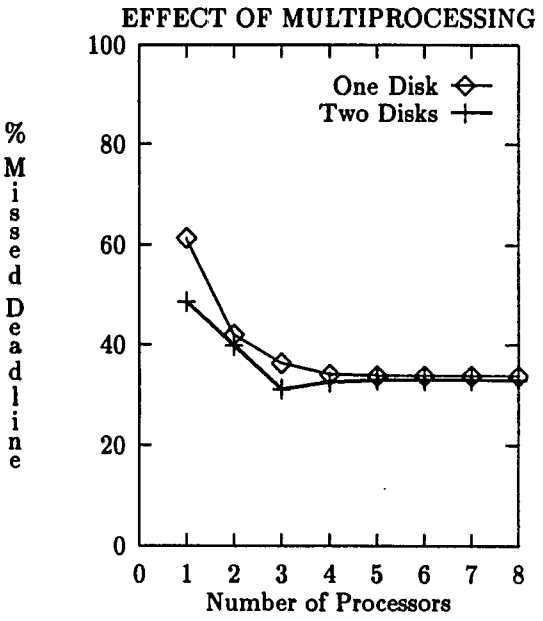


Figure 4.70: SJF (CPR) - 12 jobs/s

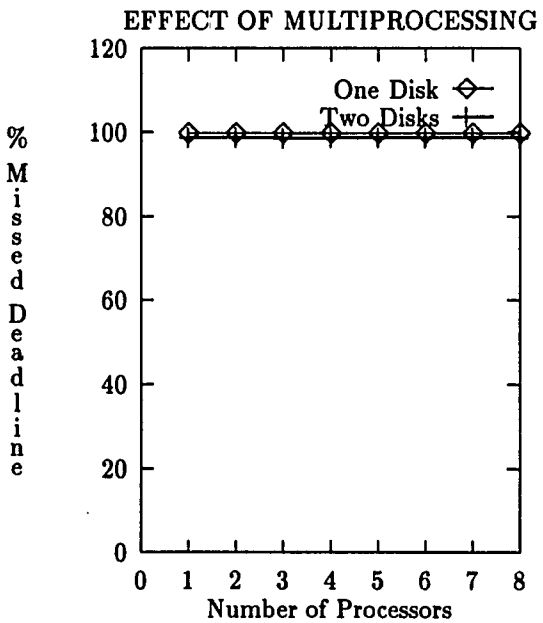


Figure 4.71: FCFS (CPR) - 12 jobs/s

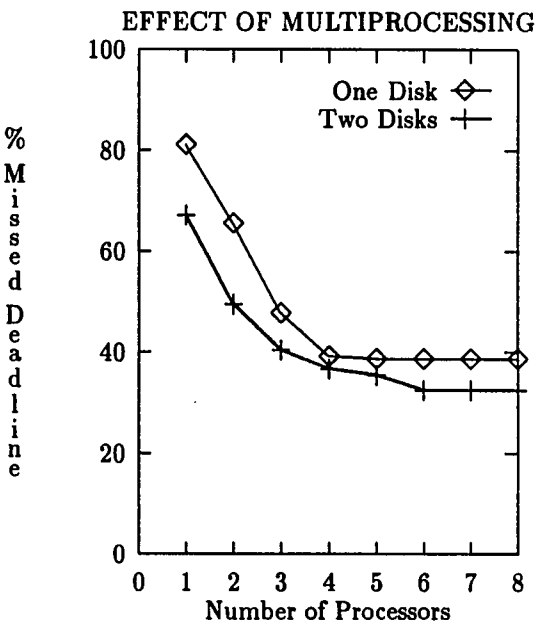


Figure 4.72: MSTF (CPR) - 12 jobs/s

CHAPTER 5. REAL-TIME NESTED TRANSACTIONS

Nested Transactions

Nested transactions are extension of traditional single-level transactions and exhibit properties which make them feasible for many distributed applications. In a nested transaction system, any transaction can invoke *child* transactions nested within it. The transactions are not only synchronized at the top level, but the transaction's descendants are also synchronized among themselves. This allows concurrent access to shared data within a transaction while satisfying the consistency constraints of the database. Thus, nested transactions inherently permit safe concurrency. Another advantage of nested transactions over single-level transactions comes from failure transparency because subtransactions of a nested transaction fail independently of each other and independently of the top-level (parent) transaction. Such graceful and controlled failures allow possibilities of partial redoing; for example, redo only the part of computation that failed while executing on a certain machine. In a single-level transaction system, if any part fails, the whole transaction fails. Because of these properties, nested transactions can be very suitable for distributed applications.

The serializability of nested transactions is ensured via read-write locking protocols that ensure nested synchronization. An advantage of nested synchronization is that the execution of two concurrent transactions from the same parent transac-

tion is correct even when they access same data item. Rules for synchronization and recovery in such an environment are presented by Moss in [31].

The concept of nested transactions is not new but an important contribution of Moss' work is the introduction of lock inheritance protocols which ensure serializability within and among all nested transactions. In [31], Moss first provides rules for exclusive and read-write locks in a single-level transactions system and then extends the rules for nested transactions. Similarly, recovery rules are provided for both single-level and nested transactions.

Nested transactions can be implemented in a centralized or distributed system, but their efficient synchronization and graceful recovery properties make them suitable for the distributed computing environment. Remote procedure calls and distributed databases are examples of the potential applications in such an environment. It has also been shown that management of nested transactions in a distributed environment can be accomplished by simple bookkeeping, by extending two-phase commit protocol, and by detecting and aborting orphan transactions [31].

Real-Time Nested Transactions

In this chapter, we examine the concept of nested transactions in the context of real-time database. In such an environment, the transactions have timing constraints associated with them and it is important to maximize the number of transactions that meet their deadlines. We attempt to extend our single-level real-time transaction model to nested transactions environment using concepts of priority assignment and inheritance. Assigning a priority according to a certain protocol is essential to determine the importance of the task. However, as discussed for single-level transactions in

earlier chapters, priority driven schemes have a potential problem of creating priority inversion which is the phenomenon where a higher priority transaction is blocked by lower priority transactions. In many cases, the duration of blocking is unpredictable and may be indefinite. Two variants of priority inheritance protocols that avoid priority inversion in a uniprocessor environment and also impose an upper bound on the blocking time for a higher priority transaction are provided in [39, 40, 41].

The following example illustrates that, in addition to lock inheritance protocol of traditional nested transactions, it is important that the parent transactions also inherit the priority so that a waiting transaction is not delayed indefinitely.

