TRANSACTION SCHEDULING IN FIRM REAL-TIME DATABASE SYSTEMS

by

Jayant Ramaswamy Haritsa

Computer Sciences Technical Report #1036

August 1991

TRANSACTION SCHEDULING

IN

FIRM REAL-TIME DATABASE SYSTEMS

by

JAYANT RAMASWAMY HARITSA

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
UNIVERSITY OF WISCONSIN — MADISON
1991

ABSTRACT

Database applications that arise in the real-time domain have to meet timing requirements. At the database system interface, these timing requirements may translate into completion deadlines for transactions. In this thesis, we study the problem of transaction scheduling in database systems supporting applications with deadlines. In particular, we focus on applications with *firm deadlines*, which consider transactions that do not complete by their deadlines to be worthless and therefore discard late transactions. Within the firm-deadline context, two cases that differ in the utility associated with completing a transaction before its deadline are examined here. In the *same-value* case, all transactions have equal utility from the application's perspective and the goal of the real-time database system is to maximize the number of in-time transactions. In the *multiple-value* case, different transactions have different utilities to the application and the goal of the real-time database system is to maximize the total value of the in-time transactions.

In this thesis, we present new real-time concurrency control protocols and priority assignment policies for transaction scheduling in the same-value and the multiple-value cases. The concurrency control protocols are based on the *optimistic* approach to maintaining database consistency. The priority policies are based on simple real-time scheduling observations and *adapt* their priority assignment to match the database operating environment. Results from a wide range of simulation experiments indicate that real-time optimistic concurrency control protocols are fundamentally better suited than their locking-based counterparts to the firm-deadline environment. The results also show that adaptive priority policies provide superior performance to fixed priority policies. In particular, for the multiple-value case, priority policies that adaptively change the relative importance of transaction values and deadlines deliver considerably better performance than policies that establish fixed tradeoffs between these characteristics.

In summary, this thesis sheds light on issues involved in real-time transaction scheduling, and presents new scheduling algorithms that come closer to meeting the challenges of the real-time domain.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Miron Livny, for instilling in me the values of a good scientist and for continually challenging me to put in my best effort. I owe a special debt of gratitude to Prof. Mike Carey for his unstinting help, sound advice, and constant encouragement. He has truly been a pillar of strength in both technical and personal terms. I am grateful to both Miron and Mike for their generous financial assistance which allowed me to work full-time on my research. I would also like to record my appreciation of the other members of my committee, Professors Jeff Naughton, Mary Vernon, and Guri Sohi, for their constructive criticisms and suggestions.

During my stay in Madison, I have counted on many good friends for sharing the lighter aspects of graduate life. Mike Franklin, S. Seshadri, Manolis Tsangaris, V. Srinivasan, B. Narendran, John Wood, V. Balasubramaniam, Mani Ganesh, Sriram Vajapeyam, Dhruva Ghoshal, Vikram and Sarita Adve, Rajiv Jauhari, Sanjay Krishnamurthy, Paul Bober, Scott and Lisa Leutenegger, Amarnath Mukherjee, Gautam Das, and the members of the Racquetball League have provided me with a continual supply of good cheer and helped relieve the stresses that are always part of working towards a difficult goal. A special note of thanks to my housemates, Sesh, Srini, and Rajesh, for being always willing to engage in my favorite pasttime of arguing into the middle of the night on just about every topic under the sun. I am grateful to my officemate, Lui Chan, for giving me virtually exclusive use of the office during the thesis-writing period.

My research has been greatly assisted by the members of the Systems Lab. In particular, I would like to thank Jim Luttinen, Mitali Lebeck, and Mike Litzkow for their prompt and cheerful response to my requests. The department secretaries, Kathy, Lorene, Sheryl, and Lynn, were always willing to help me out with administrative matters, and I am indebted to them for their assistance.

Finally, and most importantly, I thank my family for their unwavering belief in my abilities and for restoring my spirits whenever my problems seemed to get the better of me. Amma and Anna, this thesis is dedicated to you. Suchitra, Jaya, Naniji and Nanaji, your moral support was essential to the success of my efforts.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iii
TABLE OF CONTENTS iv
Chapter 1. INTRODUCTION 1
1.1 Motivation 1
1.2 Real-Time Database Systems 3
1.2.1 Types of Real-Time Database Systems 4
1.3 Transaction Scheduling 6
1.3.1 Priority Assignment 7
1.3.2 Resource Service 8
1.3.3 Concurrency Control 9
1.4 RTDBS Architecture 10
1.5 Organization of Dissertation 12
Chapter 2. SURVEY OF RELATED RESEARCH 14
2.1 Introduction 14
2.2 Time Constraints 14
2.3 Task Scheduling 16
2.4 Transaction Scheduling 17
2.4.1 Same Value Transactions 17
2.4.2 Multiple Value Transactions 20
2.5 Database Consistency 20
2.6 RTDBS Projects 22
2.7 Our Research 22
Chapter 3. MODEL AND METHODOLOGY 24
3.1 Introduction 24
3.2 Modeling a Real-Time Database System 24
3.2.1 Database 26
3.2.2 Source 26
3.2.2.1 Transaction Page Assignment 27
3.2.2.2 Transaction Deadline Assignment 27
3.2.2.3 Transaction Value Assignment 29
3.2.3 Transaction Manager 30
3.2.4 Resource Manager 31
3.2.5 Concurrency Control Manager 32
3.2.6 Sink 32
3.3 Experimental Methodology 32
3.4 Performance Metrics 34
3.5 Fixed Parameter Settings 35

3.6 Resource Time Computations 36
Chapter 4. THE POTENTIAL OF OPTIMISTIC CONCURRENCY CONTROL 37
4.1 Introduction 37
4.2 Concurrency Control Algorithms 38
4.2.1 2PL-HP 38
4.2.2 OPT-BC 39
4.3 2PL-HP versus OPT-BC 40
4.3.1 Blocking 40
4.3.2 Restarts 41
4.3.3 Priority Reversals 41
4.4 Experiments and Results 43
4.4.1 Experiment 1: Baseline Experiment 44
4.4.2 Experiment 2: Pure Data Contention 46
4.4.3 Experiment 3: Deadline Tightness / Slackness 48
4.4.4 Experiment 4: Priority Reversals 50
4.4.5 Other Experiments 52
4.5 Conclusions 54
Chapter 5. REAL-TIME OPTIMISTIC ALGORITHMS 56
5.1 Introduction 56
5.2 Limitations of OPT-BC 58
5.3 Prioritized Optimistic Algorithms 59
5.3.1 OPT-SACRIFICE 60
5.3.2 OPT-WAIT 61
5.3.3 WAIT-50 63
5.4 Experiments and Results 64
5.4.1 Experiment 1: Fixed Slack Ratio 65
5.4.2 Experiment 2: Variable Slack Ratio 68
5.4.3 Experiment 3: Wait Control Mechanism 71
5.4.4 Other Experiments 73
5.5 Conclusions 74
Chapter 6. ADAPTIVE EARLIEST DEADLINE 76
6.1 Introduction 76
6.2 Priority Mappings 77
6.2.1 Earliest Deadline (ED) 78
6.2.2 Latest Deadline (LD) 78
6.2.3 Random Priority (RP) 78
6.2.4 No Priority (NP) 78
6.2.5 Adaptive Earliest Deadline (AED) 79
6.2.5.1 Group Assignment 79
6.2.5.2 Priority Assignment 80
6.2.5.3 Discussion 81
6.2.5.4 HIT Capacity Computation 82
6.2.5.5 Feedback Process 84

6.3 Concurrency Control Algorithms 85
6.4 Experiments and Results 85
6.4.1 Experiment 1: Resource Contention (RC) 86
6.4.2 Experiment 2: Resource and Data Contention (RC + DC) 91
6.4.3 Experiment 3: Bursty Arrivals 93
6.4.4 Experiment 4: Concurrency Control 94
6.5 Conclusions 96
Chapter 7. VALUE AND DEADLINE 97
7.1 Introduction 97
7.2 Priority Mappings 100
7.2.1 Earliest Deadline (ED) 100
7.2.2 Highest Value (HV) 100
7.2.3 Value-inflated Deadline (VD) 100
7.2.4 Value-inflated Relative Deadline (VRD) 101
7.3 Concurrency Control Algorithms 101
7.4 Experiments and Results 102
7.4.1 Experiment 1: Resource Contention (RC) 102
7.4.1.1 Baseline Experiment 103
7.4.1.2 Increased Value Spread 107
7.4.1.3 Decreased Value Spread 108
7.4.1.4 Skewed Value Distribution 110
7.4.2 Experiment 2: Data Contention (DC) 112
7.4.2.1 Baseline Experiment 112
7.4.2.2 Skewed Value Distribution 115
7.4.3 Other Experiments 117
7.5 Conclusions 118
Chapter 8. HIERARCHICAL EARLIEST DEADLINE 120
8.1 Introduction 120
8.2 Hierarchical Earliest Deadline (HED) 120
8.2.1 Bucket Assignment 121
8.2.2 Group Assignment 123
8.2.3 Priority Assignment 123
8.2.4 Discussion 124
8.3 Concurrency Control Algorithms 125
8.4 Experiments and Results 125
8.4.1 Experiment 1: Uniform Value Distribution 126
8.4.2 Experiment 2: Skewed Value Distribution 128
8.4.3 Experiment 3: Concurrency Control 129
8.5 Conclusions 131
Chapter 9. SUMMARY AND FUTURE RESEARCH 133
9.1 Summary of Results 133
9.2 Future Research Directions 136
REFERENCES 138

Appendix A. 143
Appendix B. 145
Appendix C. 147

CHAPTER 1

INTRODUCTION

1.1. Motivation

In an effort to attract customers, many service enterprises, such as fast-food restaurants and courier companies, promise to deliver service to customers within a deadline. If the promise is not kept, the customer is compensated in some fashion; for example, by providing the service free of charge. Assume, for the moment, that you are the manager of such a restaurant and that you decide when each customer order is serviced. Consider the situation where you have two outstanding orders, one with a close deadline and the other whose deadline is a while away. Now, would you serve the urgent order first in the hope of meeting both deadlines, or would you serve the less urgent one first to make certain of meeting at least one deadline? If the two orders were to have different billings or different compensations, would you decide differently? You, the manager of a fast-food restaurant, have just been faced with one of the most fundamental problems of a real-time system – deciding how to schedule a set of tasks with time constraints so as to optimize some metric. In this dissertation, we address such scheduling problems in the context of **real-time database systems**, that is, database management systems whose workload is composed of transactions with deadlines.

Applications that deal with large quantities of information use database management systems for data storage, processing and retrieval. Our interest in *real-time* database systems stems from the increasing number of data-intensive applications that are faced with timing requirements. Such applications have arisen in diverse scientific, financial, manufacturing and military areas, and include aircraft control, communication systems, stock trading, factory automation and robotics. For data-intensive real-time applications, the ability of the underlying database system to satisfy transaction timing constraints becomes a key factor in determining both the feasibility of the application and the extent of its use. The real-time performance of the database system depends on several factors such as the system architecture, the processor speeds, etc. For a given system configuration, however, the primary performance determinant is the policy for scheduling transaction access to system resources, since this policy determines *when* service is provided to a transaction. Herein lies our motivation for studying scheduling issues in real-time database systems.

A real-time database system represents a "marriage" between the hitherto separate areas of real-time systems and database systems, as shown in Figure 1.1. Research in these two fields has, for the most part, been addressing different concerns. For the past several decades,

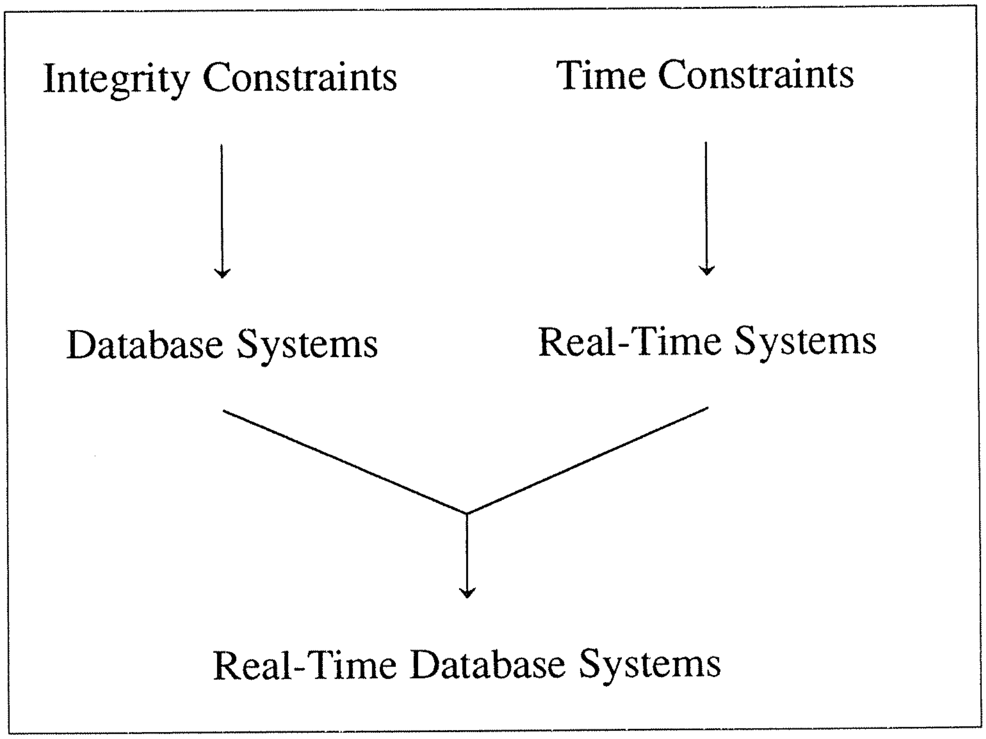

Figure 1.1: Family Tree of Real-Time Database Systems

the main focus of attention in the real-time area has been the problem of scheduling tasks with time constraints. Over the same period, one of the major challenges addressed by database researchers has been to efficiently implement the transaction model, which provides the properties of atomicity, serializability and permanence, and thereby guarantees data integrity. Conversely, real-time studies have paid little attention to maintaining the consistency of shared data, while the concept of time constraints is foreign to database systems. In designing a transaction scheduling policy for a real-time database system, an integrated approach is required to *simultaneously* enforce data integrity constraints and satisfy transaction timing constraints. This *dual* requirement makes real-time transaction scheduling more complex than task scheduling in conventional real-time systems or transaction scheduling in conventional database systems.

For the most part, scheduling algorithms used in current real-time systems assume complete a-priori knowledge of task arrival times and task processing requirements. This permits an off-line production of the best schedule through the use of optimization techniques. For database applications, however, such knowledge is usually unavailable and the execution pattern of transactions is often data-dependent. It is therefore not possible to statically compute the best schedule in most cases. As a result, transaction scheduling in real-time database systems has to be dynamic and based on heuristics. These added complexities exacerbate the scheduling problem in real-time database systems. In this study, our objective is to develop transaction scheduling algorithms that are tuned to achieving the performance objectives of real-time database systems.

1.2. Real-Time Database Systems

Our view of a Real-Time Database System (RTDBS) pictures it to be a transaction processing engine whose workload is composed of transactions with individual timing constraints. Typically, a time constraint is expressed in the form of a **deadline**, that is, the application submitting the transaction would like it to be completed before a certain time in the future. We assume that, from the application's perspective, having a transaction complete just before its deadline expires is no different than having it finish earlier. Therefore, in contrast to a conventional database management system (DBMS), where the goal usually is to minimize transaction response times, the emphasis in an RTDBS is on meeting transaction deadlines.

Apart from assigning a deadline to each transaction, some real-time applications also assign a **value** to each transaction. The value reflects the return the application expects to receive if the transaction completes *before* its deadline. For example, consider an airline reservation database system that allows customers to call in their reservations. Each reservation transaction has a time constraint, which is the delay that the customer is willing to endure before hanging up. Satisfying the request for a high-priced ticket is more beneficial to the airline than satisfying the request for a cheaper ticket since the high-priced ticket generates greater revenue. In this scenario, therefore, the value of a transaction is the fare paid by the customer. A key point to note here is that value and deadline are fundamentally different properties. The value reflects the transaction's *worth* while the deadline reflects the transaction's *urgency*.

At any time, transactions in the RTDBS can be divided into two groups: **feasible** transactions and **late** transactions. A transaction is feasible if its remaining service requirement is less than the remaining time to its deadline (if the remaining service requirement is not known, it is assumed to be zero). Therefore, a feasible transaction retains a possibility of completing before its deadline. A late transaction, on the other hand, has either already missed its deadline or has no chance of successfully meeting it. The RTDBS executes feasible transactions until they either complete before their deadline or are detected to be late. Various application-dependent policies, described in the next section, exist to deal with late transactions.

1.2.1. Types of Real-Time Database Systems

Real-time applications can be grouped into three categories: **Hard Deadline**, **Firm Deadline**, and **Soft Deadline**. The classification is based on how the application is impacted by the violation of time constraints. For a hard deadline application, missing a deadline is equivalent to a catastrophe. Life-critical applications, such as flight control systems or missile guidance systems, belong to this category. For firm deadline or soft deadline applications, however, missing deadlines leads to a performance penalty but does not entail catastrophic results. The distinction between firm deadline and soft deadline applications lies in their view of late tasks. For firm deadline applications, completing a task after its deadline has expired is of no utility and in fact may be harmful. Therefore, late tasks are required to be permanently aborted and discarded. For soft deadline applications, however, there is some diminished utility to completing tasks even after their deadlines have expired. Financial and manufacturing applications usually fall under either firm deadline or soft deadline categories.

Database systems for efficiently supporting hard deadline real-time applications, where *all* transaction deadlines have to be met, appear infeasible [Stan88a]. This is because there is usually a large variance between the average case and worst case execution times of a transaction. The large variance is due to transactions interacting with the operating system, the I/O subsystem, and with each other in unpredictable ways. Guaranteeing transactions under such circumstances requires an enormous excess of resource capacity to account for the worst possible combination of concurrently executing transactions [Stan88a].

Database systems for firm deadline or soft deadline applications, however, appear feasible since the system is only expected to make a "best effort" to meet transaction deadlines. In this dissertation, we restrict our attention to firm deadline applications, and late transactions are therefore considered to be worthless and are discarded by the database system. It is our opinion that understanding scheduling issues for firm deadline applications will provide a foundation to address the more general framework of soft deadline applications, where transactions may retain some utility to completing after their deadline.

Our studies consider both firm-deadline applications that do and do not assign values to transactions. For applications where transactions are not assigned values, we assume that the goal of the RTDBS is to maximize the number of transactions that complete before their deadline. For applications where transactions have associated values, we assume that the goal of the RTDBS is to maximize the total value of the transactions that complete before their deadline. Conceptually, the former class of applications can be viewed as a special case of the latter wherein all transactions are assigned the same value. In the remainder of this thesis, therefore, we will use the term *same value* to refer to the former class, and the term *multiple value* to refer to the latter.

1.3. Transaction Scheduling

An RTDBS transaction scheduling algorithm is composed of four components: Service Eligibility, Priority Assignment, Resource Service, and Concurrency Control, as shown in Figure 1.2. The **Service Eligibility** component decides which transactions are allowed to execute in the database system. All other transactions are immediately (and permanently) discarded from the system. The **Priority Assignment** component assigns priorities to transactions that are eligible to execute in the system. The **Resource Service** component maps a service discipline to each hardware resource. Finally, the **Concurrency Control** component maintains database consistency and resolves data conflicts.

A sample RTDBS transaction scheduling algorithm, described in terms of its components, is the following: { Feasible Only; Earliest Deadline; Priority PR (CPU), Priority LRU (Memory), Priority HOL (Disk); 2PL-HighPriority }. This algorithm implements the Feasible Only eligibility test [Abbo88b] which discards late transactions and permits only feasible transactions to execute in the system. The Earliest Deadline priority mapping [Liu73] assigns higher priority to transactions with earlier deadlines. The Priority PR [Pete86] and Priority LRU [Jauh90] scheduling disciplines provide priority-based preemptive service at the processors and memory buffers, respectively, while Priority HOL [Klei76] provides priority-based non-preemptive service at the disks. Finally, the 2PL-HighPriority concurrency control protocol [Abbo88b] maintains database consistency with a two-phase locking scheme and resolves data conflicts in favor of higher priority transactions.

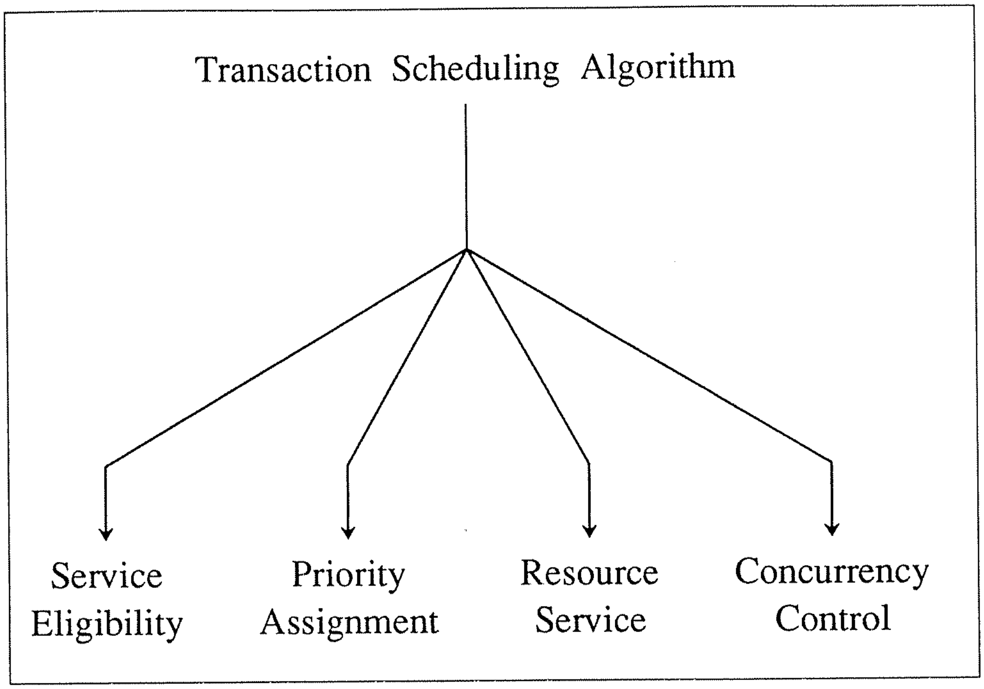

Figure 1.2: Scheduling Algorithm Components

To put the scheduling issue into perspective, transaction scheduling algorithms in conventional database systems also have a Resource Service component and a Concurrency Control component. They do not, however, have a Service Eligibility component since all transactions have to be executed to completion. They also usually do not have a Priority Assignment component since transactions are expected to be treated on an equal basis.

1.3.1. Priority Assignment

Most conventional DBMSs strive to minimize the mean response time of transactions. Consequently, improving the response time of one transaction at the expense of another is not considered an improvement in system performance. In an RTDBS, the objective is to complete transactions before their deadlines. For this objective, improving the response time of one transaction at the expense of another *can* result in a system performance improvement. To illustrate this point, consider the following possibility: Two transactions arrive at the database system, one with a tight deadline and the other with a loose deadline. If the transactions share the system resources on an equal basis, the urgent transaction misses its deadline while the slack transaction completes in time. Alternatively, if the urgent transaction progresses at the expense of the slack transaction, it completes before its deadline. The slack transaction then, by virtue of its loose deadline, completes in time despite its retarded progress.

From the above discussion, it is clear that providing differential service to transactions in an RTDBS can result in a performance improvement. Priority assignment is a mechanism to indicate the service precedence among transactions to the database system resources. In the above example, assigning higher priority to the urgent transaction would have resulted in the desired performance improvement.

Assigning priorities, however, is not always as clearly defined as in the case described above. It is easy to come up with situations where the performance impacts of different priority assignments are hard to evaluate, and making a choice is therefore difficult. The problem of assigning priorities is even more complex when transactions have different values. This is because a tradeoff has to be made between transaction value and deadline characteristics, and it is not clear what this tradeoff should be. In some situations, performance may be improved by starting each transaction slowly and increasing the pace as its deadline approaches. In such cases, the priority assignment process has to be dynamic.

1.3.2. Resource Service

The main physical resources in a database management system are the processors, the memory buffers, and the disks. In conventional DBMSs, scheduling disciplines like Round Robin, LRU, and Elevator [Pete86] are used at the processors, buffers, and disks, respectively. Such scheduling disciplines are unacceptable, however, in a real-time database system since they are priority-indifferent. In an RTDBS, service should be provided in a manner that reflects transaction priorities, since the priority assignment is based on achieving performance objectives. One approach is to make the resources behave like **priority preemptive resume** servers [Jauh90]. In such servers, the highest-priority requester is always the one receiving service. This is achieved by preempting lower priority requesters from use of the resource. With this approach, higher priority transactions do not "see" transactions with lower priority.

There are several practical reasons, however, that make the priority preemptive resume goal not entirely possible or desirable in an RTDBS [Jauh90]. First, hardware restrictions prevent certain actions, such as disk service, from being preemptable. Second, preemption does not come for free but involves overhead such as context switching at the CPU. Third, certain database actions, such as acquiring a lock on a data item, are non-preemptable due to either correctness requirements or performance reasons. Finally, servicing requests in absolute priority order may result in degraded performance for certain resources. For example, scheduling requests in priority order at the disk may perform worse than scan-based disciplines that try to minimize the distance moved by the disk head in serving multiple outstanding requests. Therefore, for all the above reasons, a strict priority preemptive resume approach is not practical in an RTDBS. The goal, however, is to develop resource scheduling disciplines that efficiently approximate this behavior.

1.3.3. Concurrency Control

Data, like processors, memory, and disks, is also a resource in a database system. It differs, however, from these physical resources in its correctness requirements. This requires data scheduling to be treated differently from physical resource scheduling. The standard notion of data correctness in a DBMS is **serializability**; that is, the outcome of concurrent execution of transactions, as reflected in the stored data, should be the same as that produced by executing the transactions in some serial order. Concurrency control protocols preserve database correctness by resolving non-serial data accesses in a manner that induces a serialization order among the conflicting transactions. Several serialization mechanisms have been proposed [Bern87], such as **locking**, **validation** and **timestamping**. Each mechanism takes a different approach to achieving serializability. A question to be decided in developing an

RTDBS concurrency control protocol is which mechanism, or combination of mechanisms, is most suited to achieving the performance objectives of the system.

Two methods, **blocking** and **restarts**, are available for resolving data conflicts, and virtually all concurrency control protocols use some combination of these methods. A blocking decision causes one of the conflicting transactions to wait for the other to complete, while a restart decision causes one of the conflicting transactions to be rolled-back. In a conventional DBMS, the decision on which transaction's progress is to be hindered is usually based on either the order of data access or on the relative ages of the conflicting transactions. This is not suitable, however, in the context of an RTDBS. A transaction with a high priority should not be either restarted or blocked by a transaction with a low priority. Therefore, an RTDBS concurrency control protocol is expected to take transaction priorities into consideration when making data conflict decisions.

1.4. RTDBS Architecture

A high-level "black-box" view of a real-time database system is shown in Figure 1.3. In this model, the input process to the RTDBS is an arrival stream of transactions with each transaction T having an associated arrival time A_T, deadline D_T, and value V_T. There are two transaction output streams from the RTDBS, InTime and Late. Transactions that complete before their deadline join the InTime stream, while transactions that miss their deadline (or are certain to do so) join the Late stream.

Transaction scheduling within the RTDBS incorporates the architecture shown in Figure 1.4. The *eligibility monitor* tracks the eligibility status of all transactions executing in the system. A transaction that becomes ineligible is immediately removed from the system and discarded. The *priority mapper* unit assigns a priority P_T to each transaction on its arrival, and this priority is used by the hardware resource schedulers and the concurrency control mechanism to distinguish transactions. It is possible that there is feedback in the priority assignment process, causing the priority of a transaction to change with time; in this case, the

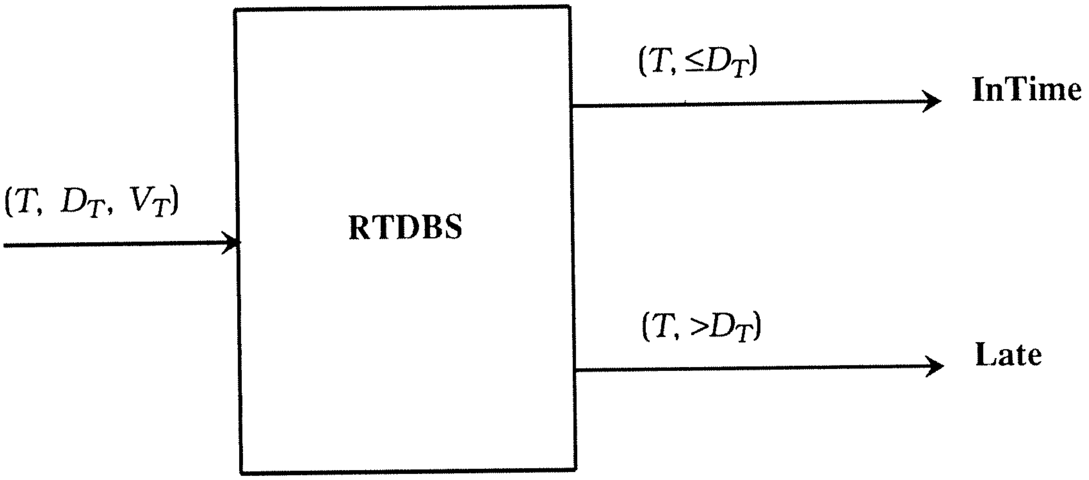

Figure 1.3: Black-Box View of RTDBS

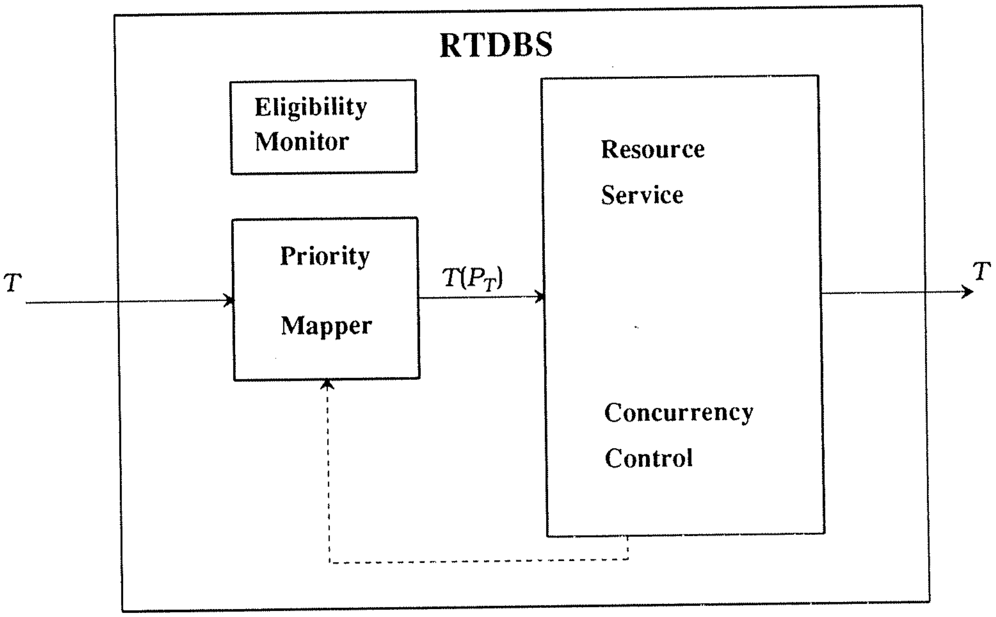

Figure 1.4: Transaction Scheduling Architecture

change in priority is transmitted by the priority mapper to the transaction. This priority architecture shields the internal database mechanisms from the details of the priority assignment process, and is modular since it separates *priority generation* from *priority usage*.

As mentioned earlier, we restrict our attention in this study to applications that have firm deadlines. This means that late transactions are immediately discarded and do not execute to completion. A second restriction is that the RTDBS has no a-priori knowledge of either transaction processing requirements or transaction data accesses. This means that, from the RTDBS perspective, transactions are distinguished only by their arrival time, deadline and value characteristics. It also means that transactions are detected to be late only when they *actually* miss their deadline since the RTDBS cannot estimate the remaining service requirements of a transaction (i.e. transactions remain feasible until their deadlines expire).

1.5. Organization of Dissertation

Our goal in this thesis is to profile the performance of various transaction scheduling algorithms in an RTDBS supporting applications with firm deadlines. In order to do this, we develop a detailed performance model of a real-time database system, and construct a simulator based on this model. We then experiment with various transaction scheduling algorithms over a wide range of workloads and system configurations.

Our performance study focuses on evaluating alternatives for the concurrency control and priority assignment components. The results of the simulation experiments indicate that transaction scheduling algorithms based on optimistic concurrency control protocols are fundamentally better suited than their locking-based counterparts to the firm-deadline environment. The results also show that scheduling algorithms which adapt their transaction priority assignment to the current operating environment provide better performance than algorithms with fixed priority assignment policies. In particular, for the multiple-value case, scheduling algorithms that adaptively change the relative importance of transaction values and deadlines perform considerably better than algorithms that establish fixed tradeoffs between these characteristics.

The remainder of this dissertation is organized in the following fashion: Chapter 2 is devoted to a review of published research on real-time database systems. A performance model of a real-time database system is described in Chapter 3. This chapter also includes an overview of the methodology, metrics and workload generation process used in the simulation studies that follow. In our initial simulation experiments, we focus on applications where all transactions have the same value. For such applications, the performance of existing concurrency control protocols is evaluated in Chapter 4, and a new protocol is presented in Chapter 5. The focus of Chapter 6 is on the priority assignment component, and a new priority mapping is developed in this chapter. Our second series of simulation experiments consider applications where transactions may differ in their assigned values. For such applications, the performance of existing priority mappings is evaluated in Chapter 7, and a new priority mapping is presented in Chapter 8. Finally, Chapter 9 summarizes the main contributions of our study and outlines future avenues to explore.

CHAPTER 2

SURVEY OF RELATED RESEARCH

2.1. Introduction

Real-time database systems are a recent concept, and have received attention only in the last few years. Consequently, research into issues pertaining to these systems is still in its infancy. There have been only a handful of papers published on real-time database systems, virtually all of which are from researchers in academia. In this chapter, a brief summary of relevant research on RTDBSs and related areas is presented.[1] Time constraints, transaction scheduling, database consistency issues, and prototype projects are covered, with special attention paid to papers that include performance studies. Only centralized real-time database systems are considered in the scope of our discussion; the reader is referred to [Lin88, Sha88, Sing88, Son87, Son90b] for coverage of issues related to distributed RTDBSs.

2.2. Time Constraints

A model of generalized time constraints for real-time tasks was presented in the seminal work of [Jens85, Lock86]. The key idea of the model is that the completion of a task has a **value** to the application that can be expressed as a function of the task completion time. These *value functions* are a powerful mechanism for expressing the time constraints of tasks since they model a task's real-time requirements over a time window, unlike deadlines which represent only a single instant in time. A soft deadline task, for example, is captured by the

[1] The list of references at the end of the thesis is more exhaustive in its coverage.

value function shown in Figure 2.1(a) – if the task completes before its deadline, its value is obtained by the parent application; as the task is delayed beyond its deadline, the value received decreases exponentially. The value function of a firm deadline task is shown in Figure 2.1(b) – the task loses its value if it misses its deadline.

While the value function model was developed in the context of task scheduling in real-time operating systems, it is equally applicable to expressing transaction time constraints in a real-time database system. Discussions on identifying value functions for real-time transactions are included in [Abbo87, Abbo88a, Buch89, Huan89]. The notion of associating a *negative value* or **penalty** with transactions that miss their deadline is also considered in [Abbo87, Abbo88a]. The penalty reflects the loss suffered by the application due to the transaction not completing in a timely fashion.

An alternative model of real-time transaction processing is proposed in [Kort90]. In this model, the timing constraints are associated with database consistency constraints, and not with the transactions themselves. The goal of the system is to ensure that no consistency constraint remains invalid for an interval longer than its deadline. Whenever a constraint becomes invalid, the system restores consistency by executing one or more internal transactions. Executing an internal transaction may itself invalidate other constraints. It is shown that finding an optimal (time-wise) sequence of internal transactions for restoring complete

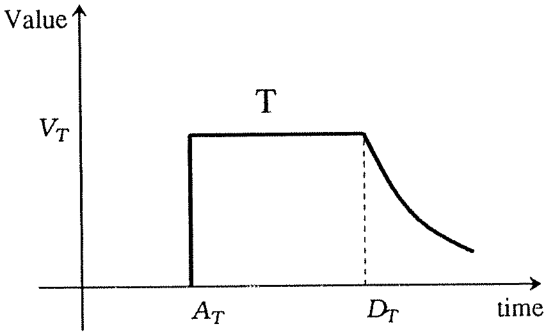

Figure 2.1(a): Soft Deadline

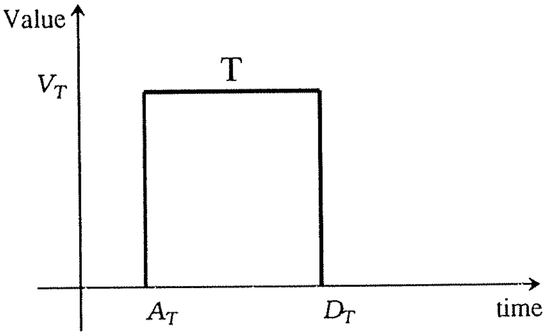

Figure 2.1(b): Firm Deadline

database consistency is computationally intractable. This result, however, does not preclude the use of heuristics to develop acceptable strategies.

