# Concurrency Control Algorithms for Real-Time Database Systems

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

**Juhnyoung Lee**

January 1994

## APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)

_____

Author's Name

This dissertation has been read and approved by the Examining Committee :

_____

Dissertation Advisor

_____

Committee Chairman

_____

_____

_____

Accepted for the School of Engineering and Applied Science :

_____

Dean, School of Engineering and
Applied Science

January 1994

*Dedicated with love and appreciation to my parents*

# Acknowledgments

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

It has been a great privilege for me to work with Professor Sang Son, an exceptional researcher and teacher, who introduced me to real-time systems as well as database systems and distributed computing. He has been extraordinarily patient and supportive, having been always available for discussion and responding speedily to research reports. I would like to take this opportunity to thank him for his continued encouragement and guidance throughout the course of my research.

My sincere thanks go to Professor Jorg Liebeherr for working together and for his constructive comments and suggestions. My eternal gratitude is also due Professors Bill Wulf, Stephen Strickland, and Jim French for being on my advisory committee and for their useful comments on this work.

I would like to acknowledge my colleagues Ying-feng Oh, Prasad Wagle, David Rasikan, Nipun Agrawal, Matthew Lehr, Young-Kuk Kim, and Shi-Chin Chiang for helpful discussions during this research.

I am grateful to my parents and family, especially to my father who built in me the resolve to fight it out, and to my mother who was always there when I needed her. Their relentless encouragement was perhaps the single most influential factor in my education since childhood. And thanks to my brothers and sister for everything they had to offer.

Finally, I am grateful for the invaluable love, understanding and support shown by my wife Hye during the second half of my grad life.

# Abstract

In addition to satisfying data consistency requirements as in conventional database systems, concurrency control in real-time database systems must also satisfy timing constraints, such as deadlines associated with transactions. Concurrency control for a real-time database system can be studied from several different perspectives. This largely depends on how the system is specified in terms of data consistency requirements and timing constraints. The objective of this research is to investigate and propose concurrency control algorithms for real-time database systems, that not only satisfy consistency requirements but also meet transaction timing constraints as much as possible, minimizing the percentage and average lateness of deadline-missing transactions.

To fulfill the goals of this study, we conduct our research in three phases. First, we develop a model for a real-time database system and study the performance of various concurrency control protocol classes under a variety of operating conditions. Through this study, we understand the characteristics of each protocol and their impact on the performance, and ensure the validity of our real-time database system model by reconfirming the results from previous performance studies on concurrency control for real-time database systems. Second, we choose optimistic technique as the basic mechanism for our study on concurrency control for real-time database systems, and investigate its behavior in a firm-deadline environment where tardy transactions are discarded. We present a new optimistic concurrency control algorithm that outperforms previous ones over a wide range of operating conditions, and thus provides a promising candidate for the basic concurrency control mechanism for real-time database systems. Finally, we address the problem of incorporating deadline information into optimistic protocols to improve their real-time performance. We present a new priority-cognizant conflict resolution scheme that is shown to provide

considerable performance improvement over priority-insensitive algorithms, and to outperform the previously proposed priority-based conflict resolution schemes over a wide operating range. In each step of our research, we report the performance evaluation results by using a detailed simulation model of real-time database system developed in the first phase.

In addition to the three phases, we investigate semantic-based concurrency control techniques for real-time database systems, in which the semantics of operations on data objects are used to increase the concurrency of transactions executing on the data objects and to meet the timing constraints imposed on the transactions. We propose an object-oriented data model for real-time database systems. We present a semantic-based concurrency control mechanism which can be implemented through the use of the concurrency control protocols for real-time database systems studied earlier along with a general-purpose method for determining compatibilities of operations.

# Table of Contents

# List of Figures

# List of Tables

# 1. Introduction

## 1.1. Real-Time Database Systems

*Real-Time Database Systems* (RTDBSs) are becoming increasingly important in a wide range of operations. As computers have become faster and more powerful, and their use more widespread, real-time database systems have grown larger and become more critical. For example, they are used in program stock trading, telephone switching systems, virtual environment systems, network management, automated factory management, and command and control systems. More specifically, in the program stock market application, we need to monitor the state of the stock market and update the database with new information. If the database is to contain an accurate representation of the current market, then this monitoring and updating process must meet certain timing constraints. In this system, we also need to satisfy certain real-time constraints in reading and analyzing information in the database in order to respond to a user query or to initiate a trade in the stock market. For other examples given above, we can consider similar operations with timing constraints.

All of these real-time database applications are characterized by their time constrained access to data and access to data that has temporal validity. They involve gathering data from the environment, processing of gathered information in the context of information acquired in the past, and providing timely responses. They also involve processing not only archival data but also *temporal data* which loses its validity after a certain time interval. Both the temporal nature of the data and the response time requirements imposed by the environment make transactions possess timing constraints in the form of either *periods*

or *deadlines.* Therefore, the correctness of real-time database operation depends not only on the logical computations carried out but also on the time at which the results are delivered. The goal of real-time database systems is to meet timing constraints of transactions.

By contrast, conventional database systems deal with *persistent data* whose validity does not decay with time. The correctness of the output depends only on the logical computations, but not on the time at which the results are delivered. The goal of transaction and query processing approaches in conventional database systems is to achieve a good throughput or to minimize average response time.

One key point to note here is that real-time computing does not imply fast computing. Rather than being fast, more important properties of real-time (database) systems should be *timeliness*, i.e., the ability to produce expected results early or at the right time, and *predictability*, i.e., the ability to function as deterministically as necessary to satisfy system specifications including timing constraints [Stan90]. Fast computing which is busy doing the wrong activity without taking individual timing constraint into account is not helpful for real-time computing. While the objective of real-time computing is to meet the individual timing constraint of each activity, the objective of fast computing is to minimize the average response time of a given set of activities. Fast computing is helpful in meeting stringent timing constraints, but fast computing alone does not guarantee timeliness and predictability. In order to guarantee timeliness and predictability, we need to handle explicit timing constraints, and to use time-cognizant techniques to meet deadlines or periodicity associated with activities.

Another important point to note is that the characteristics of real-time database systems are unique and their problems cannot be solved by simple modifications or integrations of the solutions proposed for the problems in traditional database systems and real-time systems. Recently, there have been efforts to apply the benefits of (traditional) database technology to solve problems in managing the data in real-time systems. There also

have been attempts to utilize the techniques of time-driven scheduling and resource allocation algorithms in real-time database systems. From these efforts, however, it has been shown that a simple integration of concepts, mechanisms, and tools from database systems with those from real-time systems is not feasible. The reason for this situation is that the two environments, database systems and real-time systems, have different assumptions and objectives, which result in an impedance mismatch between them (for a detailed discussion on this issue, see [Buch89] and [Rama92]). Therefore, meeting the timing constraints of real-time database systems demands new approaches to data and transaction management.

Typically, a real-time system is an *embedded computer system*, which is used in a large system to provide control and computation functions. An embedded computer system has to manage and control the rest of the system, which is typically called the *environment*. For example, in an automated factory, the environment is the factory floor with its robots, assembling stations, and the assembled parts, while the computer system and human interfaces manage and coordinate the activities on the factory floor. Figure 1.1 illustrates a typical system of this class. The computer system interacts with its environment based on the

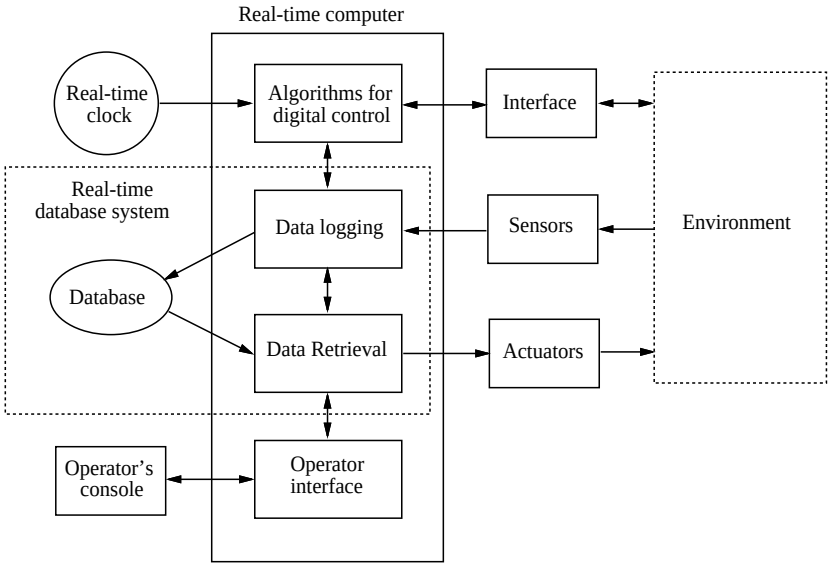



Figure 1.1 A Typical Embedded Computer System

data available about the environment, say from various sensors. It is imperative that the state of the environment, as perceived by the controlling system, be consistent with the actual state of environment. Otherwise, the effect of the computer system's activities may be disastrous. Hence, timely monitoring of the environment and timely processing of the sensed information are crucial. Timing constraints arise from the need to continuously track the environment. The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of *temporal consistency* [Liu91]. In addition, timing correctness requirements in a real-time (database) system also arise from the requirement imposed on the computer system reaction time. In a real-time database system, temporal consistency as well as logical consistency should be maintained. Thus, what is required is to satisfy logical consistency by concurrency control methods, and deal with temporal consistency requirements using time-cognizant transaction processing. Thus, we need to tailor the traditional concurrency control and other transaction management techniques to explicitly take timing constraints into account.

Based on their nature, transactions in real-time database systems are characterized along two dimensions: the source of timing constraints, and the implication of executing a transaction by its deadline, or more precisely, the implication of missing specified timing constraints. The characteristics of these temporal constraints, typically, are application-dependent. With regard to the source of timing constraints, as explained earlier, some timing constraints come from temporal consistency requirements, and some from requirements imposed on system reaction time. The former typically takes the form of periodicity requirements, as in an example: *every 20 seconds update robot position*. System reaction requirements typically take the form of deadline constraints imposed on aperiodic transactions, as in an example: *if temperature > 1000, within 10 seconds, add coolant to reactor.*

With respect to the effect of missing a transaction's deadline, we divide real-time

transactions into three groups: *hard-deadline*, *soft-deadline*, and *firm-deadline* transactions. This categorization can be shown clearly by using the *value function model* for real-time scheduling [Jens85, Lock86, Stra92]. The key idea of the value function model is that the completion of a transaction has a *value*, i.e., importance to the application that can be expressed as a function of the completion time. Whereas in reality arbitrary types of value functions can be associated with activities, the above categorization confines ourselves to simple value functions as illustrated in Figure 1.2.

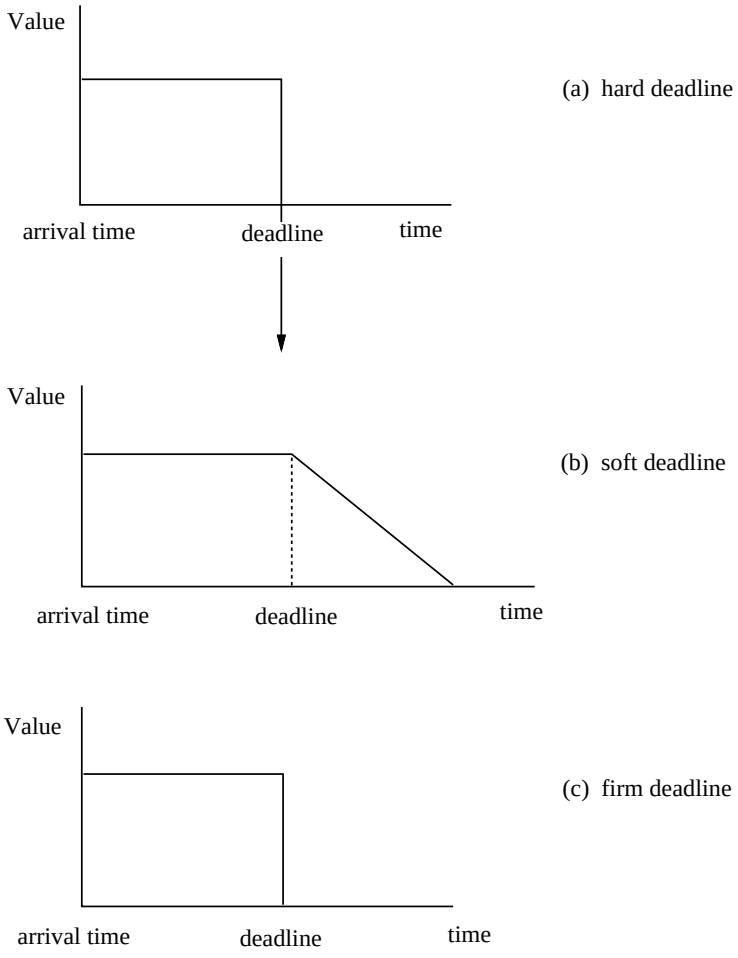



Figure 1.2 Value Function Model

A hard-deadline transaction may result in a catastrophe if its deadline is missed. That is, if a hard deadline is missed, a large negative value is imparted to the system. A soft-deadline transaction has some (positive) value even after its deadlines. Typically, the value drops to zero at a certain point past the deadline. Finally, firm-deadline transaction is a special case of soft-deadline transaction. In a firm-deadline transaction, the value of the transaction drops to zero at its deadline.

The different characteristics of transactions lead to different performance objectives and processing strategies for the transactions. The major difference between soft deadline (including firm deadline) and hard deadline is that all transactions with hard deadlines must meet their timing constraints. It is impossible to provide such a guarantee in a dynamic transaction management environment. In order to support the guarantee, the data and resource requirements as well as the computation requirement of such transactions have to be available before the actual execution starts, i.e., a static planning is required for transaction execution. For a variety of reasons, however, typical database systems do not provide such information beforehand, and thus appear infeasible in supporting transactions with hard deadlines [Stan88b, Rama92]. First, a database system for supporting hard-deadline transactions has to know when the transaction is likely to be invoked. Whereas this information is readily available for periodic transactions, for aperiodic transactions, by definition, it is not. Second, the system has to determine the resource and data requirements and worst-case execution time of the transaction. This information is not usually given and the worst-case execution time often may not be useful due to the large variance between the average-case and worst-case execution time caused by the dynamic paging mechanism and disk I/O in typical database systems.

If one is willing to deal with hard-deadline transactions using approaches analogous to those used in real-time task scheduling, many restrictions have to be placed on these transactions so that their characteristics are known beforehand. With these restrictions,

however, the functionality and applicability of transactions will be significantly limited, and poor resource utilization may result given the worst-case assumptions made about the transactions. For all these reasons, the problem of dealing with hard real-time transactions has been intractable under the state-of-the-art database technology, although there have been some effort on this area [Aud93, O'Nei92, Sha88, Sha91].

With soft-deadline transactions, we have more flexibility to process transactions, since the deadlines of such transactions are not required to meet all the time. Of course, the primary goal of the system should be to maximize the number of transactions that meet their deadline (or equivalently, to minimize the number of the transactions that miss their deadline). When transactions have different values, the value of transactions that complete by their deadlines must also be maximized. In this research, we restrict our interest to real-time database systems for processing transactions with soft or firm deadline, as most of the work done in this area (see the next section on related work). Another justification for this research direction is that a majority of real-time applications in real world is involved in soft or firm deadlines and the variety of applications imposing hard real-time is rather limited [Schw93].

Finally, we note that there is a certain tendency to reserve the term "real-time" only to refer to hard deadlines, because traditionally the problem of real-time system has implied that of scheduling activities with hard deadlines. Some prefer the term "time-critical" to "real-time" for processing activities with soft and firm deadlines as in "processing of time-critical transactions". However, the term "Real-Time Database System (RTDBS)" is widely accepted in the research community of real-time systems as representing the system for time-constrained transaction and query management. Thus, we use the term RTDBS in this study to refer to the system for transactions with "soft" and "firm" deadlines.

### 1.2. Related Work

In recent years, a number of papers have been published on issues in real-time database systems [Abbo88, Abbo89, Abbo90, Agra92, Buch89, Care89, Cook91, Daya88, Davi91, Grah91, Hari90, Hari90b, Hari91b, Hou89, Huan89, Huan91b, Huan91c, Kort90, Kort90b, Kuo92, Lee93, Lee93b, Lee93c, Lee93d, Lin90, Liu88, Marz89, O'Nei92, Pu92, Rajk89, Sha88, Son90, Son92, Song90, Stan88, Vrbs88]. The subjects addressed in these papers include the modeling of real-time transactions and real-time database systems [Abbo88, Buch89, Care89, Daya88, Hari90, Huan89, Kort90, Liu88, Sha88, Stan88], scheduling of real-time transactions [Abbo88, Abbo89, Huan89, Stan88, Sha88], concurrency control and data conflict resolution [Abbo88, Abbo89, Agra92, Buch89, Hari90, Hari90b, Huan89, Huan91b, Huan91c, Lee93, Lee93b, Lee93c, Lee93d, Lin90, Sha88, Son90, Son90b, Song90, You93], processing of queries with real-time constraints [Hou89], buffer management [Care89, Huan90], and I/O scheduling [Abbo90, Care89, Chen91]. A number of papers on related issues have also appeared. These include work on a protocol for timed atomic commitment [Davi91], fast recovery protocols for real-time database systems [Kort90b, Vrbs88], priority assignment protocols for managing an overload situation [Hari91b], and the application of imprecise computation technique [Lin87, Liu87] in real-time database systems [Hou89, Pu92, Rama91b]. There are also a number of studies on the correctness notions for data consistency other than serializability that can provide a higher level of concurrency by sacrificing data consistency temporarily to some degree [Lin89, Kuo92, Pu92].

Considerable work has been devoted to transaction scheduling and concurrency control in real-time database systems [Abbo88, Abbo89, Agra92, Buch89, Hari90, Hari90b, Huan89, Huan91b, Huan91c, Lee93, Lee93b, Lee93c, Lee93d, Lin90, Sha88, Son90, Song90, You93]. Most of these studies use the serializability for maintaining data consistency, because there is no general-purpose consistency criterion available that is less stringent than serializability, and yet as obviously correct and implementable as serializability. Most of the proposed algorithms for concurrency control are based on the two basic

mechanisms: *two-phase locking* (2PL) [Eswa76] and *optimistic concurrency control* (OCC) [Kung81], and use priority information in resolving data conflicts. Although another extensively studied concurrency control algorithm in conventional database systems, *timestamp ordering* [Lamp78] could have been used as a basic concurrency control mechanism for RTDBSs, very little work has been done in that direction. In fact, it appears that timestamp ordering is not amenable to RTDBSs, because the timestamp order usually determined by transaction arrival time has generally no relationship to transaction priority order. Finally, with a few exceptions [Buch89, Sha88], most of the above studies use a transaction model based on the assumptions that transactions arrive sporadically with unpredictable arrival times, and that data and resource requirements of each transaction are unknown to the scheduler.

The problem of scheduling transactions in an RTDBS with the objective of minimizing the percentage of transactions missing its deadline was first addressed in [Abbo88, Abbo89]. The work proposed policies for priority assignment and eligibility test in RTDBSs, and evaluated the performance of various scheduling policies through simulation experiments. A group of concurrency control protocols for RTDBSs using two-phase locking as the underlying mechanism was also proposed and evaluated. However, it was not their intention to do a comparative study of different concurrency control mechanisms.

The work in [Sha88, Sha91] assumed a transaction model in which transaction priorities and resource requirements are known *a priori*, and presents algorithms for scheduling a fixed set of periodic transactions with hard deadlines. The rate-monotonic algorithm [Liu73] is used for determining transaction priority, and a priority ceiling protocol based on locking is used for concurrency control. The priority ceiling algorithm appears to be promising for the hard real-time environment since it prevents deadlock formation and strictly bounds transaction blocking duration caused by *priority inversion* problem. A priority inversion occurs when a higher priority transaction must wait for the execution of lower pri-

ority transactions. The price, however, is the requirement of pre-knowledge of data and resource requirements, and transaction priorities. This condition appears unapplicable to many database applications, since in many cases, transaction execution is data dependent. Furthermore, the concurrency level provided by this protocol degrades significantly, if transactions can access any data objects in the database.

The work in [Hari90, Hari90b] focused primarily on the behavior of concurrency control algorithms in a real-time database environment, evaluating the relative performance of locking and optimistic concurrency control through simulation. The problem of adding transaction timing information to optimistic concurrency control for conflict resolution was also addressed. The work in [Huan89, Huan91, Huan91b] also investigated the performance of various concurrency control protocols. Their work used a real-time database testbed instead of a simulation system for experiments, and focused on the impact of the overhead involved in implementing concurrency control algorithms.

The work in [Lin90] proposed a concurrency control protocol that combines locking and optimistic approach. The protocol adjusts temporary serialization order dynamically in favor of transactions with high priority, and thus avoids blocking and aborts resulting from a mismatch between serialization order and priority order of transactions.

### 1.3. Research Scope and Goals

Concurrency control for a RTDBS can be studied from several different perspectives. This largely depends on how the system is specified in terms of data consistency requirements and timing constraints. In this study, database consistency is defined by the correctness notion of serializability, and the timing constraints associated with transactions are soft and firm deadlines. When transactions are associated with soft deadlines, we assume that every transaction has the same value. The primary metric used to evaluate the performance of various concurrency control protocols is *Miss Percentage*, the percentage

of transactions that miss their deadline. For soft-deadline transactions, we use a secondary performance metric, *Average Tardy Time*, which captures the degree of lateness of transactions that miss their deadline. The objective of this research is to propose and investigate concurrency control algorithms for RTDBSs, that not only satisfy consistency requirements but also meet transaction timing constraints as much as possible, minimizing the percentage and average lateness of deadline-missing transactions.

The transaction model used in this study assumes that transactions arrive sporadically with unpredictable arrival times, and that data and resource requirements of each transaction is unknown to concurrency control protocol beforehand. The priorities of transactions are assigned by the *earliest deadline first* algorithm [Liu73], which uses only deadline information to decide transaction priority, but not any other information such as transaction execution time. Also, the values of transactions are not considered in the priority assignment process, because we assume that every transaction has the same value. In this study, we consider a centralized disk resident database system operating on shared-memory multiprocessor as the hardware architecture of RTDBSs.

This work is motivated by the following considerations and observations on the previous work on the area of concurrency control for RTDBSs:

- There have been considerable studies on the performance of various concurrency control algorithm classes for conventional DBMSs [Agar87, Care84, Fran85, Ryu87, Tay85, Yu91]. However, since RTDBSs are significantly different from conventional DBMSs with respect to the performance goals and processing considerations, it is possible that the results obtained for the previous research on conventional concurrency control will not hold true in a RTDBS environment. This possibility motivated our investigation of the performance of various concurrency control algorithms in RTDBSs.

- Some results from the previous performance studies on concurrency control protocols for RTDBSs, instead of being definitive, are contradictory (e.g., between [Hari90] and [Huan91], and between [Abbo89] and [Huan91c]). Those results are based on and consequently limited by each study's own set of assumptions regarding RTDBS model. Explanations made until now for the possible reasons for the contradictory results are informative, but do not appear comprehensive. One of the objectives of this study is to examine the reasons for the contradictory performance results in a comprehensive way, addressing various RTDBS modeling assumptions and their implications.

- Some of the concurrency control protocols proposed for RTDBSs improve the real-time performance of the system considerably under certain conditions. However, we have observed that there are weaknesses and drawbacks in those protocols. For example, first, some of the locking-based protocols suffer from wasted restarts. Second, the performance of optimistic protocol using forward validation technique is degraded due to restarts unnecessary to ensure data consistency. Finally, the performance of optimistic protocol using a priority-wait type conflict resolution scheme, which was reported to provide significant gains over other priority-cognizant conflict resolution schemes under low and moderate levels of system contention, is worse than that of priority-insensitive protocol under a high level of contention. We are motivated to investigate these phenomena, and design a concurrency control protocol that improves the real-time performance over a wide range of operating conditions.

To fulfill the goals of this study, we conduct our research in three phases. First, we develop a model for a RTDBS and study the performance of various concurrency control protocol classes under a variety of operating conditions. Through this study, we understand the characteristics of each protocol and their impact on the performance, and ensure the

validity of our RTDBS model reconfirming the results from previous performance studies on concurrency control for RTDBSs. Second, we choose optimistic technique as the basic mechanism for our study on concurrency control for RTDBSs, and investigate its behavior in a firm-deadline environment where tardy transactions are discarded. We present a new optimistic algorithm that outperforms previous ones over a wide range of operating conditions, and thus provides a promising candidate for the basic concurrency control mechanism for RTDBS. Finally, we address the problem of adding deadline information to optimistic protocols to improve their real-time performance. We present a new priority-cognizant conflict resolution scheme that is shown to provide considerable performance improvement over a priority-insensitive algorithm, and to outperform the previously proposed priority-based conflict resolution schemes over a wide operating range.

In each phase, we report the performance evaluation results using a detailed simulation model of RTDBS developed in the first phase. The simulation program is written in SIMAN, a discrete simulation language [Pegd90]. The results of each phase is discussed in detail in the succeeding chapters.

In addition to the three phases, we investigate semantic-based concurrency control techniques for RTDBSs, in which the semantics of operations on data objects are used to increase the concurrency of transactions executing on the data objects and meet the timing constraints imposed on the transactions. For this study, we propose an object-oriented data model for a real-time database system. We present a semantic-based concurrency control mechanism which can be implemented through the use of the concurrency control protocols for RTDBSs studied earlier along with a general-purpose method for determining compatibility among operations.

### 1.4. Contributions of the Dissertation

The following is a summary of the major contributions of this dissertation:

- **RTDBS modeling**

  We have developed a fairly complete model for an RTDBS which captures both soft and firm deadlines imposed on transactions. The model provides a reasonable basis for the performance study of concurrency control algorithms in RTDBSs, under a wide range of operating conditions. We have also developed an RTDBS simulation system on the basis of the RTDBS model.

- **Performance evaluation of concurrency control algorithms for RTDBSs**

  We have shown that seemingly contradictory results from previous performance studies on concurrency control for RTDBSs [Hari90, Hari92, Huan91], some of which favored locking-based protocol and others of which favored optimistic approach, are not contradictory at all. We have examined the alternative assumptions in RTDBS modeling, and shown that the studies are all correct within the limits of their assumptions, particularly their assumptions about resource availability and policy for dealing with tardy transactions.

- **Optimistic concurrency control algorithms for RTDBSs**

  We have proposed a new optimistic concurrency control algorithm and a new deadline-sensitive conflict resolution scheme. The optimistic algorithm precisely adjusts temporary serialization order among concurrently running transactions, and thereby avoids unnecessary restarts, which could occur in conventional optimistic concurrency control algorithms. The conflict resolution scheme gives precedence to urgent transactions, while reducing the performance degradation caused by wasted sacrifices. Wasted sacrifices occur in other priority- or deadline-sensitive conflict resolution schemes that depend on incomplete information about transaction timing constraints. We have shown through simulation experiments that this protocol provides significant performance gains over other concurrency control protocols for

RTDBSs under a wide range of workload and operating conditions.

- **Semantic-based concurrency control for RTDBSs**

  We have proposed a semantic-based concurrency control technique for RTDBS which can increase the degree of concurrency allowed for transaction execution. For the technique, we have employ an object-oriented data model. Unlike the previous work in this area [Wolf93c], our approach deals with logical and temporal consistency constraints of RTDBS in separate steps. This approach provides a framework in which implementing a semantic-based real-time concurrency control mechanism can be done relatively easily through the use of the results from previous research on both traditional semantic-based concurrency control and concurrency control for RTDBSs.

### 1.5. Organization of the Dissertation

The remainder of this thesis is organized as follows. In Chapter 2, we investigate the key components of a reasonable model of real-time database systems, including the policy for dealing with tardy transactions, the availability of resources in the system, and the use of pre-knowledge about transaction processing requirement. We employ a fairly complete model of an RTDBS for studying the relative performance of locking and optimistic concurrency control protocols under a variety of operating conditions. In addition, we examine the issues on the implementation of concurrency control algorithms, which may have a significant impact on performance. We show that under a soft deadline system, the results of the relative performance of locking and optimistic approaches depend heavily on resource availability in the system as in conventional database systems. In the context of firm deadline systems, it is shown that an optimistic protocol outperforms a locking-based protocol under a wide range of resource availability and system workload level. Based on these results, we reconfirm the results from previous performance studies on concurrency

control for RTDBSs.

In Chapter 3, we investigate the performance of optimistic protocols in a firm-deadline RTDBS where tardy transactions are discarded. We show that the optimistic algorithms used in the previous studies [Hari90, Hari90b, Huan91] incur restarts unnecessary to ensure data consistency. We present a new optimistic algorithm that can avoid such unnecessary restarts by adjusting serialization order dynamically, and demonstrate that the new algorithm outperforms the previous ones over a wide range of system workload.

Chapter 4 addresses the problem of incorporating transaction deadline information into optimistic concurrency control protocols to improve their real-time performance. We discuss why this problem is non-trivial, and review some alternative solutions to this problem. We present a new deadline-sensitive scheme called *Feasible Sacrifice*, for making data conflict resolution decisions. This scheme gives precedence to urgent transactions, while reducing the number of missed deadlines due to wasted sacrifices with a feasibility test of every validating transaction. For the feasibility test, we also present an approach to estimating the execution time of restarted transactions in optimistic protocols. Feasible Sacrifice scheme is shown to provide significant gains over deadline-insensitive optimistic algorithms, and to perform constantly better than the conflict resolution scheme using a priority wait mechanism and a wait control technique studied in [Hari90b].

In Chapter 5, we present a semantic-based concurrency control technique for a real-time database systems defined on the basis of object-oriented paradigm. First, we examine the essential features of an object-oriented data model for an RTDBS. We show that the characteristics of the model provide opportunities for semantic-based concurrency control. Our approach alleviates the complexity associated with a semantic-based concurrency control mechanism for RTDBSs. This mechanism is based on a technique determining operation compatibilities by using the concept of affected-set of operations, and employs concurrency control algorithms augmented with deadline-sensitive conflict resolution

schemes. The mechanism is application independent, and hence mitigate the burden on the designers of object types to determine compatibility relation of operations.

Finally, Chapter 6 summarizes the salient results of our research, and discusses the research contributions. We also suggest future directions in which our research can be extended.

# 2. Performance of Concurrency Control Algorithms

## 2.1. Introduction

Due to its unique characteristic, the performance goals of a RTDBS is different from those for a conventional database system, and a RTDBS requires new transaction management methods to satisfy the performance goals. In RTDBSs, the primary performance criterion is timeliness level and not average response time or throughput, and scheduling of transactions is driven by priority considerations rather than fairness considerations [Rama92]. Given these differences, considerable research has recently been devoted to designing concurrency control algorithms for RTDBSs and to evaluating their performance.

Some of the work done in this area focused on the relative performance of various concurrency control protocol classes. For example, the work in [Hari90, Hari90b] first evaluated the relative performance of locking and optimistic concurrency control in a real-time database environment through simulation. This work also first addressed the problem of incorporating transaction timing information into optimistic algorithms. The work in [Huan91, Huan91b] investigated the performance of various real-time concurrency control protocols. This work used a real-time database testbed instead of a simulation system for experiments, and focused on the impact of the overhead involved in implementing concurrency control algorithms.

These performance studies on real-time concurrency control protocols are

informative, but some of the results emerged, instead of being definitive, are contradictory. For example, the studies in [Hari90] suggest that optimistic approach outperforms locking in a real-time database environment, while the experimental results in [Huan91] indicate that optimistic protocols may not always outperform priority-cognizant two-phase locking schemes when the physical implementation of concurrency control algorithms is considered. Also, it was reported in [Hari90b] that the use of priority-driven wait mechanism in conflict resolutions further improves the performance of optimistic approach, but the results in [Huan91] showed that the priority-driven wait strategy has no significant impact on performance.