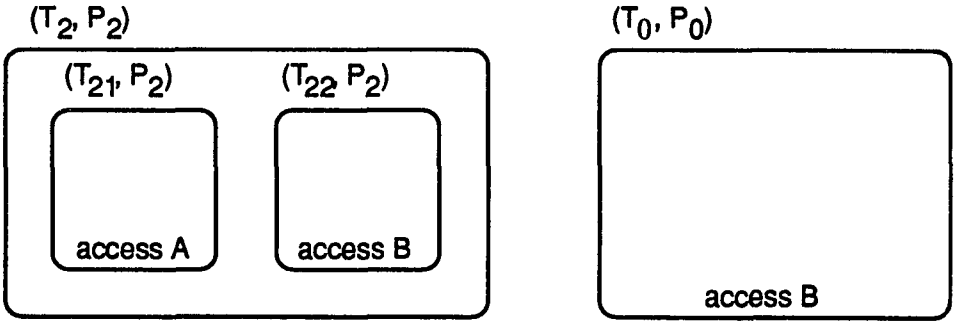


Figure 5.1: An Example of the Priority Inversion Problem

Example 5.1. Let T_0 , T_1 , and T_2 be three transactions in descending order of priority P_0 , P_1 , and P_2 respectively (Figure 5.1). Assume that the child transactions execute at their parent's priority and no priority inheritance protocol is used; that is, the priority of a transaction does not change during the course of its execution. Let T_{21} and T_{22} be the child transactions of T_2 with T_{21} accessing A

and T_{22} accessing B. Consider that while T_{21} and T_{22} are executing, T_0 arrives and wants to access B. Since B has been locked by T_{22} , T_0 will have to wait in spite of the fact that T_0 's priority is higher than that of T_{22} . While T_0 is waiting, the following scenarios can take place:

Scenario 1. T_{22} completes; T_2 inherits lock on B.

T_0 still cannot access B.

T_{21} completes; T_2 inherits lock on A.

T_2 commits and releases locks.

T_0 starts execution and can access B.

Scenario 2. T_1 arrives, does not need to access B, and preempts T_{22} .

T_1 executes and completes.

T_{22} resumes execution.

Finally, T_{22} completes.

T_{21} completes and then T_2 commits.

T_0 starts execution and can access B.

Scenario 3. T_{22} completes; T_2 inherits lock on B.

T_0 still cannot access B.

T_{21} accesses A and is executing.

T_1 arrives, does not need to access A, and preempts T_{21} .

T_1 does not need B, executes and completes.

T_{21} resumes execution.

Finally, T_{21} completes and T_2 inherits locks.

T_2 commits and releases locks.

T_0 starts execution and can access B.

In each of the above scenarios, the higher priority transaction T_0 directly or indirectly waits for the lower priority transactions to complete. In scenario 1, T_0 first waits for T_{22} to complete because there is a direct conflict arising from the data contention for the same data object, B. However, even after T_{22} completes, the lock inheritance protocol does not allow the lock on B to be released; instead, the lock is passed on to the parent transaction T_2 . Therefore, T_0 has to wait until the other child transaction also completes so that T_2 can commit and release the locks. Note that T_0 is not allowed to preempt T_{22} because of the data conflict.

Despite an indirect wait for T_0 , scenario 1 appears to have the least blocking duration of the three scenarios described above. In scenarios 2 and 3 the wait for T_0 can be unbounded because there can be other intermediate priority transactions (like T_1) which can keep preempting the lower priority transaction T_2 and thereby indirectly blocking T_0 . In scenario 2 above, a solution to avoid repeated preemptions is to allow T_{22} to execute at a higher priority if it is blocking a higher priority transaction. In this case, T_{22} should be executing at priority P_0 so that no other intermediate priority transaction is allowed to preempt it. However, with this simple solution, even though T_{22} will complete sooner, the lock is inherited by the parent, T_2 , and will not be released until the other child transactions of T_2 also complete (in this case, T_{21}). Thus, despite raising the priority of T_{22} for reducing the waiting time, T_0 may now be indirectly blocked for an unbounded period of time because of T_{21} as described in scenario 3. In other words, this requires that not only T_{22} (which is directly involved in the conflict) should inherit the priority of waiting transaction but all nested transactions which can directly or indirectly block the higher priority

transaction should inherit the priority. In the above example, this requires all the three low priority transactions, T_2 , T_{21} and T_{22} to be executing at priority P_0 when T_0 is directly blocked by T_{22} .

As illustrated by this example, a simple priority inheritance protocol is not enough in a nested environment. In fact, the propagation of inherited priority seems to become unavoidable. To solve the unbounded waiting problem for high priority transactions, we propose new protocols for nested transactions.

Definitions, Notations and Concepts

In this section, we define a few terms and concepts that will be used in subsequent discussion of the formal proof of properties of the proposed protocols.

Figure 5.2 shows the structure of our real-time nested transaction system. We assume that our system is composed of several active tasks each one of which can have a sequence of embedded *top-level transactions* and non-database operations. Each top-level transaction can have child transactions nested within it as described earlier. For notation purposes, we use a scheme similar to that proposed in [32]. A task j is denoted by τ_j and all its subtransactions are denoted by $T^j_{d_1 \ldots d_n}$ where n depends on the depth of nesting. Thus, a full transaction identity, tid, consists of a sequence of subtransaction's tid's concatenated together. The first subtransaction tid would represent the top-level task, followed by the subtransaction tid of the next level, and so on down the tree to the subtransaction being identified. Therefore, a tid identifies all the ancestors of its transaction by explicitly enumerating them along the path from the root to the transaction. As illustrated by Figure 5.2, tid's are variable in length. We expect that transaction nesting will not be very deep. In the

following sections, we denote transactions by T_i when it is not important to know the identification of the ancestors. The priority of a transaction T_i is denoted by P_i. We denote P_H as the priority of a higher priority transactions as opposed to P_L, priority of a lower priority transaction.

