

**Real-Time Transaction Processing:
Design, Implementation, and
Performance Evaluation**

**Jiandong Huang
University of Massachusetts
Amherst, MA 01003**

**COINS Technical Report 91-41
May 1991**

**REAL-TIME TRANSACTION PROCESSING:
DESIGN, IMPLEMENTATION, AND
PERFORMANCE EVALUATION**

A Dissertation Presented

by

JIANDONG HUANG

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

DOCTOR OF PHILOSOPHY

May 1991

Department of Electrical and Computer Engineering

© Copyright by Jiandong Huang 1991
All Rights Reserved

This work was supported by the National Science Foundation under Grant IRI-8908693, Grant DCR-8500332 and Grant CDA-8922572, and by the U.S. Office of Naval Research under Grant N00014-85-K0398.

**REAL-TIME TRANSACTION PROCESSING:
DESIGN, IMPLEMENTATION, AND
PERFORMANCE EVALUATION**

A Dissertation Presented

by

JIANDONG HUANG

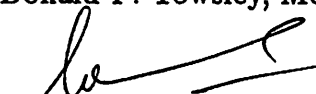
Approved as to style and content by:



John A. Stankovic, Chairperson of Committee



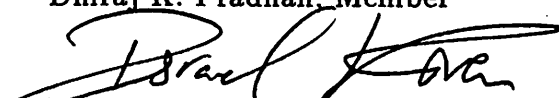
Donald F. Towsley, Member



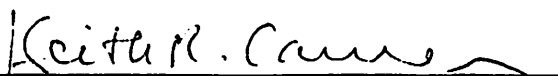
C. Mani Krishna, Member



Dhiraj K. Pradhan, Member



Israel Koren, Member



Keith R. Carver, Department Head
Department of Electrical & Computer Engineering

ACKNOWLEDGEMENTS

It has been a great privilege for me to work with Professor John A. Stankovic. 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.

Many thanks go to Professors Don Towsley and Krithi Ramamritham for working together in carrying out this research and for their constructive comments and suggestions. My special thanks go to Dr. Walter Kohler for introducing me to the CARAT research group and for his generosity and friendly advice. Thanks are also due to Professors C. Mani Krishna, Dhiraj K. Pradhan, and Israel Koren, for being on my dissertation committee and for their useful comments on this work.

I would like to acknowledge my officemates Asit Dan and Chia-shiang Shih for helpful discussions during this research. My special thanks go to Purimetla Bhaskar for his assistance in implementing a wait policy for an optimistic concurrency control protocol on the RT-CARAT testbed. I owe my thanks to Ramesh Nagarajan for his proof reading of part of this dissertation relative to grammar and clarity.

Special thanks go to Betty for sending out all the papers, and to the staff of RCF and ECS for maintaining our RT-CARAT machines.

Finally, I would like to express my gratitude to my parents who have always believed in me and encouraged me. And most importantly, I would like to thank my wife, Ni Ding, for her invaluable support, understanding and love.

ABSTRACT

REAL-TIME TRANSACTION PROCESSING: DESIGN, IMPLEMENTATION, AND PERFORMANCE EVALUATION

MAY, 1991

JIANDONG HUANG

B.S., JILIN UNIVERSITY OF TECHNOLOGY, CHINA

M.S., UNIVERSITY OF DETROIT

Ph.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor John A. Stankovic

In addition to satisfying database consistency requirements, as in traditional database systems, real-time transaction processing systems must also satisfy timing constraints, such as deadlines associated with transactions. To meet timing constraints, transactions need to be well scheduled along the course of their execution. The scheduling process involves multiple functional components in an entire database system. It is further complicated by the extensive interactions among those components.

In this dissertation, we take an integrated approach to study soft real-time database systems where data consistency needs to be guaranteed by the notion of serializability. We develop real-time algorithms for CPU scheduling, concurrency control, conflict resolution, deadlock resolution, transaction wakeup, transaction restart, and buffer management. We also investigate the interactions among the processing components and their combined effect on system performance. The goal is to maximize the number of transactions in meeting their deadlines, and also to maximize the

total value that transactions impart to the database system. In order to evaluate the algorithms and to better understand the operational behavior of real-time database systems, we implement a real-time database testbed called RT-CARAT. Using the testbed, we conduct various experiments with a wide range of parameter settings and statistical validity.

Our main experimental results show that in the integrated system, the CPU scheduling algorithm has the most significant impact in real-time transaction processing; that concurrency control and the associated conflict resolution schemes are the secondary, but still influential, factors; that optimistic concurrency control, compared with two-phase locking, performs better when integrated with priority-driven preemptive CPU scheduling, and further, the optimistic approach may not always outperform the two-phase locking scheme which takes transaction priority into account in resolving data access conflicts; that the basic priority inheritance should not be used in a real-time database that employs two-phase locking, but an extension we developed, called conditional priority inheritance, works quite well; and that the real-time buffer management, integrated with a recovery scheme, does not provide significant gain over typical buffer management techniques.

TABLE OF CONTENTS

	<u>Page</u>
ACKNOWLEDGEMENTS	iv
ABSTRACT	v
LIST OF TABLES	xi
LIST OF FIGURES	xii
 Chapter	
1. INTRODUCTION	1
1.1 What is Real-Time Transaction Processing?	1
1.2 Issues in Real-Time Transaction Processing	2
1.3 Overview of This Dissertation	4
1.3.1 Research Scope and Goals	4
1.3.2 Research Contributions	7
1.4 Organization of the Dissertation	10
2. RELATIONSHIP TO OTHER WORK	11
2.1 System and Transaction Modeling	11
2.2 Priority Scheduling	12
2.3 Concurrency Control	15
3. REAL-TIME DATABASE ENVIRONMENT	17
3.1 Real-Time Database Model	17
3.2 Real-Time Transaction Model	19
3.3 RT-CARAT: A Real-Time Database Testbed	21
3.3.1 System Organization	21
3.3.2 System Parameters	25
3.3.3 Workload Parameters	25

4.	REAL-TIME TRANSACTION PROCESSING UNDER TWO-PHASE LOCKING	28
4.1	Introduction	28
4.2	A Suite of Algorithms	29
4.2.1	CPU Scheduling	29
4.2.2	Conflict Resolution Protocols (CRP)	30
4.2.3	Policies for Transaction Wakeup	34
4.2.4	Deadlock Resolution	35
4.2.5	Transaction Restart	37
4.3	Test Environment	38
4.3.1	Parameter Settings	38
4.3.2	Performance Baseline and Metrics	39
4.4	Experimental Results	40
4.4.1	System Performance Measurements	41
4.4.2	CPU Scheduling	43
4.4.3	Conflict Resolution	46
4.4.4	CPU scheduling vs. Conflict Resolution	49
4.4.5	CPU Bound vs. I/O Bound Systems	49
4.4.6	Sensitivity of Different Value Functions	50
4.5	Conclusions	51
5.	ON USING PRIORITY INHERITANCE IN REAL-TIME DATABASES	61
5.1	Introduction	61
5.2	Transaction Scheduling Under Two-Phase Locking	62
5.2.1	The Problem of Priority Inversion	62
5.2.2	Priority Inheritance (PI)	64
5.2.3	Priority Abort (PA)	65
5.2.4	Conditional Priority Inheritance (CP)	66
5.2.5	Priority Ceiling Protocol	67
5.3	Test Environment	67
5.3.1	Parameter Settings	68
5.3.2	Performance Baselines and Metrics	69
5.4	Experimental Results	71
5.4.1	Data Contention	71

5.4.2	Sensitivity of Threshold (<i>h</i>) Settings	73
5.4.3	Deadline Distribution	74
5.4.4	Transaction Length	75
5.4.5	CPU Bound System	77
5.5	Conclusions	78
6.	BUFFER MANAGEMENT	88
6.1	Introduction	88
6.2	The Buffer Model Used in RT-CARAT	89
6.3	Buffer Management	91
6.3.1	Buffer Allocation	92
6.3.2	Buffer Replacement	94
6.4	Implementation	96
6.5	Test Environment	97
6.6	Experimental Results	100
6.6.1	System Calibration	101
6.6.2	Buffer Management with Buffer Allocation	102
6.6.2.1	The Effectiveness of Buffer Allocation Schemes	103
6.6.2.2	Buffer Allocation vs. Conflict Resolution	105
6.6.2.3	Buffer Allocation vs. CPU Scheduling	106
6.6.2.4	Discussions	107
6.6.3	Buffer Management with Buffer Replacement	108
6.7	Concluding Remarks	110
7.	OPTIMISTIC CONCURRENCY CONTROL	121
7.1	Introduction	121
7.2	Optimistic Concurrency Control for Real-Time Transactions	123
7.2.1	Principle of Optimistic Concurrency Control	123
7.2.2	Optimistic Concurrency Control Using Locking (OCCL)	124
7.2.2.1	Serial Validation-Write: OCCL-SVW	125
7.2.2.2	Parallel Validation-Write: OCCL-PVW	126
7.2.3	Some Implications	127
7.2.3.1	Locking Mechanism	127
7.2.3.2	The Starvation Problem	129

7.2.3.3	Implementation Overhead	129
7.2.4	Conflict Resolution	130
7.3	Test Environment	131
7.4	Experimental Results	134
7.4.1	Protocol Overhead	134
7.4.2	Data Contention	135
7.4.3	Deadline Distribution	138
7.4.4	I/O Resource Contention	139
7.4.5	Transaction Length	140
7.5	Conclusions	141
8.	SUMMARY AND FUTURE RESEARCH	152
8.1	Summary and Conclusions	152
8.2	Future Extensions	155
	BIBLIOGRAPHY	158

LIST OF TABLES

Table		Page
3.1	System Parameters	25
3.2	Workload Parameters	27
4.1	Experimental Settings	39
4.2	System Performance Measurements	42
5.1	Experimental Settings	68
5.2	Policies Examined	69
6.1	Experimental Settings	98
7.1	System Parameters	132
7.2	Workload Parameters	132
7.3	Schemes Examined	133

LIST OF FIGURES

Figure	Page
3.1 Real-Time Database Model	18
3.2 Value Functions for Transaction T_1 and T_2	20
3.3 RT-CARAT processes and message structure	23
3.4 Some Real-Time Related Functional Components in RT-CARAT	24
4.1 CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$	53
4.2 CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$	53
4.3 CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$	54
4.4 Deadline Distribution under EDF	54
4.5 Concurrency Measurement, $w/r = 2/6, T(x, 4, 10), \alpha = 3$	55
4.6 CPU Scheduling, $w/r = 2/6, T(12, 4, 10)$	55
4.7 CPU Scheduling, $w/r = 2/6, T(x, 4, 10), \alpha = 3$	56
4.8 CPU Scheduling, $T(12, 4, 10), \alpha = 3$	56
4.9 CPU Scheduling, $T(12, 4, 10), \alpha = 3$	57
4.10 Conflict Resolution, $w/r = 8/0, T(x, 4, 10), \alpha = 3$	57
4.11 Conflict Resolution, $w/r = 8/0, T(16, 4, 10), \alpha = 3$	58
4.12 Conflict Resolution, $w/r = 8/0, T(16, 4, 10)$	58
4.13 Conflict Resolution, $T(16, 4, 10), \alpha = 3$	59
4.14 CPU Scheduling vs. Conflict Resolution, $w/r = 8/0, T(12, 4, 10), \alpha = 3$	59
4.15 I/O Bound System, $w/r = 8/0, T(12, 4, 0), \alpha = 4$	60
4.16 Value Functions, $w/r = 2/6, T(12, 4, 10)$	60
5.1 Data Contention, $x = 6, \alpha = 4, h = 2$	80
5.2 Data Contention, $x = 6, \alpha = 4, h = 2$	80
5.3 Data Contention, $x = 6, \alpha = 4, h = 2$	81
5.4 Data Contention, $x = 6, \alpha = 4, h = 2$	81
5.5 Data Contention, $x = 6, \alpha = 4, h = 2$	82
5.6 Data Contention, $x = 6, \alpha = 4, h = 2$	82
5.7 Data Contention, $x = 6, \alpha = 4, h = 2$	83
5.8 Data Contention, $x = 6, \alpha = 4, h = 2$	83
5.9 Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$	84
5.10 Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$	84

5.11 Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$	85
5.12 Deadline Distribution, $x = 6, P_w = 0.6, h = 2$	85
5.13 Transaction Length, $P_w = 0.2, \alpha = 4, h = 2$	86
5.14 Long Transactions, $x = 16, \alpha = 4, h = 2$	86
5.15 Mixed Length, $x = \text{avg}[4, 8], P_w = 0.6, h = 2$	87
5.16 CPU Bound System, $P_w = 0.2, \alpha = 4, h = 2$	87
6.1 The Buffer Model	90
6.2 System Calibration, Uniform Access	112
6.3 System Calibration, Skewed Access	112
6.4 System Calibration, Uniform Access	113
6.5 System Calibration, Skewed Access	113
6.6 System Calibration, Uniform Access	114
6.7 System Calibration, Skewed Access	114
6.8 Comparisons of Allocation Schemes	115
6.9 Comparisons of Allocation Schemes	115
6.10 Comparisons of Allocation Schemes	116
6.11 Comparisons of Allocation Schemes	116
6.12 Allocation vs. Conflict Resolution	117
6.13 Allocation vs. Conflict Resolution	117
6.14 Allocation vs. CPU Scheduling	118
6.15 Allocation vs. CPU Scheduling	118
6.16 Allocation vs. CPU Scheduling	119
6.17 Comparisons of Replacement Schemes	119
6.18 Replacement vs. Conflict Resolution	120
7.1 Concurrency Control Overhead	143
7.2 Data Contention, $MPL = 8, x = 6, \alpha = 5$	144
7.3 Data Contention, $MPL = 8, x = 6, \alpha = 5$	144
7.4 Data Contention, $MPL = 8, x = 6, \alpha = 5$	145
7.5 Data Contention, $MPL = 8, x = 6, \alpha = 5$	145
7.6 Data Contention, $MPL = 8, x = 6, \alpha = 5$	146
7.7 Data Contention, $MPL = 8, x = 6, \alpha = 5$	146
7.8 Data Contention, $MPL = 8, x = 6, \alpha = 5$	147
7.9 Data Contention, $MPL = 8, x = 6, \alpha = 5$	147
7.10 Data Contention, $MPL = 8, x = 6, \alpha = 5$	148
7.11 Deadline Distribution, $MPL = 8, x = 6, P_w = 0.2$	148
7.12 Deadline Distribution, $MPL = 8, x = 6, P_w = 0.8$	149

7.13	Deadline Distribution, $MPL = 4, x = 6, P_w = 0.2$	149
7.14	Deadline Distribution, $MPL = 4, x = 6, P_w = 0.8$	150
7.15	Mixed Transactions, $MPL = 8, x = [4, 8], \alpha = 5$	150
7.16	Mixed Transactions, $MPL = 8, x = [4, 8], P_w = 0.2, \alpha = 2$	151

CHAPTER 1

INTRODUCTION

In this introduction the nature of real-time transaction processing is examined, followed by an identification of some of the issues unique to the study of real-time transaction processing. Then, this dissertation and its major results are summarized.

1.1 What is Real-Time Transaction Processing?

A real-time database is a database system where (at least some) transactions have explicit timing constraints such as deadlines. In such a system, transaction processing must satisfy not only the database consistency constraints but also the timing constraints. In other words, real-time databases are different from traditional databases in that the correctness of a transaction execution depends not only on data integrity, but also on the time frame in which the results are produced. Real-time database systems are becoming increasingly important in a wide range of applications where information needs to be processed in a timely manner. Examples of real-time database systems include computer integrated manufacturing systems, program trading in the stock market, radar tracking systems, command and control systems, and air traffic control systems.

With regard to timing constraints, real-time database systems can be categorized as *hard* or *soft*. Hard real-time database systems are those that must absolutely guarantee that transactions will make their deadlines; otherwise, catastrophic consequences may result. A real-time database used in a nuclear power plant, for example, is such a system. The database may store various control parameters for a reactor and a huge amount of information for describing the state of the power plant. Transactions that respond to the control messages, for instance, must be completed within the specified time period, or fatal accidents may occur. In hard real-time database

systems, the requirements on data consistency may not be absolute in certain situations. For example, real-time data, such as those arriving from sensors, have limited lifetimes - they become obsolete after a certain point in time. Consider two transactions, *update* and *read*, which periodically update and read temperature and pressure of a nuclear reactor. Since the read transaction works with real-time data, it may read either of the two parameters from the database even though the update transaction has not completed its update operations on both parameters. Thus, if necessary, real-time transaction processing in hard real-time databases may emphasize meeting timing constraints over maintaining data integrity. Hard real-time database systems require very careful design and implementation at all levels so that the execution of transactions is *predictable*, thereby translating into the ability to guarantee that timing requirements can be met.

In contrast, soft real-time database systems do not require absolute guarantees of meeting transaction deadlines. In such a system, missing a transaction deadline is not catastrophic, but the value of completing the transaction may diminish. An example of soft real-time database systems is program trading in the stock market. A transaction for updating trading information may need to be executed periodically. If the transaction execution exceeds the specified time interval, it may affect the trading activities, but will not result in catastrophic consequences. In soft real-time databases, guarantee of data consistency can be more desirable than that of timing constraints. Usually, research into algorithms and protocols for such systems explicitly address deadlines and make a best effort at meeting deadlines.

Clearly, real-time databases occur over a wide spectrum of applications, from hard real-time to soft real-time, and from stringent data consistency requirements to relaxed situations. This thesis focuses on soft real-time database systems where data consistency needs to be guaranteed using the notion of serializability [Bern87].

1.2 Issues in Real-Time Transaction Processing

Most research on traditional databases focuses on issues like database consistency, but not on meeting any time-constraints associated with transactions. On the other hand, real-time systems research deals with task scheduling to guarantee responses

within deadlines, but has largely ignored the problem of guaranteeing the consistency of shared data. In real-time transaction processing, many new and challenging issues arise as both data consistency and timing constraints are taken into account. In the following, we highlight some of these issues.

Transaction characterization: In traditional databases, a transaction is commonly characterized by *atomicity, consistency, isolation* and *durability*, i.e., the so-called ACID property. However, ACID is not sufficient in describing real-time transactions. In real-time database systems, a deadline may be associated with transactions which specifies the time by which the transaction must complete. In addition, transactions in real-time systems may be assigned different priority levels, reflecting the degree of importance in the real world.

Transaction scheduling: Because of timing constraints imposed on transactions, scheduling becomes an important part of real-time transaction processing. In traditional real-time systems, task scheduling usually takes place at individual processing components such as CPU and I/O, but scheduling processes are relatively independent of each other. Transaction scheduling is different from task scheduling. It involves direct interactions among various processing components across the entire system. For instance, transaction execution may go through CPU scheduling, concurrency control, buffer management, disk scheduling, deadlock detection, and commit procedure. In this processing environment, transaction scheduling needs to deal with all these processing components so that real-time transactions can be treated in a uniform way in meeting their timing constraints. Thus, the problem of transaction scheduling is not merely to develop algorithms that directly address real-time constraints for each individual component, but also to integrate them in a synergistic fashion. It is the dynamic interactions among the different processing components that makes the scheduling difficult.

Concurrency control: Besides enforcing consistency requirements, as used in traditional databases, concurrency control also participates in transaction scheduling in real-time database systems. To schedule real-time transactions, a concurrency control protocol should produce a schedule that reflects the priority of concurrent transactions. Unfortunately, none of the existing concurrency control protocols developed for

traditional databases directly support this kind of priority-driven concurrency control. For example, the use of two-phase locking in real-time database systems may lead to a blocking problem where a high priority transaction must wait for a low priority transaction due to an access conflict. The effect of blocking may jeopardize the scheduling efforts for real-time transactions. Thus, it is necessary to develop real-time oriented concurrency control protocols that can support transaction scheduling.

Similarly, other processing components in real-time database systems also need to be investigated in order to support the systemwide transaction scheduling. Here we give one more example.

Buffer management: Data buffering plays an important role in reducing transaction response time in disk-resident database systems. In traditional database systems, distributing the available buffer frames among concurrent transactions and capturing transaction reference behaviors are the main concerns in buffer management. In a real-time environment, however, the buffer management may need to further consider the timing constraints imposed on referencing transactions. For example, it may be necessary to allocate available buffer frames favoring transactions with earlier deadlines. The impact of buffer management on supporting real-time transactions needs to be studied.

Performance evaluation: Since transaction scheduling involves various processing components, it is inadequate to simply evaluate scheduling schemes for any particular processing component in isolation. Rather, the effect of process interaction on overall system performance needs to be considered. Furthermore, a challenging task that is often ignored in performance studies is to examine the impact of the overheads involved in protocol implementation.

1.3 Overview of This Dissertation

1.3.1 Research Scope and Goals

Real-time transaction processing can be studied from several different perspectives. This largely depends on how the system is specified in terms of consistency

requirements and timing constraints. This thesis considers centralized secondary storage real-time database systems, where database consistency is defined by the notion of *serializability* [Bern87] and the timing constraints associated with transactions are *soft real-time*.

Our objective in this thesis is to design and to evaluate algorithms and protocols in order to support real-time transaction processing in meeting consistency requirement and timing constraints. This thesis includes six aspects:

1. Transaction characterization

As an initial part of this study, we consider the characterization of real-time transactions. In order to model real-time transactions, we introduce a value function which captures both transaction deadline and importance. We are interested in the relation between deadline and importance in protocol design and their combined effect on system performance.

Given the transaction model, the performance goal is to maximize the total value that transactions impart to the system and to maximize the deadline guarantee ratio, i.e., the percentage of submitted transactions that meet their deadlines.

2. Taking an integrated approach

Because of the strong interactions among the various processing components in real-time database systems, we adopt an integrated approach to study real-time transaction processing. An integrated approach is necessary because even a single entity in the system which ignores timing issues may undermine the best efforts of algorithms which do account for timing constraints. In total, the functional components that we have studied in this thesis include: CPU scheduling, concurrency control, conflict resolution, transaction restart, transaction wakeup, deadlock detection/resolution, and buffer management. The focus of this study is to understand the effect of these processing components on system performance and further to identify the dominant factors in this integrated environment.

3. Using priority inheritance

Priority inversion is a special problem caused by the interaction between priority-driven preemptive CPU scheduling and concurrency control operations. We investigate this scheduling problem for real-time database systems that use two-phase locking [Eswa76, Stea76] for concurrency control. We examine two basic schemes for addressing the priority inversion problem, one based on *priority inheritance* and the other based on *priority abort*. We seek answers to the following questions: "Is the priority inheritance scheme appropriate to solve the priority inversion problem in real-time database systems?", "Which mechanism, priority inheritance or priority abort, is better?", and "Is there an approach better than these two basic schemes?"

4. Data buffering

Data buffering is another important aspect of database systems. In a real-time environment, the goal of data buffering is not merely to reduce transaction response time, but more importantly, to increase the number of transactions meeting their timing constraints. We study this processing component based on the existing organization of a real-time database testbed, especially in connection with a recovery scheme using after-image journaling. Here the principal questions are, "How can the transaction timing information be utilized in buffer management?" and "How effective will a real-time buffer management scheme be in an integrated real-time system?"

5. Concurrency control

In pursuing the goal of enforcing serializability, we examine two basic concurrency control approaches, two-phase locking (2PL) and optimistic concurrency control (OCC) [Kung81]. 2PL has been well studied in traditional database systems and is being widely used in commercial systems, while OCC has the property of deadlock freedom and the potential for a high degree of parallelism. Hence a natural question is, "Which of these two approaches is more suitable for real-time transaction processing?"

We also investigate 2PL and OCC in the context of the starvation problem. Because of their higher probability to conflict with other transactions, long transactions are likely to be repeatedly restarted and thus have less chance to

meet their deadline than short transactions. In traditional database systems, the starvation problem is usually addressed by limiting the number of transaction restarts. However, this resolution scheme is inappropriate for real-time transactions, since it may undermine the efforts of transaction scheduling. To cope with the starvation problem in real-time database systems, there is a need to incorporate proper scheduling schemes with proper concurrency control protocols.

6. Implementation and experimentation

Another aspect of this thesis is to build a real-time database testbed. The implementation work is twofold: First, the testbed is built as a flexible tool for testing and performance evaluation of proposed algorithms and protocols; second, the testbed captures the system overheads, which are largely ignored in simulation studies, thus providing an improved understanding of the functional requirements and operational behavior of real-time database systems.

1.3.2 Research Contributions

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

- **Testbed implementation**

We have built the first real-time database testbed. Using the testbed, we conducted various experiments, with a wide range of parameter settings and statistical validity, to study and evaluate all the protocols and algorithms developed in this study. The implementation work and experimental studies have provided a deeper insight into many of the issues encountered in the design of real-time database systems. For example, our experimental studies show that the physical implementation schemes required for optimistic concurrency control have a significant impact on the protocol performance over logical operations considered in simulation studies. This result becomes apparent only because we considered the implementation details and since ours is a testbed, the overheads of the implementation manifest themselves in the performance figures.

- **System integration**

We have taken an integrated approach to develop real-time algorithms for CPU scheduling, concurrency control (based both on locking and on optimistic concurrency control), conflict resolution, deadlock resolution, transaction wakeup, transaction restart, and buffer management. In identifying the dominant factors in such an integrated environment, the experimental results indicate

- that the CPU scheduling algorithm has the most significant impact on real-time transaction processing. It may improve the transaction deadline guarantee ratio by as much as 30%. In addition, it has been observed, for instance, that switching CPU scheduling from a multi-level feedback queue algorithm to an earliest-deadline-first policy completely reverses performance ordering of two-phase locking and optimistic concurrency control protocols;
- that concurrency control and the associated conflict resolution schemes are secondary, but still influential, factors in the integrated system. For example, some real-time oriented conflict resolution protocols may achieve up to 18% performance improvement with respect to transaction deadline guarantee ratio; and
- that the real-time oriented buffer management schemes do not significantly improve system performance over non real-time buffer management schemes.

We have also investigated the interaction between CPU scheduling and concurrency control in the context of priority inversion. To address this problem, we developed a conditional priority inheritance scheme that capitalizes on the advantages of both of the priority inheritance scheme and priority abort scheme.

We have clarified through experiments

- that the basic priority inheritance technique is sensitive to the priority inheritance period. Due to the *life-time blocking* problem under two-phase locking, the basic priority inheritance scheme, which has been shown to be effective in real-time operating systems, does not work well in real-time

database systems. Rather, the conditional priority inheritance scheme that we developed and a simple priority abort scheme perform well for a wide range of system workloads; and

- that blocking resulting from priority inversion is a more serious problem than wastage of system resources. This is especially true when transaction deadlines are loose or when a system is CPU bound.

- **Transaction characterization**

We have developed and studied a real-time transaction model which captures both transaction deadline and criticalness (importance). The experimental results show that these two factors, criticalness level and deadline distributions, strongly affect transaction performance. Under our value weighting scheme, criticalness is a more important factor than the deadline with respect to the performance goal of maximizing the deadline guarantee ratio for high critical transactions and maximizing the value imparted by real-time transactions. This has important implications for real-time scheduling research, which to date has focussed primarily on time constraints independent of the value of tasks (transactions).

- **Real-time optimistic concurrency control**

We have proposed an optimistic scheme, in connection with priority-driven preemptive CPU scheduling, as an alternative to two-phase locking for real-time concurrency control. Based on a locking mechanism to ensure the correctness of the OCC implementation, we developed a set of optimistic concurrency control protocols. To address the starvation problem, we also developed a weighted priority scheduling algorithm which is transaction length and deadline sensitive. Our experimental studies indicate

- that optimistic concurrency control, when integrated with priority-driven CPU scheduling, performs better than two-phase locking. This result is contrary to conventional wisdom in database systems;
- that the optimistic approach may not always outperform the two-phase locking scheme which takes transaction priority into account in resolving

data access conflicts. This is due to the blocking effect caused by the locking mechanism adopted in the OCC implementation; and

- that integrated with the weighted priority scheduling algorithm, optimistic concurrency control exhibits greater flexibility in coping with the starvation problem than two-phase locking.

1.4 Organization of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the relationship of this thesis to previous and concurrently ongoing work. In Chapter 3, a real-time database model and a real-time transaction model are developed. In addition, the system architecture of the real-time database testbed that is built and used for performance evaluation in this study is presented. Chapter 4 describes the work on real-time transaction processing using two-phase locking, in which a set of integrated protocols and their combined effects are examined. As a further investigation on the interaction between priority-driven CPU scheduling and two-phase locking concurrency control, the problem of priority inversion is examined in Chapter 5. In Chapter 6, the problem of buffer management in real-time database systems is discussed and the algorithms for real-time oriented buffer allocation and replacement are developed and evaluated. In Chapter 7, real-time optimistic concurrency control is studied and is further compared with the two-phase locking scheme. Finally, Chapter 8 contains conclusions and directions for future research.

CHAPTER 2

RELATIONSHIP TO OTHER WORK

Real-time transaction processing spans a wide area of research, from real-time systems to database management. Besides the related work in those "traditional" research fields, recently there has also been some work in the area of real-time transaction processing. This chapter describes the relationship of this dissertation work to previous or parallel work in terms of system and transaction modeling, priority scheduling, and concurrency control.

2.1 System and Transaction Modeling

In modeling real-time database systems, this thesis, and most other research [Stan88b, Sha88, Abbo88a, Abbo88b, Liu88, Abbo89, Care89, Son89, Son90, Hari90a, Chn90, Abbo90, Hari90b, LinS90, Jau90, Chn91], assume that timing constraints are directly associated with transactions. In such a model, the transaction is the process entity in the system and transaction scheduling is done to ensure the deadlines are met. An alternative model has been proposed in [Kort90], where timing constraints are applied to consistency constraints. In this model, consistency constraints are represented by database states which faithfully reflect the states of outside world. Any change reflected on a database state may render a consistency constraint invalid, and the state may need to be restored within a specific time period. The system restores the state by choosing one or more transactions from a pre-defined library. The execution of selected transactions may further invalidate other constraints. But the system must eventually return the entire database to a consistent state. This model introduces another framework in modeling real-world systems. However, more concrete work needs to be done in integrating and supporting such a system.

In modeling real-time transactions, an unique aspect of this thesis is that it considers not only transaction deadline but also importance. To model the two characteristics of real-time transactions, a value function is developed. This was influenced by the concept [Lock86] that completion of a process has a value to the system which is expressed as a function of time.

2.2 Priority Scheduling

In early work on real-time transaction processing, different eligibility, priority assignment, and concurrency control algorithms for a main-memory database system was studied in [Abbo88b]. The simulation results show that under the assumption of knowing transaction run time, using an eligibility test to screen out transactions that have missed or are about to miss their deadlines greatly improves system performance, that earliest-deadline is the best overall for priority assignment, and that the conditional restart policy works the best for concurrency control. Since this was the first study of real-time scheduling for database systems, some issues were not well addressed. For example, only three transaction processing components were considered in the study, and the effect of their interaction was not examined.

A theme of our thesis is to take an integrated approach to develop priority-based protocols and algorithms for major processing components which constitute a real-time database system. This study investigates not only the performance of each processing component, but also the interaction among different components and their combined effect on system performance.

In parallel, a systemwide priority scheduling approach was also adopted in [Care89, Jau90]. In [Care89], a real-time database system with three priority-based resources - CPU scheduler, I/O scheduler and buffer manager (including admission control) - was studied. It was shown through simulation that regardless of whether the system bottleneck is the CPU or the disk, priority scheduling of the critical resource must be complemented by a priority-based buffer management policy. However, the conclusion is questionable because of the inadequate design of the disk scheduling algorithm (see the review on disk scheduling below). The work was extended by further

developing algorithms for buffer management and by examining priority-based concurrency control schemes in [Jau90]. The simulation results indicate that a database system can be made to behave like a preemptive-resume server through the use of appropriate priority scheduling algorithms in individual components. While this was a good study on exploring the multiplicity of system components, some important issues were not addressed. For instance, a non-preemptive, priority-based round-robin algorithm was the only CPU scheduling scheme used throughout the study. It was not clear how the scheme relates to other real-time oriented scheduling algorithms, such as priority-base preemptive ones. Another example is that the relation between CPU scheduling and concurrency control was not examined.

Priority inversion [Sha87] is a scheduling problem that occurs due to resource sharing during priority-driven preemptive CPU scheduling. This problem was first investigated in real-time systems. The basic approach proposed to rectify the problem uses the *priority inheritance protocol* [Sha87], where a task blocked by a lower priority task imparts its priority value to the task holding its needed resource. The idea is to allow the low priority task to run and release its resources quickly so that the higher priority tasks can continue. The *priority ceiling protocol* [Rajk89, Chen90] is another scheme developed to solve the priority inversion problem. Under this scheme, the priority inversion is bound to no more than one critical section execution time. The scheme also has the property of deadlock freedom. The performance studies based on the rate-monotonic scheduling framework [Rajk89] have demonstrated that these protocols, applied to the shared resources accessed via semaphores, provides a significant performance advantage.

In early work on the priority inversion problem in real-time transaction processing, a *priority abort* scheme was used in [Abbo88b], where priority inversions are avoided by simply aborting the lower priority transaction. However, the scheme may lead to a high transaction abort rate. This may become a serious problem when a system already contains highly utilized resources. A *priority ceiling protocol* was also considered [Sha88, Son90]. However, this scheme requires a prior knowledge about data to be accessed by real-time transactions. This condition appears to be too restrictive to some real-time database systems where data access is random. Moreover, the scheme becomes extremely conservative, with respect to the degree of concurrency,

if transactions can access any data objects in the database. In [Son90], priority ceiling protocol was evaluated and compared with two-phase locking in a distributed (software) prototyping environment. However, more work on the performance evaluation needs to be done before any conclusion can be drawn from this study.

This thesis investigates the priority inheritance technique in the context of real-time transaction processing. We compare the priority inheritance scheme with the priority abort scheme and further develops a combined abort and priority inheritance scheme which capitalizes on the advantages of the two basic schemes.

Disk scheduling is one aspect of real-time transaction processing for secondary storage database systems [Abbo89, Care89, Abbo90, Chn90, Chn91]. In [Abbo89], a SCAN algorithm based on transaction priority was suggested, where the scan direction is determined by the I/O request with the highest priority. It was demonstrated that when the I/O system is highly utilized, using the priority-based I/O scheduling yields significant performance gains over scheduling I/O requests in a FIFO manner. The algorithm was further extended to take deadline feasibility into account [Abbo90]. A priority-based SCAN algorithm was also proposed in [Care89]. However, because under the algorithm the lower priority requests along the SCAN direction are not served, the average seek time can be worsen as the number of priority levels increases, thus resulting in worse performance on average. Proposed in [Chn90] were two disk scheduling algorithms which take both transaction deadline and seek distance into account. The performance studies indicated that the two algorithms perform consistently better than the real-time disk scheduling algorithms suggested in [Abbo89, Care89, Abbo90].

Besides disk scheduling, I/O subsystem architectures were also explored in the real-time database context. A model for handling read requests differently from write requests was investigated in [Abbo90]. This model buffers write requests in a separate queue from read requests. Two techniques for managing the buffer were examined and both found to be effective. In [Chn91], a mirrored disk architecture was studied. The performance results showed that a mirrored disk I/O subsystem can decrease the fraction of transaction that miss their deadlines over a single disk system by 68%.

This thesis is experimental in nature and uses a real-time database testbed. In the testbed, unfortunately, disk access is under the control of disk controllers instead of the operating system, i.e., there is no way to directly manipulate disk access through the system utilities. Thus, real-time I/O scheduling is not specifically considered in this thesis. However, through careful design of the experiments, we are able to determine the impact of not doing real-time I/O scheduling on system performance. Also, the testbed itself can be used to assist and to verify the simulation studies on disk scheduling [Chn90, Chn91].

2.3 Concurrency Control

Most research work on real-time concurrency control [Stan88b, Sha88, Abbo88a, Abbo88b, Son89, Hari90a, Hari90b, Son90, LinS90, Jau90], including this thesis, follows the notion of serializability [Bern87], while some work considers the relaxation of consistency requirements based on the argument that timing constraints may be more important than data consistency [Stan88b, Liu88, Son88, Vrbs88, Lin89, Song90].

In enforcing serializability, two-phase locking has been used as the basis of real-time concurrency control in most work [Stan88b, Sha88, Abbo88a, Abbo88b, Care89, Son89, Son90, LinS90, Jau90]. This is not surprising since 2PL has been well studied in traditional database systems and is being widely used in commercial databases. But 2PL, on the other hand, has some inherent problems such as the possibility of deadlocks and long and unpredictable blocking times. These appear to be serious problems for real-time transaction processing, since in a real-time environment, transactions need to meet their time constraints as well as consistency requirements.

As an alternative of two-phase locking, this thesis proposes an optimistic concurrency control approach in connection with priority-driven CPU scheduling. We focus on the development of protocols for physical implementation and on the performance study of the impact of the overheads involved in the implementation.

In parallel, real-time optimistic concurrency control was also studied in [Hari90a, Hari90b]. In particular, priority-based conflict resolution mechanisms, such as *priority wait*, were incorporated in the optimistic approach. The proposed schemes are

carefully examined through performance evaluation. It was shown that the real-time optimistic concurrency control outperforms the two-phase locking which aborts lower priority transaction in resolving conflict, and that using priority information in conflict resolution further improve the performance of the optimistic concurrency control. However, the results are based on simulation, where optimistic concurrency control is carried out at the logical level and detailed implementation issues at the physical level are ignored. In practice, the implementation schemes and the corresponding overheads may affect the protocol performance.

Another parallel work is the development of an algorithm that functions in-between two-phase locking and optimistic approach [LinS90]. The goal is to execute transactions with higher priorities first so that high priority transactions are never blocked by uncommitted low priority transactions, while lower priority transactions may not have to be aborted despite conflicting operations. This is achieved by dynamically adjusting the serialization order of active transactions. Since the algorithm has not been evaluated, its performance is not clear at this point.

It should be pointed out, as concluding the chapter, that all the performance studies in the area of real-time transaction processing were based on simulation where implementation issues and their overheads were largely ignored. A unique aspect of this thesis is that it takes an experimental approach. This is important because implementation and experimentation provide deeper insight into system design and performance studies.

CHAPTER 3

REAL-TIME DATABASE ENVIRONMENT

The real-time transaction processing environment considered in this thesis will now be described. A real-time database model, which contains major functional components and is sufficient to demonstrate interactions among the processing components, will first be discussed. This will be followed by the development of a real-time transaction model which captures both transaction deadline and importance. In connection with the proposed models, the overall structure of a real-time database testbed, which is implemented on VAX/VMS machine and used for performance studies in this thesis, will then be presented.

3.1 Real-Time Database Model

Figure 3.1 depicts our real-time database model from the perspective of transaction flow. This model is an extended version of the model used in [Agra87] and accurately describes the way a transaction executes in database systems.

The model represents a closed queueing system where a fixed number of users submits transaction requests one after another, with a certain think time in-between. This model captures many applications in the real world. For example, in an airline reservation system, there is a fixed number of computer terminals. The airline clerk at each terminal may check a flight, reserve a seat, or cancel a reservation for customers. After submitting a request to the system, the clerk waits for a result. He may submit another request after getting a response from the previous one. Of course, this model does not capture all applications. For instance, an open system model is more appropriate for a process control system.

As shown in Figure 3.1, any new or re-submitted transaction is assigned a priority that orders it relative to other concurrent transactions. Before a transaction performs

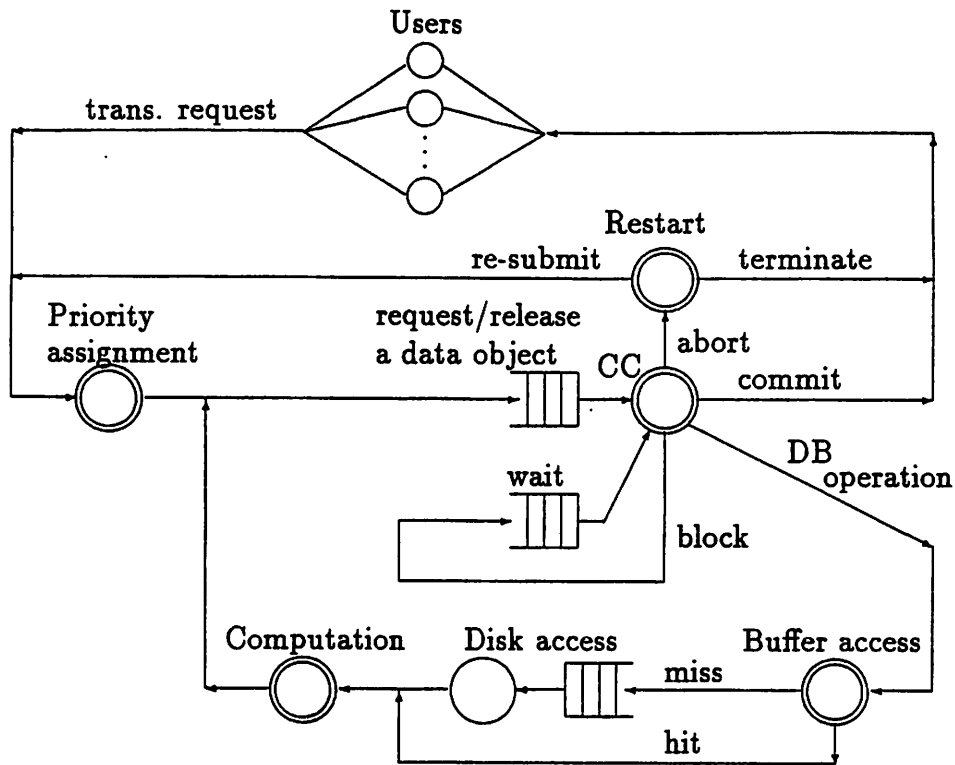


Figure 3.1: Real-Time Database Model

an operation on a data object, it must go through the *concurrency control* component (CC), e.g., to obtain a lock on that object. If the lock request is denied, the transaction will be placed into a wait queue. The waiting transaction will be awakened when the requested lock is released. If the request is granted, the transaction will perform the operation which consists of *global buffer access*, *disk access* (if there is a buffer miss) and *computation*. A transaction may continue this “request-operation cycle” many times until it commits. At its commit stage, the transaction releases all the locks it has been holding. The concurrency control algorithm may abort a transaction for any number of reasons (to be discussed later). In that case, the *restart* component will decide, according to its current policy, whether the aborted transaction should be re-submitted or terminated.

Note that this model only reflects the logical operations involved in transaction processing. It does not show the interaction between the functional components and physical resources. In practice, all of the processing components depicted by a double circle in Figure 3.1 compete for the CPU.

3.2 Real-Time Transaction Model

A real-time transaction is characterized by its length and a value function.¹ The transaction length is dependent on the number of data objects to be accessed and the amount of computation to be performed, which may not always be known. In this study, some of the protocols assume that the transaction length is known when the transaction is submitted to the system. This assumption is justified by the fact that in many application environments like banking and inventory management, the transaction length, i.e., the number of records to be accessed and the number of computation steps, is likely to be known in advance.