Regarding these apparent contradictions between the results from [Hari90, Hari90b] and [Huan91], possible reasons were examined. In [Huan91], it was pointed out that the contradictions come from the fact that unlike theirs, the results from [Hari90, Hari90b] are based on simulation experiments, where optimistic concurrency control is carried out at the logical level and detailed implementation issues at the physical level are ignored[1]. They argued that in practice, the implementation schemes and the corresponding overheads may affect the protocol performance, so that they should be carefully analyzed and included in performance study. On the other hand, in [Hari92], a performance crossover point between two-phase locking and optimistic algorithms was shown when their database system was operated over a wide range of workload. They argued that the contradictions are primarily due to basic differences in system workload rather than differences in experimental systems or protocol implementation schemes.

These arguments appear correct and complementary to each other, but do not seem comprehensive separately. Those results are based on and consequently limited by each study's own set of assumptions regarding database system resource availability,

1. In particular, it was pointed in [Huan91] that the optimistic protocol used in [Hari90, Hari90b] employs a broadcast mechanism in its validation phase but no overhead involved in this mechanism is taken into account in the performance study.

characteristics of timing constraints, the range of system workload, and the implementation schemes for concurrency control algorithms. In this study, rather than presenting "yet another performance study of real-time concurrency control algorithms," we examine the reasons for the contradictory performance results in a comprehensive way, addressing the RTDBS modeling assumptions used in the previous studies and their implications.

The remainder of this chapter is organized in the following fashion: Section 2.2 describes our choice of concurrency control algorithms for this comparative performance study, one locking-based algorithm and one optimistic algorithm. In Section 2.3, we discuss the alternatives of RTDBS model and their implications. Section 2.4 describes our simulation system of RTDBS for experiments. In Section 2.5, the results of the simulation experiments are highlighted. Finally, Section 2.6 summarizes the main conclusions of this study.

## 2.2. Concurrency Control Algorithms

For this study, we have chosen to examine two concurrency control algorithms, one locking-based algorithm called 2PL-HP, and one optimistic algorithm called OCC-FV. These algorithms were chosen because they represent concurrency control protocols for RTDBSs based on locking and optimistic concurrency control, respectively, in the previous studies.

### 2.2.1. A Locking Algorithm

In classical two-phase locking protocol [Eswa76], transactions set read locks on objects that they read, and these locks are later upgraded to write locks for the data objects that are updated. If a lock requested is denied, the requesting transaction is blocked until the lock is released. Read locks can be shared, while write locks are exclusive.

For real-time database systems, two-phase locking needs to be augmented with a priority-based conflict resolution scheme to ensure that higher priority transactions are not

delayed by lower priority transactions. In *High Priority* scheme [Abbo88], all data conflicts are resolved in favor of the transaction with the higher priority. When a transaction requests a lock on an object held by other transactions in a conflicting lock mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the lock. In addition, a new read lock requester can join a group of read lock holders only if its priority is higher than that of all waiting write lock operations. This protocol is referred to as 2PL-HP [Abbo88]. It is important to note that 2PL-HP loses some of the basic 2PL algorithm's blocking factor due to the partially restart-based nature of the High Priority scheme.

Note that High Priority scheme is similar to *Wound- Wait* scheme [Rose78], which is added to two-phase locking for deadlock prevention. The only difference is that High Priority scheme uses priority order decided by transaction timing constraints for conflict resolution decisions, while Wound-Wait employs timestamp order usually decided by transaction arrival time. It is obvious that High Priority serves as a deadlock prevention mechanism, if the priority assignment mechanism assigns unique priority value to a transaction and does not dynamically change the relative priority ordering of concurrent transactions. Also, note that 2PL-HP is free from priority inversion problem, because a higher priority transaction never waits for a lower priority transaction, but restarts it.

### 2.2.2. An Optimistic Concurrency Control Algorithm

In optimistic concurrency control, transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. Thus, the execution of a transaction consists of three phases: read, validation, and write [Kung81]. The key component among these is the validation phase where a transaction's destiny is decided. Validation comes in several flavors, but it can carried out basically in either of two ways: *backward validation* and *forward validation* [Haer84]. While in backward scheme, the

validation process is done against committed transactions, in forward validation, validating of a transaction is carried out against currently running transactions.

As explained above, in RTDBSs, data conflicts should be resolved in favor of higher priority transactions. In backward validation, however, there is no way to take transaction priority into account in serialization process, since it is carried out against already committed transactions. Thus the backward scheme is not amenable to real-time database systems. Forward validation provides flexibility for conflict resolution such that either the validating transaction or the conflicting active transactions may be chosen to restart, so it is preferable for real-time database systems. In addition, forward scheme generally detects and resolves data conflicts earlier than backward validation, and hence it wastes less resources and time.

All the optimistic algorithms used in the previous studies of real-time concurrency control in [Hari90, Hari90b, Huan91] are based on the forward validation. The broadcast mechanism in the algorithm, OPT-BC used in [Hari90, Robi82] is an implementation variant of the forward validation. We hereafter refer to the optimistic algorithm using forward validation as OCC-FV.

A point to note is that unlike 2PL-HP, OCC-FV does not use any transaction priority information in resolving data conflicts. Thus, under OCC-FV, a transaction with a higher priority may need to restart due to a committing transaction with a lower priority. Several methods to incorporate priority information into OCC-FV were proposed and studied in [Hari90b, Huan91], using priority-driven wait or abort mechanism. However, more work is needed to ensure if these methods have any significant impact on improving OCC-FV performance, because the effect of increased waiting time or increased number of aborts in these methods may overshadow the performance gain due to the preferential treatment of transactions. We have chosen OCC-FV because it shows the representative performance among optimistic algorithms studied in the context of real-time database

systems.

### 2.2.3. Qualitative Analysis

From the traditional viewpoint, various concurrency control algorithms basically differ in two aspects: the time when they detect conflicts and the way that they resolve conflicts. Locking and optimistic approach in their basic form represent the two extremes in terms of these two aspects. Locking detects conflicts as soon as they occur and resolves them using blocking. Optimistic scheme detects conflicts only at transaction commit time and resolves them using restarts. In RTDBSs, the way to incorporate transaction deadline information into data conflict resolution makes further difference among concurrency control mechanisms. The impact of these differences in concurrency control algorithms on performance has been the major theme in the performance study of concurrency control in both conventional and real-time database systems.

With respect to the impact of conflict resolution methods, the effect of blocking and restart should be considered in the context of the amount of available system resources. Generally, blocking-based conflict resolution policy conserves resources, while restart-based policy wastes more resources. Previous performance studies on conventional database systems in [Agra87, Care84] showed that locking algorithm that resolves data conflicts by blocking transactions outperforms restart-oriented algorithm in an environment where physical resources are limited. Also, the work showed that if resource utilizations are low enough so that a large amount of wasted resources can be tolerated, and there are a large number of transactions available to execute, then a restart-oriented algorithm that allows a higher degree of concurrent execution is a better choice. In this study, we investigate the effect of blocking and restart in the context of RTDBSs.

The timing of conflict detection and resolution also has a major impact on performance. With the optimistic algorithm using backward validation, the delayed conflict

resolution results in wasting more resources than locking protocol, since a transaction can end up being restarted after having paid for most of its execution. With the forward validation, however, this problem is eliminated, since any transaction that reaches the validation phase are guaranteed to commit, and transactions involved in any non-serializable execution restart early in their read phase. Also the delayed conflict resolution of optimistic approach helps in making better decisions in conflict resolution, since more information about conflicting transactions is available at this later stage. On the other hand, the immediate conflict resolution policy of locking schemes may lead to useless restarts and blocking in RTDBSs due to its lack of information on conflicting transactions at the time of conflict resolution [Hari90].

### 2.2.4. Implementation Issues

In this section, we describe a physical implementation method for optimistic algorithms with forward validation. The implementation method and its corresponding overhead are important factors for a correct performance comparison of concurrency control algorithms. For the implementation of concurrency control mechanism, we had two major concerns. One was the efficiency of validation process. At the logical level, optimistic approach detects data conflicts by comparing the read set and write set of transactions. This method is impractical for actual database systems, since the complexity of the validation test is dependent on the number of active transactions. The other concern was the comparability of locking algorithms and optimistic schemes. In the previous studies of concurrency control for RTDBSs in [Hari90, Hari90b], the validation test of optimistic algorithms were implemented using a broadcast mechanism by which the validating transaction notifies other currently running transactions with data conflicts. The concept was straightforward, but it is difficult to compare the performance cost of locking protocol implemented using lock table management with optimistic algorithm implemented using the broadcast mechanism, due to the drastic difference in their

implementation methods. It is difficult to determine the correct cost for each implementation mechanism, especially for simulation study. This difficulty also applies to performance study using actual systems or a testbed, because the implementation costs often vary from one system to another.

Considering these issues, we decided to use a locking mechanism for the implementation of optimistic protocols. The mechanism is based on the one proposed in [Huan91]. In this mechanism, the system maintains a system-wide lock table, LT, for recording data accesses by all concurrently executing transactions. There are two lock modes - read-phase lock (*R-lock*) and validation-phase lock (*V-lock*). An R-lock is further classified by the type of the data operation, i.e., read and write operation. An R-lock is set in LT whenever a transaction accesses a data object in its read phase, and an R-lock for write operation is upgraded to a V-lock when the transaction enters its validation phase. The two lock modes are not compatible. Generally, the validation process is carried out by checking the lock compatibility with the lock table. This locking-based implementation of validation test is efficient because its complexity is independent of the number of active transactions. In addition, it determines fair implementation costs for the two algorithms due to their common ground on lock table management.

It was shown in [Huan91] that the physical operations on the lock table of this implementation of optimistic protocol are almost the same as those of locking protocols. Those operations include setting of R-locks, upgrading of R-locks to V-locks, and releasing of R-locks and V-locks. Despite this similarity, there are some differences that may affect the performance of locking and optimistic protocols. First, the locking duration of optimistic algorithm is shorter than that of locking. This is due to the fact that in optimistic protocol, the locking duration is only during the validation and write phase of a transaction, while in locking protocol, it may be as long as the entire lifetime of a transaction, since write lock can be issued anytime. Second, the R-locks of optimistic protocol will not block

any concurrent transactions in read phase, while under locking protocol read locks can block write lock requests on the same data object. The R-locks in optimistic algorithm are just place holders to allow the validating transaction to know that the data object was accessed by an concurrently running transaction. Finally, the optimistic protocol maintains the property of deadlock-freedom, even though two incompatible lock modes are used. It is guaranteed by the facts that the validation phase where V-lock requests are made occurs in critical section, and that once all the V-lock requests are granted, the transaction will not request any lock afterwards.

## 2.3. RTDBS Model

In this section, we consider alternatives of modeling assumptions, which have significant impacts on the performance of concurrency control protocols in RTDBSs. Combining the alternatives of those assumptions, we design our experiments for this performance study.

### 2.3.1. Soft Deadline versus Firm Deadline

In this research, we perform simulation experiments to study the implications of different characteristics of transaction deadlines on the performance of the two concurrency control algorithms. We investigate the performance of the algorithms under soft deadline and firm deadline assumption. The distinction between soft deadline and firm deadline lies in their view of tardy transactions, and hence they require different policies for dealing with tardy transactions. For a firm deadline transaction, completing a transaction after its deadline has expired is of no value and may even be harmful. Thus, tardy transactions are required to be aborted and permanently discarded from the system. A transaction with soft deadline, however, has some (diminished) value even after their deadlines have expired. Therefore, in a soft deadline system, all transactions are required to complete eventually.

These characteristics of deadline in real-time transactions come directly from real-

time application requirements, and are not an option for the system designer. An example of real-time database application having firm deadline is a financial market program for arbitrage trading described in [Abbo88]. The detection and exploitation of arbitrage opportunities are constrained to time. In such a system, if a transaction submitted to perform an arbitrage operation misses its deadline, it may be best not to perform the operation at all, since the conditions that triggered the decision to submit the transaction may have changed. Typical examples of soft deadline application that requires all transactions to complete eventually regardless of whether they are tardy or not include transaction processing systems such as banking systems, airline reservation systems, and telephone switching systems. For instance, in a telephone switching system, where an 800 number is translated to an actual telephone number, users would rather want the transaction completed late than not at all.

Different execution requirements of firm deadline and soft deadline transactions lead to different system objectives and hence performance metrics in performance study. In a real-time database system that deals with firm-deadline transactions and hence discards tardy transactions, its objective is simply to minimize the number of transactions missing its deadline. Hence a single metric, *Miss Percentage* is sufficient to characterize the system performance. Miss Percentage is the percentage of transactions that do not complete before their deadline. In a system dealing with soft deadline, in addition to Miss Percentage, we need a secondary metric, *Average Tardy Time*, to capture the degree of lateness of tardy transactions. A point to note here is that there is a trade-off between the two metrics, Miss Percentage and Average Tardy Time, in the sense that a decrease in one will usually result in an increase in the other. Thus, it is difficult to simultaneously improve both metrics [Hari90].

The behavioral difference between soft deadline system and firm deadline system has been shown analytically using simple queueing systems [Hari90]. The analysis was

very loose and abstract, showing the immaturity of this field. Still, it clearly showed the difference between the two systems. A soft deadline system is modeled by a single *M/M/1* queue. The computation of the Miss Percentage for this system can be done as follows. Consider a transaction with slack time, *s*. Then the probability that it misses its deadline is equal to the probability that it must wait in the server queue more than *s* time units. Given an *M/M/1* system with mean inter-arrival time $1/\lambda$ and mean service time 1.0, the waiting time distribution *W(s)* is given by [Klei75]:

$$W(s) = \Pr[\text{time spent in queue} \leq s]$$

$$= 1 - \lambda e^{-(1-\lambda)s}.$$

Thus the probability that the transaction waits more than *s* time units and misses its deadline is 1 - *W(s)*. To compute the expected probability of missed deadline, we need the distribution of *s*. However, for simplicity we replace *s* by some constant, *D*, and take this probability to be analogous to the Miss Percentage of soft deadline system denoted as $MP_{SD}$. We obtain the following equation:

$$MP_{SD} = \lambda e^{-(1-\lambda)D}.$$

A sample graph of $MP_{SD}$ versus $\lambda$ for $D = 10$ is shown in Figure 2.1(a).

A firm deadline system can be modeled by a *M/M/1/N* queue, which is identical to *M/M/1* system except that the size of the queue is finite, and hence the customers are turned away and not accepted by the queue if it is full. The probability that customers are turned away is called the probability of blocking. We take the probability to be roughly analogous to the Miss Percentage of the firm deadline system, and denote it by $MP_{FD}$. Given an *M/M/1/N* system with mean inter-arrival time $1/\lambda$ and mean service time 1.0, the blocking probability, i.e., $MP_{FD}$ is given by [Klei75]:

$$MP_{FD} = ((1 - \lambda) / (1 - \lambda^{N+1})) \lambda^{N}.$$

A sample graph of $MP_{FD}$ versus λ for $N$ = 10 is shown in Figure 2.1(b).

Note that the graphs of $MP_{SD}$ and $MP_{FD}$ in Figure 2.1 show significantly different characteristics. While $MP_{SD}$ increases exponentially as the arrival rate increases, $MP_{FD}$ shows a *S*-shape because in a firm deadline system, the population in the system is regulated automatically by discarding tardy transactions. The behavioral difference between soft deadline system and firm deadline system analyzed in this section will be confirmed in the experimental results using simulation in Section 2.5.

### 2.3.2. Resource Availability

In this study, we perform simulation experiments to examine the implications of different levels of resource availability on the performance of the two concurrency control algorithms described in Section 2.2. We investigate the performance of the two algorithms under the infinite resource assumption, and with limited resources. As explained earlier, the impact of the resource availability on the system performance is closely related with the conflict resolution method employed by the concurrency control mechanism. Blocking as a conflict resolution scheme conserves resources at the expense of waiting of transactions, while restart eliminates transaction waiting, sacrificing the amount of resource and time spent by the transaction restarted. While 2PL-HP employs both blocking and restart for conflict resolution, OCC-FV depends solely on restarts.

In conventional database systems, abundant resources are usually not to be expected and a reasonable model of system resources is crucial ingredient for performance studies [Agra87]. However, it is worth noting that in the application domain of RTDBSs, functionality, rather than cost, is often the driving consideration. Because many real-time systems are involved in mission-critical applications and hence sized to handle transient heavy loading, it may be common to provide sufficient resources in RTDBS environment.

Figure 2.1 (a) Miss Probability in Soft-deadline System

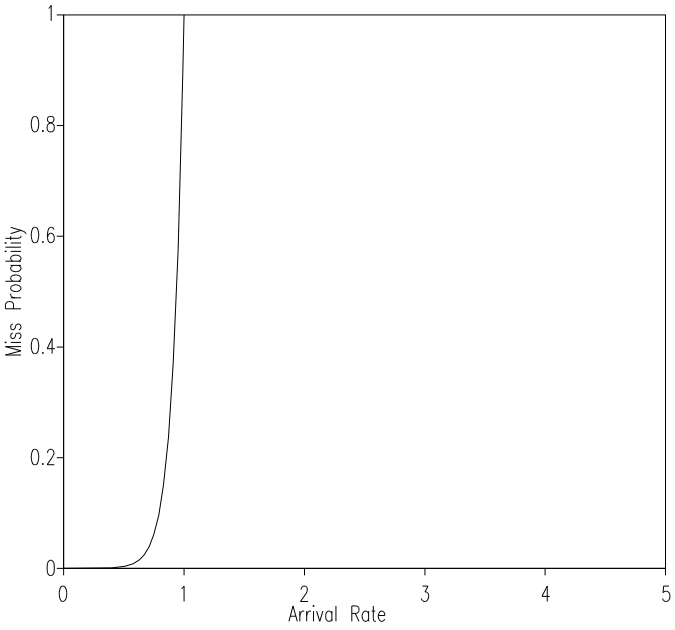
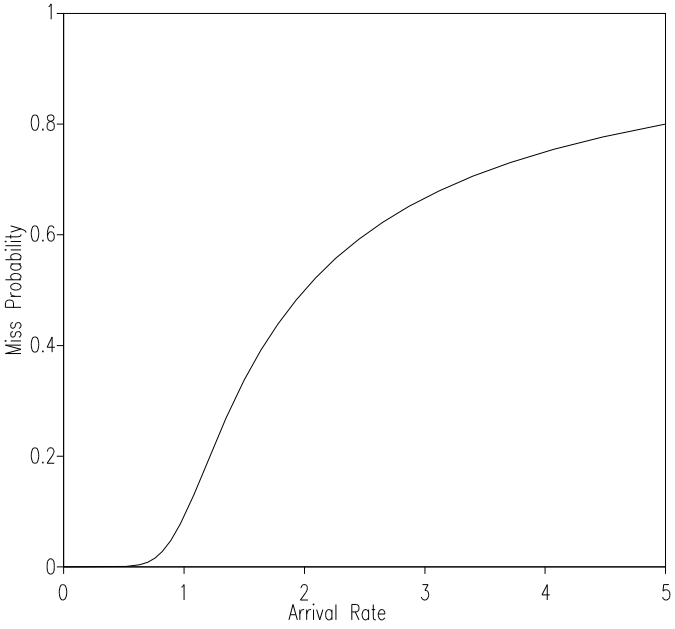



Figure 2.1 (b) Miss Probability in Firm-deadline System



## 2.4. Experimental Environment

This section outlines the structure and details of our simulation model and experimental environment which were used to evaluate the performance of concurrency control algorithms for RTDBSs.

### 2.4.1. Simulation Model

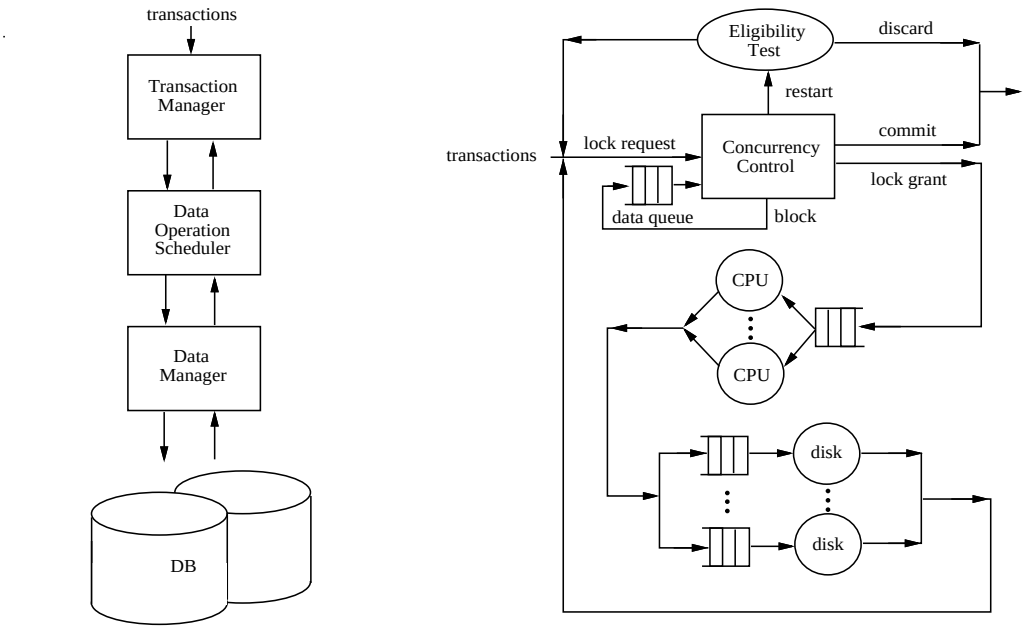
Central to our simulation model for RTDBS is a single-site disk resident database system operating on shared-memory multiprocessors. The database itself is modeled as a collection of data pages in disks, and the data pages are modeled as being uniformly distributed across all the disks.

The logical model is shown in Figure 2.2(a). This model was developed based on the conventional database system model described in [Bern87]. It consists of three main components: a *Transaction Manager* (TM), a *Data Operation Scheduler* (DOS), and a *Data Manager* (DM). TM is responsible for issuing lock requests, DOS schedules these requests according to the specifications of the concurrency control protocol, and DM is responsible for granting access to physical data objects. This abstract model of database system is useful for reasoning about the components and their interactions in a database system. However, this model is unsuitable for a basis for building an experimental system, since it does not correspond to the software and hardware architecture of actual database systems. There is a need to transform this logical model to a model physically justifiable and suitable for the development of experimental systems.

Underlying the logical model are two physical resources, CPU and disks. Associated with each logical component in the model are some use of one or both of the two resources. The amounts of CPU and I/O time per logical service are specified as model parameters. The physical queueing model is depicted in Figure 2.2(b), and the associated model parameters are described in the next section. The physical queueing model is similar

to the one used in [Agra87, Care84]. CPUs share a single queue and the service discipline used for the queue is priority scheduling without preemption. Each disk has its own queue and is also scheduled with priority scheduling.

In this model, the execution of a transaction consists of multiple instances of alternating data access request and data operation steps until all of the data operations in it complete or it is aborted for some reason. When a transaction makes a data access request, i.e., lock request on a data object, the request must go through concurrency control to obtain a lock on the data object. If the request is granted, the transaction proceeds to perform the data operation which consists of a disk access and CPU computation. The transaction passes through disk queue and CPU queue. If the request for the lock is denied, the transaction will be placed into a data queue. The waiting transaction will be awakened when the requested lock is released.




(a) Logical model

(b) Physical queueing model

Figure 2.2 The System Model

A transaction consists of a mixed sequence of read and write operations. A read operation goes through first, concurrency control to obtain access permission, then a disk I/O to read the page (if the page is not in memory), and finally, a CPU usage for processing the page. The steps of write operations are similar except that disk I/O for write operation is deferred until the transaction has committed. We assume that a write operation is always preceded by a read for the same page, that is, the write set of a transaction is always a subset of its read set [Bern87].

If a data access request leads to a decision to abort the transaction, it has to restart. The system checks the eligibility of a transaction whenever it restarts, and whenever it is put into and comes out of a queue, to see if it has already missed its deadline. With the firm deadline assumption, transactions that has missed deadline are aborted and permanently discarded from the system. On the other hand, with the soft deadline assumption, the priority of transactions that missed their deadline is promoted higher than that of any feasible transactions so that, although they complete late, they complete with minimum delay.

### 2.4.2. Parameter Setting

Table 1 gives the names and meanings of the parameters that control system resources. The parameters, *CPUTime* and *DiskTime* capture the CPU and disk processing times per data page. Our simulation system does not explicitly account for the time needed for transaction management, data operation scheduling, data management and context switching. We assume that these costs are included in *CPUTime* on a per data object basis.

The use of database buffer pool is simulated using probability, rather than each buffer page being traced individually. When a transaction attempts to read a data page, the system determines whether the page is in memory or disk using the probability, *BufProb*. If the page is determined to be in memory, the transaction can continue processing without

disk access. Otherwise, an I/O service request is created.

Table 2 summarizes the key parameters that characterize system workload and transactions. Transactions arrive in a Poisson stream, i.e., their inter-arrival times are exponentially distributed. The *ArriRate* parameter specifies the mean rate of transaction arrivals. The number of data objects accessed by a transaction is determined by a triangular distribution (as an approximation of a normal distribution) with mean *TranSize*, and the actual data objects are determined uniformly from the database. A page that is read is updated with the probability, *WriteProb*.

| Parameter | Meaning | Base value |
|---|---|---|
| *DBSize* | Number of data pages in database | 400 |
| *NumCPUs* | Number of processors | 2 |
| *NumDisks* | Number of disks | 4 |
| *CPUTime* | CPU time for processing a page | 15 msec |
| *DiskTime* | Disk service time for a page | 25 msec |
| *BufProb* | Probability of a page in memory buffer | 0.5 |

**Table 1: System Resource Parameters**

| Parameter | Meaning | Base Value |
|---|---|---|
| *ArriRate* | Mean transaction arrival rate | - |
| *TranSize* | Average transaction size (in pages) | 10 |
| *WriteProb* | Write probability per accessed page | 0.25 |
| *MinSlack* | Minimum slack factor | 2 |
| *MaxSlack* | Maximum slack factor | 8 |

**Table 2: Workload Parameters**

The assignment of deadlines to transactions is controlled by the parameters, *MinSlack* and *MaxSlack*, which set a lower and upper bound, respectively, on a transaction's slack time. We use the following formula for deadline-assignment to a transaction:

$$Deadline = AT + \text{uniform}(MinSlack, MaxSlack) * ET.$$

*AT* and *ET* denote the arrival time and execution time, respectively. The execution time of a transaction used in this formula is not an actual execution time, but a time estimated using the values of parameters, *TranSize*, *CPUTime* and *DiskTime*. This value is used only for the above deadline-assignment formula, but not used for any other purpose including conflict resolution decisions in concurrency control. In this system, the priorities of transactions are assigned by the *Earliest Deadline First* policy [Liu73], which uses only deadline information to decide transaction priority, but not any other information on transaction execution time.

Finally, the base values for parameters shown in Tables 1 and 2 are not meant to model a specific real-time application. They were chosen to be reasonable for a wide range of actual database systems.

### 2.4.3. Performance Metrics

The primary performance metric used is the percentage of transactions which miss their deadlines, referred to as *Miss Percentage*. Miss Percentage is calculated with the following equation:

$$Miss\ Percentage = 100 * (\text{number of tardy jobs} / \text{number of jobs arrived}).$$

As a secondary performance metric, *Average Tardy Time* for transactions missing their deadline is used for a soft deadline system, to capture the degree of lateness of tardy transactions.

In addition to these metrics, we measure other statistical information, including system throughput, average number of transaction restarts, average data blocking time, and average resource queueing time. These secondary measures help explain the behavior of the concurrency control algorithms under various operating conditions. The average number of transaction restarts, which we refer to as *Restart Count,* is normalized on a per-transaction basis, so that its value represents the average number of restarts experienced by a transaction until it completes, or misses its deadline and is discarded. The average data blocking time and average resource queueing time are normalized on a per-blocking basis.

### 2.5. Experiments and Results

The simulation program was written in SIMAN, a discrete-event simulation language [Pedg90]. The data collection in the experiments is based on the method of replication. For each experiment, we ran the simulation with the same parameter values for at least 10 different random number seeds. Each run continued until 1,000 transactions were executed. For each run, the statistics gathered during the first few seconds were discarded in order to let the system stabilize after initial transient condition. The statistical data reported in this study has 90% confidence intervals whose end points are within 10% of the point estimate. In the following graphs, we only plot the mean values of the performance metrics.

The experiments conducted for this study were designed to investigate the impact of data contention, resource contention, and the two policies for dealing with tardy transactions on the performance of concurrency control protocols in RTDBSs. Hence the following four system conditions were considered for experiments: soft deadline policy under finite resource situation, soft deadline policy under infinite resource situation, firm deadline policy under finite resource situation, and finally firm deadline policy under infinite resource situation. For each of the conditions, the system performance was evaluated over a wide range of workload. System workload is controlled by the arrival rate

of transactions.

### 2.5.1. Experiment 1: Soft Deadline and Finite Resources

We first evaluate the performance of the concurrency control algorithms under the condition that the system has limited amount of resources and all transactions have soft deadline. In this experiment, the number of CPUs and the number of disks are fixed two and four, respectively. Tardy transactions are given higher priority than feasible transactions. In this experiment, in addition to the two concurrency control algorithms, 2PL-HP and OCC-FV, the performance levels achievable in the absence of any data contention is also obtained under the title of NO-DC, as a basis for comparison. The performance of NO-DC is obtained by fixing the write probability at zero, and it reflects the contribution of resource contention in the system performance.

Figures 2.3 and 2.4 show Miss Percentage and Average Tardy Time behavior of the two algorithms and NO-DC under a wide range of system workload levels. Note that the shape of the graphs is similar to that of $MP_{SD}$ obtained in Section 2.3. For low arrival rate under 15 transactions/second, Miss Percentage and Average Tardy Time increase slowly. However, as the arrival rate increases and the system is saturated with transactions, Miss Percentage increases drastically and most transactions miss their deadline.

In those graphs, OCC-FV saturates slightly earlier than 2PL-HP and has worse Average Tardy Time performance. The cause of the earlier saturation of OCC-FV is its slightly higher number of restarts. When the system is limited by the amount of resources, the number of restarts has a significant impact on the performance, because the resource contention increases as the number of restarts increases. Although it loses some of the basic blocking factor due to the restart-based High Priority scheme, 2PL-HP partly uses blocking for conflict resolution, and performs better than OCC-FV under the limited resource condition. Another reason for the better performance of 2PL-HP is that it consistently

detects conflicts earlier than OCC-FV.

Throughput graphs of the two algorithms and NO-DC are shown in Figure 2.5. These graphs also show that 2PL-HP, which incurs fewer restarts, performs better than OCC-FV. As the arrival rate increases higher than 15 transactions/second, each of the graphs shows thrashing. Since the thrashing is caused primarily by resource contention, especially for NO-DC case, we refer to it as *Resource Contention (RC) thrashing*. The RC-thrashing happens because too many transactions are tied up in resource queues, thus reducing system utilization. The occurrence of thrashing is also observed as a sudden increase in average response time, as shown in Figure 2.6.

The results of this experiment conclude that 2PL-HP outperforms OCC-FV when available resources are limited and all the transactions are required to complete as in conventional database systems. This conclusion is similar to that in conventional database systems. In [Agra87, Care84], it was concluded that two-phase locking was found to outperform optimistic algorithm in an environment where physical resources are limited.

The result of better performance of 2PL-HP than OCC-FV also agrees with that of the experimental study in [Huan91]. This result is reasonable because the experimental environment used in [Huan91] is characterized as a soft deadline system with limited resources. In their system, all transactions are required to complete eventually regardless of whether they meet deadlines or not. Also their testbed system consists of one VAX workstation with two disks, one for the database and the other for the log. As other actual testbed systems for performance evaluation study, their system is limited by the amount of physical resources. A point to note is that while our simulation model is based on an open model, the work in [Huan91] used a closed queueing model where the number of active transactions in the system, i.e., the multiprogramming level, is limited at any time. Thus in the work in [Huan91], the system was not driven beyond the point where the system is saturated.