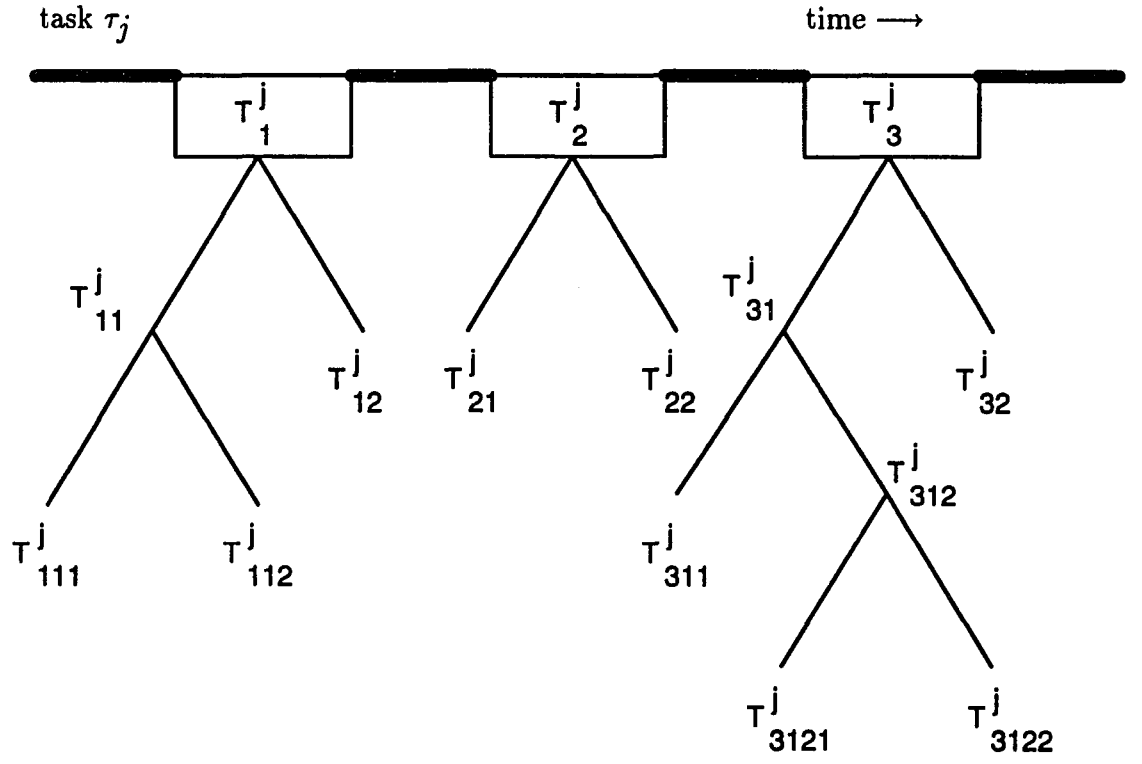


Figure 5.2: Structure of a Task with Nested Transactions

When two transactions attempt to access shared data, the access must be serialized in order to maintain consistency. If the lower priority transaction gains access first and then the higher priority transaction request access to the shared data, this higher priority transaction must wait until the object is no longer accessed by the

lower priority transaction. More formally, a high priority transaction is said to be *blocked* if it is waiting for other lower priority transactions to complete. The lower priority transactions in this case are said to be the *blocking* transactions.

As suggested in Example 5.1, under certain blocking conditions, a transaction's priority may change during its execution. For example, when a lower priority transaction T_k^j of task τ_j holds a lock and directly blocks higher priority transactions, the transaction T_k^j should execute at the highest priority of all transactions blocked by T_k^j. In this case, we say that the transaction T_k^j temporarily assumes a higher priority because of *priority inheritance.* The priority of a transaction may also temporarily be raised by indirect blocking of other transactions. For example, in the above case, though T_k^j is directly blocking other higher priority transactions by holding a required lock, even after it completes, the locks are not released but inherited by its parent. Therefore, we require that all other subtransactions $T_{d_1 \ldots d_n}^j$ within the nested transaction also execute at the priority inherited by the transaction T_k^j which is directly blocking higher priority transactions. To accomplish this, the transaction T_k^j that inherits a higher priority propagates this information to its ancestors so that all other transactions $T_{d_1 \ldots d_n}^j$ also execute at this elevated priority. Such higher priority assumed by indirect blocking is said to be the result of *priority propagation.* When the blocking nested transaction completes, the execution of the task continues at the original priority assigned at the time of initiation of the transaction.

Each data object O in the database has two fixed and one dynamic priority ceiling associated with it. The *write priority ceiling* (WPC) of a data object is defined as the priority of the highest priority task that may write this object. The *absolute priority ceiling* (APC) of a data object is defined as the priority of the highest priority task

that may either read or write this data object. When a data object O is write-locked, the *dynamic priority ceiling* (DPC) of O is defined to be equal to APC of O. When a data object O is read-locked, the DPC of O is defined to be equal to the WPC of O.

The above priority adjustment rule is similar to that proposed by [41] and is based on the following observation: when a task write-locks a data object O, O should not be read or written by any other task, and when a task read-locks a data object O, O should not be written by another task. The first condition is ensured by setting the DPC of O to its APC because then no other task can either read or write O until the lock on O is released. The second condition is ensured by setting the DPC of O to its WPC because then no other task can write O until the lock on O is released. Note that this allows read transactions with priorities higher than WPC of O to share the read-lock on O. This, at first, may appear too restrictive because we do not allow read transactions with priorities lower than or equal to WPC of O to share the read-lock on O, but such a restriction comes with an important advantage. By not allowing lower priority transactions to share the read-locks, we reduce the waiting time for a blocked higher priority write transaction. Otherwise, this transaction would wait for multiple readers resulting in the task to be blocked by multiple lower priority embedded transactions.

Dynamic Status Vector (DSV)

A transaction T can be executing at its original priority or a higher priority attained as a result of priority inheritance or priority propagation. At times, when a transaction aborts, it may be necessary to reevaluate the priority of the transaction

T. To accomplish this, it is necessary to maintain a list of priorities that a transaction may have assumed from time to time. We, therefore, require a *dynamic status vector* (DSV) associated with each transaction T. This vector represents the set of other higher priority transactions that are blocked by T. The elements of this set consist of 2-tuples $< tid, P_{tid} >$ ordered by the priority value P_{tid}. This value represents the priority that a transaction might have assumed (through priority inheritance or priority propagation) from time to time.

The DSV of a transaction T is updated whenever more transactions are blocked or when a blocked or a blocking transaction aborts according to the following update rules:

Rule U1. When a transaction *tid* is blocked, an ordered pair $< tid, P_{tid} >$ is added to the DSV associated with the blocking transaction T. The priority of T is then set to the highest priority of all the blocked transactions, that is,

$$P_T = \max_{tid \in DSV} \{P_{tid}\}.$$

Rule U2. When a blocked transaction aborts, if its priority is the highest of all the blocked transactions, the blocking transaction starts executing at the next highest priority level in the DSV. No priority change is required if the priority of the aborting blocked transaction is lower than the currently inherited priority of the blocking transaction. In either case, however, the DSV is updated by removing the corresponding tuple $< tid, P_{tid} >$ of the aborted transaction.

Rule U3. When the blocking transaction aborts, the information is propagated to all ancestors so that they may update their corresponding DSV and readjust the priority.

Locking and Priority Inheritance Protocols

As described earlier, locking, timestamps and optimistic methods have generally been used to achieve serializability which is an accepted criteria for correctness of concurrently executing transactions. To ensure serializability in our real-time nested transactions, we select to use locking because of its well-known properties, simplicity and ease of implementation. Locking protocols restrict access to a database object by requiring a lock to be obtained before any read/write can take place. The two-phase locking protocol adds another restriction to this rule: no transaction obtains a lock after it has released one. In the first phase of this protocol, a transaction can only acquire (does not release) locks, and in the second phase the transaction only releases (does not acquire) locks. This protocol ensures that the order in which any two transactions access the same object is the same as the order in which those transactions access any other objects. The underlying assumption is that if two schedules result in the same order of access at each object then the schedules are equivalent. Given this equivalence relation on schedules, it is possible to prove that if the two-phase locking protocol is used then any such schedule is equivalent to a serial schedule [17]. That is, two-phase locking ensures serializability. Therefore, all protocols described in this section assume that locking is done using the two-phase locking protocol.