2.3. Task Scheduling

A great deal of attention has been devoted by the real-time research community to the problem of dynamic (on-line) scheduling of tasks with deadlines. Liu and Layland, in their classic paper [Liu73], proved that the Earliest Deadline priority policy is optimal for *periodic* task sets with hard deadlines executing on a single processor. It is optimal in the sense that if a set of tasks can be successfully scheduled by some priority policy, then this task set is guaranteed to be successfully scheduled by the Earliest Deadline policy as well. In [Dert74], the optimality of Earliest Deadline was extended to *arbitrary* task sets executing on a single processor. For multi-processor systems, however, the policy ceases to be optimal except under restricted circumstances [Mok78]. In fact, a stronger result presented in [Mok78] is that there are *no* optimal on-line scheduling policies for arbitrary task sets executing on multiprocessor systems.

If we consider real-time systems supporting same-valued tasks with firm deadlines, scheduling optimality is defined in terms of maximizing the number of deadlines met. For this framework, Panwar and Towsley proved in [Panw88] that the Earliest Deadline policy is optimal for G/M/c/G queueing systems. (The notation G/M/c/G specifies that the task arrival process can have an arbitrary distribution, the task service times are exponentially distributed, the number of servers is arbitrary, and task deadlines can have an arbitrary distribution.) The underlying assumptions under which the optimality result holds is that task execution times are independent of their relative deadlines and inter-arrival times.[2] In practice, however, these distribution and independence constraints may not be satisfied by real-time tasks. For example, it is reasonable to expect that the execution time of a task is bounded by

[2] The relative deadline of a task is the difference of its deadline and arrival time.

its relative deadline since, otherwise, meeting its deadline is an inherently impossible task.

When tasks may have different values, performance is measured in terms of the net value realized by the system. For this metric, a high-performance scheduling algorithm called the Best Effort scheduler [Jens85, Lock86], was developed as part of the CMU Archons project. This algorithm provides excellent performance over a wide range of operating conditions and workloads [Lock86]. A constraint on the use of this scheduler, however, is that it needs a-priori knowledge of task processing requirements. It was proved recently in [Baru91] that, for the special case where task values are equal to their execution times and task arrivals are sporadic, no on-line uniprocessor scheduling algorithm can *guarantee* more than 25 percent of the value obtained by a clairvoyant scheduler.

2.4. Transaction Scheduling

Scheduling transactions with deadlines is a more complex problem than real-time task scheduling due to the multiplicity of resources in a database system, the need to maintain database integrity, and the lack of a-priori knowledge of transaction processing requirements in many database applications. The development and evaluation of transaction scheduling algorithms has been the main focus of research on real-time database systems. All of the studies in this area have considered RTDBSs that operate under either firm deadline or soft deadline applications. These studies can be divided into two general groups – those that consider applications with same-value transactions, and those that consider applications with multiple-value transactions.

2.4.1. Same Value Transactions

When all transactions have the same value, the performance objective of the RTDBS is to maximize the number of transactions that complete by their deadlines. For this framework, the problem of scheduling transactions was first addressed in a series of papers by Abbott and Garcia-Molina [Abbo88a, Abbo88b, Abbo89,. Abbo90]. In [Abbo88a, Abbo88b], several real-time scheduling algorithms for a single-processor, memory-resident database system were presented. These algorithms had different combinations of eligibility policies, priority assignments, CPU scheduling disciplines, and locking-based concurrency control protocols. A simulation study of the performance of these scheduling algorithms was presented in [Abbo88b]. Based on the simulation results, recommendations were made about the algorithm of choice under various operating conditions. For example, a scheduling algorithm that used an Earliest Deadline priority policy, a preemptive resume CPU scheduling discipline, and a prioritized variant of the Wound-Wait locking protocol [Rose78] was observed to deliver the best performance for firm-deadline applications for all the workloads considered in the study. In [Abbo89], the above study was extended by considering disk-resident databases. Since the database was disk-resident, disk service disciplines were addressed in the study. Finally, a detailed study that focused solely on real-time scheduling of disk requests was presented in [Abbo90].

Similar studies of transaction scheduling algorithms were conducted as part of the SPRING project [Rama89]. A feature of these studies [Huan89, Huan90a, Huan90b, Huan90c] is that they were conducted on a physical real-time database testbed, RT-CARAT. The results obtained therefore reflect the effect of implementation overheads, which are usually unaccounted for in simulation studies. The use of the testbed, however, has its own limitations: First, only a closed system (no external arrivals) with a fixed amount of resources is considered. Second, the disk scheduling is under the control of the operating system and therefore cannot incorporate transaction priorities. Finally, the testbed can only support small transaction populations. Some of the studies on this testbed have assumed that all transactions have the same value, while the others have considered transactions with different values. The same-value studies [Huan90a, Huan90c] are discussed in the remainder of this section, while the discussion of the multiple-value studies [Huan89, Huan90b] is deferred to the following section.

A study of the effect of buffer management policies on the ability to meet transaction deadlines was reported in [Huan90a]. The buffer organization consisted of two types of buffers: a pool of private per-transaction buffers and a shared global buffer pool. The global buffer interfaced between the pool of private buffers and the database disks. The private buffers were assumed to be large enough that buffer management was not required for them. Buffer allocation and replacement policies for the global buffer pool, however, were examined in the study. The experimental results indicated that specialized real-time buffer management schemes have little performance impact. Instead, the CPU scheduling discipline and concurrency control protocol were observed to be the primary performance determinants. A point to note about this study, however, is that the buffer pool organization is different from that seen in conventional DBMSs. In a conventional database system, all transactions share a common buffer pool in order to avoid the poor memory utilization that can result from giving individual transactions their own partitions. It is not clear to what extent the real-time buffer management results in [Huan90a] were impacted by the choice of buffer organization.

A phenomenon peculiar to priority-driven systems is *priority inversion* [Sha87]. This refers to the situation where a high-priority task is blocked by a low-priority task, and typically occurs from synchronized access to shared resources. This situation can arise in an RTDBS, for example, if a conventional locking protocol is used to control data access – a high priority transaction can be "locked out" by a low priority transaction that has accessed the object earlier. One solution to this problem is *priority inheritance* [Sha87], where the low priority transaction inherits the priority of the high priority waiter. Another solution is *priority abort* [Abbo88b] where the low priority transaction is aborted and the lock is granted to the high priority transaction. A study of the performance of these methods for firm-deadline applications was reported in [Huan90c], and it was shown that the priority abort scheme performs better than the priority inheritance scheme over a wide range of system workloads. A new scheme called *conditional priority inheritance* that has features of both the priority abort and priority inheritance schemes was also presented. It was shown to perform comparably to priority abort under low data contention, and better than priority abort under high data contention. However, the conditional priority inheritance scheme requires a-priori knowledge of transaction lengths.

2.4.2. Multiple Value Transactions

When transactions have different values, the performance objective of the RTDBS is to maximize the value realized by the system, and minimizing the number of missed deadlines becomes a secondary concern. In this situation, the scheduling problem is exacerbated since the transaction priority assignment has to take both value and deadline into account, and it is not obvious how these orthogonal characteristics should be combined. As yet, only the RT-CARAT group (apart from us) has included transactions with different values in their performance evaluation framework. In [Huan89], transactions were uniformly assigned values from a limited set of values. Using a locking protocol as the underlying serialization mechanism, the study investigated the performance of several scheduling algorithms. Each of these algorithms established a different fixed tradeoff between transaction value and deadline in assigning transaction priorities for CPU scheduling. A different set of fixed tradeoffs was used in assigning priorities for data conflict resolution. In [Huan90b], this work was extended to include optimistic methods of concurrency control. While these studies form an important step in understanding the effect of multiple transaction values on RTDBS performance, they have the following limitations: First, the range of values that transactions could take on was limited and the value distribution was uniform. Second, the concurrency control algorithms that were compared in [Huan90b] are priority-indifferent flavors of two-phase locking and optimistic concurrency control. Finally, the use of different priority mappings for CPU scheduling and data scheduling makes it difficult to identify the source of observed performance behaviors.

2.5. Database Consistency

A significant fraction of the research on real-time database systems has dealt exclusively with the concurrency control component of transaction scheduling algorithms. In [Lin90], a real-time concurrency control algorithm that has the flavor of both locking and optimistic methods was presented. Another algorithm that combines timestamp and optimistic techniques was described in [Cook91]. The objective of both these algorithms is to use the delayed conflict resolution of the optimistic technique to adjust the transaction serialization order dynamically in favor of higher priority transactions, with the locking or timestamp method being used to reduce restarts for lower priority transactions. While the ideas underlying these algorithms may have potential, their performance impacts have not yet been evaluated.

A few studies have considered increasing transaction concurrency by using one of the following methods:

(1) Multiple data versions [Son90a, Song90]: The advantage of having multiple versions of data is that it allows read-only transactions to read older versions of data while update transactions create newer versions concurrently. This increases the chances of transactions meeting their deadlines since transaction blocking times and transaction restarts are reduced. There are implementation issues, such as storage overhead and version management, however, that have to be addressed before versions can be used effectively.

(2) Weaker forms of database consistency than serializability [Lin89, Vrbs88, Sha88]: It has been argued that, in an RTDBS, satisfying transaction timing constraints is more important than maintaining data consistency. Based on this notion, algorithms that meet transaction timing constraints by sacrificing database consistency to a limited extent have been proposed. While weaker consistency levels may be acceptable for certain applications, however, serializability is currently the only general-purpose transaction consistency model.

(3) Transaction semantic information [Sing88, Ulu91a]: Semantic information, in terms of a-priori knowledge of transaction data access patterns, can be used to control data access and reduce transaction restarts, or to design non-serializable but consistent schedules. In [Ulu91a], for example, a locking protocol based on prioritizing data items

is presented. Each data item carries a priority equal to the highest priority of all transactions currently in the system that include the data item in their access lists. A transaction accessing a currently unlocked higher priority object is forced to wait until the object's priority is equal to its own. This scheme ensures preferential treatment for high priority transactions and reduces the aborts of lower priority transactions. Advance information about transaction semantics is usually, however, not available for most database applications.

2.6. RTDBS Projects

Several research centers have commissioned projects to develop prototypes of real-time database systems. These include the KARDAMOM project [Bult88] at the University of Karlsruhe, which is constructing a real-time dataflow database machine; the SPRING project [Rama89] at the University of Massachusetts, which is attempting to develop predictable real-time database systems; the HiPAC project [Daya88] at the Computer Corporation of America and the University of Wisconsin, which focused on combining active databases and timing constraints; the StarLite project [Cook90] at the University of Virginia, which is developing an RTDBS software prototyping environment; and the CASE-DB project [Oszo90] at the Case Western Reserve University, which is building a single-user relational RTDBS.

2.7. Our Research

As mentioned earlier, research efforts on real-time database systems have gained momentum only in the last few years. In fact, the first significant RTDBS scheduling study, [Abbo88b], was reported as recently as 1988. Soon afterwards, research groups at several universities started working in this area. Our research was initiated in the fall of 1989 and began as a complementary effort to the studies of [Abbo88b, Abbo89]. Those studies had focused on the priority assignment component and assumed a locking algorithm to be the underlying concurrency control mechanism. Our work focused on the concurrency control component and studied the relative performance of different classes of concurrency control algorithms. Subsequently, we moved on to addressing the problem of assigning priorities when transactions are distinguished by both value and deadline. Our work in this area followed up on the studies of [Huan89, Huan90b] by considering a wider variety of transaction workloads and by developing new priority assignments that allow the tradeoff between value and deadline to be dynamically varied.

CHAPTER 3

MODEL AND METHODOLOGY

3.1. Introduction

We developed a performance model of a centralized real-time database system and its interaction with applications to serve as the foundation for our study of real-time transaction scheduling. A simulator of this model was implemented using DeNet [Livn88], a discrete-event simulation language based on Modula-2 [Wirt82]. This simulator is the main tool employed in our performance studies of RTDBS transaction scheduling algorithms, which are detailed in subsequent chapters.

In this chapter, we describe the RTDBS performance model and discuss the experimental methodology and performance metrics used in our studies.

3.2. Modeling a Real-Time Database System

The organization of our RTDBS model is loosely based on the single-site database model of [Care88] and is shown in Figure 3.1. There are six components in this model, four of which represent the database system itself, while the remaining two capture the application utilizing the database services. The components, each of which is realized by a separate module in the simulator, are the following:

- **Database**, which models the data and its layout;
- **Source**, which generates the transaction workload;
- **Transaction Manager**, which models the execution of transactions;
- **Resource Manager**, which models the hardware resources;

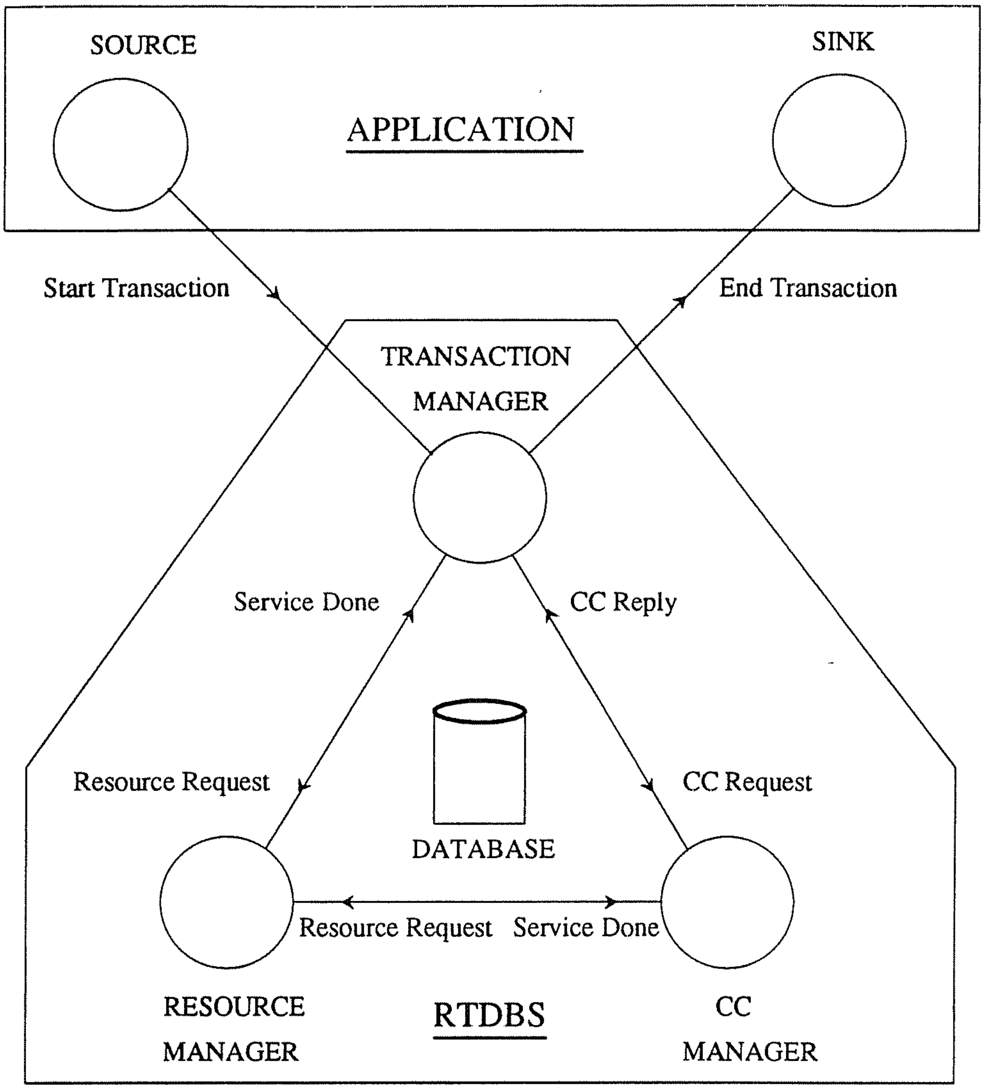

Figure 3.1: RTDBS Model

- **Concurrency Control (CC) Manager**, which controls access to shared data;
- **Sink**, which receives exiting transactions.

The interactions between these modules primarily consist of service requests and completion replies, as shown in Figure 3.1. In the remainder of this section, we describe the parameters and internals of each of the model components.

3.2.1. Database

The database is modeled simply as a collection of pages, and all database operations are modeled at the page level of granularity. For example, CPU and disk costs for processing the data are modeled on a per page basis. A parameter called *DatabaseSize* specifies the number of pages in the database. The data itself is modeled as being uniformly distributed across all of the disks.

3.2.2. Source

The Source component represents the real-time application utilizing the database services. It generates a stream of transactions that comprise the workload of the RTDBS. Table 3.1 summarizes the key workload parameters. An open system is modeled, and transactions are generated in a Poisson stream at a mean rate specified by the *ArrivalRate* parameter. Each transaction consists of a sequence of pages to be read, a subset of which are also updated. In addition, each transaction has an associated value and deadline. The mechanisms for creating a transaction and assigning its value and deadline attributes are described in the following subsections.

	Parameter	**Meaning**
	ArrivalRate	Transaction arrival rate
Page Assignment	*MeanTransSize*	Mean transaction size in pages
	SizeSprd	Spread in transaction size
	WriteProb	Write probability per accessed page
Deadline Assignment	*DeadlineFormula*	DF1, DF2, or DF3
	LSF	Low Slack Factor
	HSF	High Slack Factor
Value Assignment	*GlobalMeanValue*	Mean transaction value
	NumClasses	Number of transaction classes
	ProbClass[*i*]	Prob. of class *i*
	OfferedValue[*i*]	Fractional value offered by class *i*
	MeanValue[*i*]	Mean value of class *i* (computed)
	ValueSprd[*i*]	Spread in value of class *i*

Table 3.1: Workload Parameters

3.2.2.1. Transaction Page Assignment

The range of transaction sizes, measured in terms of the number of pages that they access, is determined by the *MeanTransSize* and *SizeSprd* parameters; transaction sizes range between (1–*SizeSprd*)***MeanTransSize* and (1+*SizeSprd*)***MeanTransSize* pages. The number of pages accessed by a transaction varies uniformly between these limits. Page requests are generated by randomly sampling (without replacement) from the entire database, that is, over the range (1, *DatabaseSize*). A page that is read is updated with probability *WriteProb*. Therefore, a page write operation is always preceded by a read for the same page; this means that the write set of a transaction is a subset of its read set and that there are no "blind writes" [Bern87]. A transaction that is restarted follows the page access sequence of the original transaction.

3.2.2.2. Transaction Deadline Assignment

In each experiment, a single formula is used to assign deadlines to transactions, and the choice of formula is determined by the *DeadlineFormula* parameter. Three different transaction deadline assignment formulas, DF1 through DF3, are employed in our studies. In describing these formulas, we will use D_T, A_T and R_T to denote the deadline, arrival time and resource time of transaction *T*, respectively. The resource time of a transaction is the total service time at the resources that it requires to commit, that is, it is the response time of the transaction if it were to execute alone in the system (refer to Section 3.6 for details). Using the above notation, we define the term **slack ratio** to represent the ratio $\frac{D_T - A_T}{R_T}$. The physical interpretation of this ratio is that it is the number of completion opportunities provided to the transaction by the application. (A slack ratio of less than 1 implies that it is impossible for the transaction to complete before its deadline.) The deadline formulas employed in our studies primarily differ in the slack ratios that they assign to transactions. All of the formulas are based on the following general expression:

$$D_T = A_T + SF_T * R_X$$

where SF_T is a multiplicative *slack factor*, and R_X is the resource time of some transaction X.

The DF1 deadline assignment is:

$$D_T = A_T + SF * R_T \tag{DF1}$$

Here, each transaction has the same slack factor SF and the resource time is that of the transaction itself. This makes all transactions, independent of their service requirement, have the *same* slack ratio, namely SF. Note that with this formula, the deadline of a transaction is *linearly* correlated with its execution time.

The DF2 deadline assignment is:

$$D_T = \begin{cases} A_T + LSF * R_T & p{=}0.5 \\ A_T + HSF * R_T & p{=}0.5 \end{cases} \tag{DF2}$$

Here, half of the transactions have a slack factor of LSF (Low Slack Factor) and the other half have a slack factor of HSF (High Slack Factor). Correspondingly, transaction slack ratios are either LSF or HSF.

The DF3 deadline assignment is:

$$D_T = A_T + Uniform(LSF, HSF) * R_{\max} \tag{DF3}$$

The resource time used in this assignment, $R_{\max}$, is the resource time of the largest transaction that could be generated by the Source (refer to Section 3.6 for details). Transaction slack factors are uniformly chosen over the range set by LSF and HSF. With this assignment, transaction slack ratios are spread over a range of values, based on the ratio of $R_{\max}$ to the R_T's and the spread in slack factors. Note that with this assignment, short transactions (i.e. those with smaller resource times) tend to have greater slack ratios than long transactions. This formula also makes the deadline of a transaction *independent* of its execution time.

The *LSF* and *HSF* workload parameters set the slack factors to be used in each of the deadline formulas. (For DF1, both the parameters are set to the same number and SF takes on this value.) An important point to note here is that while the workload generator utilizes information about transaction resource requirements in assigning deadlines, the real-time database system itself has no access to such information.

3.2.2.3. Transaction Value Assignment

The transaction workload is composed of multiple transaction value classes that are distinguished by their value distribution, and the sum of the values of the transactions submitted to the RTDBS constitutes the *total offered value*. The average value of a transaction over *all* classes is set by the *GlobalMeanValue* parameter, while the number of transaction classes is specified by the *NumClasses* parameter. Each transaction class is characterized by four parameters: *ProbClass*, *OfferedValue*, *MeanValue*, and *ValueSprd*. For a given class, *ProbClass* is the fraction of input transactions belonging to the class, while *OfferedValue* is the fractional contribution of the class to the total offered value. For example, a setting of *ProbClass* = 0.2 and *OfferedValue* = 0.8 captures a "20-80" class that constitutes 20 percent of the input transactions and accounts for 80 percent of the total offered value. The *OfferedValue* and *ProbClass* parameters, in conjunction with the *GlobalMeanValue* parameter, determine the average value of transactions in each class. This average value, *MeanValue*, is computed with the expression $\frac{OfferedValue}{ProbClass} * GlobalMeanValue$. For a *GlobalMeanValue* of 100.0, the average value of transactions in the 20-80 class described above is $\frac{0.8}{0.2} * 100.0 = 400.0$. The range of values that transactions of a class can have is bounded by the *ValueSprd* parameter, which is specified as a fraction of the *MeanValue* of the class. For the 20-80 class, a setting of 0.5 for this parameter limits its transaction values to be between 200.0 and 600.0, that is, between half and one-and-a-half times the *MeanValue* of 400.0.

A transaction's class is chosen according to the probability distribution established by the *ProbClass* parameter. The value assigned to the transaction is chosen uniformly over the value range of its class, and is independent of the transaction's other characteristics. Note that transaction values are taken from the real number domain. This means that the probability of more than one concurrently executing transaction having the same value is small, except, of course, when the *ValueSprd* parameter is set to 0.0.

3.2.3. Transaction Manager

Transactions generated by the Source are delivered to the Transaction Manager of the RTDBS. The Transaction Manager controls the execution of transactions. It implements the Service Eligibility and Priority Assignment components of the RTDBS transaction scheduling algorithm. Upon arrival, transactions receive a priority from the *priority mapper* unit (see Figure 1.4), which is embedded in the transaction manager. The priority mapper assigns priorities according to the Priority Assignment component. The transaction manager executes each transaction until it either completes or is deemed ineligible by the Service Eligibility test. Transactions that complete are marked as such and forwarded to the Sink module. A transaction that becomes ineligible is immediately "killed"; killing a transaction consists of aborting its execution, marking it as killed and forwarding it to the Sink module.

As described earlier, each transaction execution consists of a sequence of read and write page accesses. For a read page access, the Transaction Manager requests access permission from the Concurrency Control Manager. When permission is received, the Transaction Manager requests the Resource Manager to read the corresponding disk page into memory. After the page has been read in, the Transaction Manager requests CPU time to process the page from the Resource Manager. When page processing is complete, the Transaction Manager then begins executing the next page access. A write page access is executed in similar fashion to a read page access. A difference, however, is that disk write activity is deferred until the transaction has committed.[1] When all the page accesses of a transaction have been completed, the Transaction Manager initiates commit processing for the transaction.

[1] We assume sufficient buffer space to allow the retention of updates until commit time. In addition, we assume the use of a log-based recovery scheme where only log pages are forced to disk prior to commit.

Transactions may sometimes have to be aborted due to data conflicts. In this case, the Concurrency Control Manager decides that the transaction should be aborted and informs the Transaction Manager. The Transaction Manager then invokes the abort procedure for the transaction. After the abort procedure is completed, the transaction is restarted and follows the same data access pattern as the original transaction.

Transactions are also aborted when they are killed due to failing the Service Eligibility test. In this case, the Transaction Manager is the initiator of the abort process, and the aborted transaction is not restarted but sent, instead, to the Sink.

3.2.4. Resource Manager

The Resource Manager represents the RTDBS operating system and controls access to the physical resources of the database system. In our model, the RTDBS physical resources consist of multiple CPUs and multiple disks. There is a single common queue for the CPUs and the service discipline is Pre-emptive Resume, with preemption being based on transaction priorities. Each of the disks has its own queue and is scheduled according to a Head-Of-Line (HOL) policy, with the request queue being ordered by transaction priority.[2] Table 3.2 summarizes the key parameters of the resource model. The *NumCPUs* and *NumDisks* parameters specify the hardware resource composition, while the *PageCPU* and *PageDisk* parameters cap-

Parameter	**Meaning**
NumCPUs	Number of processors
NumDisks	Number of disks
PageCPU	CPU service time per data page
PageDisk	Disk service time per data page

Table 3.2: Resource Parameters

[2] Prioritized scan-based disk scheduling disciplines have been observed to perform better than a priority HOL policy [Abbo90]; for the sake of simplicity, we only consider a priority HOL policy here.

ture CPU and disk processing times per data page. We assume that preemption costs at the CPU are negligible compared to page processing times.

Memory buffer resources are an integral feature of database systems. For the sake of simplicity, however, we assume that all data is accessed from disk and buffer pool considerations are therefore ignored. While modeling buffering would certainly result in different absolute performance numbers, we do not expect that doing so would significantly alter the general conclusions of our studies.

3.2.5. Concurrency Control Manager

The Concurrency Control Manager maintains database consistency by regulating transaction access to data pages. It implements a concurrency control protocol, servicing concurrency control requests received from the Transaction Manager. These requests include requests for read access, write access, and to commit or abort a transaction. For processing each of these requests, the Concurrency Control Manager requests a period of CPU service from the Resource Manager. A parameter called *CCReqCPU* specifies the CPU overhead required to process each request. In our simulator, a separate instance of the Concurrency Control Manager was created for each of the concurrency control algorithms evaluated in our experiments.

3.2.6. Sink

The Sink module receives both completed and killed transactions from the Transaction Manager. It gathers statistics on these transactions and measures the performance of the system from the application's perspective.

3.3. Experimental Methodology

Our performance study of RTDBS transaction scheduling algorithms consists of five sets of experiments; each set is discussed in a separate chapter. Chapters 4 and 5 focus on the Concurrency Control component, while Chapters 6, 7 and 8 are primarily concerned with the Priority Assignment component. The Service Eligibility and Resource Service components of the transaction scheduling algorithm are the same across all the studies: Feasible Only eligibility test, priority Pre-emptive Resume at the CPUs, and priority HOL at the disks.

In all of our experiments, the deadline assignments are such that each transaction is guaranteed to make its deadline if it were to execute alone in the system. This means that the only reasons for transactions to miss their deadlines are resource contention and data contention. The level of resource contention in the system is determined by the quantity of resources, the transaction processing costs, and the mean number of transactions. Similarly, the level of data contention is determined by the database size, the transaction size distribution, the frequency of updates, and the mean number of transactions. The processing costs, the database size and the transaction size distribution are kept constant across almost all of our experiments, while the resource composition, data update probability, and number of transactions are varied.

Our experiments evaluate the impacts of resource contention and data contention, both in isolation and in combination. For experiments intended to isolate the effect of resource contention, the page write probability is set to 0.0. For experiments intended to isolate the effect of data contention, an "infinite" resource situation [Fran85, Agra87, Tay84], is simulated. With infinite resources, there is no queueing at the resources. A point to note here is that while abundant resources are usually not to be expected in conventional database systems, they may be more common in RTDBS environments since real-time systems are usually sized to handle transient heavy loading. This directly relates to the application domain of RTDBSs, where functionality, rather than cost, is often the driving consideration.

We began each of our studies by developing a baseline experiment. The simulation parameters used for this experiment force the real-time database system to operate in a region that brings out the performance differences between the transaction scheduling algorithms under consideration. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. These experiments evaluated the performance impact of changes in workload characteristics and system configuration. The simulator was instrumented to provide a host of statistical information, including CPU and disk utilizations, mean system population, number of transaction restarts, etc. These secondary measures help to explain the behavior of the algorithms under various loading conditions.

3.4. Performance Metrics

The sum of the values of all transactions submitted to the RTDBS constitutes the *offered value*, while the sum of the values of the transactions that are completed before their deadlines constitutes the *realized value*. With these definitions, the primary performance metric, *Loss Percent*, is computed as

$$\textit{Loss Percent} = \left[\frac{\textit{Offered Value} - \textit{Realized Value}}{\textit{Offered Value}}\right] * 100$$

Thus, Loss Percent is the percentage of the offered value that is *not* realized by the system.

A secondary performance metric is *Miss Percent*, which is computed as

$$\textit{Miss Percent} = \left[\frac{\textit{Input Transactions} - \textit{InTime Transactions}}{\textit{Input Transactions}}\right] * 100$$

Thus, Miss Percent is the percentage of the input transactions that the system is *unable* to complete before their deadline. Note that when all transactions have the same value, the Loss Percent and Miss Percent metrics are equivalent.

For each of these metrics, numbers in the range of 0 to 20 percent are taken to represent system performance under "normal" loads, while numbers in the range of 20 to 100 percent are viewed as representing system performance under "heavy" loads. A long-term operating region where the loss or miss percent is large is obviously unrealistic for a viable RTDBS. Exercising the system to high loss or miss levels, however, provides valuable information on the response of the various scheduling algorithms to brief periods of stress loading. All of our experiments evaluate the algorithms over a wide range of loads.

The Loss Percent and Miss Percent graphs in this thesis show mean values with relative half-widths about the mean of less than 5% at the 90% confidence interval. Each simulation experiment was run until at least 10000 transactions had been processed by the real-time database system simulator. Only statistically significant differences are discussed in this thesis.

Transactions are all assigned the same value in the studies of Chapters 4, 5 and 6, and the Loss Percent and Miss Percent performance metrics are therefore equivalent. We discuss the Miss Percent metric in these chapters. Different transactions may have different values in the studies of Chapters 7 and 8, so both the Loss Percent and Miss Percent metrics are considered there.

3.5. Fixed Parameter Settings

As mentioned earlier, some of the workload and system parameters have the same setting across virtually all of the experiments reported here. These "constant" parameters and their settings are listed in Table 3.3. With regard to the workload parameters, the average transaction size is maintained at 16, and the size spread parameter is set to 0.5. This means that transaction sizes are uniformly distributed over the range (8, 24) pages. While the distribution of transaction values may vary, the values are assigned such that the *average* value of a transaction is always 100.0. With regard to the system parameters, the database size is fixed at 1000 pages, while the CPU and disk costs for processing a page are set to 10 milliseconds and 20 milliseconds, respectively. The concurrency control CPU overhead is assumed to be negligible compared to the 10 millisecond CPU time for page processing and is therefore set to 0.0.

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SizeSprd	0.5	*PageCPU*	10ms
GlobalMeanValue	100.0	*PageDisk*	20ms
		CCReqCPU	0.0

Table 3.3: Fixed Parameter Settings

While the above settings hold for most of the experiments described in this thesis, we also evaluated the sensitivity of the performance results to changes in these "constant" parameters. A few of these sensitivity experiments are also included in the thesis.

3.6. Resource Time Computations

The resource time of a transaction, R_T, is computed with the following expression

$$R_T = NumReads_T * (PageCPU + PageDisk) + NumWrites_T * PageCPU;$$

where $NumReads_T$ and $NumWrites_T$ are the number of pages that are read and updated by the transaction, respectively. The disk time for writing updated pages is not included in the resource time computation since these writes occur *after* the transaction has committed.

Deadline formula DF3 uses R_{max}, the resource time of the largest workload transaction, in their deadline computations. Based on the transaction page assignment mechanism (refer to Section 3.2.2.1), the largest possible transaction is one that reads and updates (1+*SizeSprd*)**MeanTransSize* pages. R_{max} is therefore computed with the formula given above by setting *NumReads* and *NumWrites* to this value. Note that for the special case where *WriteProb* is 0.0, the largest possible transaction merely reads (1+*SizeSprd*)**MeanTransSize* pages. To maintain uniformity across experiments, however, the R_{max} value is always computed assuming that the largest possible transaction updates all of its pages.

CHAPTER 4

THE POTENTIAL OF OPTIMISTIC CONCURRENCY CONTROL

4.1. Introduction

A hallmark of modern computer systems is their support for **multi-programming**, whereby multiple programs can execute simultaneously in the computer system. The benefit of multi-programming is that it increases the utilization of system resources, and can therefore result in increased task throughput. In a multi-programmed database system, it is necessary to control concurrent transaction execution to maintain database consistency. This control is achieved by requiring transactions to follow a concurrency control protocol while accessing data.

Several different concurrency control mechanisms have been proposed in the database literature [Bern87]; two well-known classes are **locking protocols** (e.g. [Gray79]), and **optimistic techniques** (e.g. [Kung81]). Performance studies of these mechanisms in conventional database systems (e.g. [Agra87]) have concluded that, under limited resource conditions, locking protocols outperform optimistic techniques. These results may not apply to real-time database systems, however, due to the significant differences between conventional and real-time database systems.

In this chapter, we profile the performance behavior of locking protocols and optimistic techniques in the context of a firm-deadline real-time database system. All transactions have the same value in the experiments of this chapter, so the performance metric of interest is the number of missed deadlines.

4.2. Concurrency Control Algorithms

Several concurrency control algorithms have been developed based on the locking and optimistic concurrency control mechanisms [Bern81]. We compare a specific locking protocol, **2PL-HP**, with a specific optimistic technique, **OPT-BC**, in this chapter. These particular instances were chosen because they appear well-suited to an RTDBS environment, are of comparable complexity and are general in their applicability. The details of these algorithms are explained below.

4.2.1. 2PL-HP

In classical two-phase locking (2PL) [Eswa76, Gray79], transactions set read locks on objects that they read, and these locks are later upgraded to write locks for the objects that are updated. If a lock request is denied, the requesting transaction is blocked until the lock is released. Read locks can be shared, while write locks are exclusive.

In 2PL-HP, the basic 2PL algorithm is augmented with a *High Priority* [Abbo88b] conflict resolution scheme to ensure that high priority transactions are not delayed by low priority transactions. This scheme resolves all data conflicts in favor of the transaction with the higher priority. When a transaction requests a lock on an object held by other transactions in a conflicting lock mode, if the requester's priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requester's priority is lower, it waits for the lock holders to release the object.[1] An additional benefit of the High Priority scheme is that it serves as a deadlock prevention mechanism.[2]

[1] In addition, a new reader can join a group of read-lockers only if its priority is higher than that of all waiting writers.

[2] This is true only for priority assignment policies that assign unique priority values and do not change a transaction's priority during the course of its execution.

4.2.2. OPT-BC

In classical optimistic concurrency control (OPT) [Kung81], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. A transaction is restarted at its commit point if it fails its validation test. This test checks that there is no conflict of the validating transaction with any of the transactions that have committed since it began execution.

In OPT-BC, the basic OPT algorithm is modified to implement the so-called *Broadcast Commit (BC)* [Mena82, Robi82] scheme.[3] Here, when a transaction commits, it notifies other currently running transactions that conflict with it and these conflicting transactions are immediately restarted. A conflict exists between the validating transaction *V* and an executing transaction *E* iff the intersection of the write set of *V* and the current read set of *E* is non-empty.[4] Note that there is no need to check for conflicts with previously committed transactions since, in the event of a conflict, any such transaction would already have restarted the validating transaction at its (the committed transaction's) own earlier commit time. This also means that a validating transaction is always guaranteed to commit. The broadcast commit method detects conflicts earlier than the basic OPT algorithm, resulting in earlier restarts and lesser wasted resources; this increases the chances of meeting transaction deadlines. A point to note is that this algorithm does not use transaction priorities in resolving data contention. We will return to this issue later in the chapter.

[3] The Broadcast Commit scheme is also referred to as "forward" optimistic concurrency control [Hard84].

[4] A scheme for implementing the broadcast commit method is discussed in Appendix A.

4.3. 2PL-HP versus OPT-BC

Locking and optimistic concurrency control represent two extremes in terms of data conflict detection and conflict resolution – locking detects conflicts as soon as they occur and resolves them using blocking; optimistic concurrency control detects conflicts only at transaction commit times and resolves them using restarts. Earlier comparative studies of locking and optimistic algorithms for a conventional DBMS (e.g. [Agra87]) have shown that, under operating circumstances of limited resources, locking provides significantly better performance than optimistic concurrency control. Some fundamental aspects of the RTDBS environment, outlined below, indicate a potential for these previous results to be altered here.