### 2.5.2. Experiment 2: Soft Deadline and Infinite Resources

In this experiment, we study the comparative performance of the two concurrency control algorithms when the amount of available resources is infinite. As in the previous experiment, we assume that transactions have soft deadline, and tardy transactions are given higher priority than feasible transactions and required to complete. To simulate an infinite resource situation, we eliminate queueing of data operations for resources and allow any request for CPU processing or disk access to be fulfilled immediately. In this experiment, the write probability is fixed at 0.5 to maintain a relatively high data contention. Thus, we expect the system performance to be determined by concurrency control algorithm rather than resource scheduling algorithms. Under the infinite resource condition, NO-DC case will not be shown.

Figures 2.7 and 2.8 show Miss Percentage and Average Tardy Time behavior of the two algorithms under different levels of system workload. The shape of the graphs is again similar to that of $MP_{SD}$ in Figure 2.1(a). At a certain point of workload, both Miss Percentage and Average Tardy Time for the two algorithms show drastic increase, i.e., thrashing. Note that the cause of this thrashing phenomenon is different from that of RC-thrashing in the previous experiment. In this experiment, since there is no resource contention, thrashing is caused by data contention alone, and we refer to this as *Data Contention (DC) thrashing*.

Furthermore, the cause of the DC-thrashing in optimistic protocol is different from that of blocking-based algorithm. In optimistic protocol, where all data contention is resolved by restarts, the cause of thrashing is restarts. If the data contention level is high, then transactions are busy being repeatedly restarted, so transactions make little progress. Under blocking-based concurrency control algorithm, DC-thrashing happens because too many transactions are tied up in lock queues.

Unlike the previous experiment, under this infinite resource condition, OCC-FV shows better performance both in miss percentage and in average tardy time. The performance difference between the two algorithms becomes larger as the workload increases.

Throughput graphs of the two algorithms are shown in Figure 2.9. The graphs again show the DC-thrashing points of the two algorithms clearly, and that OCC-FV outperforms 2PL-HP in throughput. Figure 2.10 shows the average data blocking time, i.e., lock duration of the two protocols. For the reasons explained in Section 2.4, the optimistic protocol shows shorter blocking time than 2PL-HP.

Based on the results of this experiment, we conclude that OCC-FV outperforms 2PL-HP under an infinite resource situation. Similar conclusion was obtained in the study of conventional database systems [Agra87]. If resource utilizations are low enough so that a large amount of wasted resources can be tolerated, and in addition there are a large number of transactions available to execute, then an optimistic algorithm that allows a higher degree of concurrent execution is a better choice.

We conducted another experiment where the write probability is fixed at 0.5, the arrival rate was fixed at 15 transactions/second, and the number of resources in the system was varied. In varying the number of CPUs and disks, we decided to use one CPU and two disks as one *resource unit* as in [Agra87], and we vary the number of resource units rather than the number of CPUs and the number of disks individually. We use this setting of the number of CPUs and disks to balance the utilization of the two resources with our parameter values, as opposed to being either strongly CPU bound or strongly I/O bound. The result of this experiment is shown in Figure 2.11 and it indicates that there is a crossover point, in terms of the amount of available resources, between the relative performance of 2PL-HP and OCC-FV. This summarizes the results of the experiments in this and the previous section.

### 2.5.3. Experiment 3: Firm Deadline and Finite Resources

Until now, we have considered the comparative performance of the two concurrency control algorithms under the soft deadline assumption. Now let us turn our attention to firm deadline systems, where tardy transactions are required to be aborted and permanently discarded from the system, because completing a transaction after its deadline has expired is of no value and may even be harmful. In this section, we evaluate the performance of firm deadline database systems under the condition of limited resources. The condition of infinite resources will be considered in the next section. In this experiment, the number of CPUs and the number of disks are fixed two and four, respectively. Under the firm deadline assumption, we do not need the performance metric Average Tardy Time, since tardy transactions are discarded from the system instead being completed. We use Miss Percentage alone as the performance metric for the experiments.

Figure 2.12 shows Miss Percentage behavior of the algorithms under a wide range of system workload. In this experiment, the write probability is fixed at 0.25. Note that the shape of the graphs is similar to that of $MP_{FD}$ in Figure 2.1(b). From the graph, it is clear that for very low arrival rate under 10 transactions/second, there is no significant difference for the two protocols. However, as the arrival rate increases, OCC-FV does progressively better than 2PL-HP.

The reason for the better performance of OCC-FV over 2PL-HP in this experiment is different from that in the previous section. The cause of the performance difference can be found in the number of restarts, Restart Count, incurred by each of the protocols, shown in Figure 2.13. Unlike in the previous experiment, OCC-FV has much fewer restarts and hence makes better use of resources than 2PL-HP in this experiment. 2PL-HP suffers performance degradation caused by wasted restarts. That is, the immediate conflict resolution policy of 2PL-HP allows a transaction that will eventually miss its deadline and be discarded to restart other transactions. This performance degradation increases as the

workload level increases, since the number of transactions that miss their deadlines and have to be discarded increases. The delayed conflict resolution policy of optimistic algorithms helps them to avoid such wasted restarts.

A point to note here is that Restart Count of both the algorithms decreases after a certain workload level. The reason for this decrease is that after that workload point, resource contention dominates data contention in discarding deadline-missing transactions.

### 2.5.4. Experiment 4: Firm Deadline and Infinite Resources

In the previous section, the performance of the two concurrency control protocols was evaluated under a limited resource situation. In such situation, since resource contention dominates data contention as system workload increases, the performance of the system is primarily determined by resource scheduling algorithms rather than concurrency control. In fact, the performance difference shown in Figure 2.12 may not be very striking. To capture the performance difference of concurrency control algorithms without the effect of resource contention, we simulated an infinite resource situation, where there is no queueing for resources (CPUs and disks). However, the data contention is maintained relatively high with the write probability fixed at 0.5 in this experiment.

Figures 2.14 and 2.15 show Miss Percentage and Restart Count of the two protocols. As we expected, these graphs show the performance difference of the two more clearly than in the previous experiment. Note that since there is no resource contention, the restart counts of the two protocols ever increase as the system workload increases. The locking duration of the two protocols shown in Figure 2.16 again illustrates that the optimistic protocol has shorter blocking time than 2PL-HP.

Based on the experiment results in this and the previous section, we conclude that OCC-FV outperforms 2PL-HP in firm deadline systems in a wide range of system workload. Generally, the performance difference between the two tends to increase as the amount of

physical resources available in the system increases. This concept is quantified in the Miss Percentage graph in Figure 2.17. For this graph, we set the write probability to 0.5, fixed the arrival rate at 20 transactions/second, and varied the amount of available resources using the resource unit defined earlier.

The work in [Hari90] presented a quantitative study of the relative performance of locking and optimistic concurrency control techniques in the context of a real-time database system. In particular, they claimed that the policy for dealing with tardy transactions has a significant impact on the relative behavior of the concurrency control algorithms, and under the firm deadline assumption, OCC-FV outperforms 2PL-HP over a wide range of system loading and resource availability. Their claims were confirmed in our experiments for firm deadline systems.

## 2.6. Summary

In this study, we argued that a reasonable real-time database system (RTDBS) model is a requirement for the performance study of concurrency control in RTDBS, and that seemingly contradictory results from previous studies on this field is caused by the fact that each of the study used different assumptions on RTDBS model. We examined the key components of a reasonable RTDBS model, including the policy for dealing with tardy transactions, and the availability of resources in the system. In addition, we discussed the issues on the implementation of concurrency control algorithms, which may have a significant impact on performance. We then presented our simulation model, on which we implemented the key components of RTDBS model, and we used it to study the relative performance of locking and optimistic concurrency control protocols for RTDBSs.

One specific conclusion of this study is that under a soft deadline system where all transactions are required to complete eventually, the results of the relative performance of locking and optimistic approaches is similar to those seen in conventional database

systems. That is, under an environment where physical resources are limited, the locking protocol that tends to conserve physical resources by blocking, instead of restarting, transactions outperforms the optimistic concurrency control. However, when the amount of available resources is abundant so that a large amount of wasted resources can be tolerated and the system performance is determined primarily by concurrency control rather than other resource scheduling, the optimistic protocol that allows a higher degree of concurrent execution outperforms the locking protocol.

In the context of firm deadline systems where tardy transactions are required to be aborted and permanently discarded from the system, optimistic protocol outperforms locking under a wide range of resource availability and system workload level. The performance difference between the two under firm deadline assumption is caused by the fact that locking protocol suffers performance degradation caused by wasted restarts. The reason for the wasted restarts in 2PL-HP is its immediate conflict resolution policy which allows a transaction that will eventually miss its deadline to restart other transactions. Optimistic approach can avoid such wasted restarts because of its policy of delayed conflict resolution.

A more general result of this study is that we have reconfirmed results from previous performance studies on concurrency control in RTDBS including [Hari90, Hari92, Huan91]. We have shown that seemingly contradictory performance results, some of which favored locking-based protocol and others of which favored optimistic scheme, are not contradictory at all. The studies are all correct within the limits of their assumptions, particularly their assumptions about resource availability and policy for dealing with tardy transactions. A reasonable model for these assumptions is critical for the relative performance studies of concurrency control algorithms in RTDBSs.

Figure 2.3 Miss Percentage, Write Probability = 0.25
Finite Resource, Soft Deadline

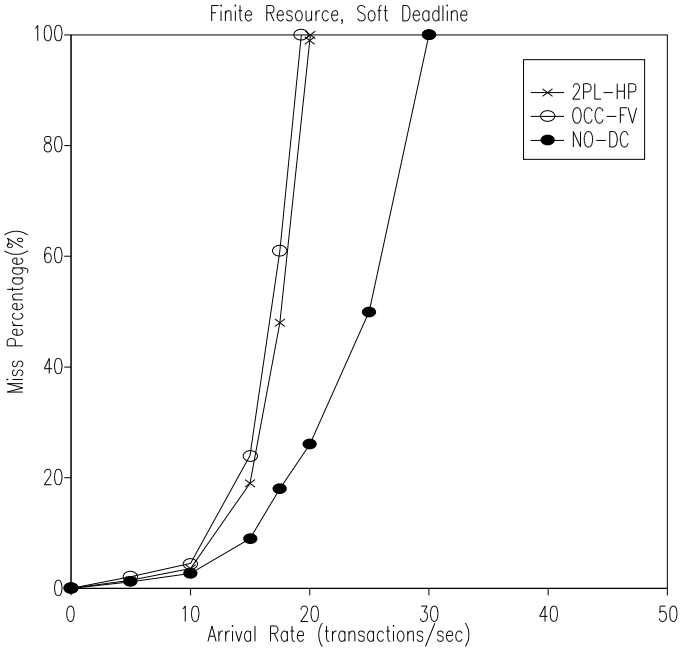



Figure 2.4 Average Tardy Time, Write Probability = 0.25
Finite Resource, Soft Deadline

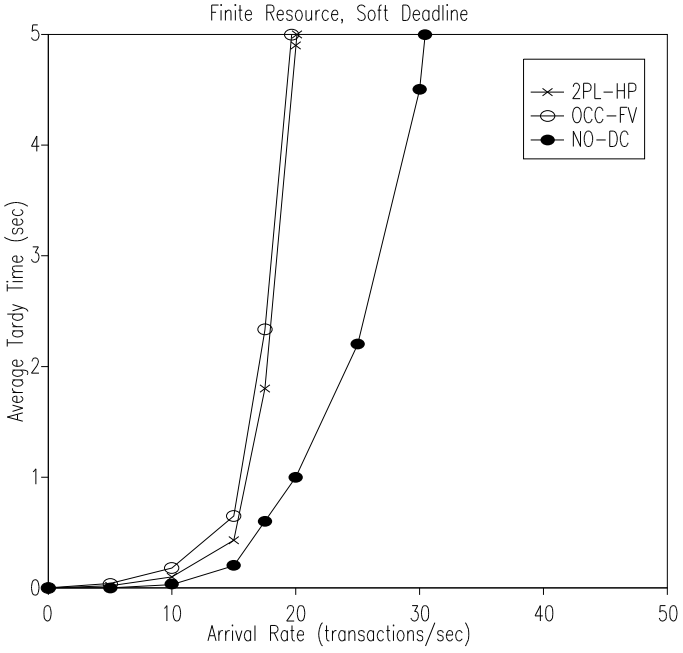


Figure 2.5 Throughput, Write Probability = 0.25
Finite Resource, Soft Deadline

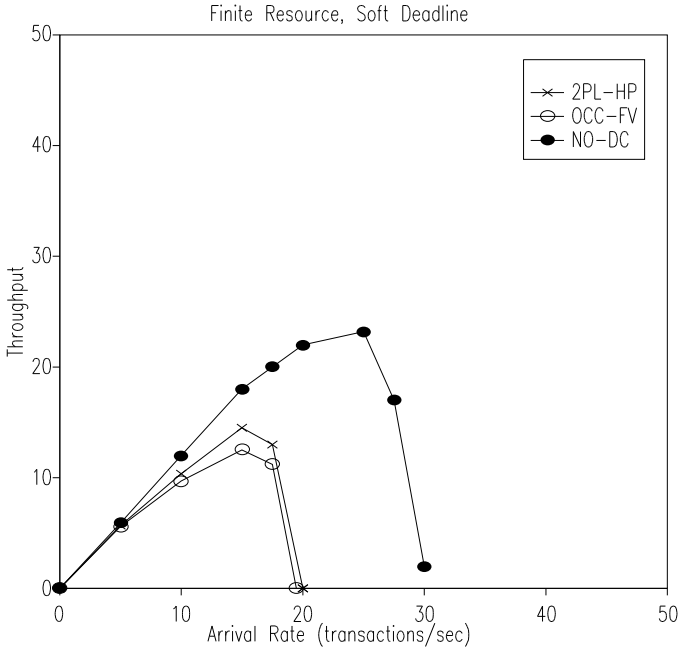



Figure 2.6 Average Response Time, Write Probability = 0.25
Finite Resource, Soft Deadline

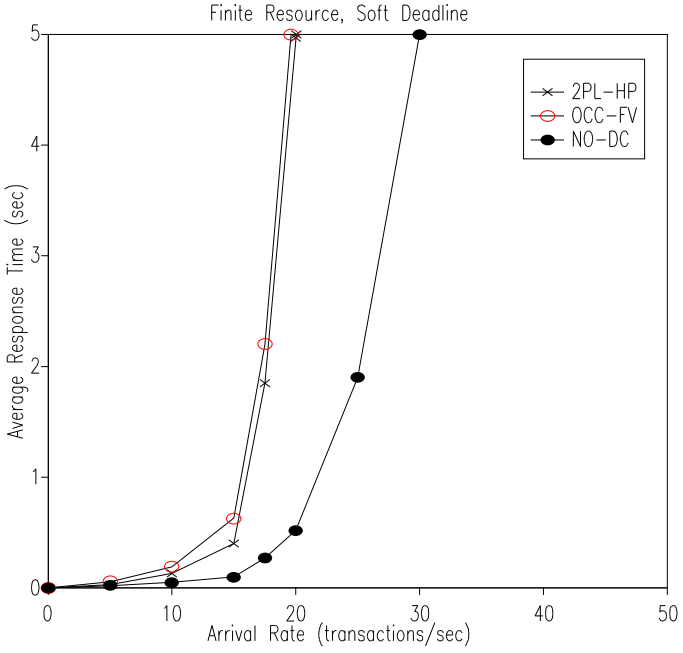


Figure 2.7 Miss Percentage, Write Probability = 0.5
Infinite Resource, Soft Deadline

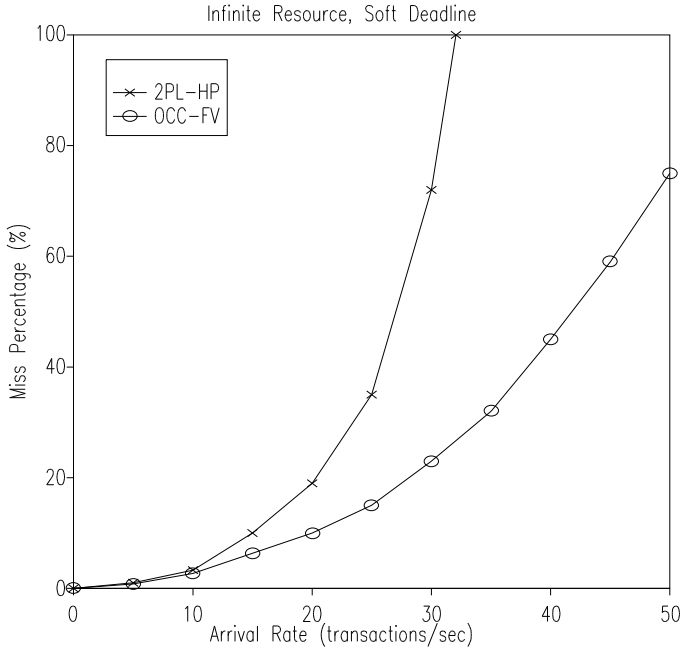



Figure 2.8 Average Tardy Time, Write Probability = 0.5
Infinite Resource, Soft Deadline

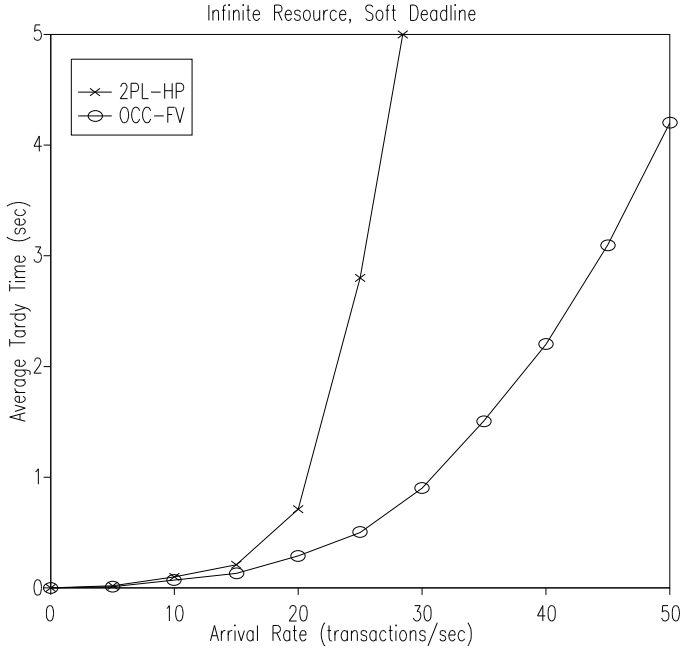


Figure 2.9 Throughput, Write Probability = 0.5
Infinite Resource, Soft Deadline

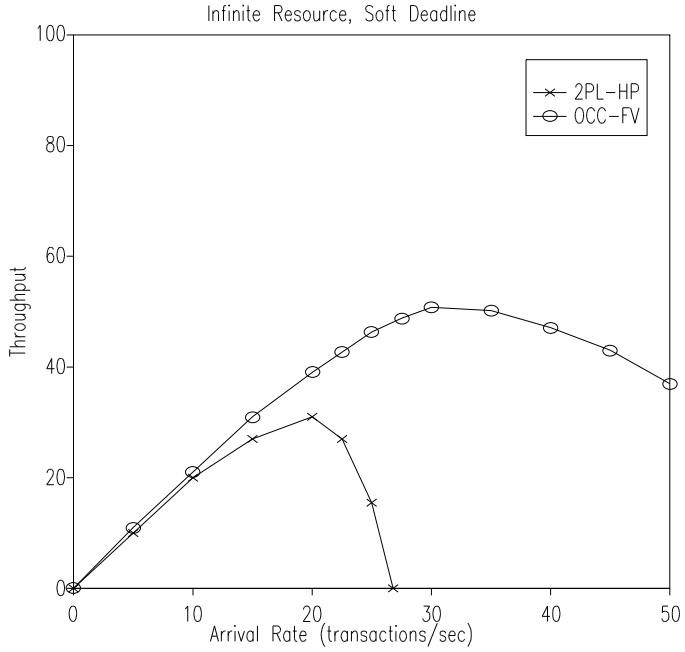



Figure 2.10 Data Blocking Time, Write Probability = 0.5
Infinite Resource, Soft Deadline

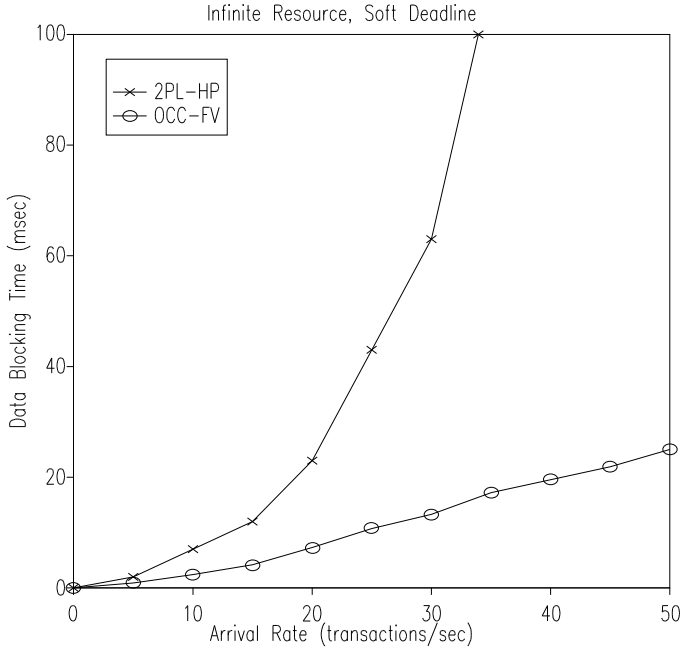


Figure 2.11 Miss Percentage, Write Probability = 0.5
Soft Deadline, Arrival Rate = 15 transactions/sec

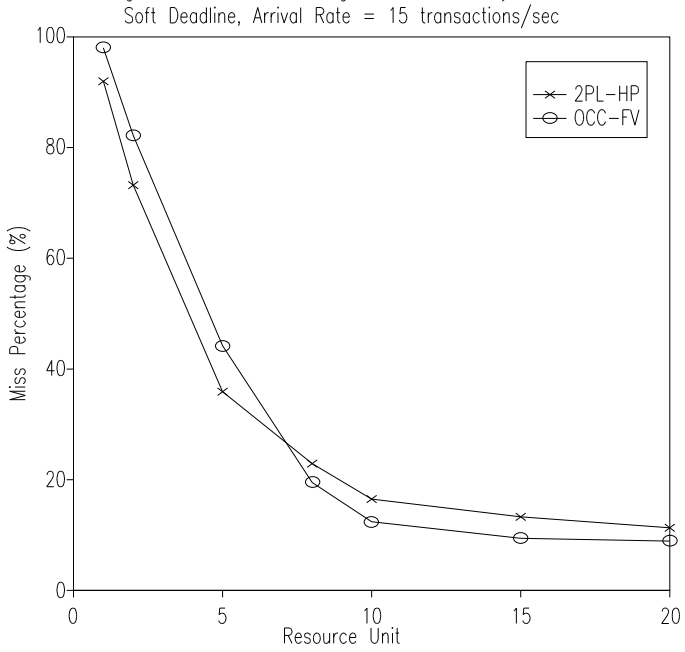



Figure 2.12 Miss Percentage, Write Probability = 0.25
Finite Resource, Firm Deadline

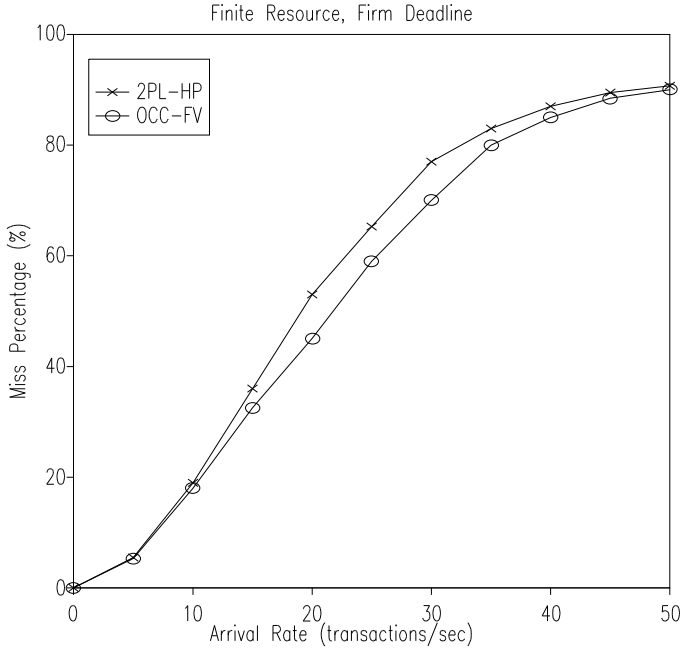


Figure 2.13 Restart Count, Write Probability = 0.25
Finite Resource, Firm Deadline

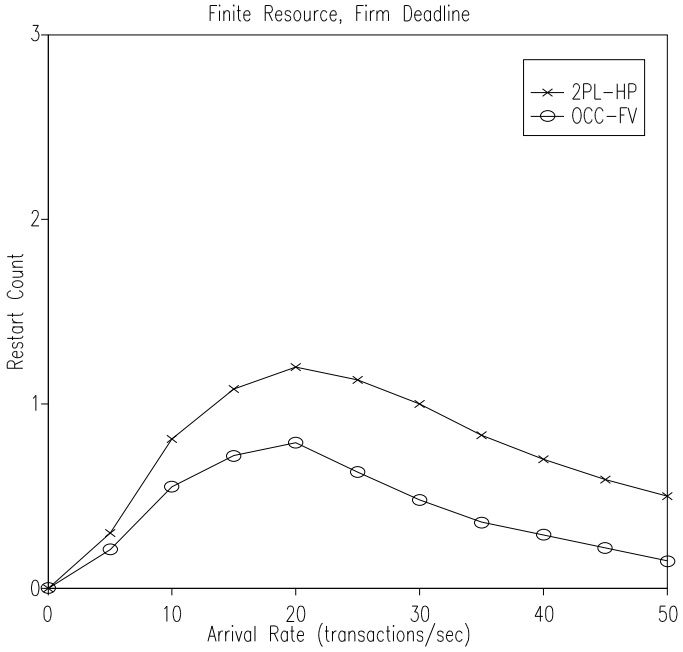



Figure 2.14 Miss Percentage, Write Probability = 0.5
Infinite Resource, Firm Deadline

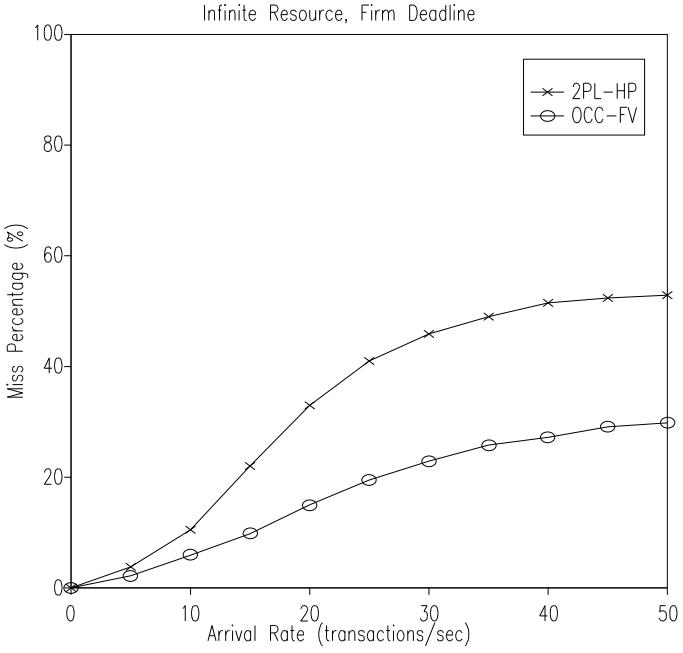


Figure 2.15 Restart Count, Write Probability = 0.5
Infinite Resource, Firm Deadline

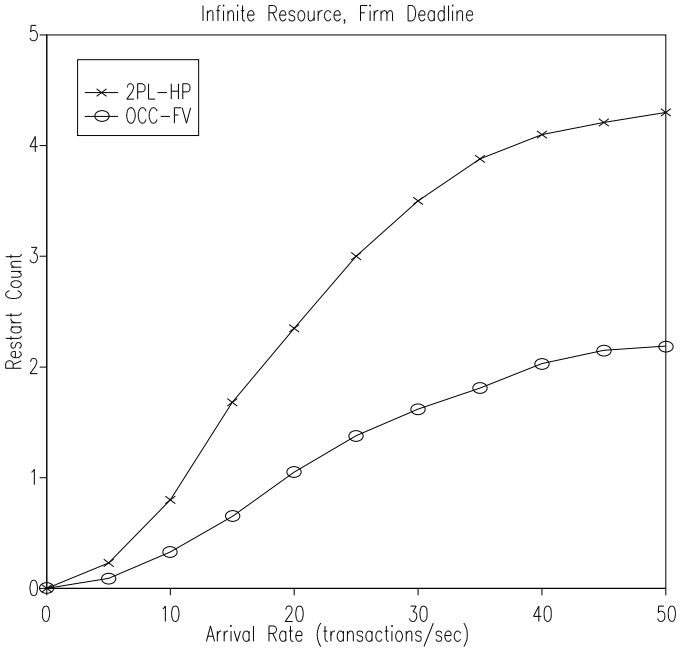



Figure 2.16 Data Blocking Time, Write Probability = 0.5
Infinite Resource, Firm Deadline

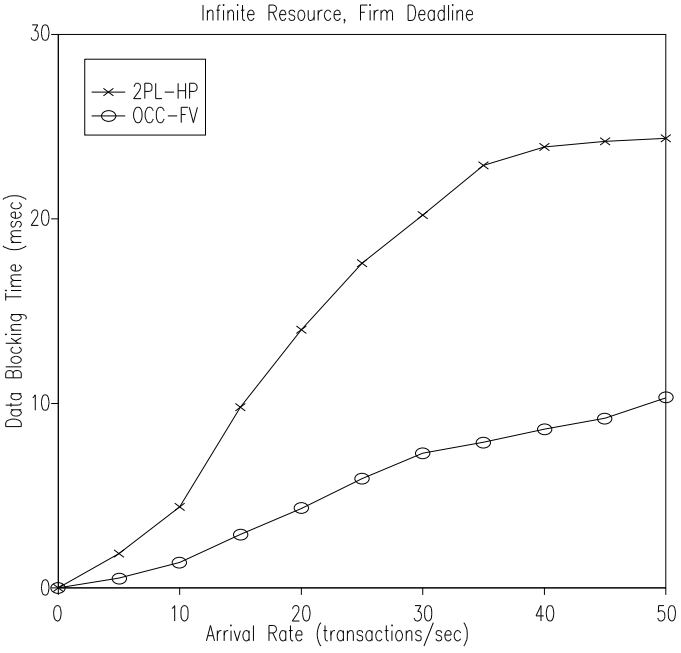
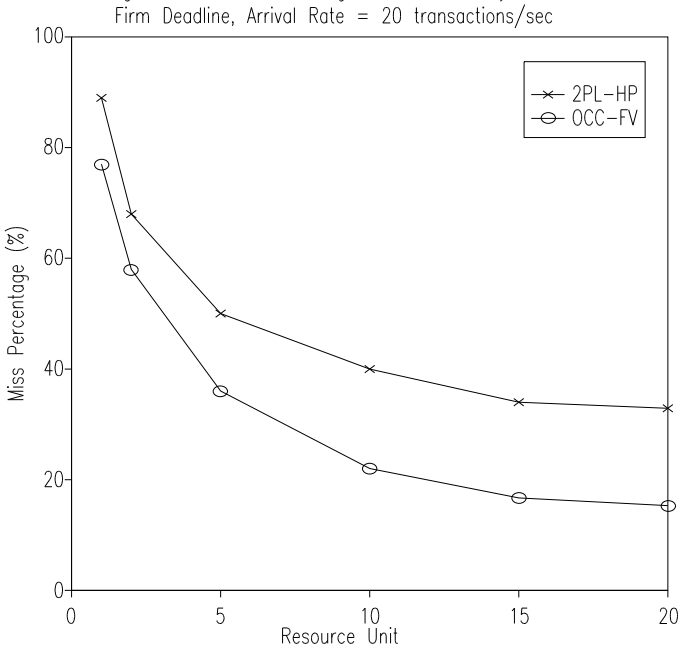


Figure 2.17 Miss Percentage, Write Probability = 0.5
Firm Deadline, Arrival Rate = 20 transactions/sec



# 3. Dynamic Adjustment of Serialization Order

## 3.1. Introduction

Studies in [Hari90, Hari90b] showed that under the condition that tardy transactions are discarded from the system, optimistic concurrency control outperforms locking over a wide range of system loading and resource availability. The key reason for this result was described that the optimistic method, due to its validation stage conflict resolution, ensures that eventually discarded transactions do not restart other transactions. In the locking approach, however, soon-to-be-discarded transactions may restart other transactions. These restarts are referred to as *wasted restarts*.