In a real-time database, each transaction imparts a value to the system, which is related to its importance and to when it completes execution (relative to its deadline). In general, the selection of a value function depends on the application [Abbo88a]. In this work, we model the value of a transaction as a function of criticalness, start time, deadline, and the current system time. Here criticalness represents the importance of transactions, while deadlines constitute the time constraints of real-time transactions. Criticalness and deadline are two characteristics of real-time transactions and they are not necessarily related. A transaction which has a short deadline does not imply that it has high criticalness. Transactions with the same criticalness may have different deadlines and transactions with the same deadline may have different criticalness values. Basically, the higher the criticalness of a transaction, the larger its value to the system. On the other hand, the value of a transaction is time-variant. A transaction which has missed its deadline will not be as valuable to the system as if it completed before its deadline. We use the following formula to express the value of transaction T:

$$V_T(t) = \begin{cases} c_T, & s_T \leq t < d_T \\ c_T (z_T - t)/(z_T - d_T), & d_T \leq t < z_T \\ 0, & \text{otherwise} \end{cases} \quad (3.1)$$

where t - current time;
 s_T - start time of transaction T;

¹Note that there are no standard workloads for real-time transactions, but a value function has been used in other real-time system work [Lock86].

d_T - deadline of transaction T;
 c_T - criticalness of transaction T, $1 \leq c_T \leq c_{Tmax}$;
 c_{Tmax} - the maximum value of criticalness.

In this model, before its deadline, a transaction has a constant value, which is simply chosen to be its criticalness. The value decays when the transaction passes its deadline and decreases to zero at time z_T . We call z_T the *zero-value point*. As an example, Figure 3.2 shows the value functions of two transactions T_1 and T_2 .

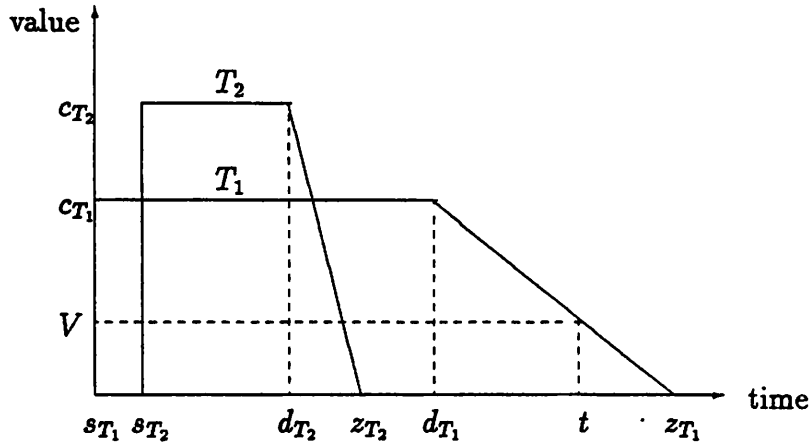


Figure 3.2: Value Functions for Transaction T_1 and T_2

The decay rate, i.e., the rate at which the value of a transaction drops after its deadline, is dependent on the characteristics of the real-time transaction. To simplify the performance study, we model the decay rate as a linear function of deadline and criticalness. Here we study two models with z_T expressed by the following two formulas.

$$z_T = d_T + (d_T - s_T)/c_T \quad (3.2)$$

$$z_T = d_T + (d_T - s_T)/(c_{Tmax} - c_T + 1) \quad (3.3)$$

For a given c_{Tmax} , when c_T increases, under Equation 3.2, z_T decreases, whereas under Equation 3.3, z_T increases. With Equation 3.2, if a transaction is extremely critical ($c_T \rightarrow \infty$), its value drops to zero immediately after its deadline. This is the case that we can see in many hard real-time systems. In this work, we use Equation 3.1

and Equation 3.2 as the base model, and we consider Equation 3.3 as an alternative for Equation 3.2 as a sensitivity study.

The transactions considered here are solely soft real-time. Given the value function, real-time transactions should be processed in such a way that the total value of completed transactions is maximized. In particular, a transaction should abort if it does not complete before time z_T (see Figure 3.2), since its execution after z_T does not contribute any value to the system at all. On the other hand, a transaction that aborts because of deadlock or data conflict may be restarted if it can still impart some value to the system.²

Finally, at times, the estimated execution time of a transaction, r_T , may be known. This information might be helpful in making more informed decisions regarding which transactions are to wait, abort, or restart. This hypothesis is tested in our experiments by using algorithms that make use of r_T .

3.3 RT-CARAT: A Real-Time Database Testbed

In order to provide deeper insight into system design, to experimentally verify the correctness of developed algorithms, to have a flexible tool for performance evaluation of various protocols, and to investigate the limitations of physical systems for real-time transaction processing, we implemented a real-time database testbed, named RT-CARAT.

3.3.1 System Organization

RT-CARAT is an extension of a previously built testbed, called CARAT (Concurrency and Recovery Algorithm Testbed) [Kohl86b]. Currently, RT-CARAT testbed is a centralized, secondary storage real-time database system, running on a VAXstation with two disks, one for the database and the other for the log. The testbed is complete. It contains all the major functional components of a transaction processing system, such as transaction management, data management, log management, and

²In some situations, a transaction may have to be completed even if time is past z_T . We do not consider such transactions here.

communication management. The testbed is built on top of the VAX/VMS operating system. By appropriate settings of various VMS system parameters, non-essential overheads of VMS, such as memory paging and process swapping, are eliminated.

The testbed is implemented as a set of cooperating server processes which communicate via efficient message passing mechanisms. Figure 3.3 illustrates the process and message structure of RT-CARAT. A pool of transaction processes (TR's) simulate the users of the real-time database. Accordingly, there is a pool of data managers (DM's) which service transaction requests from the user processes (the TR's). There is one transaction manager, called the TM server, acting as the inter-process communication agent between TR and DM processes. The communications between TR, TM and DM processes are carried out through the mailbox, a facility provided by VAX/VMS. In order to speed up the processing of real-time transactions, the communication among DM processes is implemented using a shared memory space, called a global section in VAX/VMS.

In RT-CARAT, concurrency control and recovery operations are carried out by the DM processes. Two types of real-time oriented concurrency control protocols, two-phase locking and optimistic concurrency control (as to be presented in the following chapters), are implemented. For transaction failure recovery³, an after-image journaling scheme is simply adopted from the CARAT implementation.

One of the important features of RT-CARAT is the ability to schedule real-time transactions. RT-CARAT is not a time-sharing system. Instead, the CPU is scheduled based on transaction priority with preemption using the underlying VAX/VMS operating system real-time priorities. The CPU scheduler is embedded in the TM. Upon receiving a transaction execution request from a TR, the scheduler assigns a priority to the transaction according to the CPU scheduling policy. The scheduling operation is done by mapping the assigned transaction priority to the real-time priority of the DM process which carries out the transaction execution. At this point, an executing DM will be preempted if it is not the highest priority DM process at the moment, otherwise it will continue to run. Note that an executing transaction with

³A transaction failure occurs when the transaction aborts.

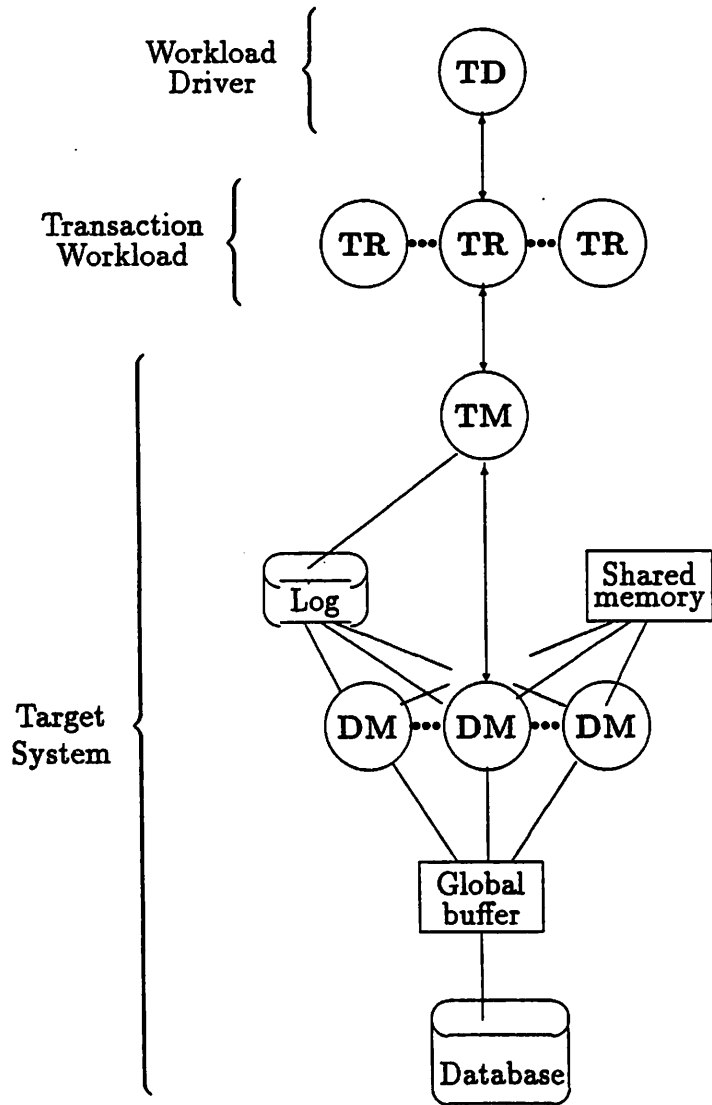


Figure 3.3: RT-CARAT processes and message structure

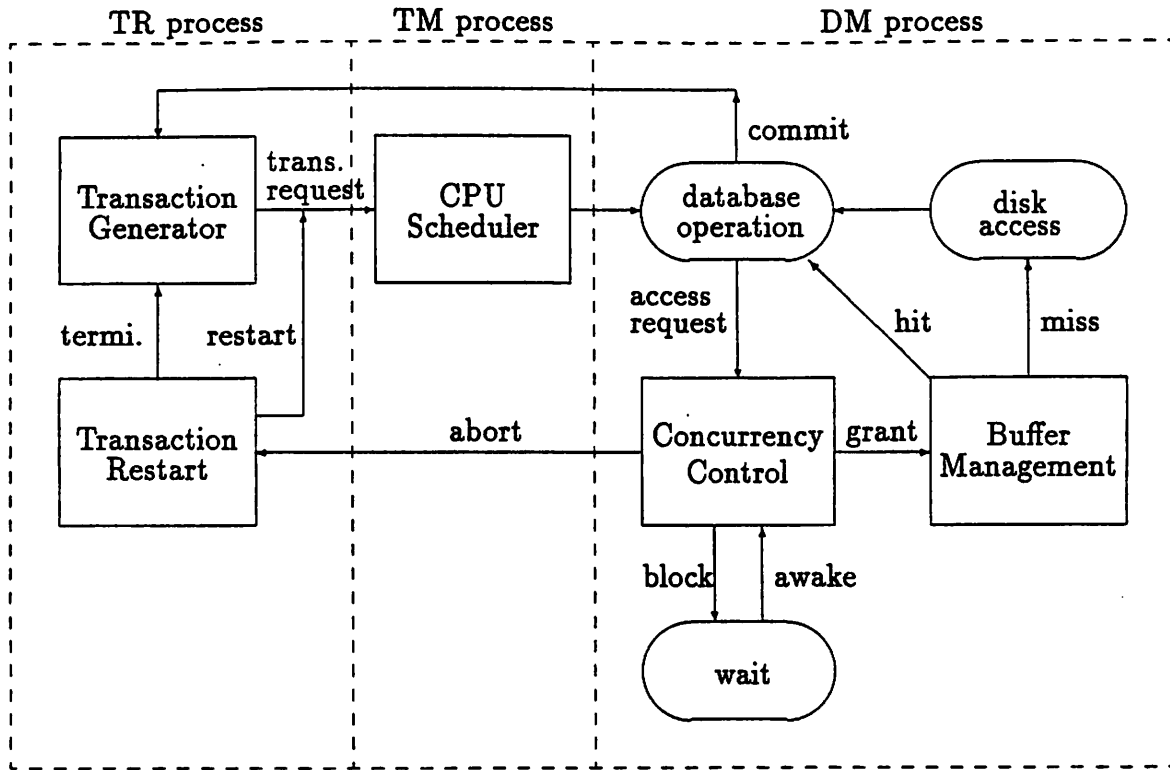


Figure 3.4: Some Real-Time Related Functional Components in RT-CARAT

high priority can be blocked by a low priority transaction because of data conflict. The blocking is resolved by the conflict resolution protocols embedded in the DM.

As an example of part of the testbed implementation, the block diagram in Figure 3.4 shows how the basic functional components of the real-time database model (see Figure 3.1) are incorporated into the TR, TM and DM processes. Designing and evaluating algorithms for those processing components are the main focus of this thesis. They will be discussed in the following chapters.

Note that disk I/O is an important operation in a secondary storage database system. From the point of view of real-time, especially for I/O bound real-time database systems, these I/O operations should be scheduled according to the characteristics of real-time transactions. In our testbed, unfortunately, disk access is under the control of disk controllers instead of the operating system, i.e., there is no way to directly manipulate disk access through the system utilities. Thus, in the current implementation, there is no component dealing with real-time I/O scheduling. However, through

Table 3.1: System Parameters

Parameter	Default Setting
<i>DB-Size</i> (database size)	1000 blocks (6000 records)
<i>GB-Size</i> (global buffer size)	0 blocks
<i>MPL</i> (multiprogramming level)	8

careful design of the experiments, we are able to determine the impact of not doing real-time I/O scheduling on system performance.

3.3.2 System Parameters

Table 3.1 lists three parameters that describe RT-CARAT system settings. In most of experiments, the database consists of 1000 physical blocks (512 bytes each) with each block containing 6 records for a total of 6,000 records. The global buffer is not used, except in Chapter 6 where the issue of data buffering is particularly investigated. The multi-programming level in the system is 8. While this is a low degree of multi-programming, compared to what we would find in practice, the database size in the experiments (1000 blocks) is also smaller than we would find in practice. With a proper system scaling, many factors, such as the level of data access conflict, can model practical situations. Thus, the performance results obtained from the smaller system can reflect the performance of a larger system.

3.3.3 Workload Parameters

As shown in Figure 3.4, a transaction generator in each TR process generates transactions according to a configuration file where length, criticalness, deadline, probability of write transactions, and access distribution are specified. A transaction performs a certain number of predefined operations, called *steps*, and each operation may access a certain number of records and do a certain amount of computation. A transaction terminates upon completion or a termination abort. The transaction generator may spend a certain think time, τ , before submitting a new transaction.

Table 3.2 lists workload parameters considered under RT-CARAT. The transaction *length* is expressed by the form $T(x, y, u)$, where x is the number of steps, y the number of records accessed in each step, and u the amount of computation units per step with 1 unit = 50 ms for (VAXstation II/GPX). The transaction *deadline* is randomly generated from a uniform distribution within a deadline window, $[d_base, \alpha \times d_base]$, where d_base is the window baseline and α is a variable determining the upper bound of the deadline window. For each workload in the experiments, d_base is specified first by the formula:

$$d_base = avg_rsp - stnd_dvi \quad (3.4)$$

where avg_rsp is the average response time of the same real-time transactions (for experiments in Chapters 4 and 6) or the read-only real-time transactions with the same length (for experiments in Chapters 5 and 7) when executed in a non real-time database environment, and $stnd_dvi$ is the standard deviation of the response time. Besides the deadline, each transaction, when initiated, is randomly assigned a *criticalness* from a uniform distribution. In this study, there are up to 8 levels of criticalness and accordingly, the transactions are classified into 8 classes. We specify the criticalness of transaction T as a function of its class, i.e.,

$$c_T(class) = 8 - class + 1, \quad class = 1, 2, \dots, 8. \quad (3.5)$$

The smaller the class number, the higher the corresponding criticalness, or vice versa. Once the deadline and criticalness are specified, the *value function* of the transaction is fixed and the transaction value can be computed at any time (see Section 3.2).

In RT-CARAT, a transaction is either *read* (where each *step* is a sequence of FIND and GET operations) or *write* (where each *step* is a sequence of FIND, GET and MODIFY operations).⁴ The probability that a transaction is a write transaction is specified by P_w . The database may be accessed uniformly or with a certain probability for some *hotspots*, called *skewed access*. The external think time, τ , is fixed at 0.

⁴FIND, GET, MODIFY etc. are the statements of Data Manipulation Language in VAX DBMS. The corresponding operations are fully implemented on RT-CARAT.

Table 3.2: Workload Parameters

Parameter	Range
$T(x, y, u)$ (transaction length):	
x (steps per transaction)	4 - 20 steps
y (records accessed per step)	3 - 4 records
u (computation time per step)	0 - 10 units
α (deadline window factor)	2 - 6
$class$ (transaction class)	1 or 8 levels of criticalness
P_w (probability of <i>write</i> trans.)	0.0 - 1.0
AD (access distribution)	uniform or skewed
τ (external think time)	0

CHAPTER 4

REAL-TIME TRANSACTION PROCESSING UNDER TWO-PHASE LOCKING

Taking an integrated approach, a suite of algorithms are developed for handling CPU scheduling, data conflict resolution, deadlock resolution, transaction wakeup, and transaction restart, based on the two-phase locking scheme for concurrency control. The performance of the algorithms and their interactions are then examined through experimentation. The relationship between transaction deadline and criticality is particularly studied in this chapter.

4.1 Introduction

A real-time database system consists of a set of integrated processing components, such as CPU scheduling, concurrency control, conflict resolution, deadlock detection/resolution, and transaction restart. The function, and the interaction, of these components can best be illustrated by examining the execution of real-time transactions. As shown in Figure 3.1, for example, upon an arrival of a transaction, the CPU scheduler assigns a process priority to the transaction, according to a certain policy, and schedules its execution based on the assigned priority. The concurrency control component controls the transaction access to the shared data objects in the database. If an access conflict occurs between a transaction which attempts to access a data object and a transaction which is currently accessing the data object, then the conflict resolution component decides which transaction has the right to access the data object. Further, if the resolution decision is to let the data requesting transaction wait, the deadlock detection/resolution component must check whether a potential deadlock cycle exists before the transaction is placed into a wait queue. Moreover, if a deadlock cycle is detected, the deadlock must be resolved. At last, if a transaction is

chosen to be the deadlock victim and is aborted, then the transaction restart component needs to decide whether the aborted transaction should be re-submitted or not. As illustrated in this example, each functional component makes control decision on transaction processing, and hence affects the transaction performance.

Thus, the task in design of a real-time transaction processing system is to incorporate the characteristics of real-time transactions (such as deadlines) into the control decision of *each* of these processing component. In other words, an integrated approach needs be taken to develop real-time algorithms for these components so that their combined effect can best meet performance requirements. Here an integrated approach is necessary because even a single component in the system which ignores timing issues may undermine the best efforts of algorithms which do account for timing constraints.

As an initial step of this thesis, we first explicitly address the problems of CPU scheduling, conflict resolution, transaction wakeup, deadlock resolution and transaction restart. The focus of this study is to understand the effect of these processing components on real-time databases and to identify the dominant components in real-time transaction processing.

Another aspect of this study is to investigate the relation between transaction timing constraints and the degree of their importance - defined as *criticalness*, and their combined effects on system performance. To date, research work on real-time transaction processing has focussed on timing constraints. In practice, however, real-time transactions may be at different levels of criticalness. With a value function described by the two factors (see Chapter 3), the algorithm design should aim at maximizing the total value that transactions impart to the system as well as the percentage of transactions which meet their deadlines at each level of criticalness.

4.2 A Suite of Algorithms

4.2.1 CPU Scheduling

There is a wide variety of algorithms for scheduling the CPU in traditional database systems. Such algorithms usually emphasize fairness and attempt to balance

CPU and I/O bound transactions. These scheduling algorithms are not adequate for real-time transactions. In real-time environments, transactions should get access to the CPU based on criticalness and deadline, not fairness. If the complete semantics of transactions, e.g., the data access requirements and timing constraints, are known in advance, then scheduling can be done through transaction preanalysis [Buch89]. On the other hand, in many cases complete knowledge may not be available. Then a priority based scheduling algorithm may be used, where the priority is set based on deadline, criticalness, length of the transaction, or some combination of these factors.

We consider three simple CPU scheduling algorithms. The first two algorithms are commonly found in real-time systems, and the third is an attempt to combine the first two so as to achieve the benefits of both.

- **Scheduling the most critical transaction first (MCF)**
- **Scheduling by earliest deadline first (EDF)**
- **Scheduling by criticalness and deadline (CDF):** In this algorithm, when a transaction arrives, it is assigned a priority based on the formula $(d_T - s_T)/c_T$. The smaller the calculated value, the higher the priority.

Under all of these three algorithms, when a transaction begin its commit phase, its priority is raised to the highest value among all the active transactions. This enables a transaction in its final stage of processing to complete as quickly as possible so that it will not be blocked by other transactions. This policy also reduces the chance for the committing transaction to block other transactions. In all three algorithms, the transactions are preemptable, i.e., an executing transaction (not in its commit phase) can be preempted by a transaction with higher priority.

4.2.2 Conflict Resolution Protocols (CRP)

Two or more transactions have a data conflict when they require the same data in non-compatible lock modes (i.e., *write-write* and *write-read*). The conflict should be resolved according to the characteristics of the conflicting transactions. Here we present five protocols for conflict resolution.

In the following descriptions, T_R denotes the transaction which is requesting a data item D , and T_H is another transaction that is holding a lock on D . The five protocols have the same algorithmic structure as follows:

```
 $T_R$  requests a lock on the data item  $D$ 
if no conflict with  $T_H$ 
  then  $T_R$  accesses  $D$ 
  else call CRP $i$  ( $i = 1,2,3,4,5$ )
end if
```

We start with the simple protocols in terms of complexity and the amount of information required.

Protocol 1 (CRP1): Based on criticalness only

This simple protocol only takes criticalness into account.

```
if  $c_{T_R} < c_{T_H}$  for all  $T_H$ 
  then  $T_R$  waits
  else
    if  $c_{T_R} > c_{T_H}$  for all  $T_H$ 
      then  $T_R$  aborts all  $T_H$ 
      else  $T_R$  aborts itself
    end if
  end if
```

Note that protocol 1 is a deadlock-free protocol, since waiting transactions are always considered in criticalness order. In addition, this protocol implements an *always-abort* policy in a system where all the transactions have the same criticalness.

Protocol 2 (CRP2): Based on deadline-first-then-criticalness

We anticipate that criticalness and deadlines are the most important factors for real-time transactions. Protocol 2 only takes these two factors into account. Here we separate deadline and criticalness by checking the two parameters sequentially. The algorithm for this protocol is:

```
if  $d_{T_R} > d_{T_H}$  for any  $T_H$ 
  then  $T_R$  waits
  else
    if  $c_{T_R} \leq c_{T_H}$  for any  $T_H$ 
```

```

        then  $T_R$  waits
        else  $T_R$  aborts all  $T_H$ 
    end if
end if

```

Protocol 3 (CRP3): Based on deadline, criticalness and estimation of remaining execution time

CRP3 is an extension of CRP2. Besides deadline and criticalness, we further examine the remaining execution time of the conflicting transactions. Here we assume that the computation time and I/O operations of a transaction are known and they are proportional. Then the remaining execution time of transaction T can be estimated by the following formula:

$$time_needed_T(t) = (t - s_T) \times (R_total_T - R_accessed_T(t)) / R_accessed_T(t)$$

where R_total_T is the total number of records to be accessed by T ; $R_accessed_T(t)$ is the number of records that have been accessed as of time t . The protocol is as follows:

```

if  $d_{T_R} > d_{T_H}$  for any  $T_H$ 
  then  $T_R$  waits
  else
    if  $c_{T_R} < c_{T_H}$  for any  $T_H$ 
      then  $T_R$  waits
      else
        if  $c_{T_R} = c_{T_H}$  for any  $T_H$ 
          then
            if  $(time\_needed_{T_R}(t) + t) > d_{T_R}$ 
              then  $T_R$  waits
              else  $T_R$  aborts all  $T_H$ 
            end if
          else  $T_R$  aborts all  $T_H$ 
        end if
      end if
    end if
  end if
end if

```

Protocol 4 (CRP4): Based on a virtual clock

Each transaction, T , has a virtual clock associated with it. The virtual clock value, $VT_T(t)$, for transaction T is calculated by the following formula.

$$VT_T(t) = s_T + \beta_T * (t - s_T), \quad t \geq s_T$$

where β_T is the clock running rate which is proportional to transaction T 's criticalness. The higher the c_T , the larger the value β_T . The protocol controls the setting and running of the virtual clocks. When transaction T starts, $VT_T(t)$ is set to the current real time s_T . Then, the virtual clock runs at rate β_T . That is, the more critical a transaction is, the faster its virtual clock runs. In this work, $\beta_T = c_T$. The protocol is given by the following pseudo code.

```

if  $d_{T_R} > d_{T_H}$  for any  $T_H$ 
  then  $T_R$  waits
  else
    if any  $VT_{T_H}(t) \geq d_{T_H}$ 
      then  $T_R$  waits
      else  $T_R$  aborts all  $T_H$ 
    end if
  end if

```

In this protocol, transaction T_R may abort T_H based on their relative deadlines, and on the criticalness and elapsed time of transaction T_H . When the virtual clock of an executing transaction has surpassed its deadline, it cannot be aborted. Intuitively, this means that for the transaction T_H to make its deadline, we are predicting that it should not be aborted. For further details about this protocol, the reader is referred to [Stan88b].

Protocol 5 (CRP5): Based on combining transaction parameters

This protocol takes into account a variety of different information about the involved transactions. It uses a function $CP_T(t)$ to make decisions.

$$CP_T(t) = c_T * (w_1 * (t - s_T) - w_2 * d_T + w_3 * p_T(t) + w_4 * io_T(t) - w_5 * l_T(t))$$

where $p_T(t)$ and $io_T(t)$ are the CPU time and I/O time consumed by the transaction, $l_T(t)$ is the approximate laxity¹ (if known), and the w_k 's are non-negative weights. The protocol is described by the following pseudo code.

```

if  $CP_{T_R}(t) \leq CP_{T_H}(t)$  for any  $T_H$ 
  then  $T_R$  waits
  else  $T_R$  aborts all  $T_H$ 
end if

```

By appropriately setting weights to zero it is easy to create various outcomes, e.g., where a smaller deadline transaction always aborts a larger deadline transaction. The reader is referred to [Stan88b] for further discussion of this protocol.

In a disk resident database system, it is difficult to determine the computation time and I/O time of a transaction. In our experiments, we simplify the above formula for CP calculation as follows:

$$CP_T(t) = c_T * [w1 * (t - s_T) - w2 * d_T + w3 * (R_accessed_T(t) / R_total_T)]$$

where R_total_T and $R_accessed_T(t)$ are the same as defined in CRP3.

In summary, the five protocols resolve data conflict by either letting the lock-requesting transaction wait or aborting the lock holder(s), depending on various parameters of the conflicting transactions.

4.2.3 Policies for Transaction Wakeup

When a lock holder releases the lock, it is possible that more than one transaction is waiting for the lock. At this point, it is necessary to decide which waiting transaction should be granted the lock. The decision should be based on transaction parameters, such as deadline and criticalness, and also should be consistent with the conflict resolution protocols (CRP) discussed in the previous section. Here we give the policies for transaction wake-up operation which correspond to each CRP.

¹Laxity is the maximum amount of time that a transaction can afford to wait but still make its deadline.

- For CRP1, wake up the waiting transaction with the highest criticalness.
- For CRP2 and CRP3, wake up the waiting transaction with the minimum deadline.
- For CRP4, wake up the waiting transaction with maximum $VT_T(t)$ - the value of virtual clock.
- For CRP5, wake up the waiting transaction with maximum $CP_T(t)$ - the value of combined transaction parameters.

4.2.4 Deadlock Resolution

The use of a locking scheme may cause deadlock. This problem can be resolved by using deadlock detection, deadlock prevention, or deadlock avoidance. For example, CRP1 presented in the previous section is a kind of scheme for deadlock prevention. In this study, we focus on the problem of deadlock detection as it is required by the remaining concurrency control algorithms.

With the deadlock detection approach, a deadlock detection routine is invoked when a transaction is to be queued for a locked data object. If a deadlock cycle is detected, one of the transactions involved in the cycle must be aborted in order to break the cycle. Choosing a transaction for abort is a policy decision. For real-time transactions, we want to choose a victim so that the timing constraints of the remaining transactions can be met as much as possible, and at the same time the abort operation will incur the minimum cost. Here we present five deadlock resolution policies which take into account the timing properties of the transactions, the cost of abort operations, and the complexity of the protocols.

Deadlock resolution policy 1 (DRP1): *Always abort the transaction which invokes the deadlock detection.* This policy is simple and efficient since it does not need any information from the transactions in the deadlock cycle.

Deadlock resolution policy 2 (DRP2): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the longest deadline.*

Recall that a transaction which has passed its zero-value point, z_T , may not have been aborted yet because it may not have executed since passing z_T , and because preempting another transaction execution to perform the abort may not be advantageous. Consequently, in this and the following protocols we first abort any waiting transaction that has passed its zero-value point.

Deadlock resolution policy 3 (DRP3): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the earliest deadline.*

Deadlock resolution policy 4 (DRP4): *Trace the deadlock cycle. Abort the first transaction T with $t > z_T$; otherwise abort the transaction with the least criticalness.*

Deadlock resolution policy 5 (DRP5): Here we use $time_needed_T(t)$ as defined in CRP3. A transaction T is *feasible* if $(time_needed_T(t) + t) < d_T$ and *tardy* otherwise. This policy aborts a tardy transaction with the least criticalness if one exists, otherwise it aborts a feasible transaction with the least criticalness. The following algorithm describes this policy.

```

Step 1: set tardy_set to empty
       set feasible_set to empty

Step 2: trace deadlock cycle
       for each  $T$  in the cycle do
         if  $t > z_T$ 
           then abort  $T$ 
           return
         else
           if  $T$  is tardy
             then add  $T$  to tardy_set
             else add  $T$  to feasible_set
           end if
         end if

Step 3: if tardy_set is not empty
       then search tardy_set for  $T$  with the least criticalness
       else search feasible_set for  $T$  with the least criticalness
       end if
       abort  $T$ 
       return

```

4.2.5 Transaction Restart

A transaction may abort for any number of reasons. Basically, there are two types of aborts.

- *termination abort*: This type of abort is used to terminate a transaction. For example, a transaction may abort itself due to some execution exception. Also, a transaction could be aborted by the system if it has a zero value. Such aborts always lead to transaction termination.
- *concurrency abort*: This type of abort results from concurrency control. For instance, a transaction may be aborted in order to resolve a deadlock, or, a transaction may be aborted by another transaction because of a data access conflict. These aborted transactions should be restarted as long as they may still contribute a positive value to the system.

Based on our transaction model, we propose three policies for transaction restart from *concurrency abort*.

Transaction restart policy 1 (TRP1): *Restart an aborted transaction T if $t < z_T$.* In other words, an aborted transaction will be restarted as long as it may still have some value to the system. Note that the transaction may have already passed its deadline at this point. This policy is intended to maximize the value that the transaction may contribute to the system.

Transaction restart policy 2 (TRP2): *Restart an aborted transaction T if $r_T + t < z_T$.* Here we assume that the runtime estimate r_T of transaction T is known. The decision on transaction restart is based on the estimate of whether the transaction can complete by time z_T , if it is restarted.

Transaction restart policy 3 (TRP3): TRP3 is an extension of TRP2 and is given by the following algorithm.

```
if  $z_T - t < r_T$ 
  then terminate  $T$ 
  else
    if  $d_T - t > r_T$ 
```

```

        then restart  $T$ 
        else increase  $c_T$  by one
             restart  $T$ 
    end if
end if

```

If it is estimated that T can not complete by d_T , then T 's criticalness is increased by one. This restarted transaction has a higher priority than it did in its previous incarnation. Thus the transaction will have a greater chance to meet its timing constraint after its restart. Note that the performance of other concurrent transactions may be affected by this dynamic change of transaction criticalness. The impact of this strategy on system performance is examined through the performance tests.

4.3 Test Environment

4.3.1 Parameter Settings

The experiments described in this chapter were conducted on a VAXstation II/GPX with two RD53 disks - one for the database and the other for the log. Table 4.3.1 summarizes the parameter settings in the experiments². The multi-programming level in the system is 8. The database consists of 3000 blocks. The global buffer is not used. The transaction length $T(x, y, u)$ is varied by changing x while fixing y . The computation time u is set at 10 units in all the experiments, except in the overhead measurement where $u = 0$. The deadline window factor α is also a variable. The smaller the value of α , the tighter the transaction deadlines are and vice versa. Transactions are divided into 8 classes according to their criticalness. The smaller the class number, the higher the corresponding criticalness. In the experiments presented in this chapter, a mix of read and write transactions is specified by w/r - the ratio of *write* transactions to *read* transactions. The access distribution is uniform.

²For details of the parameters, the reader is referred to Sections 3.3.2 and 3.3.3.

Table 4.1: Experimental Settings

Parameter	Settings
<i>MPL</i> (multiprogramming level)	8
<i>DB-Size</i> (database size)	3000 blocks (18000 records)
<i>GB-Size</i> (global buffer size)	0 blocks
<i>x</i> (steps per transaction)	4 - 20 steps
<i>y</i> (records accessed per step)	4 records
<i>u</i> (computation time per step)	0 or 10 units
α (deadline window factor)	2.0 - 5.0
<i>class</i> (transaction class)	8 levels of criticalness
<i>w/r</i> (ratio of write/read trans.)	2/6, 4/4, 6/2, 8/0
<i>AD</i> (access distribution)	uniform

4.3.2 Performance Baseline and Metrics

For these tests, the performance baseline is a non real-time transaction processing system (NRT), which is defined by the following baseline algorithms:

- CPU scheduling policy: *schedule transactions by a multi-level feedback queue (MFQ)*;
- conflict resolution: *place the lock-requesting transaction into a FIFO wait queue (CRP0)*;
- deadlock resolution: DRP1;
- transaction restart: TRP1.

We use the following metrics to evaluate the proposed algorithms and protocols.

- Deadline guarantee ratio (DGR_{class}) - the percentage of transactions in a class that complete by their deadline.
- Average deadline guarantee ratio (ADGR) - the percentage of transactions in all classes that complete by their deadline, i.e.,

$$ADGR = \frac{1}{8} \sum_{class=1}^8 DGR_{class}$$

- Weighted value (WV_{class})- the total value of all transactions in a class that complete by their zero-value points (z_T) divided by the total maximum value of the invoked transactions in all classes.
- Total weighted value (TWV) - the sum of weighted values in all classes, i.e.,

$$TWV = \sum_{class=1}^8 WV_{class}$$

- Abort ratio (AR_{class})- the percentage of aborted transactions in a class.
- Total abort ratio (TAR) - the percentage of aborted transactions in all classes, i.e.,

$$TAR = \sum_{class=1}^8 AR_{class}$$

Our data collection is based on the method of *replication*. In the experiments each test consists of multiple runs where each run was two hours long. The data was collected and averaged over the total number of runs. The number of runs for each test depends on the stability of the data. Our requirement on the statistical data is to generate 95% confidence intervals for the deadline guarantee ratio whose width is less than 10% of the point estimate of the deadline guarantee ratio.

4.4 Experimental Results

We present results from the following six *sets* of experiments:

- **System performance measurements:** The purpose of this experiment is to study the supporting system software performance and to identify the dominant overheads and their magnitudes.
- **CPU scheduling:** The effect of CPU scheduling on the performance of real-time transactions was studied by varying transaction length, deadline setting, write/read ratio, and mixing transactions with different lengths.

- **Conflict resolution:** In these experiments we compared the performance of all the conflict resolution policies by varying transaction length, deadline setting, and write/read ratio.
- **CPU scheduling vs. conflict resolution:** In these experiments we studied the impact of CPU scheduling versus conflict resolution on the performance of real-time transactions.
- **CPU bound vs. I/O bound systems:** This experiment was intended to identify the degree of the system performance degradation due to the lack of real-time based I/O scheduling.
- **Sensitivity study for value functions:** Based on the value function model proposed in Section 3.2, we investigated the impact of two different value functions on the experimental results.

Besides the above studies, we also investigated the proposed *deadlock resolution policies*. The results show that for the wide range of workloads we tested, the performance of all the policies are similar because the deadlock cycle involved only two transactions most of the time. We also did experiments for the three *transaction restart policies*, and found no significant differences between them. This is because the runtime estimate (r_T) used in the algorithms was based on the transaction average response time. Due to the large deviation of the response time, r_T was not accurate enough to support algorithms TRP2 and TRP3. For all the experiments discussed below, DRP1 and TRP1 were used for deadlock resolution and transaction restart, respectively.

Table 4.4 summarizes the protocol selections in the experiments.

4.4.1 System Performance Measurements

To study various system overheads, several experiments are conducted with different workloads and different combinations of the protocols. To focus on the overheads, the user computation time u is set to zero. This has the effect of maximizing number of data requests and the subsequent overheads. Table 4.2 shows the measurement

Protocol	Selection
CPU scheduling	MFQ, MCF, EDF, CDF
Conflict resolution	CRP i , $i = 0, 1, 2, 3, 4, 5$
Deadlock resolution	DRP1
Transaction restart	TRP1
Wakeup policy	A function of conflict resolution policy (see 4.2.2)

Table 4.2: System Performance Measurements

Process/Component	Avg. CPU util.	Standard. devi.
RT-CARAT	0.944	0.012
TRs	0.080	0.001
TM	0.167	0.017
DMs	0.697	0.019
Locking (lock/unlock)	0.438	0.021
Deadlock detection	0.003	0.002
Conflict resolution	0.001	0.000
CPU scheduling	0.006	0.003
Communication	0.213	0.017

results with respect to (average) CPU utilization over all the runs. In the table, the data for RT-CARAT is the overall system CPU utilization which is the sum of the CPU utilizations of the TR, TM and DM processes. *Locking* is a processing component of DM, which in addition includes *deadlock detection* and *conflict resolution*. *CPU scheduling* is part of the TM process. The item *communication* includes all the overhead involved in message communication through mailboxes in the system.

It can be seen that locking and message communication are the main overheads of the system, while the overhead from CPU scheduling, deadlock detection and conflict resolution is negligible. These measures indicate that in the analysis of real-time transaction processing, some overheads such as locking and process communication cannot be simply ignored, while some overheads such as conflict resolution should not be over-emphasized.

4.4.2 CPU Scheduling

In this experiment, in order to observe the effects that different CPU scheduling policies have on performance, we used CRP0 for conflict resolution, i.e., in case of data access conflict, it is always the lock-requesting transaction that is queued.

Figures 4.1-4.3 compare three scheduling schemes with respect to deadline guarantee ratio, weighted value, and total abort ratio, respectively. In this experiment, all the transactions were of equal length, $T(12, 4, 10)$, with $w/r=2/6$ and $\alpha=3$. Figure 4.1 plots the deadline guarantee ratio versus the transaction class. Our first observation is that CPU scheduling algorithms that make use of transaction information perform better than the baseline NRT, except for the transactions in classes 7 and 8 when executed under the scheduling algorithms MCF and CDF. As compared with the baseline and the scheduling algorithms EDF, both MCF and CDF result in a higher deadline guarantee ratio for the transactions with high criticalness, but a lower deadline guarantee ratio for the transactions with low criticalness. This is because the two algorithms take the criticalness of transactions into account and hence the high critical transactions perform better when they compete with low critical transactions. Of the two algorithms, CDF performs better than MCF, since CDF considers not only the criticalness but also the relative deadline of a transaction. Note that for transaction class 1, CDF improves the deadline guarantee ratio from 60% for the baseline to 97% for CDF. With the scheduling algorithm EDF, the performance was basically the same over all classes of transactions. This is understandable since the algorithm totally ignores criticalness.

Figure 4.2 depicts the weighted value that each class of transactions contributed to the system using a linear weighting scheme where each of the 8 criticalness levels differs in value by 1 unit. The performance results indicate that the system gains more value through CPU scheduling compared with the baseline. Overall, the higher the criticalness of a transaction, the larger the value it imparts to the system. Since the current value weighting scheme is linear, other weighting schemes, such as exponential, would result in even higher gains by the real-time CPU scheduling algorithm.

The transaction total abort ratio is shown in Figure 4.3. These plots are basically the inverse of plots for deadline guarantee ratio, i.e., the higher the deadline guarantee

ratio, the lower the abort ratio. For MCF and CDF, the low critical transactions. The abort ratio with EDF is low over all classes of transactions.

The low transaction abort ratio under EDF scheduling policy results from the fact that under EDF most of the time transactions execute in a kind of sequential order. This interesting result can be explained as follows. First, transaction deadline and transaction arrival time are highly correlated in the experiments. Later arriving transactions usually have farther deadlines. Figure 4.4 shows the transaction deadline distributions under scheduling policy EDF.³ Here the horizontal axis represents the deadline order of arriving transactions. Upon arrival of every transaction, we compare its deadline value with the deadlines of those transactions executing in the system. As a result, the smaller its deadline value, the smaller its deadline order. For instance, an arriving transaction will be in order 1 if its deadline value is the smallest compared to the transactions already in the system, and will be in order 8 if its deadline value is the largest. The vertical axis represents the probability of the occurrence of each order. Clearly, the large percentage of arriving transactions have the longer deadline values, compared to the transactions executing in the system. Thus, under EDF most of the later arriving transactions have lower priority than the transactions already in their execution. Therefore, transaction preemption due to transaction arrivals seldom occurs. Second, the system is highly CPU-bound and the computation time in each transaction step is very long (500ms for $u=10$), compared to I/O time (about 80ms for $y=4$). Under such a system, with the given deadline distribution as just discussed, EDF actually reduces the concurrency of transaction execution. This is corroborated by the measures of the average number of granted locks at any instant in the system. Figure 4.5 plots the average number of granted locks as a function of transaction length x , under NRT, MCF, EDF and CDF, respectively. The measures indicate that EDF results in the lowest concurrency among the four scheduling policies. The degree of concurrency under the baseline NRT is the highest, since NRT uses a multi-level feedback queue scheduling policy which emphasizes equality for concurrent transactions. After examining these performance results in detail, it is clear now that under EDF, transaction execution is almost sequential.

³Note that we also measured the transaction deadline distributions under NRT, MCF and CDF. The results are similar to what we show in Figure 4.4.

The scheduling algorithms were also tested for transactions of different lengths and the performance results were basically the same as in the figures presented above.