4.3.1. Blocking

The main reason for the good performance of locking in a conventional DBMS is that its blocking-based conflict resolution policy results in conservation of resources, while the optimistic algorithm with its restart-based conflict resolution policy wastes more resources. In an RTDBS environment, however, we expect to see a smaller difference between the useful resource utilizations of the two algorithms, thus reducing the advantage that locking has over optimistic algorithms. This is because OPT-BC implicitly derives a blocking effect due to *resource contention* – low priority transactions wait when resources are captured by high priority transactions. Low priority transactions that may conflict with high priority transactions are thus effectively prevented from making progress by priority-based resource scheduling, thereby decreasing the chances of data conflicts. Also, if a conflict does occur and the low priority transaction has to be restarted, the wasted resource utilization is at least reduced. Conversely, 2PL-HP loses some of basic 2PL's blocking factor due to the partially restart-based nature of the High Priority scheme.

4.3.2. Restarts

In locking algorithms, data conflicts are resolved as soon as they occur, while in optimistic algorithms, data conflicts are resolved only when a transaction attempts to commit. In a conventional DBMS, the delayed conflict resolution of optimistic algorithms causes them to waste more resources than locking algorithms. In an RTDBS, however, this delayed conflict resolution can actually aid in making *better decisions* since more information about the conflicting transactions is available when the conflict is resolved. For example, with 2PL-HP, a transaction could be restarted by a higher priority transaction that later misses its deadline and is discarded. This means that the restart did not result in the higher priority transaction meeting its deadline. In addition, it may cause the lower priority transaction to miss its own deadline. Therefore, such **wasted restarts** can result in performance degradation. With OPT-BC, however, a transaction that reaches its validation stage is *guaranteed* to commit and complete before its deadline. Since only validating transactions can cause restarts of other transactions, *no* wasted restarts are generated by the OPT-BC algorithm.

4.3.3. Priority Reversals

With some transaction priority assignment schemes (e.g. LeastSlack [Abbo88b]), it is possible for a pair of concurrently running transactions to have opposite priorities relative to each other at different points in time during their execution. We will refer to this phenomenon as *priority reversal.*[5] With 2PL-HP, data conflicts between members of such a pair could result in **mutual restarts**, that is, the pair may restart each other.

An example of the mutual restart phenomenon is shown in Figure 4.1. There we show the priority profile as a function of time for two concurrently executing transactions A and B, with deadlines D_A and D_B, respectively. In these profiles, although A initially has higher

[5] This is different from *priority inversion* [Sha87], which refers to the situation where a transaction is blocked (due to a data or resource conflict) by a lower-priority transaction.

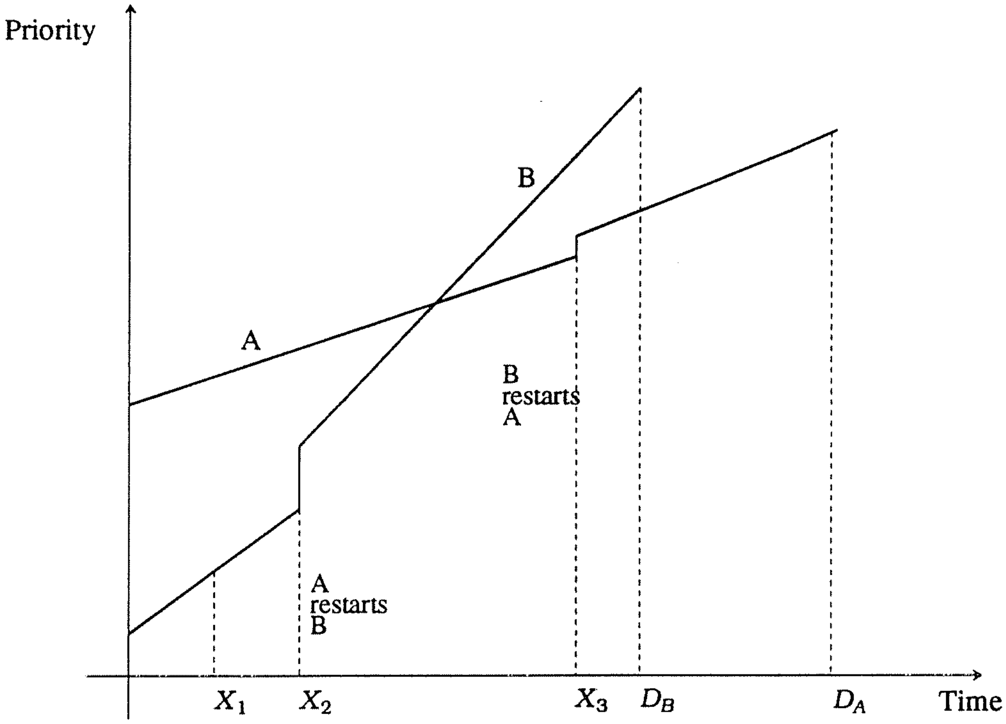

Figure 4.1: Mutual Restarts with 2PL-HP

priority than B, the situation gets reversed with the passage of time because B has a higher rate of priority increase. At time $t = X_1$, transaction B locks object X. At time $t = X_2$, transaction A attempts to access the same object in a conflicting mode. Since, at this time, the priority of A is greater than that of B, B is restarted and A is granted the lock. B begins re-executing with a new priority profile and attempts to access object X at time $t = X_3$. At this time, A is still holding the lock on object X. Since B now has a higher priority than A, it is A's turn to be restarted while B is given the lock. Finally, B misses its deadline at D_B, while A misses its deadline at D_A. The observation here is that the sequence of restarts hindered the progress of both transactions, thus resulting in degraded performance.

Based on the above example, we can envision situations where, because of dynamically shifting transaction priorities, mutual restarts take place leading to an increased number of missed deadlines. Also, depending on the dynamics of the priority profile, the database system may have to "constantly" poll all locked data objects to check that lock holders still maintain higher priority over their associated lock waiters.[6] If a priority reversal is detected, the High Priority scheme is applied between the waiters and the lock holders.

In contrast to the above behavior of 2PL-HP, such priority related problems do not arise with OPT-BC since it does not make use of priorities in resolving data conflicts.

We conducted experiments to evaluate the performance effects of the above-mentioned factors, and the following section describes the experimental setup and the results that were obtained.

4.4. Experiments and Results

In this section, we present performance results for our experiments comparing 2PL-HP and OPT-BC in a firm-deadline RTDBS environment. We began our investigation by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. These experiments evaluate the impact of data contention, resource contention, deadline tightness/slackness and priority reversals. To serve as a basis for comparison, the performance achievable in the *absence* of any concurrency control are also shown on the graphs under the title NO-CC. The NO-CC curve should be interpreted as the contribution of resource contention alone towards performance degradation. In Appendix B, a theoretical basis is established for the shapes of the curves seen in the simulation results.

In all the experiments described in this chapter, deadline formula DF1 is used to assign transaction deadlines. This means that all transactions have the same slack ratio. The transaction priority assignment in most of the experiments is *Earliest Deadline* – transactions with

[6] This is required not only to ensure that higher priority transactions are not held up by lower priority transactions, but also for deadlock prevention – the High Priority scheme does not function as a deadlock prevention mechanism if priority reversals are possible.

earlier deadlines have higher priority than transactions with later deadlines. For one experiment, however, which is designed to investigate the impact of priority reversals, the priority assignment is *discrete LTD (Least Time to Deadline)*.[7] In the basic LTD scheme [Abbo89], the priority of a transaction at time t is evaluated[8] as $(D_T - t)$. For the discrete variant, the priority of a transaction is evaluated when it arrives and reevaluated only whenever it is restarted. In between reevaluations, the priority remains at its previously computed value. Since priorities are recomputed only at restarts, it is not necessary to poll data objects to check that lock holders maintain higher priority than their associated lock waiters.

4.4.1. Experiment 1: Baseline Experiment

The settings of the workload parameters and system parameters for the baseline experiment are listed in Table 4.1. (For clarity, the workload parameters related to transaction value distribution are not shown since all transactions have the same value, namely 100.0.) These settings generate an appreciable level of both data and resource contention, thus serving to

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	10
WriteProb	0.25	*NumDisks*	20
DeadlineFormula	DF1	*PageCPU*	10ms
LSF	4.0	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		

Table 4.1: Baseline Parameter Settings

[7] Priority reversals do not occur with the Earliest Deadline priority assignment since transaction priorities remain unchanged relative to each other.

[8] The scheme described in [Abbo89] is actually *LeastSlack*, where the priority is computed as $(D_T - t - r_T)$, with r_T being the remaining service requirement of transaction T. Since we assume that the RTDBS has no knowledge of transaction service requirements, the formula has been modified accordingly.

bring out the performance differences between the algorithms. For this experiment, Figures 4.2(a) and 4.2(b) show the Miss Percent behavior under normal load and heavy load conditions, respectively. From these graphs it is clear that, at very low arrival rates, 2PL-HP and OPT-BC perform almost identically, but as the arrival rate increases, OPT-BC performs progressively better than 2PL-HP. The cause for the better performance of OPT-BC is its lower number of restarts, as shown in Figure 4.2(c).[9] With OPT-BC, only a *committing* transaction can cause the restarts of other transactions. With 2PL-HP, however, a transaction can generate restarts at any time during the course of its execution. Therefore, even a transaction which is eventually discarded may cause the restarts of other transactions during its sojourn in the system. At higher loads, when many transactions miss their deadline and have to be

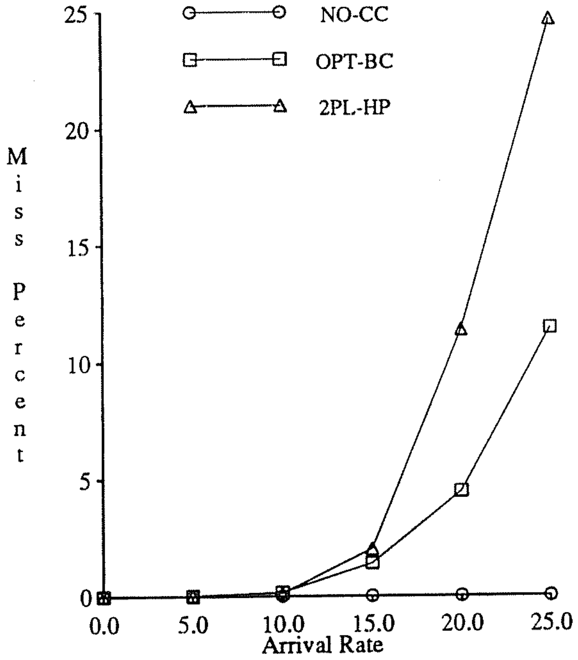

Figure 4.2(a): Baseline (Normal)

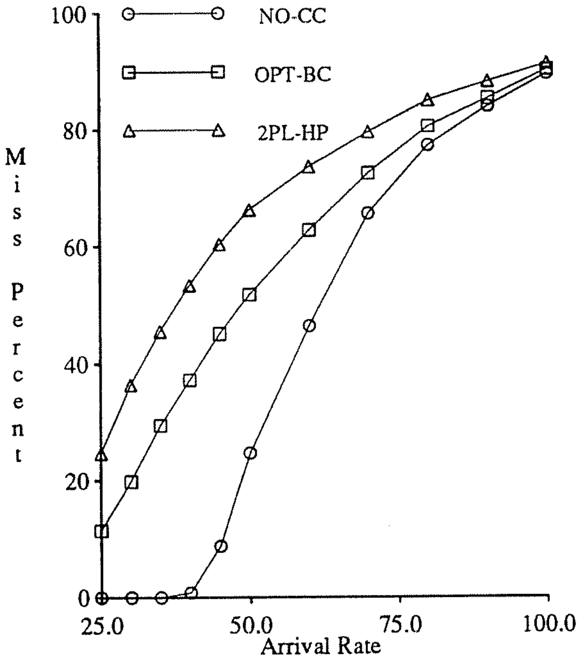

Figure 4.2(b): Baseline (Heavy)

[9] All "restart" graphs are normalized on a per-transaction basis; that is, they are computed as the number of restarts divided by the number of input transactions.

discarded, 2PL-HP has significantly more restarts than OPT-BC. This is brought out clearly in Figure 4.2(c), where a large difference is observed between the "useful restarts" curve for 2PL-HP, which shows the number of restarts caused only by eventually committed transactions, and the "total restarts" curve for 2PL-HP, which shows the total number of restarts caused by all transactions. (The restarts decrease after a certain load because resource contention, rather than data contention, becomes the more dominant reason for transactions missing their deadlines).

Figure 4.2(d) shows the average progress made by transactions before they were restarted due to data conflicts. It is clear from this figure that 2PL-HP consistently detects conflicts earlier than OPT-BC. One might therefore expect 2PL-HP to waste less resources than OPT-BC. However, since OPT-BC has far fewer restarts, it actually makes better overall use of resources than 2PL-HP. This concept is quantified in Figure 4.2(e), where the total utilization and the useful utilization of the processors are shown. Useful utilization is computed as the processor usage made by those transactions that eventually met their deadlines. The processor utilization is selected here because the processors are the bottleneck resource with the parameter settings of this experiment. From the utilization curves, it is clear that OPT-BC is more resource-efficient than 2PL-HP.

An important point to note is that the transaction workload of this experiment could be expected, in the absence of deadlines, to generate exactly the *opposite* results in a resource-limited conventional DBMS: A locking algorithm would perform better than an optimistic algorithm because (a) it has no wasted restarts since all transactions are eventually executed to completion, and (b) it is better at conserving resources.

4.4.2. Experiment 2: Pure Data Contention

The goal of our next experiment was to isolate the impact of data contention on the performance of the concurrency control algorithms. For this experiment, therefore, the resources

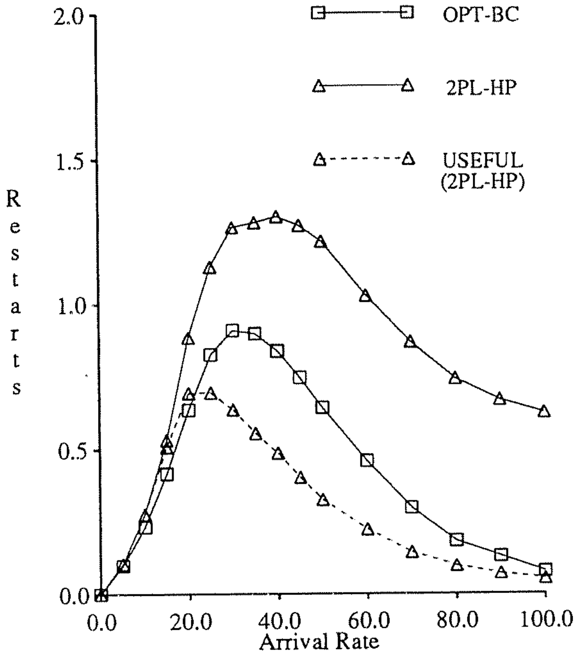

Figure 4.2(c): Restarts (Baseline)

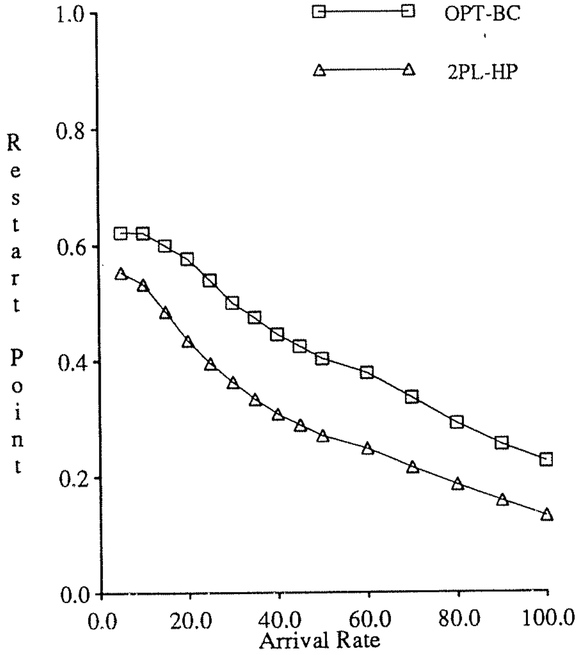

Figure 4.2(d): Restart Length (Baseline)

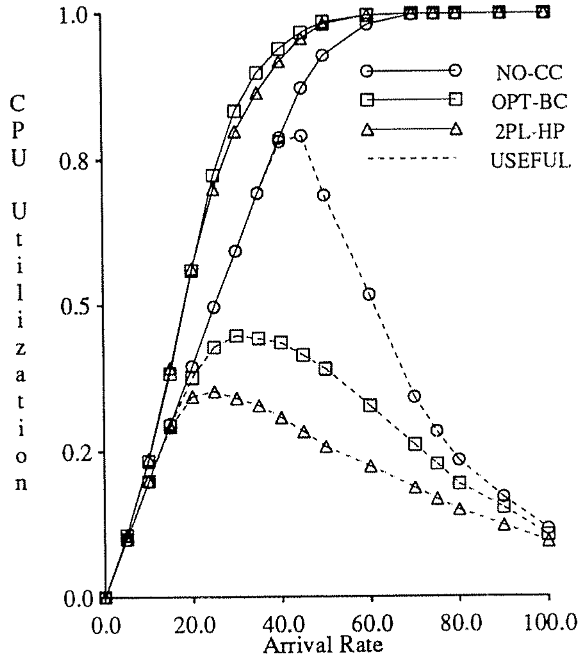

Figure 4.2(e): CPU Utilization (Baseline)

were made "infinite".[10] The performance results for this system configuration are presented in Figures 4.3(a) and 4.3(b) (the NO-CC graph is not shown since resource contention is absent in this experiment). These figures show OPT-BC to perform much better than 2PL-HP in terms of both normal load performance and heavy load performance. There are two reasons for OPT-BC outperforming 2PL-HP here: First, the basic wasted restarts problem of 2PL-HP, as outlined earlier for the baseline experiment, occurs here too. This effect is shown in Figure 4.3(c). Second, the blocking component of 2PL-HP reduces the number of transactions that are executing and making progress. The blocking causes an increase in the average number of transactions in the system, thus generating more conflicts and a greater number of restarts. This behavior of 2PL-HP is in contrast to that of OPT-BC, where transactions are always executing and are never blocked. This effect is quantified in Figure 4.3(d).

An important observation here is that while resource contention can be reduced by purchasing more resources and/or faster resources, there exists no equally simple mechanism to reduce data contention. It should also be noted that optimistic algorithms perform better than locking protocols under infinite resource conditions in a conventional DBMS setting as well [Fran85, Agra87].

4.4.3. Experiment 3: Deadline Tightness / Slackness

Our next experiment examined the effect that the tightness or slackness of deadlines had on the relative performance of the algorithms. For this experiment, the Slack Factor in the deadline formula (DF1) was varied from 1 through 10, keeping all the other parameters the same as those of the baseline experiment. The experiment was conducted for arrival rates of 10 and 30 transactions/sec and the corresponding Miss Percent graphs are shown in Figures 4.4 and 4.5, respectively. At low slack factors, both algorithms show a high Miss Percent since transactions are operating under very tight deadlines. As the slack factor increases,

[10] As mentioned in Section 3.3, infinite resources means that there is no queueing for resources.

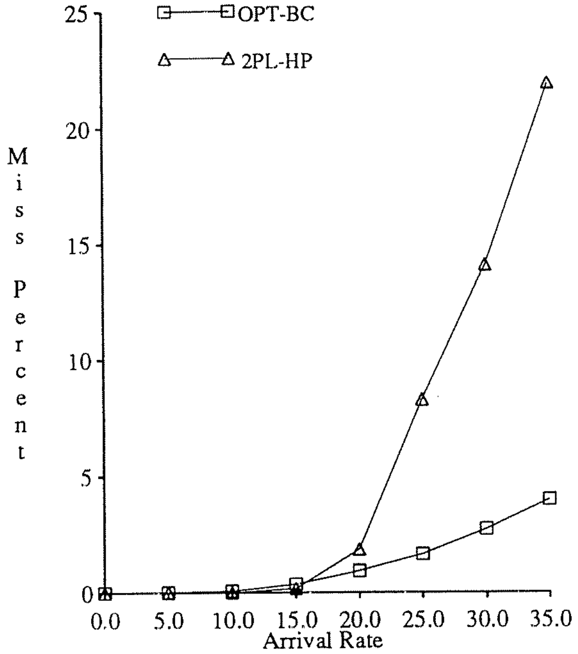

Figure 4.3(a): ∞ Resources (Normal)

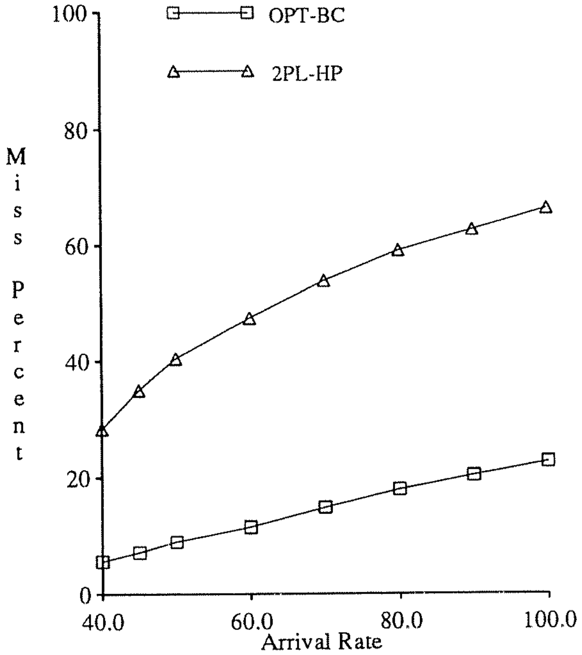

Figure 4.3(b): ∞ Resources (Heavy)

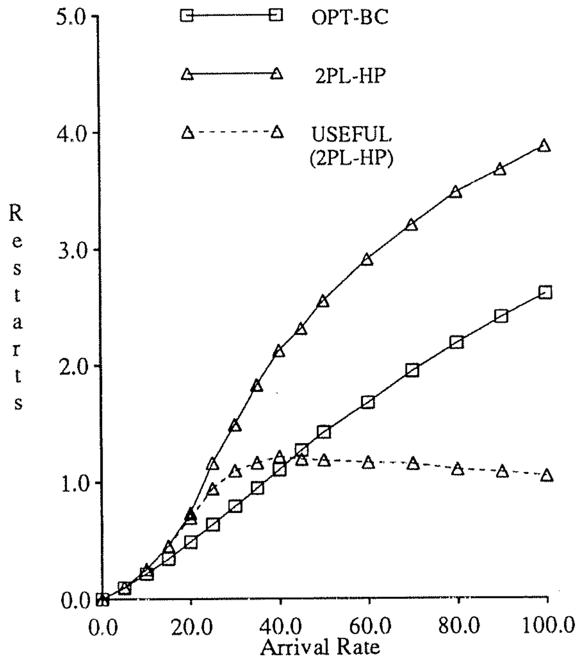

Figure 4.3(c): Restarts (∞ Resources)

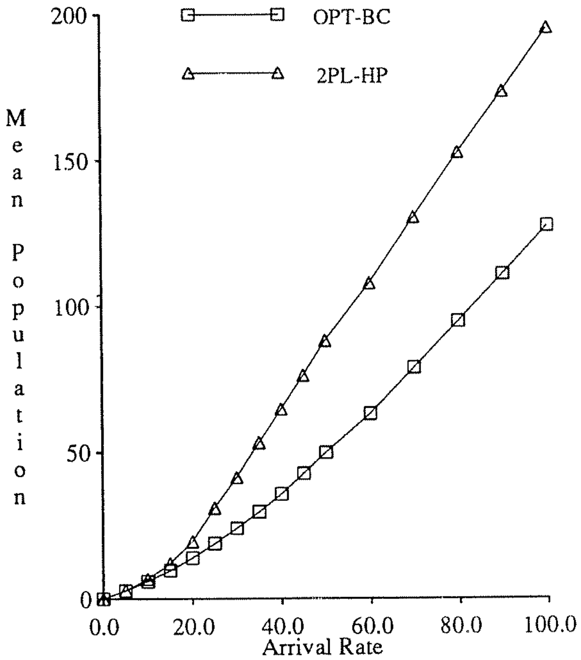

Figure 4.3(d): Population (∞ Resources)

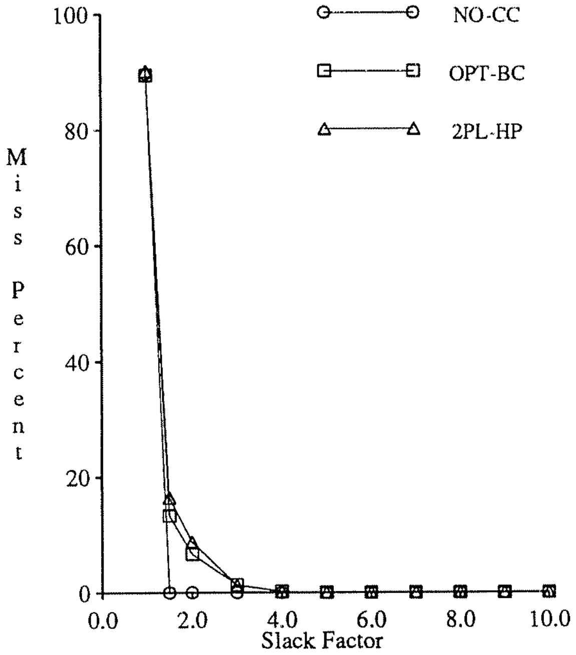

Figure 4.4: Slack (Arr. Rate = 10)

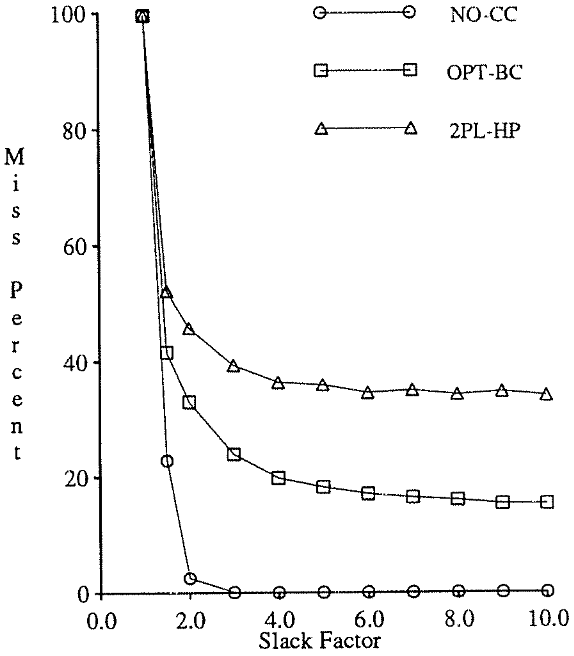

Figure 4.5: Slack (Arr. Rate = 30)

transactions are given more time to complete and the Miss Percent decreases sharply. For the 10 transactions/second arrival rate, the Miss Percent goes down all the way to zero. For the 30 transactions/second arrival rate, however, the Miss Percent stays virtually constant beyond a slack factor of 5 for both 2PL-HP and OPT-BC. This behavior is explained as follows: Since increasing the slack factor provides transactions with more time to complete, it results in more transactions concurrently executing in the system rather than being discarded as late. As the number in the system increases, the resources in the system eventually saturate and the Miss Percent then becomes constant for a fixed arrival rate.

We also observe from Figures 4.4 and 4.5 that the superior performance of OPT-BC over 2PL-HP is maintained across the entire range of slack factors.

4.4.4. Experiment 4: Priority Reversals

In order to examine the performance effect of priority reversals, we conducted an experiment where *discrete LTD* was used as the transaction priority assignment while keeping the other parameters the same as those of the baseline experiment. For this experiment, the 2PL-

HP algorithm was altered in the following manner: The priority of the requesting transaction is compared not with the current priority of the lock holders, but with the priority the lock holders would have *if they were to be restarted.* The reason for this change is to prevent *immediate* mutual restarts [Abbo89], where a pair of transactions repeatedly restart each other since the priority of each, after a restart, is greater than that of the other.

The results of this experiment are shown in Figure 4.6(a). We observe here that 2PL-HP does significantly worse than OPT-BC over almost the entire range of loadings. For the corresponding experiment using the Earliest Deadline priority assignment (Experiment 1), however, 2PL-HP performed comparably to OPT-BC at very low and very high loads, and was noticeably worse only at intermediate loads. The restart curves in Figure 4.6(b) show the reason for the performance of 2PL-HP being further degraded here: It now suffers not only from the wasted restarts problem but also from the problem of mutual restarts. The "NO-MUTUAL" curve in Figure 4.6(b) shows the total number of restarts discounting those caused due to

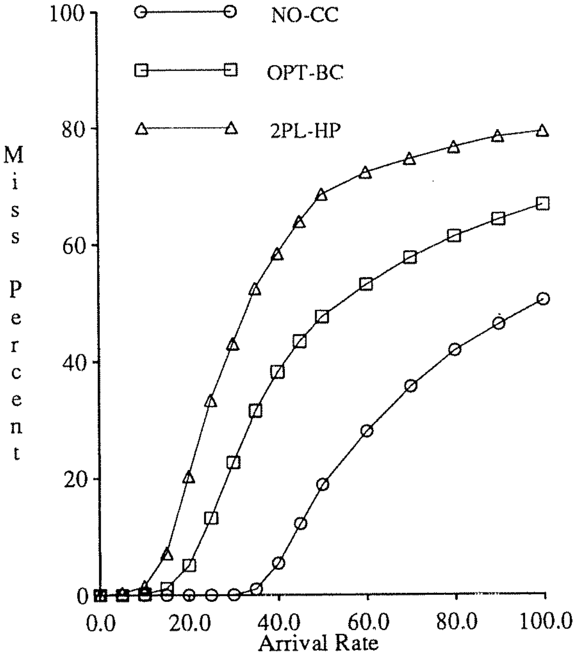

Figure 4.6(a): Priority Reversals

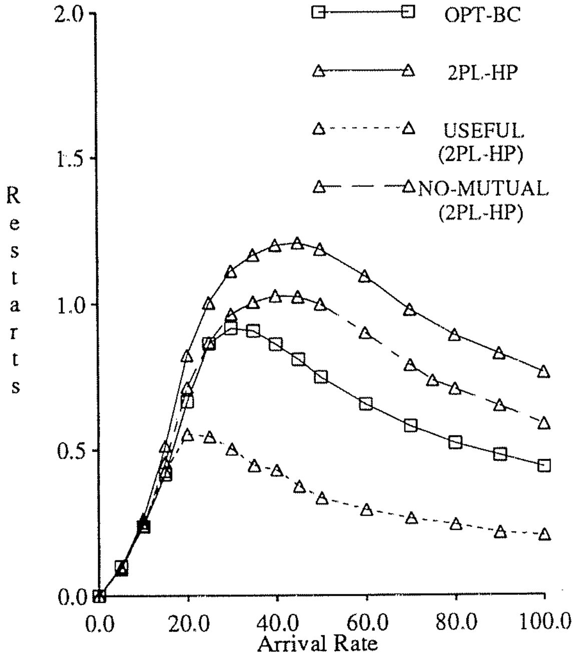

Figure 4.6(b): Restarts (Reversals)

priority reversals.[11] It is evident from this figure that mutual restarts make a noticeable contribution to the total number of restarts and therefore play a role in degrading the performance of 2PL-HP.

An important point to note here is that the discrete priority evaluation scheme used in this experiment produces limited fluctuation in transaction priorities since priorities are recomputed only at transaction restart times. For priority assignments that generate greater fluctuations in priority, such as basic LTD, the mutual restarts problem could be expected to have a greater impact on the performance of 2PL-HP.

A second point to note is that 2PL-HP was modified in this experiment to eliminate immediate mutual restarts. When the experiment was conducted with the original unmodified version of 2PL-HP, the performance of 2PL-HP was observed to be substantially worse due to immediate mutual restarts.

A final observation is that the modification of 2PL-HP to eliminate immediate mutual restarts works only for priority assignments that maintain a constant transaction priority between restarts. If transaction priorities can change in between restarts, immediate mutual restarts can occur even with the modified 2PL-HP algorithm.

4.4.5. Other Experiments

We studied the sensitivity of the results of the experiments described in this chapter to changes in the mean transaction size, the database size and the transaction page write probability (refer to [Hari90a] for details). In one experiment, the mean transaction size was varied from 4 through 32, keeping the other parameters the same as those of the baseline experiment. In another experiment, the number of pages in the database was varied from 100 through 5000, keeping the other parameters the same as those of the baseline experiment.

[11] The mutual restart counter is incremented whenever a transaction *A* is restarted by another transaction *B*, with *A* itself having restarted *B* earlier.

Both these experiments were conducted for an arrival rate of 30 transactions/sec and the corresponding Miss Percent graphs are shown in Figures 4.7 and Figure 4.8. From these figures, we observe that when data contention is low (small transactions or large database), both concurrency control algorithms perform almost the same. This is as expected since, under low data contention, performance is primarily determined by the resource scheduling algorithm rather than the concurrency control algorithm. As data contention increases, however, OPT-BC performs progressively better than 2PL-HP, since 2PL-HP suffers from a greater number of wasted restarts. A similar behavior of OPT-BC and 2PL-HP was seen in experiments where the transaction page write probability was varied.

In all of the experiments described in this chapter, deadline formula DF1 was used to assign transaction deadlines. With this formula, all transactions have the same slack ratio. As explained in the next chapter, a common transaction slack ratio minimizes the adverse impacts of OPT-BC's priority indifference. We therefore see OPT-BC outperforming 2PL-HP

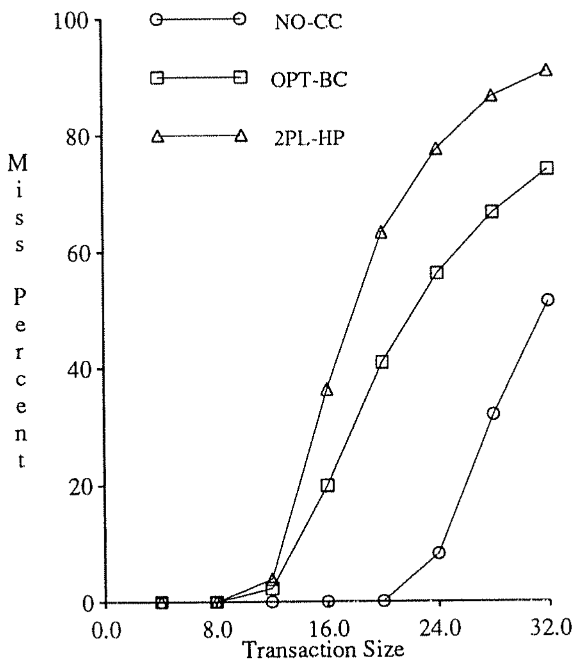

Figure 4.7: Trans. Size (Arr. Rate = 30)

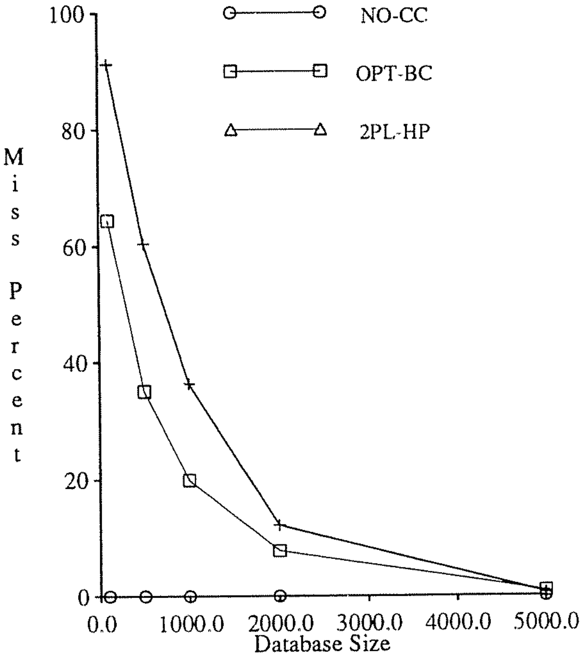

Figure 4.8: Database (Arr. Rate = 30)

over the entire range of loadings in the experiments discussed here. However, when transaction deadline formulas such as DF2 are used, which result in transactions having different slack ratios, 2PL-HP outperformed OPT-BC at very low loads. This is because OPT-BC's priority-indifference has a greater detrimental impact when transactions have different slack ratios. With increased loading, however, the performance of 2PL-HP steeply degraded due to the detrimental effects of its wasted restarts dominating the benefits of its priority-cognizance. In this loading region, therefore, OPT-BC performed better than 2PL-HP in similar fashion to that seen in the experiments discussed here.

Several other experiments that compare the performance of 2PL-HP and OPT-BC under different scenarios are detailed in [Hari90a]. In each of these experiments, OPT-BC and 2PL-HP showed similar behavioral patterns to those observed in the experiments of this chapter.

4.5. Conclusions

In this chapter, we compared the performance of locking and optimistic methods of concurrency control in a firm deadline RTDBS environment. Detailed experiments were carried out with two representative algorithms, 2PL-HP and OPT-BC. Our experiments demonstrated that OPT-BC outperforms 2PL-HP over a wide range of system loading and resource availability. This is a surprising result since optimistic algorithms perform worse than locking protocols in resource-limited conventional DBMSs. The improved performance of OPT-BC seen here stems primarily from the firm-deadline feature of discarding late transactions. In this context, the delayed conflict resolution policy of OPT-BC aids in making better conflict decisions than 2PL-HP, which resolves conflicts immediately. Since the conflict resolution period of OPT-BC and 2PL-HP are common to algorithms of their respective classes, we conclude that optimistic algorithms are basically better suited than locking algorithms to the firm-deadline environment.