In [Hari90b], the problem of adding transaction timing information to optimistic concurrency control was addressed. They showed that the problem is nontrivial partly because giving preferential treatment to high priority transactions may result in an increase in the number of missed deadlines. In particular, the delayed conflict resolution policy of optimistic algorithms significantly reduces the possibility that a validating transaction sacrificed for an active transaction with a higher priority will meet its deadline. In addition, this situation may aggravate the real-time performance of the system in two more ways. One is that the validating transaction is restarted after spending most of the time and resources for its execution. The other is that there is no guarantee that the active transaction which caused the restart of the validating transaction will meet its deadline. If the active transaction does not meet its deadline for any reason, the sacrifice of the validating

transaction is wasted. In [Hari90b], they studied several alternative schemes of incorporating transaction priority information into optimistic algorithms, including a scheme based on a priority wait mechanism with a wait control technique. However, this study and others [Huan91, Lee93c] showed that none of those schemes constantly outperforms the priority-incognizant algorithm.

The results of these studies suggest several implications. First, the choice of basic concurrency control mechanism has a significant impact on the performance of RTDBSs. As shown in [Hari90], although an optimistic algorithm does not take transaction deadlines into account in making data conflict resolution decisions, it can still outperform a deadline-cognizant locking algorithm in a real-time database environment. Second, the number of restarts incurred by concurrency control is the major factor deciding the performance of concurrency control in real-time database systems, that is, having more restarts leads to poorer performance. Therefore restarts should be avoided if possible. In fact, the same result was derived in the studies of conventional database system performance [Agra87]. Finally, more study should address the problem of designing deadline-cognizant optimistic concurrency control algorithms. A practical real-time concurrency control algorithm should provide significant performance gains especially under high levels of system workload.

In this chapter, we focus on a method to reduce the number of restarts unnecessary to ensure data consistency, and present a new optimistic concurrency control algorithm that can avoid such “unnecessary restarts” by adjusting serialization order of transactions dynamically by associating a timestamp interval with every running transaction. It appears that this protocol is a promising candidate for basic concurrency control mechanism for real-time database systems. The design of deadline-cognizant concurrency control schemes based on this algorithm for RTDBSs will be discussed in Chapter 4.

The remainder of this chapter is organized in the following fashion: Section 3.2

reviews the principle of optimistic concurrency control, and shows deficiencies of the optimistic methods used in previous studies. A new optimistic algorithm is presented in Section 3.3. Section 3.4 describes our real-time database environment for experiments. In Section 3.5, the results of the simulation experiments are highlighted. Finally, Section 3.6 summarizes the main conclusions of the study.

## 3.2. Optimistic Concurrency Control

In this section, we first discuss the principles underlying optimistic concurrency control, particularly regarding its validation. Then we show an example of unnecessary restarts incurred by the optimistic algorithms used in the previous studies.

### 3.2.1. Principles

In optimistic concurrency control, transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. Thus, the execution of a transaction consists of three phases: read, validation, and write [Kung81]. The key component among these is the validation phase where a transaction's destiny is decided. Validation comes in several flavors, but every validation scheme is based on the following principle to ensure serializability.

If a transaction $T_i$ is serialized before transaction $T_j$, the following two conditions must be satisfied:

**Condition 1**: *No overwriting*

The writes of $T_i$ should not overwrite the writes of $T_j$.

**Condition 2**: *No read dependency*

The writes of $T_i$ should not affect the read phase of $T_j$.

Generally, Condition 1 is automatically ensured in most optimistic algorithms because I/O operations in the write phase are performed sequentially in critical section. Thus most validation processes consider only Condition 2, and it can be carried out

basically in either of the following two ways [Haer84].

**Backward Validation**

In this scheme, the validation process is carried out against (recently) committed transactions. Data conflicts are detected by comparing the read set of the validating transaction and the write set of committed transactions, since it is obvious that committed transactions precede the validating transaction in serialization order. Such data conflicts should be resolved to ensure Condition 2. The only way to do this is to restart the validating transaction. The classical optimistic algorithm in [Kung81] is based on this validation process.

Let $T_v$ be the validating transaction and $T_c$ ($c = 1, 2,...., n, c \neq v$) be the transactions recently committed with respect to $T_v$, i.e., those transactions that commit between the time when $T_v$ starts executing and the time at which $T_v$ enters the validation phase. Let *RS(T)* and *WS(T)* denote the read set and write set of transaction *T*, respectively. Then the backward validation operation can be described by the following procedure:

```
validate(Tv);
{
    valid := true;
    foreach Tc (c = 1, 2,..., n) {
        if WS(Tc) ∩ RS(Tv) ≠ {} then valid := false;
        if not valid then exit loop;
    }
    if valid then commit WS(Tv) to database
    else restart(Tv);
}
```

**Forward Validation**

In this scheme, validation of a transaction is done against currently running transactions. This process is based on the assumption that the validating transaction is ahead of every concurrently running transaction still in read phase in serialization order. Thus the detection of data conflicts is carried out by comparing the write set of the validating transaction and the read set of active transactions. That is, if an active transaction, $T_i$, has read an object that has been concurrently written by the validating transaction, the value of the object used by $T_i$ is not consistent. Such data conflicts can be resolved by restarting either the validating transaction or the conflicting transactions in the read phase. Optimistic algorithms based on this validation process are studied in [Haer84, Robi82].

Let $T_a$ ($a = 1, 2,..., n, a \neq v$) be the conflicting transactions in their read phase. Then the forward validation can be described by the following procedure:

```
validate(T_v);
{
    valid := true;
    foreach T_a (a = 1, 2,..., n) {
        if RS(T_a) ∩ WS(T_v) ≠ {} then valid := false;
    }
    if valid then commit WS(T_v) to database
    else conflict resolution(T_v);
}
```

In real-time database systems, data conflicts should be resolved in favor of higher priority transactions. In backward validation, there is no way to take transaction priority into account in the serialization process, since it is carried out against already committed transactions. Thus backward validation is not amenable to real-time database systems. Forward validation provides flexibility for conflict resolution that either the validating transaction or the conflicting active transactions may be chosen to restart, so it is preferable

for real-time database systems. In addition to this flexibility, forward validation has the advantage of early detection and resolution of data conflicts.

All the optimistic algorithms used in the previous studies of real-time concurrency control in [Hari90, Hari90b, Huan91] are based on the forward validation. The broadcast mechanism in the algorithm, OPT-BC studied in [Hari90, Hari90b, Robi82], is an implementation variant of the forward validation. We refer to this algorithm as OCC-FV.

### 3.2.2. Unnecessary Restarts

As we mentioned above, forward validation is based on the assumption that the serialization order among transactions is determined by the arriving order of transactions at validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active transaction in serialization order. We claim that this assumption is not only unnecessary, but also the validation process based on this assumption can incur restarts not necessary to ensure data consistency. These restarts should be avoided. To ensure this claim, let us consider the following example.

**Example 1**: Let $r_i[x]$ and $w_i[x]$ denote a read and write operation, respectively, on the data object $x$ by transaction $i$, and let $v_i$ and $c_i$ denote the validation and commit of transaction $i$, respectively. Consider three transactions $T_1$, $T_2$, and $T_3$:

$T_1$: $r_1[x]$ $w_1[x]$ $r_1[y]$ $w_1[y]$ $v_1$
$T_2$: $r_2[x]$ $w_2[x]$ ... $v_2$
$T_3$: $r_3[y]$ ... $v_3$

and suppose they execute as follows:

$H_1 = r_1[x]\ w_1[x]\ r_2[x]\ r_3[y]\ w_2[x]\ r_1[y]\ w_1[y]\ v_1\ c_1$.

If we use the forward validation process described above for the validation of $T_1$, both the active transactions, $T_2$ and $T_3$ are conflicting with $T_1$ on data items $x$ and $y$, respectively,

and should restart. It is fair for $T_2$ to restart since it has both write-write and write-read conflicts with $T_1$. However, $T_3$, we observe, does not have to restart, if there is no more conflict with $T_1$. In fact no serialization order between $T_1$ and $T_3$ has been built except for the read-write conflict on $y$. If we set the serialization order between $T_1$ and $T_3$ as $T_3 \rightarrow T_1$ during the validation of $T_1$, we can ensure data consistency without restarting $T_3$.■

We refer to such a restart of $T_3$ in the forward validation as an *unnecessary restart*. Also, we refer to transactions having both write-write and write-read conflicts with the validating transaction like $T_2$ as *irreconcilably conflicting*, while transactions having only write-read conflicts like $T_3$ as *reconcilably conflicting*.

The design of the new optimistic algorithm presented in the next section is based on this categorization of active transactions. As we will explain, the categorization is automatically done by adjusting and recording the current serialization order dynamically using timestamp intervals associated with each active transaction. The performance gain by the new algorithm can be significant especially when reconcilable conflicts dominate, that is, the probability that a data object read is updated is low, which is true for most actual database systems. Generally, under a wide range of system workload, the algorithm provides a performance advantage by reducing the number of restarts at the expense of maintaining the serialization order dynamically.

## 3.3. A New Optimistic Concurrency Control Algorithm

In this section, we present the proposed optimistic algorithm in detail. We first explain the mechanism to guarantee serializability used in the algorithm, and then prove its correctness. At the end, we discuss the advantages and disadvantages of this protocol. We hereafter refer to this algorithm as OCC-TI.

### 3.3.1. Validation Phase

In this protocol, every transaction in the read phase is assigned a timestamp interval,

which is used to record temporary serialization order induced during the execution of the transaction. At the start of execution, the timestamp interval of a transaction is initialized as [0,∞), i.e., the entire range of timestamp space. Whenever serialization order of a transaction is induced by its data operation or the validation of other transactions, its timestamp interval is adjusted to represent the dependencies. In addition to the timestamp interval, a final timestamp is assigned to each transaction which has successfully passed its validation test and guaranteed to commit.

In this algorithm, a transaction that finishes read phase and reaches validation is always guaranteed to commit as in OCC-FV. However, unlike OCC-FV in which a transaction is validated by comparing its write set and the read sets of transactions, the validation of a transaction in OCC-TI consists of adjusting the timestamp intervals of concurrent transactions.

Let *TI(T)* and *TS(T)* denote the timestamp interval and final timestamp of transaction *T*, respectively. $T_a$ ($a = 1, 2,..., n, a \neq v$) again denotes the conflicting transactions in their read phase. Then the validation process can be briefly described by the following procedure:

```
validate(T_v);
{
    select TS(T_v) from TI(T_v);
    foreach T_a (a = 1, 2, ... , l) {
        adjust(T_a);
    }
    foreach D_i (i = 1, 2, ... , m) in RS(T_v) {
        update RTS(D_i);
    }
    foreach D_j (j = 1, 2, ... , n) in WS(T_v) {
        update WTS(D_j);
    }
```

```
        commit WS(T_v) to database;
    }
```

First, the final timestamp, $TS(T_v)$, is determined from the timestamp interval, $TI(T_v)$. In fact, any timestamp in $TI(T_v)$ can be chosen to be $TS(T_v)$, because any value in $TI(T_v)$ preserves the serialization order induced by $T_v$. In this algorithm, we always select the minimum value of $TI(T_v)$ for $TS(T_v)$ for a practical reason, which will be made clear later. Once selected, $TS(T_v)$ is used in the following steps of the validation process. Second, the timestamp intervals of all the concurrently running transactions that have accessed common objects with $T_v$ are adjusted to reflect the serialization order induced between $T_v$ and those transactions. Any active transaction whose timestamp interval shuts out by the adjustment operation should restart, because it has introduced nonserializable execution with respect to $T_v$. The details of the adjustment procedure will be described below. Finally, if necessary, the final timestamp of the committing transaction is recorded for every data object it has accessed. *RTS(D)* and *WTS(D)* denote the largest timestamp of committed transactions that have read and written, respectively, data object *D*. The need of this operation will be explained in the next section.

The salient feature of OCC-TI is that unlike other optimistic algorithms, it does not depend on the assumption of the serialization order of transactions being the same as the validation phase arriving order, but it records serialization order induced precisely and uses restarts only when necessary. Let us examine how serialization order is adjusted between the validating transaction and a concurrently active transaction for the three possible types of conflict:

**Read-write conflict** ($RS(T_v) \cap WS(T_a) \neq \{\}$)

This type of conflicts leads the serialization order between $T_v$ and $T_a$ to $T_v \rightarrow T_a$. That is, the timestamp interval of $T_a$ is adjusted to follow that of $T_v$. We refer to this ordering as

*forward ordering*. The implication of this ordering is that the read phase of $T_v$ is not affected by the writes of $T_a$.

**Write-read conflict** ($WS(T_v) \cap RS(T_a) \neq \{\}$)

In this case, the serialization order is recorded as $T_a \rightarrow T_v$. That is, the timestamp interval of $T_a$ is adjusted to precede that of $T_v$. This ordering is referred to as *backward ordering.* It implies that the writes of $T_v$ have not affected the read phase of $T_a$.

**Write-write conflict** ($WS(T_v) \cap WS(T_a) \neq \{\}$)

A write-write conflict results in forward ordering, i.e., $T_v \rightarrow T_a$. Thus the order implies that $T_v$'s writes do not overwrite $T_a$'s writes.

Non-serializable execution is detected when the timestamp interval of an active transaction shuts out. The non-serializable execution is deleted from execution history by restarting the transaction. Obviously, the timestamp interval of an active transaction that requires both backward and forward ordering to record the execution of its operations will shut out. Such transaction are irreconcilably conflicting with the validating transaction.

The adjustment of timestamp intervals of active transactions is the process of recording serialization order according to the conflict types and their corresponding ordering. It can be described by the following procedure. We assume that timestamp intervals contain only integers.

```
adjust(Ta);
{
    foreach Di (i = 1, 2, ... , m) in RS(Tv) {
        if Di in WS(Ta)  then TI(Ta) := TI(Ta) ∩ [TS(Tv), ∞);
        if TI(Ta) = [ ] then restart(Ta);
    }
    foreach Dj (j = 1, 2, ..., n) in WS(Tv) {
```

```
            if D_j in RS(T_a)  then TI(T_a) := TI(T_a) ∩ [0, TS(T_v)-1];
            if D_j in WS(T_a)  then TI(T_a) := TI(T_a) ∩ [TS(T_v), ∞);
            if TI(T_a) = [ ] then restart(T_a);
        }
    }
```

### 3.3.2. Read Phase

The adjustment of active transactions' timestamp intervals at the validation of a transaction is the process of recording the serialization order between the committing transaction and the data operations performed by the concurrently running transactions until the moment. Because the active transactions continue to execute remaining data operations, the execution order induced by the remaining operations should be checked to determine if they induce any non-serializable execution. If so, the active transaction should restart. The following example demonstrates such late restart.

**Example 2**: Consider two transactions $T_1$ and $T_2$:

$T_1$: $r_1[y]$ $w_1[y]$ $v_1$
$T_2$: $r_2[y]$ $w_2[y]$ ... $v_2$

and suppose they execute as follows:

$H_2 = r_1[y]\ r_2[y]\ w_1[y]\ v_1\ c_1\ w_2[y]$ ...

At the validation of $T_1$, $T_2$ has only a write-read conflict with $T_1$. With the backward ordering, the serialization order between $T_1$ and $T_2$ is set as $T_2 \rightarrow T_1$, and $T_2$ is not restarted. However, later the write operation of $T_2$, $w_2[y]$, induces a serialization order between $T_1$ and $T_2$ in opposite direction. Thus $T_2$ has to restart.■

The detection of non-serializable execution by remaining operations of active transactions can also be done using the timestamp intervals. Because, in this case, the

serialization order of active transactions is checked against committed transactions, we need to use the timestamps of data objects, i.e., *RTS(D)* and *WTS(D)* of data object *D*. In the read phase, whenever a transaction performs a data operation, its timestamp interval is adjusted to reflect the serialization induced between the transaction and committed transactions. If the timestamp interval shuts out, a non-serializable execution performed by the transaction is detected, and the transaction restarts. The process can be described by the following procedure:

```
read_phase(T_a);
{
    foreach D_i (i = 1, 2, ..., m) in RS(T_a) {
        read(D_i);
        TI(T_a) := TI(T_a) ∩ [WTS(D_i), ∞);
        if TI(T_a) = [ ] then restart(T_a);
    }
    foreach D_j (j = 1, 2, ..., n) in WS(T_a) {
    pre-write(D_j);
    TI(T_a) := TI(T_a) ∩ [WTS(D_j), ∞) ∩ [RTS(D_j), ∞);
    if TI(T_a) = [ ] then restart(T_a);
    }
}
```

**Example 3**: To understand how this procedure works, let us consider the previous example again. The execution history is given as follows:

$$H_2 = r_1[y]\ r_2[y]\ w_1[y]\ v_1\ c_1\ w_2[y].$$

At its validation, $T_1$ is first assigned a final timestamp $TS(T_1)$, say 74. Then with backward ordering, the timestamp interval of $T_2$ is adjusted to be [0, 73]. In addition, the timestamps of data object, *RTS(y)* and *WTS(y)*, accessed by $T_1$ are updated to be 74. After the validation process, when $T_2$ performs $w_2[y]$, its timestamp is adjusted by the following operation:

$$TI(T_2) := [0, 73] \cap [74, \infty) \cap [74, \infty).$$

Because this operation leaves $TI(T_2)$ shut out, non-serializable execution is detected and $T_2$ restarts.■

Note that in OCC-FV, transactions in read or validation phase do not need to check for conflicts with already committed transactions. In this algorithm, transactions conflicting with a committed transaction would have been restarted earlier by the committed transaction [Hari90].

### 3.3.3. Write Phase

Once a transaction is in the write phase, it is considered to be committed. All committed transactions can be serialized by the final timestamp order. In the write phase, the only work of a transaction is making all its updates permanent in the database. Data objects are copied from the local workspace into the database. Since a transaction applies the results of its write operations only after it commits the *strictness* [Bern87] of the histories produced by OCC-TI is guaranteed. This property makes the transaction recovery procedure simpler than non-strict concurrency control protocols.

### 3.3.4. Correctness

In this section, we give an argument on the correctness of the algorithm. First, we give simple definitions of history and serialization graph (*SG*). The formal definitions for these concepts can be found in [Bern87]. A history is a partial order of operations that represents the execution of a set of transactions. Any two conflicting operations must be comparable. Let *H* denote a history. The serialization graph for *H*, denoted by *SG(H)*, is a directed graph whose nodes are committed transactions in *H* and whose edges are all $T_i \rightarrow T_j$ ($i \neq j$) such that one of $T_i$'s operations precedes and conflicts with one of $T_j$'s operations in *H*. To prove a history *H* serializable, we only have to prove that *SG(H)* is acyclic [Bern87].

**Lemma**: Let $T_1$ and $T_2$ be two committed transactions in a history $H$ produced by the proposed algorithm. If there is an edge $T_1 \rightarrow T_2$ in *SG(H)*, then $TS(T_1) < TS(T_2)$.

*Proof*: Since there is an edge, $T_1 \rightarrow T_2$ in *SG(H)*, the two must have one or more conflicting operations whose type is one of the following three:

*Case 1*: $r_1[x] \rightarrow w_2[x]$

This case implies that $T_1$ commits before $T_2$ reaches its validation phase since $r_1[x]$ is not affected by $w_2[x]$. For $w_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $RTS(x)$ that is equal to or is greater than $TS(T_1)$. That is, $TS(T_1) \leq RTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

*Case 2*: $w_1[x] \rightarrow r_2[x]$

This case is possible only when the write phase of $T_1$ finishes before $r_2[x]$ executes in $T_2$'s read phase. For $r_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. That is, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

*Case 3*: $w_1[x] \rightarrow w_2[x]$

This case can be similarly proved to lead to $TS(T_1) < TS(T_2)$.■

**Theorem**: Every history generated by OCC-TI algorithm is serializable.

*Proof*: Let $H$ denote any history generated by the algorithm. Suppose, by way of contradiction, that *SG(H)* contains a cycle $T_1 \rightarrow T_2 \rightarrow \ ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 1, we have $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in *SG(H)* and thus the algorithm produces only serializable histories.■

### 3.3.5. Discussion

In this section, we discuss the advantages and disadvantages of OCC-TI, and

consider ways to include transaction deadline information in making conflict resolution decisions.

The algorithm keeps all the advantages of OCC-FV. They include high degree of concurrency, freedom from deadlock, early detection and resolution of conflicts (compared to backward validation-based optimistic algorithm) resulting in both less wasted resources and earlier restarts. All of these contribute to increasing the chances of meeting transaction deadlines. Also, like the OCC-FV algorithm, OCC-TI can avoid the "wasted restart" phenomenon of locking-based algorithms because it allows only committed transactions to restart others. Furthermore, the ability of OCC-TI to avoid unnecessary restarts is expected to provide a performance gain over the OCC-FV algorithm.

However, this expected performance gain does not come for free. The main cost for this benefit is the management of timestamp intervals for active transactions and timestamps for data objects. This management can be done efficiently by using a transaction table and a data object table. The transaction table contains information of active transactions, including the read set and write set, and the timestamp interval of every active transaction. The information that is recorded in the data object table includes for each data object, *D*, *WTS(D)*, *RTS(D)*, the list of transactions holding locks on *D*, and the waiting list of incompatible lock requests on *D*.

Another important point to note here is that the degree of performance gain due to avoiding unnecessary restarts is dependent on the probability that a data object read is updated. When this *write probability* is low, that is, a write-read conflict rarely leads to a write-write conflict, the performance advantage can be significant. However, if the write probability is high, that is, a backward ordering for write-read conflict is almost always followed by a forward ordering for write-write conflict, the cost for timestamp interval management can overwhelm the benefit of reduced number of restarts.

Finally, it should be noted that OCC-TI currently does not use any transaction deadline information to make decisions for conflict resolution. The incorporation of timing information into the algorithm to improve timeliness level is a problem to be addressed. One method to do this was studied in [Lee93]. In this method, at the validation of a transaction, the set of active transactions is divided into two groups: reconcilably conflicting set and irreconcilably conflicting set. Conflict resolution between the validating transaction and the active transactions in the irreconcilable set is done by wait control-type priority-based mechanism studied in [Hari90b]. Then the timestamp interval adjustment process for the reconcilable set follows only when the validating transaction is allowed to commit by the priority-based conflict resolution.

We are aware of the limits of the wait control-type conflict resolution mechanism using transaction priority. Study in [Hari90b] reported that optimistic algorithms based on this mechanism show significant performance gains over priority-incognizant algorithm, OCC-FV, only under a limited operating condition, that is, low levels of data contention. Study in [Hari90b] also demonstrated that adding deadline information to optimistic algorithms is a non-trivial problem.

### 3.4. Experiment Environment

For evaluating the performance of the concurrency control protocol presented in this chapter, we use the simulation system described in Chapter 2. We also use the parameters and their base values for the system resources and workload given with the system model in Chapter 2. We do not repeat the description of the system model and parameters here.

It has been shown that the policy dealing with tardy transactions has a significant impact on the relative performance of the concurrency control algorithms in real-time database systems [Hari90]. In the experiments in this study, we assume that transactions

arriving in the system have firm deadlines. Therefore, a transaction is discarded immediately after it misses its deadline.

The primary performance metric used is the percentage of transactions which miss their deadlines, referred to as *Miss Percentage*. Note that we are not particularly interested in the average tardy time of committed transactions, because it has meaning only for soft deadline applications. In fact, in our system with the tardy policy of discarding transactions as soon as they miss the deadlines, tardy time of a transaction is always zero. Also, we are not interested in average response time, because the response time of a transaction is not critical as long as it meets its deadline.

As before, the data collection in the experiments is based on the method of replication. For each experiment, we ran the simulation with the same parameter values for at least 10 different random number seeds. Each run continued until 1,000 transactions were executed. For each run, the statistics gathered during the first few seconds were discarded in order to let the system stabilize after initial transient condition. The statistical data reported in this study has 90% confidence intervals whose end points are within 10% of the point estimate. In the following graphs, we only plot the mean values of the performance metrics.

### 3.5. Experiments and Results

In this section, we present performance results from our experiments comparing concurrency control algorithms in a real-time database system. We compare three different concurrency control protocols: 2PL-HP which is basically a locking scheme, but resolves a data conflict between a lower priority lock holder and a higher priority lock requester by restarting the lower priority transaction [Abbo88], OCC-FV [Robi82, Hari90], and OCC-TI. Note again that OCC-FV and OCC-TI does not use transaction deadline information for data conflict resolution, while 2PL-HP does.

First, we examine the performance of concurrency control algorithms under the condition of limited resources. The values of parameters, *NumCPUs* and *NumDisks* are fixed two and four, respectively. Figure 3.1 shows *Miss Percentage* behavior of algorithms under different levels of system workload. System workload is controlled by the arrival rate of transactions in the system. In this experiment, the value of *WriteProb* is fixed at 0.25. From the graph, it is clear that for very low arrival rates under 10 transactions/second, there is not much difference for the three protocols. However, as the arrival rate increases, OCC-FV does progressively better than 2PL-HP, and OCC-TI does even better than OCC-FV.

One of the reasons for this performance difference is the difference in the number of restarts, *Restart Count*, incurred by each of the protocols, shown in Figure 3.2. As mentioned earlier, 2PL-HP suffers performance degradation caused by wasted restarts. That is, the immediate conflict resolution policy of 2PL-HP allows a transaction that will eventually miss its deadline and be discarded to restart (or block) other transactions. This performance degradation increases as the workload level increases, since the number of transactions that miss their deadlines and have to be discarded increases.

The delayed conflict resolution policy of optimistic algorithms helps them to avoid such wasted restarts. However, OCC-FV suffers performance degradation caused by unnecessary restarts described in Section 3.3. At the relatively low write probability of 0.25, the possibility of a backward ordering followed by a forward ordering is low, and many unnecessary restarts can be saved by OCC-TI protocol. This is shown clearly in Figure 3.2, where we observe a significant difference between the restart curves of OCC-FV and OCC-TI.

Figures 3.3 and 3.4 show similar graphs as Figures 3.1 and 3.2, i.e., *Miss Percentage* and *Restart Count* of the these protocols under different levels of system workload. In this case, however, the system operates at a higher level of data contention with the value of *WriteProb* fixed at 0.75. The performance difference between 2PL-HP

and OCC-FV becomes even bigger, since the number of wasted restarts in 2PL-HP tends to increase as data contention increases. However, OCC-TI does not show significant performance gains over OCC-FV. This is due to the fact that with the relatively high write probability of 0.75, not many restarts are made unnecessarily by OCC-FV, since most backward ordering is followed by forward ordering.

One point to note here is that the restart count of all the three protocols decreases after a certain workload. The reason for this decrease is that after that workload point, resource contention dominates data contention in discarding deadline-missing transactions.

Until now, the performance of the protocols was shown under a limited resource situation. In such situations, since resource contention dominates data contention quickly as system workload increases, the performance of the system is primarily determined by resource scheduling algorithms rather than concurrency control algorithms. In fact, the performance difference shown in Figures 3.1 and 3.3 may not be very striking.

To capture the performance difference of concurrency control algorithms without the effect of resource contention, we simulated an infinite resource situation, where there is no queueing for resources (CPUs and disks). The data contention is maintained relatively high with the value of *WriteProb* fixed at 0.5. Figures 3.5 and 3.6 show *Miss Percentage* and *Restart Count* of the three protocols. As we expected, the performance difference of the three becomes clearer. Note that since in this infinite resource situation, there is no resource contention, the restart counts of the three protocols ever increase as the system workload increases.

In addition to *Restart Count,* we measured average data queueing time, i.e., locking duration of each protocol. For the reasons explained in Section 2.2.4, optimistic protocols showed shorter locking duration than 2PL-HP.

### 3.6. Summary

In this chapter, we have presented a new optimistic concurrency control algorithm. The design of the algorithm was motivated by the recent study results in [Hari90, Hari90b] concluding that optimistic approach outperforms locking protocols in real-time database systems with the objective of minimizing the percentage of transactions missing deadline. We observed that the optimistic algorithms used in previous studies could incur restarts unnecessary to ensure data consistency. The new optimistic algorithm was designed to precisely adjust and record temporary serialization order among concurrently running transactions, and thereby to avoid such unnecessary restarts. To evaluate the effect of the unnecessary restarts, a quantitative study was carried out using a simulation system of RTDBS with three concurrency control algorithms: two-phase locking with high priority conflict resolution policy (2PL-HP), optimistic protocol with forward validation (OCC-FV) and the proposed optimistic algorithm (OCC-TI).

We showed that under the policy that discards tardy transactions from the system, the optimistic algorithms outperform 2PL-HP, and OCC-TI does better than OCC-FV among the optimistic algorithms. The performance difference between OCC-FV and OCC-TI becomes significant especially when the probability of a data object read being updated is low, which is true in most actual database systems. In conclusion, the factor of unnecessary restarts is not negligible in performance of optimistic concurrency control under both finite and infinite resource conditions, and the proposed optimistic algorithm is a promising candidate for basic concurrency control mechanisms for real-time database systems.

Like OCC-FV, the proposed optimistic does not use transaction deadline information in making decisions for data conflict resolution. We expect a better concurrency control algorithm by using an intelligent way of incorporating transaction deadline information into the basic mechanism. In the following chapter, we will address this problem.