We next study the sensitivity of the scheduling algorithms to transaction deadline settings. α is varied from 2.0 to 4.0 in steps of 0.5 with $w/r = 2/6$ and $T(12, 4, 10)$. Figure 4.6 shows the average deadline guarantee ratio over 8 classes of transactions. Among the scheduling algorithms EDF is most sensitive to deadline setting. This is because EDF uses only the information about deadline for scheduling. Also note that EDF performs best when the deadline is loose, and worst when the deadline is tight. As we mentioned above, under EDF, transactions execute almost in sequential order most of the time. Thus, when the deadline is tight, it performs worse than the baseline which is scheduled using a multi-level feedback queue. Both MCF and CDF are not sensitive to the deadline distributions. This is obvious for MCF, since it does not use deadline information. For CDF, criticalness is a dominant factor even though it uses deadline information.

We further examine the overall effects of the scheduling algorithms by varying the transaction length. Figure 4.7 presents the total weighted value versus the transaction length x , with $w/r = 2/6$ and $\alpha = 3$. The reader can see that the total value that the system gains under CPU scheduling is far more than the value gained by the baseline. However, when transactions become longer, the performance degrades as much as 20% because of higher data conflict rate, higher deadlock rate, and relatively tighter deadline windows (due to the larger *std.dvi* value).

All the above results are for workloads with $w/r = 2/6$. The CPU scheduling algorithms are also studied by varying the ratio of write/read transactions. Figures 4.8 and 4.9 depict the performance results for different w/r ratio. Under the baseline NRT and the scheduling algorithms MCF and CDF, the data access conflict increases as w/r ratio increases, thus increasing the transaction abort ratio (due to deadlock) and lowering the deadline guarantee ratio. Owing to its sequential execution nature discussed above, EDF is not sensitive to the change of w/r ratio.

These observations and discussions lead to the following points:

- CPU scheduling by MCF and CDF significantly improves the overall performance of real-time transactions for the tested workloads. Further, MCF and

CDF achieve good performance for more critical transactions at the cost of losing some transactions that are less critical. This trade-off reflects the nature of real-time transaction processing that is based on criticalness as well as timing constraints. Moreover, in order to get the best performance, both criticalness and deadline of a transaction are needed for CPU scheduling.

- EDF is most sensitive to deadline distributions and relatively independent of data access conflict. It performs well only when deadlines are not tight.

4.4.3 Conflict Resolution

In this experiment, we study the performance of conflict resolution protocols by varying transaction length, deadline settings, and w/r ratio. The CPU scheduling algorithm used in the experiment is CDF, since it was shown to be the best in the previous section. Unlike the other experiments, the baseline compared in this experiment was chosen to be NRT-CDF - non real-time, applying CPU scheduling (CDF) only. This baseline enables us to isolate the performance differences due to the use of conflict resolution protocols. To create a high conflict rate, we first exercised the workloads with all write transactions (Figures 4.10-4.12) and then the workloads with the mix of read and write transactions (Figure 4.13).

Figure 4.10 presents the performance results with respect to total weighted value versus x , with $w/r=8/0$ and $\alpha = 3$. With short transactions ($x = 4, 8$), all the protocols perform basically the same, and there is no significant performance improvement as compared with NRT-CDF. This is not surprising since with short transactions, the data access conflict is low, and thus, none of the conflict resolution protocols play an important role. The performance difference can be seen as the conflict rate becomes high with long transactions ($x = 12, 16, 20$), where all the protocols outperform NRT-CDF.

Figure 4.11 provides a detailed examination of the performance results of long transactions with $w/r=8/0$, $x = 16$ and $\alpha = 3$. Among the five protocols, CRP1, the simplest one, performs best. This is largely due to the fact that CRP1 is a deadlock-free protocol by which all transaction aborts result from conflict resolution but not from deadlock resolution. Here the point is that if a transaction will be

aborted, then it should be aborted as early as possible in order to reduce the waste in using the resources (e.g. CPU, I/O, critical section and data). Since the conflict resolution is applied before deadlock resolution in the course of transaction execution, an early abort from conflict resolution decreases the amount of resources that would be wasted if the transaction is aborted later from deadlock resolution.

The performance of CRP2 and CRP3 is almost identical, since CRP3 checks only one more condition than CRP2, namely the amount of time that the transaction needs to finish before its deadline. It is clear now that this additional estimated information does not substantially improve the performance.

CRP4 only outperforms NRT-CDF for transactions with high criticalness, but it performs slightly better than CRP2 and CRP3, as well as CRP1, for low critical transactions. This is because CRP4 does not take into account the criticalness of the lock-requesting transactions. When the deadline of lock-requesting transaction is earlier than that of lock-holding transaction, CRP4 allows the lock requester with high criticalness to wait for the lock holder with low criticalness, thus lowering the performance for higher criticalness transactions. This situation never occurs with the other conflict resolution protocols.

The performance of CRP5 is not as good as CRP1 but is better than that of other protocols.⁴ This is because the dominant factor in CRP5 is criticalness (c_T is a multiplicand in calculation of $CP_T(t)$), which results in transaction aborts similar to those due to CRP1, i.e., the large percentage of transaction aborts come from conflict resolution. But because CRP5 is not a deadlock-free protocol, there are still some aborts due to deadlock.

The conflict resolution protocols were also studied with respect to transaction deadline distribution. The performance results illustrated in Figure 4.12 show that the protocols CRP2, CRP3 and CRP4, which explicitly use deadline information, are sensitive to deadline settings. Note that the three protocols do not provide better performance than the baseline when the deadline is tight, and as the deadline window increases, the total weighted value becomes "saturated" (no relative gain occurs as

⁴Different sets of weights w_1 , w_2 and w_3 were exercised in testing of CRP5. There was very little difference in terms of CRP5's performance.

compared with the baseline). This indicates that the conflict resolution plays an important role only for a certain range of deadline distributions.

Thus far, we have shown the performance of conflict resolution protocols under the situation where only write-write conflicts exist. Now we consider the workloads which lead to write-read as well as write-write conflicts. Figure 4.13 presents the performance results in terms of total weighted value versus the ratio of write/read transactions. As we can see, the performance difference among the proposed protocols and the baseline is small when the w/r ratio is equal to $2/6$. This is because the majority of concurrent transactions are executing *read* operations, and therefore the conflict rate is low. As the w/r ratio increases, the transaction performance degrades sharply under all the conflict resolution protocols, except CRP1. There are two reasons for the good performance of CRP1. First, under our value weighting scheme, CRP1 maximizes the total value that transactions impart to the system. Second, as we mentioned previously, the implementation makes CRP1 most efficient in using the system resources.

We also studied the conflict resolution protocols with the mix of write/read transactions by varying the deadline window factor α . The results are similar to what we have shown in Figure 4.12, where only write transactions were considered.

The overall results from this set of experiments show that

- the conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information;
- the relative performance among the proposed conflict resolution protocols is consistent, under different deadline distributions, transaction lengths and the ratio of write/read transactions;
- among the five protocols, CRP1 provides the best performance. This is due to two factors: 1) it is a simple protocol which maximizes the value and 2) its implementation leads to the efficient use of system resources.

4.4.4 CPU scheduling vs. Conflict Resolution

To explicitly distinguish between the effects of CPU scheduling and conflict resolution on system performance, we conducted an experiment which tested four different schemes for real-time transaction processing: (1) NRT - the baseline; (2) CRP1 - applying conflict resolution protocol CRP1 with MFQ (the baseline of the CPU scheduling algorithm); (3) CDF - applying CPU scheduling CDF with CRP0 (the baseline of conflict resolution protocol); and (4) CDF-CRP1 - applying both CPU scheduling CDF and conflict resolution protocol CRP1. The workload for the test presented here is $T(12, 4, 10)$ with deadline setting $\alpha = 3$.

It is observed in Figure 4.14 that CRP1 improves the performance only for the transactions with very high criticalness (class 1), but severely degrades the performance, much worse than NRT, as transactions become less critical. CDF, on the other hand, greatly improves the performance of transactions in most classes. CDF-CRP1, the combination of CDF and CRP1, provides the best performance. The observations that can be made from this experiment indicate that real-time CPU scheduling improves the deadline guarantee ratio of real-time transactions as much as 80% and that there is a need to combine the CPU scheduling scheme with the conflict resolution so as to achieve up to an additional 12% performance improvement.

4.4.5 CPU Bound vs. I/O Bound Systems

In the previous experiments all of the transactions examined had the same computation time - 10 units in each transaction step. Under such workloads, the system is highly CPU bound with the (database) disk utilization being around 15% and the CPU utilization reaching as high as 97%. In this experiment we examine the behavior of workloads which yield systems which are not CPU bound. We created an I/O bound system by reducing the computation time to zero and further increasing the number of I/O operations in each transaction step. Figure 4.15 illustrates a performance result from the I/O bound system where the disk utilization was about 94% and the CPU utilization was approximately 50%. As compared with the baseline NRT, the CPU scheduling scheme (CDF), which performs very well in the CPU bound system (see Figure 4.14), does not improve the performance at all. This is

understandable, since now the system bottleneck is at I/O, not at CPU. Here the interesting observation is that in the I/O bound system, the conflict resolution protocols, combined with CPU scheduling, still improve performance in terms of meeting deadlines for high critical transactions.

Clearly, I/O scheduling sensitive to the real-time nature of transactions is needed for I/O bound systems. We hypothesize that by applying real-time based I/O scheduling algorithms, together with the conflict resolution protocols, the system performance will be improved. For Figure 4.15, in particular, we expect that the deadline guarantee ratio of transactions with higher criticalness will increase through I/O scheduling, similar to what we have achieved from CPU scheduling for a CPU bound system (see Figure 4.14).

4.4.6 Sensitivity of Different Value Functions

As we discussed in Section 3.2, for all of the above experiments, we used Equation 3.1 and Equation 3.2 as the basic value function. To study the sensitivity of our value function model, we now use Equation 3.3 as an alternative to Equation 3.2 and hence combine Equation 3.2 and Equation 3.3 to produce value function VF2.

Figure 4.16 illustrates the performance comparisons between the two different value functions used for CPU scheduling. Here we plot the total weighted value with respect to the deadline window factor α . For the baseline NRT, the total weighted value gained under VF2 is larger than that under VF1. This is what one can expect, since under VF2, the transaction's zero-value point (z_T) is proportional to its criticalness value, and thus, in the case of missing deadline, transactions with high criticalness, modeled by VF2, may impart a larger value to the system than the transactions modeled by VF1. For the same reason, we see the performance difference between VF1 and VF2 for the CPU scheduling scheme EDF. Because the deadline guarantee ratio of EDF is lower than that of NRT when deadlines are tight, and higher when deadlines are loose (see Figure 4.6), the change of the performance difference between VF1 and VF2 is greater with EDF than with NRT. For the CPU scheduling schemes MCF and CDF, there is no large performance difference between VF1 and VF2. This is because both algorithms consider transaction criticalness for

scheduling. According to our value weighting scheme, the large percentage of the total weighted value results from the committed transactions with high criticalness. Since for the tested workloads, the deadline guarantee ratio of the transactions with high criticalness is very high (above 95%, see Figure 4.1), the difference on modeling the decay value will not affect the overall performance results.

VF2 was also exercised with the proposed conflict resolution policies. The observations from these experiments were basically the same as what we just discussed for the results from CPU scheduling.

In general, our experimental results show that the two value functions may result in some performance difference for each protocol. In any case, however, the relative performance among the different protocols is the same.

4.5 Conclusions

In this chapter, we have developed a set of algorithms with regard to the functional components of CPU scheduling, data conflict resolution, transaction wakeup, deadlock resolution, and transaction restart. In addition, we have integrated and implemented these algorithms on RT-CARAT. In general, our experimental results from the testbed indicate the following:

- In a CPU-bound system, the CPU scheduling algorithm has a significant impact on the performance of real-time transactions, and dominates all the other types of protocols. In order to obtain good performance, both criticalness and deadline of a transaction should be accounted for in CPU scheduling;
- Various conflict resolution protocols which directly address deadlines and criticalness produce better performance than protocols that ignore such information. In terms of transaction's criticalness, regardless of whether the system bottleneck is the CPU or the I/O, criticalness-based conflict resolution protocols always improve performance;
- Both criticalness and deadline distributions strongly affect transaction performance. Under our value weighting scheme, criticalness is a more important

factor than the deadline with respect to the performance goal of maximizing the deadline guarantee ratio for high critical transactions and maximizing the value imparted by real-time transactions;

- Overheads such as locking and message communication are non-negligible and can not be ignored in design and analysis of real-time transaction processing.

A processing component that we are unable to directly tackle is I/O scheduling. Unfortunately, most of today's disk controllers have built-in scan algorithms that do not take deadlines and criticalness into account and are not easily modified. However, we have shown, through the experiments, that I/O scheduling is important in real-time transaction processing. This has been proved by a simulation study based on the RT-CARAT architecture [Chn90].

The study conducted in this chapter is an initial step in taking the integrated approach for real-time transaction processing. The interaction between CPU scheduling and concurrency control will be further examined in Chapter 5. Another processing component - buffer management and its interaction with CPU scheduling and concurrency control will be studied in Chapter 6. Different protocols for real-time concurrency control will be investigated in Chapter 7.

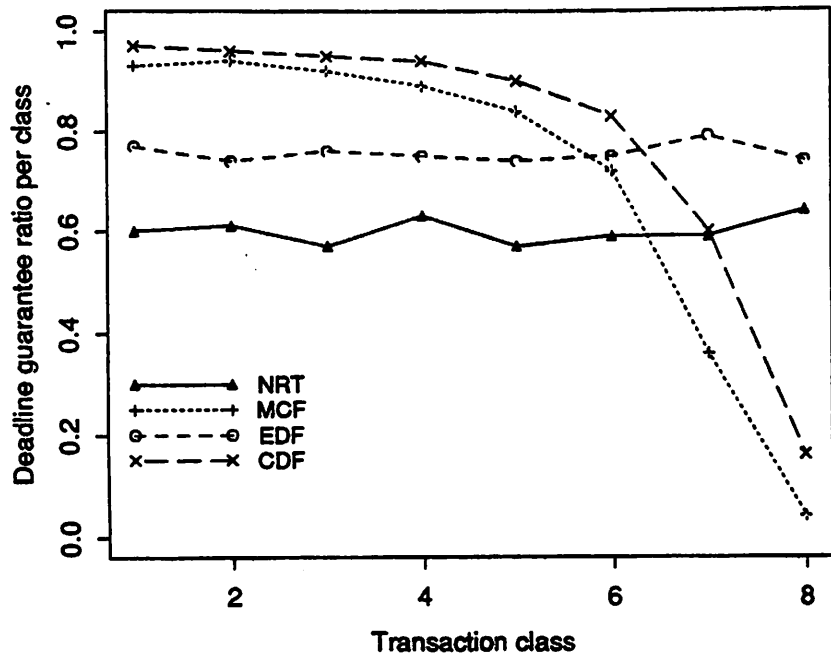


Figure 4.1: CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$

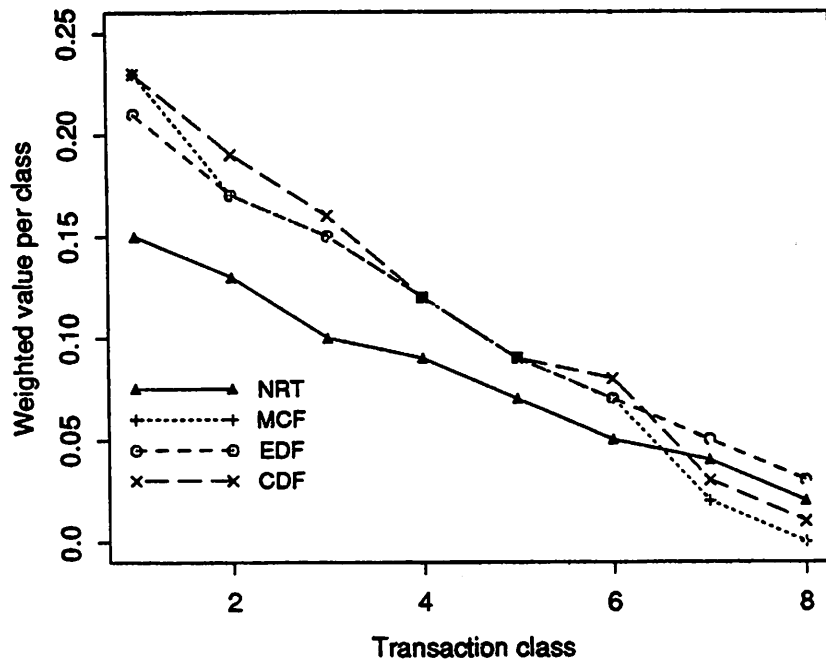


Figure 4.2: CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$

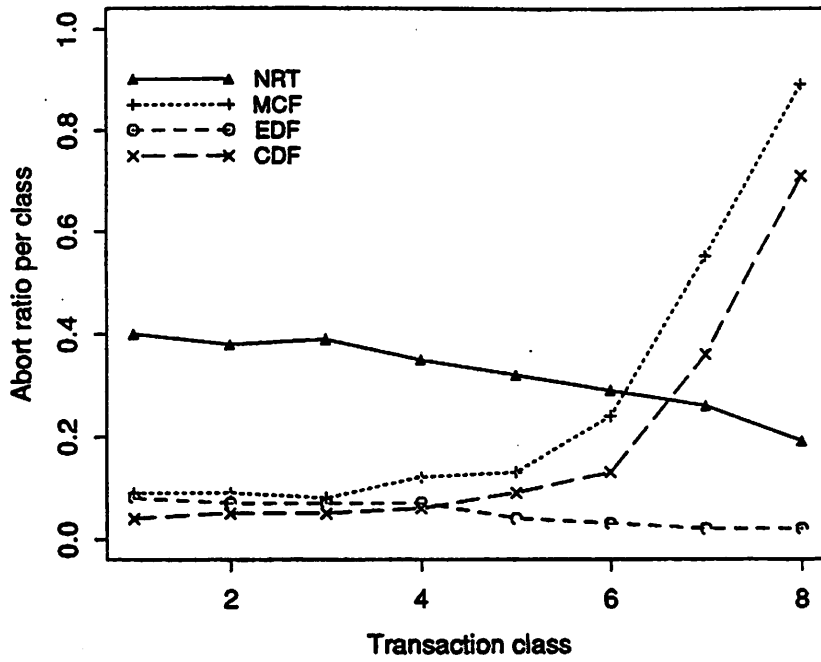


Figure 4.3: CPU Scheduling, $w/r = 2/6, T(12, 4, 10), \alpha = 3$

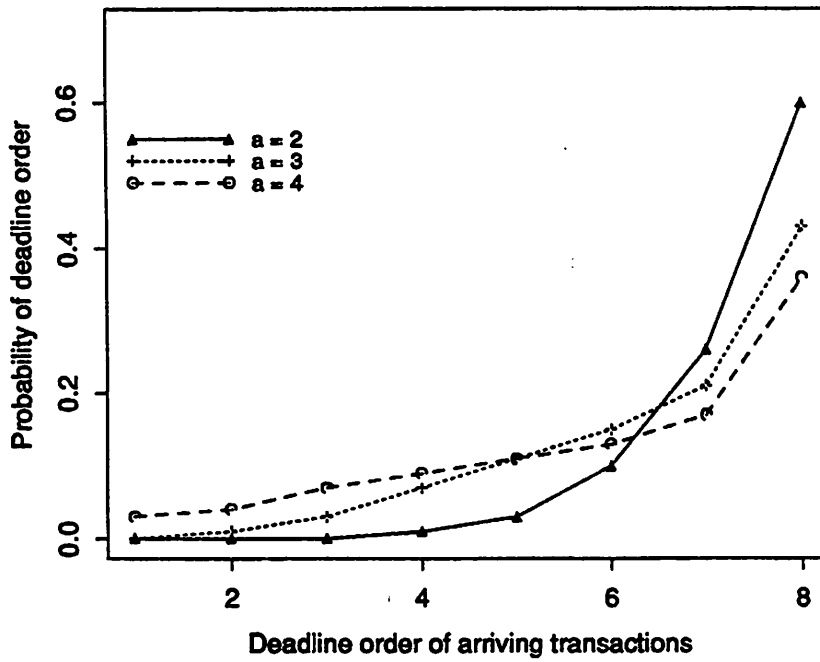


Figure 4.4: Deadline Distribution under EDF

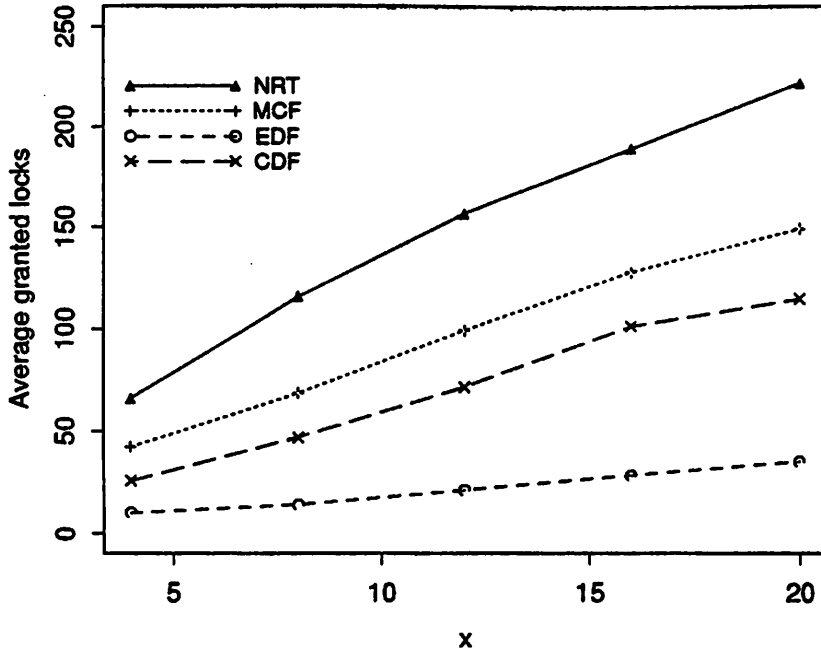


Figure 4.5: Concurrency Measurement, $w/r = 2/6, T(x, 4, 10), \alpha = 3$

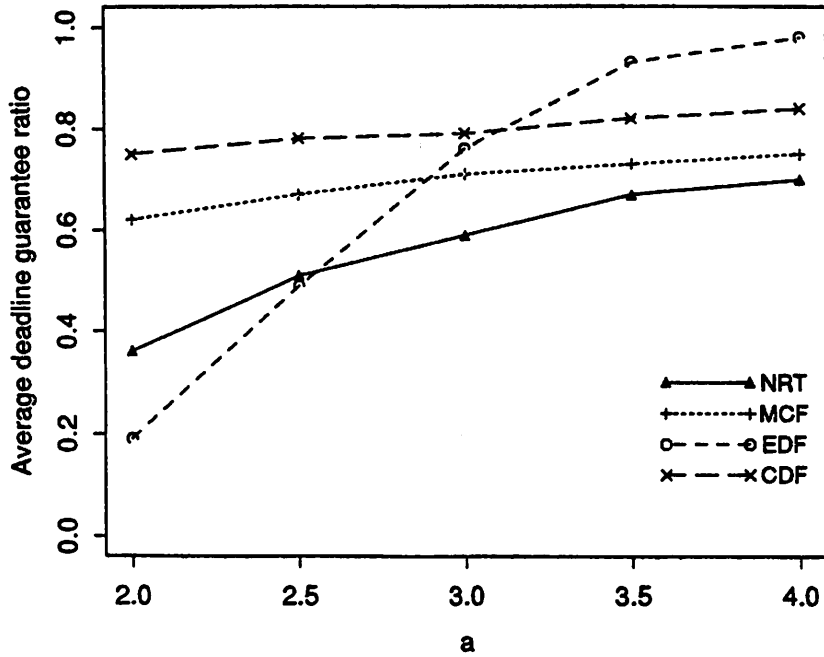


Figure 4.6: CPU Scheduling, $w/r = 2/6, T(12, 4, 10)$

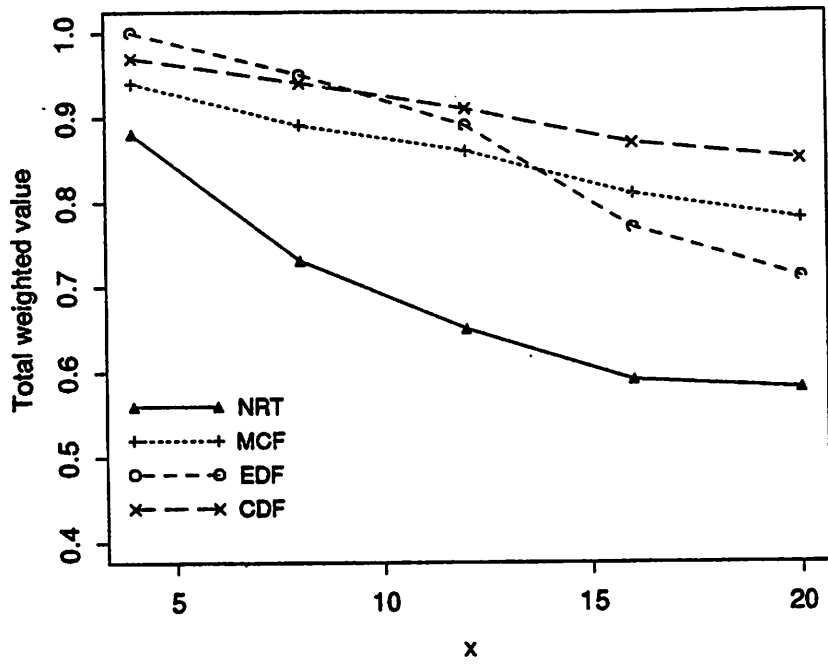


Figure 4.7: CPU Scheduling, $w/r = 2/6, T(x, 4, 10), \alpha = 3$

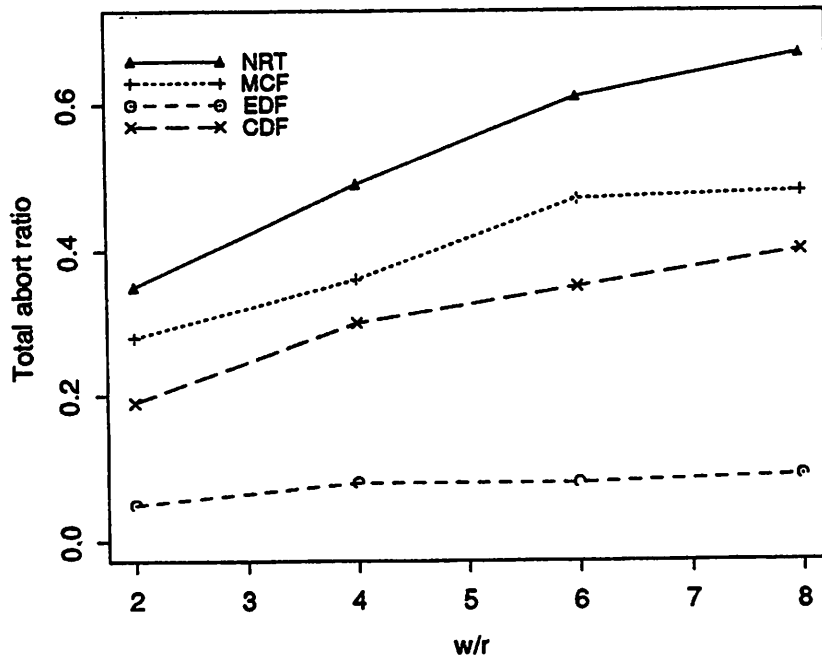


Figure 4.8: CPU Scheduling, $T(12, 4, 10), \alpha = 3$

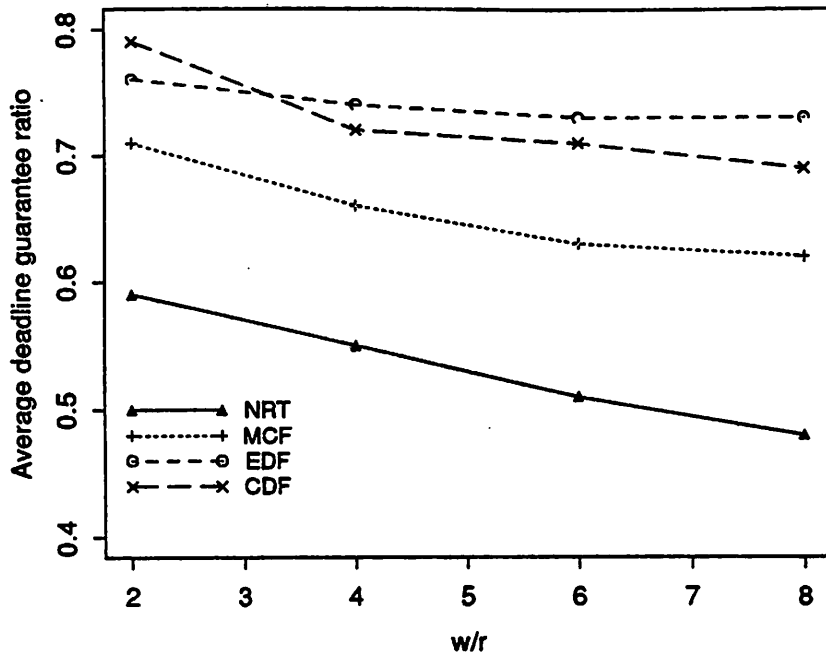


Figure 4.9: CPU Scheduling, $T(12, 4, 10)$, $\alpha = 3$

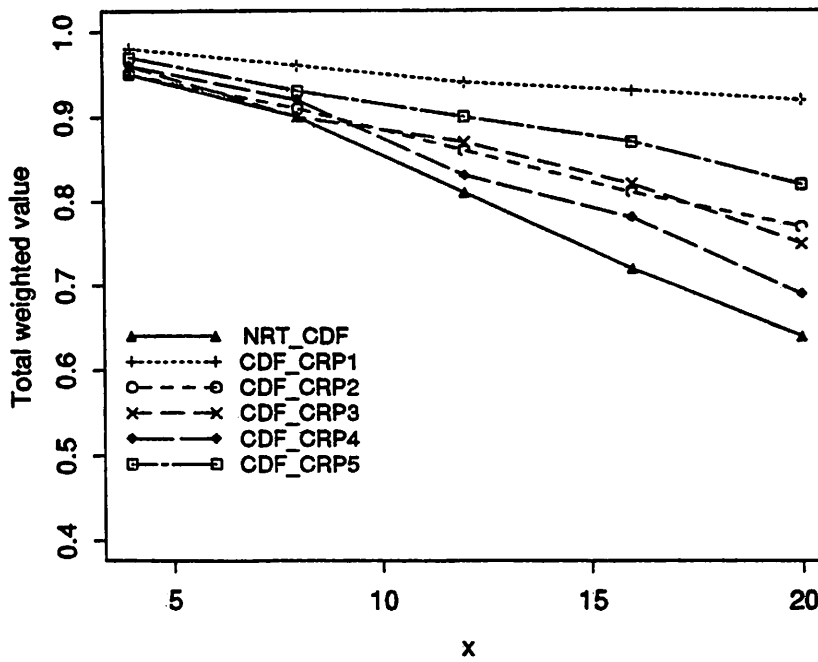


Figure 4.10: Conflict Resolution, $w/\tau = 8/0$, $T(x, 4, 10)$, $\alpha = 3$

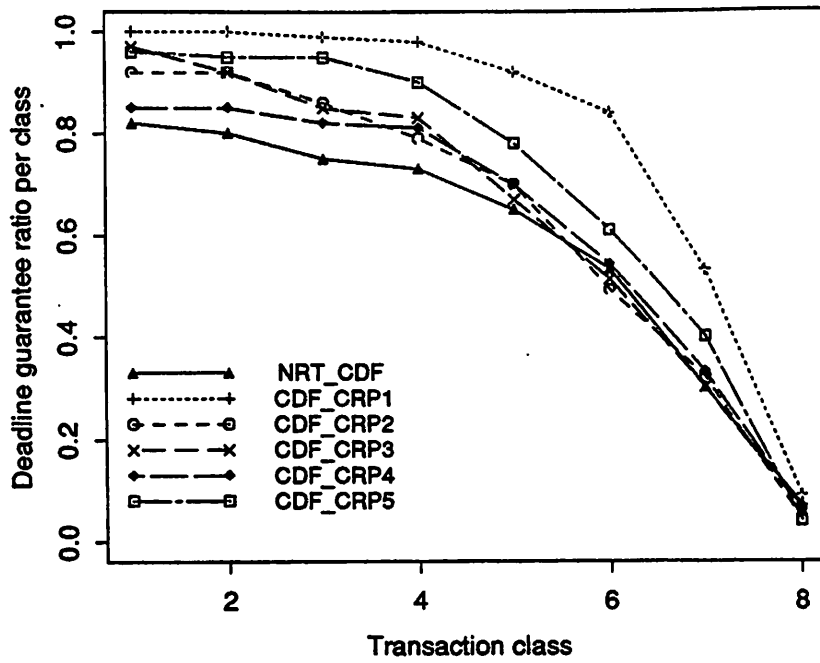


Figure 4.11: Conflict Resolution, $w/r = 8/0, T(16, 4, 10), \alpha = 3$

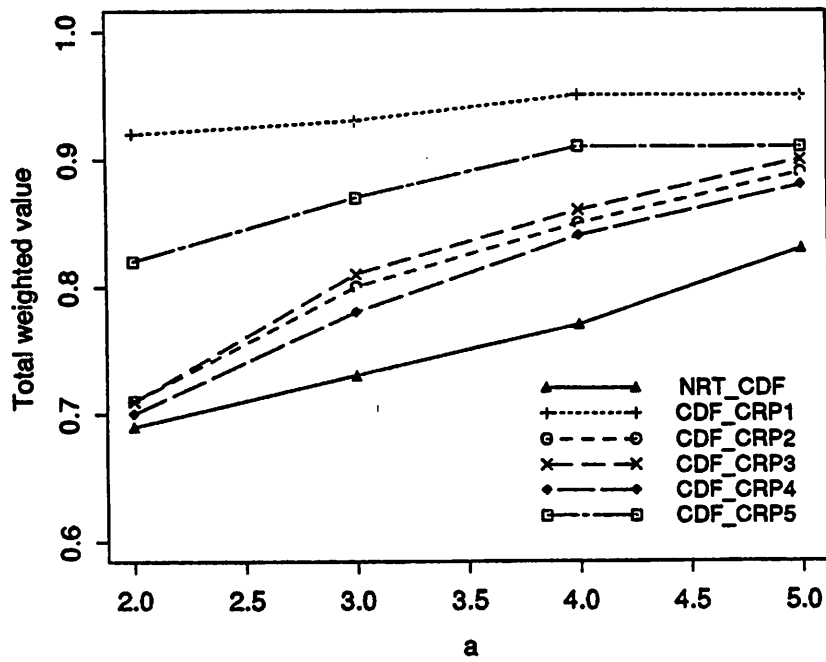


Figure 4.12: Conflict Resolution, $w/r = 8/0, T(16, 4, 10)$

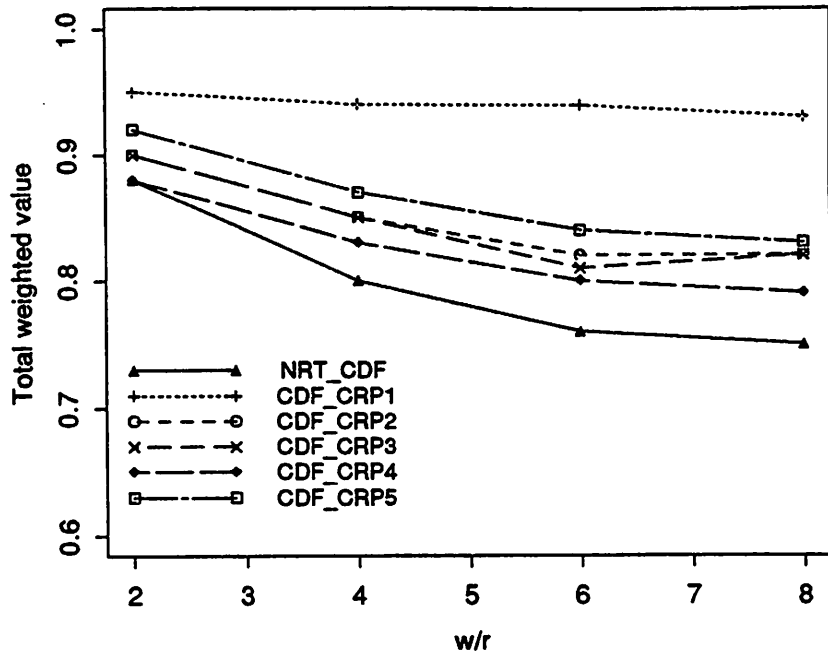


Figure 4.13: Conflict Resolution, $T(16, 4, 10), \alpha = 3$

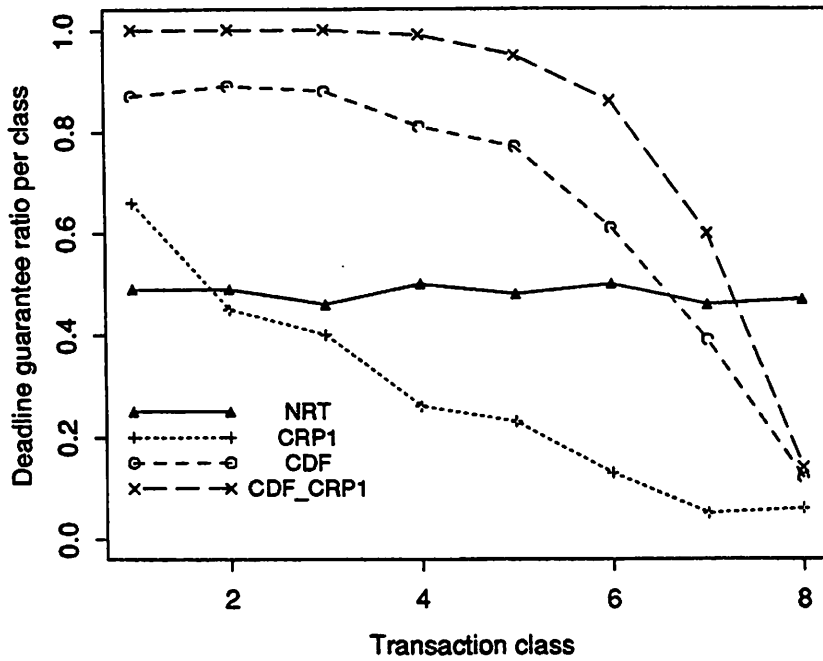


Figure 4.14: CPU Scheduling vs. Conflict Resolution, $w/r = 8/0, T(12, 4, 10), \alpha = 3$

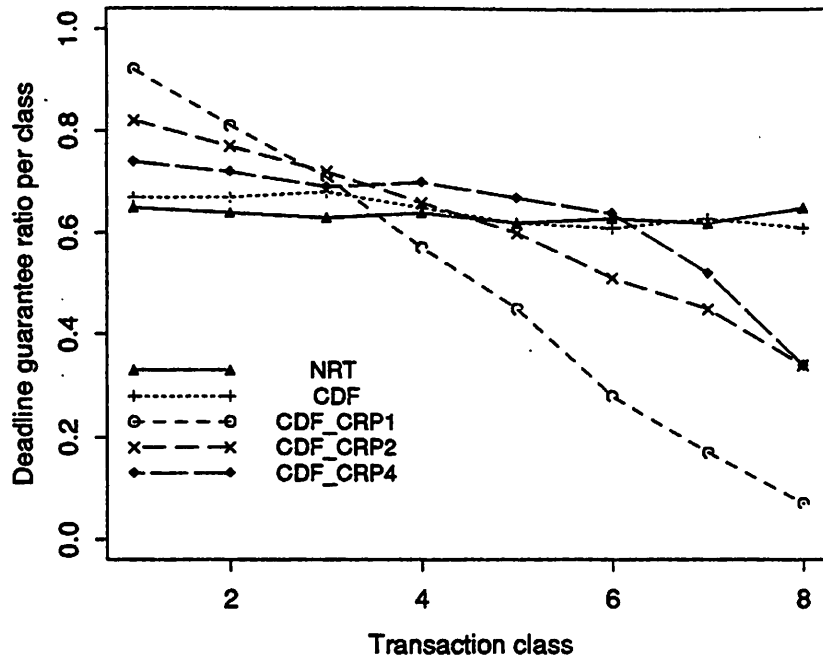


Figure 4.15: I/O Bound System, $w/r = 8/0, T(12, 4, 0), \alpha = 4$

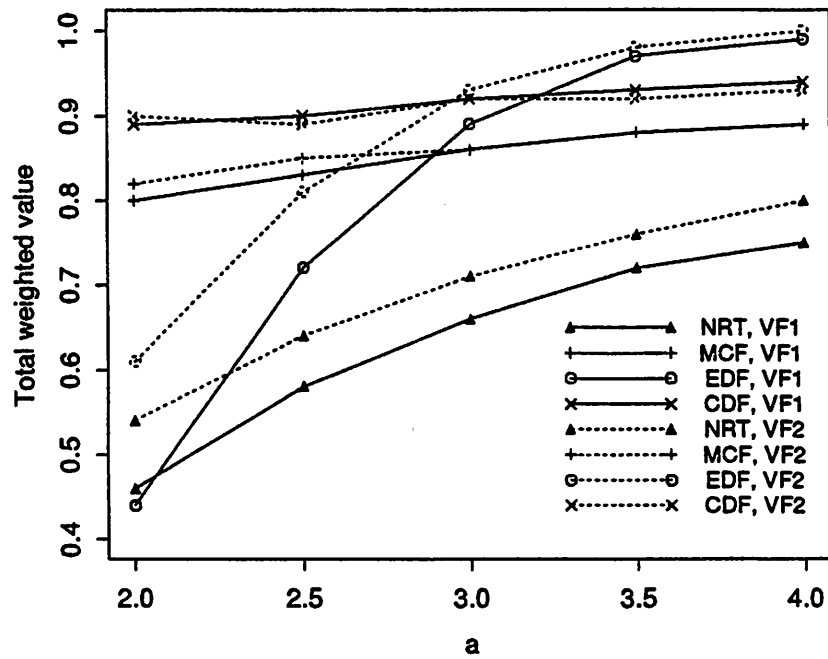


Figure 4.16: Value Functions, $w/r = 2/6, T(12, 4, 10)$

CHAPTER 5

ON USING PRIORITY INHERITANCE IN REAL-TIME DATABASES

In this chapter the interaction between CPU scheduling and concurrency control is investigated in the context of priority inversion. In addressing the priority inversion problem, the priority inheritance mechanism is studied, and is compared with the priority abort mechanism through experimentation. A conditional priority inheritance scheme is developed, which capitalizes on the advantages of both priority inheritance and priority abort schemes.