A second surprise is that OPT-BC achieved superior performance in spite of being *priority-indifferent*, unlike 2PL-HP, which used priorities to "help" transactions make their deadlines. From a performance standpoint, therefore, we conclude that in the firm real-time domain, even a "vanilla" optimistic algorithm can perform better than a locking algorithm that is "tuned" to the real-time environment. The development of an optimistic algorithm that incorporates transaction priorities to further decrease the number of missed deadlines is the subject of the next chapter.

CHAPTER 5

REAL-TIME OPTIMISTIC ALGORITHMS

5.1. Introduction

In the previous chapter, we studied the relative performance of locking and optimistic methods of concurrency control in a firm deadline RTDBS environment. Those experiments demonstrated some fundamental aspects of firm-deadline RTDBSs that result in optimistic concurrency control outperforming locking. The conventional optimistic algorithm used in those experiments, OPT-BC, did not factor in transaction deadlines in making data conflict resolution decisions. However, it still outperformed 2PL-HP, a priority-cognizant locking algorithm. The following question then naturally arises: Can priority information be used to improve the performance of the optimistic algorithm and thus further decrease the number of late transactions?

A straightforward way to introduce priority would be to use priority information in the resolution of data conflicts, that is, to resolve data conflicts always in favor of the higher priority transaction. For optimistic algorithms, this would mean that a low priority transaction would never be allowed to restart a higher priority transaction. This approach has two problems, however: First, giving preferential treatment to high priority transactions may result in an increase in the number of missed deadlines. This can happen if helping one high priority transaction make its deadline causes several lesser priority transactions to miss their deadlines. Second, if fluctuations can occur in transaction priorities, repeated conflicts between a pair of transactions may be resolved in some cases in favor of one transaction and in other cases in favor of the other transaction. This would hinder the progress of both transactions and result in degraded performance. Therefore, a priority-cognizant optimistic algorithm must address these two problems in order to perform better than a simple optimistic scheme.

In optimistic algorithms, data conflicts are detected only at the time of transaction validation. Therefore, different approaches to conflict resolution can be introduced only at this juncture. A validating transaction can follow one of three paths: *commit, restart or wait*. In the commit option, the validating transaction commits, restarting conflicting transactions in the process. In the restart option, the validating transaction restarts itself. Finally, in the wait option, the validating transaction neither commits nor restarts, but simply waits. A validating transaction that chooses the wait option may later, after waiting for a while, follow either the commit or restart options.

In this chapter, we develop several new real-time optimistic concurrency control algorithms based on the above options. A performance study of these algorithms shows one of them, **WAIT-50**, to be especially promising. The WAIT-50 algorithm incorporates a **priority wait** mechanism that makes low priority transactions wait for conflicting high priority transactions to complete, thus enforcing preferential treatment for high priority transactions. To address the first problem raised above, WAIT-50 features a **wait control** mechanism. This mechanism monitors transaction conflict states and, with a simple "50 percent" rule, dynamically controls when and for how long a transaction is made to wait. The second problem is handled by having the priority wait mechanism resolve conflicts in a manner that results in the commit of at least one of the conflicting transactions.

In this chapter, we first discuss the limitations of OPT-BC and then outline alternative methods of adding priority information to optimistic algorithms. Finally, we compare the performance of various real-time optimistic algorithms, including WAIT-50, to that of OPT-BC. All transactions have the same value in the experiments of this study, and the performance metric is the number of missed deadlines.

5.2. Limitations of OPT-BC

The validation algorithm of OPT-BC can be succinctly written as:

```
restart all conflicting transactions;
commit the validating transaction;
```

With this validation algorithm, transactions reaching their validation stage always commit, and do not consider the priorities of the transactions that they restart in the process. To illustrate the problem caused by this unilateral commit, consider the scenario in Figure 5.1, where the execution profile of two concurrently executing transactions, X and Y, is shown. X has an arrival time A_X and deadline D_X, and Y has an arrival time A_Y and deadline D_Y. Assume that transaction X, by virtue of its earlier deadline, has a higher priority than transaction Y. Now, consider the situation where at time $t = val_Y$, when transaction X is close to completion, transaction Y reaches its validation point and detects a conflict with X. Under the OPT-BC algorithm, Y would immediately commit and in the process restart X. Restarting X at this late stage guarantees that it has no chance of meeting its deadline.

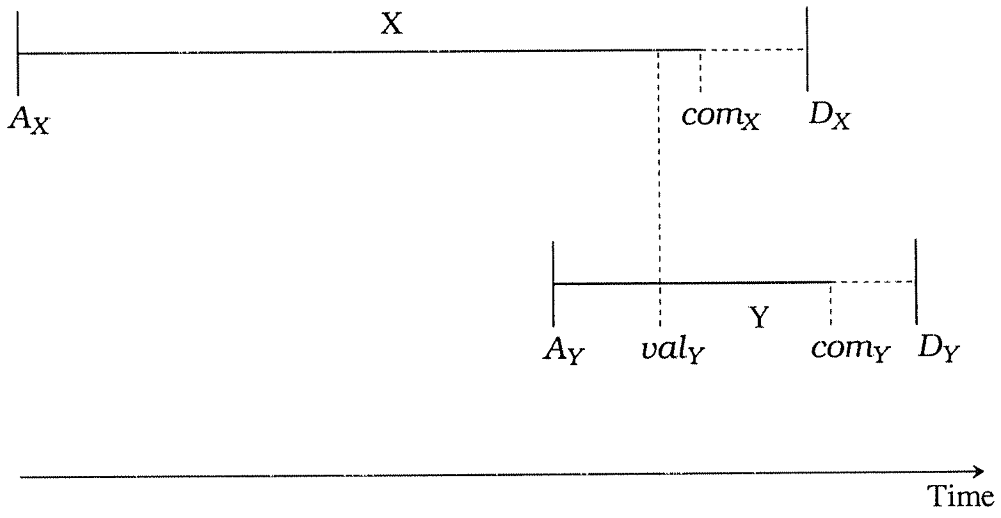

Figure 5.1: Poor OPT-BC decision

In the above example, a priority-cognizant algorithm would have *prevented Y from committing* until X had completed. With this decision, we could possibly gain the completion of *both* transactions X and Y before their deadlines, as shown in Figure 5.1, where X completes at time $t = com_X$ and Y completes later at time $t = com_Y$.

It is important to note that preventing transaction Y from committing does not *guarantee* that both transactions will eventually make their deadline. In fact, there are several possible outcomes: (1) transactions X and Y both miss their deadlines; (2) only one of the transactions makes its deadline; or, (3) both transactions make their deadlines. The above example, however, serves to show that OPT-BC's indifference to transaction priorities can result in degraded performance. Another drawback of OPT-BC is that it has an inherent bias against long transactions, just like the classical optimistic algorithm.[1] The use of priority information in resolving conflicts may help to counter this bias as well.

5.3. Prioritized Optimistic Algorithms

In this section, we describe several new optimistic algorithms that try to address the problems of OPT-BC without sacrificing the performance-beneficial aspects of the broadcast commit scheme. In the subsequent discussion, the term *conflict set* is used to denote the set of currently running transactions that conflict with a validating transaction. The acronym *CHP (Conflicting Higher Priority)* is used for the set of transactions that are in the conflict set and have a higher priority than the validating transaction. Similarly, the acronym *CLP (Conflicting Lower Priority)* refers to transactions that are in the conflict set and have a lower priority than the validating transaction. In this section, our aim is to motivate the development of the algorithms and discuss, at an intuitive level, their potential strengths and weaknesses.

[1] Locking algorithms in general, and 2PL-HP in particular, do not have this bias.

The example of the previous section, illustrating a poor conflict decision by OPT-BC, showed that a mechanism is required to prevent low priority transactions that conflict with higher priority transactions from unilaterally committing. The following two options are available:

(1) **Restart:** The low priority transaction is restarted.

(2) **Block:** The low priority transaction is blocked.

In the following section, we develop two algorithms, OPT-SACRIFICE and OPT-WAIT, based on the restart and block options, respectively. WAIT-50 is then developed as an extension of the OPT-WAIT algorithm.

5.3.1. OPT-SACRIFICE

In this algorithm, a transaction that reaches its validation stage checks for conflicts with currently executing transactions. If conflicts are detected and at least one of the transactions in the conflict set is a CHP transaction, then the validating transaction is restarted – that is, it is *sacrificed* in an effort to help the higher priority conflicting transactions make their deadlines. The validation algorithm of OPT-SACRIFICE can therefore be written as:

```
if CHP transactions in conflict set then
  restart the validating transaction;
else
  restart transactions in conflict set;
  commit the validating transaction;
```

In the example of Figure 5.1, the OPT-SACRIFICE algorithm would restart transaction T_Y at time $t = val_Y$ due to its conflict with the higher priority transaction X.

OPT-SACRIFICE is priority-cognizant and satisfies the goal of giving preferential treatment to high priority transactions. It suffers, however, from two potential problems. First, there is the problem of **wasted sacrifices**, where a transaction is restarted on behalf of another transaction that is later discarded. Such restarts are useless and cause performance degradation. Second, priority reversals can lead to **mutual sacrifices**, where a pair of transactions restart themselves for each other, thus hindering the progress of both transactions. For example, the situation may arise where transaction A is restarted for transaction B because B's priority is currently greater than A's priority, and at a later time, transaction B is restarted for transaction A because A's priority is now greater than B's priority. These two drawbacks of OPT-SACRIFICE are analogous to the "wasted restarts" and "mutual restarts" problems of 2PL-HP that were described in the previous chapter.

5.3.2. OPT-WAIT

The OPT-WAIT algorithm incorporates a **priority wait** mechanism: a transaction that reaches validation and finds CHP transactions in its conflict set is temporarily "put on the shelf", that is, it is made to wait and not allowed to commit immediately. This gives the higher priority transactions a chance to make their deadlines first. The validation algorithm of OPT-WAIT can therefore be written as[2]:

```
while CHP transactions in conflict set do
    wait;
restart transactions in conflict set;
commit the validating transaction;
```

In the example of Figure 5.1, the OPT-WAIT algorithm would force transaction Y to wait at $t = val_Y$, without committing, allowing transaction X to complete first. Of course, X's completion could cause Y to be restarted.

There are several reasons which suggest that the priority wait mechanism may have a positive impact on performance, and these are outlined below:

(1) In keeping with the original goal, precedence is given to high-priority transactions.

(2) The problem of "wasted sacrifices" does not exist here. This is because the waiter is immediately "taken off the shelf" and committed if the CHP transaction misses its

[2] A scheme for implementing the priority-wait mechanism method is discussed in Appendix A.

deadline and is discarded (assuming, of course, that no other CHP transactions remain).

(3) The problem of "mutual sacrifices" does not exist here. This is because the waiter will stop waiting and commit if the CHP transaction that is being waited for ever becomes a CLP transaction (assuming, of course, that no other CHP transactions remain).

(4) A blocking effect is derived from the transaction wait process – this results in conservation of resources, which can be beneficial to performance [Agra87].

(5) The fact that a CHP transaction commits does not necessarily imply that the waiting transaction has to be restarted (!).

The last point requires further explanation: The key observation here is that if transaction *A* conflicts with transaction *B*, this does not necessarily mean that the converse is true [Robi82, Hard84]. This is explained as follows: Under the broadcast commit scheme, a validating transaction *A* is said to conflict with another transaction *B* if and only if

$$WriteSet_A \cap ReadSet_B \neq \phi \qquad (1)$$

We will denote such a conflict from transaction *A* to *B* by $A \rightarrow B$. For transaction B to also conflict with transaction A, i.e. for $B \rightarrow A$, it is necessary that

$$WriteSet_B \cap ReadSet_A \neq \phi \qquad (2)$$

From Equations (1) and (2), it is obvious that $A \rightarrow B$ does not imply $B \rightarrow A$. Therefore, if in fact $B \rightarrow A$ is not true, then by committing the transactions in the order (B,A) instead of the order (A,B), both transactions can be successfully committed without restarting either one.

As per the explanation given above, it is possible with OPT-WAIT's waiting scheme for the CHP transaction to commit, and for the waiting transaction to commit immediately afterwards. This means that the conflict between the waiter and the CHP transaction was resolved *without* a restart. Therefore, the priority wait mechanism has a potential to actually *eliminate* some data conflicts. In Appendix C, a simple probabilistic analysis of the extent to which waiting can cause a reduction in data conflicts is presented.

Although the waiting scheme has many positive features, it has some drawbacks as well. One potential drawback is that if a transaction finally commits after waiting for some time, it causes all of its CLP transactions to be restarted at a later point in time. This decreases the chances of these transactions meeting their deadlines, and also wastes resources. A second drawback is that the validating transaction may develop new conflicts during its waiting period, thus causing an increase in conflict set sizes and leading to more restarts. Another way to view this is to realize that waiting causes objects to be, in a sense, "locked" for longer periods of time. Therefore, while waiting has the capability to reduce the probability of a restart-causing conflict between a particular pair of transactions, it can simultaneously increase the probability of having a larger *number* of conflicts per transaction. This increase may be substantial when there are many concurrently executing transactions in the system.

5.3.3. WAIT-50

The WAIT-50 algorithm is an extension of OPT-WAIT. In addition to the priority wait mechanism, it also incorporates a **wait control** mechanism. This mechanism monitors transaction conflict states and dynamically decides when, and for how long, a low priority transaction should be made to wait for its CHP transactions. A transaction's conflict state is assumed to be characterized by the index *HPpercent*, which is the percentage of the transaction's total conflict set that is formed by CHP transactions. The operation of the wait mechanism is conditioned on the value of this index. In WAIT-50, a simple "50 percent" rule is used – a validating transaction is made to wait only while HPpercent ≥ 50, that is, while half or more of its conflict set is composed of higher priority transactions. The validation algorithm of WAIT-50 can therefore be written as:

```
while CHP transactions in conflict set and HPpercent ≥ 50 do
    wait;
restart transactions in conflict set;
commit the validating transaction;
```

The aim of the wait control mechanism is to detect when the beneficial effects of waiting, in terms of giving preference to high priority transactions and decreasing pairwise conflicts, are outweighed by its drawbacks, in terms of later restarts and an increased number of conflicts. The "effective priority" of a conflict set is measured in terms of its high priority component. Based on this measure, it is decided whether or not waiting could be expected to increase the number of in-time transactions. Therefore, while OPT-WAIT and OPT-BC represent the extremes with regard to waiting (OPT-WAIT always waits for a CHP transaction, and OPT-BC never waits), WAIT-50 is a *hybrid* algorithm that controls the amount of waiting based on transaction conflict states. In fact, we can view OPT-WAIT, WAIT-50, and OPT-BC as all being special cases of a general algorithm **WAIT-X**, where X is the cutoff HPpercent level, with X taking on the values 0, 50, and ∞, respectively, for these algorithms.

We conducted experiments to evaluate the performance of the various optimistic algorithms, and the following section describes the experimental setup and the results that were obtained.

5.4. Experiments and Results

We began our experiments by evaluating the algorithms for the baseline experiment of the previous chapter. Subsequently, for reasons explained in the following discussion, a new baseline experiment was developed. Further experiments were constructed around the new baseline experiment by varying a few parameters at a time. These experiments evaluate the impact of data contention, resource contention, and the wait control mechanism parameters.

For the most part, the experimental framework of this chapter is identical to that of the previous chapter. A difference, however, lies in the deadline formulas that are used in the baseline experiments. Deadline formula DF1, which assigns the same slack ratio to all transactions, was used in the baseline experiment of the previous chapter. Here, in order to evaluate the effects of variability in transaction slack ratios, deadline assignment formula DF2 is used in the baseline experiment. In the following description, the terms FIX-SR (Fixed Slack

Ratio) and VAR-SR (Variable Slack Ratio) are used to distinguish between the baseline experiments of the previous chapter and this chapter.

5.4.1. Experiment 1: Fixed Slack Ratio

The settings of the workload parameters and resource parameters for the FIX-SR baseline experiment are listed in Table 5.1 (reproduced from Table 4.1). For this experiment, Figures 5.2(a) and 5.2(b) show Miss Percent behavior under normal load and heavy load conditions, respectively.

From this set of graphs, we observe that OPT-SACRIFICE performs poorly over the entire operating region, and in fact performs worse than the priority-indifferent OPT-BC. The poor performance of OPT-SACRIFICE is primarily due to its "wasted sacrifices" problem, which was discussed in Section 5.3.1. Turning our attention to the wait-based algorithms, OPT-WAIT and WAIT-50, we observe that at normal loads, their performance is superior to that of OPT-BC. This is due to the beneficial effects of their priority cognizance. At heavy loads, however, WAIT-50 and OPT-WAIT behave identically to OPT-BC. This is because, with heavy resource contention, it is uncommon for a low priority transaction to get access to the resources. Consequently, transactions usually reach their validation stage only when their deadline is quite close. Also, since the priority assignment is Earliest Deadline, the conflicting higher priority transactions of a waiting transaction usually end up missing their deadlines since their dead-

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	10
WriteProb	0.25	*NumDisks*	20
DeadlineFormula	DF1	*PageCPU*	10ms
LSF	4.0	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		

Table 5.1: FIX-SR Baseline Parameter Settings

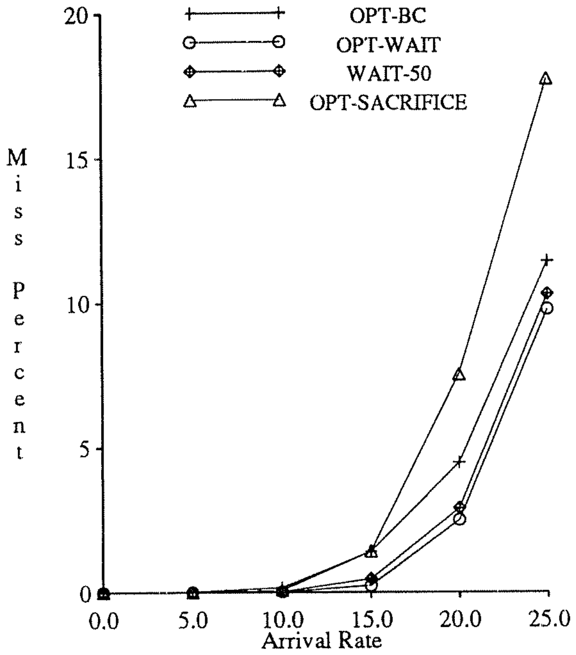

Figure 5.2(a): FIX-SR (Normal)

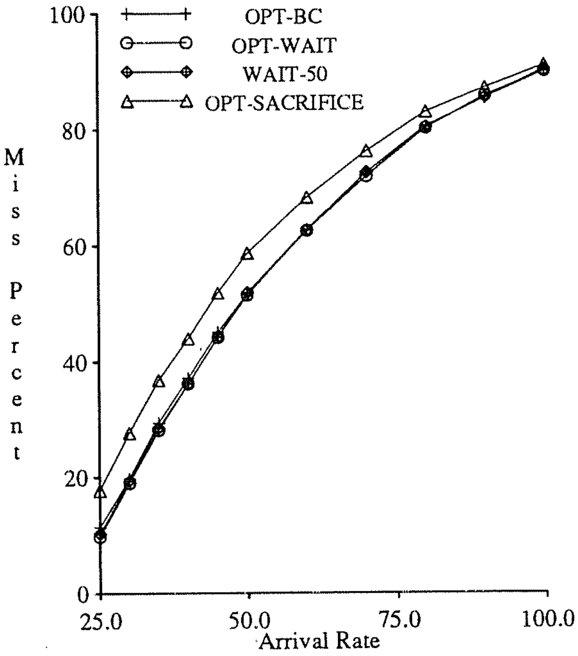

Figure 5.2(b): FIX-SR (Heavy)

lines are even closer than that of the waiting transaction. This means that waiting transactions are rarely restarted and usually commit after waiting for a short while. Therefore, the priority wait mechanism has very limited impact, and WAIT-50, OPT-WAIT, and OPT-BC become essentially the same algorithm.

When the above experiment is carried out with infinite resources, Figures 5.3(a) and 5.3(b) are obtained. In these figures, we observe that OPT-SACRIFICE performs much worse than the wait-based algorithms, relative to the corresponding performance under finite resources. For the most part, OPT-SACRIFICE also performs worse than OPT-BC. The performance of OPT-SACRIFICE is further degraded here since the high data contention levels lead to a steep increase in the number of conflicts and, consequently, in the number of "wasted sacrifices".

Turning our attention to OPT-WAIT, we observe that it performs very well at low levels of data contention due to the beneficial effects of its priority cognizance. As data contention

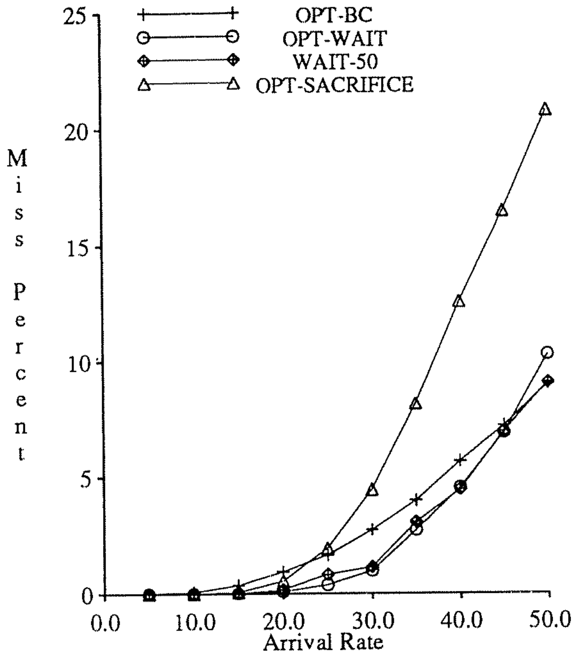

Figure 5.3(a): ∞ Resources (Normal)

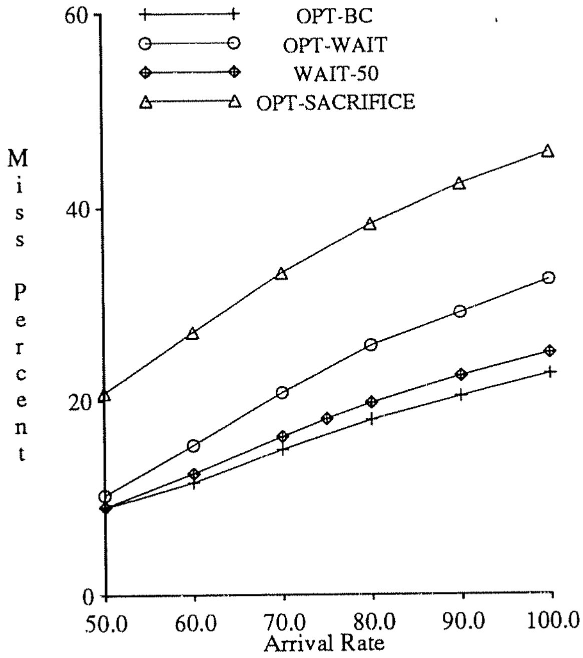

Figure 5.3(b): ∞ Resources (Heavy)

increases, however, its performance steadily degrades. Finally, at high data contention levels, it performs noticeably worse than OPT-BC. The reason for OPT-WAIT's poor performance in this region is that its priority wait mechanism causes a significant increase in the average number of transactions in the system. This increase in population leads to an increased number of data conflicts and results in degraded performance.

Moving on to the WAIT-50 algorithm, we observe that it provides the *best overall* performance. WAIT-50 behaves like OPT-WAIT under low data contention, and behaves like OPT-BC under high data contention. The explanation for this behavior of WAIT-50 is provided in the next section.

The above set of experiments were encouraging because they showed that there are performance benefits to be gained by using priority-cognizant algorithms. It was all the more encouraging that these performance improvements were obtained despite all transactions having the same slack ratio. A fixed transaction slack ratio reduces the likelihood that a *priority inversion restart*, where a transaction is restarted by a lower priority transaction, results in the higher priority transaction missing its deadline. This creates favorable circumstances for OPT-BC since the detrimental effects of its priority insensitivity are reduced. When transactions have different slack ratios, however, the chances of a priority inversion restart proving "fatal" to the restarted higher priority transactions are increased. This is because transactions with small slack ratios have limited completion opportunities and therefore easily succumb to restarts from lower priority transactions.

5.4.2. Experiment 2: Variable Slack Ratio

The VAR-SR baseline experimented was developed in order to obtain a workload with variation in transaction slack ratios. This experiment uses deadline assignment formula DF2 to generate variation in transaction slack ratios. The deadline-related workload parameters, *LSF* and *HSF*, are set at 2.0 and 6.0, respectively. This means that half the transactions have a slack ratio of 2.0 and the other half have a slack ratio of 6.0. Therefore, the *mean* transaction slack ratio is the same as that of the FIX-SR baseline experiment, namely 4.0. The remaining workload parameter settings and resource parameter settings are the same as those of the FIX-SR baseline experiment (see Table 5.1). In the subsequent discussions, we will compare the performance of only the OPT-BC, OPT-WAIT and WAIT-50 algorithms since OPT-SACRIFICE invariably performed worse than the wait-based algorithms.

For the VAR-SR baseline experiment, Figures 5.4(a) and 5.4(b) show the behavior of the algorithms under normal load and heavy load, respectively. From this set of graphs we observe that the priority-cognizant algorithms, WAIT-50 and OPT-WAIT, now perform *significantly better* than OPT-BC under normal loads. The beneficial effects of their priority cognizance show up to a greater exten here since the priority inversion restarts of OPT-BC are more harmful when transactions may have small slack ratios.

When the same experiment is carried out under infinite resources, Figures 5.5(a) and 5.5(b) are obtained. We observe that, at normal loads, the performance improvement of the wait-based algorithms over OPT-BC is greater relative to the corresponding performance under

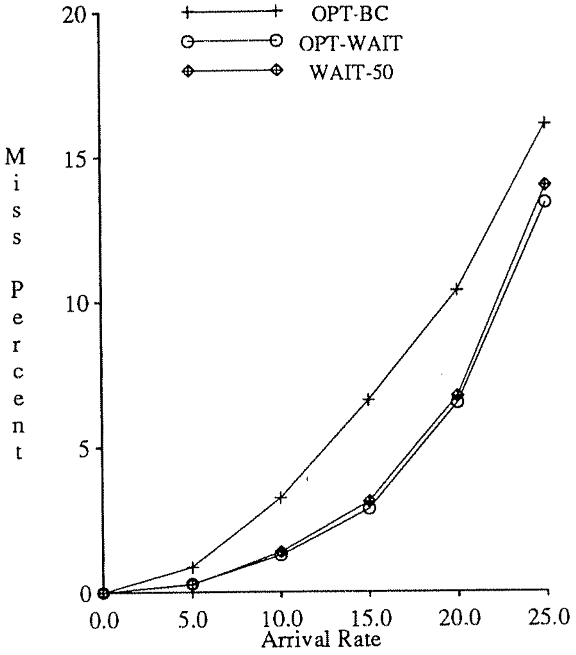

Figure 5.4(a): VAR-SR (Normal)

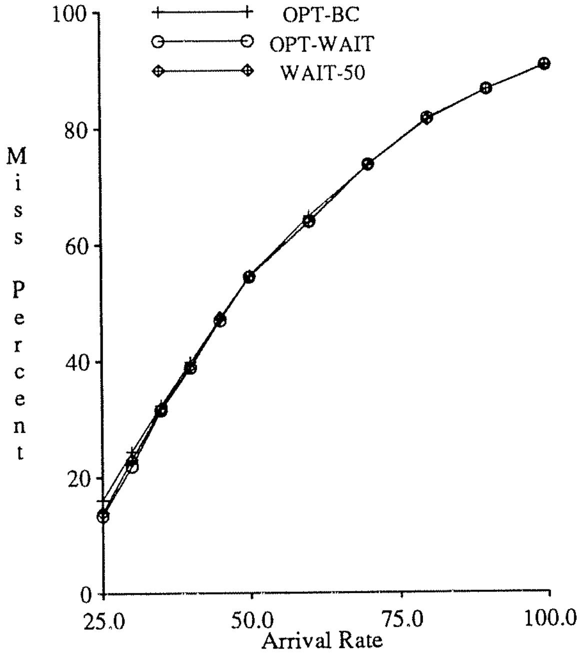

Figure 5.4(b): VAR-SR (Heavy)

finite resources. The reason for this behavior is the following: Under resource contention, the priority indifference of OPT-BC is masked to some extent by the priority scheduling at the resources. Under pure data contention, however, the negative effects of OPT-BC's priority-indifference show up in their entirety.

Considering performance at high loads, we observe that OPT-WAIT performs worse than OPT-BC. This is due to the beneficial aspects of waiting being more than countered by its negative aspects in terms of later restarts and increased conflicts. In contrast, WAIT-50, which had been behaving like OPT-WAIT at normal loads, now changes character and behaves like OPT-BC. Therefore, WAIT-50 turns in the best overall performance by behaving like OPT-WAIT when data contention is low and like OPT-BC when data contention is high.

From the results of all of the above experiments, we can conclude that WAIT-50 provides performance close to that of either OPT-BC or OPT-WAIT in operating regions where they behave well, and provides the same or slightly better performance at intermediate points.

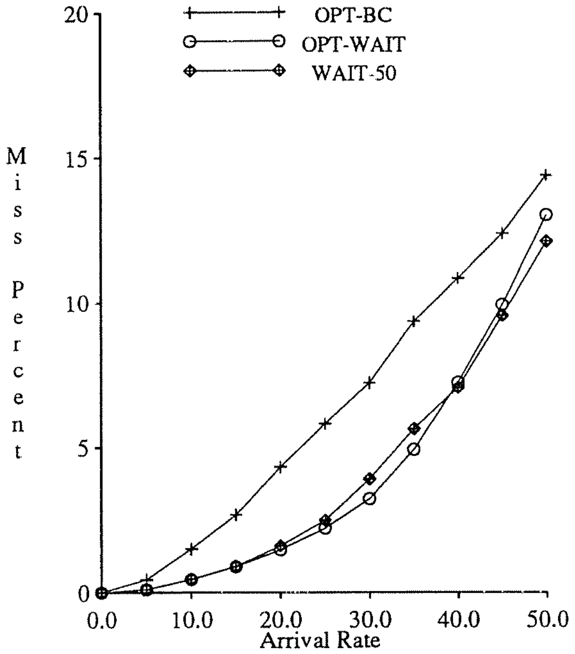

Figure 5.5(a): ∞ Resources (Normal)

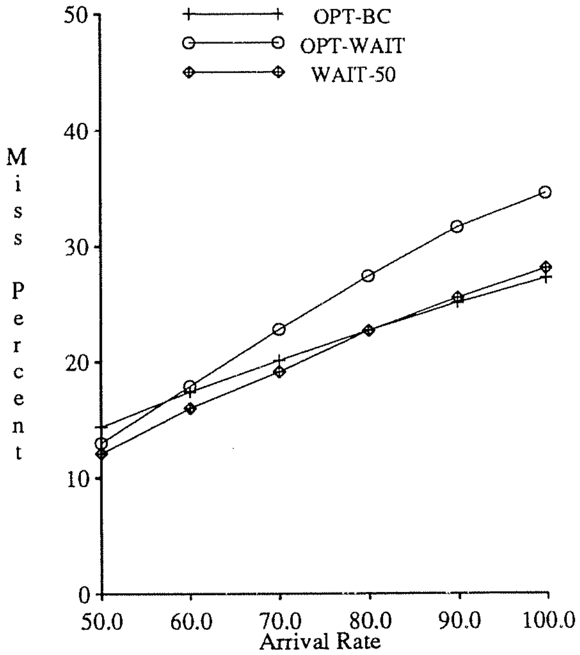

Figure 5.5(b): ∞ Resources (Heavy)

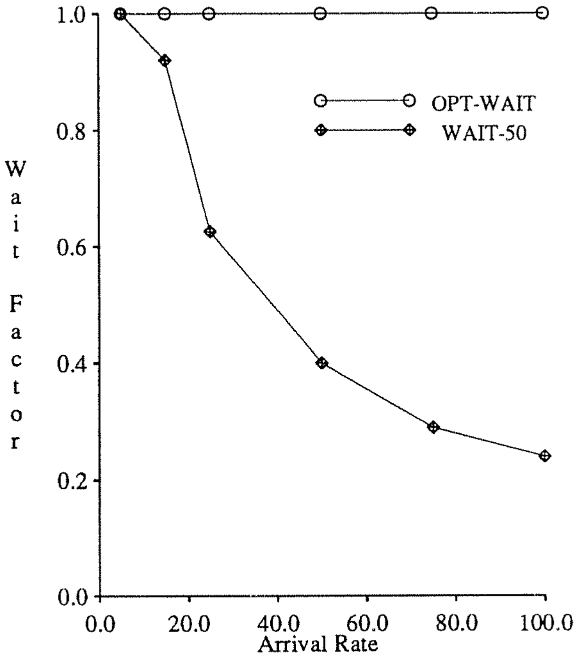

Figure 5.5(c): Wait Factor (∞ Resources)

Therefore, in an overall sense, WAIT-50 effectively integrates priority and waiting into the optimistic concurrency control framework. The control mechanism is clearly quite effective at deciding when the benefits of waiting, in terms of helping high priority transactions to make their deadlines, are outweighed by the drawbacks of causing an increased number of conflicts. In Figure 5.5(c), the "wait factor" of WAIT-50 with respect to that of OPT-WAIT is plotted. The wait factor measures the total time spent in priority-waiting using WAIT-50, normalized by the waiting time of OPT-WAIT.[3] From this figure, it is clear that WAIT-50's wait factor is close to that of OPT-WAIT at low contention levels but decreases steadily as the data contention level is increased. Therefore, while OPT-WAIT and OPT-BC represent the extremes with regard to waiting, WAIT-50 gracefully controls the degree of waiting to match the level of data contention in the system.

5.4.3. Experiment 3: Wait Control Mechanism

Our next experiment examined the effect of the choice of 50 percent as the cutoff value for the HPpercent control index. Keeping the workload and system parameters the same as those of the VAR-SR baseline experiment, we measured the performance of WAIT-25 and WAIT-75 under conditions of both finite and infinite resources. Figures 5.6(a) and 5.6(b) show the results of the finite resources experiment under normal load and heavy load, respectively, while Figures 5.7(a) and 5.7(b) give the corresponding results under infinite resources. From these graphs, we observe that lowering the cutoff value to 25 percent results in slightly improved normal load performance but worsened heavy load performance. This behavior is due to the increased wait factor that is delivered by the decreased cutoff setting. On the other hand, raising the cutoff value to 75 percent has the opposite effect: the normal load performance becomes worse while the heavy load performance improves slightly. This behavior is due to the decreased priority cognizance that is delivered by the increased cutoff setting.

[3] The wait factor of OPT-BC is trivially zero as the algorithm has no wait component.

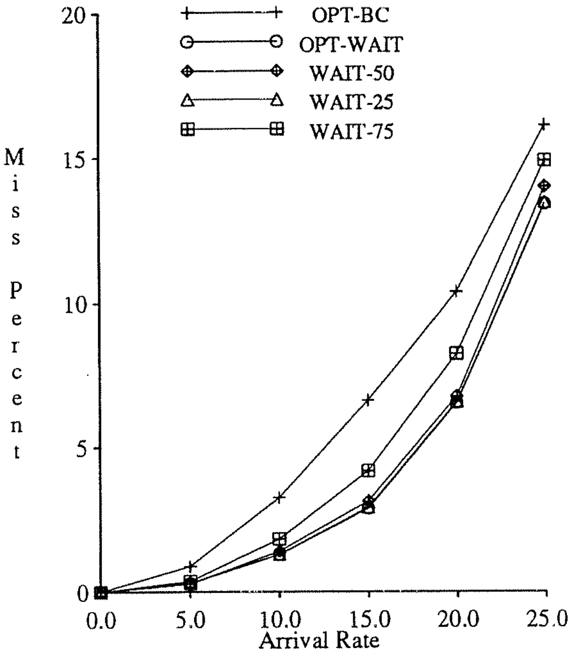

Figure 5.6(a): Wait Control (Normal)

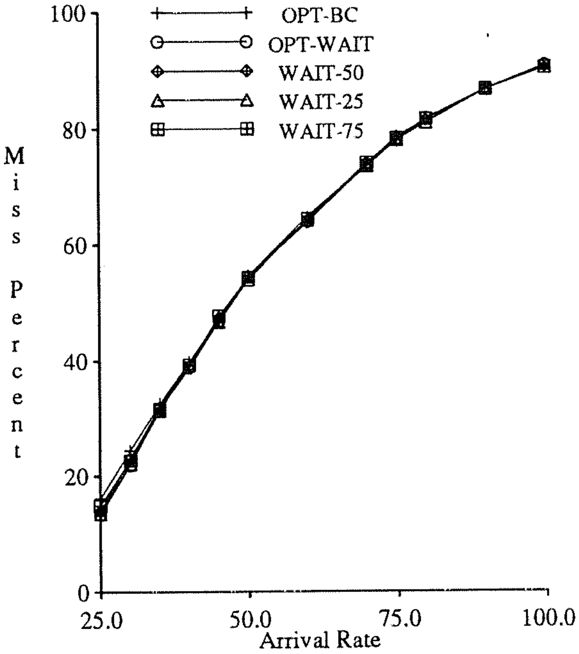

Figure 5.6(b): Wait Control (Heavy)

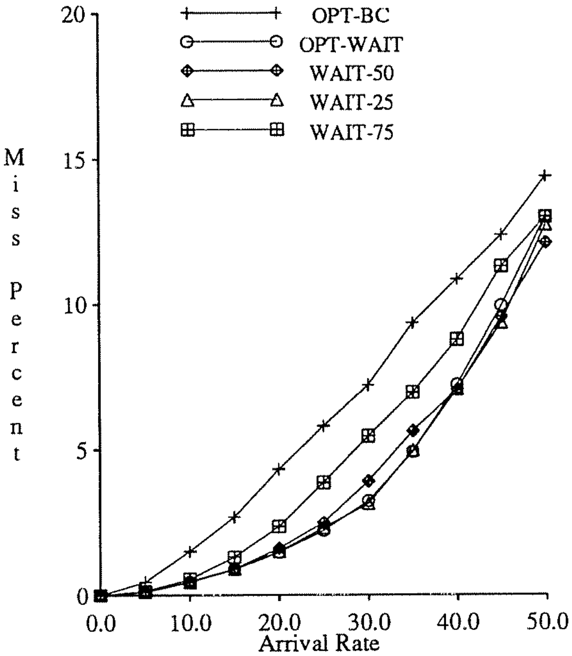

Figure 5.7(a): ∞ Resources (Normal)

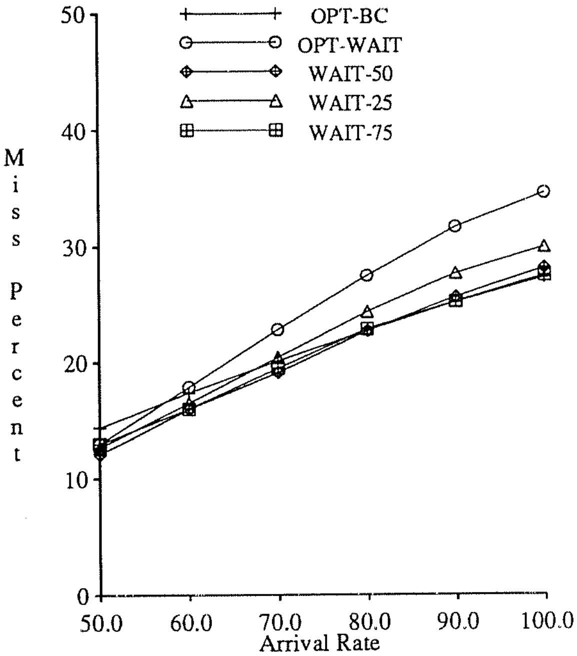

Figure 5.7(b): ∞ Resources (Heavy)

Based on these results, a 50 percent cutoff setting appears to establish a balanced tradeoff between the opposing forces of priority cognizance and increased data contention, and provides good performance across the entire range of loading. The basic philosophy is that priority-based waiting is quite beneficial under light loads, when data contention levels are low, Under heavy loads, however, when data contention levels are high, waiting becomes detrimental to performance. WAIT-50 is effective in dynamically changing its behavior to match the level of data contention in the system.