Figure 3.1 Miss Percentage, Write Probability = 0.25
Finite Resource, Tardy Transactions Discarded

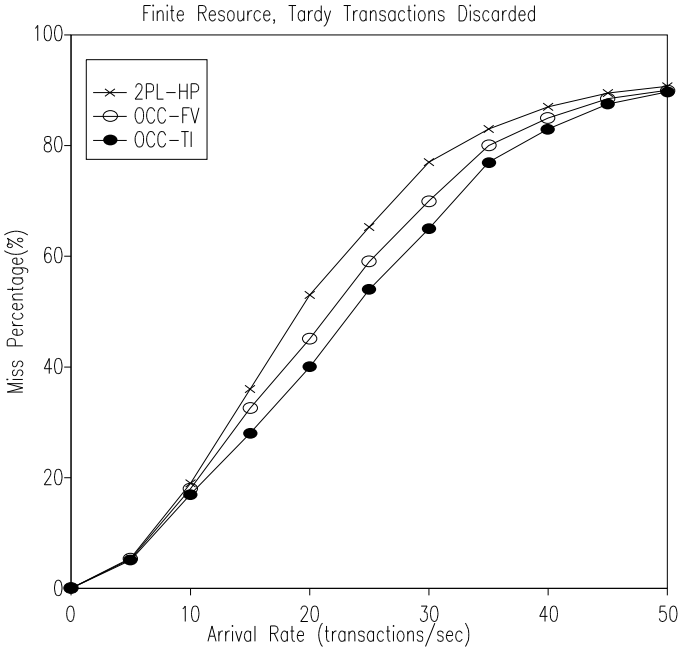



Figure 3.2 Restart Count, Write Probability = 0.25
Finite Resource, Tardy Transactions Discarded

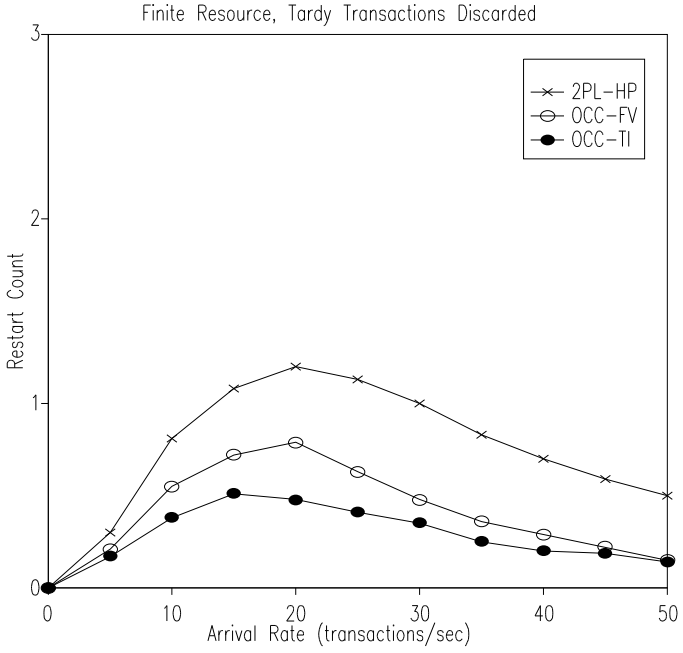


Figure 3.3 Miss Percentage, Write Probability = 0.75
Finite Resource, Tardy Transactions Discarded

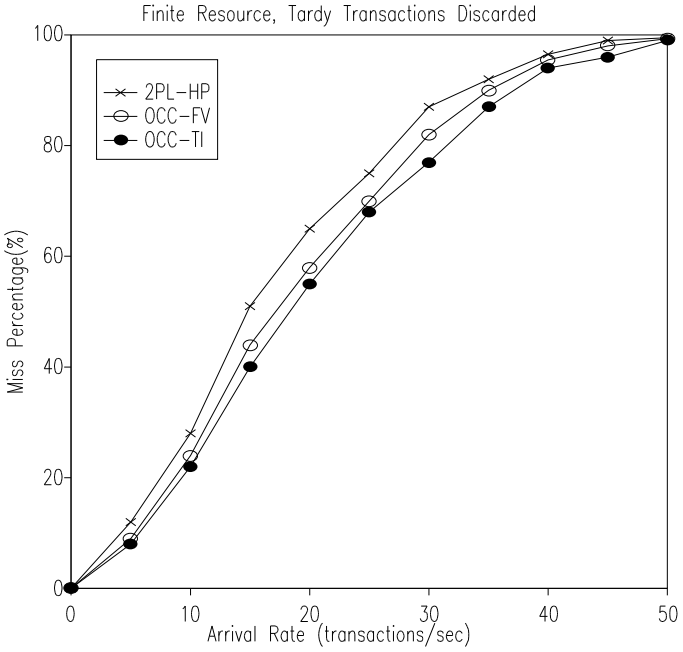



Figure 3.4 Restart Count, Write Probability = 0.75
Finite Resource, Tardy Transactions Discarded

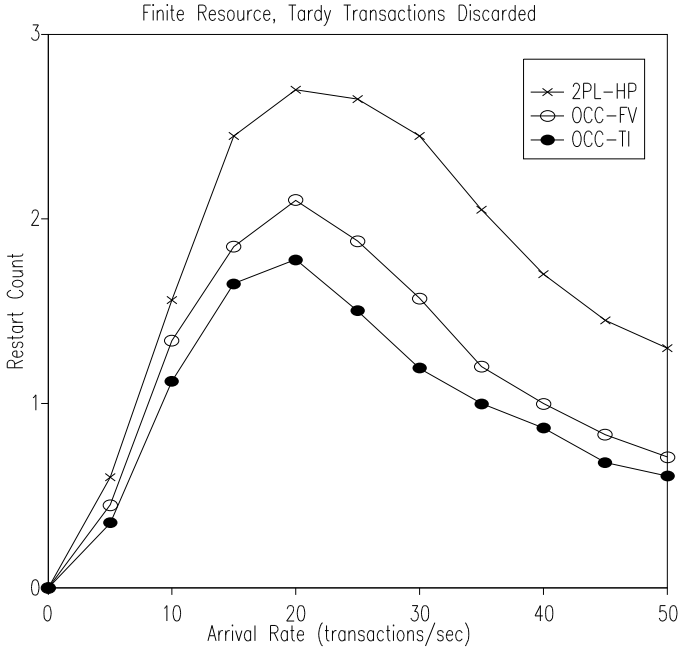


Figure 3.5 Miss Percentage, Write Probability = 0.5
Infinite Resource, Tardy Transactions Discarded

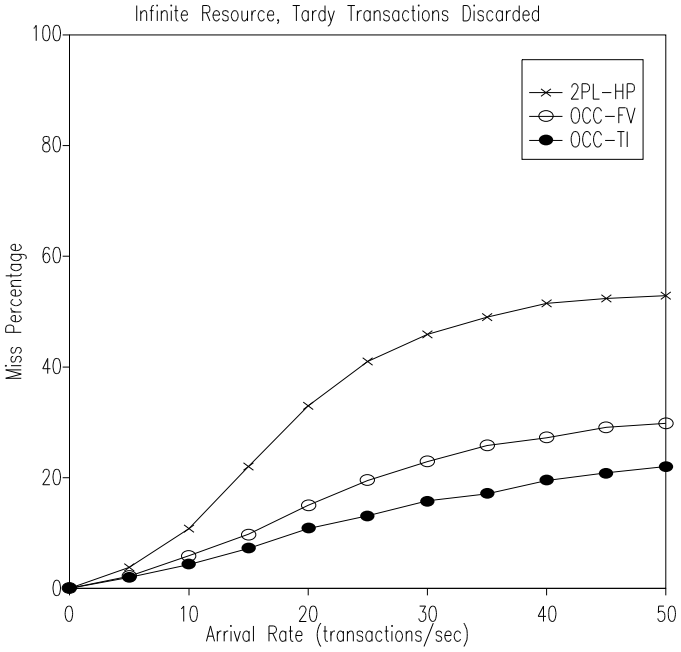
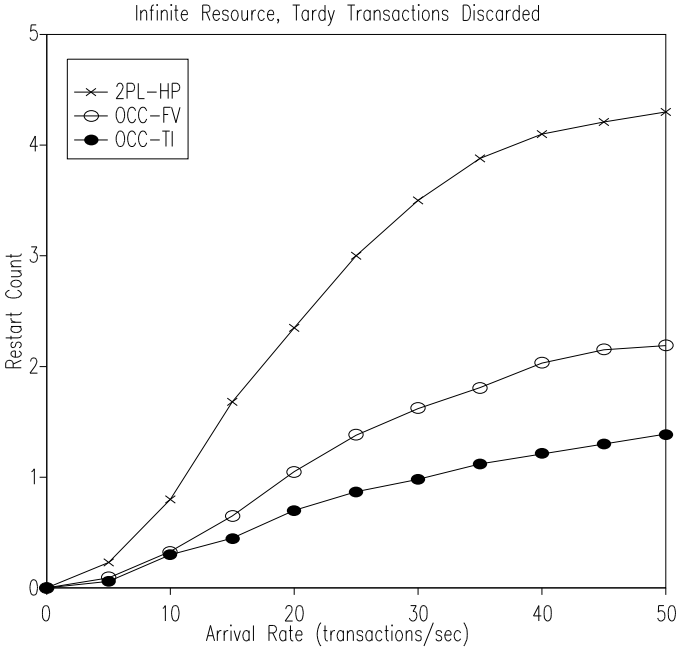



Figure 3.6 Restart Count, Write Probability = 0.5
Infinite Resource, Tardy Transactions Discarded



# 4. Design of Real-Time Optimistic Concurrency Control

## 4.1. Introduction

In the previous chapters, we have shown that optimistic concurrency control (OCC) is well-suited to real-time database systems, because the approach provides a relatively high degree of concurrency, and saves an important system overhead for data locking and deadlock detection. In particular, in the context of firm deadline systems where tardy transactions are aborted and permanently discarded from the system, optimistic protocols outperform locking-based algorithms under a wide range of operating conditions. The major reason for the performance difference between the two is that optimistic approach's policy of delayed conflict resolution in validation stage prevents the performance degradation caused by wasted restarts unlike locking-based protocols. Furthermore, several new ideas on OCC were devised to improve its performance. Forward validation technique for OCC saves resources and time by detecting and resolving conflicts early, and provides flexibility in conflict resolution which is essential for real-time concurrency control. In addition, the OCC-TI algorithm improves the performance further avoiding unnecessary restarts.

However, the optimistic algorithms we have studied until now did not use transaction deadline information, and ignored transaction priorities in resolving data conflicts. We compared conventional optimistic algorithms that do not use this information with a locking algorithm that does. The locking protocol, 2PL-HP used transaction deadline

information, which is encoded in the form of transaction priorities, to provide preferential treatment to urgent transactions. Conventional optimistic protocols, however, are indifferent to transaction priorities, and performance degradation is incurred when a low priority transaction conflicting with higher priority transactions unilaterally commits. In this study, we explicitly address the problem of using transaction deadline information to improve the performance of the optimistic algorithm and thus further decrease the number of transactions that miss the deadline.

The problem of making a concurrency control algorithm deadline-cognizant in resolving data contention, in general, does not have a straightforward solution, since the solution to this problem has to deal with the interaction of two incompatible orders of transactions: one determined by the urgency of transactions, i.e., *priority order*, and the other constructed by concurrency control algorithm to maintain data consistency, i.e., *serialization order*. Although it is possible to construct serialization order dynamically as close to the transaction priority order as possible, the work requires extra transaction blocking and/or restarts, and thus degrades performance, since serialization of data operations and priority-driven scheduling are basically incompatible [Son90c].

One most distinctive characteristic of transaction serialization order is that it is bound to past execution history with no flexibility. Thus, attempts to put data operations in priority order without considering serialization order may lead to aborting transactions which are partly executed, to eliminate the results of the transaction execution from history. Consider a simple solution to the problem of priority-cognizant conflict resolution which resolves data conflicts always in favor of the higher priority transaction. For example, this is the scheme used in 2PL-HP, i.e., High Priority scheme. This solution has several potential problems. First, giving preferential treatment to high priority transactions may result in an increase in the number of missed deadlines due to starvation of lower priority transactions. For example, this result can happen when helping a high priority transaction

to make its deadline causes several lower priority transactions to miss their deadlines. This problem becomes more serious in optimistic approach due to its policy of delayed conflict detection and resolution.

Second, resolving data conflicts in favor of a higher priority transaction does not always guarantee meeting the deadline of the transaction in the current state-of-the-art RTDBS techniques. For a variety reasons, the favored transaction with high priority may not meet its deadline. In such cases, the sacrifice made by low priority transactions for the high priority one missing its deadline is *wasted*. *Wasted sacrifice*s are similar to wasted restarts in 2PL-HP studied in previous studies [Hari90b, Lee93d]. In particular, the likelihood of wasted sacrifice increases when the system uses earliest deadline first or least slack first algorithm for transaction priority assignment. Both priority assignment policies have the weakness that they can assign the highest priority to a transaction that has already missed or is about to miss its deadline [Abbo88, Hari91].

Finally, there is another problem related with priority assignment policy. If the policy changes transaction priority dynamically and allows *priority reversal* between transactions [Hari90b], repeated conflicts between a pair of transactions may be resolved in some cases in favor of one transaction and in other cases in favor of the other transaction. This phenomenon is called *mutual sacrifice*. If this happens, the progress of both transactions is hindered and hence performance is degraded.

The objective of this work is to develop and evaluate policies for making optimistic protocols deadline-cognizant. In particular, we take into account the above problems in designing a deadline-sensitive optimistic algorithm.

Based on the argument that meeting timing constraints may be more important than data consistency in RTDBSs, attempts have been made to favor priority-driven scheduling by sacrificing data consistency temporarily to some degree [Lin89, Kuo92, Pu92].

Although it was shown that weaker consistency is acceptable in some applications [Garc83], there is no general-purpose consistency criterion proposed that is less stringent than serializability, and yet as obviously correct and implementable as serializability. Thus, in this study, we assume that data consistency is enforced using the correctness notion of serializability.

The remainder of this chapter is organized in the following fashion: Section 4.2 discusses the choice of the basic optimistic protocol into which a priority-based conflict resolution scheme will be incorporated. In Section 4.3, we review alternative policies for priority-based conflict resolution and discuss their potential strengths and weaknesses. Section 4.4 discusses our approaches to resolving conflicts taking transaction deadlines into account, and estimating the execution time of a restarted transaction in optimistic protocols. Section 4.5 describes our simulation model of RTDBS for experiments. In Section 4.6, the results of the simulation experiments are highlighted. Finally, Section 4.7 summarizes the main conclusions of this study.

### 4.2. The Choice of Optimistic Protocol

In this study, we have chosen the OCC-TI protocol as the basic optimistic mechanism to which a scheme to use transaction deadline information in resolving data contention will be added. The OCC-TI algorithm employs forward validation technique, in which validating a transaction is done against currently running transactions. Unlike the OCC-FV protocol, however, in the OCC-TI algorithm, temporary serialization order among concurrently running transactions is recorded and adjusted, if necessary, throughout the read and validation phase of transactions, and thereby unnecessary restarts can be avoided. In our earlier study, it was shown that the factor of unnecessary restarts on performance is not negligible and OCC-TI outperforms OCC-FV over a wide range of operating conditions.

In OCC-TI, most conflict detection and resolution is done at the validation phase of transactions between the validating transaction and concurrently running transactions. It is important to categorize currently running transactions in their read phase according to their relationship to the validating transaction. First, the set of currently running transactions is divided into two groups: the set of transactions conflicting with the validating transaction and the set of nonconflicting transactions. Then the set of conflicting transactions is further divided into two subsets: the set of *Reconcilably Conflicting* (RC) transactions and the set of *Irreconcilably Conflicting* (IC) transactions. The members of the RC set are transactions having only read-write conflicts with the validating transaction, while the IC set consists of transactions having both write-write and read-write conflicts with the validating transactions. The conflicts of RC transactions with the validating transaction can be serialized, i.e., resolved without any abortion using the dynamic serialization order adjustment mechanism of OCC-TI. However, irreconcilable conflicts are ones that are absolutely involved in nonserializable execution. To resolve such conflicts, it is unavoidable to restart either the validating or the running transaction involved in the conflict. Thus this is where deadline- or priority-cognizant conflict resolution should be used to improve real-time performance of the optimistic algorithm.

Figure 4.1 summarizes this categorization of transactions in the system. For the use in the priority-cognizant conflict resolution schemes introduced later, the set of IC transactions is further divided into two subsets by comparing the priorities of conflicting transactions with that of the validating transaction: the set of *Higher Priority* (HP) transactions and *Lower Priority* (LP) transactions. Unlike the basic OCC-TI algorithm, in the optimistic protocol in this study, when a transaction reaches its validation phase, it first checks the conflicts with currently running transactions, classifies them according to the type of their conflicts, and decides using this information if it needs to perform priority-cognizant conflict resolution. A priority-cognizant conflict resolution is required only when the set of IC transactions, especially HP transaction set, is not empty.

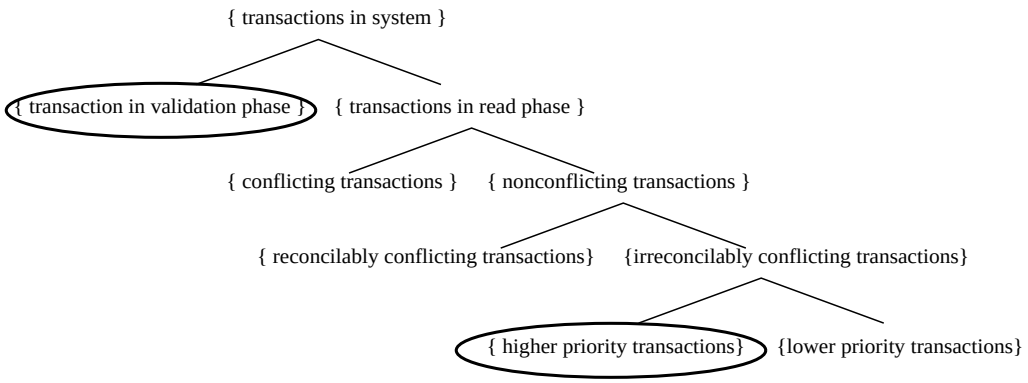



Figure 4.1 Categorization of Transactions in System

### 4.3. Deadline-Cognizant Conflict Resolution Policies

In this section, we consider alternative policies for deadline-cognizant conflict resolution and discuss, at an intuitive level, their potential strengths and weaknesses. We review conflict resolution policies studied in previous work [Hari90b, Huan91], and we reconstruct them from a different viewpoint, which provides a better insight into the problem.

The basic decision made by deadline-cognizant conflict resolution policies in optimistic approach is about the “sacrifice” of the validating transaction not to hinder the progress of more urgent, i.e., higher priority transaction(s) that are currently running. For this decision, we have to answer the following questions. How do we decide if a sacrifice is necessary? What information do we have to use to make the decision? When do we restart the validating transaction, if its sacrifice is decided? The difference of the policies presented in this section lies in the difference in their answers to these questions.

The decision for the sacrifice of a validating transaction should be made very carefully in optimistic concurrency control. Due to the delayed conflict resolution policy of optimistic approach, the sacrifice of a transaction makes it end up aborting after having paid

for most of its execution, i.e., completing the read phase, and hence the chance of the sacrificed transaction making its deadline is significantly decreased. That is, the cost of sacrifice in optimistic protocols may amount to more resources and longer time than that paid by restarts in priority-based locking protocol such as 2PL-HP. A sacrifice is useful only if running transaction(s) with higher priority can meet its deadline by the sacrifice. Wasted sacrifices in optimistic approach have particularly bad impact on performance, since they not only means that the favored higher priority transaction misses its deadline, but also results in a relatively high possibility that the sacrificed transaction misses deadline. Generally, in optimistic approach, it may be better to be conservative than aggressive in making sacrifices of validating transactions.

Below, to write each conflict resolution scheme in pseudo code, we use the following notations. $T_i$ denotes the validating transaction, and $T_j$ ($j = 1, 2,..., n, j \neq i$) denotes irreconcilably conflicting transactions. Also, $l$ and $m$ denote the number of HP and LP transactions in IC set, respectively. Note that the sum of $l$ and $m$ equals to $n$.

### 4.3.1. No Sacrifice

The first policy is extremely conservative in that it simply does not allow any sacrifice of validating transaction. In this scheme, the validating transaction immediately restarts other running transactions that irreconcilably conflict with it. Since there is no need to check for conflicts with already committed transactions, a transaction which has reached the validation stage is guaranteed to commit. This scheme can be described in a pseudo code as follows:

```
validate(T_i);
{
    restart(T_j), j = 1, 2,..., n;
    commit(T_i);
}
```

This policy does not use any transaction priority or deadline information, and hence leaves the optimistic protocol to be the same as original OCC-TI protocol. As mentioned earlier, under this scheme, a higher priority transaction may need to restart due to a committing transaction with a lower priority. However, the indifference to transaction priority of this policy protects it from problems related to priority-driven scheduling such as starvation of low priority transactions, wasted restarts, mutual sacrifice and priority inversion, and saves cost related to priority assignment and management.

Surprisingly, previous studies [Hari90, Hari90b, Lee93] reported that the performance of the optimistic algorithm with No Sacrifice policy is relatively good in general. It outperforms priority-cognizant locking-based protocol, 2PL-HP under a wide range of resource availability and system workload level, and is better than optimistic protocols with other deadline-cognizant policies under certain operating conditions.

### 4.3.2. Always Sacrifice

This policy represents the other end of the spectrum, i.e., it is extremely aggressive in making sacrifices. In this scheme, when a transaction reaches its validation phase, it is aborted if at least one conflicting transaction has higher priority than the validating one; otherwise it commits and all the conflicting transactions restart immediately. The pseudo code of this scheme is given as follows:

```
validate(T_i);
{
    if (l > 0)
    then restart(T_i);                    /* sacrifice */
    else {
          restart(T_j), j = 1, 2,..., n;
          commit(T_i);
         }
}
```

This scheme is priority-cognizant and satisfies the goal of giving preferential treatment to high priority transactions. However, it suffers from problem of wasted sacrifice due to its aggressiveness in making sacrifices of validating transactions. In addition, under this scheme, mutual sacrifices can also occur when the priority assignment mechanism employed allows priority reversal among transactions. This scheme is similar to OPT-SACRIFICE studied in [Hari90b], which reported that the performance of OPT-SACRIFICE is poor in general due to those problems.

### 4.3.3. Conservative Sacrifice

A conflict resolution scheme that takes a similar form as Always Sacrifice, but shows completely different characteristics is Conservative Sacrifice. In this scheme, when a transaction reaches its validation phase, it restarts if its priority is less than that of all the conflicting transactions, i.e., all the IC transactions have higher priority; otherwise, it commits and all the conflicting transactions restart. This scheme is described in the following pseudo code:

```
validate(T_i);
{
    if (l = n)
    then restart(T_i);                    /* sacrifice */
    else {
        restart(T_j), j = 1, 2,..., n;
        commit(T_i);
        }
}
```

This scheme is conservative in making sacrifices, using a more strict condition for sacrifice. Its conservativeness reduces the possibility of the problems of wasted sacrifices and mutual sacrifices. However, the strictness of the condition, especially under high data

contention levels makes the scheme perform analogously to No Sacrifice. Hence, the goal of giving preferential treatment to high priority transactions may not be efficiently satisfied with this scheme.

### 4.3.4. Unavoidable Sacrifice

The purpose of this policy is to solve the problem of wasted sacrifices in Always Sacrifice scheme by making sacrifices on demand. For this, this scheme employs a priority wait mechanism.

In this scheme, when a transaction reaches its validation phase, if its priority is not the highest among the conflicting transactions, it is made to wait for the conflicting transactions with higher priority to complete and not allowed to commit immediately. If all the higher priority transactions are restarted by some other transactions or discarded due to missing of their deadline, then the waiting transaction can commit. However, the waiting transaction should restart if one of the higher priority transactions commits. In such cases, the sacrifice of validating transaction is unavoidable. Thus, this scheme gives the higher priority transactions a chance to make their deadline first and solve the problem of wasted sacrifices. The problem of mutual sacrifice is also removed because conflict resolution is first made by waiting rather than restarting transaction. The pseudo code of this policy can be given as follows:

```
validate (Ti);
{
    while (l > 0) {
        wait;
        if committed(Tj) then restart(Ti);          /* sacrifice */
    }
    restart(Tj), j = 1, 2,..., n;
    commit(Ti);
}
```

Although the priority waiting mechanism improves performance with preferential treatment for transactions and avoiding wasted restarts, it requires some price to be paid for the benefits. First, if a transaction finally commits after waiting for some time, it causes all its conflicting transactions with low priority to restart at a later point in time, hence decreasing the chance of these transactions meeting their deadlines. This problem becomes worse with the possibility of chained blocking, which may cause cascaded aborts. Second, the validating transaction may develop new conflicts during its waiting period, thus causing an increase in the number of conflicting transactions and leading to more restarts. This increase may be substantial when there are many concurrently executing transactions in the system and the data contention rate is high.

Unavoidable Sacrifice is similar to OPT-WAIT algorithm studied in [Hari90b], which reported that although OPT-WAIT performs better than priority-incognizant optimistic protocol under low contention conditions, its performance degrades significantly even below that of priority-incognizant one as contention increases.

### 4.3.5. Adaptive Sacrifice

The purpose of this scheme is to maximize the beneficial effects of priority wait mechanism of Unavoidable Sacrifice scheme, while reducing the effects of its drawbacks, i.e., later restarts and an increased number of conflicts. In this scheme, the state of transaction conflicts are dynamically monitored, and a validating transaction is made to wait as long as more than half of the transactions conflicting with it have higher priorities; otherwise it commits and all the conflicting transactions are restarted immediately as in conventional optimistic protocol. The pseudo code of the policy is given as follows:

```
validate(T_i);
{
    while (l > m) {
        wait;
```

```
            if committed(T_j) then restart(T_i);          /* sacrifice */
        }
        restart(T_j), j = 1, 2,..., n;
        commit(T_i);
    }
```

Note that the only difference between Unavoidable Sacrifice and this scheme is the condition used to decide if priority wait is necessary. This scheme was similar to WAIT-50 algorithm studied in [Hari90b]. The condition of this policy referred to as “50 percent rule” is more likely satisfied when the number of concurrently executing transactions in the system is small and data contention is low, and becomes more difficult to satisfy as the data contention increases. After all, this scheme adaptively behaves similarly to Unavoidable Sacrifice policy under low data contention and No Sacrifice when data contention becomes high. The work in [Hari90b] reported this behavior of WAIT-50 scheme quantitatively, i.e., it behaves almost like OPT-WAIT when contention in the system is low, and switches to behave like conventional optimistic algorithm as the contention passes beyond a certain point.

Adaptability of this scheme is desirable, because it enables the optimistic protocol to perform reasonably good over a wide range of operating conditions. However, the adaptability does not improve performance *per se*, and the performance of this scheme is limited by the basic schemes, i.e., No Sacrifice and Unavoidable Sacrifice. These schemes have weaknesses explained above and we believe that the real-time performance of the optimistic protocol can be further improved.

### 4.4. Our Approach

In this section, we present a new deadline-cognizant conflict resolution scheme that has the potential to overcome the weaknesses of the previous schemes and to improve the real-time performance of OCC-TI protocol. Also, we consider a method for executing

restarted transactions in a predictable way taking advantage of the properties of optimistic protocols such as knowledge of data and computation requirements and necessary data objects retained in memory at the validation phase of transactions.

### 4.4.1. Feasible Sacrifice

This policy is conservative in that it allows sacrifice of a validating transaction for the execution of running transaction with higher priority only if the sacrificed transaction can meet its deadline when restarted, that is, this scheme attempts to save the resources and time already invested for the validating transaction, and reduce the cases of deadline missing caused by wasted sacrifice. One assumption used by this policy is that the execution time of the sacrificed transaction when restarted can be known beforehand at its validation phase. We will explain later how this assumption can be justified taking advantage of the characteristics of optimistic approach. This scheme is different from the previous policies, because it uses information of transaction deadline directly instead of that encoded in the form of transaction priority. Deadline and execution time estimate of a transaction are used for its feasibility test to decide the necessity of sacrifice.

In this scheme, when a transaction reaches its validation phase, it is aborted if there are one or more conflicting transactions with higher priority, and the validating transaction can meet its deadline although it is restarted; otherwise the validating transaction commits and all the conflicting transactions restart immediately. The pseudo code of this scheme is given as follows:

```
validate(T_i);
{
    if ((l > 0) and ((D_i - C) > E_i))
    then restart(T_i);                          /* sacrifice */
    else {
        restart(T_j), j = 1, 2,..., n;
```

```
            commit(T_i);
            }
}
```

$D_i$ and $E_i$ denote the deadline and the execution time estimate of the validating transaction $T_i$, and $C$ denotes the current time at which the test is conducted. This scheme is deadline-cognizant and gives preferential treatment to more urgent transactions. Wasted sacrifices do exist under this scheme. However, they do not have negative impact on the real-time performance, since a sacrificed transaction can still meet its deadline, under the condition that its estimated execution time is accurate. Also, this scheme is free from the performance degradation caused by priority wait mechanism in Unavoidable Sacrifice.

The major weakness of this scheme is that its performance heavily depends on the accuracy of the estimated execution time of transactions. In general, it appears infeasible to estimate transaction execution time accurately due to the large variance between the average case and worst case execution time of a typical database transaction [Rama92, Stan88b]. However, we argue that when optimistic approach is used, there is a chance for a rather accurate estimation of transaction execution time at the validation phase of a transaction. At the validation time, all the data objects in the read set and write set of the transaction is fetched in the buffer, and we also have information on the execution behavior of the transaction such as its length, data objects in its read set and write set, and execution time of its previous runs. In the next section, we review the sources of unpredictability in typical database systems, and show how to resolve those sources of unpredictability and how to predict transaction execution time in this scheme.

A point to note is that like WAIT-50 algorithm, Feasible Sacrifice scheme can behave either No Sacrifice or Always Sacrifice scheme adaptively according to the value of $E_i$ in the feasibility test. If $E_i$ is overestimated, then the feasibility test is less likely to be satisfied. When the feasibility test fails, the scheme becomes No Sacrifice scheme, which

is conservative in sacrificing. On the other hand, if $E_i$ is underestimated, the likelihood of the feasibility test's success increases. When the feasibility test succeeds, Feasible Sacrifice scheme becomes the one on the other end of the sacrifice policy spectrum, i.e., Always Sacrifice. However, even in this case, its performance is better than Always Sacrifice, because the feasibility test filters wasted sacrifices that have antagonistic impact on performance.

### 4.4.2. Predictable Transaction Execution

The approach used to estimate the execution time of a restarted transaction in this study is as follows:

1. The data objects accessed by the transaction which were fetched into memory in the previous run are retained, and are not transferred to disk during its restart delay and the next run unless the transaction misses its deadline and is discarded.

2. The basic execution time estimate is calculated using the formula for transaction execution time derived below.

3. To reflect dynamic changes of contention in the system, the execution time estimate is adjusted iteratively using dynamic monitoring of the system status and tuning parameters.

To see how this approach work, let us first review sources of unpredictability in typical database systems [Rama92]:

- data dependence of transaction execution,
- data conflicts,
- resource conflicts, and
- dynamic paging and disk I/O.

First, because a transaction's execution path can change depending on the state of the data items it accesses, it may not be possible in general to predict the behavior of a transaction in advance and what the worst case execution time of a transaction is likely to be. Second, because a typical transaction dynamically acquires the data items it needs, concurrency control algorithm may force the transaction to wait until a data item is released by other transactions currently using it, or force the transaction to restart. The duration of data blocking time and the number of restarts that a transaction goes through cannot be known in advance. Third, similarly, a transaction may be forced to wait for resources, such as CPU and I/O devices, to become available for unbounded time. Finally, in the case of disk-resident database systems that use demand-paged memory management, delays can occur while accessing disks for fetching data. Such unpredictable delay can lead to a large variance in transaction execution time.

In order to see how these sources of unpredictability affect the execution time of transactions, let us consider the composition of a transaction's execution time, *E*. Basically, there are three major components of the time spent by a transaction inside the system, data waiting time ($T_{w\_data}$), CPU time ($T_{CPU}$), and disk time ($T_{disk}$). The data waiting time is the duration of a transaction to be blocked in a data queue until its data request is satisfied. The CPU time for a data consists of two subcomponents, the waiting time due to CPU contention ($T_{w\_CPU}$) and the actual CPU service time ($T_{s\_CPU}$). The disk time for a data access is also composed of a waiting time ($T_{w\_disk}$) and a service time ($T_{s\_disk}$). The execution time for a transaction is the product of the number of data access requests that it makes ($N_{data}$) and the sum of $T_{w\_data}$, $T_{CPU}$, and $T_{disk}$. Thus, we see that:

$$E = N_{data} * (T_{w\_data}+T_{w\_CPU}+T_{s\_CPU}+T_{w\_disk}+T_{s\_disk}).$$

Note that all the sources of unpredictability except for the first one, are included in this expression in the form of waiting times.

Now we discuss our method how to predict transaction execution time by using this formula in the context of the optimistic protocol with Feasible Sacrifice scheme. First, in our approach, the data fetched to memory in the previous execution of the sacrificed transaction's read phase are retained in memory to avoid transferring them between memory and disk back and forth in the re-execution. Thus, we can save disk waiting time ($T_{w\_disk}$) and disk service time ($T_{s\_disk}$) in the re-execution of the transaction.

Second, we assume that the data dependent portion of transactions are such that a transaction's execution path, and hence its data and computational requirements, are unlikely to change due to possible concurrent changes to the data by other transactions during the time between previous and subsequent execution. This property is known as *access invariance*, and was investigated in [Fran90, O'Nei92]. A transaction with this property will access the same set of data objects or at least related data objects which have been prefetched. Although the degree of access invariance depends on how transactions are written and how system is used, we can reasonably assume, in general, that there is a high degree of access invariance. Also, studies have been done on how to ensure access invariance and how to detect the violation of access invariance with help from compilers and by proper design and coding of transactions [Fran90, O'Nei92]. The access invariance property of transactions eliminates the variance in transaction execution time caused by data dependence of transaction execution. In addition, it guarantees that when a validating transaction is sacrificed, all the necessary data for the next execution is fetched in main memory.