Basic Priority Inheritance Protocol with Exclusive Locks

We first provide a set of rules assuming that transactions follow a basic priority inheritance protocol and can lock objects in an exclusive mode. It is also assumed that all transactions are assigned a priority according to some priority assignment protocol

(for example, earliest-deadline-first) and a transaction executes at this priority until it blocks other transactions. When blocking higher priority transactions, the priority of the executing transaction may only change through inheritance and propagation as described in the following rules.

Rule A1. A transaction may access an object only if it can lock the object in an exclusive mode.

Rule A2. A transaction may lock an object if and only if all other transactions holding the lock on the object are ancestors of the requesting transaction.

Rule A3. A transaction executes at its assigned priority unless it is blocking higher priority transactions. When a transaction T blocks higher priority transactions, T inherits and executes at P_H , the highest priority of transactions blocked by T.

Rule A4. When a transaction inherits a higher priority, it immediately propagates that information to its parent transaction. Upon receiving this information, the parent transaction upgrades its priority and starts executing at the higher priority. The parent also propagates this information to all its ancestors and descendants so that ultimately the top-level transaction and all its descendant transactions are executing at the inherited priority.

Rule A5a. When a child transaction commits, its parent transaction (if any) inherits the locks held by the committing transaction.

Rule A5b. When the transaction aborts, its locks are discarded (after undoing the transactions effects) and any priority inheritance that might have occurred

by propagation of priority information needs to be re-evaluated based on the update rules provided in the section describing the Dynamic Status Vector.

Rule A6. A transaction T_0 can preempt another transaction T_1 only if T_0 is not blocked and its priority is higher than the inherited or assigned priority at which T_1 is executing.

Rule A1 states that locking an object is necessary before accessing it in order to ensure data consistency. Rule A2 ensures that the ancestors of a transaction do not interfere with their children's ability to hold locks. Rule A3 expresses the need for priority inheritance when a lower priority transaction is blocking a higher priority transaction.

Rules A4 and A5 describe the protocols for lock and priority inheritance. Note that we require the priority inheritance to take effect immediately whereas the locks are inherited by parent transactions only after the child transaction has committed. As illustrated earlier, this is essential to bound the delay of a blocked higher priority transaction. When a transaction aborts, the priorities of its ancestors and other subtransactions may have to be changed. A transaction T can be executing at a priority higher than its initial priority under two circumstances: T is directly blocking a higher priority transaction and, therefore, inherited the priority, or a higher priority is propagated to it through its parent. If the aborting transaction had been executing at a higher priority through propagation from its parent then the priorities need not be readjusted. This is because the transaction which is directly blocking higher priority transactions and caused priority propagation is still executing and requires all other transactions to continue to execute at the same priority. This is implemented using the DSV maintained by each transaction as described earlier. This vector is assumed

to contain all priorities that a transaction may have inherited while blocking other higher priority transactions. We require that the operations of priority inheritance, priority adjustment and resumption of original priority must be indivisible. Rule A6 outlines the preemption protocol.

For all of our protocols, we assume $none < read < write$ ordering for locks. This implies that a write lock is the most exclusive of all the locks. Using this ordering, rule A5 describes that parent's new lock mode is set as follows when a child commits:

$$\text{parent's new lock mode} = max(\text{parent's current mode,child's mode})$$

Further, a transaction's mode also obeys the rule that the lock mode never decreases. Thus, whenever a transaction requests and is granted a lock, it holds the lock in the maximum of the requested mode and the mode in which it previously held the lock:

$$\text{new lock mode} = max(\text{requested mode,old mode})$$

Basic Priority Inheritance with Read-Write Locks

Exclusive locks are too restrictive and are orthogonal with timing constraints imposed by real-time transactions. In order to enhance concurrency which eventually reduces the delays and the number of transactions that miss deadlines, it is highly desirable to allow both shared and exclusive locks. Shared locks, however, demand protocols that can ensure data consistency. The following rules allow for using read and write locks.

Rule B1r. A transaction may *read* an object only if it can lock the object in either read or write mode.

Rule B1w. A transaction may *write* an object only if it can lock the object in write mode.

Rule B2r. A transaction can lock an object in read mode if and only if all other transactions holding the lock on the object in write mode are ancestors of the requesting transaction.

Rule B2w. A transaction can lock an object in write mode if and only if all other transactions holding the lock on the object in any mode are ancestors of the requesting transaction.

Rule B3. Same as rule A3.

Rule B4. Same as rule A4.

Rule B5a. When a transaction commits, its parents (if any) holds the locks held by the committing transaction. If the parent already holds the lock, it will choose the more exclusive mode of the two lock modes.

Rule B5b. Same as rule A5b.

Rule B6. Same as rule A6.

Rules B1r and B1w states that locking an object in the appropriate mode is necessary before accessing it in order to ensure data consistency. Rules B2r and B2w allow sharing of read locks with other transactions and also ensure that the parents of a transaction do not interfere with their children's ability to hold locks. Rule B5a describe the protocols for lock inheritance. Note that we require the parent to inherit the lock in the more exclusive mode if it is already holding the lock. All other rules and their description is similar to that discussed in the previous section.

Priority ceiling protocol with read-write locks

The basic priority inheritance protocols have two inherent problems: first, they do not prevent deadlocks and second, a blocking chain can be formed resulting in a substantially large duration of blocking. To overcome these problems, we first define a new protocol and then prove that this protocol does not suffer from the inherent disadvantages of the basic priority inheritance protocols. Further, we also include state restoration capability under this protocol and provide rules for maintaining enough state information necessary to restore an old state in case of an abort.

Rule C1r. Same as rule B1r.

Rule C1w. Same as rule B1w.

Rule C2. Let S_O be the set of all objects that are currently locked by transactions other than those of τ. Let O^* be the data object with the highest dynamic priority ceiling of all data objects in S_O. When task τ's transaction attempts to read-lock (write-lock) a data object O, the lock will be granted only if both of the following conditions hold true:

i) task τ's priority is higher than the dynamic priority ceiling of O^*, and

ii) all other transactions holding the lock on object O in write (any) mode are ancestors of the requesting transaction.

If either of the above conditions do not hold, the lock will be denied.

Rule C2b. When a transaction starts to hold a *write* lock on an object, the restoration information sufficient to restore the object's current state is created and becomes the associated state for that object and transaction. This is done only

if there is not already an associated state for the same object and transaction. Note that an associated state might already exist if one of the transaction's children modified the object and committed.