5.4.4. Other Experiments

We studied the sensitivity of the results of the experiments described in this chapter to changes in transaction write probabilities, deadline formulas, and priority assignment policies. In these sensitivity experiments, the various optimistic algorithms showed similar behavioral patterns to those observed in the experiments described here (refer to [Hari90b] for details). In some of the experiments, however, the control mechanism of WAIT-50 slightly underestimated the benefits of waiting at normal loads under infinite resources, and WAIT-50 therefore did not track OPT-WAIT as closely as in the other experiments. This is related to the fact that the control mechanism uses a single, simple parameter to characterize transaction conflict states and therefore cannot be expected to completely capture the performance tradeoffs of waiting versus not waiting. For the most part, however, the choice of 50 percent for the cutoff parameter was effective in delivering good performance.

Another observation of the above experiments was that the performance improvement of the wait-based algorithms over OPT-BC at normal loads increased with the range of variation in slack ratios. This is because the "fatality factor" of priority inversion restarts increases with the variance in slack ratios.

Another interesting observation was that the heavy load performance of OPT-WAIT became extremely poor for workloads with high transaction write probabilities. This is due to two reasons: First, the high write probability generates heavy data contention which, in combination with the population increase effect of the wait mechanism, results in a steep increase in the number of restarts. Second, the conflict-elimination capability of OPT-WAIT decreases with increased write probability. In fact, at a write probability of 1.0, OPT-WAIT's conflict elimination capability completely vanishes since *all* conflicts become *bi-directional*.

5.5. Conclusions

In this chapter, we addressed the problem of incorporating transaction deadline information in optimistic concurrency control algorithms. We presented and studied several new real-time optimistic concurrency control algorithms. Among these algorithms, one of them, WAIT-50, was observed to provide the best overall performance over a range of workloads and operating conditions. The WAIT-50 algorithm monitors transaction conflict states and gives precedence to urgent transactions in a controlled manner. It features a priority wait mechanism that provides preferential treatment to high priority transactions, eliminates some data conflicts by changing the commit order of transactions, and provides immunity to priority fluctuations.

While the priority wait mechanism of WAIT-50 works well at low data contention levels, it can cause significant performance degradation at high contention levels by generating a steep increase in the number of data conflicts. A simple wait control mechanism consisting of a "50 percent" rule is used in the WAIT-50 algorithm to address this problem. The "50 percent" rule is the following: If half or more of the transactions conflicting with a validating transaction are of higher priority, the transaction is made to wait; otherwise, it is allowed to commit. WAIT-50 was shown to provide significant performance gains at normal loads over the priority-insensitive OPT-BC algorithm, especially when there was variance in transaction slack ratios. In summary, we conclude that the WAIT-50 algorithm successfully utilizes transaction priority information to provide improved performance in a stable manner.

In the experiments of this chapter, the Earliest Deadline priority assignment was used to assign transaction priorities. This assignment guarantees that a transaction at its deadline has the highest priority in the system. This means that a waiting transaction in either OPT-WAIT or WAIT-50 will never be still waiting at its deadline. Priority assignments that do not provide the above guarantee, however, raise an interesting question for the wait-based algorithms: If the deadline of a waiter is reached during the waiting process, should the waiter be committed or discarded? A decision to commit implies that the algorithms lose some of their priority cognizance. On the other hand, a decision to discard makes "wasted sacrifices" possible since the higher priority transactions being waited for may themselves be eventually discarded. Obviously, the choice of committing or discarding should be made based on the associated performance impacts, but predicting these impacts is not a simple problem. This is an open research issue that we hope to address in the future.

CHAPTER 6

ADAPTIVE EARLIEST DEADLINE

6.1. Introduction

In the preceding two chapters, we focused on alternatives for the concurrency control component of a firm deadline real-time database system. For priority assignment, we used the Earliest Deadline mapping which is widely used in existing real-time systems. In this context, we showed that optimistic concurrency control is fundamentally better suited than locking to the firm deadline environment, and developed a new high-performance real-time optimistic CC algorithm.

If we take a second look at the experimental results of Chapters 4 and 5, an additional observation is that the performance of all the concurrency control algorithms degrades steeply under heavy load conditions. Since all of them are affected in this manner, it would seem that their instability does not lie in the concurrency control scheduling but, instead, in the common Earliest Deadline priority policy. In this chapter, we therefore shift our focus from the concurrency control aspect to the **priority assignment** aspect of real-time transaction scheduling.

A common observation of earlier studies on real-time database systems (e.g. [Abbo88, Abbo89]) is that the Earliest Deadline priority policy, compared to other priority assignments, minimizes the number of late transactions under low or moderate levels of resource and data contention. This is due to Earliest Deadline giving the highest priority to transactions that have the least remaining time in which to complete. These earlier studies have also observed, however, that the performance of Earliest Deadline steeply degrades in an overloaded system. This is because, under heavy loading, transactions gain high priority only when they are close to their deadlines. Gaining high priority at this late stage may not leave sufficient time for transactions to complete before their deadlines. Under heavy loads, then, a fundamental weakness of the Earliest Deadline policy is that it assigns the highest priority to transactions that are close to missing their deadlines, thus delaying other transactions that might still be able meet their deadlines.

From the above discussion, the following question naturally arises: Can a priority assignment policy be developed based on the Earliest Deadline approach that stabilizes its overload performance without sacrificing its light-load virtues? A scheme based on simple real-time principles was presented in [Jens85] for realizing this objective. In order to use this scheme, which was developed in the context of task scheduling in real-time operating systems, a-priori knowledge of task processing requirements is necessary. Unfortunately, as mentioned in earlier chapters, knowledge about transaction resource and data requirements is usually unavailable in database applications. Therefore, the scheme described in [Jens85] cannot be used and methods applicable to transaction scheduling in real-time database systems have to be developed. The challenge is to develop a computationally simple priority policy that adapts to system loading levels and thereby provides better performance.

In this chapter, we present **Adaptive Earliest Deadline (AED)**, a new priority assignment algorithm that stabilizes the overload performance of Earliest Deadline. The AED algorithm uses a feedback control mechanism to achieve this objective and does not require knowledge of transaction characteristics. We compare the performance of the AED algorithm to that of Earliest Deadline and other fixed priority mappings. All transactions have the same value in the experiments of this study, as in the preceding chapters, and the performance metric is the number of missed deadlines.

6.2. Priority Mappings

The choice of priority mappings in an RTDBS is limited when transactions are distinguished only by their deadlines. Apart from the previously discussed Earliest Deadline policy, there are a few other mappings mentioned in the literature that fit our operating constraints. These mappings are described first in this section. Subsequently, the new priority mapping, Adaptive Earliest Deadline, is presented. The priority assignments of all the mappings are such that smaller transaction priority (P_T) values reflect higher system priority. The details of the mappings are presented below.

6.2.1. Earliest Deadline (ED)

The Earliest Deadline mapping assigns higher priority to transactions with earlier deadlines, and the transaction priority assignment is $P_T = D_T$.

6.2.2. Latest Deadline (LD)

The Latest Deadline mapping is the opposite of the Earliest Deadline mapping. It gives higher priority to transactions with later deadlines, and the transaction priority assignment is $P_T = \frac{1}{D_T}$. We expect that, for many real-time applications, newly-submitted transactions will tend to have later deadlines than transactions already executing in the system. Therefore, it seems plausible that the Latest Deadline mapping would rectify the overload drawback of Earliest Deadline by giving transactions high priority early on in their execution.

6.2.3. Random Priority (RP)

The Random Priority mapping randomly assigns priorities to transactions without taking into account any of their characteristics. The transaction priority assignment is P_T = Random $(0, \infty)$. The performance obtained with this priority mapping reflects the performance impact of the mere existence of *some* fixed priority ordering among the transactions.

6.2.4. No Priority (NP)

The No Priority mapping gives all transactions the *same* priority, and the transaction priority assignment is $P_T = 0$. This effectively means that scheduling at each resource is done in order of arrival to the resource (i.e., local FCFS). The performance obtained under this mapping should be interpreted as the performance that would be observed if the real-time database system were replaced by a conventional DBMS but the feature of discarding late transactions was retained.

6.2.5. Adaptive Earliest Deadline (AED)

Our new Adaptive Earliest Deadline priority assignment algorithm modifies the classical Earliest Deadline mapping based on the following observation: Given a set of tasks with deadlines that can all *somehow* be met, an Earliest Deadline priority ordering will *also* meet all (or most of) the deadlines [Jens85]. The implication of this observation is that in order to maximize the number of in-time transactions, an Earliest Deadline schedule should be used among the largest set of transactions that can all be completed by their deadlines. The flaw of the pure ED mapping is that it uses this schedule among *all* transactions in the system, even when the system is overloaded. Instead, if the system could "magically" determine at arrival time what the eventual completion status (in-time or late) of each transaction would be if these transactions were scheduled by a clairvoyant scheduler, it should use an Earliest Deadline schedule among the in-time transactions and discard the late transactions. In the absence of such perfect foresight, alternate methods are required to estimate the completion status of a transaction. The AED algorithm takes the approach of using a feedback control process as the estimation method.

6.2.5.1. Group Assignment

In the AED algorithm, transactions executing in the system are collectively divided into two groups, **HIT** and **MISS**, as shown in Figure 6.1. Transactions in the HIT group are expected to complete before their deadlines, while transactions in the MISS group are expected to miss their deadlines. Each transaction, upon arrival, is assigned to one of the groups. The assignment is done in the following manner: The newly-arrived transaction is assigned a

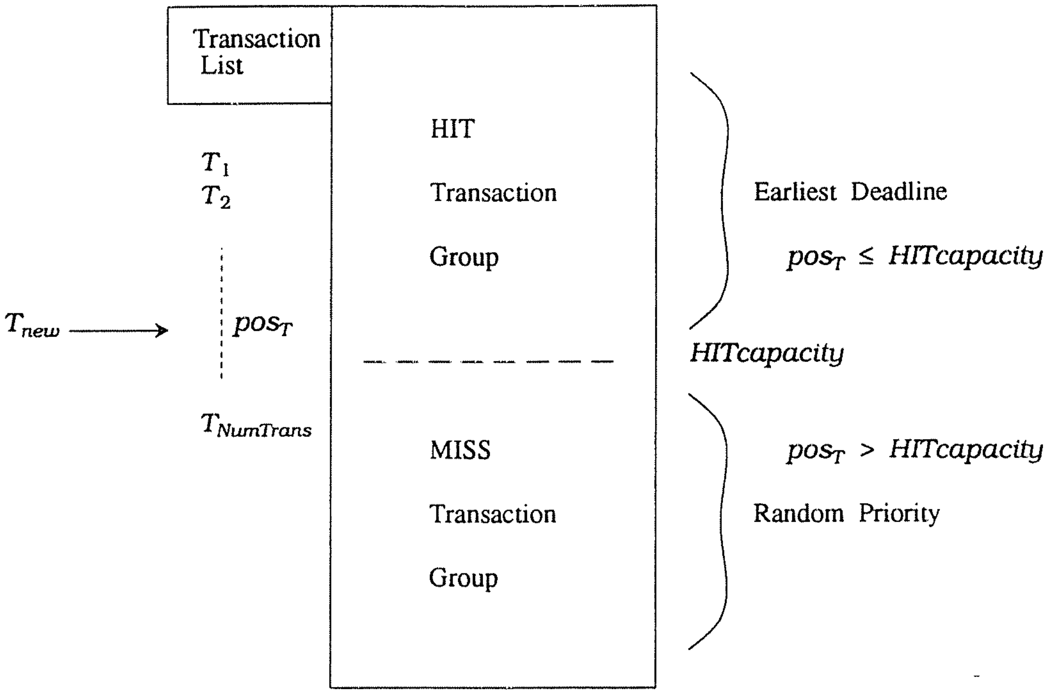

Figure 6.1: AED Priority Mapping

unique[1] key, I_T, with the key being a randomly chosen integer. The transaction is then inserted into a *key-ordered* list of the transactions currently in the system, and its position in the list, pos_T, is noted. If pos_T is less than **HITcapacity**, which is a dynamic control variable of the AED algorithm, the new transaction is assigned to the *HIT* group; otherwise, it is assigned to the *MISS* group.

6.2.5.2. Priority Assignment

After a new transaction is assigned to a group, it is then assigned a priority using the following formula:

[1] Transaction keys are sampled uniformly over the set of 32-bit integers. In the unlikely event that a new key matches that of an existing transaction, the key is re-sampled until a unique key is obtained.

$$P_T = \begin{cases} (0,\ D_T,\ I_T) & \text{if } Group = HIT \\ (1,\ 0,\ I_T) & \text{if } Group = MISS \end{cases}$$

With this priority assignment scheme, all transactions in the *HIT* group have a higher priority than transactions in the *MISS* group. (Since the priority is a vector, priority comparisons are made in lexicographic order.) The transaction priority ordering in the *HIT* group is Earliest Deadline. In contrast, the priority ordering in the *MISS* group is Random Priority since the I_T's are selected randomly. For transactions in the *HIT* group that may have identical deadlines, the I_T component of the priority serves to break the tie, thus ensuring a *total* priority ordering.

An important point to note about the AED algorithm is that transactions retain their initial group and priority assignments for the entire duration of their residence in the system.

6.2.5.3. Discussion

The goal of the AED algorithm is to collect the largest set of transactions that can be completed before their deadline in the *HIT* group. It trys to achieve this by controlling the size of the *HIT* group, using the *HITcapacity* setting as the control variable. Then, by having an Earliest Deadline priority ordering within the *HIT* group, the algorithm incorporates the observation made in [Jens85] that was discussed earlier. The motivation for having a Random Priority mapping in the *MISS* group is explained in Section 6.4.

We define the "hit ratio" of a transaction group to be the fraction of transactions in the group that meet their deadline. Using this terminology, we would ideally like to have a hit ratio of 1.0 in the *HIT* group and a hit ratio of 0.0 in the *MISS* group, since this combination of hit ratios ensures that *all* the "doable" transactions, and *only* these transactions, are in the *HIT* group. Achieving this goal would require absolute accuracy in predicting the completion status of a transaction; this is impossible as the RTDBS has no advance knowledge of transaction processing requirements and of the future transaction arrival pattern. From a practical standpoint, therefore, our aim is to maintain a high hit ratio in the *HIT* group and a low hit ratio in the *MISS* group. The key to achieving this lies in the *HITcapacity* computation, which is discussed below.

6.2.5.4. HIT Capacity Computation

A feedback process that employs system output measurements is used to set the *HITcapacity* control variable, as shown in Figure 6.2. The measurements used are **HitRatio(HIT)** and **HitRatio(ALL)**. HitRatio(HIT) is the fraction of transactions in the *HIT* group that are making their deadline, while HitRatio(ALL) is the corresponding measurement over *all* transactions in the system. Using these measurements, and denoting the number of transactions currently in the system by *NumTrans*, the *HITcapacity* is set with the following two-step computation:

```
(STEP 1)   HITcapacity := HitRatio(HIT) * HITcapacity * 1.05;
(STEP 2)   if HitRatio(ALL) < 0.95 then
      HITcapacity := Min (HITcapacity, HitRatio(ALL) * NumTrans * 1.25 );
```

STEP 1 of the *HITcapacity* computation incorporates the feedback process in the setting of this control variable. By conditioning the new *HITcapacity* setting based on the observed hit ratio in the *HIT* group, the size of the *HIT* group is adaptively changed to achieve a 1.0 hit ratio. Our goal, however, is not just to have a HitRatio(HIT) of 1.0, but to achieve this goal with the *largest* possible transaction population in the *HIT* group. It is for this reason that STEP 1

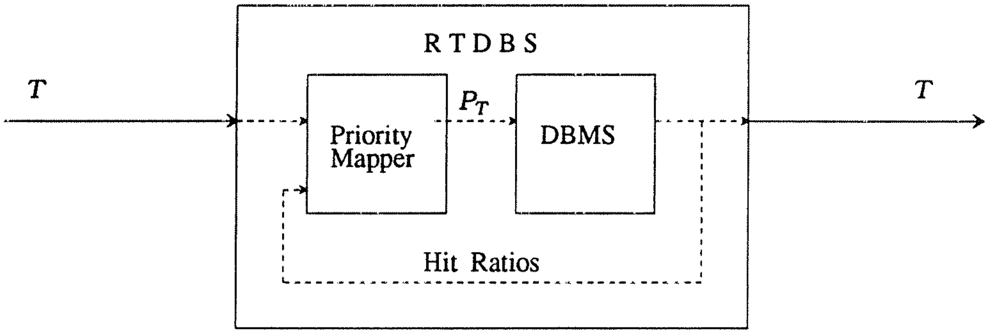

Figure 6.2: Feedback Model

includes a 5 percent expansion factor. This expansion factor ensures that the *HITcapacity* is steadily increased until the number of transactions in the *HIT* group is large enough to generate a HitRatio(HIT) of 0.95.[2] At this point, the transaction population size in the *HIT* group is close to optimal, and the *HITcapacity* remains stabilized at this setting (since $0.95 * 1.05 \approx 1.0$).

STEP 2 of the *HITcapacity* computation is necessary to take care of the following special scenario: If the system experiences a long period where HitRatio(ALL) is close to 1.0 due to the system being lightly loaded, it follows that HitRatio(HIT) will be virtually 1.0 over this extended period. In this situation, the *HITcapacity* can become very large due to the 5 percent expansion factor, that is, there is a "runaway" effect. (For a 5 percent expansion factor and a HitRatio(HIT) of 1.0, the *HITcapacity* doubles every fifteen cycles of the feedback loop.) If the transaction arrival rate now increases such that the system becomes overloaded (signaled by HitRatio(ALL) falling below 0.95), incrementally bringing the *HITcapacity* down from its artificially high value to the right level could take a considerable amount of time. This means that the system may enter the unstable high-miss region of Earliest Deadline as every new transaction will be assigned to the *HIT* group due to the high *HITcapacity* setting. To prevent this from occurring, an upper bound on the *HITcapacity* value is used in STEP 2 to deal with the *transition* from a lightly-loaded condition to an overloaded condition. The upper bound is set to be 25 percent greater than an estimate of the "right" *HitCapacity* value. (The choice of 25 percent is based on our expectation that the estimate is fairly close to the "right" value.) This estimate is derived by computing the number of transactions that are *currently* making their deadlines. After the *HITcapacity* is quickly brought down in this fashion to near the appropriate setting, the HitRatio(HIT) value then takes over as the "fine tuning" mechanism in determining the *HITcapacity* setting. The mechanism for computing HitRatio(ALL) and HitRatio(HIT)

[2] The 0.95 cutoff used for HitRatio(HIT) is unrelated to the 0.95 value used for HitRatio(ALL) in STEP 2.

is described in the next section.

6.2.5.5. Feedback Process

The feedback process for setting the *HITcapacity* control variable has two parameters, *HITbatch* and *ALLbatch*. These parameters determine the sizes of transaction batches that are used in computing the output hit ratios. The feedback process operates in the following manner: Assume that the *priority mapper* has just set the *HITcapacity* value. The next *HITbatch* transactions that are assigned to the *HIT* section of the bucket are marked with a special label. At the RTDBS output, the completion status (in-time or late) of these specially-marked transactions is monitored. When the last of these *HITbatch* transactions exits the system, HitRatio(HIT) is measured as the fraction of these transactions that completed before their deadline. HitRatio(ALL), on the other hand, is continuously measured at the output as the hit ratio of the last *ALLbatch* transactions that exited from the system.[3] After each measurement of HitRatio(HIT), the HitRatio(HIT) value is fed back to the *priority mapper* along with the current HitRatio(ALL) value. The *priority mapper* then reevaluates the *HITcapacity* setting, after which the whole process is repeated.

At system initialization time, both HitRatio(HIT) and HitRatio(ALL) are set to 1.0, while the *HITcapacity* is set equal to the database administrator's estimate of the number of concurrent transactions that the RTDBS can handle without missing deadlines. Note that this estimate does not have to be accurate; even if it were grossly wrong, it would not impact system performance in the long run. The error in the estimate only affects how long it takes the *HITcapacity* control variable to reach its steady state value at system startup time.

[3] The reason that HitRatio(ALL) is measured continuously, rather than in batches, is that it serves only to detect a transition from a lightly loaded to a heavily loaded system and is therefore not usually involved in the HITcapacity computation feedback process.

6.3. Concurrency Control Algorithms

We have shown in Chapters 4 and 5 that optimistic algorithms outperform locking algorithms in a firm deadline RTDBS. Those chapters assumed an Earliest Deadline priority policy. One of the goals of the performance evaluations in this chapter is to determine whether the results of Chapters 4 and 5 are also applicable under the Adaptive Earliest Deadline priority mapping.

If the OPT-WAIT or WAIT-50 optimistic algorithms are used with the AED priority mapping, it is possible for a transaction to still be priority-waiting when its deadline is reached. As mentioned in Chapter 5, it is not clear whether such a transaction should be committed or sacrificed. In this chapter, we compare the performance of 2PL-HP with two versions of OPT-WAIT and WAIT-50. In the first version, a waiting transaction always commits at its deadline. In the second version, the waiting transaction is always discarded, that is, it is sacrificed. In the following experiment section, the "commit" version of OPT-WAIT and WAIT-50 are referred to by OPT-WAIT(C) and WAIT-50(C), while the "sacrifice" versions are referred to as OPT-WAIT(S) and WAIT-50(S), respectively.

6.4. Experiments and Results

In this section, we present performance results for our experiments comparing the various priority mappings in a firm-deadline RTDBS environment. We discuss our results with regard to the impact of resource contention, data contention, and fluctuations in the transaction arrival pattern. In order to represent a general workload where transactions may have a range of slack ratios, deadline formula DF3 is used to assign transaction deadlines in all the experiments of this chapter.

While describing the AED algorithm in Section 6.2, we mentioned two parameters, *HITbatch* and *ALLbatch*, that determine the sample sizes used in computing transaction hit ratios. The choice of the sample sizes is constrained by two opposing considerations: A large sample size reduces the responsiveness of the feedback system. On the other hand, a small sample size makes the system extremely sensitive to short-term input fluctuations. Experimentation with several different sample sizes showed that settings between 10 and 30 delivered a reasonable tradeoff between these opposing considerations for the workloads considered here. We therefore chose 20 as the setting for the *HITbatch* and *ALLbatch* parameters.

6.4.1. Experiment 1: Resource Contention (RC)

Our first experiment investigated the performance of the priority mappings when resource contention is the sole performance limiting factor. The settings of the workload parameters and system parameters for this experiment are listed in Table 6.1. The *WriteProb* parameter, which specifies the update probability for each page that is read, is set to 0.0 to eliminate data contention. Therefore, no concurrency control is required in this experiment since all transactions are *queries*. The slack factor parameters, *LSF* and *HSF*, are set to 1.33 and 4.0, respectively, thus ensuring a significant spread in transaction slack ratios.

For this experiment, Figures 6.2(a) and 6.2(b) show the Miss Percent results under normal load and heavy load conditions, respectively. From this set of graphs, we observe that under normal loads, the ED (Earliest Deadline) mapping misses the fewest deadlines among the non-adaptive priority mappings. As the system load is increased, however, the performance of ED steeply degrades, and its performance actually is close to that of NP (No Priority) under heavy loads. This is because under heavy loads, where the resources become saturated,

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16.0 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	8
WriteProb	0.0	*NumDisks*	16
DeadlineFormula	DF3	*PageCPU*	10ms
LSF	1.33	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		

Table 6.1: Baseline Parameter Settings

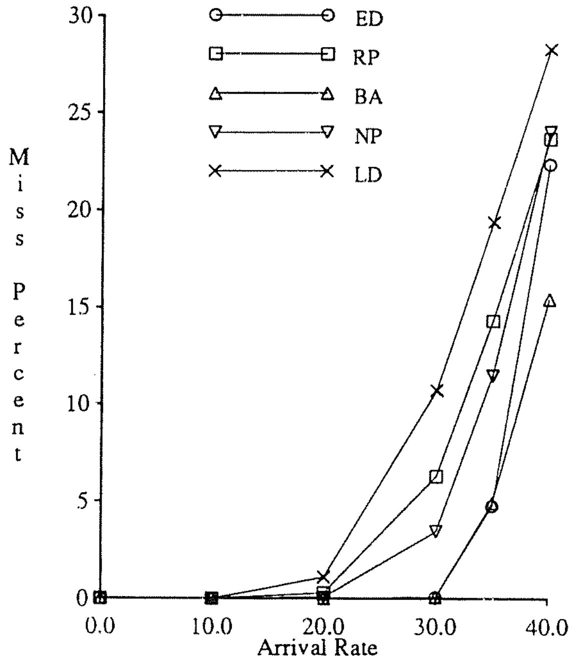

Figure 6.2(a): RC (Normal Load)

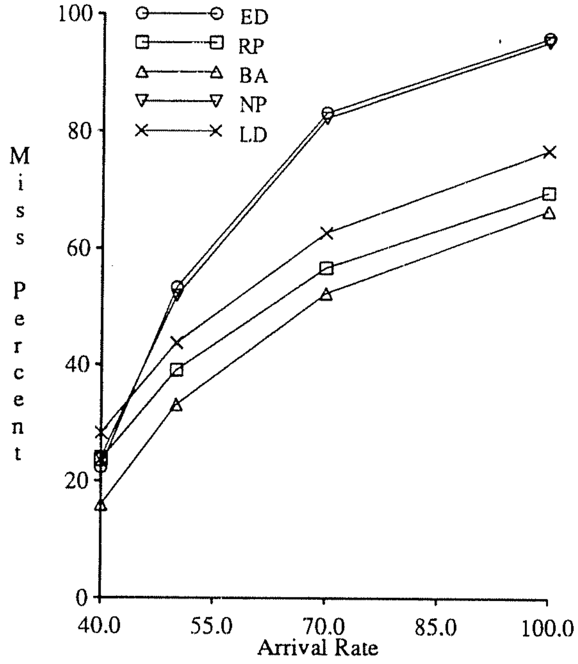

Figure 6.2(b): RC (Heavy Load)

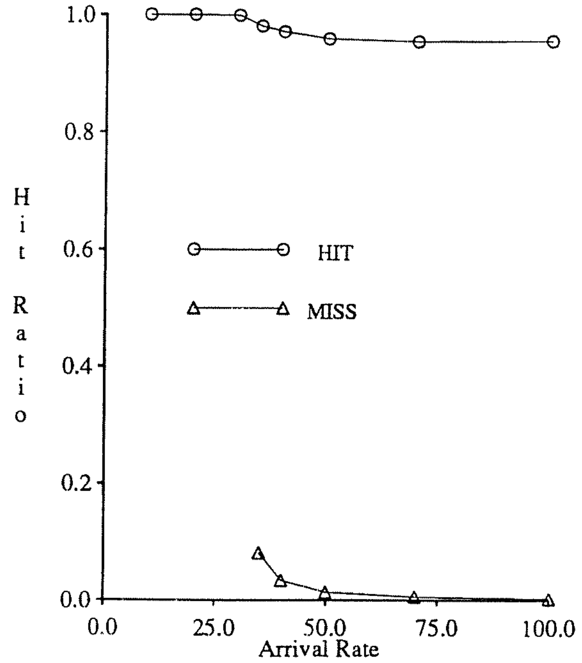

Figure 6.2(c): AED (Group Hit Ratio)

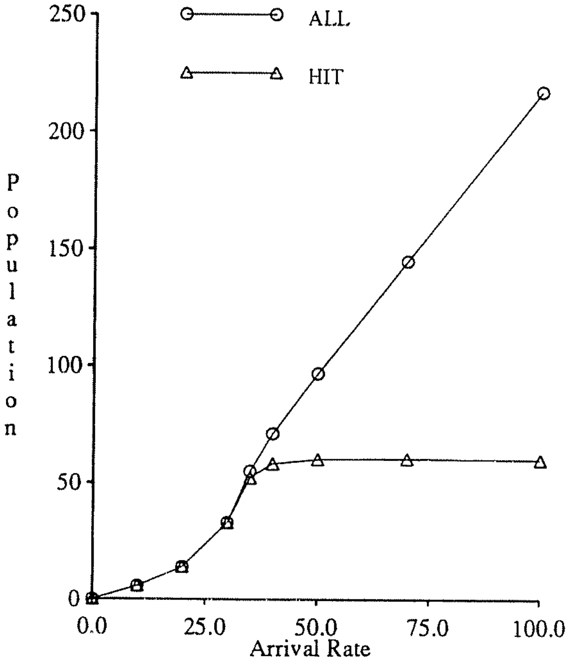

Figure 6.2(d): AED (Population)

transactions under ED and NP make progress at similar *average* rates. This is explained as follows: Under NP, every transaction makes slow but steady progress from the moment of arrival in the system since all transactions have the same priority. Under ED, however, a new transaction usually has a low priority since its deadline tends to be later than those of the transactions already in the system. Therefore, transactions tend to start off at low priority and become high priority transactions only as their deadline draws close. This results in transactions making little progress initially, but making fast progress as their deadline approaches. The *net* progress made by ED, however, is about the same as that of NP. This was experimentally confirmed by measuring the average progress that had been made (i.e. number of steps executed) by transactions that missed their deadline; indeed, we found that once the resources become saturated, the average progress made by transactions is almost the same for NP and ED.

Turning our attention to the RP (Random Priority) mapping, we observe that it behaves poorly at normal loads since it does not take transaction time constraints into account. At heavy loads, however, it (surprisingly) performs significantly better than ED. The reason for this behavior is the following: Under ED, as discussed above, transactions gain priority slowly. At heavy loads, this gradual process of gaining priority causes most transactions to miss their deadlines. The RP mapping, on the other hand, due to its static random assignment of priorities, allows some transactions to have a high priority right from the time when they first arrive in the system. Such transactions tend to make their deadlines, and therefore RP ensures that there is always some fraction of the transactions in the system that will almost certainly make their deadlines.

Focusing next on the LD (Latest Deadline) mapping, we observe that it performs worse than all the other mappings at normal loads. The reason is that this mapping gives the highest priority to transactions that have loose time constraints, thus tending to miss the deadlines of transactions that have tight time constraints. At heavy loads, it performs better than ED, however, since transactions with loose time constraints continue to make their deadlines as they retain high priority for a longer period of time.

Moving on to the AED mapping, we observe that it behaves identically to Earliest Deadline at normal loads. As the overload region is entered, it changes its behavior to be qualitatively similar to that of RP, and in fact, performs even better than RP. Therefore, in an overall sense, it delivers the best performance. In Figure 6.2(c), the hit ratios in the *HIT* and *MISS* groups are shown. It is clear from this figure that a hit ratio of more than 0.90 in the *HIT* group and less than 0.10 in the *MISS* group is achieved throughout the entire loading range. (The HitRatio(MISS) is not shown for arrival rates of less than 35 transactions/sec because *no* transactions are assigned to the *MISS* group in this region as the *HitCapacity* is greater than the maximum number of transactions in the system.)

The AED performance results indicate that the feedback mechanism used to divide transactions into the *HIT* and *MISS* groups is effective and achieves the goal of having a high hit ratio in the *HIT* group and a low hit ratio in the *MISS* group. In Figure 6.2(d), the average number of transactions in the *HIT* group and the average number of transactions in the whole system are plotted. From this figure, we can conclude that for the given workload, the RTDBS can successfully schedule about 60 concurrently executing transactions under an Earliest Deadline schedule. For system loadings above this level, a pure Earliest Deadline schedule causes most transactions to miss their deadline since they receive high priority only when they are close to missing their deadline. The AED mapping, however, by dividing transactions into different priority groups, creates a "core set" of transactions in the *HIT* group that are virtually certain to make their deadlines independent of system loading conditions. Viewed from a different perspective, we have revisited the classic multi-programming thrashing problem where increasing the number of transactions in a system can lead to a *decrease* in throughput. In our real-time framework, adding transactions to a set of transactions that can just be completed with an Earliest Deadline schedule results in an increase in the number of missed deadlines.

As promised in the description of the AED algorithm in Section 6.2.5, we now provide the rationale for using a Random Priority mapping in the *MISS* group. The reason is the following: Transactions assigned to the *MISS* group essentially "see" a heavily-loaded system due to having lower priority than the transactions in the *HIT* group. Since our experiments show Random Priority to have the best performance among the non-adaptive mappings at heavy loads, we have chosen this priority ordering for the *MISS* group. The reason that AED does better than the pure RP mapping at heavy loads is that the transaction population in the *HIT* group is sufficiently large that using ED, instead of RP, among this set has an appreciable performance effect. As the loading level is increased even further, however, the performance of AED would asymptotically reach that of RP since the number of transactions in the *MISS* group would be much larger than the number in the *HIT* group.

Summarizing the results of the above set of experiments, we can draw the following conclusions for the query workloads examined in this section: First, the AED mapping provides the best overall performance among the priority mappings examined here. Its feedback mechanism is effective in detecting overload conditions and limiting the size of the *HIT* group to a level that can be handled by Earliest Deadline scheduling. Second, at normal loads, the Earliest Deadline priority ordering meets most transaction deadlines and is therefore the right priority mapping in this region. At heavy loads, however, the Random Priority mapping delivers the best performance among the non-adaptive mappings due to guaranteeing the completion of some fraction of the transactions by assigning them high priority throughout their residence in the system.

Earlier studies have observed that the No Priority mapping performs worse than Earliest Deadline at low loads and about the same or worse at heavy loads. These behavioral characteristics were also seen in our experiments. In addition, Latest Deadline was observed to consistently perform worse than Random Priority for the workloads considered here. Therefore, for subsequent experiments, we will present results only for the Earliest Deadline, Random Priority and Adaptive Earliest Deadline priority mappings.

6.4.2. Experiment 2: Resource and Data Contention (RC + DC)

Our next experiment explored an RTDBS situation where both resource contention *and* data contention contribute towards system performance degradation. This was done by changing the write probability from 0.0 to 0.25, which implies that one-fourth of the data items that are read will also be updated. The settings of the remaining workload parameters are identical to those for the Resource Contention case (listed in Table 6.1). In this experiment, the 2PL-HP algorithm is used as the concurrency control mechanism. (A comparison of the various concurrency control alternatives is made in Experiment 4).

For this experiment, Figures 6.3(a) and 6.3(b) show the Miss Percent results for the various priority mappings under normal load and heavy load conditions, respectively. From these figures it is evident that Earliest Deadline performs the best at low loads, while Random Priority is superior at heavy loads. The AED mapping behaves almost as well as ED at low loads and behaves like RP in the overload region, thus providing the best overall performance. In this experiment, the increased contention levels cause the population in the *HIT* group to be quite small compared to the overall system population at heavy loads. Therefore, using ED instead of RP in this group does not have an appreciable performance effect. This is why the performance of AED approaches that of RP at a lower load than in the pure resource contention experiment (see Figure 6.2(b)).