Regarding the data waiting time, $T_{w\_data}$, we assume that in optimistic protocols, the time represents the delay caused by restarts of the transaction instead of data blocking time. We argue that the delay may not be taken into account in the transaction execution time used for the feasibility test of Feasible Sacrifice scheme, because the execution time estimate used in the test has to consider only the execution time of the next run. Aborting

a restarted transaction in its read phase by an urgent validating transaction (that cannot accommodate another execution delay) is unavoidable, and is not included in the transaction execution time estimate.

Therefore, the formula for the transaction execution time estimate, $E_i$, is simplified as:

$$E_i = N_{data} * (T_{w\_CPU} + T_{s\_CPU}).$$

Note that the problem of estimating a transaction execution time ends up being the problem of CPU scheduling as in real-time task scheduling. The problem of task scheduling in real-time systems has been studied extensively, and there are various algorithms proposed with their properties analyzed. Also, note that at the validation phase, we know the size of the transaction, i.e., $N_{data}$.

We add an elaboration upon the expression of $E_i$. It appears that when a transaction is sacrificed and restarted, it may be helpful to have a certain amount of restart delay to reduce the possibility of repeated restarts. Adding a restart delay to the execution of a transaction should improve performance especially at high levels of contention in the system. One can come up with various approaches for deciding the length of the restart delay, and especially an adaptive restart delay will work reasonably over a wide range of operating conditions.

Finally, if the system is dynamically changing, we need a method to reflect the changes of the system in the estimation of transaction execution time. For this, we introduce a $T_{w\_CPU}$ adjusting factor $\alpha$, which has a value confined to the range $[0, \beta]$ ($\beta > 1$), and will be multiplied to $T_{w\_CPU}$ in $E_i$ to decide the current $T_{w\_CPU}$ as follows:

$$E_i = N_{data} * (\alpha * T_{w\_CPU} + T_{s\_CPU}) + T_{delay}.$$

Now we describe one possible implementation of the dynamic tuning of $T_{w\_CPU}$, which is

based on the nearest neighbor algorithm. We can start by operating the system at $\alpha = 1$, which leads to the system operating with the average $T_{w\_CPU}$. Our strategy is to continuously explore the neighborhood values of $\alpha$ for improvement. To achieve this, we use a system monitor whose job is to periodically collect "missed deadlines" statistics, and decide on a new $\alpha$ (by incrementing or decrementing the current value by some amount $\delta$). If the new $\alpha$ does not cause a reduction in missed deadlines, the monitor will switch the direction of $\alpha$ updates.

This nearest neighbor algorithm is capable of locating the optimal $T_{w\_CPU}$ value with a discrepancy of the size of the step-wise update $\delta$. The size of $\delta$ should be decided small enough so that the resultant $T_{w\_CPU}$ can be close to the optimal value, and also it should be large enough so that the convergence of $T_{w\_CPU}$ is efficient.

### 4.5. Experimental Environment

For the evaluation of the performance of conflict resolution schemes presented in this chapter, we use our simulation system whose model was described in Chapter 2. Central to our simulation model for RTDBS is a single-site disk resident database system operating on shared-memory multiprocessors. Also, we use the same parameters for the system resources and workload, and their base values as given in Chapter 2.

In this study, we assume that our RTDBS model is a firm deadline system, where completing a transaction after its deadline has expired is of no value and may even be harmful. Thus, a transaction that has missed deadline is aborted and permanently discarded from the system. The system checks the eligibility of a transaction whenever it restarts, and whenever it is put into and comes out of a queue, to see if it has already missed its deadline.

To estimate transaction execution time that is used in the feasibility test of validating transactions, we use the formula of $E_i$ discussed in Section 4.4 and the following parameters values. For the values of $T_{w\_CPU}$ and $T_{s\_CPU}$, we use the average CPU waiting

time and service time obtained from simulation results. For the value of restart delay, $T_{delay}$, we use a constant value which is the average execution time of a transaction obtained from simulation results. Finally, in the experiments of this study, we assume that for each experiment point, the system parameters such as *ArriRate* and *WriteProb* are fixed, and thus we do not use the nearest neighbor algorithm to iteratively adjust the transaction execution time estimate.

As before, the data collection in the experiments is based on the method of replication. For each experiment, we ran the simulation with the same parameter values for at least 10 different random number seeds. Each run continued until 1,000 transactions were executed. For each run, the statistics gathered during the first few seconds were discarded in order to let the system stabilize after initial transient condition. The statistical data reported in this study has 90% confidence intervals whose end points are within 10% of the point estimate. In the following graphs, we only plot the mean values of the performance metrics. The primary performance metric used is the percentage of transactions which miss their deadlines, referred to as *Miss Percentage* as before.

### 4.6. Experiments and Results

For the following experiments, we will compare the performance of the conflict resolution policies discussed in Section 4.3, i.e., No Sacrifice, Always Sacrifice, Unavoidable Sacrifice, Adaptive Sacrifice, and Feasible Sacrifice scheme. Conservative Sacrifice scheme is not included because it behaves almost identically to No Sacrifice over the entire operating range.

#### 4.6.1. Experiment 1: Moderate Data Contention

We first evaluate the performance of the priority-cognizant conflict resolution schemes under a moderate level of data contention. Figure 4.2 shows Miss Percentage behavior under a relatively high resource contention level. For this model, the write

probability is fixed at 0.25, and the number of CPUs and the number of disks are fixed two and four, respectively. Figure 4.3 shows the result of similar experiment of a moderate level of data contention under infinite resources. To simulate an infinite resource situation, we eliminate queueing of data operations for resources and allow any request for CPU processing or disk access to be fulfilled immediately. For this experiment, the write probability is fixed at 0.5.

From this set of graphs, we can make the following observations. First, Always Sacrifice scheme performs significantly worse than all the other schemes, including No Sacrifice scheme which ignores the information of transaction priority, over the entire operating region and under both finite and infinite resource scenarios. The poor performance of this scheme is primarily due to the increased number of wasted sacrifices, as discussed in Section 4.3.

Second, Unavoidable Sacrifice scheme performs better than No Sacrifice at low arrival rates, i.e., low levels of contention. As the arrival rate increases, however, its performance steadily degrades. Under finite resources, as the contention in the system passes beyond a certain point, Unavoidable Sacrifice scheme behaves almost identically to No Sacrifice. This is because a heavy resource contention makes it difficult for low priority transaction to reach their validation phase, leading the priority wait mechanism of Unavoidable Sacrifice scheme to have little impact on performance. Under infinite resources, as the arrival rate increases, the priority wait mechanism causes an increase in the number of transactions in the system, resulting in a significant degraded performance.

Adaptive Sacrifice scheme performs reasonably good over the entire operating range. It behaves like Unavoidable Sacrifice at low contention levels, and switches to behave like No Sacrifice as the contention increases beyond a certain point. The 50 percent rule of this scheme gives it the adaptability which provides the benefit of priority wait mechanism, reducing the effects of its weaknesses, i.e., later restarts and an increased

number of conflicts under high contention conditions.

Finally, Feasible Sacrifice scheme provides the best overall performance. The feasibility test for deciding the sacrifice of validating transactions helps avoiding the performance degradation by wasted sacrifices without the drawbacks of the priority wait mechanism, thus improving the real-time performance of optimistic protocol significantly. The scheme for estimating transaction execution time is clearly competent. The performance of Feasible sacrifice scheme converges to that of No Sacrifice as the arrival rate increases. This is because under a heavy contention, it becomes more difficult for the feasibility test for a validating transaction to be satisfied, and the Feasible Sacrifice scheme behaves analogously to the No Sacrifice scheme.

### 4.6.2. Experiment 2: High Data Contention

In this experiment, we study the comparative performance of the priority-cognizant conflict resolution schemes under a high level of data contention. Keeping the other parameters the same as before, the write probability is set to 0.75 for the finite resource model, and is fixed at 1.0 for the infinite resource scenario. The results are shown in Figures 4.4 and 4.5. In the figures, Always Sacrifice scheme is not included to simplify the presentation because it invariably perform worse than all the other schemes.

From this set of figures, we can make the following observations. First, Unavoidable Sacrifice scheme suffers a substantial performance degradation and does worse than No Sacrifice over almost the entire operating range. The increased level of data contention caused by the high write probability, in combination with the effect of increased conflicts of priority wait mechanism, results in a drastic increase in the number of conflicts, which in turn degrades the real-time performance because more low priority transactions will be restarted at a later point in time after being blocked.

Adaptive Sacrifice scheme does not suffer significant performance degradation

under the high data contention level, although it also employs the priority wait mechanism. This is because its 50 percent rule ensures that it behaves like No Sacrifice scheme as the data contention increases.

Feasible Sacrifice performs constantly better than all the other schemes, because this scheme is free from the performance degradation of the priority wait mechanism caused by later restarts and increased number of conflicts under high levels of data contention. Also, the scheme for estimating transaction execution time works successfully for deciding the next execution time of validating transactions.

### 4.6.3. Experiment 3: Write Probability

In this experiment, the write probability is varied from 0.0 to 1.0, keeping the arrival rate constant. For the finite resource experiment, the arrival rate is kept constant at 20 transactions/second, while it is fixed at 40 transactions/second for the infinite resource situation. Figures 4.6 and 4.7 show how the alternative schemes behave under conditions of finite and infinite resources, respectively. These graphs clearly show that while Unavoidable Sacrifice performs well at low data conflicts levels, No Sacrifice does much better at high levels of data conflicts. We also observe that while Adaptive Sacrifice provides good performance, Feasible Sacrifice performs constantly better than Adaptive Sacrifice over the entire region of operating conditions.

## 4.7. Summary

In this study, we have addressed the problem of using transaction deadline information to decrease the number of transactions that miss deadline, and thus improve the performance of optimistic concurrency control algorithm. We reviewed policies for making optimistic protocols deadline-cognizant. We presented a new deadline-cognizant conflict resolution scheme, called Feasible Sacrifice, which gives precedence to urgent transactions, while reducing the performance degradation caused by wasted sacrifices

using a feasibility test of validating transactions. For the feasibility test, we also proposed an approach to predict execution time of restarted transactions executed using optimistic protocol. We showed that this scheme has a potential to overcome the weaknesses of the previously proposed priority-based conflict resolution schemes and to improve the real-time performance of optimistic protocols.

Using a simulation model of a RTDBS, we studied the performance of the Feasible Sacrifice scheme over a wide range of workload and operating conditions. It was shown to provide significant performance gains over conventional optimistic protocol which is not sensitive to transaction deadline information in resolving conflicts. It was also shown to perform constantly better than Adaptive Sacrifice scheme which uses a priority wait mechanism to avoid wasted sacrifices and a wait control mechanism to avoid performance degradation under high levels of contention in system. In summary, we conclude that the Feasible Sacrifice scheme constantly provides improved performance, utilizing transaction deadline information.

Figure 4.2 Miss Percentage, Write Probability = 0.25
Finite Resource

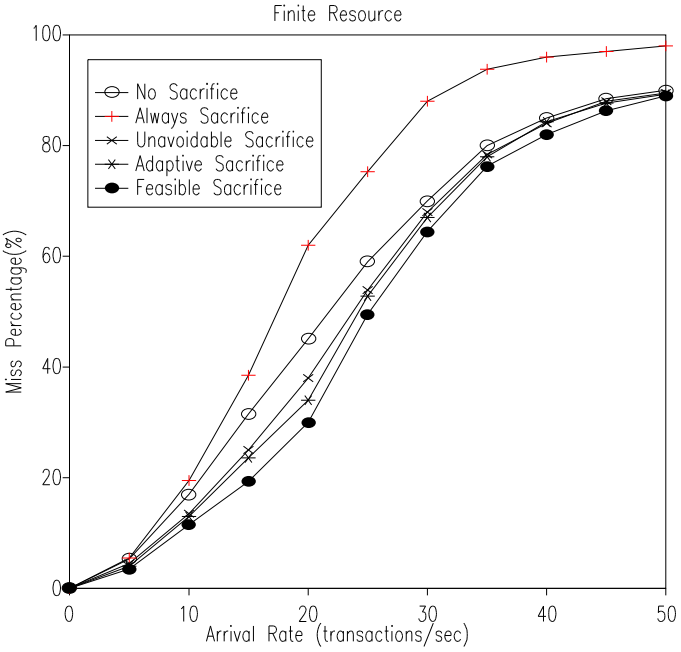



Figure 4.3 Miss Percentage, Write Probability = 0.5
Infinite Resource

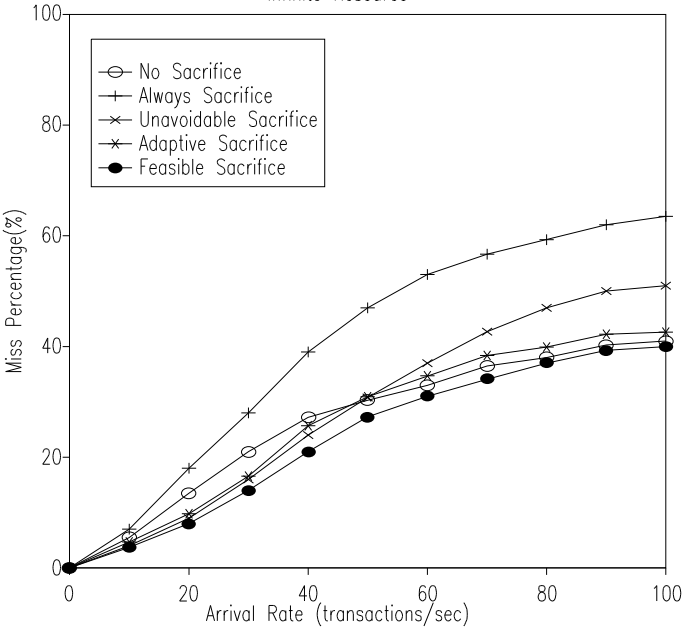


Figure 4.4 Miss Percentage, Write Probability = 0.75
Finite Resource

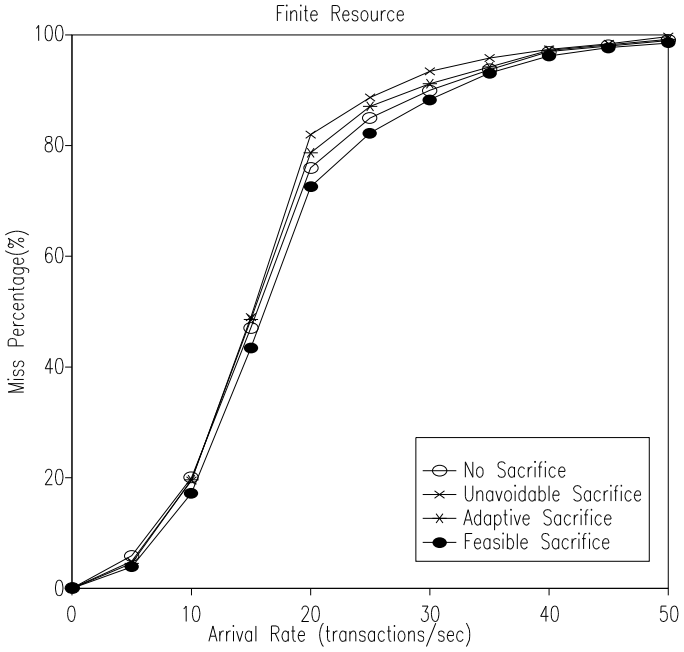



Figure 4.5 Miss Percentage, Write Probability = 1.0
Infinite Resource

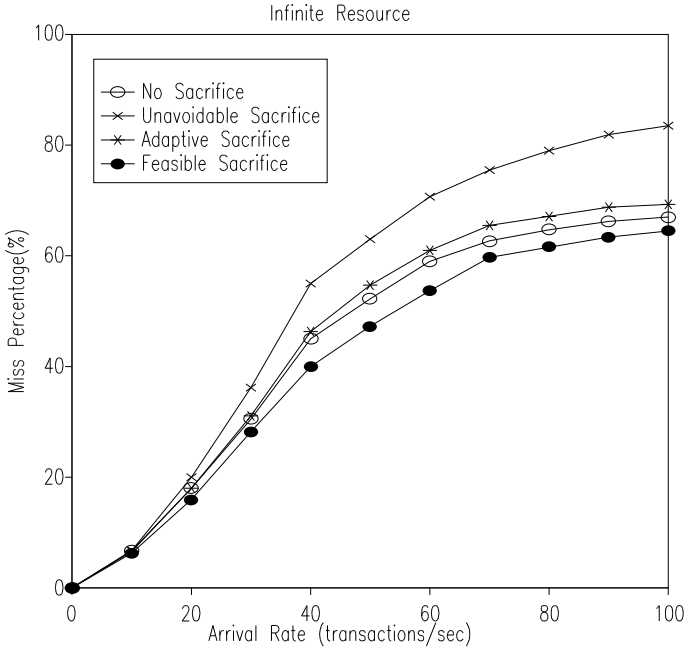


Figure 4.6 Miss Percentage, Arrival Rate = 20 transactions/sec
Finite Resource

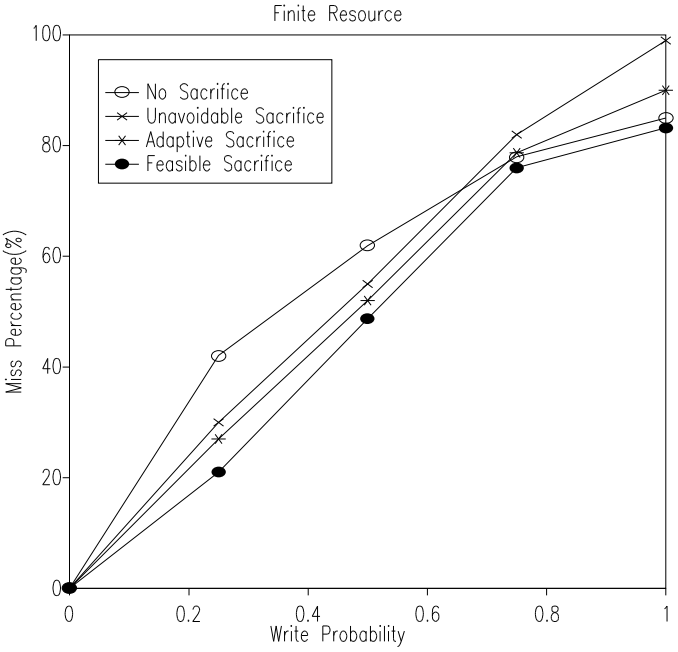
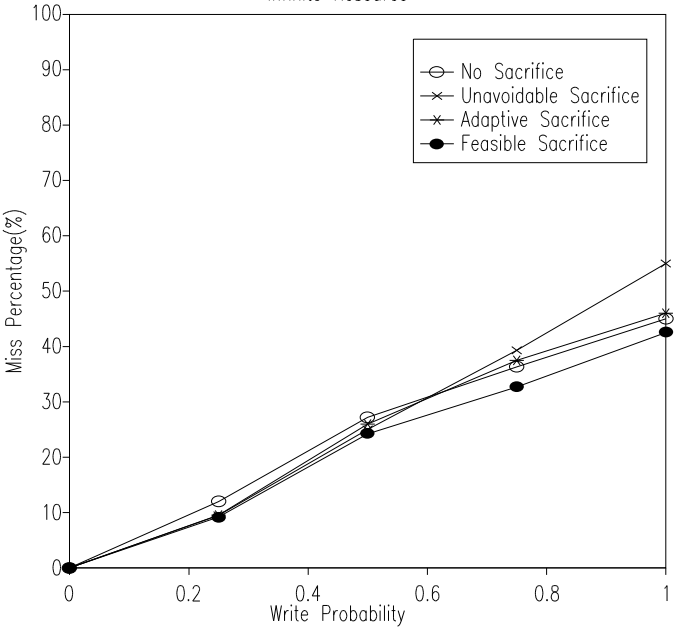



Figure 4.7 Miss Percentage, Arrival Rate = 40 transactions/sec
Infinite Resource



# 5. Semantic-Based Concurrency Control

## 5.1. Introduction

### 5.1.1. Background

Most work in *Real-Time Database Systems* (RTDBSs) has been done on the basis of the relational data model [Hou89, Son92]. Recently, however, the area of object-oriented databases has attracted the attention of researchers in RTDBSs [Lort92, Wolf92, Wolf93b, Wolf93c]. The motivation of the RTDBS researchers is, in general, to bring to bear many of the benefits of object-oriented database technology to solve the problems in managing the data in real-time systems.

The field of *Object-Oriented Database Management Systems* (OO DBMSs) has emerged in the mid-1980s, bringing together concepts from three research fields: traditional database management systems, object-oriented programming languages, and semantic data models [Hurs93]. The object-oriented data model was developed differently from the relational data model. Instead of having a simple definition and a strong mathematical foundation, the object-oriented data model originates from the tradition of semantic data modeling, which provides powerful abstractions for specifying database schemas than the relational data modeling. Also, it takes a view of data that is closely aligned with concepts of object-oriented programming languages such as, in Smalltalk terms, object, class, message, method, instance, encapsulation, and inheritance. (Brief definitions of these concepts will be given in Section 5.2.)

In general, the object-oriented data model gives users flexibility and extensibility in

handling complex data. It was pointed out that the object-oriented data model has a number of advantages over the relational data model [Gupt91, Kim89, Zdo90]. First, the object-oriented data model provides better modeling capability for representing real-world data because data representations in object-oriented databases do not have to be relational tables. In particular, an object-oriented data model can represent relatively easily unstructured data, non-homogeneous data, and complex data entities as complex objects, possibly composite objects.

Second, the object-oriented data model provides greater data abstraction. Because of object-oriented encapsulation, the representation of an object can be changed without affecting the rest of the database, and users do not have to know the structure, methods and constraints applicable to objects.

Third, OO DBMSs present an opportunity to provide more concurrency than traditional approaches allow [Weih88]. In the object-oriented approach, the database system knows more about the operations being performed. They are not simply read or write operations, but rather have more semantics. The concurrency of transactions executing on objects can be enhanced through the use of the semantic information about operations defined on the objects.

Finally, other advantages of the object-oriented data model related to software engineering such as better modularity, facilitation of software reuse, portability, support for generic programming, and a uniform design paradigm from specification to implementation, are described in literature [Gupt91, Zdo90]. Also, OO DBMSs can be free from the “impedance mismatch” between the languages for database access and application development [Zdo90].

The potential of the object-oriented data model provides a rich set of topics for database researchers. The research of OO DBMSs is still in an early stage compared to

relational systems. Some of the fundamental issues in OO DBMS research include the followings. First, research needs to be done for the definition of a universal object-oriented data model with a standard query language like SQL in relational DBMSs. Second, there are issues related with the implementation of complex objects and their operations, which is critical to the performance of an object-oriented database. Finally, more studies needs to be done for on the strategies for the following mechanisms for OO DBMSs; concurrency control, recovery, protection and locking mechanism, object identity, and object version control.

The powerful modeling capabilities of the object-oriented data model show promise for some of the “new” database applications, which require the representation of complex data elements, complex operations on them, and complex relationships among them. These applications include design databases, multimedia databases, and knowledge bases [Kort86]. These applications are different from traditional database applications such as business data processing, which relational DBMSs were originally designed for. Users in these new applications have found relational system inadequate in terms of flexibility, modeling power, and efficiency [Kort86]. The challenge from the OO DBMSs side is to realize these database applications by using the capabilities provided by the object-oriented data model. The need for supporting a real-time database system on the basis of the object-oriented database paradigm arises from many of these complex database applications, which require to manage data whose validity is constrained by time and to process transactions that have certain response time requirements.

### 5.1.2. Our Work

The design of a RTDBS based on the object paradigm presents a number of new and challenging problems: What is the best object-oriented model for real-time databases? How do we specify timing constraints in the model? What mechanisms are needed for handling constraints in various forms? How are transactions scheduled? How does the knowledge

about semantics of operations on objects affect concurrency control?

In this study, we concentrate on the concurrency control mechanism for object-oriented real-time database systems. In particular, we study techniques to improve the degree of concurrency of transactions executing on data objects through the use of semantic information about operations defined on objects. Such techniques are referred to as *semantic-based concurrency control* [Badr92].

The remainder of this chapter is organized in the following fashion: Section 5.2 reviews conventional object-oriented data models, and develops an object-oriented data model for RTDBSs by extending the conventional one. Section 5.3 discusses general design issues for semantic-based concurrency control mechanism for the object-oriented real-time database system. In Section 5.4, we review related work on semantic-based concurrency control both in conventional and real-time database systems. Section 5.5 describes our approach to semantic-based concurrency control for real-time database systems. Finally, Section 5.6 summarizes the main conclusions of this study.

## 5.2. An Object-Oriented Data Model for RTDBS

### 5.2.1. Basic Concepts

Object-oriented data models are often developed by incorporating the features and functionality of modern database management systems into the object-oriented programming environment [Hurs93, Zdo90]. At present no common definition for “the” object-oriented data model has emerged. Fortunately, however, there is some agreement on the essential features of object-oriented data model, as a guideline, from both sides of database management systems and object-oriented programming environments. In this section, we review the basic concepts of object-oriented programming languages and some of the essential features of database management systems.

An *object* is an abstract machine that contains information (data) on the object and

defines a protocol through which users of the object may interact. The protocol of an object is typically defined by a set of *messages* with typed *signatures*. A message can be sent to an object (called the *receiver*) to perform some action. As long as the types of the parameters match the signatures, the object is guaranteed to interpret this message and to carry out the action. A message is implemented by a *method*, which is a piece of code that accomplishes the desired effect and returns the result to the sender. A method has special privileges in that it can access the private data of the object. Since the only way to access an object's data is by sending a message to the object, and only the methods of that object can access its private memory, data within an object is protected from the outside world. This characteristic of objects is referred to as *encapsulation*.

A *class* is a template for its instances. Every object is an instance of some class. The class of an object defines the messages to which the object will respond and the way objects of this class are implemented. Classes are typically arranged in a directed graph, with the edges connecting superclasses to their subclasses, reflecting the relationship between the classes caused by *inheritance*. The edges express a compatibility relationship between the classes. Namely, a subclass inherits all the behavior defined by its superclass and may define additional behavior of its own. Instances of the subclass are completely substitutable in contexts that expect instances of the superclass.

As the essential features of modern database management systems, we consider data persistency, data sharing, relationships between data entities, and integrity constraints. We do not imply by any means that this list covers all the necessary features and functionality of modern DBMSs, but rather we believe that it represents a minimal set of features that make a DBMS distinctive from other systems. First, a DBMS provides a persistent and stable store. *Persistent data* are accessible past the end of the process that creates them. *Stable data* have some resiliency in the face of process failure, system failure, and media failure [Bern87]. DBMSs cope with such failures with a *recovery* mechanism

which uses secondary storage and/or duplicates storage devices.

Second, a DBMS permits database to be shared simultaneously by multiple users. *Concurrency control* mechanism of the DBMS prevents users from executing inconsistent actions on the database. Concurrency control is generally coupled with a *transaction* mechanism that provides *atomicity*. That is, a group of related operations for one user is executed in an all-or-nothing manner; thus, the database is never left partially updated, and partial changes are not visible to other users [Bern87].

Third, a DBMS can represent *relationships* between data entities, the relationships can be named, and the language can query those relationships. There are many ways in which the relationships might be captured: binary associations of entities, *n*-way associations, physical links from one entity to another, or logical links.

Finally, a DBMS can help to ensure the correctness and consistency of the data it contains, by enforcing integrity constraints, which are statements that must always be true for data items in the database. Some DBMSs (called *active database systems*) provide triggers to help in *constraint enforcement*: Actions can be initiated on access to particular data items, either to check that constraints hold, or to perform additional updates to bring the database to a state of consistency [Ditt91].

In the following sections, we present an object-oriented data model for RTDBSs based on the model having the features described in this section. We do not intend to develop a complete model, but rather extend the conventional data model to specify timing constraints and other information for processing real-time transactions. For the purpose of this study, it is sufficient to examine three components of the model; objects, relationships, and transactions. Objects represents data entities. Relationships are special type objects that represent associations among data objects. Transactions are executable entities that access the objects and relationships in the database.

We assume that the system has a database manager that takes care of most of the basic data management tasks including physical access to data objects, handling recovery from failure and deadlocks, and scheduling data operations on processors and disks. Note that concurrency control is not conducted by the database manager. It is a salient point of this model that concurrency control is incorporated into data objects, as will be explained later. The description of the model in this section is rather conceptual and given mostly in Smalltalk terms. Examples of language constructs implementing such a model are found in, e.g., [Agra89, Rich89], both of which borrow and extend the object definition facility of C++.

### 5.2.2. Object Model

An object consists of five components; a *unique name or identifier* of the object, a set of *attributes*, a set of *methods*, a set of *constraints*, and a *concurrency control* mechanism. Attributes characterize the state of the object. Methods of an object provide means by which transactions can access the attributes of the object. Constraints define the correctness of the object with respect to the system specification. Finally, an object has a mechanism for concurrency control, which defines allowable concurrent execution of each pair of method invocations in the object.

The structure of this object model with the five components is not very different from that of the traditional object-oriented data model. However, each component need to be extended to specify various forms of real-time constraints and incorporate information needed in scheduling data operations to meet their timing constraints. In real-time databases, some real-time constraints come from temporal consistency requirements and some from requirements imposed on system reaction time. The former is caused by data that has temporal validity and typically takes the form of periodicity requirement. The latter takes the form of deadline constraints imposed on aperiodic transactions.

First, each attribute of an object needs to contain at least two fields; its *value* and *absolute timestamp*. These will be used in the constraints component of the object to specify the logical consistency constraints such as integrity and range constraints, and the temporal consistency constraints involved in the objects, respectively.

Second, each method of an object carries not only the piece of code that performs the desired operations including read and/or write to the object's attributes, but also timing information such as the worst-case execution time of the method, and a set of temporal scopes, each of which defines absolute timing constraints on part of the method's execution. A *temporal scope* is usually represented by an absolute earliest start time, an absolute latest start time, and an absolute latest complete time (deadline) [Lee85]. This timing information is determined by the timing constraints imposed on the transaction that invokes the method. A method invocation also inherits a priority from the transaction that invokes it. Since each attribute of an object has fields representing value and timestamp, read and write operations on attributes must read and write the timestamp field as well as the value.

Third, each constraint of an object is defined by a pair of *predicate* and an *enforcement rule*. A predicate signifies logical and temporal constraints, such as the range of legal values for an attribute, the time at which an attribute loses its temporal validity, the start time or completion time of an data operation or method invocation. These constraints are specified using the value and timestamp fields of attributes. When a constraint's predicate becomes false by a method invocation, the corresponding enforcement rule is invoked, aborts the method invocation, and restore the object to its original state. Enforcement rules take the form of methods, carrying timing information like any other methods.

Finally, the concurrency control mechanism of an object is composed of a *compatibility relation* of its methods, which defines allowable concurrent execution of

method invocations in the object, and a concurrency control algorithm, which satisfies both logical consistency constraints and timing constraints simultaneously. Because concurrency control algorithm is defined for each object in this model, different object may use different concurrency control algorithm, taking the characteristics of the object and system conditions into account.

### 5.2.3. Relationship Model

One of the strengths of the object-oriented data model is that it allows complex and various relationships among data objects. Some of the relationships representable in the object-oriented data model include IS-A relationship, where an object is an instance of another object; HAS-SPECIALIZATION relationship, where an object has another object as a restricted instance; HAS-PART relationship, where an object contains another objects; HAS-SET relationship, where an object contains a set of other objects of all of which are same type [Zdo90].

There are a number of questions to be addressed for the effects of the relationships on the involved objects and the implementation of the relationships; How do the various relationships affect the local state, operations, compatibility relations, and constraints of the involved objects? How do we model the state of one objects by using the related object's state? How are the sets of methods of related objects related? How are constraints of an object related with those of the related objects? In our model, the effect of relationships among objects are captured explicitly by relationship objects. An explicit representation of relationships makes it easy to support inter-object constraints.