5.1 Introduction

As discussed in the previous chapters, real-time transaction processing involves extensive interaction between CPU scheduling and concurrency control. However, such interaction may result in a problem, called *priority inversion* [Sha87]. Priority inversion is said to occur when a high priority transaction must wait for a low priority transaction to release a shared resource such as data or critical section. Priority inversion can cause unbounded delay to high priority transactions. This prolonged wait may result in the higher priority transactions missing their deadlines, thus degrading the performance of real-time database systems.

Priority inversion has been studied in real-time systems [Rajk89]. The basic approach proposed to rectify the problem uses the *priority inheritance protocol* [Sha87], where the low priority task will execute at the highest priority of all of the high priority tasks that it blocks, and eventually return to its original priority level after using all of the shared resources. The idea is to allow the low priority task to run and release its resources more quickly so that the higher priority tasks can continue. Performance studies based on the rate-monotonic scheduling framework [Rajk89] have demonstrated that the priority inheritance protocol, applied to the shared resources accessed via semaphores, provides a significant performance advantage.

Unlike real-time systems where task conflict over shared system resources may last for a short period of time, in real-time database systems, transaction conflict over shared data may last as long as the execution time of a transaction. It can last even longer in the case of cascaded blocking. Here we consider two basic schemes for rectifying the problems due to priority inversion, one based on priority inheritance and the other based on *priority abort* (abort the lower priority transaction when priority inversion occurs). We seek answers to the following questions.

- Is the priority inheritance scheme appropriate to solve the priority inversion problem in real-time database systems?
- Which mechanism, priority inheritance or priority abort, is better?
- Is there a better approach other than the two basic schemes?

In this study, we first analyze the implications of using the priority inheritance scheme and the priority abort scheme. Based on the analysis, we then develop a combined priority abort and priority inheritance scheme, called *conditional priority inheritance*, which capitalizes on the advantages of each of the above two schemes.

5.2 Transaction Scheduling Under Two-Phase Locking

5.2.1 The Problem of Priority Inversion

In a real-time database system, the execution of concurrent transactions should be scheduled so as to meet their timing constraints, and at the same time, transaction execution should also be governed by a concurrency control protocol in order to achieve serializability. In this chapter, we examine real-time database systems that employ *priority-driven preemptive scheduling*¹ for CPU scheduling and *two-phase locking* for concurrency control.

For real-time priority-driven preemptive scheduling, each transaction is assigned a priority according to its deadline. The execution of concurrent transactions is

¹Non-preemptive scheduling is not appropriate in a database setting since long transactions and I/O cause unnecessary blocking to other transactions thus preventing them from meeting their deadlines.

scheduled based on their assigned priority. Ideally, a high priority transaction should never be blocked by any lower priority transactions. In particular, a transaction may get CPU service by preempting a lower priority transaction from the CPU. The preempted transaction is placed in the ready queue and resumes its execution when the CPU becomes available later.

Under two-phase locking, on the other hand, a transaction must obtain a lock before accessing a data object and must release the lock when it terminates (commits or aborts). A lock-requesting transaction will be placed in a wait queue if its lock mode is found to be incompatible with that of the lock-holding transaction(s). The queued transaction can proceed only when it is granted the lock.

When the priority-driven preemptive scheduling approach and the two-phase locking protocol are simply integrated together in a real-time database system, *priority inversion* will arise. This problem results from the use of locking for concurrency control. Under the two-phase locking protocol, as we just described, a high priority transaction which requests a lock may conflict with a lower priority transaction which holds the lock. The high priority transaction has to wait for the lock while the lower priority transaction continues. Here transaction wait is necessary in order to ensure data consistency. However, blocking with priority inversion implies that higher priority transactions are waiting while lower priority transactions are in execution. This defeats the purpose behind priority assignment. Even worse, the blocking delay may be unbounded. This can be illustrated by the following two, or combination of the two, situations.

- **Data resource blocking:** Priority inversion can occur due to data access conflicts. For instance, a transaction requesting a *write* lock on a data object may conflict with a group of transactions holding *read* locks on that object. Furthermore, it is possible that the lock holder(s) is waiting for other transactions due to data access conflict. Thus, the blocked high-priority transaction can be delayed for an unbounded period of time.
- **CPU resource blocking:** The blocking of a high priority transaction can be prolonged by *intermediate priority* transactions that have no access conflict with it, but need CPU time. Assume that transaction T_H has higher priority than

T_L and is blocked by T_L due to an access conflict. While T_H is waiting for the lock and T_L is executing, some transaction(s) T_M may arrive, whose assigned priority is intermediate between the priority of T_H and T_L . In this case, T_M will preempt T_L and take over the CPU. Clearly, even if T_M has no access conflict with the higher priority T_H , its execution will delay T_H . Thus, the high priority transaction can be blocked indefinitely due to the execution of the intermediate priority transactions.

Clearly, the blocking resulting from priority inversion is a serious problem to real-time transaction scheduling. In the following, we present four schemes to address the problem.

5.2.2 Priority Inheritance (PI)

Under priority inheritance, when priority inversion occurs, the low priority transaction holding the lock will execute at the priority of the highest priority transaction waiting for the lock, until it terminates. Because of the increase in priority, the lock-holding transaction may run faster than it will without priority change, thus releasing its lock more quickly. As a result, the blocking time for the high priority transaction may be reduced.

The priority inheritance scheme also eliminates the problem of CPU resource blocking. Consider the example that we discussed above. Suppose T_H is blocked by T_L due to data access conflict. Then, using priority inheritance, T_L will execute at the priority of T_H . Now if T_M , an intermediate priority transaction, arrives, it cannot preempt T_L since its priority is less than the inherited priority of T_L . Thus, T_H will not be delayed by T_M .

The priority inheritance scheme provides a significant performance improvement in real-time systems, as shown in [Rajk89]. However, it has shortcomings when used in real-time database systems. First, under strict two-phase locking, a transaction may hold a lock throughout its execution. This "life-time blocking" might be too long for a high priority transaction to wait, even though the lock-holding transaction may execute faster after inheriting a higher priority. The problem will be worse if a high

priority transaction encounters priority inversion with *data resource blocking*, or if the high priority transaction is blocked by low priority transactions many times along the course of its execution, a situation called *chained blocking* [Sha88]. Second, an increase in priority for one transaction may affect other concurrent transactions. In other words, a low priority transaction with an inherited high priority will compete for system resources (CPU, I/O, data objects, critical sections, etc.) with other non-blocking high priority transactions, which might lower the performance of those high priority transactions. Also, due to the increase of competition for data objects, the conflict rate may increase.

As we can see, there are many ramifications when priority inheritance is combined with two-phase locking. On the one hand, the scheme may reduce the duration of priority inversion and eliminate the problem of CPU resource blocking. But on the other hand, the “life-time blocking” nature of two-phase locking may prevent the scheme from being effective.

5.2.3 Priority Abort (PA)

The *priority abort* scheme overcomes the priority inversion problem by aborting the low priority transaction. When a lock-requesting transaction T_1 conflicts with a lock-holding transaction T_2 , T_2 is aborted if T_1 's priority is higher than that of T_2 ; otherwise, T_1 will wait for T_2 .² In this way, a high priority transaction will never be blocked by any lower priority transactions. Therefore, priority inversion is completely eliminated. Note that under the priority abort scheme, the problems of “life-time blocking” and “chained blocking” encountered by priority inheritance scheme do not exist.

The “non-blocking” nature of the priority abort scheme is highly desirable to real-time transaction scheduling, but it may have a negative side effect to the system, i.e., it may lead to a high transaction abort rate due to data access conflict. The higher the abort rate, the more the wasted system resources. This may become a serious problem when a system already contains highly utilized resources.

²If T_1 conflicts with more than one transaction, T_1 will abort the conflicting transactions only if its priority is higher than that of all the conflicting transactions.

Each of the two schemes, PI and PA, for addressing the priority inversion problem has its advantages and disadvantages. The main trade-off between the two schemes is the potential long blocking time versus the high conflict abort rate. In order to minimize the effects of these two problems, we propose a combined priority inheritance and priority abort scheme, called *conditional priority inheritance*.

5.2.4 Conditional Priority Inheritance (CP)

The basic idea behind this scheme is the following. When priority inversion is detected, if the low priority transaction is near completion, it inherits the priority of the high priority transaction, thus reducing high abort rate and wasting few resources; otherwise, the low priority transaction is aborted, thereby avoiding the long blocking time for the high priority transaction, and also reducing the amount of wasted resources used thus far by the low priority transaction.

Here we assume that transaction length (defined as the number of *steps*) is known in advance. This assumption is justified by the fact that in many application environments like banking and inventory management, the transaction length, i.e., the number of records to be accessed and the number of computation steps, is likely be known in advance. (Otherwise, it might be an estimate.) Let L_T be the length of transaction T , x_T be the number of steps that T has executed thus far, and p_T be T 's priority. Then, our conditional priority inheritance scheme can be described by the following algorithm.

```

 $T_1$  requests a data object being locked by  $T_2$  with an incompatible lock mode.
if  $p_{T_1} \leq p_{T_2}$ 
  then  $T_1$  waits for  $T_2$ 
  else
    if  $(L_{T_2} - x_{T_2}) > h$ 
      then abort  $T_2$ 
      else  $T_2$  inherits  $T_1$ 's priority
    end if
  end if

```

In the algorithm, h is a threshold. For the lock-holding transaction with a lower priority, if its remaining work ($L_T - x_T$) is less than h , then we apply PI; otherwise we

use PA. This threshold policy, CP, is expected to reduce blocking time with respect to PI, and to reduce the abort rate with respect to PA. We define the *priority inheritance period* to be the time interval in which the priority inheritance scheme is used rather than priority abort, i.e., the last h steps of the transaction.

The principal question regarding CP is the choice of threshold value, h . If h is too small, CP may behave like PA. On the other hand, if h is too large, CP may behave like PI. In addition, the setting of h should take into account the distribution of data conflicts with respect to transaction length. The sensitivity of h on performance has been studied through experiments and the results are discussed in Section 5.4.

5.2.5 Priority Ceiling Protocol

The *priority ceiling protocol* [Sha88, Chen90, Son90] is another scheme developed to solve the priority inversion problem. Under this scheme, the priority inversion is bounded to no more than one transaction execution time. The scheme also has the property of deadlock freedom. However, this scheme requires prior knowledge about data objects to be accessed by each transaction. This condition appears to be too restrictive in many applications. Moreover, the scheme becomes extremely conservative, with respect to the degree of parallelism, if transactions can access any data objects in the database. Owing to these reasons, we do not consider this particular scheme in this study.

In summary, the presented three schemes solve the priority inversion problem based on either one or both of the two basic mechanisms, namely, priority inheritance and priority abort. The trade-off among the schemes is blocking time versus resource consumption.

5.3 Test Environment

The three schemes, PI, PA and CP, for addressing the priority inversion problem have been implemented and evaluated on RT-CARAT. In this section, we briefly describe the system parameter settings and the performance baselines and metrics.

Table 5.1: Experimental Settings

Parameter	Settings
<i>MPL</i> (multiprogramming level)	8
<i>DB-Size</i> (database size)	1000 blocks (6000 records)
<i>GB-Size</i> (global buffer size)	0 blocks
<i>AD</i> (access distribution)	uniform
<i>y</i> (records accessed per trans. step)	4 records
<i>u</i> (computation per trans. step)	0
system type	I/O bound, CPU bound
<i>x</i> (steps per transaction)	4 - 16 steps
P_w (prob. of write transactions)	0.2 - 1.0
α (deadline window factor)	2.0 - 6.0
<i>h</i> (threshold parameter)	0 - 16 steps

5.3.1 Parameter Settings

Table 5.1 summarizes the parameter settings in the experiments. The table is divided into two parts. The first part presents the parameters that are kept constant across all workloads, and the second part are those that change according to the different performance measurements. In all the experiments, the multi-programming level in the system is 8. The database consists of 1000 physical blocks (6,000 records). The global buffer is not employed in this study. Data accesses are uniform across the entire database. y and u are simply fixed at 4 and 0. Thereby, the transaction length, $T(x, y, u)$, can be varied by one parameter x .

The experiments were conducted in two different types of systems. One is I/O bound (a VAXstation 3100/M38 with two RZ55 disks) and the other is CPU bound (a VAXstation II/GPX with two RD53 disks). The purpose is to see how different kinds of resource contention will affect protocol performance. A critical factor in this performance study is transaction length $T(x, y, u)$, since it is directly related to transaction blocking time. As mentioned above, we vary x , while fixing y and u . P_w , the probability of write transactions among the concurrent ones, is another parameter we are interested in, since it directly affects transaction conflict rate and, hence, the chance of priority inversion. In addition, the deadline window factor, α , is a timing-related parameter which specifies the deadline distribution of real-time

Table 5.2: Policies Examined

Policy	CPU scheduling	Conflict resolution
NRT	multi-level feedback queue	always wait
WAIT	earliest deadline first	always wait
PI	earliest deadline first	priority inheritance
PA	earliest deadline first	priority abort
CP	earliest deadline first	conditional PI/PA

transactions. The smaller the value of α , the tighter the transaction deadlines are and vice versa. Moreover, to study the conflict resolution scheme CP, we vary h , the threshold parameter.

5.3.2 Performance Baselines and Metrics

Table 5.2 lists the policies we examined in this performance study. Here each policy is a combination of a CPU scheduling scheme and a conflict resolution scheme. The CPU scheduling scheme is either priority-driven or non priority-driven (*multi-level feedback queue*). The former represents the nature of CPU scheduling in real-time databases, while the latter in traditional databases. Transaction priority can be assigned based on various transaction parameters, such as deadline, criticalness (i.e., the degree of importance), and length. Since the study of priority inheritance scheme is considered to be orthogonal to priority assignment policies and also meeting transaction deadline is the main concern on protocol performance in this study, we employ the *earliest-deadline-first* (EDF) policy for transaction priority assignment. For studies on other priority assignment policies based on transaction slack time, criticalness, or length, the reader is referred to Chapters 4 and 7 and to [Abbo88b, Chn90].

Besides PI, PA and CP discussed in Section 5.2, we also look at two other policies, NRT and WAIT, for the sake of performance comparisons. Representing a non real-time transaction processing system, NRT is the performance baseline in these experiments. Under NRT, transactions are scheduled by a *multi-level feedback queue* policy, and in case of data access conflict the lock-requesting transaction is always

placed in a FIFO wait queue. In other words, transaction timing information is used neither in CPU scheduling nor in conflict resolution under NRT.

Unlike NRT, WAIT uses priority-driven CPU scheduling, based on EDF. However, it does not take transaction priority into account for conflict resolution: It is always the lock-requesting transaction that will be placed into a wait queue. This policy enables us to isolate the performance differences due to the use of PI, PA, or CP.

In the experiments we use the following metrics for performance evaluation.

- Deadline guarantee ratio - the percentage of submitted transactions that complete by their deadlines.
- Priority inversions per run - the average number of instances, for each run, that a lock-requesting transaction is blocked by lower priority transaction(s).
- Data resource blocking - the average number of (lower priority) transactions blocking a lock-requesting transaction. (See Section 5.2.)
- PD abort ratio - the total number of priority aborts and deadlock aborts, divided by the number of submitted transactions.
- Number of waits per run - the average number of lock-waiting instances that a transaction encountered at each run, including blocking instances caused by priority inversion.
- Waiting time - the average waiting time (in seconds) in each wait instance.
- Total waiting time per run - *waiting time times number of waits per run* (in seconds).
- Wasted operations per transaction - the average total number of transaction steps wasted due to priority abort or deadlock abort for each submitted transaction.

We also collect statistics on CPU utilization, I/O utilization, and transaction restart ratio.

The data collection in the experiments is based in the method of replication. The statistical data has 95% confidence intervals with less than $\pm 2\%$ of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

5.4 Experimental Results

In this section, we present performance results for the experiments conducted on the RT-CARAT testbed. The schemes for addressing priority inversion are evaluated in a four dimensional test space defined by *resource contention* (I/O or CPU bound), *transaction length*, *data contention*, and *deadline distribution*. In addition, we examine the effect of varying h on the performance of the CP scheme. In the following, we first present 4 sets of experiments carried out in an I/O bound system (CPU utilization = 65% and I/O utilization = 95%, on average), and then 1 set of experiments in a CPU bound system (CPU utilization = 92% and I/O utilization = 55%, on average).

5.4.1 Data Contention

In this set of experiments, we vary P_w , the probability of write transactions, so as to vary the level of data contention in the system. Transactions are all equal in length with $x = 6$ steps, deadline window factor α is fixed at 4, and the threshold parameter h is set at 2 (steps).

Figure 5.1 plots the *transaction deadline guarantee ratio* versus P_w for policies NRT, WAIT, PI, PA and CP, respectively. As one would expect, the deadline guarantee ratio drops as data contention increases. Among the five policies, CP performs the best, especially when data contention becomes high. PA works very well for low data contention. However, compared to CP, its performance degrades as P_w increases. PI and WAIT perform basically the same with PI being slightly better than WAIT when P_w is small. The deadline guarantee ratio under NRT is the lowest, because it does not make use of any transaction information such as deadline and transaction length.

As we discussed in Section 5.3, the performance of these schemes is affected by several factors. In the following, we explain the results shown in Figure 5.1 by analyzing the performance with respect to priority inversion, blocking time, abort rate, and resource utilization.

Figure 5.2 shows the average number of *priority inversions per transaction run* for WAIT, PI and CP, respectively. PI reduces the number of priority inversions compared to the WAIT scheme. CP achieves much larger reduction in the number of priority inversions by conditionally aborting the lower priority transaction(s). Note that for all the schemes, the average number of priority inversions per transaction run is small (even for long transactions). Therefore, *chained blocking* [Sha87] does not happen frequently in RT-CARAT.

Figure 5.3 demonstrates the situation of *data resource blocking*. As we can see, when priority inversion occurs, the average number of blocking transactions with lower priority is just slightly more than one and basically does not change with P_w .

The average number of *transaction wait instances per run* and the average *waiting time per wait instance* are illustrated in Figures 5.4 and 5.5, respectively. As data contention increases, both the average number of wait instances and average waiting time increase. Comparing the five policies, PA has the lowest number of waits and the shortest waiting time. Note that under PA, priority inversion hardly occurs. Thus, PA results in the minimal waiting for real-time transactions. From these Figures we also note that priority inheritance, PI, does reduce transaction waiting time with respect to WAIT policy by about 8% for ($P_w = 0.2$). CP further reduces the waiting time over PI, but is still no better than the pure abort scheme PA.

Figure 5.6 plots *PD abort ratio*. At one extreme, PA has the highest abort ratio. At the other extreme, NRT, WAIT and PI result in the lowest abort ratio. CP falls in-between. This is understandable since PA relies on transaction abort, while NRT, WAIT and PI are based on a wait mechanism. The proposed conditional priority inheritance scheme combines both abort and wait strategies. Hence, the abort ratio under CP is higher than NRT, WAIT and PI, but lower than PA. It is important to mention that there is (almost) no deadlock abort under PA. This is because PA enforces a total ordering by priority among the concurrent transactions (except the

situation where a high priority transaction conflicts with a group of transactions, some of which are at the lower priority). For CP, on average, the ratio of priority abort and deadlock abort is 7:1.

To see the negative effect of transaction abort, we plot the *wasted operations per transaction* in Figure 5.7. Clearly, due to the high abort rate, PA wastes much more work than other schemes. The conditional abort policy CP wastes much fewer operations than PA and nearly the same as NRT, WAIT and PI. Figure 5.8 shows the *CPU and I/O utilizations*, respectively. As one can expect, PA consumes more CPU and I/O resources than any other scheme.

Our observations and discussions in this set of experiments lead to the following points:

- Applying priority inheritance does reduce transaction blocking time. However, PI, the basic priority inheritance scheme, does not provide significant performance improvement over the WAIT scheme. PI performs even worse than WAIT when data contention is high.
- PA and CP perform comparably, in terms of transaction deadline guarantee ratio, when data contention is low. CP works better than PA when data contention becomes high.
- Blocking resulting from priority inversion (including the period of priority inheritance) is a more serious problem than wasting system resources. PA and CP, which attempt to eliminate or reduce transaction blocking, perform better than WAIT and PI, which attempt to reduce resource waste.
- Chained blocking and data resource blocking are not frequent and are negligible under PA and CP.

5.4.2 Sensitivity of Threshold (h) Settings

The proposed CP scheme employs a *threshold* policy where the decision to use priority inheritance or priority abort depends on the settings of h . In the above

experiments, h is fixed at 2 (steps). In this set of experiments, we further study the performance of CP by varying the threshold parameter h .

Figures 5.9, 5.10 and 5.11 depict the total waiting time per transaction run, PD abort ratio, and transaction deadline guarantee ratio, respectively, for the workload with $x = 6$, $P_w = 0.6$, and $\alpha = 4$. To see the relation between CP and PI and PA, we also show the performance of PI and PA, even though they are independent of h . In the figures, CP performs the same as PA when h is equal to 0, and the same as PI when h is equal to 6. This is exactly how CP should behave according to the algorithm described in Section 5.2. As h changes from 0 to 6, the total waiting time increases while wasted operations decrease. Clearly, due to its threshold policy - switching between priority abort and priority inheritance based on h setting, CP is bounded by PA and PI in terms of total waiting time and wasted operations. With respect to deadline guarantee ratio (see Figure 5.11), however, CP performs the best for $1 \leq h \leq 3$.

We have also exercised workloads with longer transactions and have observed a similar performance trend to that observed for $x = 6$, with the effective range of h being extended from $1 \leq h \leq 3$ for $x = 6$ to $1 \leq h \leq 6$ for both $x = 12$ and $x = 16$. The performance of CP indicates that, in general, the priority inheritance strategy only works well over a certain range. In other words, the priority inheritance period of a lock-holding transaction cannot be too long (i.e., h should be small with respect to L_T); otherwise, the resulting long blocking time will degrade the performance of other higher priority transactions. We will further explain this result in the following sections.

5.4.3 Deadline Distribution

In this experiment, we examine the schemes along another dimension of our test space, i.e., deadline distribution. We vary the deadline window factor α so as to change the tightness of transaction deadlines.

Figure 5.12 shows the transaction deadline guarantee ratio for the workload with $x = 6$, $P_w = 0.6$, and $h = 2$. Unlike the results we have shown above where CP performs the best with $\alpha = 4$, here PA becomes the best when transaction deadlines

are loose ($\alpha > 4.7$). This is because under PA, a high priority transaction (almost) never waits for a lower priority transaction. With dynamic EDF scheduling policy, a transaction restarted due to conflict abort may get a higher priority, and eventually complete its execution. This is true as long as the deadline of a transaction is long enough to allow it to be (repeatedly) aborted and restarted. Unlike PA, CP¹ may have a high priority transaction wait for lower priority transactions when its conditional priority inheritance is applied. The experimental result implies that when transaction deadlines are loose, it is better for high priority transactions to proceed by aborting lower priority transactions than to wait by applying priority inheritance to lower priority transactions.

Comparing PI and WAIT, one can see the intersection of their performance curves. When transaction deadlines are tight, WAIT performs slightly better than PI. But when deadlines are loose, PI works better than WAIT. This is due to the fact that PI raises the process priority of the lower priority transaction which holds the lock. This operation will increase the actual degree of process parallelism in the system. With a higher priority to compete for system resources, the priority inheriting transaction may degrade the performance of other concurrent transactions. This becomes true when transaction deadlines are tight. However, the concurrent transactions can withstand this negative effect when deadlines are loose.

We can see from these results that all of the schemes are sensitive to deadline distribution. Overall, the non-blocking scheme based on priority abort, like PA (and CP, which has a limited blocking duration), performs far better than wait oriented schemes, like WAIT and PI, as long as transaction deadlines are relatively loose.

5.4.4 Transaction Length

All of the results shown above are for transactions with $x = 6$. We now vary transaction length x from 4 steps to 16 steps, with P_w fixed at 0.2, α at 4, and h at 2. Figure 5.13 plots the transaction deadline guarantee ratio for the five different schemes. For short transactions ($x = 4$), the access conflict rate is low (less than 10%). In this case, the particular scheme used to avoid priority inversion has no significant impact. Since WAIT, PI, PA and CP use the same scheduling policy,

namely EDF, their performance is quite close. Although there is little difference between the schemes, their relative order is the same as in the earlier experiments. Here PI is slightly better than WAIT, and PA is slightly better than CP. These are the results that we obtained in the earlier experiments (see Figure 5.1 for $P_w = 0.2$).

As transaction length increases, on the other hand, the performance difference among the four schemes increases. In addition, the differences between CP and PA and between PI and WAIT are reversed. Now PA performs better than CP and WAIT performs better than PI. Note that for a fixed P_w value, varying the transaction length also changes the access conflict rate among the concurrent transactions. Hence, we need to isolate the two factors for any further analysis.

To examine the performance for long transactions, we fix transaction length to $x = 16$ while varying P_w from 0.05 to 0.20. Figure 5.14 shows the deadline guarantee ratio for such workloads. Comparing Figure 5.14 with Figure 5.1, we can see that for long transactions, the schemes making use of priority abort, namely PA and CP, perform much better than wait-based schemes PI and WAIT. Here the reason is that under PI and WAIT, access conflict over long transactions leads to longer waiting time. In other words, the *life-time blocking* becomes a severe problem as transactions become long.

Because of the problem of life-time blocking, the priority inheritance scheme does not work well. As one observes from Figure 5.14, PI performs even worse than WAIT, regardless of the degree of data contention. This is due to the fact that priority inheritance increases the degree of process parallelism. As a result, PI causes a higher deadlock abort rate than WAIT for long transactions (the results are not plotted here). At this point, it is important to compare PI with CP, which also employs a priority inheritance scheme, but on the basis of transaction execution length. Here h is set to 2 (steps) for CP. That means the priority inheritance period under CP is only one eighth of that under PI. Without the life-time blocking problem, CP works well for long transactions, and it outperforms PA as data contention becomes high.

We have also exercised workloads with a mix of different transaction lengths. Here we show one set of the experimental results for such mixed workloads. Figure 5.15 depicts the average deadline guarantee ratio of two lengths of transactions, one

being 4 steps ($x = 4$) and the other 8 steps ($x = 8$) with mean value 6 (i.e., $P[x = 4] = P[x = 8] = 1/2$). Comparing Figure 5.15 with Figure 5.12 which show the performance of transactions with equal length $x = 6$, we can see that the behavior of the five policies and their relative performance under the two different workloads are basically the same. We have also observed the similar performance results for individual transactions with $x = 4$ and with $x = 8$, respectively. In addition, we note that the deadline guarantee ratios of different length transactions are different, with shorter transactions having higher deadline guarantee ratio than longer transactions on average. This phenomenon, called *transaction starvation*, relates to the issue of scheduling fairness and is out of the scope of this study. The reader is referred to Chapter 7 for details.

5.4.5 CPU Bound System

As we described at the beginning of this section, the schemes for addressing priority inversion are evaluated in a four-dimensional test space. In the experiments demonstrated above, the performance evaluation was carried out by varying *data contention*, *deadline distribution* and *transaction length*, while fixing the *resource contention* in an I/O bound system. We also examined a CPU bound system, where average CPU and I/O utilizations were 92% and 55%, respectively. Due to the similarity of the experiments, we only illustrate one set of the performance results from such a CPU bound system.

Figure 5.16 shows the deadline guarantee ratio versus transaction length for the workload with $P_w = 0.2$, $\alpha = 4$ and $h = 2$. Our first observation is that PA and CP, which are based on priority abort, perform better than WAIT and PI, which are based on a wait mechanism, especially for long transactions. This result is consistent with what we have obtained in an I/O bound system. Secondly, the reader can see that the priority inheritance scheme works well only for short transactions and it performs worse than WAIT for long transactions. This result is also the same as the one we obtained from the I/O bound system.

It is interesting to compare Figure 5.16, the results obtained from the CPU bound system, with Figure 5.13, the results from the I/O bound system. We observe that

the deadline guarantee ratio of WAIT, PI, PA and CP in the CPU bound system is higher than in the I/O bound system. This is because CPU scheduling plays a more important role in CPU bound systems than in I/O bound systems (see Chapter 4). We also observe that the performance difference between PA/CP and WAIT/PI is larger in the CPU bound system than in the I/O bound system. This is understandable since CPU scheduling will make non-blocking (or less-blocking) conflict resolution schemes work better in a CPU bound system than in a I/O bound system.

These results suggest that in a CPU bound system we may simply incorporate CPU scheduling with a non-blocking resolution scheme, like PA. In other words, it is most important to eliminate priority inversion when CPU contention exists.

5.5 Conclusions

We have studied several schemes for addressing the priority inversion problem in a real-time database environment where two-phase locking is employed for concurrency control. We first examined two basic schemes, one based on priority inheritance (PI) and the other on priority abort (PA). Based on the analysis, we proposed a combined priority inheritance and priority abort scheme, called conditional priority inheritance (CP). The three schemes, plus two performance baselines NRT and WAIT, have been implemented and evaluated on a real-time database testbed. Our performance studies indicate that with respect to deadline guarantee ratio, the basic priority inheritance scheme does not work well. Rather, the conditional priority inheritance scheme and the priority abort scheme perform well for a wide range of system workloads.

We have clarified through experiments that the priority inheritance scheme is sensitive to the priority inheritance period. A long priority inheritance period (*life-time blocking*) will affect not only the blocked higher priority transactions but also other concurrent non-blocked higher priority transactions. It is the life-time blocking that makes the basic priority inheritance scheme infeasible in resolving priority inversion in real-time database systems. On the other hand, the proposed conditional priority inheritance scheme works well because of its reduced priority inheritance period. From this result, we hypothesize that the basic priority inheritance approach will also not work well in real-time systems where the priority inheritance period is long.

Besides the problem of life-time blocking, it has also been found that the basic priority inheritance scheme has another shortcoming when used in real-time database systems. Applying priority inheritance may increase the degree of actual process concurrency. We have observed that this side effect causes a higher deadlock rate, especially with long transactions.

As a result of our comparison of the abort-oriented scheme, PA and CP, and the wait-oriented scheme, WAIT and PI, we have identified that blocking resulting from priority inversion is a more serious problem than wasting of system resources. This is especially true when transaction deadlines are loose or when a system is CPU bound³. In addition, PA is sensitive to data contention, particularly in I/O bound systems. In such an environment, CP, which wastes less resources and yet incurs less blocking (with shorter blocking time), should be used to achieve better performance.

Knowing the performance consequence of priority inheritance, the scheme can be applied to real-time databases depending upon the priority inheritance period. This is not limited to applications where transaction length is known (thereby the h value can be measured). It may be used, for example, in real-time database systems which employ optimistic concurrency control (OCC) (see Chapter 7). Since the validation phase, the last stage of transaction execution, is rather short compared to transaction length, priority inheritance may be applied to the validating transaction which blocks a higher priority transaction.

We may further extend this work by incorporating real-time I/O scheduling [Care89, Abbo90, Chn90, Jau90]. It would be interesting to re-examine the various schemes studied in this chapter in a system that integrates both priority-based CPU scheduling and real-time I/O scheduling.

³These results are similar to what we obtained in our studies on real-time optimistic concurrency control scheme (OCC) in Chapter 7, where we show that the abort-oriented OCC performs better than wait-oriented two-phase locking approach.