In Figure 6.3(c), we show the hit ratio in the HIT group and the corresponding HITcapacity settings as a function of time for an arrival rate of 30 transactions/sec.. This figure graphically shows how the HITcapacity keeps increasing whenenver the HITratio is 0.95 or greater and decreases for a HITratio of less than 0.95. We also observe some steep falls occurring occasionally in the HITcapacity setting. These falls are due to STEP2 kicking in since it detects that the system has transitioned from a lightly loaded condition to a heavily loaded condition.

From the above results, we conclude that the AED algorithm delivers good performance across the entire loading range under both resource contention and data contention.

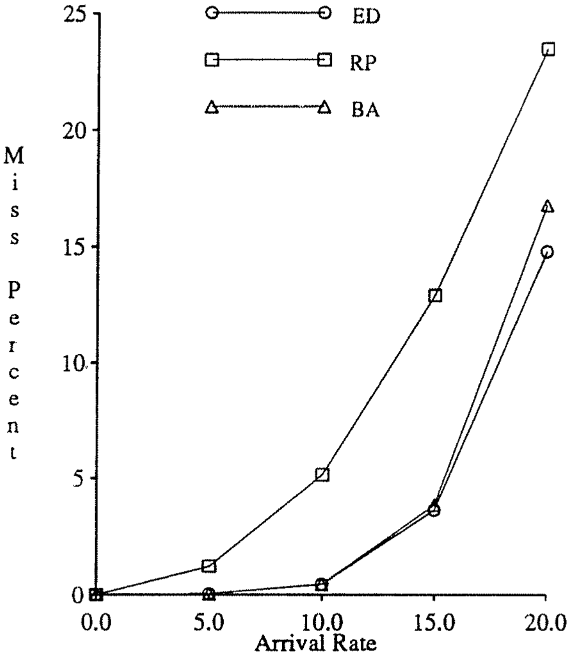

Figure 6.3(a): RC+DC (Normal Load)

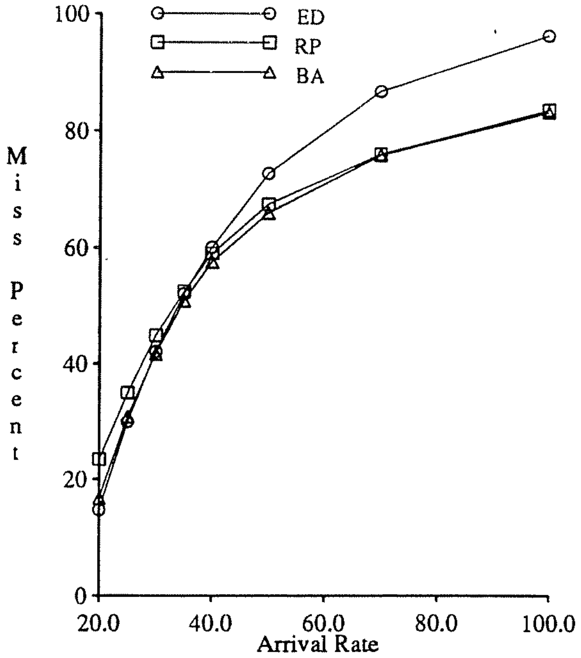

Figure 6.3(b): RC+DC (Heavy Load)

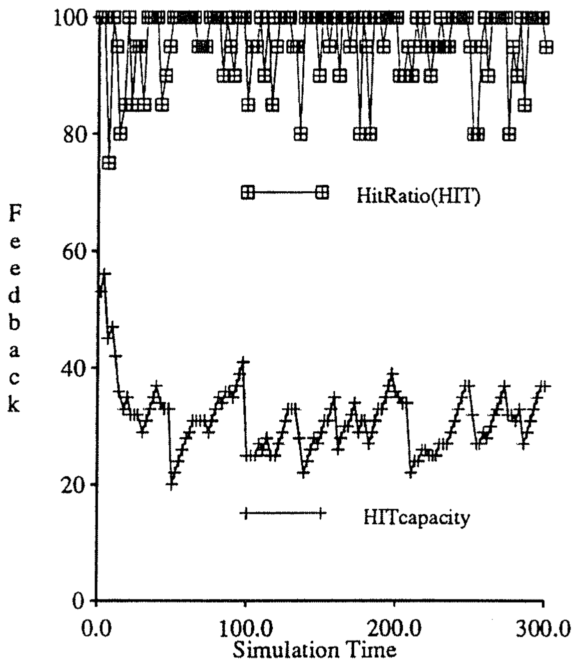

Figure 6.3(c): Feedback Process (Arr. Rate = 30)

6.4.3. Experiment 3: Bursty Arrivals

In the previously described experiments, each simulation was run for a particular arrival rate. In practice, however, the transaction arrival rate may change over time. Therefore, we also conducted experiments to determine how well the AED mapping could adapt to fluctuations in the transaction arrival pattern. For this experiment, the transaction arrival process was constructed in the following manner: The transaction arrival rate is repeatedly toggled between a base arrival rate and a secondary arrival rate. The time period for which each arrival rate is in effect is chosen from a common uniform distribution. This means that the *effective* transaction arrival rate is the average of the base and secondary arrival rates.

In our experiments with this type of transaction arrival process, the base arrival rate was kept fixed at 20 transactions/second and the experiment was conducted for different secondary arrival rates. The time period for which each arrival rate was in effect ranged uniformly between 10 and 40 seconds. For this workload, Figure 6.4 shows the Miss Percent characteristics as a function of the *secondary* arrival rate for the pure resource contention workload of Experiment 1. Figure 6.5 shows the corresponding Miss Percent characteristics for the workload of Experiment 2 which results in both resource and data contention. From these figures, it is evident that the performance characteristics of the AED mapping are similar to those seen for the fixed arrival rate experiments. This implies that the algorithm is robust with respect to fluctuations in the transaction workload pattern.

In Experiments 1-3, we have seen that the AED algorithm exhibits ED-like behavior in the light-load region and RP-like behavior in the overload region. From these results, it might appear that a much simpler approach than AED would be to switch from ED to RP (for all transactions) when the miss percentage exceeds a threshold. The threshold, of course, would be the miss level at which RP starts performing better than ED. The problem with this approach is that we do not a-priori *know* this changeover threshold. Also, the threshold is a function of workload characteristics and may vary dynamically with changes in the input workload. For

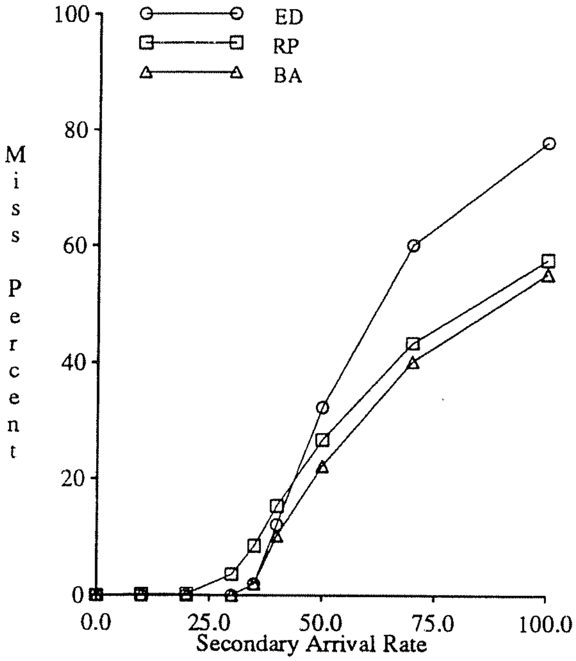

Figure 6.4: Bursty Arrivals (RC)

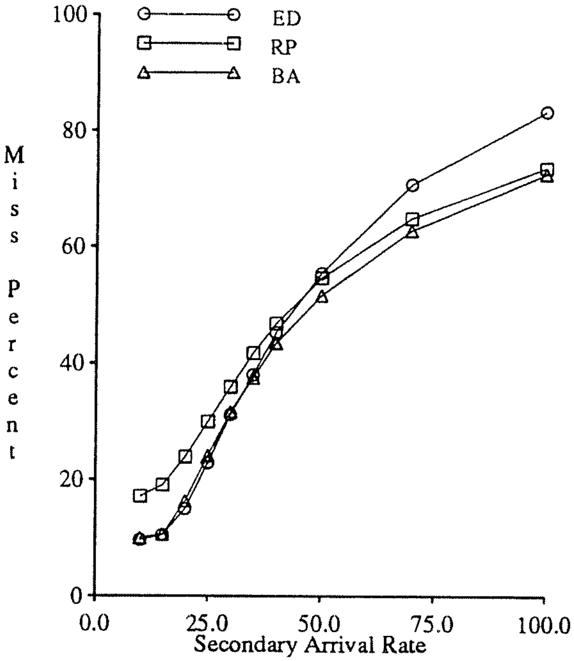

Figure 6.5: Bursty Arrivals (RC+DC)

example, in the pure resource contention experiment (Experiment 1), the ED to RP changeover miss percent threshold is about 20 percent (see Figure 6.2(a)); in the resource plus data contention experiment (Experiment 2), however, the threshold is about 50 percent (see Figure 6.3(b)). Therefore, while the AED algorithm is somewhat complicated, the complexity appears necessary to make the priority assignment truly adaptable to changing workload and system conditions. In addition, when resource contention is the main performance determinant, the AED algorithm performs somewhat better than RP at high loads.

6.4.4. Experiment 4: Concurrency Control

The experiments thus far in this chapter have shown that the AED priority assignment algorithm provides the best overall performance. In this experiment, we investigate the behavior of various concurrency control algorithms in association with the AED algorithm. We compare the performance of 2PL-HP with the commit and sacrifice versions of both OPT-WAIT and WAIT-50. The workload parameters are the same as for Experiment 2.

For this experiment, Figures 6.6(a) and 6.6(b) show the Miss Percent results under normal load and heavy load conditions, respectively. From these figures, we observe that the optimistic algorithms clearly outperform 2PL-HP over the entire loading range. This is important since it shows that optimistic algorithms are the concurrency control mechanism of choice with AED, just as they were with ED.

An interesting point is that the sacrifice and commit options of the wait-based algorithms show virtually no performance difference over the entire loading range. At high loads, this is because the priority-wait mechanism rarely comes into play – the heavy resource contention prevents low-priority transactions from reaching validation much before their deadlines. At low loads, AED is almost identical to an ED policy since most of the transactions are making their deadlines. Therefore, transactions rarely have to wait until their deadlines in either case.

From this experiment, we conclude that the basic results about the superior performance of optimistic algorithms hold under the AED priority assignment policy also.

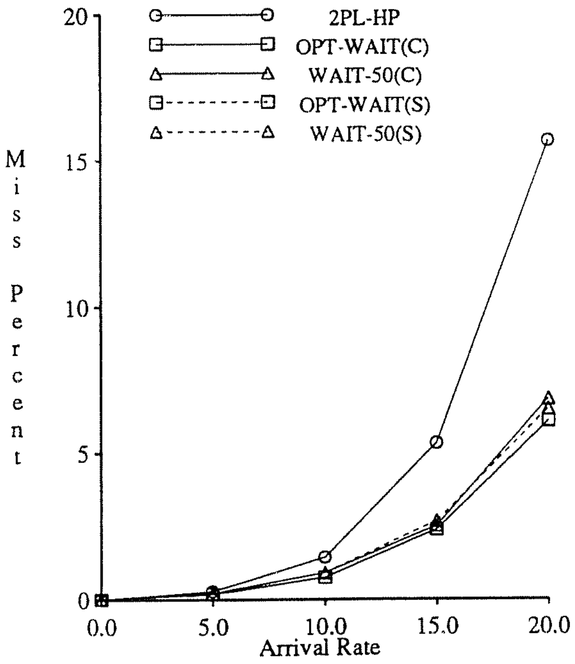

Figure 6.6(a): CC Algorithms (Normal)

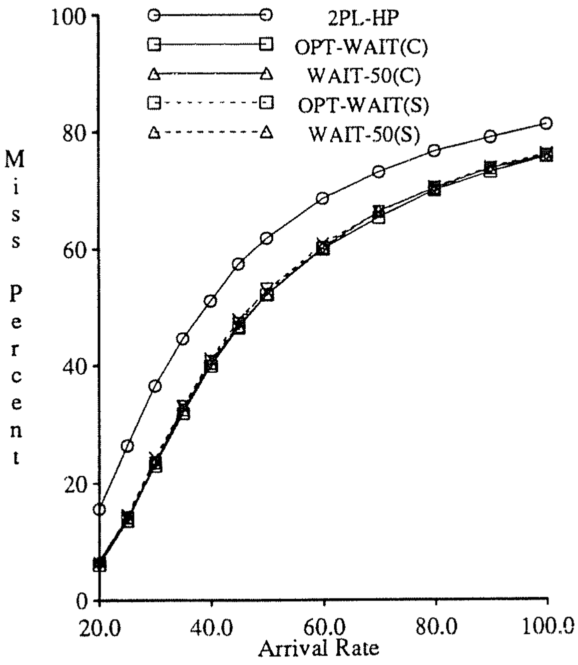

Figure 6.6(b): CC Algorithms (Heavy)

6.5. Conclusions

In this chapter, we addressed the issue of stabilizing the overload performance of Earliest Deadline in a firm-deadline RTDBS environment. We introduced the Adaptive Earliest Deadline (AED) priority policy and studied its performance relative to Earliest Deadline and other fixed priority mappings. Our experiments showed that for workloads generating only resource contention, the AED priority policy delivered the best performance over the entire loading range. At light loads, it behaved exactly like Earliest Deadline; at high loads its behavior was better than that of Random Priority, which was the best performer among the fixed priority mappings studied. The feedback control mechanism of AED was found to be accurate in estimating the number of transactions that could be successfully handled with an Earliest Deadline schedule. AED's policy of restricting the use of the Earliest Deadline approach to the *HIT* group delivered stabilized performance at high loads. For workloads that generated both data and resource contention, the AED policy again delivered the best overall performance. At low loads AED performed similarly to Earliest Deadline as before, while at high loads its behavior followed that of Random Priority. The AED policy was also shown to be robust to fluctuations in the transaction arrival pattern.

CHAPTER 7

VALUE AND DEADLINE

7.1. Introduction

Our studies thus far have considered applications that associate the same value with all transactions. Consequently, the performance objective has been to minimize the number of missed deadlines. We now move on to consider firm-deadline applications that assign different values to different transactions. When transactions have different values, the goal of the system is to maximize the sum of the values of transactions that commit by their deadlines. Minimizing the number of missed deadlines becomes a secondary concern in such systems. A fundamental problem in this situation is how to establish a priority ordering among transactions that are distinguished by both values and deadlines [Biya88]. In particular, the "correct" tradeoff to be established between transaction values and deadlines in generating the priority ordering is not obvious.

In this chapter, we address this issue of priority assignment in a value-based RTDBS. The goal is to establish a priority ordering that reflects the objective of maximizing the total realized value. In the absence of detailed knowledge of transaction resource requirements and data accesses, two basic principles, Earliest Deadline and Highest Value, can be used to guide the priority ordering. The Earliest Deadline principle is that transactions with closer deadlines should be given higher priority since delaying them might cause their deadlines to be missed and result in their value being lost. The Highest Value principle is that transactions with higher values should be given higher priority since it would be beneficial to make certain that their deadlines are met and thereby realize their high values.

When the transactions competing for service in the RTDBS have similar deadlines, it would appear, from an intuitive standpoint, that the ordering established by the Highest Value principle is the right ordering. This is because, if transactions are of similar urgency, completing the more valuable transactions first ensures that more value is realized. Conversely, when the competing transactions have similar values, it would seem that the Earliest Deadline principle provides the right priority ordering. This is because, if transactions have similar utility, completing the more urgent transactions first should result in more realized value.

When transactions differ in both their value *and* deadline characteristics, it is not obvious which principle should be followed. To provide a simple example, consider the scenario where a pair of transactions, *A* and *B*, arrive at time *t*=30 and compete for service. Assume that V_A=50 and D_A=80, while V_B=100 and D_B=110, as shown in Figure 7.1. In this case, following the Earliest Deadline principle would yield the priority ordering (*A*, *B*), while following the Highest Value principle would yield the priority ordering (*B*, *A*). It is not clear which of these

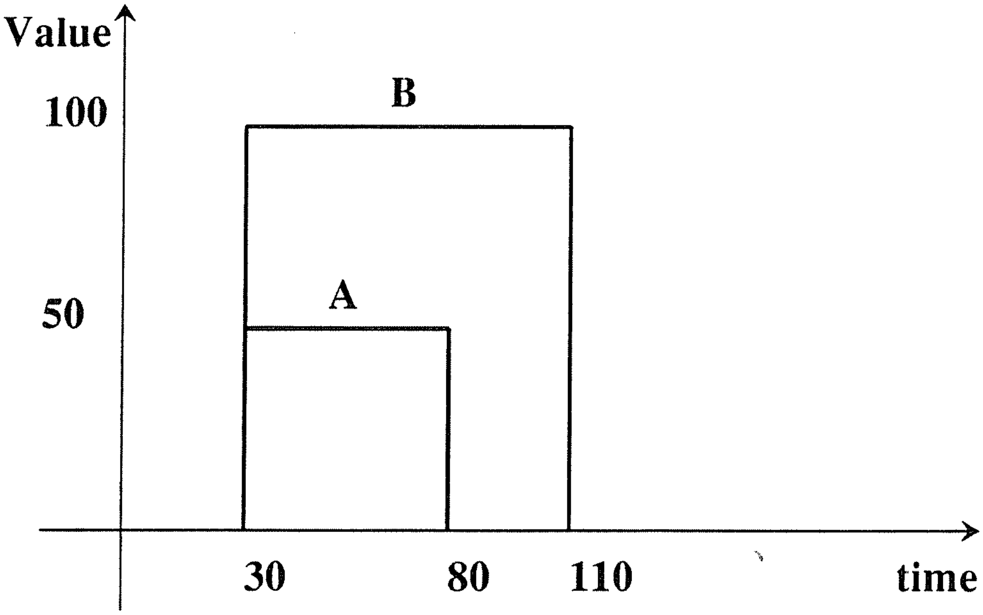

Figure 7.1: Priority Order Dilemma

priority orderings would realize more value. The dilemma here is to decide whether the value difference of 50 between A and B is more important or less important than their deadline difference of 30 time units. In a more general sense, deciding on a priority ordering when transactions differ in both value and deadline requires the value and deadline characteristics to be *weighted* in some fashion. A succinct statement of this requirement is that a priority mapping has to be established from the pair (D_T, V_T) to P_T, where P_T denotes the priority of transaction T. For example, a possible priority mapping, which gives equal weight to value and deadline, is $P_T = \frac{D_T}{V_T}$. In the example mentioned above, this mapping would result in the priority ordering (B, A).

The performance of several priority mappings that combine the Earliest Deadline and Highest Value principles with different *fixed* tradeoffs between value and deadline was examined in [Huan89]. This study was conducted on the RT-CARAT real-time database testbed. There are some aspects of this study that leave room for further investigation: First, the range of values that transactions could take on was limited and the value distribution was uniform. Second, a locking scheme was used as the underlying concurrency control mechanism in all the experiments. Finally, the RTDBS was configured as a closed queueing system with a fixed amount of resources.

In this chapter, we evaluate the impact of various priority mappings on the value realized by a firm-deadline real-time database system. These mappings are a representative subset of the mappings that were examined in [Huan89]. Our work differs from [Huan89] in that we consider a variety of transaction workloads that have different degrees of spread and skew in transaction value. In addition, we consider both locking and optimistic concurrency control algorithms. Finally, an open system with different levels of resource availability is modeled.

7.2. Priority Mappings

In this section, we describe the priority mappings that are evaluated in the experiments of this chapter. These mappings cover a range of fixed tradeoffs between value and deadline. The first two mappings implement extreme tradeoffs between value and deadline, while the others implement intermediate tradeoffs.

7.2.1. Earliest Deadline (ED)

The Earliest Deadline mapping follows the Earliest Deadline principle, so the transaction priority assignment is $P_T = D_T$. It represents an extreme tradeoff as the value of the transaction is not taken into consideration. As discussed in the previous chapter, using an Earliest Deadline schedule results in the fewest missed deadlines in lightly-loaded or moderately-loaded real-time systems.

7.2.2. Highest Value (HV)

The Highest Value mapping follows the Highest Value principle, so the transaction priority assignment is $P_T = \frac{1}{V_T}$. It represents the other extreme tradeoff as the deadline of the transaction is not taken into consideration. Note that this mapping does not distinguish between transactions that have the same value in terms of the priority assigned to them. Therefore, if all transaction values are the same, this mapping is equivalent to having *no* priority in the system.

7.2.3. Value-inflated Deadline (VD)

The Value-inflated Deadline mapping combines the Earliest Deadline and Highest Value principles by using the transaction priority assignment $P_T = \frac{D_T}{V_T}$. It gives *equal* weight to deadline and value. Moreover, within a group of transactions that have the same value, the priority ordering established by this mapping is identical to that of the ED mapping; within a group of transactions that have the same deadline, the priority ordering established is identical to that of the HV mapping.

7.2.4. Value-inflated Relative Deadline (VRD)

The Value-inflated Relative Deadline mapping is similar in flavor to VD, but it uses the *relative* deadline, instead of the absolute deadline, in combining the Earliest Deadline and Highest Value principles. The transaction priority assignment here is $P_T = \frac{D_T - A_T}{V_T}$. It gives equal weight to *relative deadline* and value. Note that if all transactions have their deadlines at a fixed distance from their arrival times (i.e. $D_T - A_T = constant$), this mapping produces a priority ordering identical to that established by the HV mapping. If transaction relative deadlines are linearly correlated to their execution times, the VRD mapping gives priority to transactions that can deliver the most value for the smallest amount of resource consumption. In this scenario, if transactions also all have the same value, the VRD mapping establishes a Shortest Job First priority ordering. For these cases, therefore, the VRD mapping behaves like a simple "greedy" algorithm that tries to maximize short-term benefits without taking transaction time constraints into account.

7.3. Concurrency Control Algorithms

Three different concurrency control algorithms are evaluated in this chapter. The selected algorithms are 2PL-HP, OPT-BC and OPT-WAIT (refer to Chapters 4 and 5 for details of these algorithms). In Chapter 4, it was shown that OPT-BC, in spite of being priority-indifferent, provided better performance than 2PL-HP in a firm deadline environment. That chapter assumed that all transactions have the same value. One of the goals of this chapter is to determine whether those results are still applicable when transactions have different values.

The OPT-WAIT variant considered in the experiments of this chapter is the OPT-WAIT(S) algorithm of Chapter 6. This algorithm implements a policy where a transaction that is priority-waiting at its deadline is always aborted and discarded, thus ensuring that high-priority transactions are never restarted by low priority transactions. The reason for choosing the OPT-WAIT(S) option is explained in the following experiment section.

In Chapter 5, the dynamic WAIT-50 algorithm was shown to provide better overall performance than either OPT-BC or OPT-WAIT. There, the 50 percent rule in WAIT-50 was developed on the basis that meeting the deadline of any transaction was equally useful to the application. When transactions have different values, however, this is not the case and the 50 percent rule is inappropriate. Also, the dilemma described above for OPT-WAIT (with respect to priority mappings that do not assign the highest priority to a transaction at its deadline), is also faced by WAIT-50. Finally, our focus here is on the priority assignment component and less so on the concurrency control component. Therefore, for all these reasons, we do not address the issue of developing an appropriate dynamic optimistic algorithm for the value-based framework here. It is, however, an open research problem that we hope to address in the future.

7.4. Experiments and Results

In this section, we present the results of our experiments comparing the performance of the various priority mappings. These experiments evaluate the impact of data contention, resource contention, and distribution of transaction values. An important point to note is that, unlike the previous studies, the *Loss Percent* and *Miss Percent* metrics are not equivalent here since transactions have different values. Therefore, both of these metrics are discussed in the experiments of this chapter.

To serve as a basis for comparison, apart from the candidate priority mappings described in Section 7.2, the performance of the No Priority and Random Priority mappings described in Chapter 6 are also evaluated in our experiments here.

7.4.1. Experiment 1: Resource Contention (RC)

Our first set of experiments investigated the performance of the priority mappings when resource contention is the sole performance limiting factor. As usual, we began our experiments by first developing a baseline experiment. Further experiments were constructed around the baseline experiment by varying a few parameters at a time. The settings of the workload parameters and the resource parameters for the baseline experiment are listed in Table 7.1. The *WriteProb* parameter, which gives the probability that a page that is read will also be updated, is set to 0.0 to eliminate data contention. Therefore, no concurrency control is necessary for this set of experiments. There is a single transaction class, and transaction values range between 50.0 and 150.0. As in the previous chapter, deadline formula DF3 is used to assign transaction deadlines in the experiments described here. With DF3, a general workload where transactions have a range of slack ratios is constructed. The slack factor parameter settings are the same as those of the pure resource contention experiment in the previous chapter. These value and deadline parameter settings ensure that both characteristics play a role in determining overall system performance.

7.4.1.1. Baseline Experiment

For the baseline experiment, Figures 7.2(a) and 7.2(b) show the Loss Percent results under normal load and heavy load conditions, respectively. Figure 7.2(c) shows the corresponding Miss Percent results. (Note that the curves for HV and VD are identical in these

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.25	*NumCPUs*	8
WriteProb	0.0	*NumDisks*	16
DeadlineFormula	DF3	*PageCPU*	10ms
LSF	1.33	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		
NumClasses	1		
ProbClass[i]	1.0		
OfferedValue[i]	1.0		
MeanValue[i]	100.0		
SprdValue[i]	0.5		

Table 7.1: Baseline Parameter Settings

figures). From this set of graphs, it is clear that at low loads, the ED (Earliest Deadline) mapping realizes the most value (smallest Loss Percent). This might be considered surprising since ED is a value-indifferent mapping, while some of the other mappings are value-cognizant. The reason for ED's good performance can be understood, however, by examining the Miss Percent characteristics (Figure 7.2(c)) at low loads. Since ED misses the deadlines of very few (if any) transactions, it delivers the most value. The value-cognizant mappings, HV (Highest Value), and to a lesser extent, VRD (Value-inflated Relative Deadline), focus their effort on completing the high-value transactions. In the process, they prevent some lower value transactions from making their deadlines, even though most of the deadlines could have been met (as demonstrated by ED), thereby losing more of the offered value. As the system load is increased, however, the performance of ED steeply degrades and becomes close to that of NP (No Priority) at high loads. As discussed in Chapter 6, this is because Earliest Deadline assigns the highest priority, under heavy loads, to transactions that are close to missing their deadlines, thus delaying other transactions that can still meet their deadlines.

Focusing next on the HV (Highest Value) mapping, we observe that it performs worse than ED at low loads but improves its performance as the load increases. In fact, at high loads, it outperforms all the other mappings (except for VD, which is discussed next). This is because following the Highest Value principle is a good idea at high loads, where the system has sufficient resources to handle only a fraction of the transactions in the system. In such a situation, the transactions that should be run are clearly those that can deliver high value. If we look at the Miss Percent characteristics (Figure 7.2(c)), we observe that HV and RP (Random Priority) behave identically with respect to this metric. The reason for this behavior is that transaction values in the workload are independent of other transaction characteristics and all transaction values are distinct. For this case, a HV priority ordering is no different from an RP priority ordering in terms of the ability of the RTDBS to make transaction deadlines. Note that if there were groups of transactions that had the *same* value, then this would not be the case,

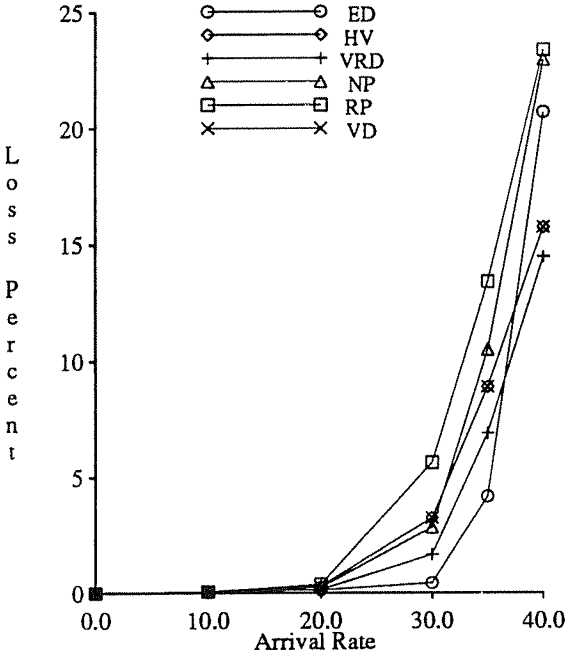

Figure 7.2(a): RC Baseline (Normal)

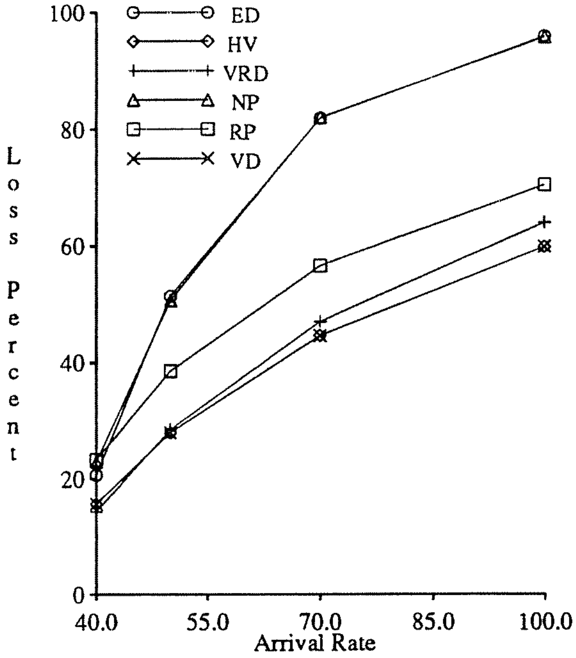

Figure 7.2(b): RC Baseline (Heavy)

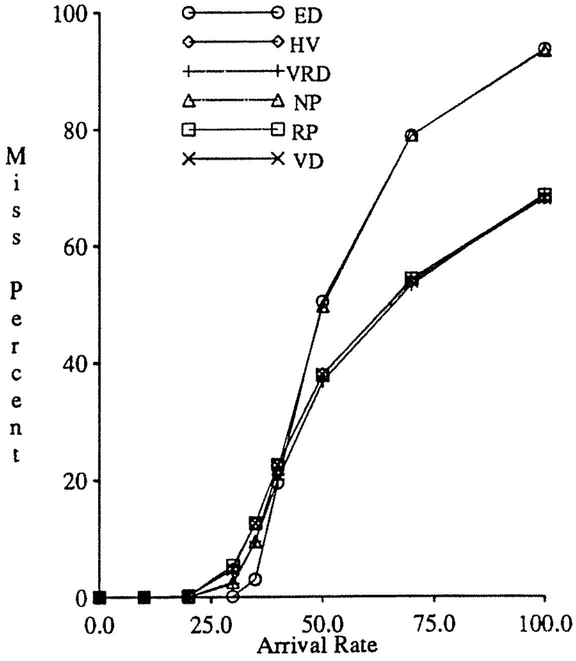

Figure 7.2(c): Miss Percent (RC Baseline)

as same-value transactions introduce NP-type behavior into the performance of HV.

Moving on to the VD (Value-inflated Deadline) mapping, we observe that although this mapping appears to combine the Earliest Deadline and Highest Value principles in its priority assignments, it performs identically to the HV mapping. This is not a coincidence, but is, in fact, always true: As time progresses, the D_T term in $\frac{D_T}{V_T}$ becomes large enough that it is approximately the same for all transactions. Therefore, once the clock time is sufficiently large, VD behaves exactly like HV. For this reason, we will not consider the VD mapping any further in this study. The more general lesson that can be learned from the behavior of VD is that priority computations that combine values and absolute deadlines should be designed with care to ensure that the above problem is not encountered. In [Huan89], it was observed that a priority assignment of $P_T = V_T\ (w_1(t - A_T) - w_2 {*} D_T)$, where w_1 and w_2 are weighting factors, displayed little change in performance with different settings for the weights. The probable reason is that with any non-zero value for w_2, the absolute deadline (D_T) term in the formula dominates the other term once the clock time is sufficiently large, and thus the priority assignment degenerates to an HV mapping. Therefore, the actual weights should not, in fact, be expected to impact the long-term performance of this mapping.

Turning our attention to the VRD (Value-inflated Relative Deadline) mapping, we observe that its performance is intermediate to that of ED and HV. At low loads it is slightly worse than ED, while at high loads it is slightly worse than HV. In a sense, therefore, it delivers the best overall performance. Note that while VRD, like VD, takes both deadlines and values into account, it does not behave like HV. The reason is that the mapping uses the *relative* deadline, rather than the absolute deadline, to compute transaction priorities. This makes the VRD mapping both value and deadline cognizant for this workload. The reason that the VRD mapping does better than HV at low loads is that it has a partial Earliest Deadline effect in that jobs with smaller relative deadlines are given priority over jobs with larger relative deadlines. Among sets of similar-valued jobs that arrive at around the same time, the priority ordering is therefore approximately Earliest Deadline. Due to this effect, fewer deadlines are missed by VRD at low loads when compared to HV (Figure 7.2(c)). Conversely, at high loads, when a large fraction of deadlines are missed, the fact that VRD takes deadline into account works against it since a high-value transaction may not be completed due to a large relative deadline.

7.4.1.2. Increased Value Spread

Our next experiment examined the effect of increasing the spread in transaction values. For this experiment, the *SprdValue* parameter was increased from the baseline value of 50 percent up to 99 percent, keeping the other parameters the same as in the baseline experiment. This means that transaction values ranged uniformly between 1.0 and 199.0. The Loss Percent results for this experiment are shown in Figures 7.3(a) and 7.3(b). We first note that the performance of the ED, RP and NP mappings remains the same as in the baseline experiment. This is because these mappings are value-indifferent, and therefore changes in the value distri-

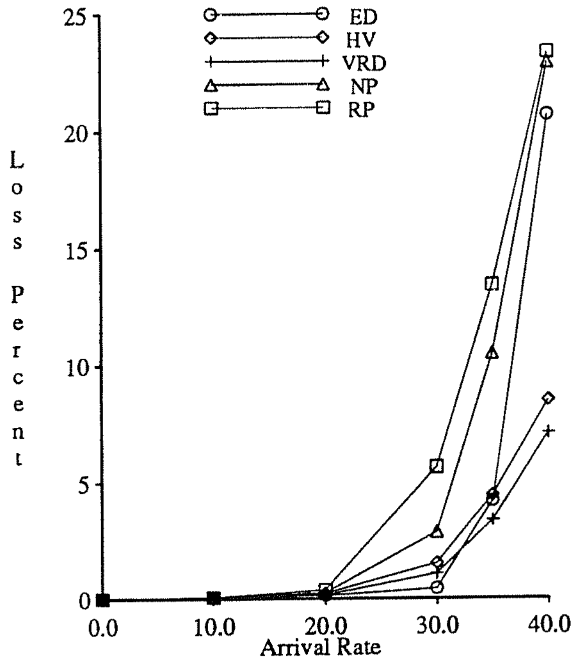

Figure 7.3(a): Increased Spread (Normal)

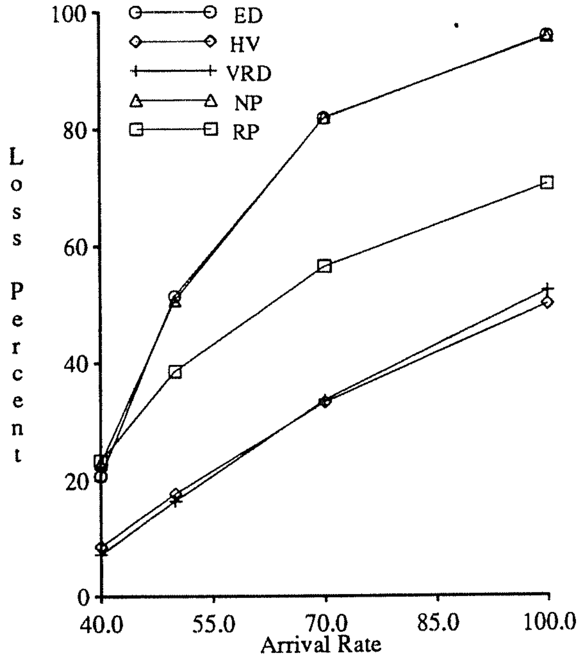

Figure 7.3(b): Increased Spread (Heavy)

bution do not affect their performance (as long as the mean value remains the same). However, the value-cognizant mappings, HV and VRD, improve their performance considerably. This is because these mappings concentrate on the more valuable transactions, and increasing the value spread implies that, on the average, greater value is obtained for each high-value transaction that is completed. Also, the low-value transactions that are missed have a lesser effect on the realized value since their values are smaller due to the increased spread. Note that the Miss Percent characteristic of HV is the same as in the baseline experiment (Figure 3c) since the workload assigns values to transactions independently of their other characteristics.

7.4.1.3. Decreased Value Spread

The next experiment examined the effect of decreasing, rather than increasing, the spread in transaction values. For this experiment, the *SprdValue* parameter was set to 0 percent, keeping the other parameters the same as those of the baseline experiment. This means that all transaction values had the same value of 100.0. The Loss Percent results for this experiment are shown in Figures 7.4(a) and 7.4(b). The value-cognizant mappings, HV and VRD, perform worse here when compared to the baseline experiment. The HV mapping, in fact, behaves just like the NP mapping. The reason for HV behaving like NP is that when all values are the same, HV gives every transaction the same priority. While all transactions having the same value is an extreme case, similar problems will arise when the workload consists of multiple transaction classes where all transactions within a class have the same value. The VRD mapping, unlike HV, does not behave like NP; this is because the relative deadline component of its priority mapping ensures that there is a priority ordering among the transactions. Also, at high loads, VRD behaves similar to RP rather than ED. This implies that VRD is more a value-oriented mapping than a deadline-oriented mapping at high loads since the relative deadline component has only a randomizing effect when all values are the same.