Rule C3. Same as rule A3.

Rule C4. Same as rule A4.

Rule C5a. When a child transaction commits, the following actions are taken:

(i) its parent transaction (if any) inherits the locks held by the committing transaction, and

(ii) each of its associated states is offered to the committing transaction's parent. The parent accepts each state (making it the parent's own associated state for the same object) if and only if the parent does not already have an associated state for the same object.

Rule C5b. When the transaction aborts, the following actions are taken:

(i) its locks are discarded (after undoing the transactions effects) and any priority inheritance that might have occurred by propagation of priority information needs to be re-evaluated based on the update rules provided in the section describing the Dynamic Status Vector.

(ii) each of its associated states is used to restore the objects directly or indirectly modified by the transaction. Those associated states can then be discarded.

Rule C6. Same as rule A6.

Rule C2 ensures that no deadlock or chained blocking is possible. Condition (i) is a requirement related to all transactions other than those of τ whereas condition (ii) is related to all the ancestors of requesting transaction. Note that the transaction T of τ may or may not need any data items from the set S_O. Similarly, O may or may not be accessed by transactions other than those of τ. Therefore, O may or may not belong to S_O. If O belongs to S_O, then condition (ii) does not hold true and the lock will be denied even if condition (i) holds. If O does not belong to S_O, then it cannot be locked by any other transaction than those of τ's; in this case, if condition (ii) holds, then the lock is granted if task τ's priority is higher than the dynamic priority ceiling of O^*, that is, if condition (i) also holds.

Rule C2b requires that sufficient information about a data object be saved to restore the state in case of transaction aborts, communications failures or node crashes. The real state of an object is recorded in permanent memory. Transactions work on a volatile memory copy, which is backed up to permanent storage copy at the appropriate time. In case a transaction aborts, the state can be restored from the volatile memory copy. In case of node crashes, the permanent memory copy can be used to restore the state. Rule C5a(ii) ensures that if the parent does not have an associated state for one of the objects, then the associated state will not be lost, and the object will be correctly restored in case of a later abort. However, if the parent already has an associated state for the object, then the one the parent has should take precedence, because the parent's associated state is earlier than that of the child's. All other rules and their description is similar to that discussed in the previous section.

An important advantage of Rule C2 is that there will not be any read-write conflicts on the object O and we need not check if O has been locked. In other

words, under this protocol, we need not explicitly check for the possibility of read-write conflicts. For instance, when an object O is write-locked by a transaction T, $DPC(O)$ is equal to the priority of the highest priority transaction that can access O. Hence, the protocol will block a high priority transaction that may want to write or read O. On the other hand, suppose that the object O is read-locked by T. Then $DPC(O)$ is equal to the priority of the highest priority transaction that may write O. Hence, a transaction that attempts to write O will have a priority no higher than $DPC(O)$ and will be blocked. Only the transactions that read O and have priority higher than $DPC(O)$ will be allowed to read-lock O. This is not a problem because read-locks are compatible.

Properties of Proposed Protocols

In this section, we formally prove that our proposed protocols are free from deadlock and enforce tightly bounded waiting period for higher priority transactions. Similar proofs for a uniprocessor environment are provided in [41]. Freedom from deadlock guarantees progress and together with the bounded waiting period provides a solution to the priority inversion problem. Theorem 5.2 applies to the global structure of a real-time transaction processing system where each embedded nested transaction is considered to be a unit. We assume that each unit by itself is free from deadlocks.

Lemma 5.1 *Under the priority ceiling protocol, each transaction will execute at a higher priority level than the level that the preempted transaction can inherit.*

Proof. Let $S_0 = \{O_1, \ldots, O_n\}$ be the set of objects locked by a task τ. Then, highest priority that the task τ or its transaction T can ever inherit is

$$max\{DPC(O_1), \ldots, DPC(O_n)\} \quad \ldots\ldots\ldots \quad (i)$$

By Rule C2 of the priority ceiling protocol, the requesting task τ_R will have its transactions execute only if

$$P(\tau_R) > P(O^*) \quad \ldots\ldots\ldots \quad (ii)$$

Case (i). $O^* \in S_0$. In this case, $P(T) = P(O^*)$.

Case (ii). $O^* \notin S_0$. In this case, $P(T) < P(O^*)$.

So, in either case,

$$P(T) \leq P(O^*) \quad \ldots\ldots\ldots \quad (iii)$$

From (ii) and (iii), if any transaction T_R executes, it will execute at a priority greater than $P(T)$, the priority that the preempted transaction T can inherit.

Theorem 5.2 *There is no deadlock among the tasks under the priority ceiling protocol using read/write locks.*

Proof. Suppose a deadlock can occur. Let $\tau_1, \tau_2, \ldots, \tau_n$ be the tasks involved in the deadlock. Let $P = max[P(\tau_1), P(\tau_2), \ldots, P(\tau_n)]$. Since priority inheritance and propagation is transitive, eventually

$$P(\tau_1) = P(\tau_2) = \ldots = P(\tau_n) = P$$

which implies that all transactions will eventually be executing at the same priority. However, this contradicts lemma 5.1 and hence the theorem follows.

Lemma 5.3 *Under the priority ceiling protocol, until task τ either completes its execution or suspends itself, task τ can be blocked for at most a single embedded nested transaction of a lower priority task τ_L, even if τ_L has several embedded transactions.*

Proof. Let a task τ be blocked by a lower priority task τ_L. By theorem 5.2 there will be no deadlock. This implies that at some time t_1, τ_L will exit its current transaction and start executing at its original priority. At this time t_1, τ_L will be preempted by τ. Since now τ_L is not in a transaction, it cannot inherit a higher priority until it executes another transaction. But τ_L cannot execute another transaction (or resume execution at all) until τ completes or suspends itself. The lemma follows.

Theorem 5.4 *Under the priority ceiling protocol, until task τ either completes its execution or suspends itself, task τ can be blocked by at most a single embedded nested transaction of one lower priority task, even if there are multiple lower priority tasks.*

Proof. Suppose τ is blocked by n lower priority transactions, where $n > 1$. By lemma 5.3, each one of these transactions must belong to a different lower priority task. Let $\{\tau_1, \ldots, \tau_n\}$ be the set of all these lower priority tasks ordered by priority, that is, $P(\tau_i) > P(\tau_{i+1})$. Now, for all τ_i blocking τ, each τ_i must be in its transaction (because otherwise it could be preempted). Thus, τ is blocked by τ_n implies that $P(\tau_n) = P(\tau)$ (by inheritance). Let P be the highest priority that τ_n can inherit. Since τ_n can block τ,

$$P(\tau) \leq P \quad \ldots\ldots\ldots \quad (i)$$

By lemma 5.1,

$$P(\tau_{n-1}) > P \quad \ldots\ldots\ldots \quad (ii)$$

From (i) and (ii),

$$P(\tau_{n-1}) > P(\tau)$$

and this contradicts with our initial assumption. Thus, τ cannot be blocked by more than one embedded nested transaction of one lower priority task, even if there are multiple lower priority tasks.