Because a relationship is an object, it may have the same five components of an object described above, i.e., the object identity, a set of attributes, a set of methods, a set of constraints, and a concurrency control mechanism. In addition to these components, a relationship has two more components; a set of objects participating in the relationship, and

a set of inter-object constraints placed on the objects in the participant set. Inter-object constraints are defined by a pair of predicate and enforcement rule, analogously to the object constraint. The only difference is that now the predicate of a relationship object is allowed to span multiple objects in the participant set, and that the set of enforcement rules is used to meet the inter-object constraints. The enforcement rules take the form of methods of the participant object. If any change to any object in the participant set causes a constraint in the inter-object constraint set to be violated, that constraint's corresponding enforcement rule is triggered.

### 5.2.4. Transaction Model

Transactions support updates and queries to the database. In a traditional database system, a transaction is defined as a sequence of data operations terminated by a commit or abort operation. In our model, a transaction is characterized by a set of method invocations on one or more objects and/or relationships in the database. A transaction carries its timing information including its worst-case execution time, temporal scope, and a priority assigned by the database manager to support its real-time scheduling. While concurrent execution of more than one transactions is allowed in the system, data consistency is maintained by concurrency control mechanism, which is implemented by the concurrency control component in each object in the system. We discuss the design and implementation of concurrency control mechanism in Sections 5.3 and 5.5.

### 5.2.5. An Example

Figure 1 illustrates an example of a database schema that models the world of a robotics application. (The figure omits parts of the schema that are not relevant to this example.) A robot is represented as a *robot* object that consists of its name, a set of attributes, a set of legal operations, a set of constraints, and a concurrency control mechanism. The *robot* object is related to two *arm* objects, *arm1* and *arm2*, representing

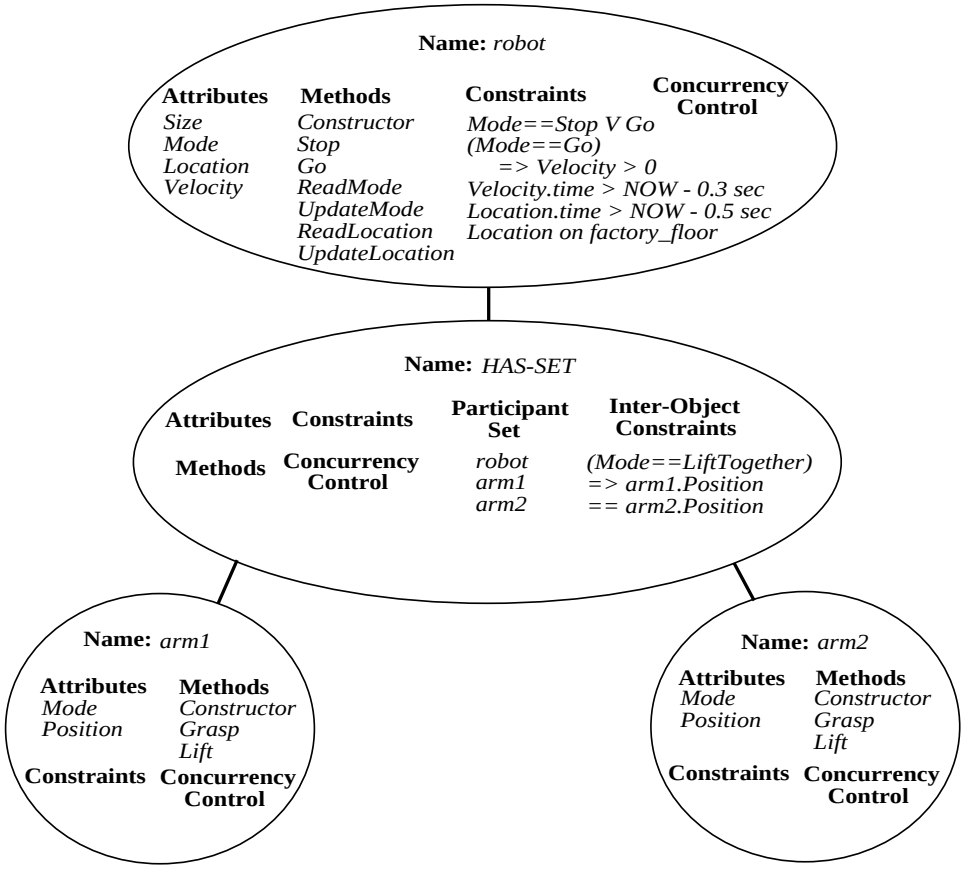



Figure 5.1 An Object-Oriented Real-Time Database Schema

the robot's arms. Each *arm* object has also its components. The relationship of the *robot* object with the two arm objects is a HAS-SET, since the *robot* object contains two *arm* objects of the same type.

The *robot* object has four attributes; *Size*, *Mode*, *Location*, and *Velocity*. Note that along with each attribute value is the timestamp of its last update. In some cases, as with the static *Size* attribute, the timestamp will always be the time at which the object was created. Other attributes, such as *Location* and *Velocity*, will be updated periodically while the robot is moving. To specify the temporal consistency of these attributes, timing constraints are placed on the object indicating when the attributes must be updated. For

example, the absolute temporal consistency constraint on the *Location* attribute has a predicate that states *Location.time > NOW - 0.5sec* while the robot's *Mode* is moving, and the corresponding enforcement rule marks the location invalid if the constraint is violated. *Robot* object also has integrity constraints, for instance, the location of the robot should be on the factory floor. The corresponding enforcement rule will deny any update that makes this predicate false. The object provides a number of legal methods that access the attributes of the object. For example, such method as *UpdateLocation* writes *Location.value*, and is invoked by transactions generated by the sensor signal processing subsystem. Users of this database may issue queries that use methods such as *ReadMode* and *ReadLocation* to find the current values of robot's *Mode* and *Location* attributes.

The objects, *arm1* and *arm2*, are composed in a similar way. In addition, there is a special object, HAS-SET relationship. It has a set of participating objects consisting of *robot*, *arm1*, and *arm2*. The HAS-SET relationship also has a set of inter-object constraints and corresponding enforcement rule that spans its participating objects. For example, if the robot lifts something with its two arms, i.e., the *Mode* of the two arms is set to *LiftTogether*, the constraint, *arm1.Position == arm2.Position* needs to be enforced.

### 5.2.6. Discussion

In general, the object-oriented data model provides greater opportunities for supporting semantic-based concurrency control than the traditional database model for a variety of reasons. First, the capability of including user-defined operations (methods) that can be of arbitrary complexity in data object representation provides greater semantic information about the operations that can be exploited for concurrency control.

Second, because of the encapsulation mechanism of the object-oriented model, the operations defined on a data object provide the only means to access the object's data. Thus, data contention can occur only among the operation invocations within the object.

This characteristic of the object-oriented data model provides greater flexibility for concurrency control in that it allows concurrency control specific to a data object. Furthermore, instead of imposing a general correctness criterion such as serializability on all data objects, in theory, concurrency can be maximized preserving logical data consistency specific to each object.

Finally, the improved capability of handling constraints in the object-oriented data model helps concurrency control mechanism. In particular, the capability of handling both logical and temporal consistency constraints in a uniform manner is beneficial for concurrency control in RTDBSs.

The characteristics of the object-oriented data model provide opportunities for using sematic information of operations on objects to increase concurrency allowed for transaction execution. In some sense, however, the performance improvement through the use of semantic information is required to be done in object-oriented databases. Because data objects in object-oriented databases are often large and complex, locking an entire data object for concurrency control (or transaction validation for accesses to an entire data object, in case of optimistic concurrency control) may be unnecessary and inefficient. If the semantics of the operations are taken into account in concurrency control, the necessity of locking the entire data object vanishes, and multiple operations that do not affect the same part of an object can potentially execute concurrently without violating the logical consistency constraints of the object.

### 5.3. Design Issues

Use of complete semantic information may help us attain higher concurrency. However, this expected performance gain does not come for free. In general, the use of semantic information in the object-oriented data model increases the complexity of the concurrency control mechanism for a variety of reasons. The additional complexity may

arise due to the complex domain of interpretation for various operations, and the possibility of using different concurrency control mechanism specific to an object. In an RTDBS where concurrency control mechanism has to consider not only logical but also temporal consistency constraints, the concurrency control mechanism may become even more complex.

For the design of a semantic-based concurrency control mechanism for an object-oriented database system, we need to consider the following three major issues: the determination of compatibility relation of operations defined on each object, the determination of a proper concurrency control algorithm for each object, and the support of the logical data consistency across objects in the system.

### 5.3.1. Compatibility Relation

One of essential tasks in implementing a semantic-based concurrency control is to interpret the semantic information about operations defined on each object to determine allowable interleaving of each pair of operation invocations in the object. For a complete use of semantic information, in general, this process has to consider various domains of semantic information, including the set of attributes affected by a particular operation invocation, and the parameters of operation invocations. In particular, in an RTDBS, temporal consistency constraints imposed on data objects and transactions also needs to be considered in the process. Furthermore, the process may become even more complex, when the system allows relaxation of serializability as the correctness criterion for logical data consistency. Due to the variety of domains of semantic information, the process of building compatibility relation tends to be application dependent. An *ad hoc* process of determining operation compatibilities often places a heavy burden on the user to define the semantics of operations.

Another question related to the issue of compatibility relation determination is

whether a compatibility relationship of an operation against other operations in an object is determined statically when the object is created or dynamically when the operation is requested to be executed on the object. In case of the static policy, a compatibility relation is defined for an object in the form of a table when the object is created. Alternatively, the compatibility relation can be defined by a compatibility function, where compatibility is expressed as a run-time function. In general, improved concurrency may be expected as the compatibility is determined dynamically, because we can exploit the information such as the current state of the object, and actual parameters of the operations as well as other information.

### 5.3.2. Concurrency Control Algorithms

Once the compatibility relation of an object is given, an easy way to implement a concurrency control mechanism of the object is to use a simple concurrency control protocol such as two-phase locking [Eswa76] which controls concurrent data operations by using the given compatibility relation to satisfy consistency constraints. However, because the encapsulation capability of object-oriented data model enables different concurrency control algorithms specific to a data object to be allowed, we may expect further improved performance by choosing for each object a concurrency control algorithm proper for its operating characteristics, resource availability and workload level in the system.

Various concurrency control algorithms basically differ in two aspects: the time when they detect conflicts and the way that they resolve conflicts. Locking and optimistic approach in their basic form represents the two extremes in terms of these two aspects. Locking detects conflicts as soon as they occur and resolves them using blocking. Optimistic scheme detects conflicts only at transaction commit time and resolves them using restarts.

The impact of these differences in concurrency control algorithms on performance

has been the major theme in the performance study of concurrency control in both conventional and real-time database systems. The results from these studies can be summarized as follows: In conventional database systems, locking algorithms that resolve data conflicts by blocking transactions outperforms restart-oriented algorithm in an environment where physical resources are limited. If resource utilizations are low enough so that a large amount of wasted resources can be tolerated, and there are large number of transactions available to execute, then a restart-oriented algorithm that allows a higher degree of concurrent execution is a better choice [Agra87]. In addition, the delayed conflict resolution of optimistic approach helps in making better decisions in conflict resolution, since more information about conflicting transactions is available at this later stage. On the other hand, the immediate conflict resolution policy of locking schemes may lead to useless restarts and blocking in RTDBSs due to its lack of information on conflicting transactions at the time of conflict resolution [Hari90].

### 5.3.3. Inter-Object Data Consistency

In an object-oriented database system, concurrency control mechanisms defined in objects have to resolve inter-object inconsistency as well as satisfy local data consistency within each object. Transactions, in an object-oriented database system, consist of a set of operation invocations on one or more objects. Thus, transactions may read and/or write inconsistent data across objects, when the serialization order among transactions locally determined in one object does not agree with one built in another object. Without any information about status of transactions executing across objects, local concurrency control mechanisms in individual objects have no way to recognize and resolve inter-object data inconsistency.

The problem of maintaining inter-object data consistency in an object-oriented database system is analogous to the problem of satisfying global data consistency in a distributed database system. Thus, local concurrency control schemes in an object-oriented

database system can be made to support inter-object data consistency with similar methods used for distributed database environments. For instance, the lock managers in locking protocols and the validation phase of transactions in optimistic methods, need to take into account the serialization order determined in other objects. There are several possible schemes for implementing such modifications, including the *single coordinator approach* in which the system maintains one single coordinator that resides in one single chosen object, and the *multiple coordinator approach* in which the coordinator function is distributed over several objects. Each coordinator knows the names of all the participating objects, and sends them messages to inform status of executing transactions.

With object-oriented data models, we may obtain opportunities to improve performance by choosing different local concurrency control mechanisms and data consistency correctness criteria specific to individual objects taking object characteristics into account. However, this benefit is mitigated by the problem of inter-object data inconsistency. With allowing object-specific concurrency control mechanisms, the problem of inter-object inconsistency becomes more difficult, as the situation becomes analogous to one in a distributed database system consisting of heterogeneous database systems in different sites. When various correctness criteria are allowed for different objects, locally consistent data within an object using one correctness criterion may be inconsistent externally with data objects using different correctness criteria. In addition, if various concurrency control protocols are allowed for different objects, we need a mechanism to understand and maintain serialization order enforced by different local concurrency control methods in different objects to support inter-object data consistency.

### 5.4. Related Work

A number of studies in the area of conventional database systems have been done on enhancing the degree of concurrency of transactions executing on data objects by using the semantics of data operations. However, the techniques presented in these studies

typically only support limited forms of logical consistency; they do not address temporal consistency. There are two types of sematic-based concurrency control techniques. With *transaction-based semantic concurrency control*, the user explicitly specifies which transactions may be interleaved to preserve data consistency; some of these techniques are presented in [Garc83, Lync83]. *Object-based semantic concurrency control* techniques allow the users to specify allowable interleaving of object operations to preserve logical consistency of data; some of these are presented in [Badr88, Badr92, Schw84, Weih88, Wolf93].

The studies in [Wolf92, Wolf93b, Wolf93c] focused on semantic-based concurrency control mechanism for an object-oriented real-time database system. They presented a method for dynamically determining compatibility relations of operations by using a run-time compatibility function for each pair of operation invocations. In the process, the compatibility functions take into account a broad domain of semantic information that may affect the consistency constraints of the database system including the set of object attributes affected by a particular operation invocation, the parameter values of operation invocations, and the maximum number of invocations of a particular operation.

In this work, the process of building compatibility functions also considers temporal consistency constraints as well as logical consistency constraints. Each compatibility function includes predicates involving timing constraints imposed on data attributes and transactions, and offers the capability of allowing to trade-off logical consistency for temporal consistency. The work also allows a wide range of correctness criteria for logical data consistency, including *epsilon-serializability* [Pu92] and *similarity-based correctness* [Kuo92], both of which are more relaxed criteria than serializability.

The approach taken in this work provides a comprehensive view over constraints in a real-time database system, and appears to have potentials for real-time concurrency

control, particularly by supporting a trade-off between the two requirements, logical and temporal consistency constraints. However, with this approach, there is no general method for systematically determining compatibility relation of operations due to the complexity in the interpretation of the two often semantically unrelated constraints. The process of building compatibility relations with such complexity will be a significant burden on the object designers to interpret the semantics of operations.

This work used a simple locking protocol uniformly for all objects in the system, in which locking compatibility is determined by a run-time compatibility function. One significant drawback of this work is that it did not account for the problem of inter-object inconsistency, although the problem may occur due to the variety of allowable correctness criteria in the system.

### 5.5. Our Approach

One of the objectives of this study is to present an approach to semantic-based concurrency control mechanism for a real-time database system, which overcomes the problems of the work in [Wolf93c]. Thus our design strategy is to alleviate the complexity associated with the design procedure of the mechanism, especially with the step of compatibility relation construction. By doing so, we expect to provide a semantic-based concurrency control mechanism easy to implement, and to provide a general method for systematically building compatibility relation of operations. In addition, in the design of a semantic-based concurrency control mechanism for RTDBSs, we hope to apply the results from previous research on concurrency control for real-time database systems.

To mitigate the complexity associated with semantic-based concurrency control mechanism for RTDBSs, we made a number of decisions. First, we decided to use only serializability as the correctness criterion for logical data consistency in the system. Although there are studies on relaxation of serializability that presented correctness criteria

such as epsilon-serializability and similarity-based correctness, at present there is no general purpose correctness criterion as easily understandable and implementable as serializability.

Second, we decided to use only one concurrency control protocol uniformly for all of the objects in the system. However, this decision does not fix any particular concurrency control algorithm for the system. The concurrency control algorithm may be either locking or optimistic approach. The choice of the concurrency control algorithm will be decided on the basis of resource availability and workload level of the system. The first two decisions alleviate the complexity of designing a sematic-based concurrency control mechanism significantly, and especially eliminate the problem of inter-object inconsistency.

Finally, we decided not to consider temporal consistency constraints in the process of building compatibility relation of operations, to simplify and systematically perform the process. Timing constraints imposed on data objects and transactions are taken into account only in concurrency control protocols, which use priority and deadline information in the resolution of data conflicts, i.e., resolves data conflicts in favor of more urgent operation with higher priority. A number of studies on concurrency control for RTDBSs presented various algorithms with such capability and evaluated their performance [Abbo88, Abbo89, Hari90, Hari90b, Huan91, Lee93c]. This approach will significantly lessen the complexity of the process of compatibility relation construction. Also, with this approach, we can take advantage of the results from previous research on both semantic-based concurrency control for conventional database system and concurrency control algorithms for real-time database systems. One disadvantage of this approach is that we cannot maneuver the trade-off between logical and temporal consistency constraints in the process of determining operation compatibilities.

In the following, we briefly describe a technique for determining a compatibility relation of operations in an objects, and two representative concurrency control algorithms

for RTDBSs, one is based on locking and the other on optimistic approach.

### 5.5.1. Compatibility Relation by Affected-Set

This technique considers only logical consistency constraints of objects, and uses the concept of an *affected-set* originally presented in [Badr88]. When an operation is invoked, the set of attributes affected by the particular invocation is computed. The affected-set must be computed for each invocation of an operation because the affected attributes may depend on the arguments of the operation. The attributes in the affected-set for an operation include all attributes that are read by or written by the operation as well as any attributes affected by enforcement rules that may be triggered by the operation. We group subsets of the affected-set that contain only attributes read by the operation and only attributes written by the operation. These subsets are referred to as *read-affected-set* and *write-affected-set*, respectively. By checking the intersection of their read-affected-sets and write-affected-sets, the compatibility relation between a pair of operation invocations, $OI_1$, and $OI_2$ in an object is determined. In general, $OI_1$ and $OI_2$ are incompatible if at least one of read-affected-set($OI_1$) $\cap$ write-affected-set($OI_2$), write-affected-set($OI_1$) $\cap$ read-affected-set($OI_2$), and write-affected-set($OI_1$) $\cap$ write-affected-set($OI_2$) is not empty.

With this technique, the designer of an object type only needs to specify the semantics of operations; their compatibilities are systematically determined from these specifications. Another advantage of this approach is that the compatibility of operations may be determined dynamically when the operation is requested to be executed on an object [Badr88].

### 5.5.2. Real-Time Concurrency Control Algorithms

In the classical two-phase locking (2PL) protocol [Eswa76], transactions set read locks on objects that they read, and these locks are later upgraded to write locks for the data objects that are updated. If a lock requested is denied, the requesting transaction is blocked

until the lock is released. Read locks can be shared, while write locks are exclusive. The only difference in our semantic-based concurrency control mechanism is that operations, not transactions, invoked by transactions set locks on attributes in their affected-sets. An operation sets read locks on attributes in its read-affected-set, and write locks on attributes in its write-affected-set. It is important to note that the locks set by an operation are not released until the transaction invoked the operation is completed (or restarted), to ensure serializable execution of transactions within objects.

For real-time database systems, two-phase locking needs to be augmented with a priority-based conflict resolution scheme to ensure that higher priority transactions are not delayed by lower priority transactions. In the *High Priority* scheme [Abbo88], all data conflicts are resolved in favor of the transaction with the higher priority. When a transaction requests a lock on an object held by other transactions in a conflicting lock mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the lock. In addition, a new read lock requester can join a group of read lock holders only if its priority is higher than that of all waiting write lock operations. This protocol is referred to as *2PL-HP* [Abbo88]. Although *2PL-HP* loses some of the basic 2PL algorithm's blocking factor due to the partially restart-based nature of the *High Priority* scheme, it was shown to outperform optimistic schemes under an environment where physical resources are limited [Lee93d].

The locking protocol automatically supports the logical data consistency across objects, under the condition that no object is replicated in the system. The local serialization orders on individual objects can be extended to a unique global serialization order without introducing any cycles due to the *High Priority* conflict resolution policy.

In optimistic concurrency control (OCC), transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. Previous

studies on concurrency control for real-time database systems have shown that OCC is well-suited to RTDBSs because of its provision of high degree of concurrency, and its policy of validation stage conflict resolution. In studies [Hari90, Lee93d], it was shown that under the condition that tardy transactions are discarded from the system, OCC outperforms locking over a wide range of system workload and resource availability.

Among a number of OCC-based protocols designed for RTDBSs, it was shown that *OCC-TI* algorithm augmented with *Feasible Sacrifice* scheme for deadline-sensitive conflict resolution outperforms other algorithms currently known over a wide range of operating conditions [Lee93c]. This algorithm uses *forward validation* technique [Haer84], because it provides flexibility required for priority-based conflict resolution unlike backward validation, and reduces the wastage of resources and time by detecting and resolving data conflicts early. Also, unlike other OCC algorithms, *OCC-TI* can prevent restarts unnecessary to ensure data consistency by recording and adjusting serialization order dynamically [Lee93b]. Finally, the *Feasible Sacrifice* scheme for deadline-sensitive conflict resolution, improves performance by reducing the number of missed deadlines due to wasted sacrifices of validating transactions, while giving precedence to more urgent transactions.

Although no study previously done on semantic-based concurrency control has employed OCC, we argue that an OCC-based protocol is promising for semantic-based concurrency control for RTDBSs, as results from studies on real-time concurrency control in [Hari90, Lee93d] show. Once affected-sets of operations are given, the validation phase of an OCC algorithm may be easily implemented by using locks. We use two lock modes; *Read-Phase-Lock* (*R-Lock*) and *Validation-Phase-Lock* (*V-Lock*). An *R-lock* on an attribute is further classified as either read lock or write lock, depending on the membership of the attribute with respect to the affected-set of the operation. An *R-Lock* is set whenever a transaction accesses a data object in its read phase, and an *R-Lock* for attributes in write-

affected-set is upgraded to a *V-Lock* when the transaction enters its validation phase. The validation process is carried out by checking the lock compatibility. In the forward validation, read locks and *V-Locks* are incompatible [Haer84].

To support inter-object data consistency, we can incorporate into an OCC algorithm a *distributed validation scheme*, with which a transaction validates at all objects that are involved in its read phase to ensure that validation requests are processed in the same order at all objects. There are a number of possible approaches to implementing such distributed validation schemes, including the use of multi-cast messages or timestamps [Thom90].

### 5.6. Summary

In this chapter, first, we examined issues related with designing of semantic-based concurrency control for real-time database systems defined on the basis of an object-oriented data model. Then we presented a semantic-based concurrency control mechanism for a real-time database system. The mechanism is based on a technique for determining operation compatibilities by using the concept of *affected-set* of operations. With this technique, since the compatibilities of operations are systematically determined from their specification, the designer of an object type only needs to specify the semantics of operations. Also, the semantic-based concurrency control mechanism employs concurrency control algorithms that gives precedence to more urgent operations through the use of priority- and/or deadline-sensitive conflict resolution schemes. We showed how optimistic concurrency control algorithms can be used in the semantic-based mechanism, and argued that optimistic approach is promising for semantic-based concurrency control for real-time database systems.

Our work on semantic-based concurrency control is different from the traditional work in the area in that it considers timing constraints imposed on data objects and transactions as well as logical consistency constraints of the database. Also, unlike the work

in [Wolf93c], our approach alleviates the complexity associated with a semantic-based concurrency control mechanism for real-time database systems. The mechanism presented is application independent and easy to implement.

In this study, we also reviewed the essential features of an object-oriented data model for a real-time database system. We showed that the characteristics of the object-oriented data model provide opportunities for using semantic information about operations, and flexibility for supporting various concurrency control mechanisms and correctness criteria for logical consistency specific to objects. We examined a number of issues related to the design of semantic-based concurrency control for an object-oriented RTDBS.

# 6. Conclusions

## 6.1. Summary of the Work

Concurrency control that schedules data access operations should be considered separately from hardware resource scheduling because of the uniqueness of data as a resource. First, data is a logical resource, and so more than one transactions may be concurrently scheduled to utilize a particular data object. Hardware resources such as CPUs and disks, however, are physically constrained to serve only a single request at a time. Second, the order in which transactions access data objects is restricted by a notion of correctness such as serializability. In hardware resources, however, the order of service has no impact on the validity of the received service in general. Third, a preemption of a transaction from using a data object always leads to aborting the transaction. The transactions has to restart from the beginning later. A transaction that is preempted from service at hardware resources, however, does not lose the service that it has received from the system resources. Finally, a data object is identified by its value, and cannot be substituted for another with a different value. In hardware resources, however, service received from any of a set of similar servers is equivalent. For all these reasons, the policies used for hardware resource scheduling are not directly applicable to concurrency control. This argument is equally valid in real-time database systems. Thus in this research, we have considered concurrency control in real-time database systems separately from scheduling hardware resources such as CPUs, disks, and buffers.

We have analyzed the characteristics of various concurrency control protocol classes, and developed new concurrency control algorithms for real-time database systems.

In addition, we have examined various modeling assumptions for real-time database systems. In order to evaluate the algorithms and to understand the implications of the alternative modeling assumptions on the performance of real-time database systems, we have implemented a real-time database system simulator. Using the simulation system, we have conducted various experiments with a wide range of parameter settings and statistical validity.

In examining the effect of alternative modeling assumptions on the performance of locking and optimistic concurrency control protocols, our experimental results indicate

- that under a soft deadline assumption where all transactions are required to complete eventually, the results of the relative performance of locking and optimistic approaches are similar to those reported in conventional database systems. That is, under an environment where physical resources are limited, locking protocol that tends to conserve physical resources by blocking, instead of aborting transactions, outperforms optimistic protocol. However, when the amount of available resources is abundant so that a large amount of wasted resources can be tolerated and the system performance is determined primarily by concurrency control rather than other resource scheduling, optimistic protocol that allows a higher degree of concurrency outperforms locking protocol.

- that under a firm deadline assumption where tardy transactions are required to be aborted and permanently discarded from the system, optimistic protocol outperforms over a wide range of resource availability and system workload level. The performance difference between the two is caused by the fact that locking protocol suffers performance degradation caused by wasted restarts. The reason for the wasted restarts in locking algorithm is its immediate conflict resolution policy which allows a transaction that will eventually miss its deadline to restart other transactions. Optimistic approach can avoid such wasted restarts because of its policy of delayed

conflict resolution.

We have further investigated the behavior optimistic concurrency control in a firm-deadline environment. We have observed that conventional optimistic protocol using forward validation (OCC-FV) could incur transaction restarts unnecessary to ensure data consistency. We have developed a new optimistic algorithm (OCC-TI) that precisely adjusts and records temporary serialization order among concurrently running transactions, and thereby avoids such unnecessary restarts. Also, we have addressed the problem of using transaction deadline information to improve the performance of optimistic concurrency control protocols, and presented a new deadline-cognizant conflict resolution scheme (Feasible Sacrifice) that uses a feasibility test of a validating transaction before making its sacrifice to favor higher priority running transactions. We have clarified through experiments

- that OCC-TI outperforms OCC-FV over the entire operating conditions, and the performance difference between the two becomes progressively more significant as the level of data contention becomes lower. This shows that the factor of unnecessary restarts is not negligible in the performance of optimistic concurrency control under both finite and infinite resource conditions, and the OCC-TI protocol has the potential to overcome the problem of unnecessary restarts.

- that for optimistic concurrency control, in general, it is better to be conservative than aggressive in terms of making sacrifice of validating transactions to favor running transactions with higher priority. We have shown that the Feasible Sacrifice scheme, over a wide range of workload and operating conditions, provides significant gains over conventional optimistic protocols which are not sensitive to transaction deadline information in resolving conflicts, and performs constantly better than other schemes that are based on a priority wait and wait control mechanisms.

## 6.2. Future Directions

Our research work can be extended in several directions. First, alternative approaches to evaluating the performance of real-time database systems other than using simulation can be explored. These approaches include the measurement from actual running system such as testbed, and the use of analytical methods. Although the testbed implementation and its use for performance evaluation has some drawbacks such as the obstacle of a large programming effort, the inflexibility in system modeling caused by the need of resource commitment, the difficulty in generating realistic workload, and the lack of agreement about the features of real-time database systems, it may provide a concrete experience with a real system and an improved understanding of the functional requirements and operational behavior of real-time database systems. Also, it is interesting to develop an analytical method for evaluating the performance of concurrency control algorithms for real-time database systems. Until now, very little work has been done to-date on this area for a variety of reasons.

Another direction for future work is to consider the issue of scheduling real-time transactions at various resources in the system in connection with real-time concurrency control. One limitation of our current work is that we have considered only the issue of concurrency control, i.e., scheduling of data access operations in real-time database systems. However, real-time database systems have many other resources in addition to data that are shared among transactions. These resources include CPU, memory, communication links, and storage devices. In pursuing the goal of meeting timing constraints, real-time transactions need to be well scheduled at each of these resources along the course of their execution. A single component in the system that ignores timing issues may significantly undermine the best efforts of other components that take timing constraints into account.

Next, this research can be extended to a distributed real-time database system.

Currently, we have focused on a real-time database that resides on a single site. It is pointed out that distribution of data is not only essential feature, but indeed is the key to the success of most practical real-time systems [Rodd89]. There are a number of important issues that will have to be addressed in the design of a distributed real-time database system. First, with regard to the concurrency control in a distributed real-time system, to enforce serializability, optimistic concurrency control requires global validation in addition to local validation. The global validation operation is a complex process, because it may lead to transaction blocking and even deadlock situation. On the other hand, two-phase locking in a distributed database system requires a mechanism to deal with distributed deadlocks. The relative performance of locking and optimistic concurrency control is again an interesting issue in a distributed real-time database system. Second, a distributed real-time database system requires a special handling of transaction commit process. Since a deadline typically signifies the deadline until the end of the two-phase commit, but since the decision on whether or not to commit is taken in the first phase, we have to enter the second phase only if we know that the second phase will complete by the deadline. Third, transaction processing in a distributed real-time database systems becomes more complicated especially when the structure of transactions go beyond flat. A number of options for assigning priorities to transactions in a nested transaction model can be considered. While in a flat transaction model, transactions are competing against each other for resources, components of a nested transactions, even if they have individual deadlines, are executing on behalf of that transaction. Hence scheduling and conflict resolution strategies have to be tailored to handle this difference. Finally, a related topic is the replication of data. Although its potential for fault-tolerance is especially important for distributed real-time database systems, very little work has been done to-date on this issue.

Another issue that has to be addressed in the future is the handling of hard real-time constraints. Although tremendous research efforts have been made in hard real-time systems in dealing with hard deadline constraints [Liu91, Stan88b], little work has been

done in the context of hard real-time database systems. Research into hard real-time database systems may require a drastic departure from the conventional wisdom of real-time database systems in various ways [Aud93, Kim93]. For example, in a hard real-time database, data consistency may not be absolutely necessary for some real-time data which have limited lifetime. Also, we may have to relax the requirement of data consistency to deal with stringent hard real-time constraints. Furthermore, we may need architectural support to guarantee different timing constraints and data consistency requirements.

# Bibliography


[Abbo88] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, August 1988.

[Abbo89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, August 1989.

[Abbo90] Abbott, R. and H. Garcia-Molina, "Scheduling I/O requests with Deadlines: A Performance Evaluation," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, December 1990.

[Agra87] Agrawal, R., M. J. Carey, and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems*, 12, 4: 609-654, December, 1987.

[Agra89] Agrawal, R., and N. Gehani, "Ode: Object Database and Environment," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, 1989.