As in Chapter 6, we observe here that the RP (Random Priority) mapping performs quite well at high loads. This means that if a *random "noise"* element is added to priority values,

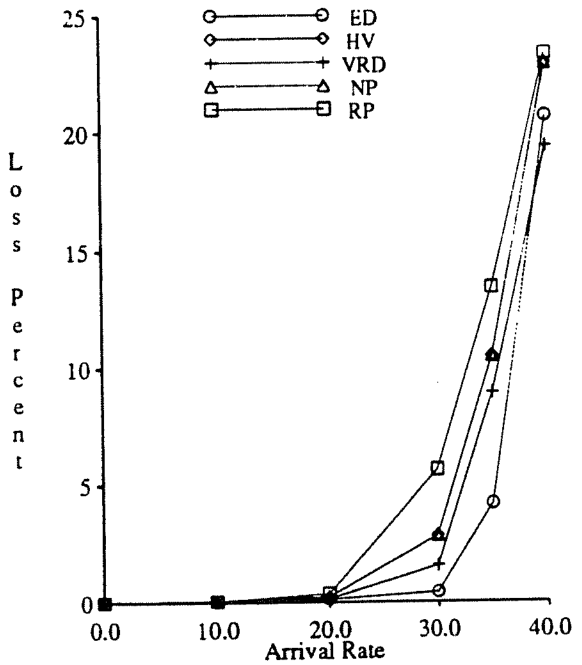

Figure 7.4(a): Decreased Spread (Normal)

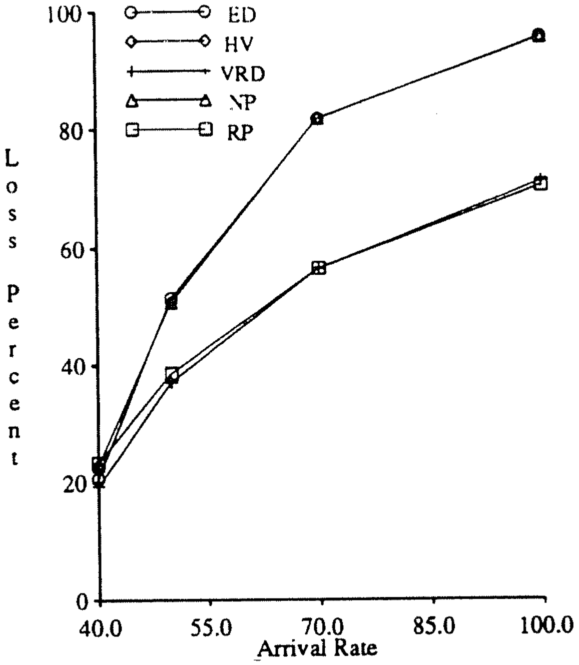

Figure 7.4(b): Decreased Spread (Heavy)

stability in high-load performance can be obtained even when most or all of the priority values would otherwise be the same. For example, if even an infinitesimally small noise is added to the transaction priorities generated by the HV mapping for this experiment, the heavy load performance would be like that of RP rather than that of NP. This is because the addition of the noise would cause a priority ordering to *exist* where there was none originally. Note that the noise should be random, and *not* based on transaction characteristics. If transaction deadlines were used to generate the noise, for example, the performance would be like that of ED, and not of RP, at high loads. It should also be noted that the high-load stability obtained by the addition of noise is gained at some cost in normal load performance, as RP performs worse than NP under normal loads.

Summarizing the results of the above resource contention experiments, we can draw the following conclusions for the uniform-value workloads examined in this section: First, at low loads, when the Miss Percent is low, the Earliest Deadline priority ordering is the right choice, while at high loads, when the Miss Percent is high, the priority ordering given by the Highest

Value principle realizes the most value. Second, the degree of spread in transaction values has a significant effect on the performance of the value-cognizant mappings. In particular, their performance improves with an increased spread in values. Third, the use of absolute deadlines in priority assignments should be handled with care. Finally, priority mappings should have a built-in noise factor to guard against the possibility of transactions having identical priorities, as otherwise transactions can hinder the progress of each other and thus degrade performance at high loads.

7.4.1.4. Skewed Value Distribution

The next experiment examined the effect of having a skew in the transaction value distribution. For this experiment, the parameters are set as shown in Table 7.2. They construct a two-class workload where 10 percent of the transactions deliver 90 percent of the offered value. The values of the transactions from the first class vary between 450.0 and 1350.0, while the values of the second class vary between 5.5 and 16.0. The Loss Percent results for this experiment are shown in Figure 7.5. As in the previous experiments, the performance of the ED, RP and NP mappings remains the same as in the baseline experiment since these mappings are value-indifferent. The figure also shows that the performance of the value-

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	8
WriteProb	0.0	*NumDisks*	16
DeadlineFormula	DF3	*PageCPU*	10ms
LSF	1.33	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		
NumClasses	2		
ProbClass[i]	0.1,0.9		
OfferedValue[i]	0.9,0.1		
MeanValue[i]	900.0,11.1		
SprdValue[i]	0.5,0.5		

Table 7.2: Skewed Value Settings

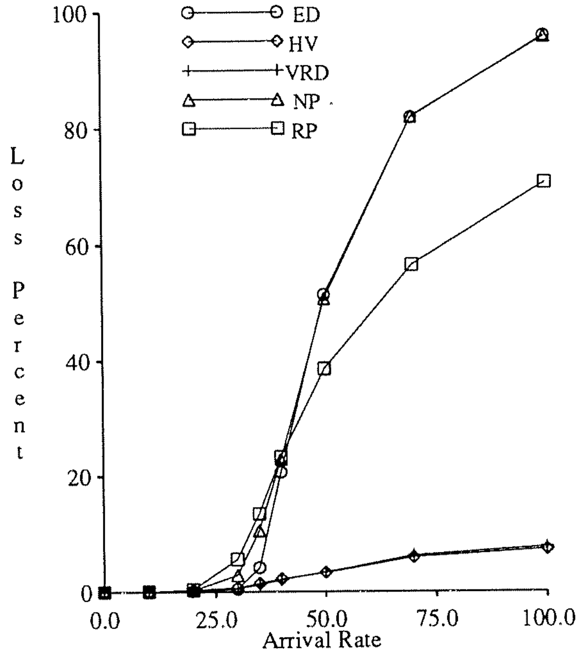

Figure 7.5: RC Value Skew

cognizant mappings, HV (Highest Value) and VRD (Value-inflated Relative Deadline), improves greatly as compared to the baseline experiment and they are now much superior to the value-indifferent mappings. Note that even at low loads here, they perform almost as well as the Earliest Deadline mapping. The value-cognizant mappings, by making certain that all of the relatively few high-value transactions make their deadlines, ensure that they realize at least 90 percent of the offered value. In addition, at low loads, the value of the missed transactions constitutes a very small fraction of the total value, and therefore the performance impact of having a higher number of missed deadlines than ED is negligible. Note also that the performance of the VRD mapping here is almost identical to that of the HV mapping. This is because when the spread in value is much larger than the spread in relative deadline, the V_T component of the VRD mapping dominates the $(D_T - A_T)$ component in determining relative transaction priorities. Therefore, for workloads with these features, the VRD mapping is only marginally deadline-cognizant and therefore generates a priority ordering very similar to that of the Highest Value (HV) mapping.

We conclude from this experiment that skew in transaction values causes the value-cognizant mappings to perform much better. For workloads that have a considerable spread in transaction values, the priority ordering established by the Highest Value principle ensures good performance through the entire loading range. These results also demonstrate the significant impact of value distributions on the relative performance of the priority mappings.

7.4.2. Experiment 2: Data Contention (DC)

The second set of experiments investigated the performance of the priority mappings when data contention is the sole performance degradation factor. As before, we began our experiments by first developing a baseline experiment around which we then constructed further experiments by varying a few parameters at a time. The settings of the workload parameters for this baseline experiment are identical to those for the Resource Contention case (listed in Table 7.1) except that the *WriteProb* parameter is set to 0.25 instead of 0.0. The settings of the resource parameters are made "infinite", and therefore the performance differences observed between the mappings are solely due to data contention. For graph clarity, we do not consider the RP (Random Priority) and NP (No Priority) mappings in the following sections.

7.4.2.1. Baseline Experiment

For the baseline experiment, Figures 7.6(a) and 7.6(b) show the Loss Percent results for the various priority mappings under normal loads and heavy loads, respectively. Figure 7.6(c) shows the corresponding Miss Percent behavior. The results shown were separately obtained with the 2PL-HP, OPT-BC and OPT-WAIT concurrency control algorithms. Focusing our attention on the performance of 2PL-HP (the solid lines), we observe that, *qualitatively*, the mappings exhibit the same behavior as in the case of the resource contention baseline experiment (Figures 7.2(a), 7.2(b)). The ED (Earliest Deadline) mapping performs the best at low loads, while the HV (Highest Value) mapping outperforms all of the other mappings at high loads. As before, ED performs well at low loads since it misses far fewer deadlines. Moreover, data contention (unlike resource contention) is not *work-conserving* because already performed work has to be redone after a transaction restart; therefore, ensuring that the most urgent transactions are given the highest priority is even more beneficial at low loads here. At high loads, following the Highest Value principle is again the right approach, as the data contention level is high enough that only a fraction of the transactions in the system are able to complete before their deadlines; in such a situation, the transactions that should be given priority are those that can deliver high values.

Turning our attention to OPT-BC (the dashed line), here all of the priority mappings behave exactly the same. This is because OPT-BC is a priority-indifferent algorithm and there is no resource contention; therefore, transaction priority does *not* play a role in determining system performance. In spite of this priority indifference, however, OPT-BC performs better than 2PL-HP for most of the loading range, especially at higher loads. The reason for this is obvious when we compare the Miss Percent characteristics, where we observe that OPT-BC misses far fewer deadlines than 2PL-HP (Figure 7.6(c)). The primary reason for the lower number of misses is that discussed in Chapter 4: The optimistic approach, due to its validation stage conflict resolution, ensures that eventually discarded transactions do not cause the restart of other transactions. The locking approach, on the other hand, allows these soon-to-be-discarded transactions to cause other transactions to be either blocked or restarted due to lock conflicts, thereby increasing the number of late transactions.

Moving on to OPT-WAIT (the dotted lines), we observe that it performs worse than OPT-BC for all of the priority mappings except ED at low loads. The reason for OPT-WAIT doing better than OPT-BC at low loads for the ED priority mapping is that priority waiting is a good idea here since the more urgent transactions are not restarted by less urgent transactions. At high loads, however, the priority wait mechanism causes performance degradation due to an increase in system population (many waiters), which causes a steep increase in the number of conflicts. This behavior of OPT-WAIT was observed and discussed in detail in Chapter 5. Although the experiments of Chapter 5 did not consider transaction values, the explanations

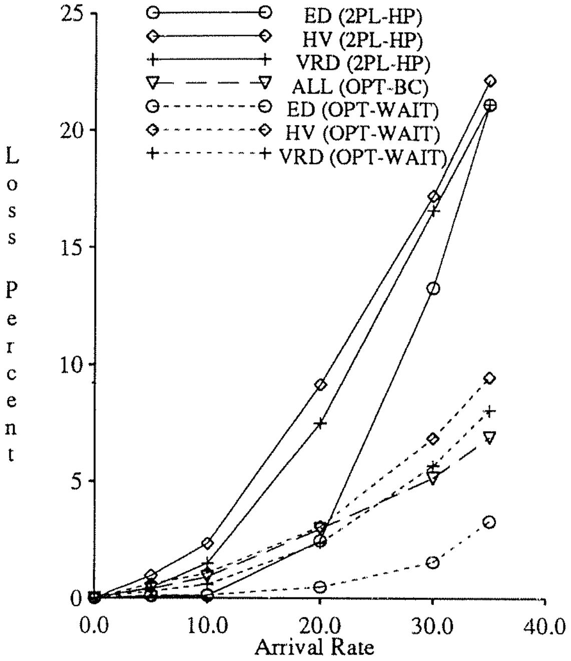

Figure 7.6(a): DC Baseline (Normal)

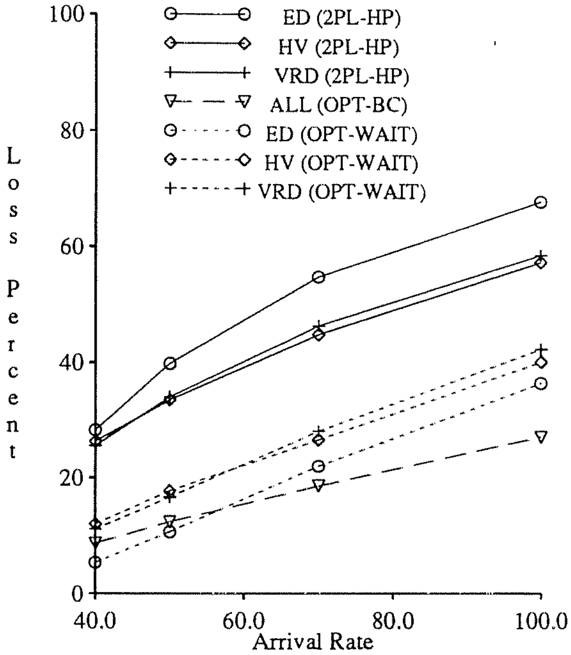

Figure 7.6(b): DC Baseline (Heavy)

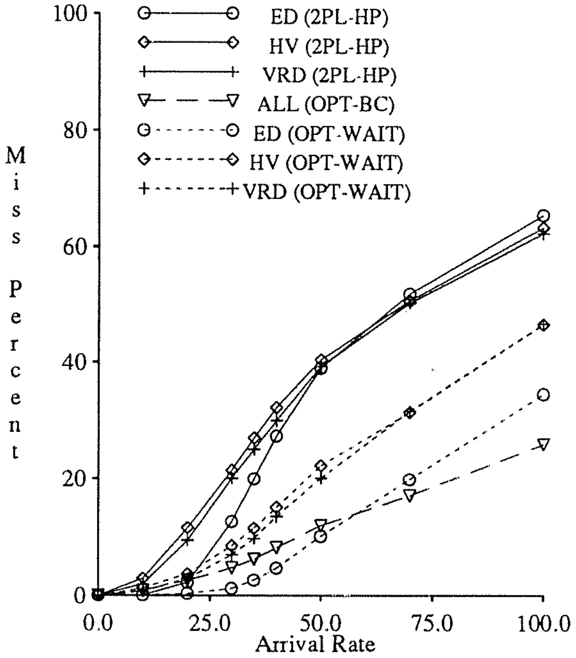

Figure 7.6(c): Miss Percent (DC Baseline)

carry over here since ED is a value-indifferent priority mapping.

The reason that OPT-WAIT performs worse than OPT-BC over most of the loading range for the mappings other than ED is the following: With the ED priority mapping, a waiting transaction never has to wait beyond its deadline, as it will have the highest priority in the system when it reaches its deadline. For the other mappings, however, this is not necessarily the case. If we consider HV, for example, it is clear that the waiting process could extend beyond the waiter's deadline since some or all of the higher-value conflicting transactions may not have completed by the waiter's deadline. In such a case the waiter is aborted and discarded, and the waiter's value is therefore lost. This wouldn't be so bad if the higher priority transactions then made their deadlines and the system realized their values. There is no guarantee, however, that this will actually happen. We could, instead, have many *wasted sacrifices* – cases where a transaction is discarded on behalf of another transaction that later does not complete. Such sacrifices are useless and cause performance degradation as discussed in Chapter 5. It is due to these wasted sacrifices that we also see a seemingly odd performance ordering of the various mappings with OPT-WAIT: ED performs better than HV and VRD. It should be noted, however, that the performance of OPT-WAIT is still better than that of 2PL-HP for all of the mappings throughout the entire loading range.

7.4.2.2. Skewed Value Distribution

The next experiment examined the effect of having a skew in the transaction value distribution. For this experiment, the workload parameters are the same as for experiment 7.4.1.4 (listed in Table 7.2, except that the *WriteProb* parameter is set to 0.25. The Loss Percent results of the experiment are shown in Figure 7.7 for the 2PL-HP, OPT-BC and OPT-WAIT concurrency control algorithms. Focusing our attention on 2PL-HP (solid lines), we observe that the performance of the value-cognizant mappings improves greatly and that they are now far superior to the ED mapping, as in the pure resource contention case. The reason for this improvement is the following: 2PL-HP ensures that the highest priority transactions are

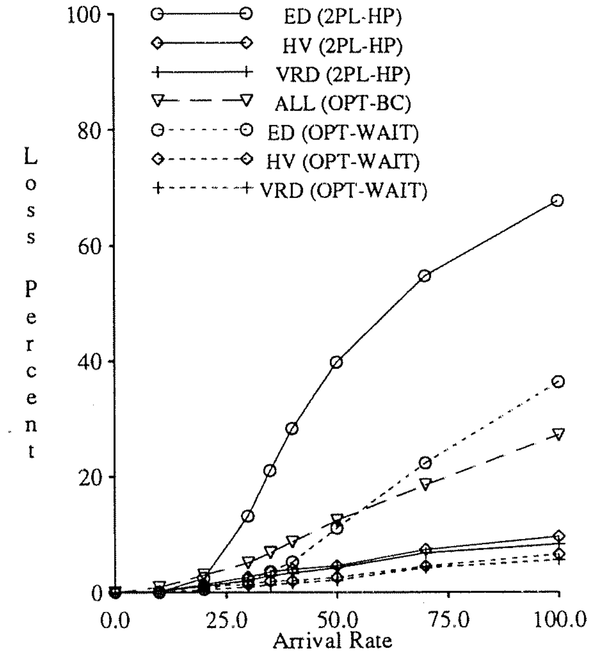

Figure 7.7: DC Value Skew

virtually guaranteed to make it to their deadlines. The value-cognizant mappings, in combination with 2PL-HP, realize a high value since they assign the highest priorities to the high-valued transactions. Successfully making the deadlines of the few high-value transactions is by itself sufficient to realize at least 90 percent of the workload's offered value.

Moving on to OPT-BC (dashed line), we note that its performance remains the same as in the baseline data contention experiment (Experiment 7.4.2.1). This is because OPT-BC does not take into account transaction values, and therefore changes in the transaction value distribution do not affect its performance. From Figure 7.7, it is clear that here the performance of the value-cognizant mappings under 2PL-HP is superior to their performance under the OPT-BC algorithm. Note that this is in spite of 2PL-HP having a much higher Miss Percent than OPT-BC. Since 2PL-HP concentrates on the high-value transactions, the value it derives from these transactions more than compensates for the value lost due to missing the deadlines of a large number of low-value transactions. OPT-BC, on the other hand, treats all transactions equally, which can cause high-value transactions to be restarted (and therefore miss their deadline) due to the commits of low-value transactions.

Turning our attention to OPT-WAIT (dotted lines) and comparing these results with the 2PL-HP results, it can be observed that all of the mappings perform better for OPT-WAIT than for 2PL-HP, including the value-cognizant mappings. The reason for this is that, since OPT-WAIT is priority cognizant and is willing to sacrifice low priority transactions for high priority transactions, the high-value transactions are guaranteed to complete before their deadlines. This is similar to the behavior of 2PL-HP. In addition, OPT-WAIT gains some extra value over 2PL-HP due to missing the deadlines of a smaller number of low-value transactions. To sum up, as compared to 2PL-HP, OPT-WAIT meets the deadlines of all of the high-value transactions and misses fewer low-value transactions. Note that the "commit at deadline" version of OPT-WAIT (i.e. the OPT-WAIT(C) algorithm of Chapter 6) could be expected to perform poorly here since it allows low-value waiters to restart high-value conflicting transactions. This expectation was confirmed experimentally. It is for this reason that we chose the OPT-WAIT(S) algorithm for evaluation in the experiments of this chapter. Another point to note is that the various mappings with OPT-WAIT no longer have the "odd" performance ordering seen for the uniform, limited spread value distribution of the previous experiment: HV and VRD now perform better than ED.

In Chapter 4, it was shown that optimistic algorithms outperform locking algorithms when transactions all have the same value. The results from our experiments here demonstrate that optimistic algorithms can also perform better than locking algorithms when the real-time environment incorporates the notion of value and the priority mapping is value-cognizant.

7.4.3. Other Experiments

In addition to the experiments presented here, we conducted several experiments where both resource contention *and* data contention contribute towards system performance degradation (refer [Hari91a] for details). The qualitative results were the same as those obtained for resource contention or data contention alone. The ED (Earliest Deadline) mapping performs the best at normal loads, while the HV (Highest Value) mapping provides the best performance at high loads. Also, the performance of the value-cognizant mappings improves with the transaction value spread or with the transaction value skew.

7.5. Conclusions

In this chapter, we addressed the issue of how to assign priorities to transactions in a firm-deadline RTDBS when the workload consists of transactions that are characterized by both values and deadlines. We studied the performance of several priority mappings that establish different fixed tradeoffs between values and deadlines. Our experiments showed that for workloads with a limited, uniform spread in the transaction values, the Earliest Deadline (ED) mapping provided the best performance among the fixed-tradeoff mappings under light loads. Although ED is a value-indifferent mapping, the database system had sufficient resources at low loads to meet most transaction deadlines; consequently, prioritizing transactions according to their urgency led to the fewest missed deadlines and generated the most value. Under heavy loads, however, it was the Highest Value (HV) mapping that delivered the best performance in spite of being deadline-indifferent. A large fraction of the deadlines were missed under all the mappings at high loads, and the fact that HV prioritizes transactions by value alone ensured that high-value transactions rarely missed their deadlines. The Value-inflated Deadline (VD) mapping, which combines both values and deadlines by weighting them equally, was found (perhaps surprisingly) to behave identically to HV. Finally, the Value-inflated Relative Deadline (VRD) mapping, which equally weights *relative* deadlines and values, provided the best overall performance among the fixed-tradeoff mappings; it was almost as good as ED at low loads, and was close to HV at high loads.

For workloads that had a large spread or a pronounced skew in the distribution of transaction values, the HV mapping was found to deliver the best performance throughout almost the entire loading range. Although HV missed more deadlines than the ED mapping at low loads, the value gained by HV's ability to complete virtually all of the high-value transactions more than compensated for its missing more deadlines of low-value transactions. In addition to these results regarding the relative performance of the fixed-tradeoff mappings, our experiments also showed that they are susceptible to performance breakdown based on workload characteristics. For example, workload characteristics that lead a priority mapping to assign the same priority to a number of high-value transactions were shown to be quite detrimental to performance at high loads; adding a random noise component to the priority mappings alleviated this problem by inducing a total priority ordering among the transactions.

Experiments were also conducted to explore the impact of data contention on the performance of the various priority mappings. These experiments were conducted with several concurrency control algorithms in order to evaluate their performance and to study their impact, if any, on the priority mapping results. The same qualitative behavior that was observed in the presence of resource contention was obtained in the pure data contention experiments; this was also the case when data and resource contention were combined. In Chapter 4, we showed that optimistic concurrency control outperforms locking in a firm real-time environment. The experiments of Chapter 4 employed an Earliest Deadline priority mapping and assumed that all transactions have the same value. The conclusion of the present chapter's experiments is that these results generally carry over to the value-based RTDBS domain for all of the priority mappings that we have considered.

CHAPTER 8

HIERARCHICAL EARLIEST DEADLINE

8.1. Introduction

In the preceding chapter, we demonstrated that, from a performance perspective, there is no single fixed tradeoff between transaction value and deadline that is appropriate under all circumstances. Rather, the choice of the "right" tradeoff depends on the workload and system operating conditions. Therefore, a mechanism is required for *varying* the tradeoff to match the operating environment in order to achieve good performance in a stable fashion.

In this chapter, we present **Hierarchical Earliest Deadline (HED)**, a new priority assignment algorithm that integrates the value and deadline characteristics of transactions. The HED algorithm is a value-based extension of the AED algorithm described in Chapter 6. It adaptively varies the tradeoff between value and deadline to maximize the value realized by the system. When all transaction values are the same, the HED algorithm reduces to the AED algorithm.

In the experiments of the previous chapter, one of two fixed-tradeoff mappings – either Earliest Deadline (ED) or Highest Value (HV), which implement extreme tradeoffs – almost always provided the best performance. We evaluate the performance of the HED algorithm with respect to these two mappings in this chapter.

8.2. Hierarchical Earliest Deadline (HED)

The Hierarchical Earliest Deadline algorithm extends the Adaptive Earliest Deadline algorithm by accounting for transactions having different values. Informally, the HED algorithm groups transactions, based on their values, into a hierarchy of prioritized buckets. It then uses an AED-like algorithm within each bucket to determine the relative priority of transactions belonging to the bucket. The details of the HED algorithm are described below, after which the rationale behind the construction of the algorithm is discussed.

8.2.1. Bucket Assignment

The HED algorithm functions in the following manner: The *priority mapper* unit maintains a *value-based* dynamic list of buckets, as shown in Figure 8.1. Every transaction, upon arrival, is assigned based on its value to a particular bucket in this list. Each bucket in the list has an associated *MinValue* and *MaxValue* attribute – these attributes bound the values that transactions assigned to the bucket may have. Each bucket also has an identifier, and bucket identifiers in the list are in monotonically increasing order. There are two special buckets, *Top* and *Bottom*, which are always at the head and tail of the list, respectively. The *MinValue* and *MaxValue* attributes of *Top* are set to ∞, while the *MinValue* and *MaxValue* attributes of *Bottom* are set to zero. Since we assume that all transaction values are finite and positive, no transactions are ever assigned to these buckets, and their function is merely to serve as permanent list boundaries. The identifiers of the *Top* and *Bottom* buckets are preset to 0 and MAXINT, respectively.

When a new transaction, T_{new}, arrives in the system, it is assigned to the bucket closest to *Top* that satisfies the constraint $MinValue \leq Value_{new} \leq MaxValue$. If no such bucket exists, a new bucket is inserted in the list between the bucket closest to *Top* that satisfies $MinValue < Value_{new}$ and its predecessor, and the transaction is assigned to this bucket. A newly created bucket is assigned its identifier by halving the sum of the identifiers of its predecessor and successor buckets. For example, a bucket inserted between buckets with identifiers 256 and 512 will have 384 as its identifier. When a transaction leaves the system, it is removed from its assigned bucket. A bucket that becomes empty of transactions is immediately deleted from the bucket list.

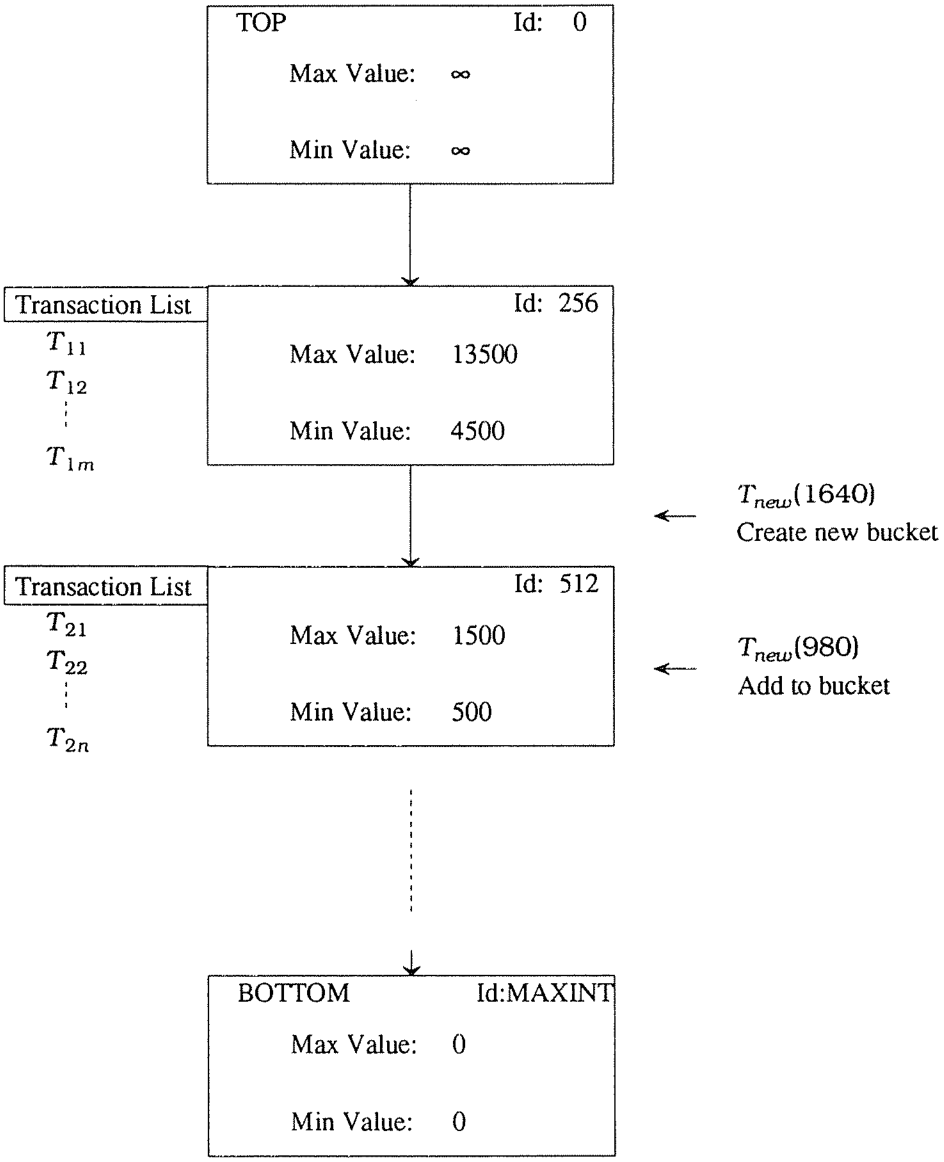

Figure 8.1: HED Bucket Hierarchy

The *MinValue* and *MaxValue* attributes of a bucket are set as follows: Each bucket maintains an *AvgValue* attribute that monitors the average value of the set of transactions that are *currently* assigned to the bucket. The *MinValue* and *MaxValue* attributes of the bucket are then computed as (*AvgValue/SpreadFactor*) and (*AvgValue*SpreadFactor*), respectively, where *SpreadFactor* is a parameter of the HED algorithm. The *SpreadFactor* parameter controls the maximum spread of values allowed within a bucket. Whenever a transaction enters or leaves the system, the associated bucket updates its *AvgValue*, *MinValue* and *MaxValue* attributes.

8.2.2. Group Assignment

In similar fashion to the AED algorithm, each bucket has transactions divided into *HIT* and *MISS* groups, with the *HIT* group size controlled by a *HITcapacity* variable. After a new transaction has been assigned to a bucket, its group assignment within the bucket is as follows: The transaction is assigned a unique[1] key, I_T, with the key being a randomly chosen integer. It is then inserted into a *value-ordered* list of transactions belonging to the bucket, with transactions that have identical values being ordered by their keys. The position of the new transaction in the list, pos_T, is noted. If pos_T is less than the *HITcapacity* of the bucket, the new transaction is assigned to the *HIT* group in the bucket; otherwise, it is assigned to the *MISS* group. The *HITcapacity* computation in each bucket is implemented with a separate feedback process; each feedback process is identical to that described in Chapter 6 for the AED algorithm.

8.2.3. Priority Assignment

After its bucket and group assignment, a new transaction is assigned its priority using the following formula:

$$P_T = \begin{cases} (B_T,\ 0,\ D_T,\ I_T) & \text{if } Group = HIT \\ (B_T,\ 1,\ \dfrac{1}{V_T},\ I_T) & \text{if } Group = MISS \end{cases}$$

where B_T is the identifier of the transaction's bucket.

[1] As in the AED algorithm, transaction keys are sampled uniformly over the set of integers. In the unlikely event that a new key matches that of an existing transaction, the key is re-sampled until a unique key is obtained.

The above priority assignment results in transactions of bucket i having higher priority than all transactions of bucket j for $j > i$, and lower priority than all transactions of bucket g for $g < i$. Within each bucket, transactions in the *HIT* group have a higher priority than transactions in the *MISS* group. The transaction priority ordering in the *HIT* group is Earliest Deadline, while the priority ordering in the *MISS* group is Highest Value. The I_T priority component serves to break the tie for transactions in the *HIT* or *MISS* group that have identical deadlines or values, respectively. This ensures a *total* priority ordering of all transactions in the system.

As mentioned earlier, the priority assignment process within each bucket is similar to that of the AED algorithm. There are, however, two important differences: First, the transaction list within a bucket is ordered based on transaction values, instead of transaction keys. Second, the priority ordering within the *MISS* group is Highest Value instead of Random Priority. In the special case where all the transactions of a bucket have the same value, however, the priority assignment process is identical to that of the AED algorithm.

An important point to note here is that transactions retain their initial bucket, group and priority assignments for the entire duration of their residence in the system.

8.2.4. Discussion

The core principle of the AED mapping is to use an Earliest Deadline schedule among the largest possible set of transactions that can be completed by their deadline, i.e. the *HIT* group. The HED mapping extends this principle in two ways: First, within a bucket, it ensures that higher-valued transactions are given precedence in populating the *HIT* group, as this should increase the realized value. Second, by creating a value-based hierarchy of buckets, the HED algorithm ensures that transactions with substantially different values are not assigned to the same bucket. The reason for doing this is the following: The AED algorithm only *approximates* a hit ratio of 1.0 in the *HIT* group. Therefore, there is always the risk of losing an extremely high-valued transaction since transactions within the *HIT* group are prioritized by deadline and not by value. Missing the deadlines of such "golden" transactions can seriously affect the realized value; our solution is to establish a value-based bucket hierarchy, thus ensuring the completion of these high-valued transactions.

8.3. Concurrency Control Algorithms

We have shown in Chapters 4, 5 and 6 that optimistic algorithms outperform locking algorithms in a firm deadline RTDBS when all transactions have the same value. In Chapter 7, these results were extended to the case where transactions have different values assuming fixed-tradeoff priority assignments. One of the goals of this chapter is to determine whether the superior performance of optimistic algorithms also holds for the HED algorithm. We compare the performance of the 2PL-HP, OPT-BC and OPT-WAIT concurrency control algorithms in this chapter. As in Chapter 7, the OPT-WAIT variant considered in the experiments of this chapter is the OPT-WAIT(S) algorithm. This algorithm implements a policy where a transaction that is priority-waiting at its deadline is always aborted and discarded, thus ensuring that high-priority transactions are never restarted by low priority transactions.

8.4. Experiments and Results

In this section, we present performance results for a set of experiments comparing the Earliest Deadline, Highest Value and Hierarchical Earliest Deadline priority mappings. We discuss our results with regard to the impact of different transaction value distributions. As in the previous chapter, deadline formula DF3 is used to assign transaction deadlines in the experiments described here. With DF3, a general workload where transactions have a variety of slack ratios is constructed.

While describing the HED algorithm in Section 8.2, we mentioned a parameter, *SpreadFactor*, which determines the allowable spread of transaction values in each bucket. The choice of *SpreadFactor* is constrained by two opposing considerations: Having a large *SpreadFactor* results in high-valued transactions being grouped with low-valued transactions. As an extreme case, a *SpreadFactor* of ∞ assigns all transactions to a single bucket. On the other hand, having a small *SpreadFactor* causes the algorithm to give undue importance to value over deadline. As an extreme case, again, a *SpreadFactor* of 0 assigns a different bucket for each distinct transaction value. We conducted several experiments on the performance sensitivity of HED to the *SpreadFactor* settings for different value distributions. Our experiments showed that settings between 2 and 4 delivered a reasonable tradeoff between the above conflicting considerations for the range of workloads studied in this chapter. For the experiments presented here, we chose a value of 3 for the **SpreadFactor** setting. In addition, the *HITbatch* and *ALLbatch* parameters of the AED algorithm employed in each bucket are set to 20 (as in Chapter 6) for the experiments described here.

8.4.1. Experiment 1: Uniform Value Distribution

Our first experiment investigated the case where transaction values are uniformly distributed over a limited range. The settings of the workload parameters and the system parameters for this experiment are shown in Table 8.1. The value-related parameter settings are identical to those of the uniform-value experiments of Chapter 7. They construct a single-class workload where transaction values range uniformly between 50.0 and 150.0. The settings also

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	8
WriteProb	0.25	*NumDisks*	16
DeadlineFormula	DF3	*PageCPU*	10ms
LSF	1.33	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		
NumClasses	1		
ProbClass[i]	1.0		
OfferedValue[i]	1.0		
MeanValue[i]	100.0		
SprdValue[i]	0.5		

Table 8.1: Baseline Parameter Settings

result in the system performance being limited by both resource contention and data contention. For this experiment, 2PL-HP is used as the concurrency control mechanism, and the various concurrency control alternatives are compared in Experiment 3.