Summary

In this chapter, we have introduced the concept of real-time nested transactions. In doing so, we have introduced a framework for real-time database environment based on two independent approaches: nested transactions and priority inheritance. These two approaches are very important to ensure that transactions meet the stringent temporal requirements while maintaining data consistency in a distributed environment. Since nested transactions are particularly suitable for distributed processing, our proposed protocols can be efficiently implemented in a distributed real-time transaction processing environment. A set of protocols for locking, priority inheritance and state restoration is also defined together with a formal proof of the facts that the proposed protocols are free from deadlocks and have tightly bounded waiting period for higher priority transactions. Due to the concurrent execution of subtransactions, the overall performance of the protocol will be enhanced significantly. Furthermore, abortion of a subtransaction will not result in cascading aborts, but will allow restoration to a previously correct state.

Previous studies in this area [40, 41] have not addressed the issues related to aborting transactions. We have introduced a new concept, namely priority propagation, which is particularly effective when the transactions abort in a nested environment. The information regarding the aborted nested transactions need to be propagated for priority readjustment of all other nested transactions that may be directly or indirectly affected. We have proposed implementation of this readjustment by requiring each transaction to maintain a dynamic status vector because the aborted transactions may require reevaluation of the priorities of the other nested transactions. This vector contains enough information to determine the priority that a transaction may assume when another transaction aborts. A formal update protocol has been defined for the dynamic status vector. Since, we do not anticipate frequent abortion of transactions, the overhead of re-evaluating priorities using this vector will be minimal. The ultimate goal of priority propagation and priority inheritance is to solve the priority inversion problem. Using our protocol, we have shown that a high priority transaction can be delayed by at most a single embedded nested transaction of one lower priority task. This bound on the waiting time provides a solution to the problem of unbounded waiting inherent in simple priority inheritance protocols.

The next chapter summarizes our research, provides a discussion of results obtained and gives directions for future research. Based on the encouraging results obtained, we have identified several areas which should be explored by extending the ideas emanating from this research.

CHAPTER 6. CONCLUSIONS

Summary and Discussion of Results

Real-time systems span many application areas. In addition to automated factories, applications can be found in avionics, process control, robot and vision systems, as well as military systems such as command and control. A relatively little knowledge related to issues in real-time transaction processing has resulted in inflexible and expensive design of such systems. Many computing systems now use real-time databases and allow soft deadlines, that is, missing some deadlines will not result in catastrophic circumstances. Some applications that typically allow such soft deadlines can be found in banking, airline reservation and aircraft tracking. Scheduling transactions for real-time databases has received much attention in the very recent years but there are still a number of issues that have not been addressed. A majority of scheduling algorithms reported in the literature perform static scheduling and hence have limited applicability because of *a priori* information requirement.

In real-time database systems, the goal is not only to minimize response time, but to have dynamic, on-line, adaptive scheduling algorithms which ensure that deadlines are met while maintaining consistency of the database. Thus, scheduling algorithms for real-time database require an integrated approach in which the 'schedule' does not only guarantee execution before the deadline, but also maintains data consis-

tency. Serializability is a widely accepted criterion for ensuring database consistency. Serializability requires that the combined action of a group of transactions accessing the database is equivalent to some serial schedule, that is, the same as if all the transactions would have executed serially in some order.

Our research is clearly divided into two parts. First, we have developed a real-time transaction processing model and used heuristics based protocols to study the performance in a centralized environment. Several experiments have been conducted to study the behavior of the proposed protocols and transaction parameters in a real-time transaction processing environment. Some system parameters are modified to study the effect of the underlying system configuration on performance. The testbed developed can be used to perform numerous other studies for this environment. Second, we propose new concepts for real-time transaction processing in a distributed environment. We have developed protocols and provide formal proof that the protocols do exhibit the properties that make them very suitable for distributed processing.

Centralized Environment

We have studied the problem of scheduling real-time transactions under a common framework which considers both concurrency control issues and the real-time constraints. By using efficient scheduling algorithms, the performance of a real-time transaction processing system can be significantly enhanced. The performance can be further improved by fine tuning parameters that control the underlying system configuration. In order to prove these assertions, we defined a real-time transaction processing model for a centralized system. Various components of the model interact with each other to achieve the goal of maximizing concurrency control and meeting

real-time constraints at the same time. In order to test the behavior of the model under the proposed protocols, we have developed a real-time transaction processing testbed using discrete event simulation techniques. Preemption, which is not available in the original language is implemented using other data structures. Different protocols have been found to work better under different load scenarios and the overall performance is significantly enhanced by modifying the underlying system configuration. We also study the effect of altering various system and transaction parameters on the overall performance of real-time transaction processing.

The results from our simulation model indicate that a significant performance improvement can be achieved by simple modifications to the underlying system configuration for real-time transaction processing. For instance, there is a four-fold improvement in the mean tardy time and the average response time just by partitioning the data on two disks. Similarly, by using a simple buffer management scheme and by allowing preemption based conflict resolution policies, the performance is significantly enhanced. For instance, the mean tardy time and the mean response time are improved by a factor of 16 for the EDF and FCFS schemes between the NN1 and YY2 configurations at an arrival rate of 8 jobs/sec. Further improvement is obtained by allowing shared read-locks.

The choice of locking mode can also significantly improve the performance because when locks are shared the waiting time of transactions decrease and less preemptions are required. In one experiment we observed that the number of preemptions drop from 78 to 30 when shared locks were used as opposed to using exclusive locks only. This results in number of transactions missing deadlines to improve by about 16%. The mean tardy time and the average response time also improve by 9% and

11% respectively.

We have also studied the effect of I/O requirements of a transaction on the performance of a real-time transaction processing system. Since the transactions spend a significant amount of time doing the I/O, it has been observed that a system with read-only transactions performs better compared to the one where transactions update the database frequently. Frequent updates result in a higher I/O contention which eventually degrades performance. The effect of priority assignment schemes depends on both the system load and the conflict resolution policy used.

The experiments on the slack time have shown that regardless of the priority assignment scheme and the concurrency control protocol used, a very high percentage of transactions miss their deadlines when the deadlines are tight (smaller slack). In fact, for a job arrival rate of 8.0 jobs per second (a high load scenario), none of the transactions could meet its deadline when EDF, FCFS or MSTF is used for priority assignment. For low and medium load levels, all the transactions meet their deadlines when the slack factor is high. However, for higher loads, even at a slack factor of 8, a significant number (22-33%) of transactions still miss their deadlines. A general observation from all these figures is that a higher slack factor does result in reduced miss rate.