[Agra92] Agrawal, D., A. El Abbadi, and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1992.

[Aud93] Audsley, N. C., A. Burns, M. F. Richardson, and A. J. Wellings, "Data Consistency in Hard Real-Time Systems," *Technical Report*, Department of Computer Science, University of York, England, 1993.

[Badr88] Badrinath, B. R., and K. Ramamritham, "Synchronizing Transactions on Objects," *IEEE Transactions on Computers*, 37(5): 541-547, May 1988.

[Badr92] Badrinath, B. R., and K. Ramamritham, "Semantic-based Concurrency Control: Beyond Commutativity," *ACM Transactions on Database Systems*, March 1992.

[Baru91] Baruah, S., G. Koren, D. Mao, B. Mishra, A. Raghunathan, L. Rosier, D. Shasha, F. Wang, "On the Competitiveness of On-Line Real-Time Scheduling," *Proceedings of the 12th IEEE Real-Time System Symposium*, December 1991.

[Bern87] Bernstein, P. A., V. Hadzilacos, and N. Goodman, *Concurrency Control*

*and Recovery in Database Systems*, Addison-Wesley, Reading, MA, 1987.

[Biya88] Biyabani, S., J. A. Stankovic, and K. Ramamritham, "The Integration of Deadline and Criticalness in Hard Real-Time Scheduling," *Proceedings of the 9th IEEE Real-Time Systems Symposium*, December 1988.

[Buch89] Buchmann, A., D. McCarthy, M. Hsu, and U. Dayal, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control," *Proceedings of the 5th Data Engineering Conference*, February 1989.

[Care84] Carey, M. J. and M. R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proceedings of the 10th VLDB Conference*, 1984.

[Care87] Carey, M. J., "Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, June 1987, pp. 746 - 751.

[Care89] Carey, M. J., R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference*, 1989.

[Chen91] Chen, S., J. A. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *Journal of Real-Time Systems*, September, 1991.

[Cook91] Cook, R. P., S. H. Son, H. Y. Oh, and J. Lee, "New Paradigms for Real-Time Database Systems," *The 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, Georgia, May 1991.

[Daya88] Dayal, U., et. al., "The HiPAC Project: Combining Active Databases and Timing Constraints," *ACM SIGMOD Record*, March 1988.

[Davi91] Davidson, S., I. Lee, and V. Wolfe, "Timed Atomic Commitment," *IEEE Transactions on Computer*, Vol. 40, No. 5, May 1991.

[Ditt91] Dittrich, K. R., and U. Dayal, "Active Database Systems (Tutorial Notes)," *Proceedings of the 17th VLDB Conference*, September 1991.

[Eswa76] Eswaran, K. P., J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, 19(11), November 1976.

[Fran85] Franaszek, P., and J. T. Robinson, "Limitations of Concurrency in Transaction Processing," *ACM Transactions on Database Systems*, Vol. 10, No. 1, March 1985, Pages 1-28.

[Fran90] Franaszek, P. A., J. T. Robinson, and Alexander Thomasian, "Access

Invariance and Its Use in High Contention Environment," *Proceedings of the 7th International Conference on Data Engineering*, Los Angeles, CA, February, 1990, pp. 47-55.

[Garc83] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, vol. 8, no. 2, pp 186-213, June 1983.

[Grah91] Graham, M. C., "Issues in Real-Time Data Management," *Technical Report* CMU/SEI-91-TR-17, July 1991.

[Gray78] Gray, J. N., "Notes on Database Operating Systems," *Operating Systems: An Advanced Course, Lecture Notes in Computer science* 60:393-481, Springer-Verlag, Berlin, 1978.

[Gupt91] Gupta, R., and E. Horowitz, *Object-Oriented Databases With Applications to CASE, Networks and VLSI CAD*, Prentice Hall, Englewood Cliffs, NJ., 1991.

[Haer84] Haerder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, 9(2), 1984.

[Hari90] Haritsa, J. R., M. J. Carey, and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proceedings of the 1990 ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems (PODS)*, 1990.

[Hari90b] Haritsa, J. R., M. J. Carey, and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Hari91] Haritsa, J. R., Miron Livny, Michael J. Carey, "Value-Based Scheduling in Real-Time Database Systems," *Technical Report* TR1024, Department of Computer Science, University of Wisconsin, May 1991.

[Hari91b] Haritsa, J. R., M. J. Carey, and M. Livny, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[Hari92] Haritsa, J. R., M. J. Carey, and M. Livny, "Data Access Scheduling in Firm Real-Time Database Systems," *Journal of Real-Time Systems*, Kluwer Academic Publishers, 4, 1992.

[Hou89] Hou, W., G. Ozsoyoglu, and B. Taneja, "Processing Aggregate Relational Queries with Hard Time Constraints," *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, June 1989.

[Huan89] Huang, J., J. A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the*

*10th IEEE Real-Time Systems Symposium*, December 1989.

[Huan90] Huang, J., and J. Stankovic, "Buffer Management in Real-Time Database," *COINS TR 90-65*, University of Massachusetts, July 1990.

[Huan91] Huang, J., "Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation," *Ph.D. Dissertation*, May 1991.

[Huan91b] Huang, J., J.A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings of the 17th VLDB Conference*, September 1991.

[Huan91c] Huang, J., J. A. Stankovic, K. Ramamritham, and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[Hurs93] Hurson, A. R., S. H. Pakzad, and J. Cheng, "Object-Oriented Database Management Systems: Evolution and Performance Issues," *IEEE Computer*, February 1993, pp.48-60.

[Jens85] Jensen, E. D., C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the 6th IEEE Real-Time System Symposium*, December 1985.

[Kim89] Kim, W., and F. H. Lochovsky, *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, MA., 1989.

[Kim93] Kim, Y.-K., and S. H. Son, "An Approach Toward Predictable Real-Time Transaction Processing," *Proceedings of the IEEE EuroMicro Workshop on Real-Time Systems*, 1993.

[Klei75] Kleinrock, L., *Queueing Systems*, Volume I, John Wiley and Sons, New York, 1975.

[Kohl81] Kohler, W., "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Survey*, 13, 2: 149-183, June 1981.

[Kort86] Korth, H., and A. Silberschatz, *Database System Concepts*, McGraw-Hill, NY, 1986.

[Kort90] Korth, H., N. Soparkar and A. Silberschatz, "Triggered Real-Time Databases with Consistency Constraints," *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, August 1990.

[Kort90b] Korth, H., N. Soparkar and A. Silberschatz, "Compensating Transactions: A New Recovery Paradigm," *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, August 1990.

[Kung81] Kung H. T., and J. T. Robinson, “On Optimistic Methods for Concurrency Control,” *ACM Transactions on Database Systems*, 6(2): 213-226, June 1981.

[Kuo92] Kuo, T.-W., and A. K. Mok, “Application Semantics and Concurrency Control of Real-Time Data-Intensive Applications,” *Proceedings of the 13th IEEE Real-Time Systems Symposium*, Dec 1992.

[Lampo78] Lamport, L., “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, 21(7):558-565, July, 1978.

[Lamps76] Lampson, B., and H. Sturgis, “Crash Recovery in a Distributed Data Storage System,” *Technical Report*, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, CA, 1976.

[Lee85] Lee, I., and Vijay Gehlot, “Language Constructs for Distributed Real-Time Programming,” *Proceedings of the 6th IEEE Real-Time System Symposium*, December 1985.

[Lee93] Lee, J., and S. H. Son, “An Optimistic Concurrency Control Protocol for Real-Time Database Systems,” *The 3rd International Symposium on Database Systems for Advanced Applications*, Daejon, Korea, April 1993.

[Lee93b] Lee, J., and S. H. Son, “Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems,” *Proceedings of the 14th IEEE Real-Time Systems Symposium*, December 1993.

[Lee93c] Lee, J., and S. H. Son, “Design of Optimistic Concurrency Control for Real-Time Database Systems,” *IEEE Transactions on Data and Knowledge Engineering*, submitted for publication.

[Lee93d] Lee, J. and S. H. Son, “Performance of Real-Time Concurrency Control Algorithms in Parallel Database Systems,” *International Journal of Distributed and Parallel Databases*, submitted for publication.

[Lin87] Lin, K. J., S. Natarajan, and J. Liu, “Imprecise Results: Utilizing Partial Computations in Real-Time Systems,” *Proceedings of the 8th IEEE Real-Time Systems Symposium*, San Jose, CA, December 1987.

[Lin89] Lin, K., “Consistency Issues in Real-Time Database Systems,” *Proceedings of the 22nd Hawaii International Conference on System Science*, January 1989.

[Lin90] Lin Y., and S. H. Son, “Concurrency Control in Real-Time Database Systems by Dynamic Adjustment of Serialization Order,” *Proceedings of the 11th IEEE Real-Time Systems Symposium*, Orlando, Florida, December 1990.

[Liu73] Liu, C. L., J. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” *J.ACM*, 10(1), 1973.

[Liu87] Liu, J., K. Lin and C. Liu, “A Position Paper,” *The 4th IEEE Workshop on Real-Time Operating Systems and Software*, Cambridge, MA, May 1987.

[Liu88] Liu, J., K. Lin, and X. Song, “Scheduling Hard Real-Time Transactions,” *The 5th IEEE Workshop on Real-Time Operating Systems and Software*, May 1988.

[Liu91] Liu, J., K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao, “Algorithms for Scheduling Imprecise Computation,” *IEEE Computer*, 24(5), May 1991.

[Lock86] Locke, C. D., “Best-Effort Decision Making for Real-Time Scheduling,” *Ph.D. Dissertation*, Carnegie-Mellon University, 1986.

[Lort92] Lortz, V. B., and K. G. Shin, “DARTS: A Database Architecture for Real-Time Systems,” *Technical Report*, Department of Computer Science, University of Michigan, 1992.

[Lync83] Lynch, N. A., “Multilevel Atomicity: A New Correctness Criterion for Database Concurrency Control,” *ACM Transactions on Database Systems*, 8, 4,: 484-502, December 1983.

[Marz89] Marzullo, K., “Consistency Control for Transactions with Priorities,” *Technical Report* TR-89-996, Department of Computer Science, Cornell University, 1989.

[O’Nei92] O’Neil, P. E., K. Ramamritham, and C. Pu, “Towards Predictable Transaction Executions in Real-Time Database Systems,” *Computer Science Technical Report* 92-35, University of Massachusetts, 1992.

[Pang92] Pang, H., M. Livny, and M. J. Carey, “Transaction Scheduling in Multiclass Real-Time Database Systems,” *Proceedings of the 13th IEEE Real-Time Systems Symposium*, December 1992.

[Pegd90] Pegden, C. D., R. E. Shannon, and R. P. Sadowski, *Introduction to Simulation using SIMAN*, McGraw-Hill, Inc., NJ, 1990.

[Pu92] Pu, C., and Krithi Ramamritham, “A Formal Characterization of Epsilon Serializability,” *Technical Report*, Department of Computer Science, University of Massachusetts at Amherst, 1992.

[Rajk89] Rajkumar, R., “Task Synchronization in Real-Time Systems,” *Ph.D. Dissertation*, Carnegie-Mellon University, 1989.

[Rama90] Ramamritham, K., J. A. Stankovic, and P. Shiah, “Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems,” *IEEE Transactions on*

*Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[Rama91] Ramamritham, K., S. Son, A. Buchmann, K. Dittrich, and C. Mohan, "Real-Time Databases," panel statement, *Proceedings of the 17th VLDB Conference*, September 1991.

[Rama91b] Ramamritham, K., and P. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties," *Technical Report* 91-92, Computer Science Department, University of Massachusetts, December 1991.

[Rama92] Ramamritham, K., "Real-Time Databases," *International Journal of Distributed and Parallel Databases*, Vol. 1, No. 1, 1992.

[Rich89] Richardson, J. E., M. J. Carey, and D. H. Schuh, "The Design of the E Programming Language," *Computer Sciences Technical Report #791*, Univ. of Wisconsin, Madison, February 1989.

[Robi82] Robinson, J. T., "Design of Concurrency Controls for Transaction Processing Systems," *Ph.D. Thesis*, Technical Report CMU-CS-82-114, Carnegie-Mellon University, 1982.

[Robi82b] Robinson, J. T., "Experiments with Transaction Processing on Multiprocessors," *IBM Research Report*, RC9725, Yorktown Heights, NY, December 1982.

[Rodd89] Rodd, M. G., "Real-Time Issues in Distributed Data Bases for Real-Time Control," *IFAC Distributed Databases in Real-Time Control*, Budapest, Hungary, 1989, pp. 1-7.

[Rose78] Rosenkrantz, D. J., R. E. Stern, and P. M. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. on Database Systems*, 3(2), pp 178-198, June 1978.

[Ryu87] Ryu, I. K., and A. Thomasian, "Performance Analysis of Centralized Databases with Optimistic Concurrency Control," *Performance Evaluation*, 7, 3: 195-211, 1987.

[Schw84] Schwartz, P. M., and A. Z. Spector, "Synchronizing Shared Abstract Types," *ACM Transactions on Computer Systems*, 2(3):223-250, 1984.

[Schw93] Schwan, K. (panel chair), "Semantics of Timing Constraints: Where Do They Come From and Where Do They Go?," Panel Discussion in *the 10th IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[Sha87] Sha, L., R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer

Science Department, Carnegie-Mellon University, 1987.

[Sha88] Sha, L., R. Rajkumar and J. P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[Sha91] Sha, L., R. Rajkumar, S. Son, and C. Chang, "A Real-Time Locking Protocol," *IEEE Transactions on Computers*, Vol. 40, No. 7, July 1991, pp. 793-800.

[Son90] Son, S. H., and J. Lee, "Scheduling Real-Time Transactions in Distributed Database Systems," *The 7th IEEE Workshop on Real-Time Operating Systems and Software*, Charlottesville, Virginia, May 1990.

[Son90b] Son, S. H., and C. Chang, "Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[Son90c] Son, S.,"Real-Time Database Systems: A New Challenge," *IEEE Data Engineering*, 13(4), December 1990, pp 39-43.

[Son92] Son, S. H., J. Lee, and Y. Lin, "Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control," *Journal of Real-Time Systems*, Kluwer Academic Publishers, 4, 1992, pp. 269-276.

[Son92b] Son, S., H, S. Yannopoulos, Y. Kim, and C. Iannacone, "Integration of a Database System with Real-Time Kernel for Time-Critical Applications," *International Conf. on System Integration*, June 1992.

[Song90] Song, X., and K. Lin, "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," *COMPSAC'90*, October 1990.

[Sopa92] Soparkar, N., H. F. Korth, and A. Silberschatz, "Time-Constrained Transaction Scheduling," *TR-92-46*, Department of Computer Sciences, University of Texas at Austin, December 1992.

[Stan88] Stankovic, J. A., and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[Stan88b] Stankovic, J. A., "Misconceptions about Real-Time Computing," *IEEE Computer*, October 1988.

[Stan90] Stankovic, J. A., and K. Ramamritham, "What is Predictability for Real-Time System?," *Journal of Real-Time Systems*, 2: 247-254, Kluwer Academic Publishers, 1990.

[Stra92] Strayer, W. T., "Function-Driven Scheduling: A General Framework for

Expression and Analysis of Scheduling," *Ph.D. Dissertation*, Department of Computer Science, University of Virginia, May 1992.

[Tay85] Tay, Y. C., N. Goodman, and R. Suri, "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, Vol. 10, No. 4, December 1985, pp. 415-462.

[Thom90] Thomasian, A., and E. Rahm, "A New Distributed Optimistic Concurrency Control Method and a Comparison of its Performance with Two-Phase Locking," *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, June 1990.

[Vrbs88] Vrbsky, S. and K. Lin, "Recovering Imprecise Transactions with Real-Time Constraints," *Proceedings of the Symposium on Reliable Distributed Systems*, October 1988.

[Weih88] Weihl, W., "Commutativity-Based Concurrency Control for Abstract Data Types," *IEEE Transactions on Computers*, 37(12):1488-1505, December 1988.

[Wolf92] Wolfe, V. F., L. B. Cingiser, "Issues in Real-Time Object-Oriented Databases," *The 9th IEEE Workshop on Real-Time Operating Systems and Software*, May 1992.

[Wolf93] Wolfe, V. F., S. Davidson, and I. Lee, "RTC: Language Support for Real-Time Concurrency," *Journal of Real-Time Systems*, 5(1): 63-87, March 1993.

[Wolf93b] Wolfe, V. F., L. B. Cingiser, J. Peckham, and J. Prichard, "A Model for Real-Time Object-Oriented Databases," *The 10th IEEE Workshop on Real-Time Operating Systems and Software*, May 1993.

[Wolf93c] Wolfe, V. F., and L. B. Cingiser, "Object-based Semantic Real-Time Concurrency Control," *Proceedings of the 14th IEEE Real-Time System Symposium*, December 1993.

[Yu91] Yu, P., D. Dias, and S. S. Lavenberg, "On the Analytical Modeling of Database Concurrency Control," *IBM Research Report*, RC 15386, February 1991.

[You93] Young, M., and L. Shu, "Real-Time Concurrency with Analytic Worst-Case Latency Guarantees," *The 10th IEEE Workshop on Real-Time Operating Systems and Software*, New York, NY, May 1993.

[Zdo90] Zdonik, S., and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kauffman, San Mateo, CA., 1990.

# Appendix: Concurrency Control Theory

In this Appendix, we give a brief overview of concurrency control problem in traditional database systems, and explain the concept of transaction and serializability. Then we review a number of approaches to the problem proposed for traditional database systems. A more comprehensive discussion on the problem and approaches to the problem can be found in [Bern87].

## A.1. Concurrency Control Problem

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multiprogrammed fashion while preserving the illusion that each user is executing alone on a dedicated system. The main technical difficulty in attaining this goal is to prevent database updates performed by one user from interfering with database retrievals and updates performed by another. Concurrency control has been actively investigated for the past twenty years, and is well understood. A broad mathematical theory has been developed to analyze the problem.

### A.1.1. Examples of Concurrency Control Anomalies

The goal of *concurrency control* is to prevent interference among users who are simultaneously accessing a database. In this section, we illustrate the problem by presenting two "canonical" examples of interuser interference, which are originally from [Bern87]. Both are examples of an on-line electronic funds transfer system accessed via remote *automated teller machines (ATMs)*. In response to customer requests, ATMs

retrieve data from a database, perform computations, and store results back into the database.

**Anomaly 1: Lost Updates**

Suppose two customers, $C_1$ and $C_2$, simultaneously try to deposit money into the same account. In the absence of concurrency control, these two activities could interfere (see Figure A-1). The two ATMs handling the two customers could read the account balance at approximately the same time, compute new balances in parallel, and then store the new balances back into the database. The net effect is incorrect: although two customers deposited money, the database only reflects one activity; the other deposit is lost by the system.

**Anomaly 2: Inconsistent Retrievals**

Suppose two customers, $C_1$ and $C_2$, simultaneously execute the following transactions: $T_1$ by which $C_1$ moves $1,000,000 from Cavalier Corporation's savings account to its checking account, and $T_2$ by which $C_2$ prints Cavalier Corporation's total balance in savings and checking.In the absence of concurrency control, these two transactions could interfere (see Figure A-2). The first transaction might read the savings account balance, subtract $1,000,000, and store the result back in the database. Then the second transaction might read the savings and checking account balances and print the total. Then the first transaction might finish the funds transfer by reading the checking account balance, adding $1,000,000, and finally storing the result in the database. Unlike Anomaly 1, the final values placed into the database by this execution are correct. Still, the execution is incorrect because the balance printed by $C_2$ is $1,000,000 short.

These two examples do not exhaust all possible ways in which concurrent users can interfere. However, these examples are typical of the concurrency control problems that arise in DBMSs.

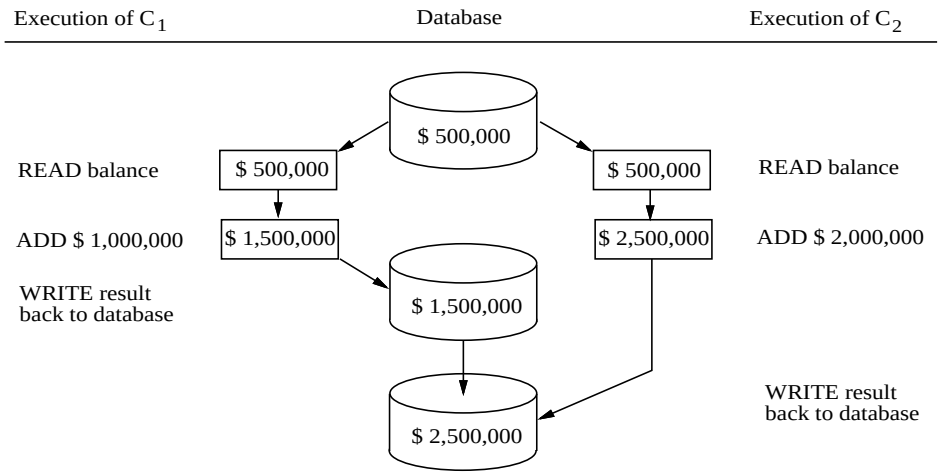



Figure A.1 Lost Update Anomaly

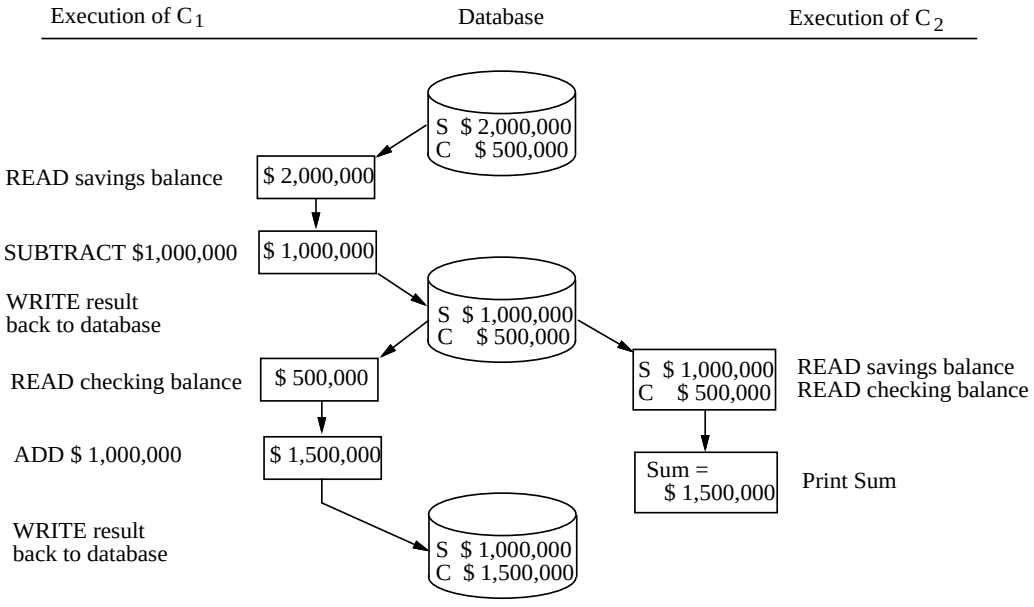



Figure A.2 Inconsistent Retrieval Anomaly

### A.1.2. The Concept of Transaction

To solve the concurrency control problem, the operations performed by a program accessing the database are grouped into sequences called *transactions* [Eswa76]. Users interact with a DBMS by executing transactions. In traditional DBMSs, transactions serve three distinct purposes [Lync83]:

- *logical units* that group together operations comprising a complete task;
- *atomicity units* whose execution preserves the consistency of the database; and
- *recovery units* that ensure that either all the steps enclosed within them are executed or none are.

It is thus by definition that if the database is in a consistent state before a transaction starts executing, it will be in a consistent state when the transaction terminates. In a multiuser system, users execute their transactions concurrently. The DBMS must provide a concurrency control mechanism to guarantee that consistency of data is maintained in spite of concurrent accesses by different users. From the user's viewpoint, a concurrency control mechanism maintains the consistency of data if it can guarantee that each of the transactions submitted to the DBMS by a user eventually gets executed and that the results of the computations performed by each transaction are the same whether it is executed on a dedicated system or concurrently with other transactions in a multiprogrammed system.

The overlap between two concurrent transactions results in a sequence of actions from both transactions, called a *schedule*. A schedule that gives each transaction a consistent view of the state of the database is considered a *consistent schedule*. Consistent schedule is a result of synchronizing the concurrent operations of transactions by allowing only those operations that maintain consistency to be interleaved.

### A.1.3. Serializability

In this section, we give a more formal definition of a consistent schedule. A schedule is consistent if the transactions comprising the schedule are executed serially. In other words, a schedule consisting of transactions $T_1$, $T_2$,...,$T_n$ is consistent if for every $i = 1$ to $n - 1$, transaction $T_i$ is executed to completion before transaction $T_{i+1}$ begins. We can then establish that a *serializable* execution, one that is equivalent to a serial execution, is also consistent. From the perspective of a DBMS, all computations in a transaction either read or write a data item from the database. Thus, two schedules $S_1$ and $S_2$ are said to be computationally equivalent if [Kort86]:

- The set of transactions that participates in $S_1$ and $S_2$ is the same.
- For each data item $Q$ in $S_1$, if transaction $T_i$ executes read($Q$), and the value of $Q$ read by $T_i$ is written by $T_j$, the same will hold in $S_2$ (i.e., read-write synchronization).
- For each data item $Q$ in $S_1$, if transaction $T_i$ executes write($Q$) before $T_j$ executes write($Q$), the same will hold in $S_2$ (i.e., write-write synchronization).

Now the concurrency control problem in traditional database systems reduces to that of testing for serializable schedules. Each operation within a transaction is abstracted into either reading a data item or writing one. Achieving serializability in DBMSs can thus be decomposed into two subproblems: *read-write synchronization* and *write-write synchronization* [Bern81]. The read-write synchronization refers to serializing transactions in such a way that every read operation reads the same value of a data item as it would have read in a serial execution. The write-write synchronization refers to serializing transactions so the last write operation of every transaction leaves the database in the same state as it would have left it in a serial execution. The read-write and write-write synchronizations together result in a consistent schedule. Concurrency control algorithms can be categorized into those that guarantee read-write synchronization, those that are concerned with write-write synchronization, and those that integrate both.

Even though a DBMS may not have any information about application-specific consistency constraints, it can guarantee consistency by allowing only serializable executions of concurrent transactions. This concept of serializability is central to all the concurrency control mechanisms described in the next section. If more semantic information about transactions and their operations is available, schedules that are not serializable but do maintain data consistency could be permitted.

## A.2. Traditional Approaches to Concurrency Control

Most of concurrency control mechanisms for traditional database systems follow one of the three main approaches: two-phase locking, timestamp ordering, and optimistic concurrency control. There have been a few comprehensive discussions and surveys of these mechanisms, including [Bern81, Bern87, Kohl81].

### A.2.1. Two-Phase Locking

Two-phase locking (2PL) mechanism first introduced in [Eswa76] is now accepted as the standard solution to the concurrency control problem in traditional DBMSs. 2PL depends on *well-formed* transactions, which do not lock again entities that have been locked earlier in the transaction, and whose execution is divided into a *growing phase* in which locks are only acquired and a *shrinking phase* in which locks are only released. During the shrinking phase, a transaction is prohibited from acquiring locks. If during the growing phase a transaction attempts to acquire a lock that has already been acquired by another transaction, it is forced to wait until the lock is released. This situation might result in deadlock if transactions are mutually waiting for each other's resources. It has been proven that 2PL guarantees serializability [Bern87].

### A.2.2. Timestamp Ordering

Timestamp Ordering (TO) is a technique whereby a serialization order is selected *a priori* and transaction execution is forced to obey this order. Each transaction is assigned a

unique number, called a timestamp, chosen from a monotonically increasing sequence. TO makes a total order of data operation requests from transactions according to the transactions's timestamps. The mechanism forces a transaction $T_1$ requesting to access a data item *x* that is being held by another transaction $T_2$ to wait until $T_2$ terminates, to abort itself and restart if it cannot be granted access to *x*, or to preempt $T_2$ and get hold of *x*. A scheduling protocol decides which one of these three actions to take after comparing the timestamps of $T_1$ and $T_2$. TO guarantees that a deadlock situation will not arise. It has been proven that TO guarantees serializability [Bern87].

### A.2.3. Multiversion Timestamp Ordering

The basic timestamp ordering mechanism above assumes that only one version of a data item exists. Consequently, only one transaction can access a data item at a time. This restriction can be relaxed by allowing multiple transactions to read and write different versions of the same data item as long as each transaction sees a consistent set of versions for all the data items it accesses. This is the basic idea of the first multiversion timestamp ordering scheme introduced in [Reed78]. In this mechanism, each transaction is assigned a unique timestamp when it starts; all operations of the transaction are assigned the same timestamp. In addition, each data item has the a set of transient versions, each of which is a <*write_timestamp*, *value*> pair, and a set of read timestamps. If a transaction reads a data item, the transaction’s timestamp is added to the set of read timestamps of the data item. A write operation, if permitted by the concurrency control protocol, causes the creation of a new transient version with the same timestamp as that of the transaction requesting the write operation.

The existence of multiple versions eliminates the need for write-write synchronization since each write operations produces a new version and thus cannot conflict with another write operation. The only possible conflicts are those corresponding to read-from relationships [Bern87], as explained below.

Multiversion TO accomplishes read-write synchronization as follows. Let $T_i$ be a transaction with timestamp *TS(i)*, and let *R(x)* be a read operation requested by $T_i$. Then *R(x)* will also be assigned the timestamp *TS(i)*. *R(x)* is processed by reading a value of the version of *x* whose timestamp is the largest timestamp smaller than *TS(R)*, i.e., the latest value written before $T_i$ started. *TS(i)* is then added to the set of read timestamps of *x*. Read operations are always permitted. Write operations, in contrast, might cause a conflict. Let $T_j$ be another transaction with timestamp *TS(j)*, and let *W(x)* be a write operation requested by $T_j$ that assigns value *v* to item *x*. *W(x)* will be permitted only if other transactions with a more recent timestamp than *TS(j)* have not read a version of *x* whose timestamp is greater than *TS(j)*.

### A.2.4. Optimistic Concurrency Control

In many applications, locking has been found to constrain concurrency and to add an unnecessary overhead. The following disadvantages of locking have been pointed out in [Kung81, Thom90]:

1. Lock maintenance represents an unnecessary overhead for read-only transactions, which do not affect the integrity of the database.

2. There are no locking mechanisms that provide high concurrency in all cases. Most of the general-purpose, deadlock-free locking mechanisms work well only in some cases but perform rather poorly in other cases.

3. When large parts of the database reside on secondary storage, locking of objects that are accessed frequently (referred to as *hot spots*) while waiting for secondary memory access causes a significant decrease in concurrency.

4. Not permitting to release locks except at the end of the transaction execution, which although not required is always done in practice to avoid cascaded aborts, decreases concurrency.

5. Most of the time it is necessary to use locking to guarantee consistency since most transaction do not overlap; locking may be necessary only in worst cases.

6. The possibility of deadlock makes 2PL even more inappropriate in distributed database systems, since deadlock detection schemes for distributed database systems tend to be complex and have frequently shown to be incorrect [Knap87]. Even a time-out scheme as an alternative deadlock resolution method has difficulty in the determination of an appropriate time-out interval.

7. The performance of 2PL is severely degraded in distributed database systems, because the significant increase in the total number of concurrent transactions in distributed database systems results in a high lock contention level and hence a high lock conflict probability. Communication delay that leads to a longer lock holding duration makes the lock conflict probability even higher.

The goal of Optimistic Concurrency Control (OCC) proposed in [Kung81] is to avoid these problems of 2PL. OCC requires each transaction to consist of three phases: a *read phase*, a *validation phase*, and a *write phase*. During the read phase, all writes take place on local copies of the records to be written. Then, if it can be established during the validation phase that the changes the transaction made will not violate serializability with respect to all committed transactions, the local copies are made global. Only then, in the write phase, do these copies become accessible to other transactions. There are two properties of OCC that distinguish it from other approaches. First, synchronization is accomplished entirely by restarts, never blocking. Second, the decision to restart or not is made after the transaction has finished executing.