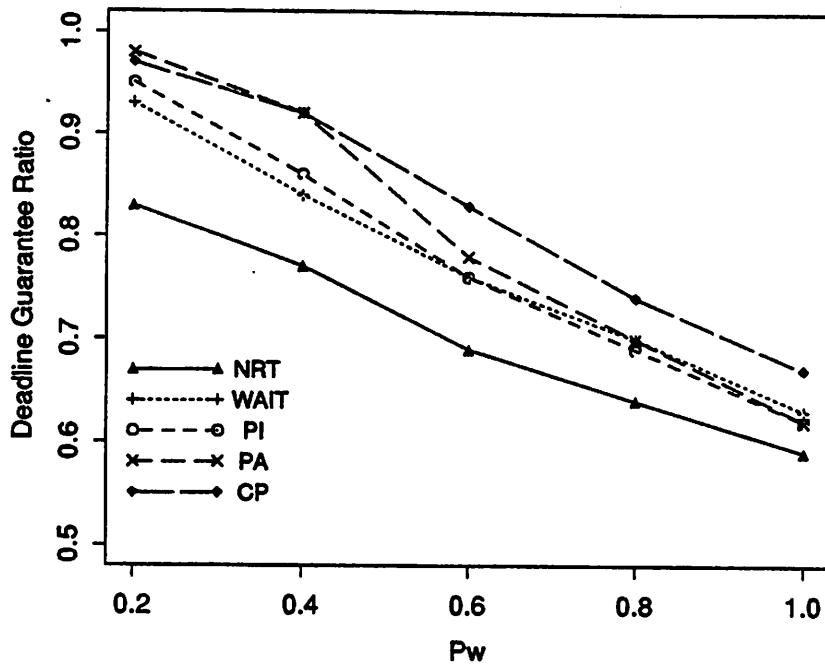


Figure 5.1: Data Contention, $x = 6, \alpha = 4, h = 2$

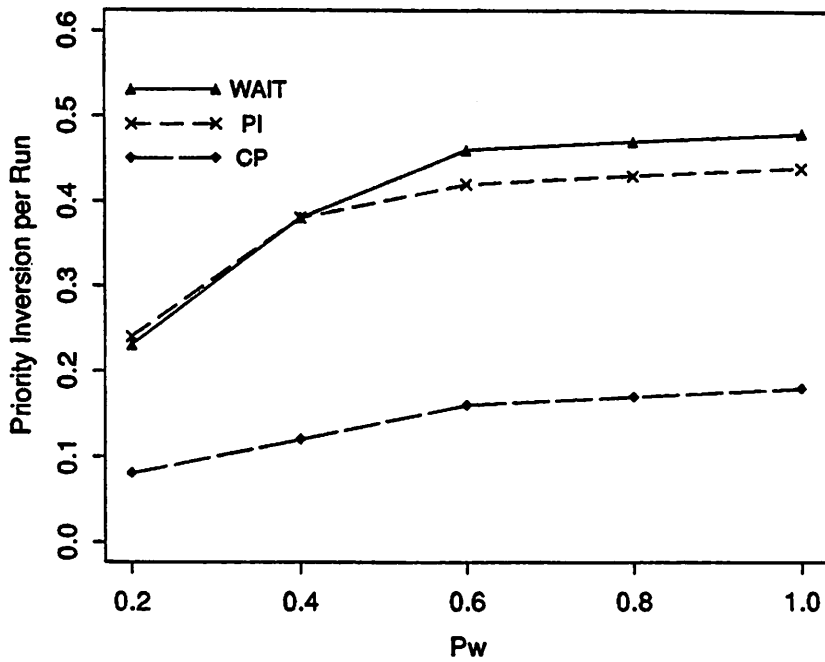


Figure 5.2: Data Contention, $x = 6, \alpha = 4, h = 2$

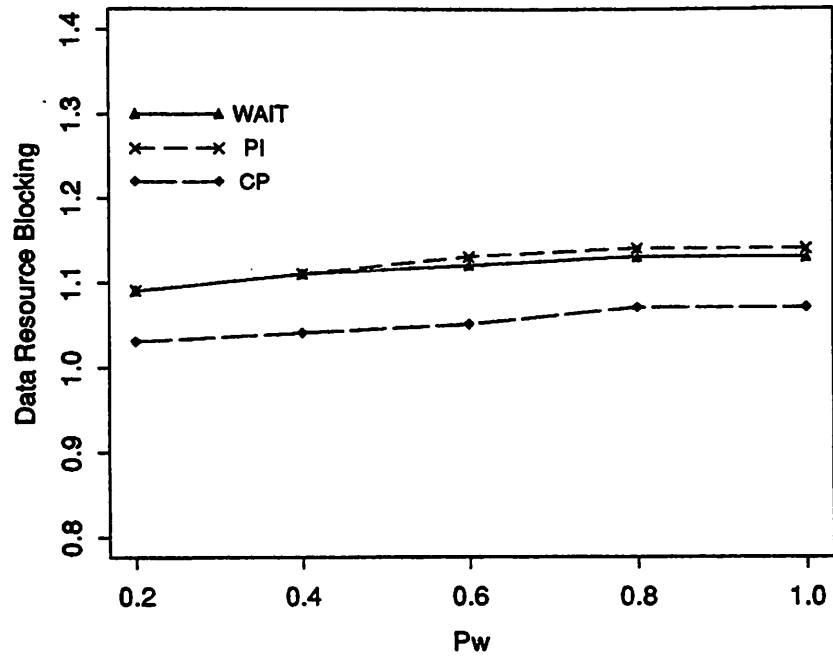


Figure 5.3: Data Contention, $x = 6, \alpha = 4, h = 2$

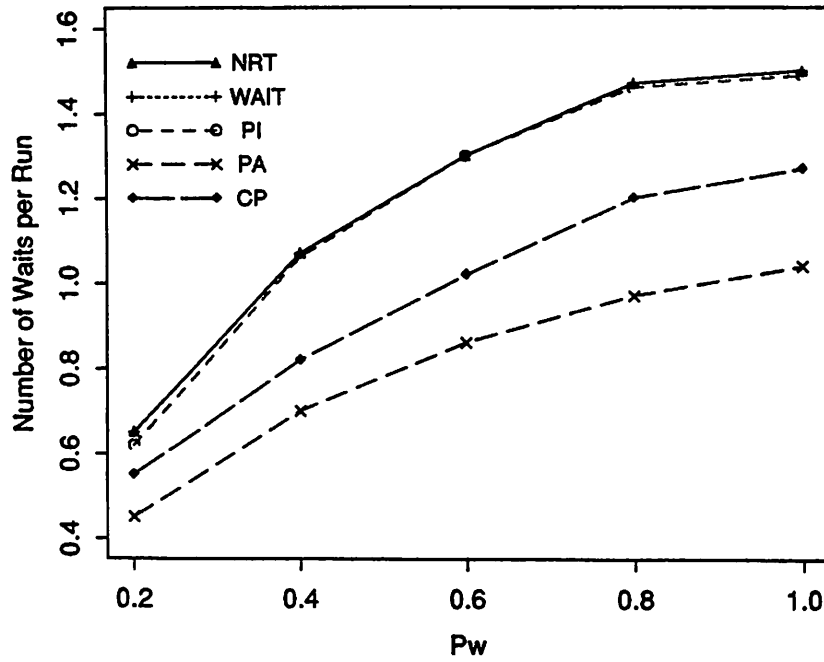


Figure 5.4: Data Contention, $x = 6, \alpha = 4, h = 2$

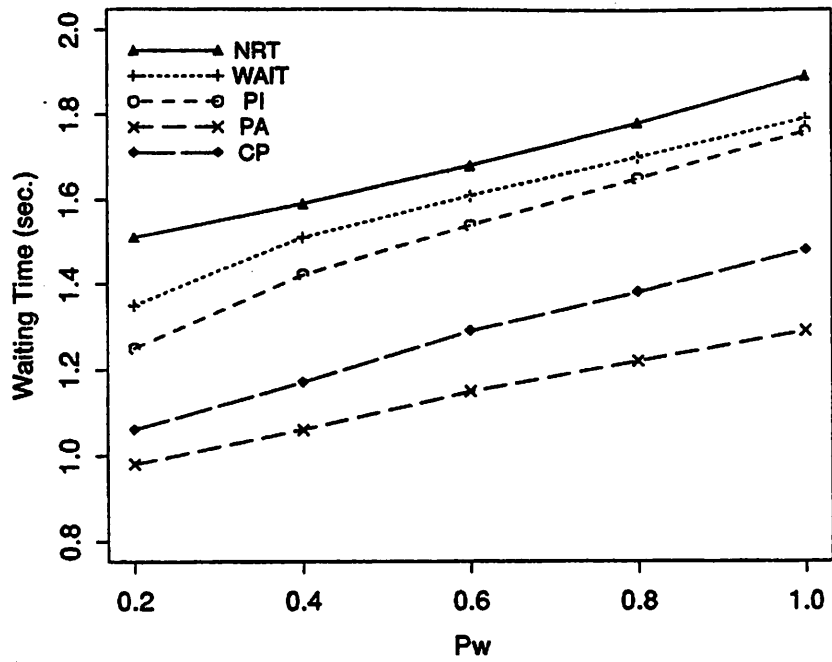


Figure 5.5: Data Contention, $x = 6, \alpha = 4, h = 2$

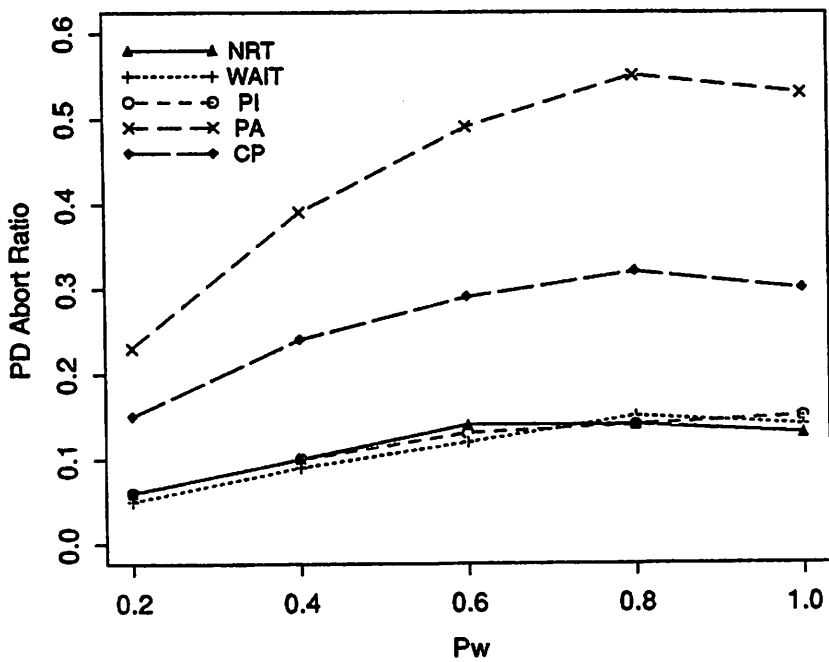


Figure 5.6: Data Contention, $x = 6, \alpha = 4, h = 2$

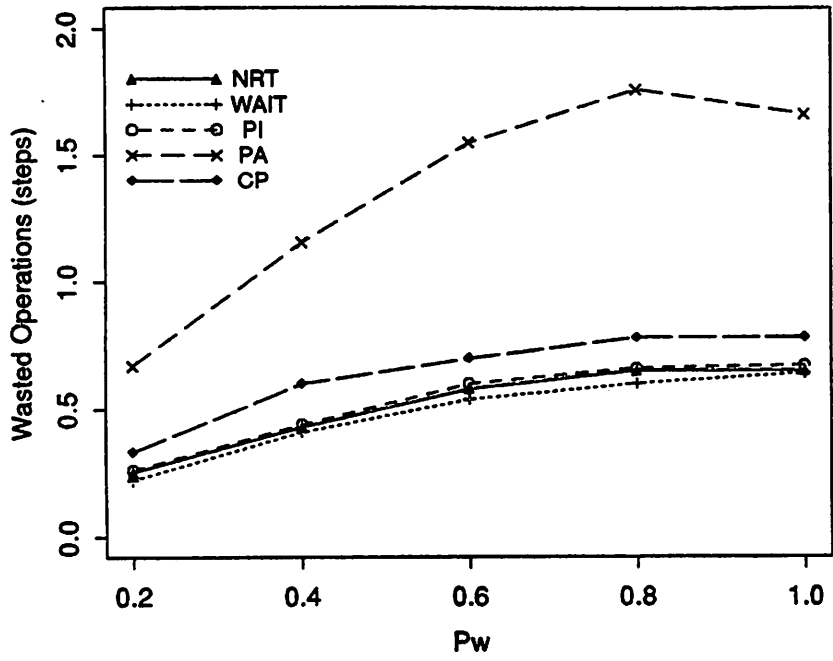


Figure 5.7: Data Contention, $x = 6, \alpha = 4, h = 2$

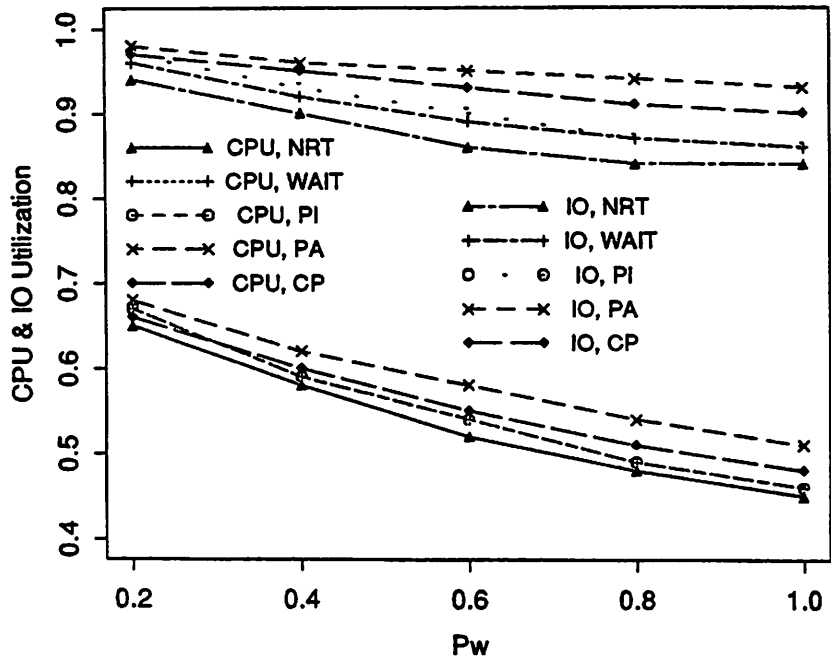


Figure 5.8: Data Contention, $x = 6, \alpha = 4, h = 2$

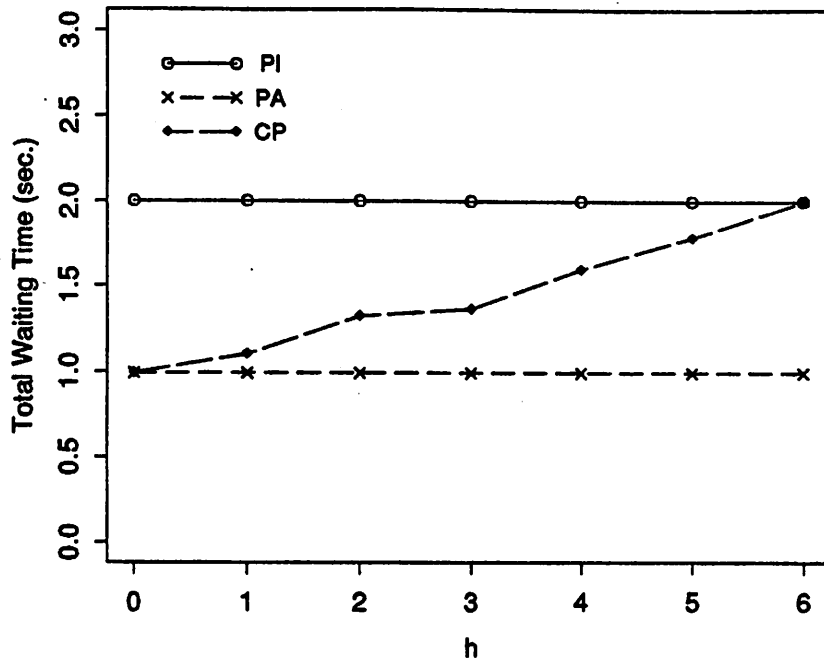


Figure 5.9: Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$

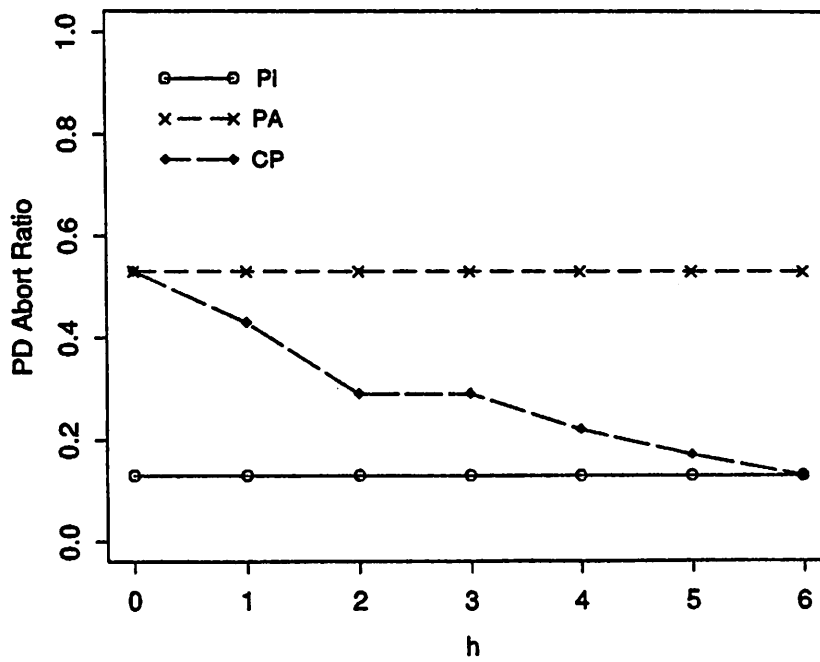


Figure 5.10: Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$

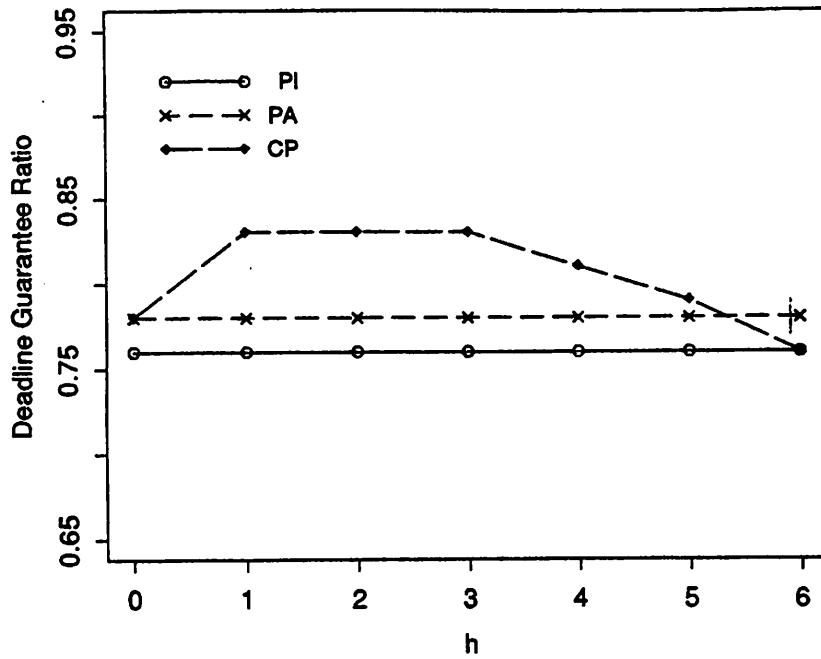


Figure 5.11: Sensitivity of Threshold, $x = 6, P_w = 0.6, \alpha = 4$

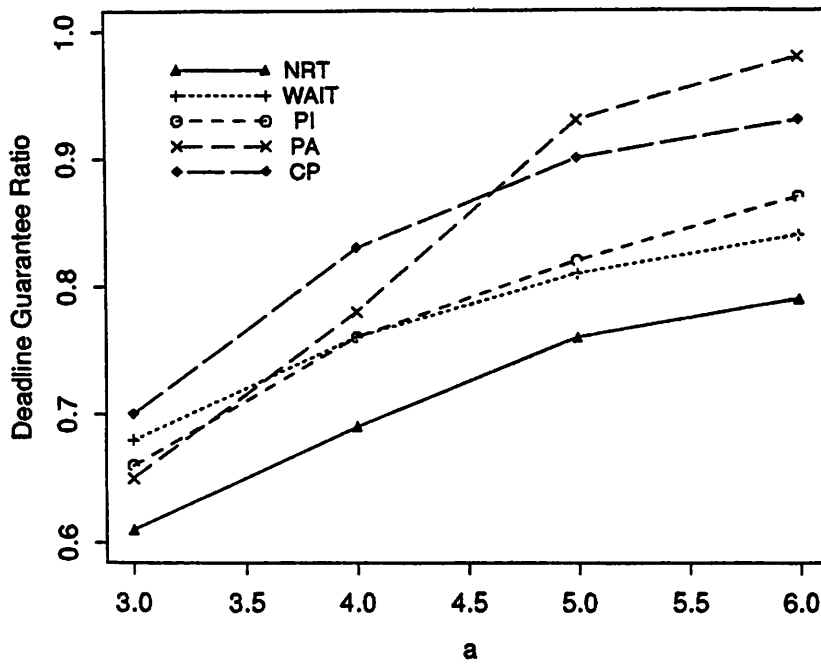


Figure 5.12: Deadline Distribution, $x = 6, P_w = 0.6, h = 2$

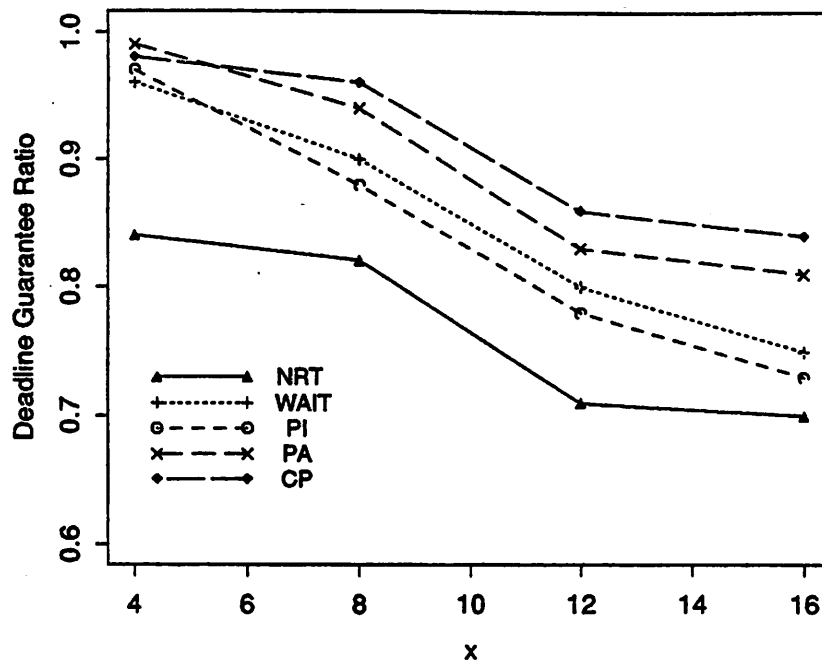


Figure 5.13: Transaction Length, $P_w = 0.2, \alpha = 4, h = 2$

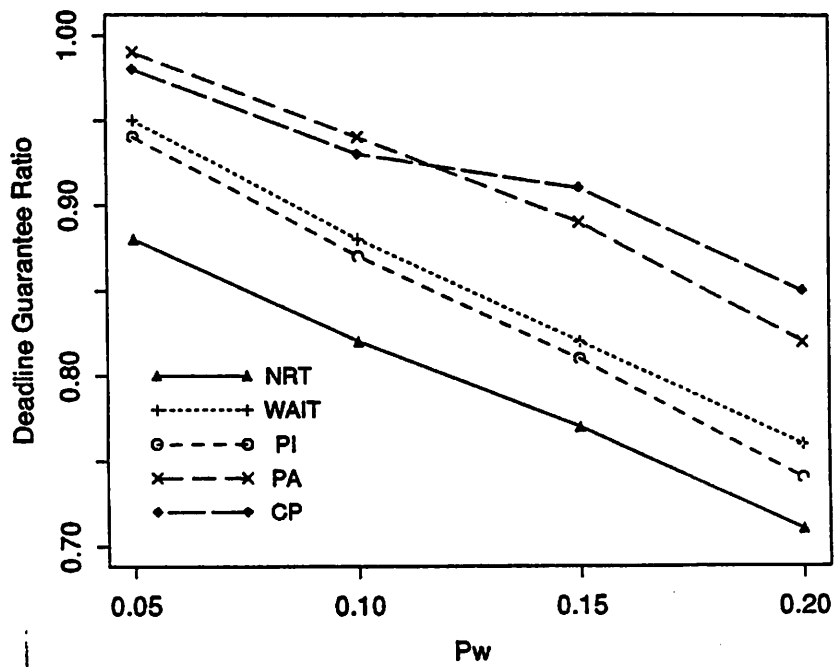


Figure 5.14: Long Transactions, $x = 16, \alpha = 4, h = 2$

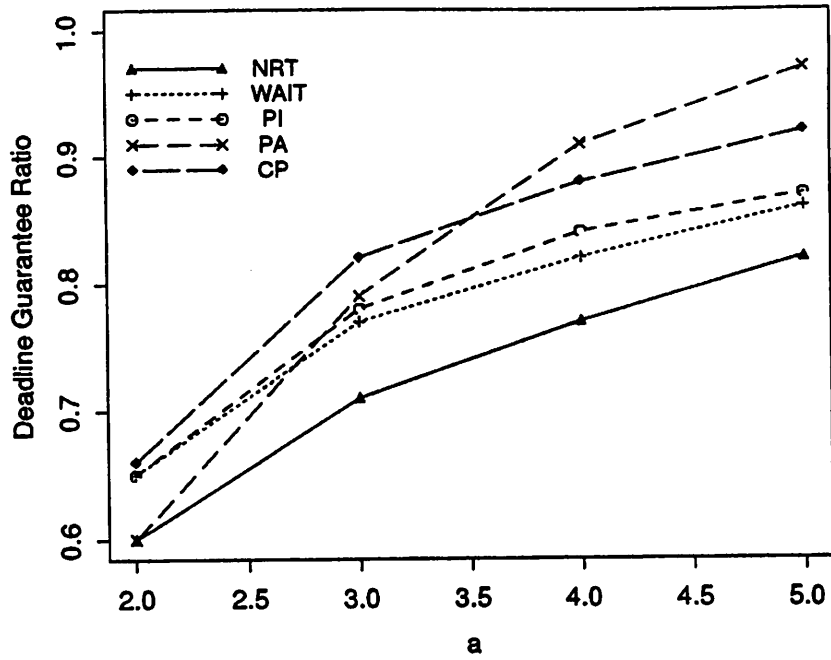


Figure 5.15: Mixed Length, $x = \text{avg}[4, 8]$, $P_w = 0.6$, $h = 2$

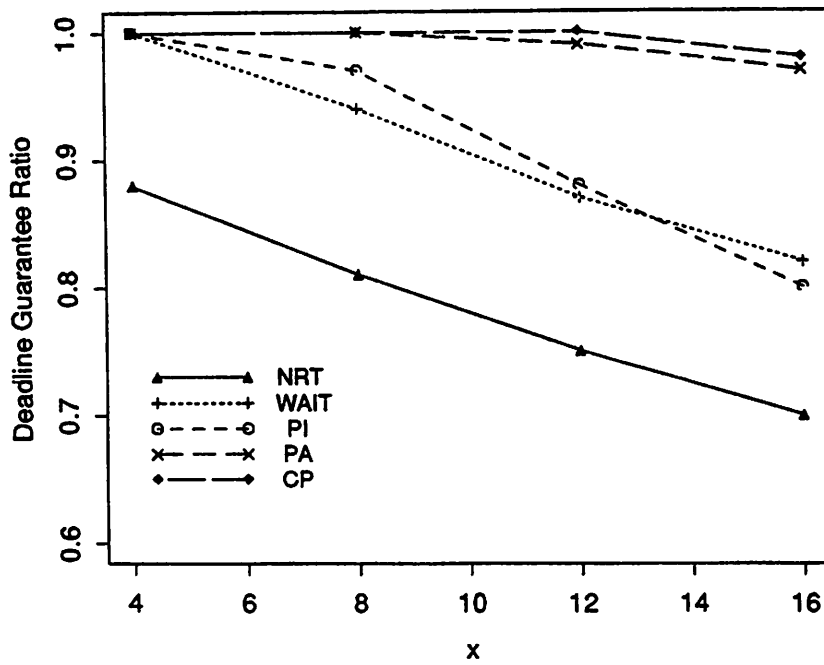


Figure 5.16: CPU Bound System, $P_w = 0.2$, $\alpha = 4$, $h = 2$

CHAPTER 6

BUFFER MANAGEMENT

In this chapter the problem of buffer management in real-time database systems is first discussed. Then based on the existing organization of RT-CARAT, algorithms for real-time buffer allocation and buffer replacement are developed and implemented. The algorithms are then evaluated in connection with two processing components - CPU scheduling and concurrency control.

6.1 Introduction

Data buffering plays an important role in database systems where part of the database is retained in a main memory space so as to reduce disk I/O and, in turn, to reduce the transaction response time. The principle of buffer management is based on transaction reference behaviors [Kear89]. In terms of *locality*, there are basically three kinds of reference strings in database systems:

1. *intra-transaction locality*, where each transaction has its own reference locality, i.e., the probability of reference for recently referenced pages is higher than the average reference probability.
2. *inter-transaction locality*, where concurrent transactions access a set of shared pages.
3. *restart-transaction locality*, where restarted transactions completely repeat their previous reference strings.

Buffer management policies should capitalize on one or more of these three types of locality.

Buffer allocation and buffer replacement are considered to be two basic components of database buffer management [Effe84]. Buffer allocation strategies attempt to distribute the available buffer frames among concurrent database transactions, while buffer replacement strategies attempt to minimize the buffer fault rate for a given buffer size and allocation. The two schemes are closely related to each other and are usually integrated as a buffer management component in database systems.

In this chapter, we consider the problem of buffer management in real-time database systems where transactions have timing constraints, such as deadlines. In a real-time environment, the goal of data buffering is not merely to reduce transaction response time, but more importantly, to increase the number of transactions meeting their timing constraints. To achieve this goal, buffer management should consider not only transaction reference behaviors, but also the timing requirements of the referencing transactions.

This study investigates one buffer organization which is based on the system structure of RT-CARAT. In RT-CARAT, we implement a global buffer in connection with a transaction recovery scheme using after-image journaling. Based on the overall system structure, we study both buffer allocation and buffer replacement for the management of this global buffer which captures inter-transaction locality and restart-transaction locality. We develop several buffer management schemes that attempt to support real-time transactions in meeting their timing constraints. Taking an integrated approach we evaluate the buffer management schemes in connection with real-time CPU scheduling and real-time concurrency control. We attempt to identify the dominant factors in the integrated environment as well as to examine the impact of the individual buffer management schemes on system performance.

6.2 The Buffer Model Used in RT-CARAT

Figure 6.2 illustrates the architecture of RT-CARAT with respect to data buffering. In RT-CARAT, an after-image journaling approach is used for database recovery. Under such a scheme, each transaction is allocated a piece of memory as its working space. If an (update) transaction is successfully committed, the "image" of this memory space will be written back to the permanent storage (disk) of the database;

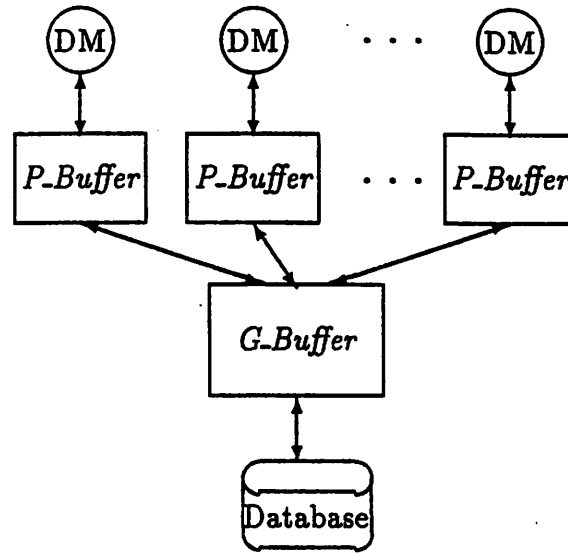


Figure 6.1: The Buffer Model

otherwise the “image” will be discarded. We view each piece of this memory space as a *private buffer* (*P-Buffer*) dedicated to each active transaction. We assume that there will always be enough memory space for the private buffers of concurrent transactions. Clearly, the working space has twofold use. First, it is used for after-image journaling in terms of database recovery. Second, it acts as a buffer that captures the intra-transaction locality.

There is a *global buffer* (*G-Buffer*) which lies in between the pool of private buffers and the database disk. The global buffer is shared among the concurrent transactions. The global buffer also has twofold use. First, it may buffer the data shared among the concurrent transactions. Second, it can hold transaction working data such that a restarted transaction (aborted due to any reason) can fetch its data from the global buffer directly, eliminating the disk I/O. In other words, the purpose of using the global buffer is to capture the inter-transaction locality and the restart-transaction locality.

Based on the buffer organization presented here, we have designed algorithms for global buffer management.

6.3 Buffer Management

As noted in the previous section, the buffer organization consists of two types of buffers - a pool of private buffers and a shared global buffer. For the private buffers, since they are used as transaction working space for the purpose of database recovery and it is assumed that there will always be enough space, there is no need to be concerned with allocation and replacement of private buffers. Therefore, we will not discuss the private buffer any further. In this work, we focus on global buffer management.

The global buffer management consists of two processing components:

- *buffer allocation*, which distributes global buffer space among concurrent transactions;
- *buffer replacement*, which is responsible for global buffer access and page replacement operations.

Upon arrival of a new transaction, the *buffer allocation* component decides if, and how much, global buffer space can be allocated to the transaction, according to a certain buffer allocation policy. When a transaction makes a request for a page, it will first access its private buffer (the working space). If the requested page is not found, the *buffer replacement* component searches the global buffer space allocated to it. If there is a hit in the global buffer, the transaction will use the page by copying it to its working space. Otherwise, the buffer replacement component will fetch the requested page from the disk to the global buffer, where page replacement operation may take place if there is no free space available in the buffer. Note that data consistency between database and global buffer must be maintained. This is guaranteed by the underlying concurrency control mechanism (e.g. two-phase locking protocol) and a buffer write-through scheme. When a transaction comes to its commit stage, it checks if the original copy of any page modified in its working space exists in the global buffer. If yes, it will write the updated page back to the global buffer as well as the database.

In this study, we investigate buffer allocation and buffer replacement for the global buffer management. Although the two components are closely related to each

other, the problem of allocating buffer space to real-time transactions in an optimal way is logically different from the problem of selecting a page for replacement. Thus, we treat buffer allocation and buffer replacement separately. In the following, we first discuss allocation schemes and then replacement schemes.

Here we define some notations that are used in the following sections.

- t - the current time of the system;
- T_i - transaction i ;
- $T_i.dl$ - the deadline of transaction T_i ;
- $T_i.ws$ - the working set (pages) of transaction T_i ;
- $P_buffer_{T_i}$ - the private buffer of transaction T_i ;
- G_buffer - the global buffer;
- G_buffer_size - the global buffer size.

6.3.1 Buffer Allocation

The function of buffer allocation is to distribute the available buffer space among concurrent transactions. Buffer allocation appears more important when there is contention for the global buffer. Here the problem is how to let the concurrent real-time transactions make use of the limited buffer space. The basic idea for buffer allocation is to allocate the global buffer space to the real-time transaction(s) in such a way that the overall transaction deadline guarantee ratio can be improved compared to a buffer allocation scheme which ignores the timing information. In particular, we consider the following four policies for buffer allocation.

- **Alloc0:** *Allocating the global buffer to all the concurrent transactions.* Under this scheme, all the transactions are treated equally in using the global buffer, regardless of their timing constraints. This scheme represents a general method used for the buffer management in non real-time database systems. In this work, we use this scheme as a baseline for performance comparisons.

- **Alloc1:** *Allocating the global buffer to the transaction which has the earliest deadline among the concurrent transactions.* This scheme attempts to speed up the execution of the most time-critical transaction.

Note that Alloc1 may lead to low buffer utilization, since it allocates the entire global buffer space to only one transaction at a time. If the buffer space is relatively large, we may allocate the buffer to a group of transactions so that the buffer can capture more inter-transaction locality and restart-transaction locality. The problem of low buffer utilization is overcome by the scheme described in the following.

- **Alloc2:** Let $T_i (i = 1, 2, \dots, n)$ be the total of n concurrent transactions in the system. The allocation scheme is described by the following algorithm.
 1. sort T_i by $T_i.dl$ in *ascending* order, for $i = 1, 2, \dots, n$;
 2. allocate the global buffer to the first m T_j 's such that the following condition holds.

$$\sum_{j=1}^m T_j.ws \leq G-buffer-size < \sum_{j=1}^{m+1} T_j.ws \quad (6.1)$$

This approach is an extension of Alloc1. Here the global buffer may be allocated to more than one transaction depending on the total buffer size and the size of transaction working sets. However, the basic idea is still the same, i.e. allocating the buffer space to the transactions with shorter deadlines.

- **Alloc3:** *Allocating the global buffer to any m concurrent transactions such that Equation 6.1 holds.* This policy does not address transaction's timing constraints. This random allocation scheme is used for performance comparison with the allocation policy, Alloc2.

To describe how these 4 policies are actually utilized we use the concept of *buffer ownership*. A transaction is called a *buffer owner* if it has been allocated the global buffer space by any buffer allocation policy used; otherwise, it is called a *buffer user*. For any of its page references not in its private buffer, a transaction, no matter whether

it is a buffer owner or a buffer user, will search the global buffer first. If there is a buffer hit, then the transaction will read the referenced page from the global buffer to its private working space (*P-buffer_{T_i}*). The buffer owner and buffer user will behave differently in the situation where a buffer miss occurs: The buffer owner(s) can fetch the missing page from the disk to the global buffer, whereas a buffer user can only read the missing page from the disk to its private buffer.

With regard to transaction reference locality, this buffer allocation scheme captures (a part of) the inter-transaction locality by letting all the concurrent transactions access the global buffer. On the other hand, it captures the restart-transaction locality only for the buffer owners. Under this scheme, it is likely for the transactions with the buffer ownership to make better use of the global buffer, thus reducing their execution time.

6.3.2 Buffer Replacement

The replacement policy comes into play when there are no free buffer frames for newly fetched pages. In a real-time database environment, the replacement scheme should aim not only at minimizing the buffer fault rate, but also at maximizing the number of transactions in meeting their timing constraints. In this study, we examine two replacement policies:

- **LRU** - *replacing the least recently used page*. LRU is the one most commonly implemented in commercial database systems [Effe84, Sacc86]. In this study, it is used as a base algorithm for the purpose of performance comparisons.
- **LRU_{dl}** - *LRU with transaction deadline constraints*. This policy is a modification of LRU, in which we use transaction deadline information such that the pages in the buffer accessed by more time-critical transactions will not be as "easily" replaced compared to what would be done under the original LRU policy.

Now let us look at the LRU_{dl} replacement policy in detail. In LRU_{dl}, there is an LRU stack - a *logical* presentation of the *G-buffer*. Each entry of the stack has three fields:

- *LRU_stack.pdl* - the page deadline, which is the largest deadline value of the transactions that are accessing the page;
- *LRU_stack.usr* - the number of transactions that are accessing the page;
- *LRU_stack.ptr* - the pointer to the *physical* page residing in *G-buffer*.

We also define a parameter *LRU_s_wnd*, called the *search window* of the LRU stack, which is the distance from the bottom entry to the entry counted by *LRU_s_wnd*. With this data structure and the defined parameter, LRU-dl is described by the following algorithm.

```

Search the LRU stack backwards from the bottom up to LRU_s_wnd to
replace one page,
  if a page satisfies (LRU_stack.pdl < t) or (LRU_stack.usr = 0), then
    replace the page
  if no such page is found within the search window then
    replace the page at the bottom of LRU stack;
    put the new page on top of the LRU stack.

```

There are three major concerns in LRU-dl. First, the basic LRU replacement discipline is used to capture the inter-transaction locality and the restart-transaction locality. Second, the replacement condition (*LRU_stack.pdl* < *t*) takes transaction deadline information into account. Here, within the search window, a page held by a transaction already missing its deadline will be replaced. This policy tries to prevent the bottom page from being replaced if it is still being used by some active transaction still trying to meet its deadline. Third, the condition (*LRU_stack.usr* = 0) enhances the algorithm performance for the situation where a page within the search window is not being used by any concurrent transactions while its *LRU_stack.dl* has not yet expired.

Note that the deadline search window is a critical parameter to LRU-dl. If the window is too small, LRU-dl may perform the same as the original LRU. On the other hand, if the window is large, without hardware assistance, LRU-dl may incur a large amount of overhead due to search operations. Even worse is that a large search window may destroy the property of capturing inter-transaction locality - one

of our goals in the design of the algorithm. The sensitivity of search window setting is studied through experiments.

In summary, we plan to investigate the global buffer management in two aspects, i.e., buffer allocation and buffer replacement. The allocation and replacement algorithms presented here are considered to be orthogonal to each other. That is, they can be combined with each other in various ways to construct the buffer management component.

6.4 Implementation

As discussed in the previous section, the global buffer management consists of two processing components - buffer allocation and buffer replacement. In the following, we show how these two components are integrated and implemented in RT-CARAT.

Based on the RT-CARAT process structure (see Figure 3.3), we embed the two components of the global buffer management in TM and DM processes, respectively. We let TM carry out the buffer allocation operation, and DM the replacement policy and buffer write-through operation. The following pseudo code gives the functional description of our implemented buffer management system.

- Buffer allocation (TM):

- Upon arrival of a new transaction,
decide *G-buffer* ownership for each T_i
based on $Alloc_i$, ($i = 0, 1, 2, 3$)

- Buffer replacement (DM):

- For each page reference of T_i not in *P-buffer*, search *G-buffer*.
If *G-buffer* hit, then
copy the page: $G_buffer \Rightarrow P_buffer_{T_i}$
else
read the page: $disk \Rightarrow P_buffer_{T_i}$;
If T_i has *G-buffer* ownership, then
If there is no free buffer frame available for a new page, then
do page replacement in *G-buffer* by LRU or LRU-dl;
copy the page: $P_buffer_{T_i} \Rightarrow G_buffer$

The above is a high-level description of the implementation. The details differ from one algorithm to another. For instance, the implementation for LRU-dl involves the operations on data structures such as *LRU_stack.pdl*, *LRU_stack.ws*, and *LRU_stack.usr*.

Note that a transaction's buffer ownership is dynamic. It is possible that a particular T_i may be a *G-buffer* owner upon arrival and later be preempted (with respect to the ownership) because other transactions with tighter deadlines arrive after it.

6.5 Test Environment

Besides the global buffer management, there are two other major processing components, *CPU scheduling* and *two-phase locking concurrency control*, considered in the experiments. From our initial studies presented in Chapter 4, it has been shown that CPU scheduling and conflict resolution play an important role in real-time database systems. In a real-time database using a global buffer, we want to know how important the buffer management is compared with CPU scheduling and conflict resolution. Here we consider the following algorithms for CPU scheduling and conflict resolution, respectively.

- CPU scheduling policies:
 - SCH0 - a multi-level feedback queue;
 - SCH1 - earliest deadline first.
- Conflict resolution policies:
 - CRP0 - the lock-holding transaction will never be aborted;
 - CRP1 - Among the conflicting transactions, the one with the earliest deadline becomes the lock holder. Others with compatible locks can also hold the lock.

Note that SCH1 and CRP1 are the real-time oriented policies, while SCH0 and CRP0 are the policies commonly found in traditional database systems. In the experiments, SCH0 and CRP0 are used as default algorithms for CPU scheduling and conflict resolution, unless otherwise specified.

Table 6.1: Experimental Settings

Parameter	Setting
<i>MPL</i> (multiprogramming level)	8
<i>DB-Size</i> (database size)	1000 blocks (6000 records)
<i>x</i> (steps per transaction)	8 steps
<i>y</i> (records accessed per step)	3 records
<i>u</i> (computation time per step)	0 units
<i>GB-size</i> (global buffer size)	0 - 500 blocks
<i>AD</i> (access distribution)	uniform, skewed
<i>P_w</i> (prob. of <i>write</i> transactions)	0.05 - 0.30
α (deadline window factor)	2.0 - 6.0
<i>IO</i> (I/O operations per I/O request)	1 - 10
<i>LRU-s.wnd</i> (search window)	0 - 50

The experiments described in this chapter were conducted on a VAXstation II/GPX with two RD53 disks - one for the database and the other for the log. Table 6.1 summarizes the parameter settings in the experiments. The table is divided into two parts. The first part presents the parameters that are kept constant across all workloads, and the second part are those that change according to the different performance measurements. The database consists of 1000 physical blocks (6,000 records). In all of the experiments, the multi-programming level in the system is 8. Transaction length $T(x, y, u)$ is fixed at $T(8, 3, 0)$, where x is the number of steps per transaction, y is the number of records accessed in each step, and u is the computation time per step.

A critical factor in buffer management is the global buffer size - *GB-size*. A small buffer size may cause buffer contention among concurrent transactions. In our experiments, we study various effects on buffer performance by varying the buffer size. Access distribution is another important factor in studying buffer management. In order to exercise the proposed buffer management protocols with different data access patterns, we consider both uniform and skewed access in our experiments¹. In addition, we choose P_w , the probability of write transactions among the concurrent ones,

¹Note that the reference string from each transaction is random. The looping behavior is not considered, since intra-transaction locality is captured by the private buffer in our buffer model which is not the focus of this study.

as a variable, since it affects transaction conflict rate and hence restart-transaction locality. Also, the deadline window factor, α , is a timing-related parameter which specifies the deadline distribution of real-time transactions. The smaller the α value, the tighter the transaction deadlines and vice versa. In our performance studies, in order to create an I/O-bound system, we have to physically increase the number of disk accesses for each I/O request. The number of such I/O operations is specified by the parameter IO . For instance, if IO is set to 1, each I/O request will lead to one disk access operation; if IO is set to 4, then each I/O request will perform four disk access operations (three of which are redundant). Finally, the parameter $LRU-s.wnd$ is a variable used in the performance studies on buffer replacement policies.

Note that the notion of *working set* is used in our proposed buffer management schemes. But for the tested workloads, we have no exact knowledge about the working set of each transaction. Here we extend the transaction working set to a larger data group, called *working page set*, which is equal to $x \times y$, assuming that every record reference ends up at a different page. In practice, any transaction's working set is always less than or equal to its working page set.

In the following experiments we use the following metrics for performance evaluation.

- Deadline guarantee ratio - the percentage of submitted transactions that complete by their deadlines.
- Total hit ratio - the percentage of buffer references that result in a buffer hit.
- Weighted hit ratio (WHR) -

$$WHR = \frac{1}{buffer\ references} \times \sum_{i=1}^n hit(i) \times 2^{(n-i+1)}$$

where n is the maximum number of concurrent transactions; $hit(i)$ is the number of buffer hits resulting from the transaction that has the i th smallest deadline value among the n concurrent transactions. In other words, every time a buffer hit occurs, we compare the deadline of this referencing transaction with those of other concurrent transactions. The counter $hit(i)$ is incremented by one if the

referencing transaction's deadline is the i th smallest among the n concurrent transactions. We weight $hit(i)$ by an exponential factor $2^{(n-i+1)}$. Obviously, the smaller the i , the larger the $hit(i) \times 2^{(n-i+1)}$. For a given number of buffer references, the more the buffer hits from referencing transactions with short deadline values, the larger the WHR will be. Thus, from the measurement of WHR we can tell how the buffer allocation algorithms perform with respect to transaction timing constraints. With the same buffer hit ratio, the larger the WHR , the better the buffer allocation algorithm.

- **Hit-deadline ratio** - the percentage of buffer references that result in a buffer hit from transactions which have not missed their deadlines yet. This metric is used for the evaluation of buffer replacement algorithms. For a given number of buffer references, the greater the number of the buffer hits resulting from referencing transactions whose deadlines have not expired yet, the larger the hit-deadline ratio, and the better the buffer replacement algorithm.

We also collect statistics on CPU utilization, I/O utilization, response time, and transaction restart ratio.

The data collection in the experiments is based in the method of replication. The statistical data has 95% confidence intervals with no greater than $\pm 2\%$ of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

6.6 Experimental Results

In this section, we discuss some performance studies conducted on RT-CARAT. Our goal is to analyze the effectiveness of the proposed buffer management schemes in the presence of various overheads and resource (data, CPU and I/O) contentions. In addition, we want to know how important the real-time oriented buffer management is in real-time databases, as compared to other processing components such as CPU scheduling, conflict resolution and I/O scheduling. The experiments consist of three parts:

1. **System calibration:** This experiment attempts to characterize the testbed system along the dimensions of CPU overhead, I/O overhead, buffer size, and data access distribution. This measurement is essential to our performance studies, since we need to set up system parameters and to identify proper workloads for the following experiments.
2. **Buffer management with buffer allocation:** The purpose of this experiment is to study the proposed real-time buffer allocation schemes.
3. **Buffer management with buffer replacement:** In this experiment we investigate the proposed real-time buffer replacement scheme.

6.6.1 System Calibration

Running on VAXstation II/GPX with two RD53 disks, RT-CARAT is a CPU-bound system. In a CPU-bound system, a software-oriented buffer management component might not be able to improve system performance due to CPU contention. To study the performance of the proposed buffer management schemes, we need to work with a workload where the buffer management system can improve performance. To identify such a workload, we increase I/O operations in each transaction step, making the system vary gradually from CPU-bound to I/O-bound. We calibrate the system by measuring its CPU utilization, buffer hit ratio, and transaction response time as functions of the I/O operations.

In this experiment, since we focus on system calibration, we consider a non real-time database system where transactions have no timing constraints. The buffer management component is Alloc0 and LRU for buffer allocation and buffer replacement, respectively. Transactions are equal in length with $T(x, y, u) = T(8, 3, 0)$ and $P_w = 0.2$.

Figures 6.2 and 6.3 show the system performance in terms of CPU utilization, with uniform access and skewed access, respectively. The skewed access follows the 80-20 rule, i.e., 80% of the invoked transactions access 20% of the database. As one would expect, the CPU utilization decreases as I/O operations increase. Here an important observation is that the CPU utilization with $GB_size > 0$ is much higher

than that with $GB_size = 0$. In general, CPU utilization will increase as buffer size increases, since the larger the buffer, the more references will be in the buffer, and the more I/O operations will be eliminated. But the results in Figures 6.2 and 6.3 show that the buffer management leads to an increase of CPU utilization, even when the buffer size is very small ($GB_size = 25$). This means that the buffer management in RT-CARAT does have significant overhead.

The measures of buffer hit ratio are illustrated in Figures 6.4 and 6.5. The results restate the knowledge that buffer hit ratio is independent of I/O operations, and is increased by skewed access as well as by increase of buffer size. Here we just want to show the relative performance of the global buffer with respect to the parameters of buffer size and access distribution.

Figures 6.6 and 6.7 plot the transaction response time as a function of I/O operations. It is interesting to note that when I/O operations are less (i.e., when the system is CPU-bound), the transaction response time under buffer management is even longer than without data buffering (i.e. $GB_size = 0$). This is because, as we discussed above, the implemented buffer management component has relatively large overhead. The buffer management will not improve system performance unless the CPU utilization is relatively low and I/O overhead is relatively high. For the workload tested in this experiment, the buffer management is seen to be useful only when the CPU utilization is lower than 80% or $IO > 4$. Another critical factor to the system performance is buffer size. If the size is too small to capture the transaction reference locality, the buffer management will do nothing but incur overhead to the system. This can be clearly seen in the two figures for $GB_size = 25$.

The results from this experiment have identified a workload setting, i.e. $IO = 4$, beyond which the system will benefit from global buffer management. Thus, we should start our performance studies on the global buffer management with $IO > 4$.

6.6.2 Buffer Management with Buffer Allocation

In this set of experiments, to focus on the performance study of real-time oriented buffer allocation schemes, we choose LRU as buffer replacement policy. In other words, the global buffer management considered here is a combination of LRU

replacement policy with various proposed buffer allocation schemes. Buffer size is a critical parameter to system performance. Since we study the buffer allocation policies, we need to consider the situation where buffer contention exists. Hence the buffer size is varied within a range from the value around a single *working page set* of a transaction (24 blocks) to the value around the total *working page sets* of concurrent transactions (192 blocks). To create a high inter-transaction locality and restart-transaction locality, we exercise the workloads with skewed access using 80-20 rule.

6.6.2.1 The Effectiveness of Buffer Allocation Schemes

We first determine the effectiveness of the proposed real-time oriented buffer allocation policy (Alloc2) through experiments.

Figures 6.8-6.11 show the performance of buffer allocation schemes with respect to CPU utilization, deadline guarantee ratio, buffer hit ratio, and weighted hit ratio, with $IO = 6$, $P_w = 0.2$ and $\alpha = 4.0$. The CPU utilization shown in Figure 6.8 indicates again that buffer management does have large overheads. With $GB_size = 30$, CPU utilization increases at least 30% once the buffer management is employed. Among the four allocation schemes, Alloc0 incurs the largest overhead, since the scheme allocates the global buffer space to all the concurrent transactions. In that case, every buffer miss will result in page fetching, and even replacement operations, for the global buffer. In contrast with Alloc0, the overhead of Alloc1 appears the lowest. This is because under Alloc1, there is only one buffer owner at any time. Remember that it is only the buffer owner that can do page fetching and replacement for the global buffer.