Our experiments with multiprocessing environment does confirm that providing unlimited instances of a resource does not always result in a continuous performance enhancement. The reason for this is that other limited resources tend to become the bottleneck for the system. After a certain level of improvement, any further gain in performance can only be achieved by first removing the bottleneck.

Distributed Environment

Priority inversion is said to occur when a high priority transaction must wait for the execution of lower priority transactions. Even worse, the duration of such a blocking can also become unbounded and prolonged durations of blocking may lead to the missing of deadlines even at a low level of resource utilization. A common approach to bound such arbitrary delays is to execute the transaction that holds the lock at a higher priority. However, simple priority inheritance schemes have inherent disadvantages.

Nested transactions, an extension of traditional atomic transactions, permit safe concurrency within as well as among transactions and also enable transactions to fail partially in a graceful and controlled manner. Thus, nested transactions have at least two advantages over traditional single-level transactions. First, nested transactions provide appropriate synchronization between concurrently running parts of the same transaction. This implies that more work can be processed concurrently without the danger of inconsistencies arising through improper concurrent access to data. Second, subtransactions of a nested transaction fail independently of each other and independently of the containing transaction. This allows possibilities such as attempting a part of a computation at one node and redoing that part at another node if the first node fails. In the single-level transaction system, if any part fails, the whole transaction fails [32].

For the distributed environment, we have examined the concept of nested transactions in the context of real-time database. In doing so, we have merged two independent approaches together - nested transactions and priority inheritance. These two approaches are very important to ensure that transactions meet the stringent tempo-

ral requirements while maintaining data consistency in a distributed environment. As a result, a new concept, namely *real-time nested transactions*, has been introduced. A set of protocols for locking, priority inheritance and state restoration is also defined together with a formal proof of the facts that the proposed protocols are free from deadlocks and have tightly bounded waiting period for higher priority transactions. Since nested transactions are particularly suitable for distributed environment, our proposed protocols can be efficiently implemented in distributed real-time transaction processing. Due to the concurrent execution of subtransactions, the overall performance of the protocol will be enhanced significantly. Furthermore, abortion of a subtransaction will not result in cascading aborts, but will allow restoration to a previously correct state.

Previous studies in this area [40, 41] have not addressed the issues related to aborting transactions. We have included the state restoration mechanism in our algorithm. Furthermore, we have introduced a new concept, namely *priority propagation*, which is particularly effective when the transactions abort in a nested environment. The information regarding the aborted nested transactions need to be propagated for priority readjustment of all other nested transactions that may be directly or indirectly affected. We have proposed implementation of this readjustment by requiring each transaction to maintain a dynamic status vector because the aborted transactions may require reevaluation of the priorities of the other transactions. This vector contains enough information to determine the priority that a transaction may assume when another transaction aborts. A formal update protocol has been defined for the dynamic status vector. Since, we do not anticipate frequent abortion of transactions, the overhead of re-evaluating priorities using this vector will be minimal. The ul-

timate goal of priority propagation and priority inheritance is to solve the priority inversion problem. Using our protocol, we have shown that a high priority transaction can be delayed by at most a single embedded nested transaction of one lower priority task. This bound on the waiting time provides a solution to the problem of unbounded waiting inherent in simple priority inheritance protocols.

Directions for Future Research

In databases, there are two aspects to the scheduling of transactions: concurrency control for serializing the execution of transactions and CPU and I/O scheduling for the execution of read and write operations. We have focused our research on the transaction scheduling aspects for concurrency control in real-time databases. It may be interesting to study the other aspect, that is, the physical resource scheduling in such an environment. This may lead to the development of efficient load balancing algorithms that may further enhance the performance. In such an environment, the transactions may arrive at any node and access data from a central database. Depending upon the load at the local node, an arriving transaction may be scheduled at a remote node. A *guarantee* protocol in the local scheduler can be used to determine whether the transaction can be scheduled locally. If not, a global scheduler will be invoked to find the most feasible node where the transaction should be transferred. This selection can be made by a *bidding* or *focused addressing* scheme similar to that used by [46, 54]. Furthermore, we have seen tremendous improvement with data partitioning and also anticipate that data replication may also lead to efficient algorithms.

The simulation study does not show any overhead that may result by using the

proposed protocols. However, from a theoretical perspective, it may be advisable to do a complete complexity analysis of the proposed protocols which is beyond the scope of this thesis. Similarly, the effectiveness of our protocols for real-time nested transactions can be quantitatively measured by simulation study similar to that described in earlier chapters. Further, to avoid priority inversion, protocols that use techniques such as priority inheritance should be tested with our RTP model. In this scheme, the priority of the executing transaction is raised to the priority of the waiting transaction so that it finishes sooner and allow the waiting transaction to resume execution.

We use exceution time estimates in priority assignment and concurrency control protocols and assume that estimates are exact. Inaccurate runtime estimates in scheduling decisions can lead to performance degradation. In order to determine the robustness to inaccuracies in the estimate, it may be interesting to study the sensitivity of the proposed protocols to such errors.

Finally, among other issues related to distributed real-time transaction processing [10, 13, 42], this research can be extended to incorporate areas such as precedence constraints, placement constraints for fault-tolerance, modeling time constraints (assigning *value functions*), and studying finer granularity of database (instead of *page*).

REFERENCES

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-time Transactions," *SIGMOD Record, ACM,* vol. 17, no. 1, Mar. 1988, 71-81.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-time Transactions: A Performance Evaluation," *Proc. 14th Intl. Conf. on Very Large Data Bases,* Los Angeles, CA, Aug. 1988, 1-12.

[3] R. Abbott and H. Garcia-Molina, "Scheduling Real-time Transactions with Disk Resident Data," *Proc. 15th Intl. Conf. on Very Large Data Bases,* Amsterdam, The Netherlands, Aug. 1989, 385-396.

[4] R. Agrawal, M. Carey and M. Livny, "Concurrency Control Performance Modelling: Alternatives and Implications," *ACM Trans. on Database Systems,* Dec. 1987.

[5] D. Agrawal, A. El Abbadi and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases," *ACM SIGMOD,* Jun. 1992, 104-113.

[6] P. Bernstein and N. Goodman, "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems," *Proc. 6th Intl. Conf. on Very Large Data Bases,* Oct. 1980.

[7] P. Bernstein and N. Goodman, "Multiversion Concurrency Control - Theory and Algorithms," *ACM Trans. on Database Systems,* vol. 8, no. 4, Dec. 1983, 465-483.

[8] P. A. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems," *Addison-Wesley, Reading Massachusetts,* 1987.

[9] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *IEEE Conf. on Data Engineering,* Feb 1989, 470-480.

[10] M. J. Carey and M. Livny, "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution and Replication," *Proc. 14th Intl. Conf. on Very Large Data Bases,* Los Angeles, CA, Aug. 1988, 13-25.