Figure 6.9 illustrates the performance of real-time transactions with respect to deadline guarantee ratio. As one would expect, the deadline guarantee ratio increases as the buffer size increases. Considering the buffer allocation schemes, however, there is no significant performance difference. Figure 6.10 plots the buffer hit ratio as a function of GB_size . Again, there is no significant performance difference among the proposed allocation schemes. Note that the random allocation scheme, Alloc3,

performs the same as Alloc2 which is expected to provide best performance among the four protocols.

We also exercised the workloads with the deadline setting $\alpha = 2$. The performance results are basically the same as what we presented above.

At this point the reader may wonder if the allocation schemes function as intended. The answer is yes. We measured the effectiveness of the allocation schemes by *WHR*, the weighted hit ratio. From Figure 6.11 we can see that Alloc2 outperforms all other schemes with respect to the weighted value, i.e., Alloc2 does allocate the available buffer space to more time-critical transactions as compared to other schemes. We also observed that the performance difference between Alloc2 and others (except Alloc1) diminishes as the buffer size becomes large. This is because when the buffer size is large enough, it will capture the inter-transaction locality and restart-transaction locality for all the concurrent transactions, regardless of their timing constraints. Alloc1 becomes the worst as the buffer size increases, since this single ownership allocation scheme captures the least transaction reference locality. Comparing the two non real-time allocation schemes, Alloc0 and Alloc3, Alloc3 has larger *WHR* value than Alloc0. This is because unlike Alloc0 which lets all the concurrent transactions share the global buffer space, Alloc3 allocates the buffer only to the limited number of concurrent transactions, depending on the available buffer size. Due to its non real-time nature, however, Alloc3 does not perform as well as Alloc2 does.

Our experiments have shown that the proposed buffer allocation scheme Alloc2 does work in favor of more time-critical transactions. But with respect to the metric - deadline guarantee ratio, the use of Alloc2 does not improve performance.

In a locking-based database system with limited CPU and I/O resources, transaction deadline guarantee ratio under (real-time oriented) buffer management may be constrained by one or more of the following factors: It may be that the overhead from protocol implementation overshadows the benefit from buffer management. Even if the overhead from buffer management is negligible, CPU contention can still block transaction processing. Similarly, I/O may be a bottleneck when the system becomes

I/O bound. Finally, data contention may result in transaction blocking due to the use of locking scheme for concurrency control.

We have shown in the above discussion that the overhead incurred from buffer management is large and not negligible. In order to get performance gains from data buffering, we increased physical I/O operations. As a result, the implementation overhead is no longer a significant factor. To identify the main reason why the proposed buffer allocation scheme does not improve the performance of real-time transactions, we need to analyze the situations where resource contention (data, CPU, and I/O) exist.

In the following experiments, we only compare Alloc2 with two baselines NBuf (no global buffer) and Alloc0. We still consider the situation where buffer contention exists. From previous experiments (see Figure 6.11), we know that in terms of weighted hit ratio, there is a large performance difference between Alloc2 and Alloc0 when *GB-size* is less than 150 blocks. In the following set of experiments, we set *GB-size* at 90 blocks, which is large enough to hold about half of the total *working page sets* of concurrent transactions and yet “small enough” to show the performance gain of Alloc2 over Alloc0 with respect to weighted hit ratio. The deadline setting is $\alpha = 4$.

6.6.2.2 Buffer Allocation vs. Conflict Resolution

In this experiment, we consider the use of CRP1 so as to determine if real-time oriented conflict resolution is more important than the buffer management in the presence of data contention. To create data contention, we gradually increase P_w from 0.05 to 0.30.

Figure 6.12 compares the weighted hit ratio for the buffer management with and without applying CRP1. First of all, the four curves show increasing hits as P_w increases. This is because an increase of P_w results in an increase in the data conflict rate, and in turn, the increase of restart-transaction locality, and the increase of the weighted hit ratio. Second, applying CRP1 increases the weighted hit ratio for the allocation scheme Alloc0 but not for Alloc2. This can be explained as follows. Under CRP0, the transaction aborts result from deadlock only, whereas under CRP1 most

transaction aborts result from conflict resolution. Overall, the transaction abort rate under CRP1 is higher than that under CRP0 for any given P_w . Note that it is the transactions with longer deadlines that are aborted under CRP1. For Alloc2, since it allocates the buffer space only to some of the concurrent transactions with shorter deadlines, the increased restart-transaction locality (the increased abort rate) due to the use of CRP1 will not affect the weighted hit ratio. For Alloc0, because of its random allocation policy, the increased restart-transaction locality will lead to the increase in overall buffer hit ratio, and, in turn, the weighted hit ratio.

The transaction deadline guarantee ratio is plotted in Figure 6.13. The reader can clearly see that under CRP1 transactions perform much better than they do without the use of CRP1. This holds even for the system where no global buffer is employed. Comparing Figures 6.12 and 6.13, we know that the performance gain does not come from the buffer management, otherwise NBUF-CRP1 would not outperform NBUF and LRU_Alloc2-CRP1 would not outperform LRU_Alloc2 either. It is CRP1 - the real-time oriented conflict resolution that significantly improves the performance.

6.6.2.3 Buffer Allocation vs. CPU Scheduling

From the above experiment we have seen that it is the real-time oriented conflict resolution (CRP1), not the real-time oriented buffer allocation scheme, that improves transaction performance in the presence of data contention. Now we examine what kind of role CPU scheduling plays in connection with the buffer management.

Figures 6.14, 6.15 and 6.16 depict the performance of real-time transactions versus IO , with respect to CPU and I/O utilization, weighted hit ratio and deadline guarantee ratio. Here the highest CPU utilization, as shown in Figure 6.14, is less than 80%, which means that there is no severe CPU contention in the system. When SCH1 is applied, we can still see a slight performance improvement in terms of deadline guarantee ratio in Figure 6.16. Comparing Figures 6.15 and 6.16, especially the curves of LRU_Alloc2 (Alloc2, but no SCH1), LRU_Alloc0-SCH1 (SCH1, but no Alloc2), and LRU_Alloc2-SCH1 (both Alloc2 and SCH1), we notice that the performance gain does not come from real-time oriented buffer management, otherwise LRU_Alloc2 would

outperform LRU_Alloc0_SCH1 in terms of deadline guarantee ratio, since its weighted hit ratio is much higher than that of LRU_Alloc0_SCH1.

6.6.2.4 Discussions

As we mentioned above, the real-time oriented buffer management may be constrained by one or more factors. Through carefully designed experiments we have shown that one such factor is data contention. Under high data contention, the performance improvement can be achieved through real-time oriented conflict resolution, but not real-time oriented buffer allocation.

The conducted experiments also indicate that CPU scheduling is more important than buffer allocation. Note that to avoid the negative effect from the implementation overhead of the buffer management, we created a system (by increasing physical I/O operations per I/O request) which is not CPU bound. In other words, for the tested workloads ($IO=4,5, \dots,8$), there was no strong CPU contention. For a CPU bound system, we can foresee, from our experimental results obtained here, that CPU scheduling is the key factor to system performance and real-time oriented buffer allocation is not important even if the overhead of the buffer management is negligible.

Another factor which may block transaction processing is disk I/O. Under I/O contention, the real-time oriented buffer allocation scheme is expected to play an important role in improving the performance of real-time transactions, since the essential goal of data buffering is to reduce disk I/O. But on the other hand, the allocation scheme may not be helpful if data contention exists or reference locality is not high. In our experiments, the I/O utilization is over 80% (see Figure 6.14) and the average I/O queue length is over 2.3 (not plotted), for LRU_Alloc0 and LRU_Alloc2, when $IO \geq 7$. In such an environment, we do not see any significant performance improvement due to the real-time oriented buffer allocation scheme. Here the main reason, we believe, is the data contention - the issue we just discussed above.

Another issue we haven't addressed is I/O scheduling. We would like to know in an I/O bound environment how the I/O scheduling affects the system performance as compared to the buffer allocation scheme. Unfortunately, we can not explore this interesting issue due to limitations on the physical testbed. A recent study based

on simulation [Chen90] has demonstrated that I/O scheduling is a key processing component in real-time database systems. It is likely that I/O scheduling is more important than buffer allocation.

The last factor we want to point out is buffer contention. Our discussion on buffer allocation has focused on the situation where buffer contention exists. We notice from the performance results that the allocation schemes Alloc0 and Alloc2 will produce the same performance when the buffer size is large enough to hold the total working page sets of the concurrent transactions (see Figure 6.11). Obviously, we do not need to worry about buffer allocation if buffer contention is no longer a problem.

6.6.3 Buffer Management with Buffer Replacement

In this experiment, we investigate the proposed buffer replacement scheme. To focus on the performance of buffer replacement, we choose Alloc0 - the simplest scheme for buffer allocation. Also, to make the replacement operation meaningful, the buffer size should be chosen at least as large as the total *page working sets* of concurrent transactions. In the experiments, the buffer size is first set at 250 blocks which is larger than the total page working sets of 192 blocks for $x = 8$ and $y = 3$. Transactions access the database uniformly with $P_w = 1.0$, $\alpha = 6$ and $IO = 6$. Under such a workload, the measured CPU utilization is about 40% and the I/O utilization is above 90%. The measured transaction restart ratio is around 15% under CRP0 and 55% under CRP1, respectively.

We first examine the effectiveness of the proposed buffer replacement scheme, LRU_dl. Figure 6.17 shows the total hit ratio and the hit-deadline ratio versus the LRU search window (*LRU_s_wnd*). As compared with the total hit ratio under LRU policy which is irrelevant to the search window, the total hit ratio under LRU_dl decreases as *LRU_s_wnd* increases. This is due to the fact that LRU_dl breaks the LRU discipline, i.e., it may replace a page which is not the least recently used. On the other hand, LRU_dl reduces the buffer hits mostly for those transactions that have missed their deadlines. This is illustrated by the curve of hit-deadline ratio, where the ratio is a constant (25%) when *LRU_s_wnd* is less than 20. Remember

that the global buffer is designed to capture the inter-transaction locality as well as restart-transaction locality. Increasing *LRU_s_wnd* will reduce buffer hits resulting from inter-transaction locality. This is why the hit-deadline ratio decreases too when *LRU_s_wnd* is greater than 20. Note that the hit-deadline ratio becomes constant again when *LRU_s_wnd* is larger than 40. This is because for the exercised workload, the conditions for page replacement (see LRU_dl algorithm) can often be satisfied within last 40 pages of the LRU stack. Overall, the results presented here indicate that the proposed buffer replacement algorithm LRU_dl is effective in terms of preventing the not-yet-missing-deadline pages from being replaced from the global buffer. However, it is not good enough to raise the hit-deadline ratio.

To see how LRU_dl affects the overall transaction performance and to see how other processing components, like *conflict resolution policy*, performs as compared to buffer management, we examine the transaction deadline guarantee ratio under LRU, LRU_dl, LRU-CRP1 and LRU_dl-CRP1. The results in Figure 6.18 show that LRU_dl performs basically the same as LRU no matter whether CRP1 is applied or not. This is understandable since LRU_dl does not change the hit-deadline ratio. On the other hand, incorporating CRP1 into the system increases the transaction deadline guarantee ratio from 79% to 90%, achieving about 14% performance improvement.

We further exercised workloads with different buffer sizes, deadline distribution, and access distribution (skewed access). The results are similar to what we have shown above.

Discussions

Our experiments have shown that buffer replacement in real-time database systems is not a simple issue. The proposed replacement algorithm, LRU_dl, attempts to balance the trade-off between capturing transaction locality and meeting transaction timing constraints. In practice, the performance of the algorithm is complicated by system parameters and different workloads, such as buffer size, data access distribution, transaction deadline distribution, data contention, resource (CPU and I/O) contention as well as the buffer search window.

However, the changes of various parameters and workloads in the experiments do not change the performance of the algorithm, i.e., it does not improve the buffer performance with respect to hit-deadline ratio and deadline guarantee ratio. This may result from the following reasons. First, the simple LRU is superior to the sophisticated LRU-dl. The mechanism used in LRU supports the basic idea of "keeping the pages in the buffer for not-yet-missing-deadline transactions". Second, in an integrated system, like RT-CARAT, the performance of buffer replacement schemes may be limited by other processing components where system bottlenecks exist. Our experimental results have shown that under data contention, the use of real-time oriented conflict resolution largely improves the performance of real-time transactions. This implies that rather than developing sophisticated replacement policies, it is more important to resolve data conflicts. Similarly, as we discussed for buffer allocation, I/O scheduling can be another more important processing component than buffer replacement.

6.7 Concluding Remarks

Buffer management plays an important role in traditional database systems. In this chapter, we explore this processing component for supporting real-time transactions. We have developed several buffer allocation and buffer replacement policies which take transaction timing constraints into account. The proposed allocation and replacement schemes are integrated to serve as a buffer management component and are implemented on our real-time database testbed.

The experimental results obtained from the testbed indicate that under two-phase locking, the real-time oriented buffer management schemes do not significantly improve system performance over non real-time buffer management schemes. With regard to buffer allocation, we have shown that data contention is a constraint on the performance improvement of buffer management. Under data contention, conflict resolution becomes a key factor in real-time transaction processing. In addition, CPU scheduling is more important than buffer allocation, even if the system is not CPU bound. Concerning buffer replacement, we have seen that the complicated real-time oriented replacement algorithm performs no better than a simple LRU policy. Again,

under data contention, it is conflict resolution that significantly improves transaction performance. This study indicates that in the integrated system, it is the conflict resolution and CPU scheduling, not the real-time oriented buffer management, that are the dominant factors for real-time transaction processing.

Note that the results presented in this work are obtained from the system that employs two-phase locking mechanism for concurrency control. The conclusion that conflict resolution is a dominant factor is relevant to the locking approach used for concurrency control. The performance of real-time oriented buffer management may differ from one concurrency control mechanism to another. For example, under real-time optimistic concurrency control, as discussed in Chapter 7, conflict resolution is not an important factor. Rather, high transaction restart rate is a main problem. Under such a system, real-time oriented buffer management may play an important role in improving the performance of real-time transactions. Buffer management with optimistic concurrency control remains as part of future work.

Another remark is that the buffer model considered in this work is tailored to the existing organization of RT-CARAT. Particularly, each transaction uses an unlimited working space (for recovery purpose) as its private buffer. This buffer pool organization may have a potential impact on buffer allocation schemes. Also, since the system does not distinguish between *read* request and *update* request, it always fetches the requested page from the global buffer to a private buffer. But for a shared read-only page, it is sufficient to simply keep it in the global buffer. With the assumption that the allocated memory space is always larger than the size of total working sets of concurrent transactions, an alternative scheme is to use a single memory space for both after-image journaling and data buffering. Such an organization needs to be investigated in the real-time context.

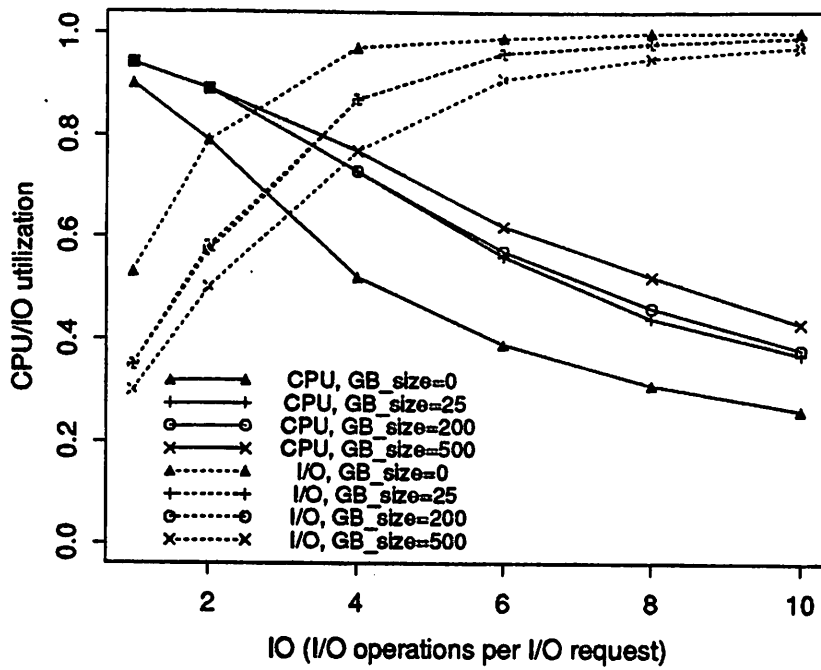


Figure 6.2: System Calibration, Uniform Access

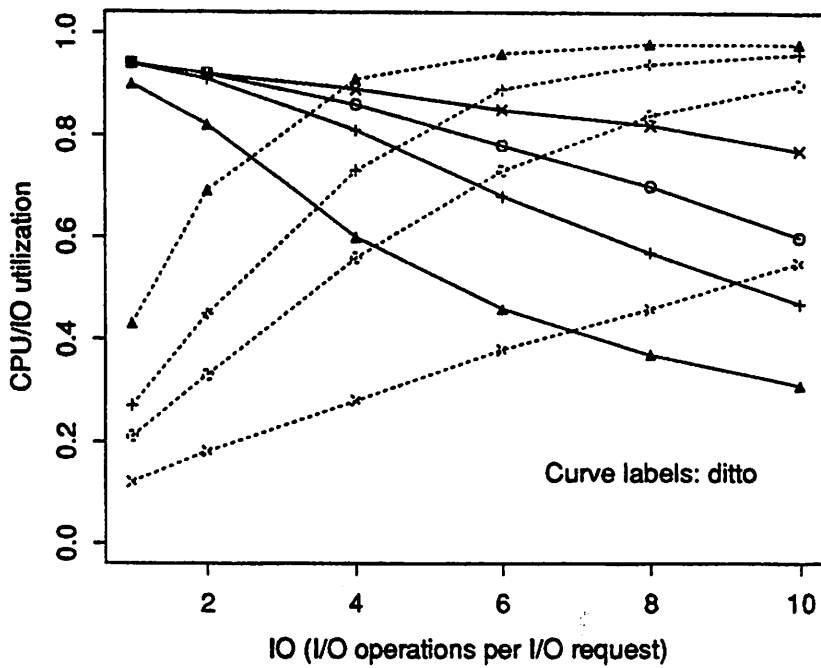


Figure 6.3: System Calibration, Skewed Access

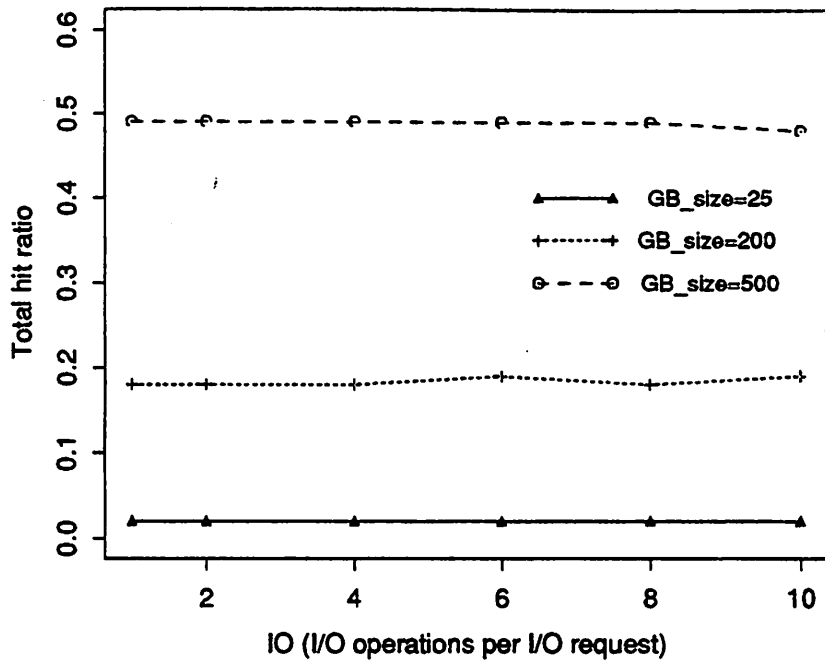


Figure 6.4: System Calibration, Uniform Access

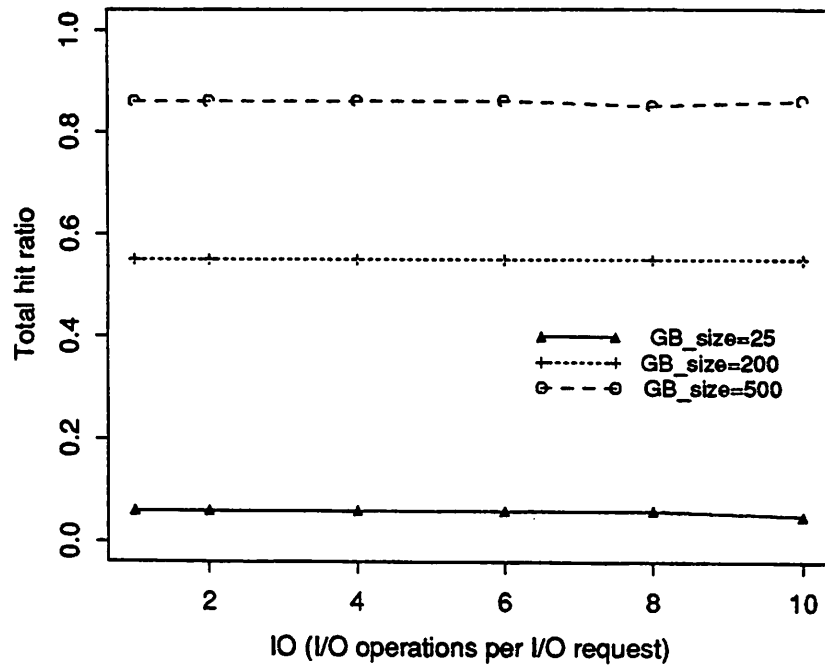


Figure 6.5: System Calibration, Skewed Access

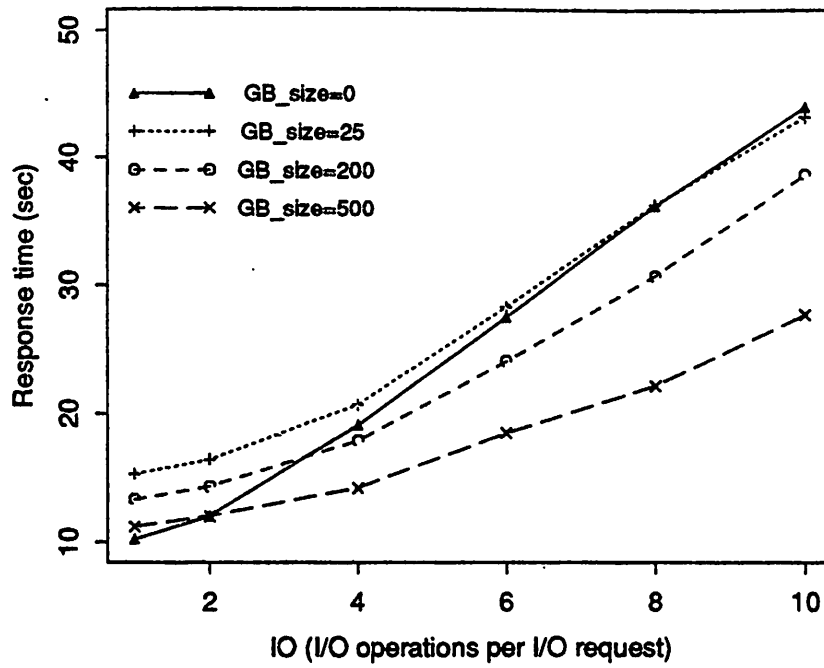


Figure 6.6: System Calibration, Uniform Access

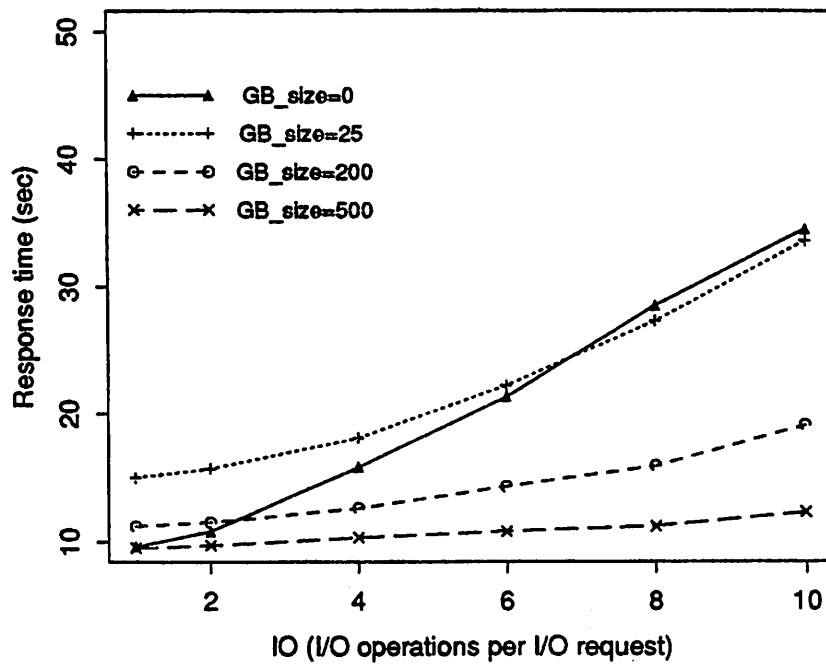


Figure 6.7: System Calibration, Skewed Access

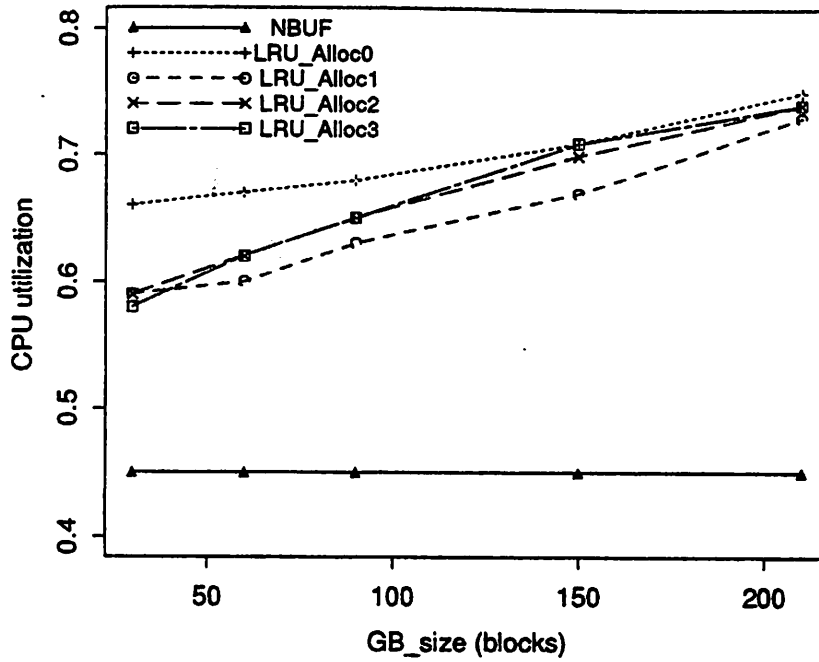


Figure 6.8: Comparisons of Allocation Schemes

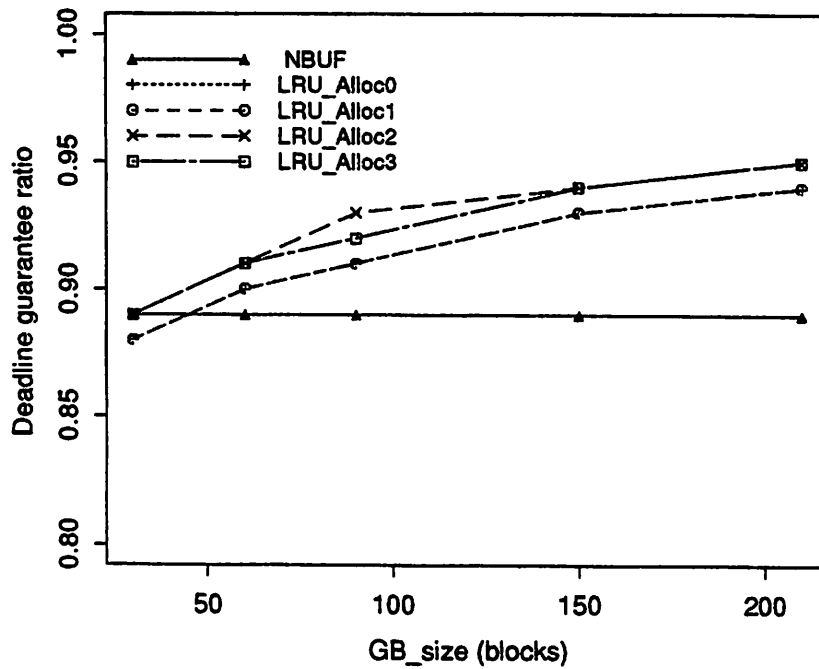


Figure 6.9: Comparisons of Allocation Schemes

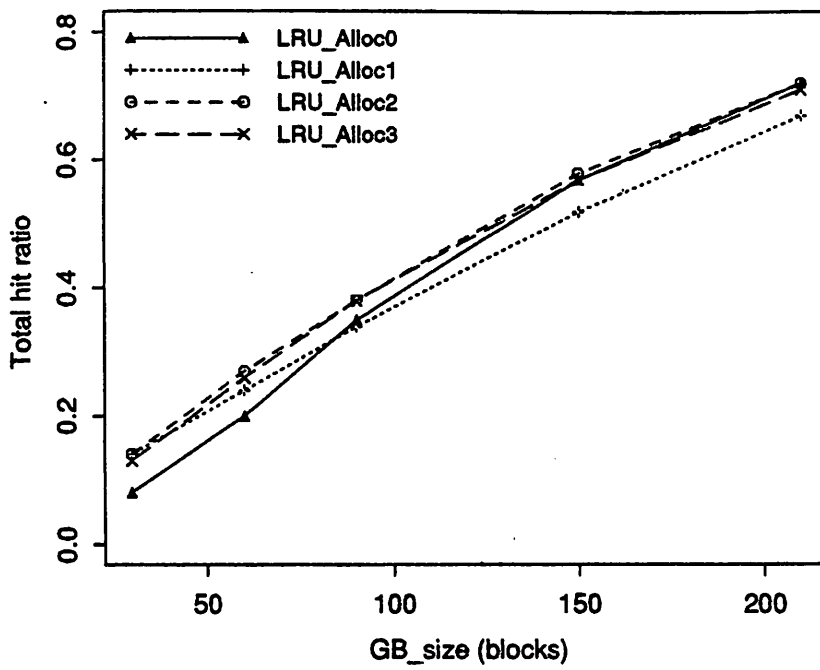


Figure 6.10: Comparisons of Allocation Schemes

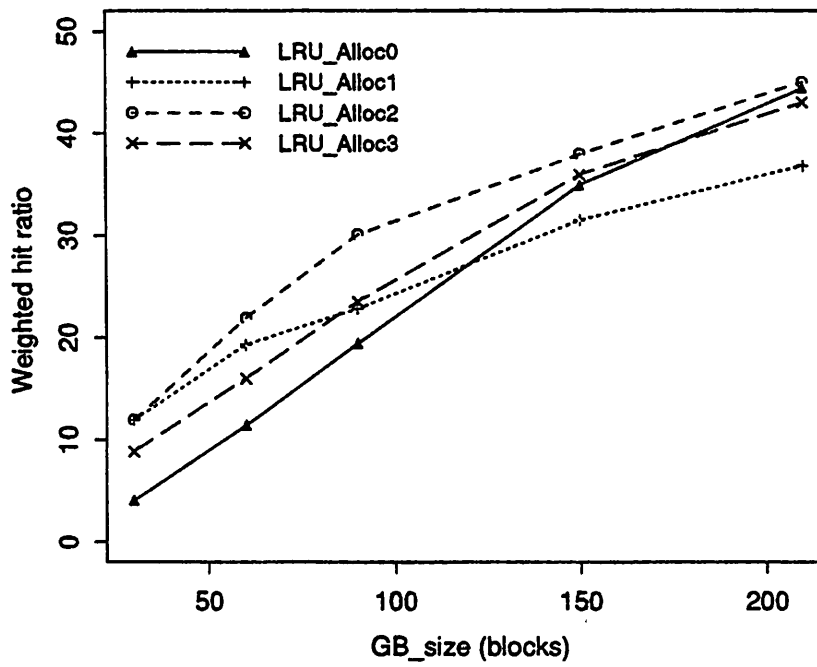


Figure 6.11: Comparisons of Allocation Schemes

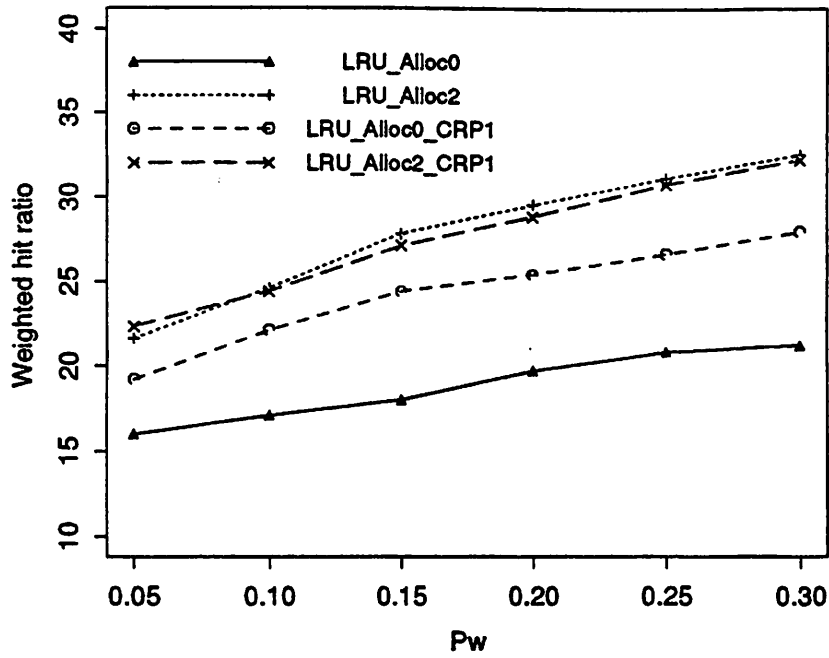


Figure 6.12: Allocation vs. Conflict Resolution

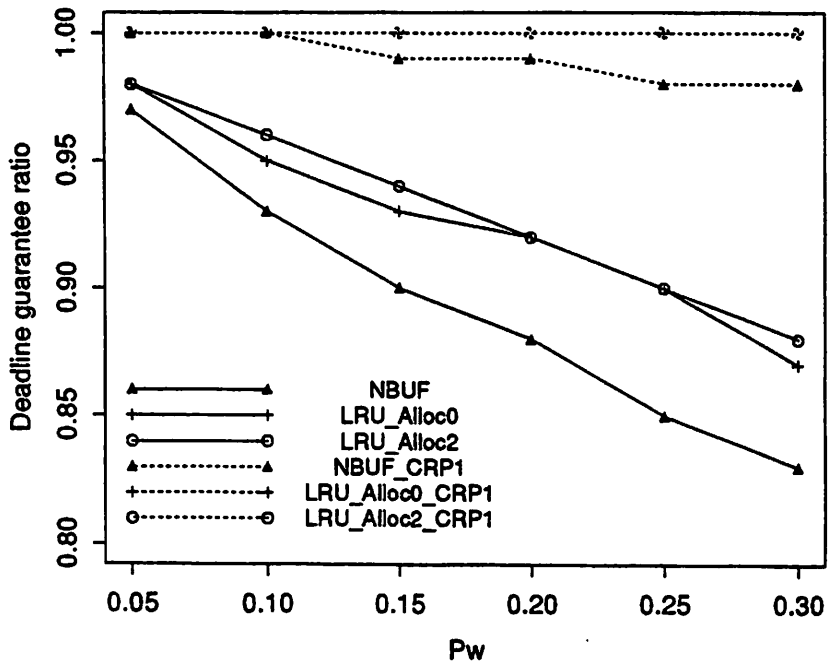


Figure 6.13: Allocation vs. Conflict Resolution

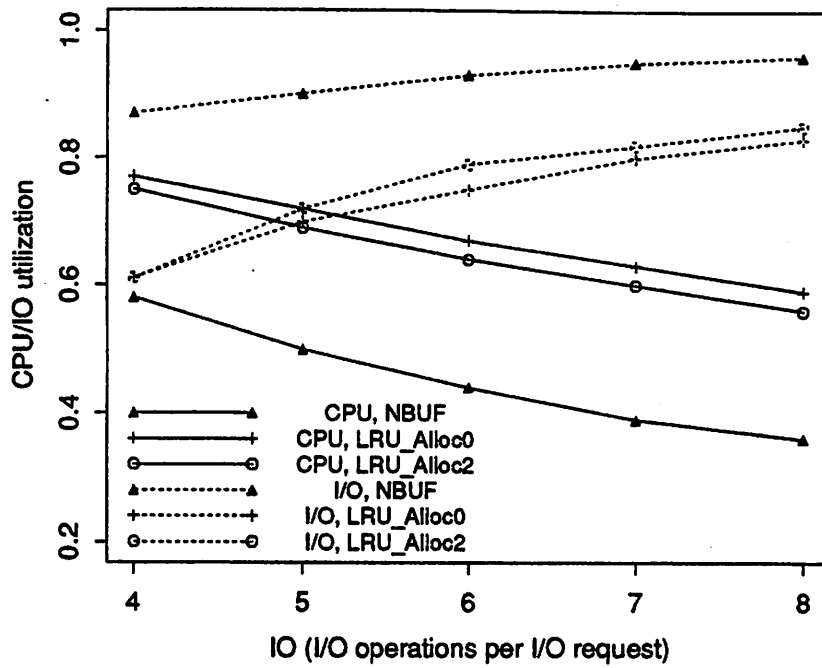


Figure 6.14: Allocation vs. CPU Scheduling

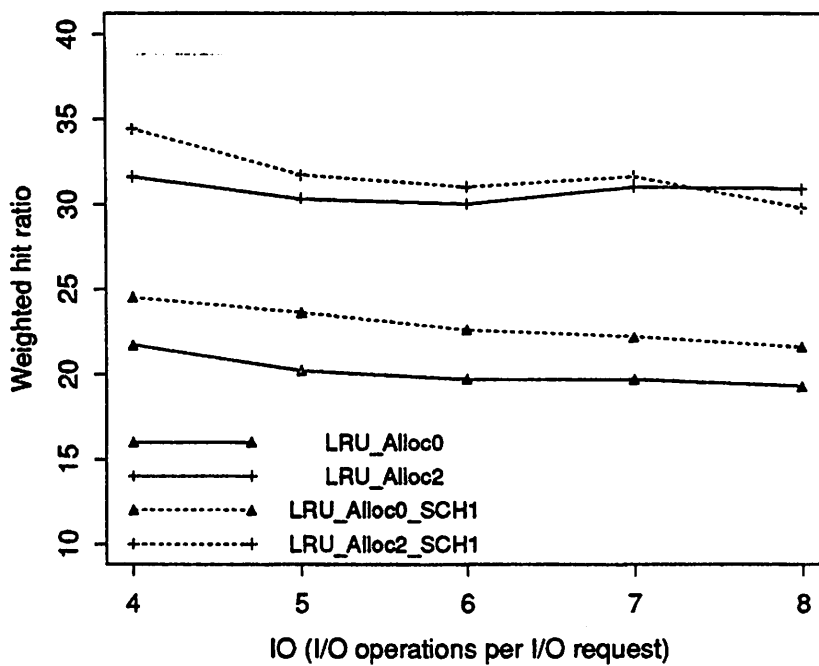


Figure 6.15: Allocation vs. CPU Scheduling

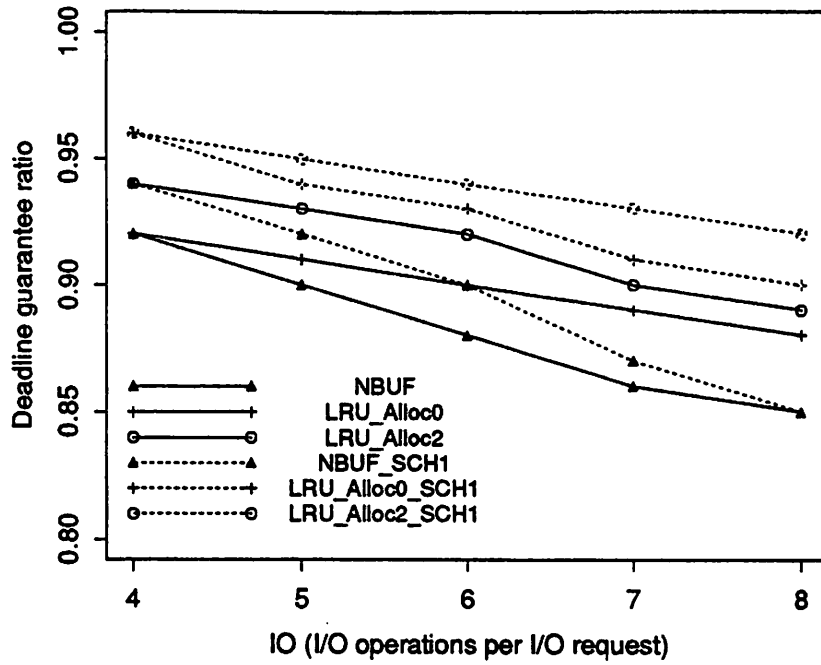


Figure 6.16: Allocation vs. CPU Scheduling

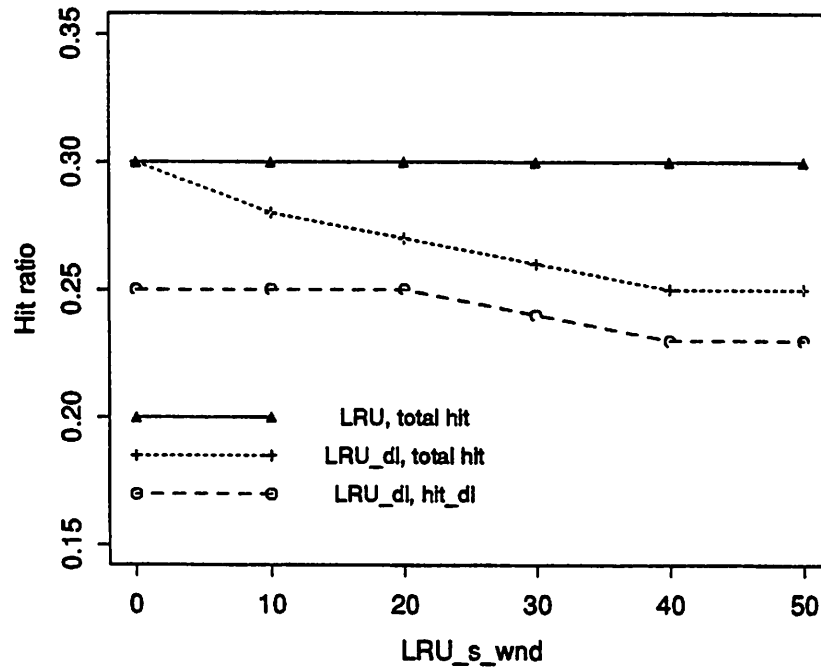


Figure 6.17: Comparisons of Replacement Schemes

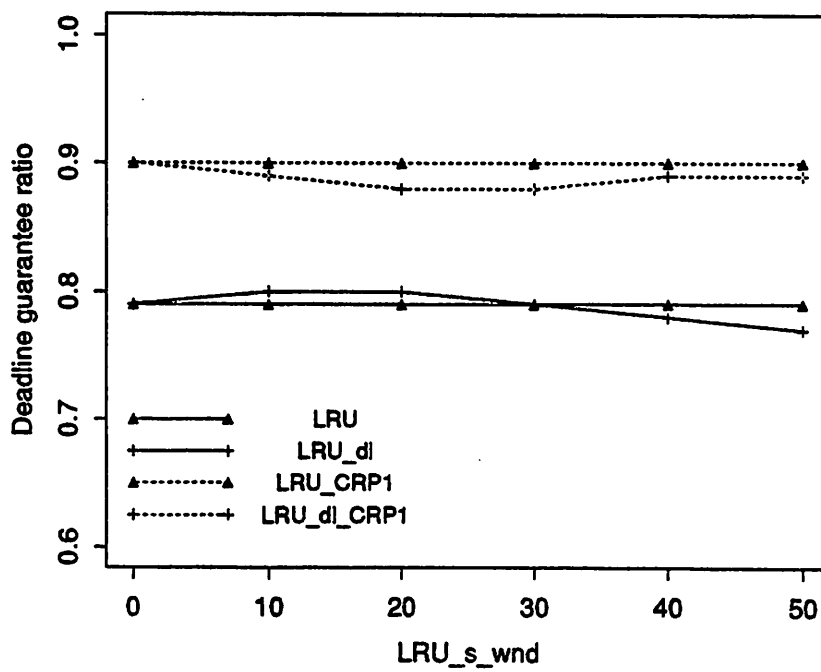


Figure 6.18: Replacement vs. Conflict Resolution

CHAPTER 7

OPTIMISTIC CONCURRENCY CONTROL

In this chapter an optimistic approach is proposed as an alternative to two-phase locking for real-time concurrency control. Based on a locking mechanism to ensure the correctness of the OCC implementation, a set of optimistic concurrency control protocols are developed. The protocols are implemented on the RT-CARAT testbed and then evaluated against each other and against two-phase locking. In this study, the issue of physical implementation is particularly emphasized in the algorithm design and experimentation.

7.1 Introduction

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement as defined by the notion of serializability, most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [Stan88b, Abbo88b, Care89, Jau90, Son90]. This is not surprising since 2PL has been well studied in traditional database systems and is being widely used in commercial databases. But 2PL, on the other hand, has some inherent problems such as the possibility of deadlocks and long and unpredictable blocking times. These appear to be serious problems for real-time transaction processing, since in a real-time environment, transactions need to meet their time constraints as well as consistency requirements.

Another class of concurrency control schemes is the well-know optimistic approach [Kung81]. Ideally, optimistic concurrency control (OCC) has the properties of non-blocking and deadlock freedom. These properties make the scheme especially attractive to real-time transaction processing. However, little work has been done on real-time oriented optimistic concurrency control. This is probably because OCC is

not widely used in practice. And it might also lie in the fact that analytical studies (e.g., [Agra85, Mena82]) and simulations (e.g., [Agra87, Care84]) have revealed that in most cases the locking approach performed better than the pure OCC [Kung81]. The main reason for this is the risk of a high abort rate or even cyclic restarts, especially for long transactions or in the presence of hot spot objects. The optimistic approach will apparently become a loser if there are insufficient system resources (CPU and I/O) available. Nevertheless, the situation is different in real-time database systems where transaction scheduling is a main concern. As already shown in Chapter 4, CPU scheduling is a key component in real-time transaction processing and it interacts with concurrency control during transaction scheduling. Because the non-blocking nature of optimistic concurrency control can give the CPU scheduler a greater freedom to schedule real-time transactions, OCC may be in a better position, than 2PL, to be integrated with priority-driven CPU scheduling in real-time databases.

In this study, we investigate the optimistic approach for real-time transaction processing. Using a locking mechanism to ensure the correctness of the OCC implementation, we develop a set of optimistic concurrency control protocols. The protocols possess the property of deadlock freedom and have the potential for a high degree of parallelism. Integrated with priority-driven preemptive scheduling, the blocking time under the proposed protocols is limited and is predictable compared with 2PL.

The focus of this study is to examine the overall effect and the impact of the overheads involved in implementing real-time optimistic concurrency control. Here the implementation issue is critical. For example, *validation*, a key process in OCC, can be carried out by a *broadcast* mechanism. But this is just a logical operation. In practice, the implementation of the broadcast and the corresponding overheads may adversely affect the performance of a protocol.

In this work, we also investigate optimistic concurrency control in the context of the starvation problem. Because of their higher probability to conflict with other transactions, long transactions are likely to be repeatedly restarted and thus have less chance to meet their deadline than short transactions. Instead of limiting the number of transaction restarts, as is often proposed for traditional database systems

[Pein83, Prad86, Thom90], we use length and deadline sensitive priority assignment to address the problem.

7.2 Optimistic Concurrency Control for Real-Time Transactions

In this section, we first discuss the principles underlying optimistic concurrency control for real-time transactions, particularly regarding its validation. Then we propose a set of locking-based optimistic concurrent control protocols and discuss their implications in comparison with a two-phase locking approach. At the end, we present some conflict resolution policies used in conjunction with the proposed protocols.

7.2.1 Principle of Optimistic Concurrency Control

With the original OCC [Kung81], the execution of a transaction consists of three phases: read, validation, and write. The key component in OCC is the validation phase where a transaction's destiny is decided. To ensure the serializability criterion, it must be satisfied that if transaction T_i is serialized before transaction T_j , then

Condition 1: the writes of T_i should not affect the read phase of T_j ; and

Condition 2: T_i 's writes should not overwrite T_j 's writes.

Basically, the validation process can be carried out in either of the following two ways¹:

- **Backward validation:** validating against committed transactions. When a conflict is detected, the validating transaction will abort itself.
- **Forward validation:** validating against active transactions. When a conflict is detected, either the validating transaction in validation phase or the conflicting transactions in read phase can be aborted. Furthermore, the validating transaction may defer validation until later on, thus avoiding any unnecessary abort.

¹For detailed discussion, the reader is referred to [Hard84].

In real-time database systems, conflicts should be resolved according to the priority associated with real-time transactions. As a result, either the validating transaction or the conflicting transaction(s) may be chosen for abort or possible delay. Clearly, to provide flexibility for conflict resolution, a transaction should be validated against active transactions instead of committed ones, i.e., forward validation is preferable.

Let T_j be the validating transaction and T_i ($i = 1, 2, \dots, n, i \neq j$) be the transactions in their read phase. Let $RS(T)$ and $WS(T)$ denote the *read set* and *write set* of transaction T , respectively. Then, the validation operation for T_j can be described by the following procedure.

```

VALID := true
for  $T_i$  ( $i = 1, 2, \dots, n$ ) do
    if  $WS(T_j) \cap RS(T_i) \neq \emptyset$ 
        then VALID := false
if VALID
    then execute write phase
    else invoke real-time conflict resolution

```

The condition $WS(T_j) \cap RS(T_i) \neq \emptyset$ guarantees that data read by the T_i 's have not been written by T_j . To ensure Condition 2, I/O operations in the write phase must be done sequentially in validation order.

7.2.2 Optimistic Concurrency Control Using Locking (OCCL)

Validating against active transactions is straightforward at the logical level. For example, a *broadcast* mechanism can be used for the validation, where the validating transaction "notifies" other currently running transactions of data access conflict. In the following, we describe a physical implementation of a set of validation protocols.

The proposed protocols are based on a locking mechanism [Hard84], thus being named OCCL. In the system, each transaction T_i maintains its own *read set*, $RS(T_i)$, and *write set*, $WS(T_i)$. In addition, a systemwide lock table, LT , is shared by all concurrently executing transactions. We define two lock modes - *read-phase lock* (R-lock) and *validation-phase lock* (V-lock), where an R-lock for a data item is set in LT by transaction in its read phase while a V-lock on a data item is set only by a

transaction in its validation phase. The two lock modes are not compatible. Briefly, the validation operation is carried out by checking the lock compatibility with the lock table.

As we mentioned earlier, to satisfy the requirement of *serializability*, two conditions must hold. Here Condition 1 is ensured by the locking mechanism discussed above. We give two protocols for ensuring Condition 2. Again, we use T_i ($i = 1, 2, \dots, n$) to denote transactions in read phase and T_j as the transaction in validation phase. We bracket a critical section by "<" and ">".

7.2.2.1 Serial Validation-Write: OCCL-SVW

A simple way to guarantee Condition 2 is to embed the validation phase and write phase in one critical section. We call this scheme *serial validation-write* in the sense that the validation phase and the write phase are indivisible.

During the read phase, each transaction T_i works on its own local copy and sets an R-lock in LT for every data object in its $RS(T_i)$. During the validation phase, the validating transaction T_j checks its $WS(T_j)$ against $RS(T_i)$ by setting the V-lock for all data objects in its $WS(T_j)$ in LT. If all the V-locks can be obtained, that means that the write set of the validating transaction does not intersect with the read set of any other active transactions. At this point, the validating transaction deletes its R-locks in LT and proceeds to its write phase. A failure to set a V-lock, on the other hand, indicates that Condition 1 has been violated, since this implies that a data object in $WS(T_j)$ also falls in $RS(T_i)$ of other transaction(s). In this case, the real-time conflict resolution policy is invoked to resolve the conflict (see Section 7.2.4). We give the protocol for the serial validation-write scheme via the following pseudo code.

OCCL-SVW:

Read phase:

```
< for every data object in  $RS(T_i)$  do  
    set an R-lock in LT >
```

Validation and Write Phase:

VALID := true

```
< for every data object in WS( $T_i$ ) do
    set a V-lock in LT
    if an R-lock of that object exists
        then VALID := false
    release  $T_j$ 's R-locks in LT
    if VALID
        then execute write phase
    else invoke real-time conflict resolution
    release  $T_j$ 's V-locks in LT >
```

OCCL-SVW is simple and is easy to implement. It can be applied both to main memory resident database systems where the write phase is done in main memory and to disk resident databases. The latter will produce good performance if most of the transactions are queries.

On the other hand, *serial validation-write* may not be necessary if conflicts occur rarely between update transactions. Also, since the write phase and the validation phase are embedded together in a critical section, the critical section can easily become a bottleneck. This is especially true of disk resident databases. To separate the write phase from the critical section, we now develop another protocol, called *parallel validation-write*.

7.2.2.2 Parallel Validation-Write: OCCL-PVW

In order to separate the write phase from the critical section and at the same time guarantee Condition 2, transactions in the read phase need to set R-locks based on their *write set* as well as their *read set*. Let T_{RH} be the set of transactions which hold an R-lock on the same data object in LT. We give the protocol for parallel validation-write in the following.

OCCL-PVW:

Read phase:

```
< for every data object in RS( $T_i$ ) and WS( $T_i$ ) do
```

```

set an R-lock in LT
if a V-lock of that object already exists in LT >
then wait for V-lock release

```

Validation and Write Phase:

```

VALID := true

< for every data object in WS( $T_j$ ) do
  set a V-lock in LT
  if an R-lock of that object exists and  $T_j$  is not the only one in  $T_{RH}$ 
    then VALID := false
  release  $T_j$ 's R-locks in LT
  if not VALID
    then invoke real-time conflict resolution >
  else execute write phase
< release  $T_j$ 's V-locks in LT >

```

In this protocol, a *write-write* conflict between a transaction in its validation/write phase and a transaction in its read phase can be detected and the two write operations on the database are serialized by the locking scheme. Thus Condition 2 is guaranteed.

Compared with OCCL-SVW, OCCL-PVW provides greater concurrency by taking the write phase out of the critical section. On the other hand, OCCL-PVW may cause a higher conflict rate, since there exist not only read-write conflicts but also write-write conflicts now. Which protocol is chosen depends on the type of database system (memory or disk-resident) and the kind of workload (write/read ratio).

7.2.3 Some Implications

Since a locking scheme is used in OCCL, it is necessary to compare OCCL with the two-phase locking (2PL) approach in terms of locking mechanism and implementation overhead.

7.2.3.1 Locking Mechanism

With respect to the locking mechanism, OCCL presented here is different from 2PL. Note that a V-lock is issued at the end of a transaction and the locking period is the duration of validation phase plus write phase. With 2PL, however, the write lock

is issued whenever the update transaction accesses a data object for update and the locking period may be as long as the transaction lifetime. Furthermore, the R-lock used here will not block any concurrent transactions in their read phase, while under 2PL any conflict between read/write locks will block the conflicting transactions.

In addition, OCCL is deadlock-free, even though R-lock and V-lock are used. This is guaranteed by letting the validating transaction set V-locks in the critical section. Since a transaction that has been granted all of its V-locks will not request any lock after leaving its critical section, it is not possible for it to wait for any other (lock-holding) transactions. Thus, a wait-for cycle cannot be formed.

Because OCCL and 2PL use locks, *priority inversion* may occur. With 2PL, priority inversion can be avoided by forcing the high priority transaction to abort the low priority transaction so that a higher priority transaction is never blocked. The problem is more complicated in OCCL than 2PL. Priority inversion may occur in two places with OCCL. One is in the validation phase, where setting the V-lock fails. This problem is addressed by various conflict resolution policies (see Section 7.2.4). Priority inversion may also happen in the read phase when a transaction attempts to set an R-lock for the data object to be accessed. In this case, it is preferable to let the higher priority transaction in the read phase wait for the low priority transaction. This is because the low priority transaction is already in its validation stage or perhaps even in its write phase. Aborting a transaction near completion may cost more, on average, than blocking a higher priority transaction for a limited period of time. To shorten the blocking period, a priority inheritance scheduling scheme can be applied during the validation phase and write phase (see Chapter 5). For instance, the CPU scheduler may raise the process priority of the validating transaction to the highest among the concurrent transactions, thus reducing the time for validation processing. In addition, we may use transaction priority to manage access to the critical section. When more than one transaction is waiting for the critical section, then the one with the highest priority will get access first. Therefore, the worst case blocking time for the higher priority transaction is limited to the delay involved in transaction validation (under OCCL-PVW).

7.2.3.2 The Starvation Problem

Another problem that 2PL and OCCL may encounter is *starvation*. In this context, starvation occurs when transactions are restarted again and again until they miss their deadline. Long transactions have a higher probability of being starved because of their higher probability of access conflict. This results in a lower deadline guarantee ratio for long transactions than for short transactions. In traditional database systems, OCCL may result in more severe starvation because of its high degree of parallelism. Many solutions to the starvation problem have been proposed (e.g., [Pein83, Prad86, Thom90]). These schemes basically rely on limiting the number of transaction restarts: Given the timing constraints in real-time database systems, we use CPU scheduling to address the starvation problem. Based on our earlier studies on transactions with different characteristics (see Chapter 4), here we group transactions into classes by transaction length and assign a weight to each class. The weighting factor is incorporated in the CPU scheduling such that long transactions may have higher priority over short transactions. Using transaction deadline information, the weighted transaction priority is calculated by

$$p = (d - t)/w, \quad d > 0, t > 0, w \geq 1.$$

where d is the transaction deadline, t is the time when CPU scheduling takes place, and w is the length weighting factor. The smaller the p value, the higher the transaction priority. The specific weights used are discussed in Section 7.3. Note that for transactions with the same length, this corresponds to the *earliest-deadline-first* scheduling strategy.

7.2.3.3 Implementation Overhead

In terms of physical implementation, both OCCL and 2PL require a central lock table. For the sake of comparison, we list the lock table operations required by the two schemes.

- OCCL
1. insert a data object ID with an R-lock into the lock table during the read phase;

2. search for a data object ID and convert the corresponding R-lock into a V-lock (if the object has been updated) during the validation phase;
 3. delete a data object ID when an R-lock is released during validation phase or when a V-lock is released at the end of the write phase.
- 2PL
1. search for a data object ID and check its lock compatibility against the lock mode of lock holder(s);
 2. insert a data object ID with *read* or *write* lock into the lock table;
 3. delete a data object ID when a lock is release at the end of the transaction.

It is clear that the physical operations on the lock table are the same for the two protocols. Despite the similarity, there are some differences between OCCL and 2PL. For example, 2PL needs to detect potential deadlock before a lock request is queued while OCCL does not. The implementation overhead of the two concurrency control protocols has been examined through experiments and the results are presented in Section 7.4.

7.2.4 Conflict Resolution

With OCCL, an algorithm is needed to resolve the access conflicts during the validation phase. As discussed above, this conflict resolution should consider transaction priority based on transaction deadlines and length. In other words, the resolution policy should aim at improving the performance of real-time transactions in terms of meeting transaction deadlines. Here are some basic resolution policies:

1. Commit: CMT

Always let the validating transaction commit and abort all the conflicting transactions. This strategy guarantees that as long as a transaction reaches its validation phase, it will always finish. The advantage of this strategy is that the resources (CPU, I/O, etc.) consumed by a finishing (validating) transaction are never wasted. Applying priority-driven CPU scheduling, we expect that transactions with higher priority have a higher probability of reaching the validation phase and, in turn, have a higher probability of committing.

2. Priority abort: PA

Abort the validating transaction only if its priority is less than that of all the conflicting transactions. This strategy takes transaction priority into account, but still favors the validating transaction. It aims at reducing the resources wasted due to aborted transactions.

3. Priority wait: PW

If the priority of the validating transaction is not the highest among the conflicting transactions, wait for the conflicting transactions with higher priority to complete. In some cases, the strategy of aborting conflicting transactions appears too conservative, causing unnecessary transaction abort. Consider the situation where the validating transaction conflicts with transactions which have only read operations. If the validating transaction has a lower priority compared with other conflicting ones, instead of being aborted, it may be deferred. In other words, this transaction is "preempted" from its validation phase and is placed in a waiting queue to wait until all of the conflicting transactions with higher priority finish their validation. One variation of the priority wait strategy is WAIT-50 proposed in [Hari90a], where a validating transaction will wait if at least 50% of the conflicting transactions have a higher priority over the validating transaction. The protocol aims at balancing the wait factor and the priority cognizance.

There can be other variations of the conflict resolution strategy. Since in this study we emphasize the fundamental analysis of OCC performance with respect to its implementation, we only examine the three simple conflict resolution policies discussed above.

7.3 Test Environment

The experiments were conducted on a VAXstation 3100/M38 with two RZ55 disks, one for the database and the other for the log. Table 7.1 summarizes the system parameter settings. Given the physical machine, in order to examine the degree of resource contention (CPU and I/O), the system multi-programming level

Table 7.1: System Parameters

Parameter	Settings
<i>MPL</i> (multi-programming level)	10, 8, 6, 4
<i>DB-Size</i> (database size)	<i>MPL</i> * 100 blocks (6 records/block)
<i>GB-Size</i> (global buffer size)	0 blocks

Table 7.2: Workload Parameters

Parameter	Settings
x (steps per transaction)	4, 6, 8 steps
y (records accessed per trans. step)	4 records
u (computation time per trans. step)	0 units
α (deadline window factor)	2.0 - 6.0
P_w (prob. of write transactions)	0.0 - 1.0
AD (access distribution)	uniform

(*MPL*) is varied from 10 to 4. To examine the effect of data contention in isolation of resource contention, the database size is set proportional to *MPL*. The global buffer is not employed in this study.

Table 7.2 describes the workload parameters and their settings in the experiments. Transaction length $T(x, y, u)$ is varied by changing x (the number of steps) while fixing $y = 4$ (the number of records to be accessed per step) and $u = 0$ (the amount of computation needed per step). We consider two workloads; one where all transactions consist of 6 steps, $P[x = 6] = 1$, and the other where half have 4 steps and the other half 8 steps, $P[x = 4] = P[x = 8] = 1/2$. The latter workload is used particularly for analyzing the starvation problem. The deadline window factor, α , is a timing-related parameter which specifies the deadline distribution of real-time transactions. The smaller the value of α , the tighter the transaction deadlines and vice versa. The probability that a transaction is a write transaction, P_w , is another parameter that directly affects transaction conflict rate. Data accesses are uniform across the entire database.

Table 7.3: Schemes Examined

Scheme	Conflict resolution	CPU scheduling
2PL-NRT	wait	Multi-level feedback queue
2PL-WAIT	wait	Earliest deadline first
2PL-PA	priority abort	Earliest deadline first
OCCL-NRT	commit	Multi-level feedback queue
OCCL-CMT	commit	Earliest deadline first
OCCL-PW	priority wait	Earliest deadline first

Table 7.3 lists the schemes examined in the experiments. We consider two basic concurrency control protocols, 2PL and OCCL, in combination with different conflict resolution policies.² 2PL-NRT and OCCL-NRT are two baselines for the purpose of performance comparisons. They correspond to 2PL and OCCL schemes in non real-time (NRT) database systems, where a multi-level feedback queue algorithm is used for CPU scheduling. In case of access conflict, under 2PL-NRT, the lock-requesting transaction is put into a *wait* queue; under OCCL-NRT, the validating transaction *always commits*. 2PL-WAIT and OCCL-CMT employ priority-driven, preemptive scheduling. Transaction priority is assigned according to *earlier-deadline-first* policy. Still, the two schemes do not take transaction timing constraints into account for resolving access conflict. 2PL-PA and OCCL-PW consider transaction priority for both CPU scheduling and conflict resolution. Note that here the conflict resolution scheme PW refers to WAIT-50 [Hari90a].

Besides the above schemes, we also examined the conflict resolution policy PA for OCCL (i.e., OCCL-PA). The results show that PA performs no better than CMT due to aborts of the validating transaction near its completion. Because of its bad performance, we remove PA from the candidates for OCCL conflict resolution scheme.

The basic metric used for performance evaluation is *deadline guarantee ratio*, which is the percentage of transactions that complete by their deadline. We also collect statistics on transaction abort ratio, blocking time, wasted operations, and CPU and I/O utilizations so as to provide insights into the protocol performance.

²The optimistic concurrency control protocol implemented on RT-CARAT is OCCL-PVW. This is because the testbed is a disk-resident real-time database and cannot afford the long waits (for writing) inherent in OCCL-SVW. In the rest of the paper, we refer to it as OCCL.

The data collection in the experiments is based on the method of replication. The statistical data has 95% confidence intervals whose end points are within 2% of the point estimate for deadline guarantee ratio. In the following graphs, we only plot the mean values of the performance measures.

7.4 Experimental Results

In this section, we present the results of our performance studies from five sets of experiments which deal with protocol overhead, data contention, deadline distribution, resource contention, and transaction length.

7.4.1 Protocol Overhead

In our performance studies, we first compare the implementation overhead of the two types of concurrency control protocols, 2PL and OCCL.

The overhead is measured by the average CPU processing time spent on concurrency control per page. To capture the overhead under all the execution paths, we vary the write probability P_w . At this point, other parameter settings are irrelevant. Figure 7.1 indicates that the implementation overheads of the two protocols are quite close. This is due to the fact that even though the two protocols differ at the logical level (two-phase locking vs. optimistic approach), the underlying physical implementations are very similar. Both protocols rely on a locking technique for data access control, and they both involve hashing operation and lock table management. Despite the similarities, 2PL employs deadlock detection while OCCL does not. However, our previous studies have shown that the deadlock detection on RT-CARAT does not incur significant overhead (see Chapter 4). On the other hand, the implementation of OCCL costs more to maintain read/write sets for each individual transaction. This may be the reason why OCCL has slightly larger overhead than 2PL.

Knowing that the two logically different protocols have similar overhead, we now analyze how the implementation schemes affect the performance of the two protocols.

7.4.2 Data Contention

In this experiment, we examine the protocol performance under different data contention levels by varying the write probability, P_w . We fix the multi-programming level at 8 with $x = 6$ and $\alpha = 5$.

Figure 7.2 shows the *transaction deadline guarantee ratio* for six schemes. As one would expect, the deadline guarantee ratio drops as data contention increases. The performance of two baselines, 2PL-NRT and OCCL-NRT, is consistent with the results from previous studies (e.g., [Care84, Agra87]), i.e., non real-time two-phase locking outperforms non real-time optimistic approach under data and resource contention. Here an interesting observation is that combined with priority-driven preemptive scheduling, the optimistic approach (OCCL-CMT) performs better than two-phase locking (2PL-WAIT). Furthermore, as we incorporate transaction priority into conflict resolution for the two types of protocols, 2PL-PA further increases the deadline guarantee ratio, with respect to 2PL-WAIT, by as much as 17% for $P_w = 0.6$, while OCCL-PW performs only slightly better than OCCL-CMT.

The performance of these schemes may be affected by several factors, such as transaction blocking time, priority inversion and abort ratio. Based on the implementation details, we now explain the results shown in Figure 7.2.

A transaction can be blocked due to access conflict. Under OCCL, this happens in the transaction read phase where an R-lock requesting transaction has to wait for the transaction holding the V-lock. In addition, under OCCL-PW, a validating transaction may be blocked when it conflicts with higher priority transactions in read phase. Under 2PL, blocking can occur at any point along the course of its execution whenever there is a *read-write* or *write-write* conflict. Figure 7.3 depicts the *average transaction waiting time* (in seconds) for each blocking instance. Overall, the waiting time under OCCL scheme is shorter than under 2PL. This is because even though both schemes rely on locking, OCCL shrinks the locking period to the final stage of transaction execution, thus reducing the waiting time. Furthermore, as we discussed in Section 7.2.3.1, applying priority-driven CPU scheduling to OCCL further reduces the waiting time as much as 40% (comparing OCCL-NRT with OCCL-CMT and

OCCL-PW). Compared with OCCL-CMT, the waiting time under OCCL-PW is increased by about 10%, from 0.59 to 0.65 (seconds), for $P_w = 0.2$. On the other hand, when P_w is high, the two schemes perform the same. This is a direct result of the implementation which avoids cyclic V-lock conflicts between two write transactions.

Figure 7.4 shows the *total waiting time* for each transaction run. The trend here is similar to what we observed in Figure 7.3.

As discussed in Section 7.2.3.1, *priority inversion*, a special case of transaction blocking, may occur under both 2PL and OCCL. Figure 7.5 plots the *average number of priority inversions* encountered per transaction run. In 2PL-PA, a high priority transaction will not wait for a low priority transaction when a conflict occurs. Hence 2PL-PA performs the best in terms of avoiding the problem of priority inversion. Note that the priority inversion under 2PL-PA is slightly greater than 0. This is because on RT-CARAT, the high priority transaction is forced to wait for a lower priority transaction if the low priority transaction has already completed its write operations on the database and is about to release its locks. Under OCCL, since a blocked higher priority transaction in read phase has to wait for a V-lock holder to complete its validation phase and write phase, the probability for priority inversion to occur is higher than 2PL-PA, especially when P_w is large. Again, combined with priority-driven CPU scheduling, OCCL-CMT and OCCL-PW perform much better than OCCL-NRT. Under OCCL-PW, a transaction in read phase has less chance to be blocked since the validating transaction might be in the validation-wait state. Thus, the probability of priority inversion under OCCL-PW is slightly lower than that under OCCL-CMT.

Transaction abort rate is another major factor that affects the protocol performance. Figure 7.6 illustrates the *average transaction abort ratio* (i.e., the percentage of transactions aborted due to deadlock or access conflict). Clearly, the wait-oriented schemes, 2PL-NRT and 2PL-WAIT, result in a much lower abort ratio than the abort-oriented schemes - 2PL-PA and OCCL. With a high degree of parallelism and the shorter blocking time (see Figures 7.3 and 4), all the OCCL schemes have a lower abort ratio than 2PL-PA when the data contention is low, but a higher abort ratio when the data contention becomes high. The saturation behavior under 2PL is due to

its increased blocking effect when the data contention becomes high. Among the three OCCL schemes, OCCL-PW has the lowest abort ratio (12% lower than OCCL-CMT for $P_w = 0.2$), since it incorporates a wait mechanism in the validation phase.

In the context of transaction abort, we also measure the *transaction abort length*, i.e., the number of steps that have been processed when a transaction is aborted. To reduce the wasted system resources, the abort length should be as short as possible. Figure 7.7 shows the abort length for the six schemes. Since 2PL-PA may detect access conflict and immediately apply priority abort at any point along the course of its execution, it results in the shortest abort length. 2PL-NRT and 2PL-WAIT allow aborts only when a deadlock is detected. This “pessimistic” wait strategy leads to longer abort lengths. For OCCL, because of its high degree of parallelism, a transaction may proceed until it is aborted by a validating transaction, thus resulting in the abort length longer than for 2PL-PA. Integrated with CPU scheduling, OCCL-CMT and OCCL-PW make the higher priority transaction proceed faster than other lower ones, which means that the abort length of the transactions in read phase becomes shorter than that under OCCL-NRT.

Figure 7.8 depicts the *wasted operations per transaction execution*, i.e., the number of steps wasted for each submitted transaction. This measurement reflects the combined effect of both *transaction abort ratio* and *abort length*.

With respect to *resource consumption*, CPU and I/O utilizations are plotted in Figure 7.9 and Figure 7.10, respectively. As one expects, the wait-oriented schemes, 2PL-NRT and 2PL-WAIT, consume less resources than the abort-oriented schemes - 2PL-PA and OCCL. Due to a high degree of parallelism and the shorter blocking time, OCCL results in higher CPU and I/O utilizations than 2PL-PA.

Having examined the protocol performance in detail, we come to the following points with respect to the performance results demonstrated in Figure 7.2.

- A CPU scheduling algorithm that takes transaction deadlines into account plays an important role in improving the performance of concurrency control protocols, particularly for OCCL which provides a high degree of parallelism and short blocking period.

- The three schemes, 2PL-PA, OCCL-CMT and OCCL-PW, with the fewest priority inversions, perform the best. The difference between the three schemes depends on the amount of wasted operations. 2PL-PA performs the best when data contention is high, since it results in the least wasted operations.
- The wait strategy employed by OCCL-PW has no significant impact on improving OCCL performance. It is the effect of increased waiting time that overshadows the performance gain due to reducing wasted operations. In addition, the implementation scheme for avoiding cyclic V-lock conflicts prevents the wait strategy from taking part in conflict resolution when the probability of write-write conflict is high.

Our results show that 2PL-PA, OCCL-CMT and OCCL-PW are superior to the other protocols. Moreover, the additional experiments with various workload settings show that there is no significant performance difference between OCCL-CMT and OCCL-PW. To simplify the presentation, we only demonstrate and compare the performance of 2PL-PA and OCCL-CMT in the following sections.

7.4.3 Deadline Distribution

The deadline distribution may also affect protocol performance. Extending Experiment 2, we vary the tightness of transaction deadlines while fixing the probability of write transactions, P_w .

We first examine the possible effect of the deadline distribution on performance when low data contention is low, $P_w = 0.2$. Figure 7.11 plots the deadline guarantee ratio versus deadline window factor α . As we have observed in Figure 7.3, when the deadline is loose ($\alpha = 5$), 2PL-PA and OCCL-CMT show similar performance, since transactions complete by their deadline most of the time. Now as α decreases, OCCL-CMT becomes superior to 2PL-PA. This can be explained as follows: When data contention is low, the two protocols have nearly the same probability of priority inversion (see Figure 7.5) and the same amount of *wasted operations* (see Figure 7.8). Under such a condition, the protocol with shorter blocking time (see Figure 7.4) wins.

Next we vary the deadline window factor α under high data contention with $Pw = 0.8$. Figure 7.12 shows the deadline guarantee ratio for 2PL-PA and OCCL-CMT, respectively. In contrast to the results shown in Figure 7.11, here 2PL-PA outperforms OCCL-CMT when the deadlines are relatively loose. This is mainly due to the fact that both the wasted operations and the probability of priority inversion under OCCL-CMT increase as data contention becomes high. Even though 2PL-PA has a longer blocking time, it works better as long as the transaction deadlines are long enough.

7.4.4 I/O Resource Contention

All of the experiments presented above are carried out in a system with I/O resource contention, where the I/O utilization under 2PL-PA and OCCL-CMT was always above 93% with average queue length > 4 (see Figure 7.10). In this set of experiments, we examine the protocol performance in a system where there is no severe resource contention. To do so, we reduce MPL from 8 to 4. Note that the database size is also reduced correspondingly, from 800 to 400 (blocks), so that the level of data contention for $MPL = 4$ is comparable with that for $MPL = 8$.

We first exercise the two concurrency control schemes under low data contention. Figure 7.13 illustrates the deadline guarantee ratio versus deadline window factor α with $Pw = 0.2$. Under such workloads, the I/O utilization drops below 83%. Comparing Figure 7.13 with Figure 7.11, we observe again that the two protocols perform basically the same. We have also observed (not shown here) that the two protocols perform the same with respect to priority inversion and wasted operations, but 2PL-PA results in longer waiting time than OCCL-CMT on locking. This is the main reason why reducing resource utilization does not affect the protocol performance.

The possible effect of resource contention is then examined under high data contention. Figure 7.14 shows the deadline guarantee ratio for $Pw = 0.8$. Comparing it with Figure 7.12, we see the similarity again, despite the drop of I/O utilization from 95% for $MPL = 8$ to 80% for $MPL = 4$ (under OCCL-CMT). Under high data contention, the high abort ratio and the long abort length of OCCL-CMT lead to a larger number of wasted operations, about 25% higher than 2PL-PA. Furthermore,

the chance of priority inversion for OCCL-CMT becomes high (0.16), as compared to 2PL-PA (0.04). These two factors, particularly the priority inversion, degrade the OCCL-CMT performance.

Here we can see that reducing resource utilization does not improve OCCL performance. Under OCCL, due to the use of locking, the effect of priority inversion is sensitive to the duration of the write phase. Therefore, it is the I/O speed that needs to be improved.

7.4.5 Transaction Length

The transactions we have exercised thus far are equal in length ($x = 6$). We now look at workloads with a mix of different transaction lengths. To make data analyzable and yet comparable with previous results, we exercise the workload with two lengths of transactions, $x = 8$ (*long*) and $x = 4$ (*short*), with mean value 6 (i.e., $P[x = 4] = P[x = 8] = 1/2$).

Figure 7.15 shows the transaction deadline guarantee ratio versus P_w for 2PL-PA and OCCL-CMT. Examining the average deadline guarantee ratio, we can see that the result is similar to what we have observed in Figure 7.2 for $x = 6$, i.e., 2PL-PA performs better than OCCL-CMT when data contention is high. However, as we examine the deadline guarantee ratio of long and short transactions, we see that under data contention, OCCL-CMT outperforms 2PL-PA for short transactions while 2PL-PA performs much better than OCCL-CMT for long transactions. In addition, under both schemes, the deadline guarantee ratio of short transactions is much higher than that of long transactions. This observation identifies the starvation problem. Clearly, both of the abort-oriented schemes result in transaction starvation. Due to its high degree of parallelism, OCCL-CMT leads to more severe starvation than 2PL-PA.

We have developed a *weighted priority scheduling* policy to cope with the starvation problem (see Section 7.2.3.2). Figure 7.16 shows the effect of such a CPU scheduling scheme on transaction starvation. Here we associate a weighting factor w to long transactions, varying from 1.0 to 2.6, while fixing w at 1.0 for short transactions. When w is equal to 1.0, the scheduling scheme follows the *earliest-deadline-first*

policy. At this point, the average deadline guarantee ratio coincides with the previous results for $x = 6$ (see Figure 7.11). But, long transactions suffer from the starvation problem. As w increases, under OCCL-CMT, the deadline guarantee ratio of long transactions increases while the deadline guarantee ratio of short transactions decreases. Under 2PL-PA, however, the deadline guarantee ratio of long and short transactions changes slowly. Note that the average deadline guarantee ratio under both schemes does not change with w .

The observation from the experiment indicates that OCCL-CMT is a more flexible scheme in that it can be integrated with an appropriate CPU scheduling policy in order to resolve transaction starvation. This is due to the fact that the transaction blocking time under OCCL-CMT is much shorter than that under 2PL-PA (see Figure 7.3), which gives the CPU scheduler more freedom to carry out priority scheduling. In addition, the weighted priority scheduling scheme follows the conservation law, i.e., the increase of the deadline guarantee ratio for long transactions leads to the decrease of the deadline guarantee ratio for short transactions, and the average deadline guarantee ratio is kept constant. This brings up the question of *fairness* on transaction scheduling. At this point, there is no criterion for a "fair scheduling". In practice, the system designer may choose the weighting factors for different groups of transactions such that their performance requirements can be met.

7.5 Conclusions

In seeking alternatives of two-phase locking for real-time concurrency control, we have proposed an optimistic scheme in connection with priority-based preemptive CPU scheduling. With emphasize on physical implementation, we have developed a set of locking-based protocols for the optimistic approach. The protocols, together with several conflict resolution schemes, have been implemented on the RT-CARAT testbed. Our experimental studies show that integrated with CPU scheduling, optimistic concurrency control does perform better than two-phase locking. Further, the optimistic scheme may not always outperform the two-phase locking protocol which incorporates priority information in its conflict resolution. This is due to the blocking

effect caused by the locking mechanism adopted in the OCC implementation. In particular, the performance difference between the two concurrency control schemes is sensitive to the amount of data contention, but not to the amount of I/O contention. The optimistic scheme performs better than the two-phase locking scheme when data contention is low, and vice versa when data contention is high.

Note that the recent simulation results from [Hari90a, Hari90b] show that under priority-based CPU scheduling, OCC *always* outperforms the two-phase lock protocol that uses priority information in its conflict resolution. Our conclusion differs from this one because the simulation studies in [Hari90a, Hari90b] are conducted at a logical level. Particularly, a *broadcast* mechanism is used in the validation phase of the OCC protocol and no overhead is taken into account for the broadcast operation. The "degraded" performance of our optimistic approach becomes apparent only because we considered the implementation details and since ours is a testbed, the overheads of the implementation manifest themselves in the performance figures. This experimental study indicates that the physical implementation schemes required for OCC have a significant impact on the performance of real-time optimistic concurrency control.

In this chapter, we have also explored the starvation problem with respect to the deadline guarantee ratio for transactions of different length. The performance studies show that both the abort-oriented two-phase locking and optimistic approaches result in starvation for long transactions. Integrated with the proposed *weighted priority scheduling*, the optimistic concurrency control scheme exhibits a greater flexibility in coping with the starvation problem.

Even though this study reveals some weakness of the optimistic approach with respect to its implementation, we believe that this approach is still a candidate for real-time concurrency control owing to its high degree of parallelism and its flexibility in handling conflict resolution and in closely related to its physical implementation scheme, its performance can be further improved by adding certain processing components into the system. For example, regarding the locking-based scheme developed in this work, if a disk controller can perform the *write* operations in transaction validation order and it can also intelligently manage the order of *read* and *write* operations [Abbo90], the V-lock holding period can be largely reduced. The integration

of concurrency control with I/O scheduling is an interesting topic for future work. Another example for improving the performance of the optimistic approach is the use of a database cache which can accommodate data pages to be accessed by restarted real-time transactions. The development of such a technique also remains part of future work.