[11] M. J. Carey, R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling," *Proc. 15th Intl. Conf. on Very Large Data Bases,* Aug. 1989, 397-410.

[12] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.*, vol. 14, no. 2, Feb. 1988, 141-154.

[13] S. C. Cheng, J. A. Stankovic, and K. Ramamritham, "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems," *Proc. of 7th IEEE Real-Time Systems Symposium,* New Orleans, LA, 1986, 166-174.

[14] S. C. Cheng, J. A. Stankovic, and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey," *Hard Real-Time Systems - Tutorial, IEEE Computer Society Press,* 1988, 150-173.

[15] H. Chetto and M. Chetto, "Some Results of the Earliest Deadline Scheduling Algorithm," *IEEE Trans. on Software Engineering,* vol. 15, no. 4, Oct 1989, 1261-1269.

[16] R. P. Cook, S. H. Son, H. Y. Oh, and J. Lee, "New Paradigms for Real-Time Database Systems," *Eighth IEEE Workshop on Real-Time Operating Systems and Software,* Atlanta, GA, May 1991, 103-108.

[17] K. P. Eswaran, 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,* vol. 19, no. 11, Nov 1976, 624-633.

[18] A. A. Farrag and M. T. Ozsu, "Towards a General Concurrency Control Algorithm for Database Systems," *IEEE Trans. on Software Engineering,* vol. SE-13, no. 10, Oct 1987, 1073-1079.

[19] R. L. Graham *et al.*, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics,* 5, 1979, 287-326.

[20] W. Haque and J. Wong, "Performance Enhancement in Real-Time Transaction Processing," vol. 11, no. 22, *The International Journal of Microcomputer Applications,* 1992, 62-74.

[21] J. R. Haritsa, M. J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proc. of 9th ACM Symposium on Principles of Database Systems,* Nashville, TN, 1990, 331-343.

[22] J. R. Haritsa, M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proc. of 11th Real-Time Systems Symposium,* Lake Buena Vista, FL, 1990, 94-103.

[23] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proc. of 10th Real-Time Systems Symposium,* Santa Monica, CA, 1989, 144-153.

[24] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proc. of 12th Real-Time Systems Symposium,* San Antonio, TX, 1991, 210-221.

[25] W. Kim and J. Srivastava, "Enhancing Real-Time DBMS Performance with Multiversion Data and Priority Based Disk Scheduling," *Proc. of 12th Real-Time Systems Symposium,* San Antonio, TX, 1991, 222-231.

[26] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. on Database Systems,* vol. 6, no. 2, June 1981, 213-226.

[27] Y. Lin and S. H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proc. of 11th Real-Time Systems Symposium,* Dec. 1990, 104-112.

[28] C. L. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM,* vol. 20, no. 1, Jan. 1973, 174-189.

[29] M. Maekawa, A. E. Oldehoeft and R.R. Oldehoeft, "Operating Systems - Advanced Concepts," Benjamin/Cummings Publishing Company, 1987.

[30] A. K. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. Seventh Texas Conference on Computing Systems,* Nov. 1978.

[31] J. E. B. Moss, "Nested Transactions and Reliable Distributed Computing," *Proc. of 2nd Symposium on Reliability in Distributed Software and Database Systems,* July 1982, 33-39.

[32] J. E. B. Moss, "Nested Transactions - An Approach to Reliable Distributed Computing," The MIT Press, Cambridge, Massachusetts, 1985.

[33] C. H. Papadimitriou and P. C. Kanellakis, "On Concurrency Control by Multiple Versions," *ACM Trans. on Database Systems,* vol. 9, no. 1, Mar. 1984, 89-99.

[34] J. L. Peterson and A. Silberschatz, "Operating System Concepts," Addison-Wesley Publishing Company, 1985.

[35] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software,* vol. 1, no. 3, July 1984, 65-74.

[36] K. Ramamritham, J. A. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *IEEE Trans. Computers,* vol. 38, no. 8, Aug. 1989, 1110-1123.

[37] K. Ramamritham, J. A. Stankovic, and P. F. Shiah, "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 2, Apr. 1990, 184-194.

[38] D. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Trans. on Computer Systems,* vol. 1, no. 1, Feb. 1983, 3-23.

[39] L. Sha, R. Rajkumar and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *SIGMOD Record, ACM,* vol. 17, no. 1, Mar. 1988, 82-98.

[40] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers,* vol. 39, no. 9, Sep. 1990, 1175-1185.

[41] L. Sha, R. Rajkumar, S. H. Son and C.H. Chang, "A Real-Time Locking Protocol," *IEEE Trans. Computers,* vol. 40, no. 7, Jul. 1991, 793-800.

[42] K. G. Shin and Y. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Trans. Computers,* vol. 38, no. 8, Aug. 1989, 1124-1142.

[43] M. Singhal, "Issues and Approaches to Design of Real-Time Database Systems," *SIGMOD Record, ACM,* vol. 17, no. 1, Mar. 1988, 19-33.

[44] S. H. Son and R. P. Cook, "Scheduling and Consistency in Real-Time Database Systems," *Sixth IEEE Workshop on Real-Time Operating Systems and Software,* Pittsburgh, PA, May 1989, 42-45.

[45] S. H. Son and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *Seventh IEEE Workshop on Real-Time Operating Systems and Software,* Charlotsville, VA, May 1990, 39-43.

[46] J. A. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Trans. Computers,* vol. C-34, no. 12, Dec. 1985, 1130-1143.

[47] J. A. Stankovic and W. Zhao, "On Real-Time Transactions," *SIGMOD Record,* ACM, vol. 17, no. 1, Mar. 1988, 4-18.

[48] J. A. Stankovic, "Real-Time Computing Systems: The Next Generation," *Hard Real-Time Systems - Tutorial, IEEE Computer Society Press,* 1988, 14-36.

[49] L. Svobodova, "Resilient Distributed Computing," *IEEE Trans. on Software Engineering,* vol. SE-10, no. 3, May 1984, 257-268.

[50] H. Tokuda, C. W. Mercer, Y. Ishikawa and T. E. Marchok, "Priority Inversions in Real-Time Communication," *Proc. of 10th IEEE Real-Time Systems Symposium,* Santa Monica, CA, Dec. 1989, 348-359.

[51] I. L. Traiger, et al, "Transactions and Consistency in Distributed Database Systems," *ACM Trans. on Database Systems,* vol. 7, no. 3, Sep. 1982.

[52] J. Xu and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations," *IEEE Trans. Software Eng.,* vol. 16, no. 3, Mar. 1990, 360-369.

[53] W. Zhao and K. Ramamritham, "Simple and Integrated Heuristic Algorithms for Scheduling Tasks with Time and Resource Constraints," *J. Systems and Software,* vol. 7, 1987, 195-205.

[54] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Trans. Software Eng.,* vol. SE-13, no. 5, May 1987, 564-577.