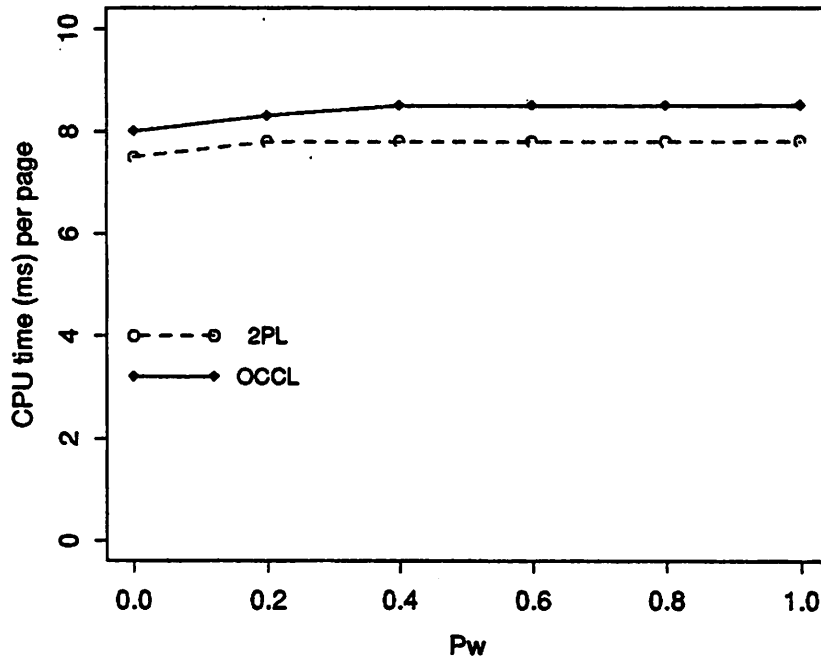


Figure 7.1: Concurrency Control Overhead

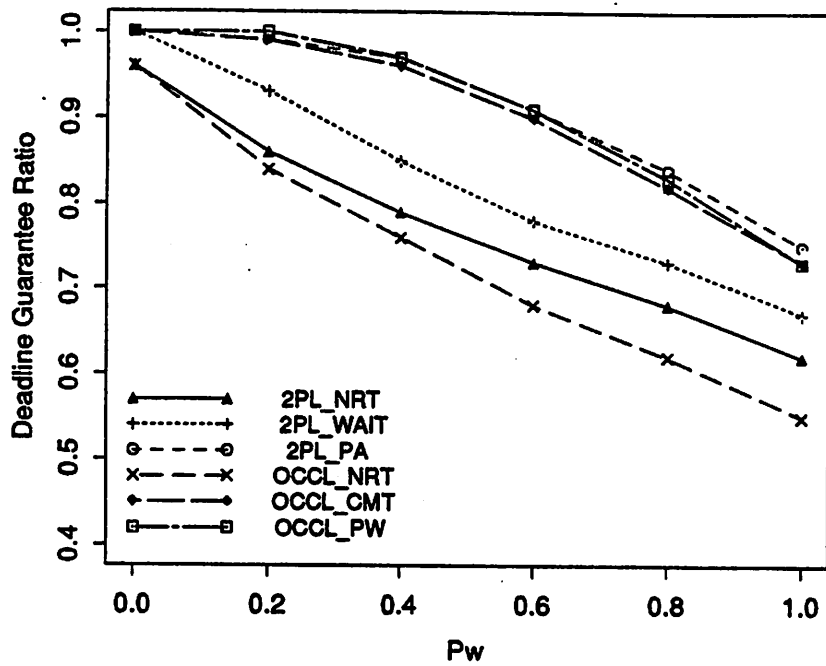


Figure 7.2: Data Contention, $MPL = 8, x = 6, \alpha = 5$

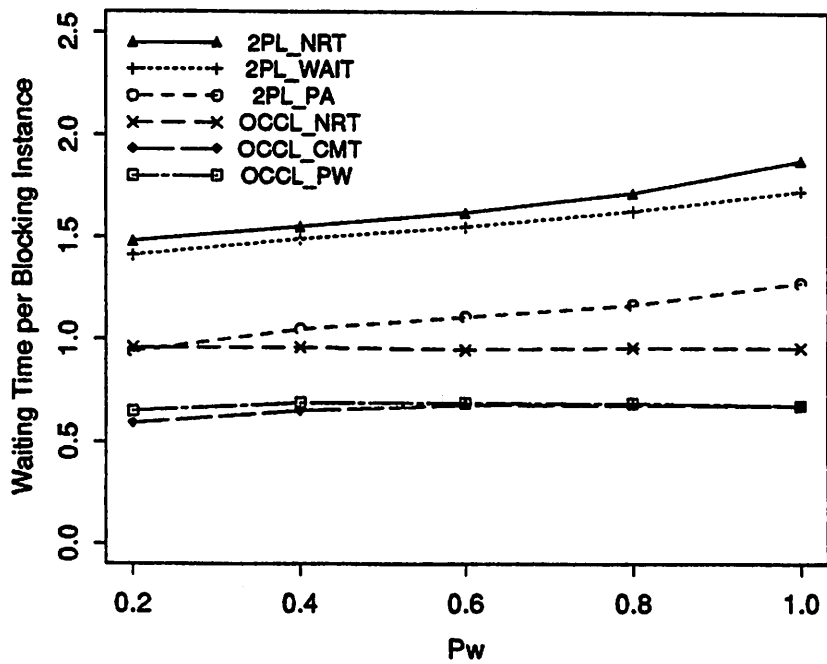


Figure 7.3: Data Contention, $MPL = 8, x = 6, \alpha = 5$

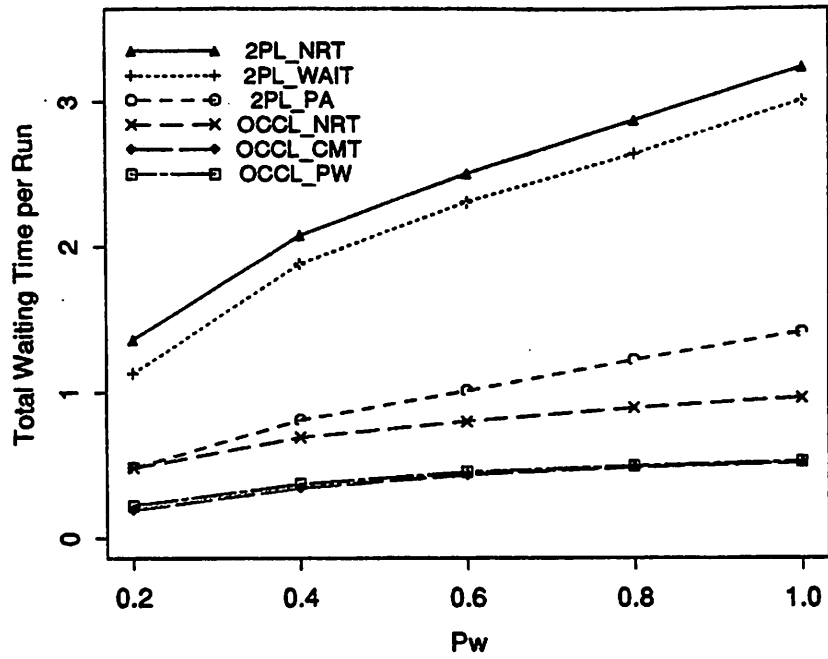


Figure 7.4: Data Contention, $MPL = 8, x = 6, \alpha = 5$

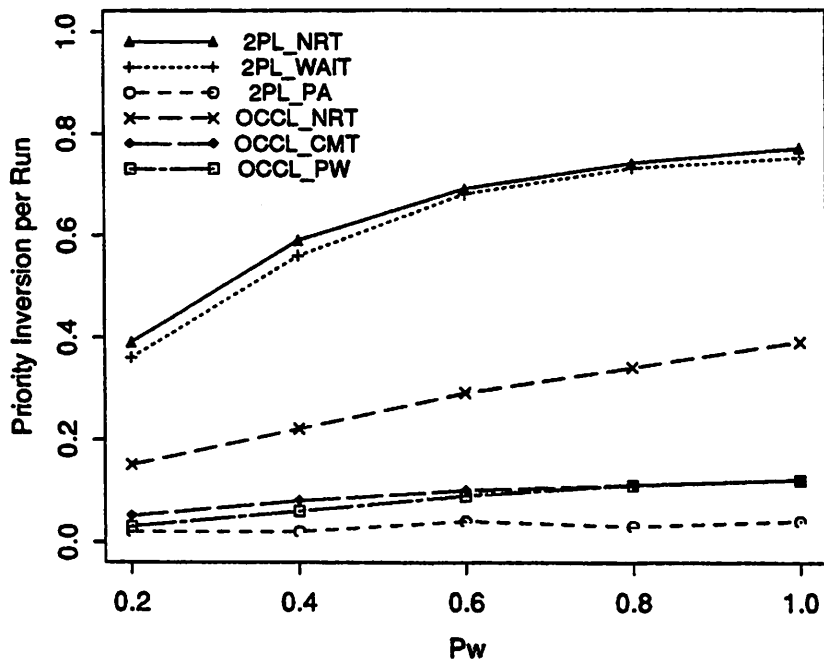


Figure 7.5: Data Contention, $MPL = 8, x = 6, \alpha = 5$

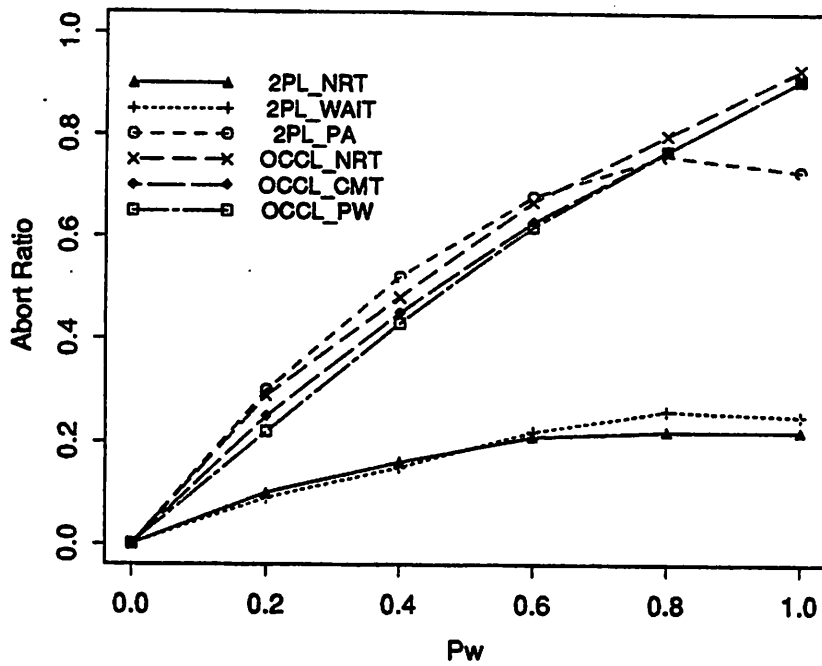


Figure 7.6: Data Contention, $MPL = 8, x = 6, \alpha = 5$

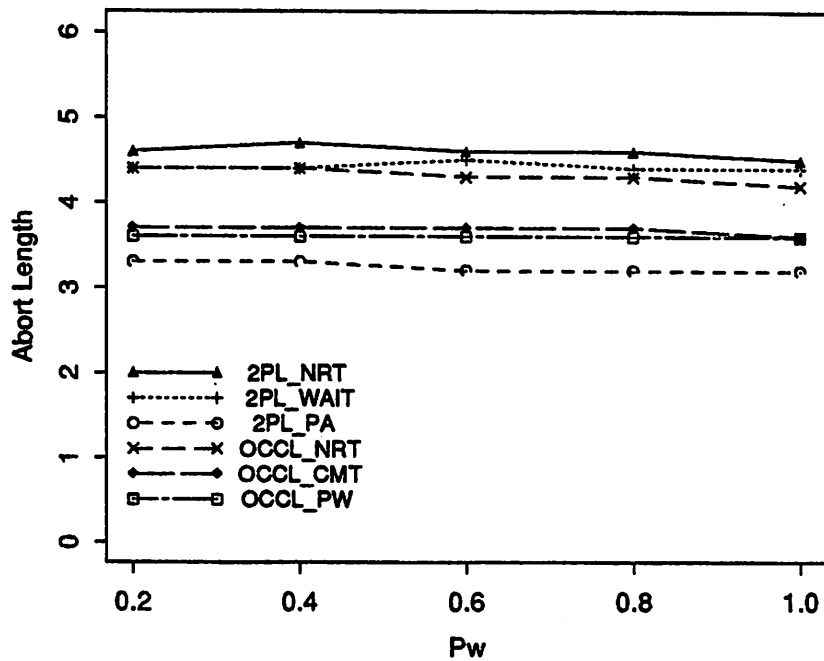


Figure 7.7: Data Contention, $MPL = 8, x = 6, \alpha = 5$

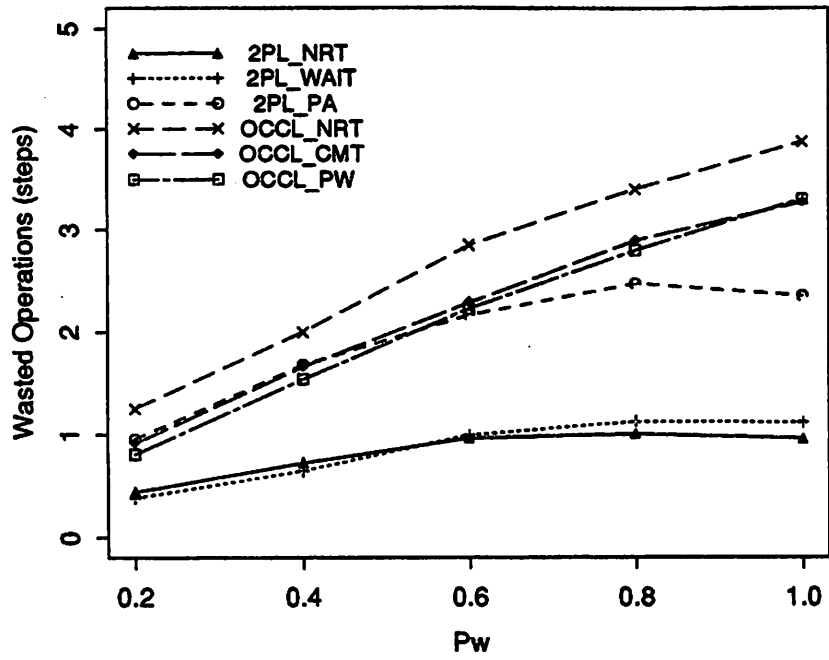


Figure 7.8: Data Contention, $MPL = 8, x = 6, \alpha = 5$

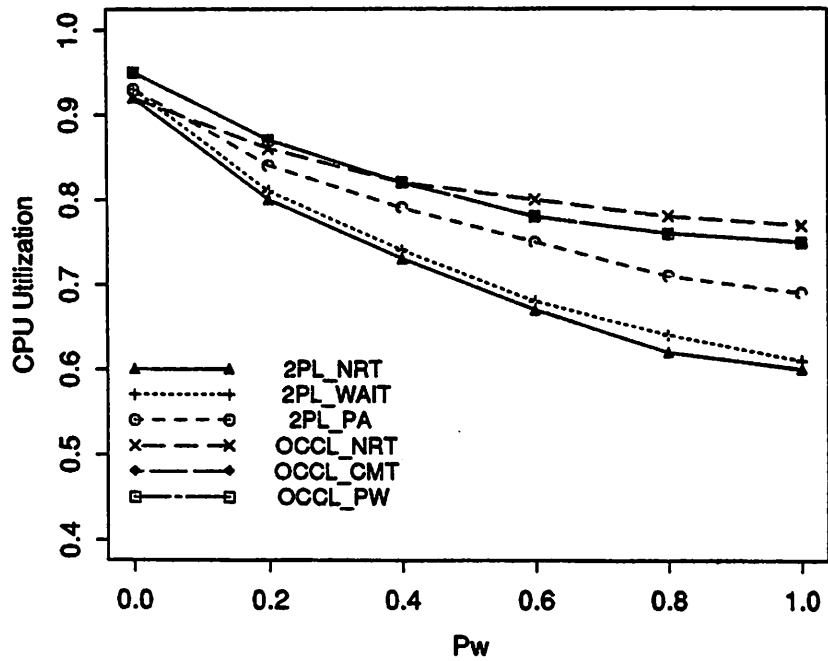


Figure 7.9: Data Contention, $MPL = 8, x = 6, \alpha = 5$

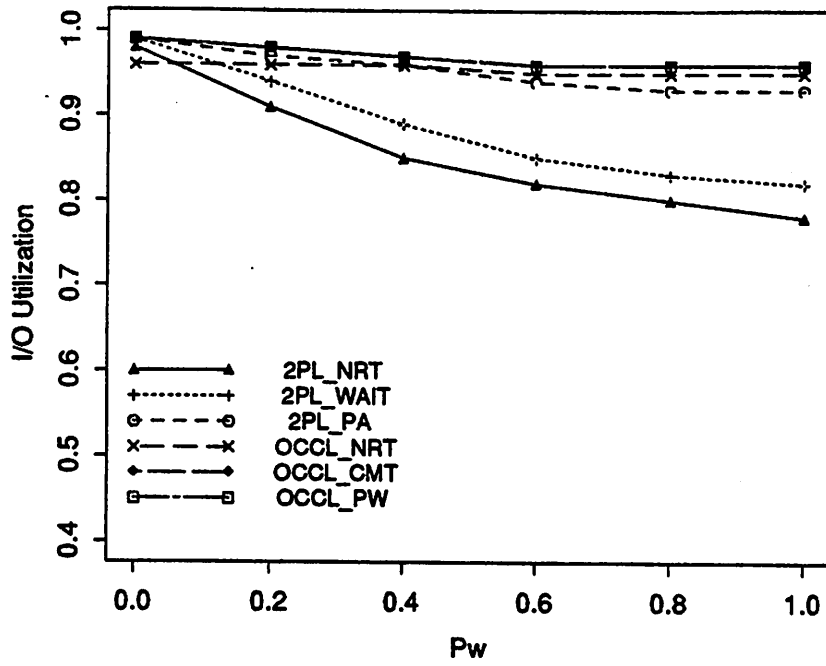


Figure 7.10: Data Contention, $MPL = 8, x = 6, \alpha = 5$

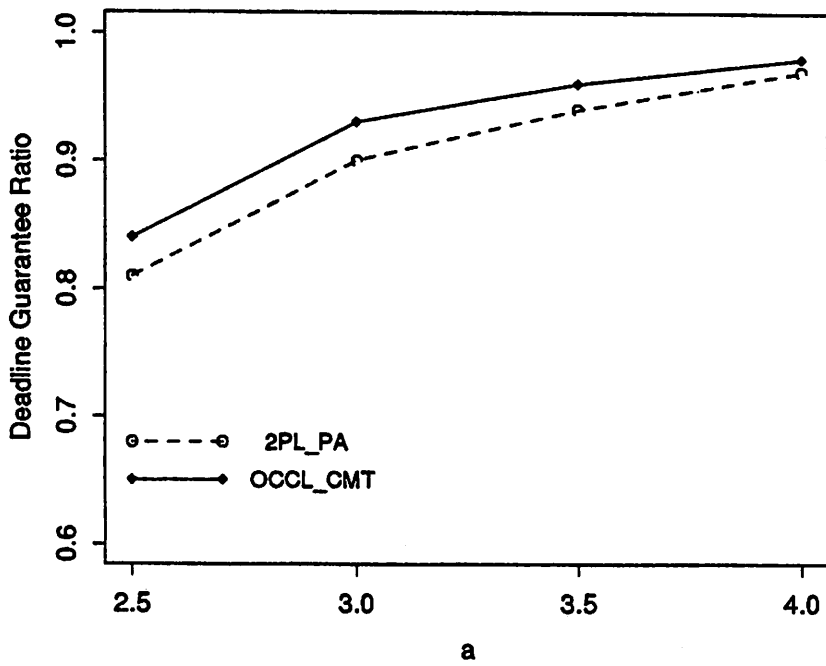


Figure 7.11: Deadline Distribution, $MPL = 8, x = 6, P_w = 0.2$

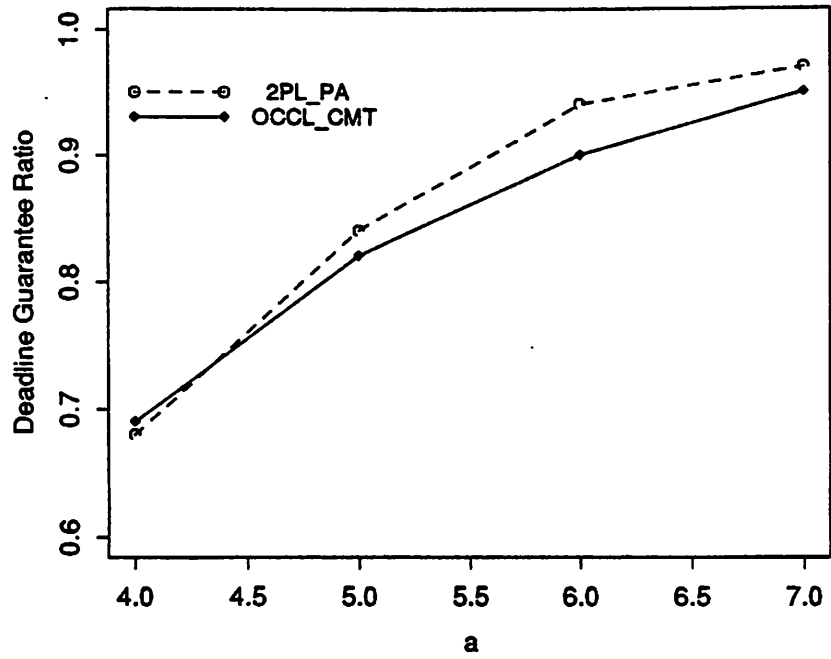


Figure 7.12: Deadline Distribution, $MPL = 8$, $x = 6$, $P_w = 0.8$

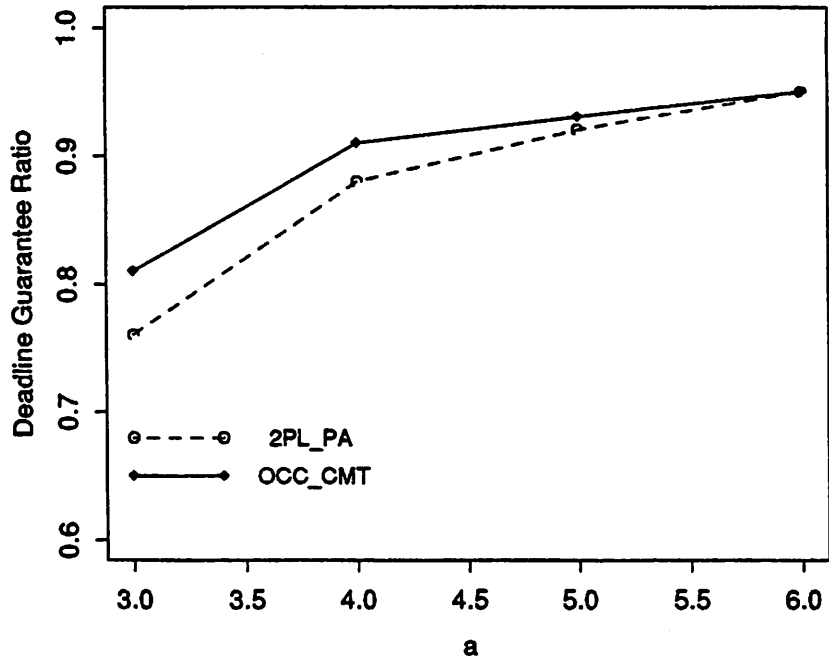


Figure 7.13: Deadline Distribution, $MPL = 4$, $x = 6$, $P_w = 0.2$

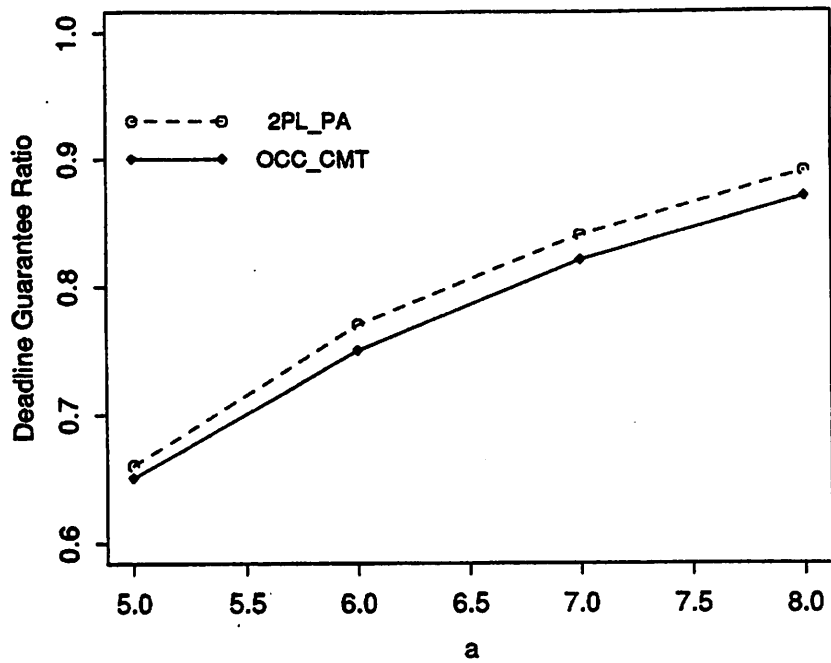


Figure 7.14: Deadline Distribution, $MPL = 4, x = 6, P_w = 0.8$

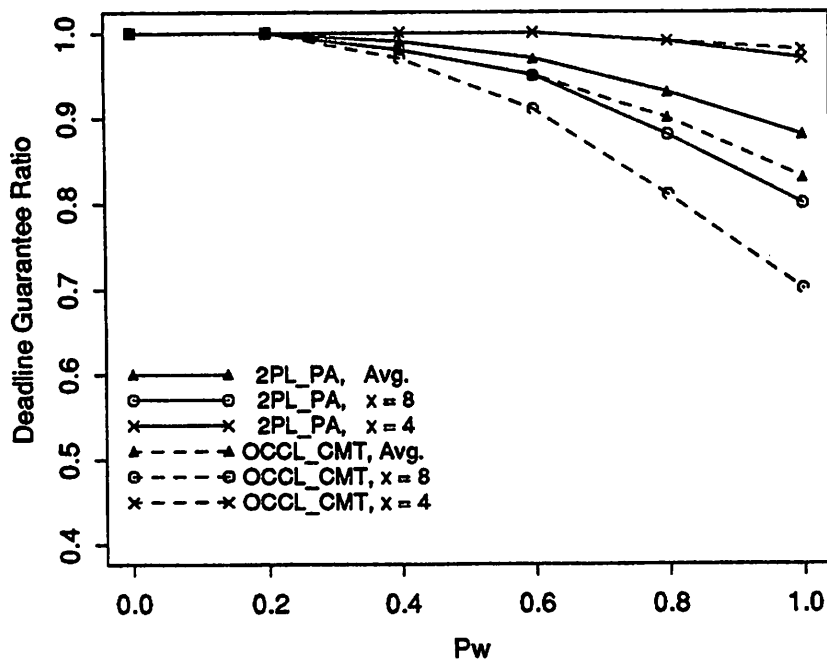


Figure 7.15: Mixed Transactions, $MPL = 8, x = [4, 8], \alpha = 5$

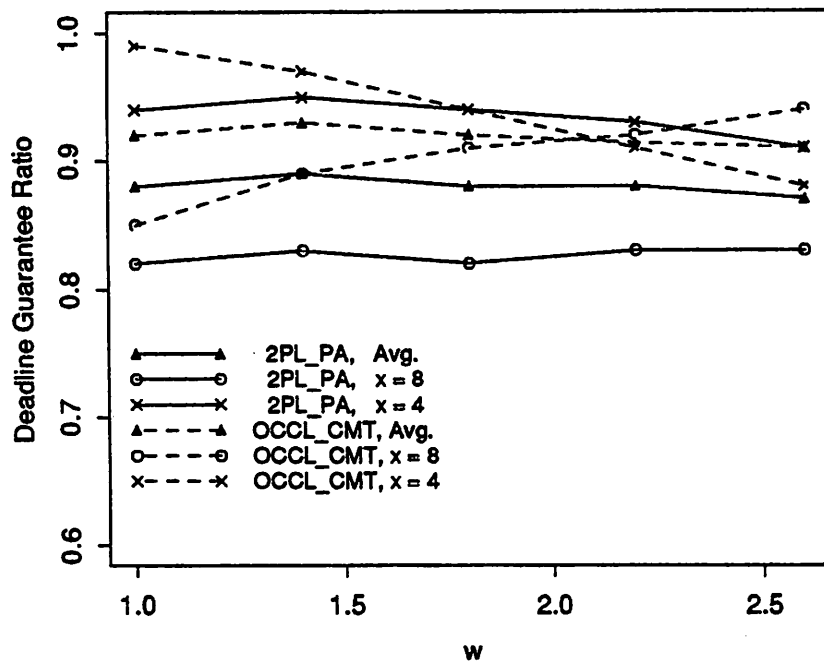


Figure 7.16: Mixed Transactions, $MPL = 8$, $x = [4, 8]$, $P_w = 0.2$, $\alpha = 2$

CHAPTER 8

SUMMARY AND FUTURE RESEARCH

8.1 Summary and Conclusions

Real-time databases are becoming increasingly important in a wide range of applications, such as computer-integrated manufacturing, stock trading, air traffic control, and command and control systems. In real-time database systems, transaction processing must satisfy not only data consistency requirements but also timing constraints. Depending on the degree of timing constraints, a real-time database system can be either *hard* or *soft*. In this thesis we have focused on soft real-time database systems where data consistency is required and defined by the notion of *serializability*.

Real-time transaction processing is complex. It involves multiple functional components in an entire database system. It is further complicated by the extensive interactions among these components. In pursuing the goal of meeting timing constraints, real-time transactions need to be well scheduled at every processing component along the course of their execution. This is necessary because even a single entity in the system which ignores timing issues may undermine the best efforts of processing components which do account for timing constraints. Thus, in this thesis, we have taken an integrated approach to study real-time transaction processing. We consider transaction scheduling in a systemwide manner by taking key processing components and their interactions into account.

Taking the integrated approach, we have developed real-time algorithms for CPU scheduling, concurrency control, conflict resolution, deadlock resolution, transaction wakeup, transaction restart, and buffer management. In order to evaluate the algorithms and to better understand the operational behavior of those processing components in a real system, we have implemented a real-time database testbed called

RT-CARAT. Using the testbed, we have conducted various experiments with a wide range of parameter settings and statistical validity.

In examining the effect of the different processing components on overall system performance, our experimental results indicate

- that the CPU scheduling algorithm has the most significant impact. It may improve the transaction deadline guarantee ratio by as much as 30%. In addition, it is found, for instance, that switching CPU scheduling from a multi-level feedback queue algorithm to an earliest-deadline-first policy completely reverses performance ordering of two-phase locking and optimistic concurrency control protocols;
- that concurrency control and the associated conflict resolution schemes are secondary, but still influential, factors in the integrated system. For example, some real-time oriented conflict resolution protocols may achieve up to 18% performance improvement with respect to transaction deadline guarantee ratio; and
- that the real-time oriented buffer management schemes do not significantly improve system performance over non real-time buffer management schemes.

We have further investigated the interaction between CPU scheduling and concurrency control in the context of the priority inversion problem. To address the problem, we developed a conditional priority inheritance scheme that capitalizes on the advantages of both priority inheritance scheme and priority abort scheme. We have clarified through experiments

- that the basic priority inheritance technique is sensitive to the priority inheritance period. Due to the *life-time blocking* problem under two-phase locking, the basic priority inheritance scheme, which has been shown to be effective in real-time operating systems, does not work well in real-time database systems. Rather, the conditional priority inheritance scheme that we developed and a simple priority abort scheme perform well for a wide range of system workloads; and

- that overall, long time blocking resulting from priority inversion is a more serious problem than wasting of system resources. The priority abort and conditional priority inheritance schemes, which attempt to eliminate or reduce the period of priority inversion, perform better than the basic priority inheritance and simple wait schemes, which attempt to reduce resource waste. This is especially true when transaction deadlines are loose.

We have particularly studied the real-time concurrency control component. We have proposed an optimistic scheme, in connection with priority-driven preemptive CPU scheduling, as an alternative to two-phase locking. Based on a locking mechanism to ensure the correctness of the OCC implementation, we developed a set of optimistic concurrency control protocols. In addition, we developed a weighted priority scheduling algorithm to cope with the starvation problem encountered by long transactions. Our experimental studies indicate

- that optimistic concurrency control, compared with two-phase locking, performs better when integrated with priority-driven CPU scheduling. This result is contrary to conventional wisdom in database systems;
- that the optimistic approach may not always outperform the two-phase locking scheme which takes transaction priority into account in resolving data access conflicts. This is due to the blocking effect caused by the locking mechanism adopted in the OCC implementation; and
- that integrated with the weighted priority scheduling algorithm, optimistic concurrency control exhibits greater flexibility in coping with the starvation problem than two-phase locking.

In this thesis we have also developed and studied a real-time transaction model which captures both transaction deadline and criticalness (importance). The experimental results show that these two factors, criticalness level and deadline distributions, strongly affect transaction performance. Criticalness is a more important factor than the deadline with respect to the performance goal of maximizing the deadline guarantee ratio for high critical transactions and maximizing the value imparted by

real-time transactions. This has important implications for real-time scheduling research, which to date has focussed primarily on time constraints independent of the value of tasks (transactions).

This work is experimental in nature. Although the testbed implementation and experimental studies provide a first-hand experience with a real system and an improved understanding of the functional requirements and operational behavior of real-time databases, there are some weak points. First, the design-implementation-evaluation cycle takes longer time, compared with simulation studies, since the system development must take every detailed step into account. Second, a physical machine may limit performance studies to some extent. For example, CPU power is always fixed for any given machine. This limitation makes it difficult to examine the effect of resource contention on system performance. Finally, a physical system may constrain investigation on certain processing components. The I/O subsystem on RT-CARAT is such an example. Thus, in order to overcome the limitations on the experimental work, it is necessary to conduct simulation studies in parallel. Simulation is relatively flexible in selecting system parameters, and it may provide, quickly, a basic understanding of any object being investigated.

8.2 Future Extensions

Our research work can be extended in several directions. First, alternatives of our testbed architecture can be explored. As discussed in Chapter 6, the memory space of the current RT-CARAT is allocated to a global buffer and to a pool of private buffers for individual transactions (for recovery purpose), respectively. To reduce the overhead caused by data transfer between the global and private buffers and to have a better utilization of memory space, an alternative scheme is to use a single memory space for both recovery and data buffering. The management of such a buffer organization should be investigated in the real-time context. In addition, as discussed in Chapter 7, the buffer management needs to be incorporated into real-time optimistic concurrency control in order to reduce possible blocking time. Furthermore, we have observed that one of the major overheads in the current RT-CARAT implementation is due to the extensive message communication between the

transaction manager and data managers. This message overhead will be eliminated if a shared memory space between the transaction manager and data managers is used for inter-process communication.

Another issue that has not been well addressed so far is the identification of real workloads. Since research on real-time databases is still in its infancy, there are no "standard workloads" for real-time transactions. The value function that has been used in this study is tentative. Real workloads are application dependent. It is necessary and important to determine and to categorize them for performance studies.

Next, this research can be extended to a distributed real-time database. Currently, we have focused on a real-time database that resides on a single site. Our experience suggests a number of important issues that will have to be addressed in the design of an integrated distributed real-time database. For example, in order to enforce serializability in a distributed setting, optimistic concurrency control requires global validation as well as local validation. However the global validation operation may lead to wait and even deadlock situation. On the other hand, distributed two-phase locking may result in distributed deadlock. Here the principal questions are, "How should data conflicts be resolved in a distributed setting?", "Should conflict resolution favor the use of blocking or aborts?", and "Which scheme, two-phase locking or optimistic approach, is better suited for distributed real-time concurrency control?" Another issue, for example, is distributed resource scheduling. We have discovered in our current study that the choice of the priority assignment policy has tremendous effect on the performance of a single site real-time database. This is further complicated in a distributed system with the presence of nested transactions whose component transactions may execute concurrently on different sites. Specifically, deadlock may occur if transaction priorities are not set in a consistent fashion at all the sites visited by a transaction. Hence, great care will be required during the development of rules for assigning priorities to component transactions to ensure that this problem will not happen and that they will not be counterproductive, even in the absence of deadlock.

Another direction for future work is to consider hard real-time constraints. Although tremendous research efforts have been made in hard real-time systems in

dealing with hard real-time constraints [Stan88a], little work has been done in the context of hard real-time database systems. Research into hard real-time databases will deal with a variety of new problems, from scheduling to concurrency control, and from system modeling to architecture design. For example, in a hard real-time database, data consistency may not be absolutely necessary for some real-time data which have limited lifetimes. On the other hand, due to stringent timing constraints, requirements on data consistency may have to be relaxed. Hence, not all transactions possess the traditional ACID property (atomicity, concurrency, isolation and durability) and, in turn, new concurrency control protocols need to be developed to support such types of transactions. Furthermore, in real-time systems, real-time data may exist at different levels with respect to their time, synchronization, consistency and permanence properties and there exist certain correlations between the different levels [Stan88b]. To guarantee different timing constraints and data consistency requirements, a real-time system may consist of different types of subsystems in a hierarchical structure, with a hard real-time system being at bottom, a soft real-time database at top, and a hard real-time database inbetween. Developing a proper interface between the subsystems will be a challenging task.

Finally, future research in real-time transaction processing may explore semantics-based concurrency control. Semantic information about stored data is available in certain applications [Badr87, Weih88, Schw84]. Using semantic information can reduce the number of conflict situations and increase concurrency. It has been shown that concurrency control protocols that utilize such information can be very effective in reducing transaction response time in object-oriented database environments [Badr90]. Clearly, real-time database systems shall take this advantage. Here concurrency control needs to take both semantic and timing information into account. Furthermore, since semantic-based concurrency control requires more information, and in turn results in more overheads, it is necessary to investigate the effect of the overheads on transaction scheduling.

BIBLIOGRAPHY

- [Abbo88a] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, March 1988.
- [Abbo88b] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.
- [Abbo89] Abbott, R. and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 14th VLDB Conference*, Aug. 1989.
- [Abbo90] Abbott, R. and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.
- [Agra85] Agrawal, R. and D.J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transaction on Database Systems*, Vol.10, No.4, Dec. 1985.
- [Agra87] Agrawal, R., M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems*, Vol.12, No.4, Dec. 1987.
- [Badr87] Badrinath, B. and K. Ramamritham, "Semantics-Based Concurrency Control Protocol: Beyond Commutativity," *International Conference on Data Engineering*, Feb. 1987.
- [Badr90] Badrinath, B. and K. Ramamritham, "Performance Evaluation of Semantics-Based Multilevel Concurrency Control Protocols," *ACM-SIGMOD International Conference on Management of Data*, Jan. 1990.
- [Bern87] Bernstein, P., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.
- [Buch89] Buchmann, A.P., D.R. McCarthy, M. Hsu and U. Dayal, "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, Feb. 1989.
- [Care84] Carey, M.J., M.R. Stonebraker, "The Performance of Concurrency Control Algorithms for Database Management Systems," *Proceedings of the 10th VLDB Conference*, 1984.
- [Care89] Carey, M.J., R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference*, 1989.

- [Chen90] Chen, M. and K. Lin, "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," to appear in *Real-Time Systems*, Vol.2, No.4, Dec. 1990.
- [Chn90] Chen, S., J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *A Technical Report, COINS 90-77*, University of Massachusetts, Aug. 1990.
- [Chn91] Chen, S. and D. Towsley, "Performance of a Mirrored Disk in a Real-Time Transaction System," *To appear in Proc. of SIGMETRICS'91*, May 1991.
- [Dan90a] Dan, A., D.M. Dias and P.S. Yu, "Database Buffer Model for the Data Sharing Environment", *Proceedings of the 6th Data Engineering Conference*, LA, Feb. 1990.
- [Dan90b] Dan, A., D.M. Dias and P.S. Yu, "An Approximate Analysis of the LRU and FIFO Buffer Replacement Schemes", *ACM SIGMETRICS*, May 1990.
- [Dan90b] Dan, A., D.M. Dias and P.S. Yu, "Buffer Modelling for a Data Sharing Environment with Skewed Data Access", submitted to *IEEE Transaction on Computer Systems*, 1990.
- [Daya88a] Dayal, U. et. al., "The HiPAC Project: Combining Active Database and Timing Constraints," *ACM SIGMOD Record*, March 1988.
- [Daya88b] Dayal, U., "Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.
- [Dias87] Dias, D.M., B.R. Iyer, J.T. Robinson and P.S. Yu, "Design and Analysis of Integrated Concurrency-Coherency Controls," *Proceedings of the 13th VLDB Conference*, Brighton, 1987.
- [Dias89] Dias, D.M., B.R. Iyer, J.T. Robinson and P.S. Yu, "Integrated Concurrency-Coherency Controls for Multisystem Data Sharing," *IEEE Transaction on Software Engineering*, Vol.15, No.4, April, 1989.
- [Effe84] Effelsberg, W. and Theo Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol.9, No.4, Dec. 1984.
- [Elha84] Elhardt K. and R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems*, Vol.9, No.4, Dec. 1984.
- [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," *ACM Communication*, 19(11), November 1976.
- [Hard84] Harder, T. "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, Vol. 9, No.2, 1984.

- [Hari90a] Haritsa, J.R., M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," *PODS*, 1990.
- [Hari90b] Haritsa, J.R., M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.
- [Hsu88] Hsu, M., R. Ladin and D.R. McCarthy, "An Execution Model for Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.
- [Jau90] Jauhari, R., "Priority Scheduling in Database Management Systems," *Computer Sciences Technical Report #959*, University of Wisconsin - Madison, Aug. 1990.
- [Kear89] Kearns, J.P., and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review*, Vol.17, No.1, May 1989.
- [Kirk89] Kirk, D., "SMART (Strategic Memory Allocation for Real-Time) Cache Design," *Proceedings of the 10th Real-Time Systems Symposium*, Santa Monica, CA, Dec. 1989.
- [Kohl86b] Kohler, W. and B.P. Jenq, "CARAT: A Testbed for the Performance Evaluation of Distributed Database Systems," *Proceedings of the Fall Joint Computer Conference*, Nov. 1986.
- [Kort90] Korth, H. F., N. Soparkar and A. Silberschatz, "Triggered Real-Time Databases with Consistency Constraints," *Proceedings of the 16th VLDB Conference*, Brisbane, Australia, Aug. 1990.
- [Kung81] Kung, H.T. and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol.6, No.2, June 1981.
- [Lin89] Lin, K.J., "Consistency issues in real-time database systems," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Jan. 1989.
- [LinS90] Lin Y. and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the 11th Real-Time Systems Symposium*, Dec. 1990.
- [Liu88] Liu, J. W., K. J. Lin and X. Song, "Scheduling Hard Real-Time Transactions," *The 5th IEEE Workshop on Real-Time Operating Systems and Software*, May, 1988.
- [Lock86] Locke, C.D., "Best-Effort Decision Making for Real-Time Scheduling," *Ph.D. Dissertation*, Canegie-Mellon University, 1986.
- [Mena82] Menasce, D.A. and T. Nakanishi, "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," *Information Systems 7(1)*, 1982.

- [Pein83] Peinl, P. and A. Reuter, "Empirical Comparison of Database Concurrency Control Schemes," *Proceedings of the 9th VLDB Conference*, Florence, Italy, 1983.
- [Prad86] Pradel, U., G. Schlageter and R. Unland, "Redesign of Optimistic Methods: Improving Performance and Applicability," *Proc. IEEE 2nd Int. Conf. on Data Engineering*, 1986.
- [Rajk89] Rajkumar, R., "Task Synchronization in Real-Time Systems," *Ph.D. Dissertation*, Dept. of Electrical and Computer Engineering, Carnegie-Mellon University, Aug. 1989.
- [Sacc86] Sacco, G.M. and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM Transaction on Database Systems*, Vol.11, No.4, Dec. 1986.
- [Schw84] Schwarz, P. and A. Z. Spector, "Synchronizing Shared Abstract Data Types," *ACM Transaction on Computer Systems*, Aug. 1984.
- [Sha87] Sha, L., R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocol: An Approach to Real-Time Synchronization," *Technical Report*, Computer Science Dept., 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.
- [Son87] Son, S.H., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th Real-Time Systems Symposium*, Dec. 1987.
- [Son88] Son, S.H., "Semantic Information and Consistency in Distributed Real-Time Systems," *Information and Software Technology*, Vol. 30, Sep. 1988.
- [Son89] Son, S.H. and C.H. Chang, "Priority-Based Scheduling in Real-Time Database Systems," *Proceedings of the 15th VLDB Conference*, 1989.
- [Son90] 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, May 1990.
- [Song90] Song, X. and K. J. Liu, "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," *COMPSAC '90*, Oct. 1990.
- [Stan88a] Stankovic, J.A. and K. Ramamritham, editors, "Tutorial: Hard Real-Time Systems," *Computer Society Press of the IEEE*, 1988.
- [Stan88b] Stankovic, J.A. and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.
- [Stea76] Stearns, R. E., P. M. Lewis and D. J. Rosenkrantz, "Proceedings of the 7th Symposium on Foundations of Computer Science," 1976.

- [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, May 1990.
- [Vrbs88] Vrbsky, S. and K. J. Lin, "Recovering Imprecise Transactions with Real-Time Constraints," *Proceedings of the Symposium on Reliable Distributed Systems*, Oct. 1988.
- [Weih88] Weihl, W., "Commutativity-Based Concurrency Control for Abstract Data Types," *IEEE Transactions on Computers*, Dec. 1988.