For this experiment, Figures 8.2(a) and 8.2(b) show the Loss Percent results under normal load and heavy load conditions, respectively. From this set of graphs, it is clear that at normal loads, the Earliest Deadline (ED) mapping delivers the most value. As the system load is increased, however, ED's performance steeply degrades and becomes considerably worse than that of HV at high loads. This behavior of ED and HV is exactly what would be expected from our observations in the experiments of Chapter 7. Moving on to the HED mapping, we see that at normal loads it behaves almost identically to Earliest Deadline. Then, as the overload region is entered, it changes its behavior to be similar to that of Highest Value. Therefore, in an overall sense, the HED mapping delivers the best performance. It should be noted that for this uniform workload, all transactions are assigned to the same bucket since transaction

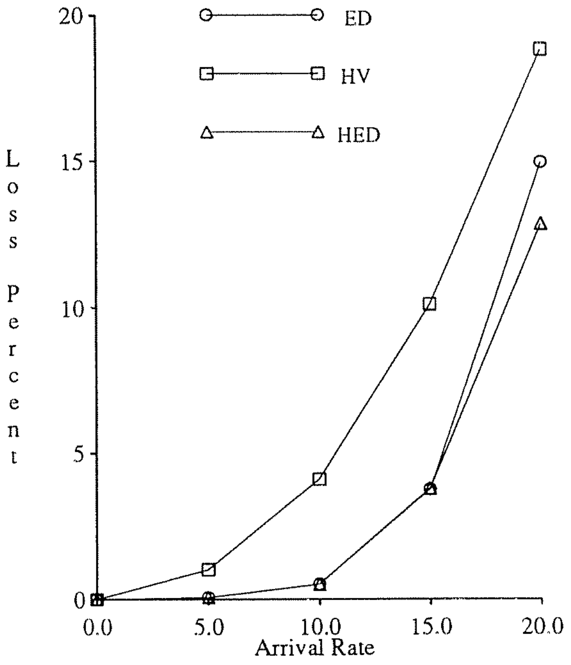

Figure 8.1(a): Baseline (Normal)

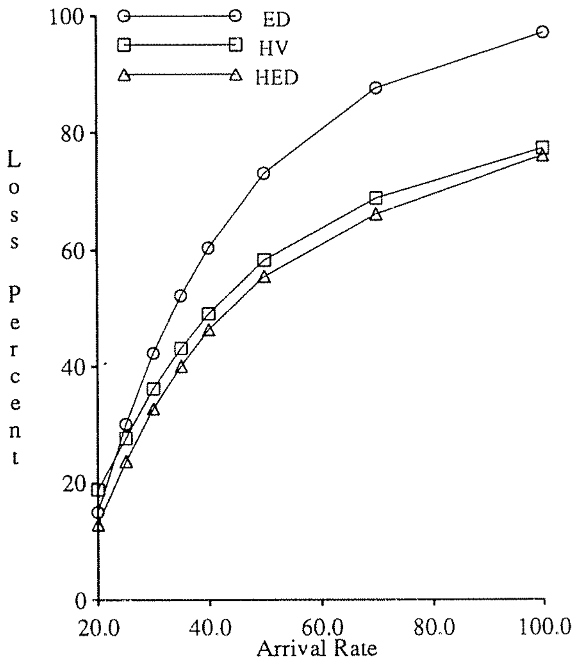

Figure 8.1(b): Baseline (Heavy)

values are all within a factor of 3 (the SpreadFactor setting) of each other. The HED mapping's feedback mechanism is effective in detecting overload conditions and limiting the size of the *HIT* group to a manageable number. HED's priority assignment mechanism then realizes a high value by populating the *HIT* group with higher-valued transactions.

8.4.2. Experiment 2: Skewed Value Distribution

The next experiment examined the effect of having a skew in the transaction value distribution. For this experiment, the parameters are set as shown in Table 8.2. The value-related parameter settings are identical to those of the skewed value experiments of Chapter 7. They construct a two-class workload where 10 percent of the transactions deliver 90 percent of the offered value. The values of the transactions from the first class vary between 450.0 and 12.75.0, while the values of the second class vary between 5.5 and 16.0.

The Loss Percent results for this experiment are shown in Figures 8.3(a) and 8.3(b). From these figures we note that the performance of the Earliest Deadline (ED) mapping remains the same as for the uniform value distribution (compare with Figures 8.2(a) and 8.2(b)). since the ED mapping is value-indifferent. The figures also show that the performance of the Highest Value (HV) mapping improves greatly as compared to the uniform value case. This is exactly

Workload Parameter	**Value**	**System Parameter**	**Value**
MeanTransSize	16 pages	*DatabaseSize*	1000 pages
SprdSize	0.5	*NumCPUs*	8
WriteProb	0.25	*NumDisks*	16
DeadlineFormula	DF3	*PageCPU*	10ms
LSF	1.33	*PageDisk*	20ms
HSF	4.0	*CCReqCPU*	0.0
GlobalMeanValue	100.0		
NumClasses	2		
ProbClass[i]	0.1,0.9		
OfferedValue[i]	0.9,0.1		
MeanValue[i]	900.0,11.1		
SprdValue[i]	0.5,0.5		

Table 8.2: Skewed Value Settings

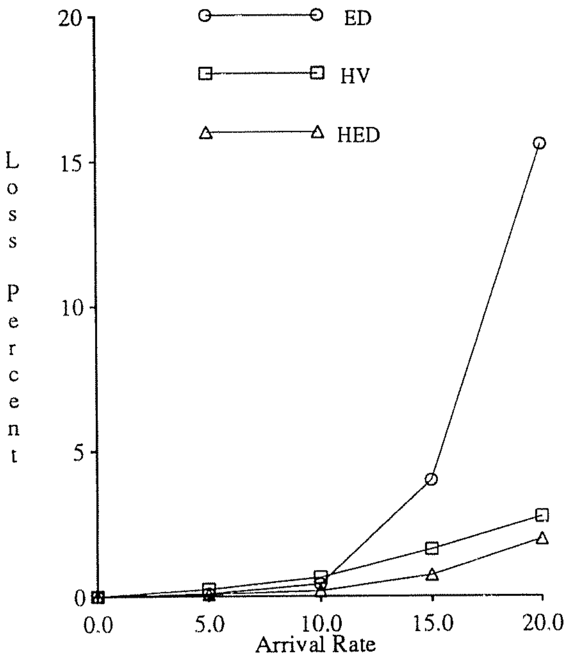

Figure 8.3(a): Skewed Values (Normal)

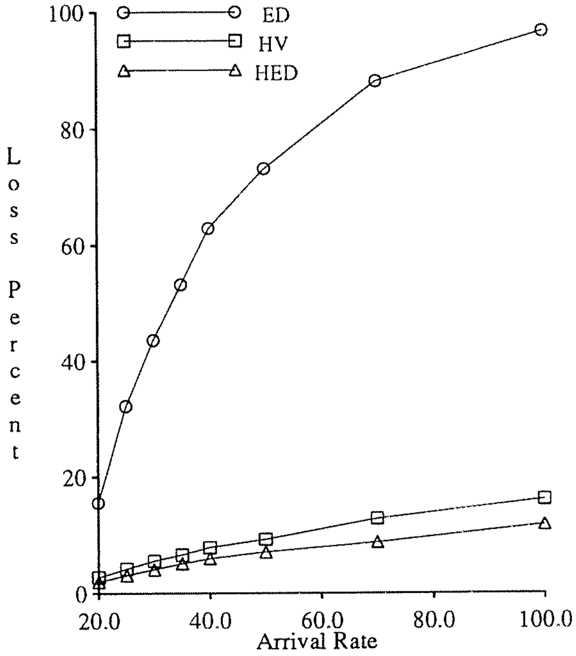

Figure 8.3(b): Skewed Values (Heavy)

the behavior that would be expected from our observations in the experiments of Chapter 7. Moving on to the HED mapping, we observe that it performs better than both ED and HV over the entire loading range. The reason for its good performance is twofold: First, the bucket hierarchy construction ensures that the few high-valued transactions in the workload at any given time are assigned to a separate higher priority bucket. This guarantees that these transactions are completed and therefore their value is realized. Second, using the AED algorithm within each bucket results in more deadlines being made and provides a corresponding increase in the realized value.

8.4.3. Experiment 3: Concurrency Control

The earlier experiments have shown that the HED priority assignment algorithm provides the best overall performance. Here, we investigate the behavior of various concurrency control algorithms in association with the HED algorithm. We compare the performance of 2PL-HP with that of OPT-BC and OPT-WAIT for both the uniform value distribution of Experiment 1 and the skewed value distribution of Experiment 2.

For the uniform value case, Figure 8.4 shows the Loss Percent results. We observe here that OPT-WAIT performs better than 2PL-HP over the entire loading range. This is because OPT-WAIT, which is as priority-cognizant as 2PL-HP, misses fewer deadlines than 2PL-HP. While OPT-WAIT does suffer from the problem of "wasted sacrifices," the effects of this problem are smaller than those of the "wasted restarts" problem of 2PL-HP.

OPT-BC performs worse than 2PL-HP at very low loads due to its priority-indifference. At slightly higher loads, however, it begins to perform better since its fewer missed deadlines more than compensate for its priority-indifference. OPT-WAIT and OPT-BC perform identically at heavy loads since the priority wait-mechanism rarely comes into play – heavy resource contention prevents low-priority transactions from reaching validation much before their deadlines. This is similar to our observations in the experiments of Chapter 5.

For the skewed value case, Figure 8.5 shows the Loss Percent results. We observe here

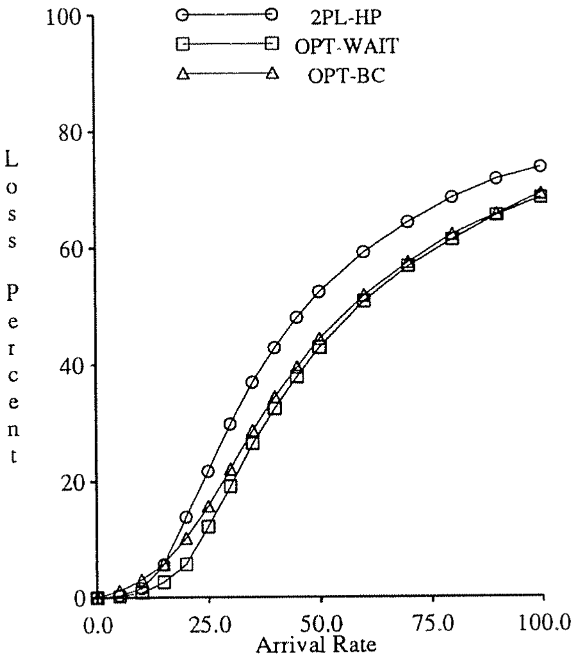

Figure 8.4: Concurrency (Uniform)

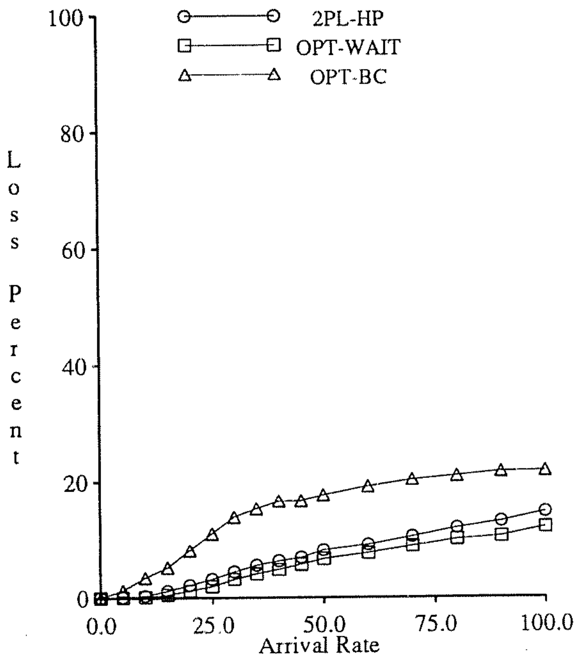

Figure 8.5: Concurrency (Skew)

that OPT-WAIT performs better than 2PL-HP over the entire loading range. Since OPT-WAIT is willing to sacrifice low priority transactions for high priority transactions, high value transactions are essentially guaranteed to complete before their deadlines. This is similar to the behavior of 2PL-HP. In addition, OPT-WAIT gains some extra value over 2PL-HP due to missing the deadlines of a smaller number of low-value transactions. To sum up, as compared to 2PL-HP, OPT-WAIT meets the deadlines of all of the same high-value transactions and misses fewer low-value transactions. The performance of OPT-BC, on the other hand, is significantly worse than that of the other two algorithms. This is because OPT-BC does not take transaction priorities into account, and therefore permits high-value transactions to be restarted by low-value transactions. This results in some of the "golden" transactions missing their deadlines.

From the above experiments, we conclude that the basic results about the superior performance of optimistic algorithms hold under the HED priority assignment policy as well.

8.5. Conclusions

In this chapter, we introduced the Hierarchical Earliest Deadline (HED) algorithm to address the issue of priority assignment when transactions are distinguished by both values and deadlines. The HED algorithm groups transactions, based on their values, into a hierarchy of prioritized buckets; it then uses the AED algorithm within each bucket. Our experiments showed that, both for workloads with limited spread in transaction values and for workloads with pronounced skew in transaction values, the HED algorithm provided the best overall performance. At light loads, its behavior was identical to that of Earliest Deadline, while at heavy loads its performance was better than that of Highest Value. Use of the AED algorithm within the transactions of a bucket decreased the number of missed deadlines. Also, by giving preference to more valuable transactions in populating the *HIT* group of each bucket, the HED algorithm increased the realized value as compared to HV. For workloads with pronounced skew in transaction values, the hierarchical nature of the HED algorithm was effective in ensuring that "golden" (high-valued) transactions were completed and their value realized.

CHAPTER 9

SUMMARY AND FUTURE RESEARCH

9.1. Summary of Results

In recent years, applications from increasingly complex domains have started using database technology. For many such applications, the complexities of the domain are reflected in the underlying database system. Therefore, the database system can no longer be viewed in isolation but must be viewed, instead, as an integral component of the application. In this thesis, we have considered data-intensive applications that have timing requirements, such as program trading in stock markets and operation controllers in computer integrated manufacturing systems. The objective of our investigation has been to determine how database support for such applications can include helping them meet their timing requirements. In particular, we have addressed the issue of how transactions with application-defined deadlines should be scheduled by a database system in order to meet their deadlines.

We have restricted our attention in this dissertation to real-time database applications that have firm deadlines. For such applications, transactions that miss their deadline are considered to be worthless and are therefore immediately discarded. In this framework, we considered two cases: (1) the case where transactions all have the same value from the application's perspective and the goal of the RTDBS is to maximize the number of in-time transactions, and (2) the case where transactions have different values and the goal of the RTDBS is to maximize the total value of the in-time transactions. Using a detailed simulation model of a multiprocessor RTDBS, we studied the real-time performance of various transaction scheduling algorithms. These algorithms varied in their priority assignment and concurrency control components but had common priority-based resource scheduling disciplines. None of the algorithms assumed any a-priori knowledge of transaction processing requirements or transaction data accesses. Five sets of experiments were conducted, the first three dealing with the same-value case and the remaining two with the multiple-value case. Before summarizing the results of these experiments, we caution the reader that the results are applicable solely to firm-deadline applications.

We began our study of RTDBS transaction scheduling algorithms by investigating the case where transactions all have the same value. In the first set of experiments, we evaluated the performance of various concurrency control alternatives. In particular, we compared a real-time locking protocol with a conventional optimistic concurrency control algorithm. Our results showed the optimistic algorithm to perform better than the locking protocol over a wide range of system loading and resource availability. This was an important result because it is opposite to that seen in resource-constrained conventional DBMSs, where locking algorithms outperform optimistic protocols. The change in behavior primarily arises from the RTDBS feature of discarding late transactions. This feature caused the locking algorithm, which resolved data conflicts as soon as they occur, to be adversely affected by transactions that were eventually discarded. In contrast, the optimistic algorithm was unaffected by such transactions due to its delayed conflict resolution approach. Interestingly, this effect was strong enough to make the conventional (non-real-time) optimistic algorithm perform better than the real-time locking algorithm.

Having established that the basic optimistic approach to concurrency control has features that are suited to the firm-deadline RTDBS environment, we followed up by evaluating different approaches to developing real-time optimistic algorithms. Our experiments showed that a priority-wait approach, where low priority validating transactions wait for high-priority conflicting transactions to complete first, delivers the best performance at low loads. At high loads, however, priority-waiting actually degraded performance by significantly increasing the level of data contention in the system. Therefore, we developed the WAIT-50 dynamic optimistic algorithm, which uses a simple 50% rule to control the priority-wait mechanism. Our experiments showed WAIT-50 to deliver improved performance in a stable manner.

In these initial sets of experiments, transactions were prioritized according to Earliest Deadline, a priority policy that is commonly used in existing real-time systems. Apart from the concurrency control results, a secondary observation of these experiments was that using Earliest Deadline resulted in few missed deadlines at light loads but in steeply degraded performance at heavier loads. In fact, at sufficiently heavy loads, even a random priority assignment was found to provide better performance than Earliest Deadline. To address this instability problem of Earliest Deadline, we developed the Adaptive Earliest Deadline (AED) priority assignment algorithm. AED stabilizes the overload performance of Earliest Deadline while retaining its low load virtues. The AED algorithm uses a feedback process to estimate the number of transactions that could be successfully completed by the system, and restricts the use of an Earliest Deadline priority ordering to a set of transactions whose number is equal to this estimate. Experiments comparing AED with several other fixed priority mappings showed it to provide the best overall performance under a variety of workloads.

Having studied the priority assignment and concurrency control components of real-time transaction scheduling for the same-value case, we then moved on to the case where transactions have different values. Our first set of experiments here focused on the priority assignment component. We conducted experiments for various transaction value distributions with several priority mappings that establish different fixed tradeoffs between values and deadlines. Our experiments showed that no single tradeoff is appropriate under all circumstances. Rather, the right tradeoff is a function of the workload and system operating conditions. This result highlighted the need for a priority assignment algorithm that could adaptively vary the value-deadline tradeoff to match the operating environment. With this goal in mind, we developed the Hierarchical Earliest Deadline (HED) priority assignment algorithm. The HED algorithm groups transactions, based on their value, into a hierarchy of prioritized buckets; it then uses the AED algorithm within each bucket. Our final set of experiments showed that the

HED algorithm consistently provides better overall performance than all of the fixed-tradeoff algorithms that were examined. With regard to concurrency control, our experiments showed that real-time optimistic algorithms outperform real-time locking protocols in the value-based framework as well.

To summarize, this thesis has shown that fixed priority assignment policies and locking-based concurrency control protocols are not the methods of choice for transaction scheduling in a firm-deadline RTDBS. Instead, we recommend the use of adaptive algorithms like HED for priority assignment and priority-wait-based optimistic algorithms for concurrency control.

9.2. Future Research Directions

We developed the WAIT-50 algorithm assuming the use of an Earliest Deadline priority policy and that all transactions have the same value. An attractive avenue for future work is to develop similar dynamic optimistic algorithms that are suitable for use with priority assignments like AED or HED. One of the main problems here is how to decide whether a transaction that is waiting at its deadline is to be committed or discarded.

Memory resources (i.e. buffer frames) are an integral feature of database systems. For the sake of simplicity, we have not included this resource in our performance evaluation framework. An interesting future area of research would be to develop real-time policies for buffer allocation and replacement and evaluate their performance effects.

A basic assumption in our study is that the RTDBS has no a-priori knowledge of transaction processing requirements or of transaction data accesses. It would be worthwhile to investigate how, and to what extent, performance could be improved if the RTDBS were to be provided more information about transaction characteristics (e.g., transaction size estimates).

As mentioned earlier, our study has focused on firm deadline real-time database applications. Late transactions are therefore considered to be worthless and are simply discarded. There exist real-time database applications, however, that have soft deadlines; that is, there is some residual utility to completing transactions after their deadlines. Developing transaction scheduling algorithms to maximize the realized value for such applications is a challenging future research problem. The problem is complicated since the database system has to effectively schedule both feasible and late transactions. Also, the priority assignment policy has to take into consideration the values of a transaction both before and after its deadline; if the value of late transactions decreases with their delay, the priority assignment must adjust to the transaction's decreasing value. Finally, since the feature of discarding late transactions is absent in a soft deadline application, the performance behavior of the various concurrency control mechanisms may differ from what was seen here for the firm deadline environment.

An implicit assumption in this study, and in most related work on real-time database systems, is that timing constraints are associated only with transactions. There are several applications, however, where the data itself is subject to time constraints. For example, in applications involving object tracking, the data describing object locations is time-constrained in that it is valid for a limited time only. Transactions operating with such data have to ensure that they use temporally consistent values. This may call for the use of multiple versions of each data item to provide a history of data values. We therefore need to look into the performance of different real-time multi-version concurrency control algorithms.

Finally, data distribution and replication offer opportunities for improving real-time performance through load balancing, increased availability of data, and increased reliability. At the same time, though, they magnify the problems of transaction scheduling since factors like inter-site communication and consistency of replicated data have to be considered. A potentially fruitful area of future research is to develop and evaluate distributed real-time transaction scheduling algorithms.

REFERENCES

[Abbo87] Abbott, R., and Garcia-Molina, H., "What is a Real-Time Database System," *Proc. of 4th IEEE Workshop on Real-Time Software and Operating Systems*, July 1987.

[Abbo88a] Abbott, R., and Garcia-Molina, H., "Scheduling Real-Time Transactions," *ACM SIGMOD Record* , March 1988.

[Abbo88b] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions: A Performance Evaluation," *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[Abbo89] Abbott, R., and Garcia-Molina, H., "Scheduling Real-time Transactions With Disk Resident Data," *Proc. of 15th Intl. Conf. on Very Large Data Bases*, August 1989.

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

[Agra87] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transactions on Database Systems*, Vol. 12, No. 4, December 1987.

[Baru91] Baruah, S., and Rosier, L., "Limitations Concerning On-Line Scheduling Algorithms for Overloaded Real-Time Systems," *Proc. of 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[Bern81] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, June 1981.

[Bern87] Bernstein, P., Hadzilacos, V., and Goodman, N., "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.

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

[Buch89] Buchmann, A., et al, "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Proc. of 5th Intl. Conf. on Data Engineering*, February 1989.

[Bult88] Bultzingsloewen, G., et al, "KARDAMOM – A Dataflow Machine for Real-Time Applications," *ACM SIGMOD Record* , March 1988.

[Care88] Carey, M. J., and Livny, M., "Distributed Concurrency Control Performance: A Study of Algorithms, Distribution, and Replication," *Proc. of 14th Intl. Conf. on Very Large Data Bases*, August 1988.

[COMP91] *IEEE Computer*, Special Issue on Real-Time Systems, C. Krishna and Y. Lee, eds., May 1991.

[Cook90] Cook, R., and Oh, H., "The Starlite Project," *Proc. of 3rd Symposium on Frontiers of Massively Parallel Computation*, Univ. of Maryland, College Park, Oct. 1990.

[Cook91] Cook, R., et al, "New Paradigms for Real-Time Database Systems," *Proc. of 8th IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

[Davi88] Davidson, S., and Watters, A., "Partial Computation in Real-Time Database Systems," *Proc. of 5th Workshop on Real-Time Software and Operating Systems*, May 1988.

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

[Dert74] Dertouzos, M., "Control Robotics: the procedural control of physical processes," *Proc. of IFIP Congress*, 1974.

[Eswa76] Eswaran, K., et al, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of ACM* , Nov. 1976.

[Fran85] Franaszek, P., and Robison, J., "Limitations of Concurrency in Transaction Processing," *ACM Trans. on Database Systems* 12(1), March 1985.

[Gray79] Gray, J., "Notes On Database Operating Systems," in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Hard84] Harder, T., "Observations on Optimistic Concurrency Control Schemes," *Information Systems*, Vol.9, No.2, 1984.

[Hari90a] Haritsa, J., Carey, M., Livny, M., "On Being Optimistic about Real-Time Constraints," *Technical Report 906*, Computer Sciences Dept., Univ. of Wisconsin, Madison, January 1990.

[Hari90b] Haritsa, J., Carey, M., Livny, M., "Dynamic Real-time Optimistic Concurrency Control," *Technical Report 981*, Computer Sciences Dept., Univ. of Wisconsin, Madison, November 1990.

[Hari91a] Haritsa, J., Livny, M., Carey, M., "Value-Based Scheduling in Real-Time Database Systems," *Technical Report 1024*, Computer Sciences Dept., Univ. of Wisconsin, Madison, May 1991.

[Hari91b] Haritsa, J., "Earliest-Deadline Scheduling for Real-Time Database Systems," *Technical Report 1025*, Computer Sciences Dept., Univ. of Wisconsin, Madison, May 1991.

[Hong89] Hong, J., Tan, X., and Towsley, D., "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Transactions on Computers*, Dec. 1989.

[Huan89] Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K., "Experimental Evaluation of Real-Time Transaction Processing," *Proc. of 10th IEEE Real-Time Systems Symposium*, Dec. 1989.

[Huan90a] Huang, J., and Stankovic, J., "Buffer Management in Real-Time Databases", *COINS Technical Report* 90-65, Univ. of Massachusetts, Amherst, July 1990.

[Huan90b] Huang, J., and Stankovic, J., "Concurrency Control in Real-Time Database Systems: Optimistic Scheme vs. Two-Phase Locking", *COINS Technical Report* 90-66, Univ. of Massachusetts, Amherst, July 1990.

[Huan90c] Huang, J., Stankovic, J., Ramamritham, K., and Towsley, D., "On Using Priority Inheritance in Real-Time Databases", *COINS Technical Report 90-121*, Univ. of Massachusetts, Amherst, November 1990.

[IEEE87] *IEEE Transactions on Computers*, Vol. C-36, Special Issue on Real-Time Systems, K. Shin, ed., August 1987.

[Jack63] Jackson, J.R., "Jobshop-like Queuing Systems," *Management Science* , No. 12-1, Oct. 1963.

[Jauh90] Jauhari, R., "Priority Scheduling in Database Systems," *Ph.D. Thesis*, Computer Sciences Dept., Univ. of Wisconsin, Madison, August 1990.

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

[Klei76] Kleinrock, L., "Queueing Systems", Vol. II, John Wiley & Sons, 1976.

[Kort90] Korth, H. F., Soparkar, N., and Silberschatz, A., "Triggered Real-Time Databases with Consistency Constraints," *Proc. of 16th Intl. Conf. on Very Large Data Bases*, August 1990.

[Kung81] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems* Vol. 6, No. 2, June 1981.

[Lin88] Lin, K-J., and Lin, M-J., "Enhancing Availability in Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[Lin89] Lin, K., "Consistency Issues in Real-Time Database Systems," *Proc. of 22nd Hawaii Intl. Conf. on System Sciences*, January 1989.

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

[Liu73] Liu, C. and Layland, J., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM*, Jan. 1973.

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

[Livn88] Livny, M., "DeNet User's Guide," Version 1.0, Computer Sciences Department, Univ. of Wisconsin, Madison, 1988.

[Lock86] Locke, C., "Best Effort Decision Making for Real-Time Scheduling," *Ph.D. Thesis*, Dept. of Computer Science, Carnegie-Mellon University, May 1986.

[Mena82] Menasce, D., and Nakanishi, T., "Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems," *Information Systems* , vol. 7-1, 1982.

[Mok78] Mok, A. and Dertouzos, M., "Multi-processor Scheduling in a Hard Real-Time Environment," *Proc. of the 7th Texas Conf. on Computing Systems*, Oct. 1978.

[Mok83] Mok, A., "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," *Ph.D. Thesis*, Laboratory for Computer Science, Massachussetts Institute of Technology, May 1983.

[Oszo90] Ozsoyoglu, G., et al, "CASE-DB: A Real-Time Database Management System," *Tech. Rep.*, Case Western Reserve University, 1990.

[Panw88] Panwar, S. and Towsley, D., "On the Optimality of the STE Rule for Multiple Server Queues that Serve Customers with Deadlines," *COINS Technical Report 88-81*, Univ. of Massachusetts, Amherst, July 1988.

[Pete86] Peterson, J., and Silberschatz, A., "Operating Systems Concepts," Addison-Wesley, 1986.

[Pein88] Peinl, P., Reuter, A., and Sammer, H., "High Contention in a Stock Trading Database: A Case Study," *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, June 1988.

[Rama89] Ramamritham, K., and Stankovic, J., "Overview of the SPRING Project," *IEEE Real-Time Systems Newsletter* , Winter 1989.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System," *Ph. D. Thesis*, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1978.

[Robi82] Robinson, J., "Design of Concurrency Controls for Transaction Processing Systems," *Ph.D. Thesis* , Carnegie Mellon University, 1982.

[Rose78] Rosenkrantz, D., Stearns, R., and Lewis, P., "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978.

[Sha87] Sha, L., Rajkumar, R., Lehoczky, J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *Technical Report CMU-CS-87-181*, Departments of CS, ECE, and Statistics, Carnegie Mellon University, 1987.

[Sha88] Sha, L., Rajkumar, R., Lehoczky, J. P., "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record 17*, March 1988.

[SIGM88] *SIGMOD Record*, Vol. 17, No. 1, Special Issue on Real-Time Data Base Systems, S. Son, ed., March 1988.

[Sing88] Singhal, M., "Issues and Approaches to Design of Real-Time Database Systems," *ACM SIGMOD Record* , March 1988.

[Son87] Son, S., "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proc. of 8th IEEE Real-Time Systems Symposium*, Dec. 1987.

[Son88a] Son, S., "Real-Time Database Systems: Issues and Approaches," *ACM SIGMOD Record*, March 1988.

[Son88b] Son, S., "A Message-Based Approach to Distributed Database Prototyping," *Proc. of 5th IEEE Workshop on Real-Time Software and Operating Systems*, May 1988.

[Son89] Son, S., and Cook, R., "Scheduling and Consistency in Real-Time Database Systems," *IEEE Real-Time Systems Newsletter*, Summer 1989.

[Son90a] Son, S. and Haghighi, N., "Performance Evaluation of Multiversion Database Systems," *Proc. of 6th IEEE Intl. Conf. on Data Engineering*, February 1990.

[Son90b] Son, S. and Lee, J., "Scheduling Real-Time Transactions in Distributed Database Systems," *Proc. of 7th IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[Son90c] Son, S. and Cook, R., "StarLite: An Environment for Prototyping and Integrated Design of Distributed Real-Time Software," *Proc. of 2nd Intl. Conf. on Computer Integrated Manufacturing*, May 1990.

[Son90d] Son, S., and Chang, C., "Performance Evaluation of Real-Time Locking Protocols using a Distributed Software Prototyping Environment," *Proc. of 10th Intl. Conf. on Distributed Computing Systems*, June 1990.

[Son90e] Son, S., "Real-Time Database Systems: A New Challenge," *Data Engineering*, vol. 13, no.4, December 1990.

[Song90] Song, X., and Liu, J., "Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency," *Proc. of COMPSAC*, October 1990.

[Stan88a] Stankovic, J., Zhao, W., "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[Stan88b] Stankovic, J., "Real-Time Computing Systems: The Next Generation," *COINS Technical Report 88-06*, Univ. of Massachusetts, Amherst, June 1988.

[Stan88c] Stankovic, J., "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, Vol. 21, No. 12, October 1988.

[Tay84] Tay, Y., "A Mean Value Performance Model for Locking in Databases," *Ph.D. Thesis*, Harvard University, Feb. 1984.

[Ulu91a] Ulusoy, O., and Belford, G., "Concurrency Control in Real-Time Database Systems," *Unpublished manuscript*, Univ. of Illinois, Urbana-Champaign.

[Ulu91b] Ulusoy, O., and Belford, G., "Real-Time Lock-Based Concurrency Control in a Non-replicated Distributed Database System," *Unpublished manuscript*, Univ. of Illinois, Urbana-Champaign.

[Vrbs88] Vrbsky, S., and Lin, K., "Recovering Imprecise Transactions with Real-Time Constraints," *Proc. of 7th Symposium on Reliable Distributed Systems*, October 1988.

[Vrbs90] Vrbsky, S., Liu, J., and Smith, K., "An Object-Oriented Query Processor that Returns Monotonically Improving Approximate Answers", *Technical Report 90-1568*, Univ. of Illinois, Urbana-Champaign, February 1990.

[Wirt82] Wirth, N., "Programming in Modula-2," Springer-Verlag, 1982.

APPENDIX A

Implementation of OPT-BC

We present here a lock-based mechanism for implementing OPT-BC, the broadcast commit optimistic concurrency control algorithm. Our method is as follows: Whenever a transaction wants to read an object, it sets a read lock on it. If an object is to be written, the update is made to a private copy. After the transaction comes to validation and decides to commit, which requires all of its private copies to be made public, write locks are set on all the objects that are to be updated, and the updates are then performed.

Read locks are individually requested with the synchronous lock call (using System R lock manager notation [Gray79]) LOCK (*ReadSet$_i$*, SHARED, WAIT), where *ReadSet$_i$* refers to the specific data object on which the read lock is being requested. All write locks are requested simultaneously at commit time with the non-blocking lock call LOCK (*WriteSet*, EXCLUSIVE, TEST). Write locks preempt read locks and therefore the writeset lock request will always succeed[1], except as noted below.

In the process of giving locks on the writeset, the lock manager informs the recovery manager of the list of preempted conflicting transactions to be aborted. The transaction abort processing can be done asynchronously by having the lock manager maintain a table of the current status of all executing transactions. The status of a transaction can be either *Running*, *AbortScheduled* or *Committing*. Any further lock request from a transaction scheduled for abort is refused by the lock manager. In particular, a validating transaction will be refused its

[1] A special case where the writeset lock request may fail temporarily is for transactions with "blind writes" [Bern87]. A transaction with blind writes will have its writeset lock request granted once the write locks currently existing on objects in its "blind write set" are released.

writeset lock request if it has been scheduled for abort by a previously committed (or currently committing) transaction. A transaction whose lock request is thus denied is put to sleep.and aborted at a later time.

Once a transaction has obtained its write locks, it releases all its read locks with the following call to the lock manager UNLOCK(*ReadSet* – *WriteSet*). It then makes its updates and commits, releasing all write locks at the end with the call: UNLOCK(*WriteSet*).

Implementation of OPT-WAIT

In the OPT-BC algorithm, when a validating transaction makes the request for simultaneous locks on its entire writeset, the request is guaranteed to succeed unless the transaction has been scheduled for abort. For the OPT-WAIT algorithm, however, the success of the request is also dependent on the current composition of the conflict set of the validating transaction. If there are higher priority transactions in the conflict set, the writeset lock request is refused and the transaction is made to wait. A waiting transaction needs to periodically reissue its writeset lock request since the composition of its conflict set is a function of time. A reasonably efficient method to do this is for the lock manager to maintain a list of waiters for each running transaction together with a count of conflicting higher priority transactions for each waiting transaction. This count is evaluated at the time of making the writeset lock request, and if the count is non-zero, the requesting transaction is put to sleep. Whenever a transaction commits, restarts, or is discarded, the high priority count of each of its waiters is decremented. The terminated transaction is also taken off any list of waiters in which it is a member. If a waiter's high priority count goes to zero, it is awakened. The awakened waiter then reissues its writeset lock request. This process continues until the waiter is either restarted or is given the writeset lock.

The WAIT-50 algorithm can be implemented, in a similar fashion, by making simple extensions to the scheme described above.

APPENDIX B

Analytical Results

In all the resource-constrained experiments of this thesis, the Miss Percent characteristics of transaction scheduling algorithms exhibited a *S-shape*. In this section, we try to provide a theoretical basis for this shape. Using the terminology of queueing networks we can, in a very loose and abstract fashion, compare a firm deadline system to a M/M/1/K system. A M/M/1/K queueing model characterizes a system with Poisson customer arrivals, exponential customer service times, a single server, and a maximum of K customers in the system. New customers that arrive when there are already K customers in the system are thrown away. If we take the percentage of customers thrown away by the server to be analogous to the Miss Percent metric, and denote it by α, we then have the result (using Jackson's Theorem [Jack63], and assuming a mean customer service requirement of 1 time unit),

$$\alpha = 100 * \left[1 - \frac{\lambda^K - 1}{\lambda^{K+1} - 1} \right]$$

where λ is the customer arrival rate. A sample graph of α versus λ for $K = 10$ is shown in Figure B.1 and the observed behavior is very similar to that seen in the resource-limited experiments described in this thesis.

The formula for α can be split up in the following fashion :

For $\lambda \ll 1$,

$$\alpha \approx 100 * \lambda^K .$$

For $\lambda \gg 1$,

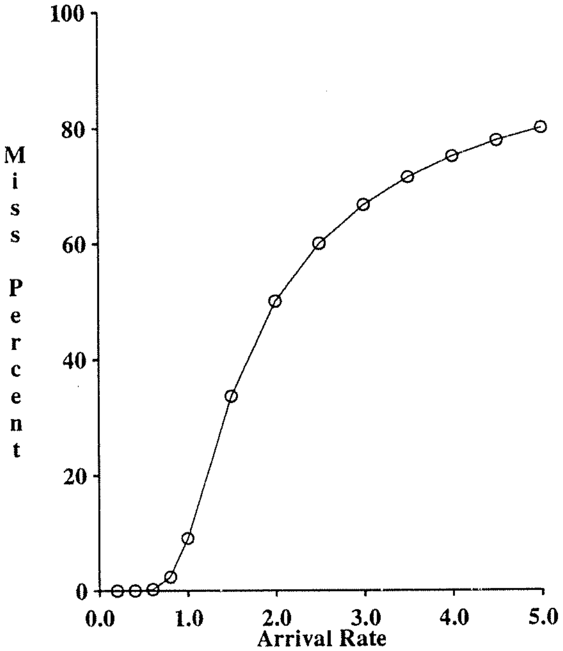

Figure B.1: ALPHA (M/M/1/K Model)

$$\alpha \approx 100 * \left[1-\frac{1}{\lambda}\right].$$

These two formulas give good approximations for the basic shape of the curves seen at normal and heavy loadings, respectively.

APPENDIX C

Conflict Elimination Probability

In this section, we carry out a simple probabilistic analysis of the extent to which the waiting scheme described in Chapter 5 can reduce data conflicts. To do this, we ask the following question: Given that transaction A conflicts with transaction B ($A \rightarrow B$), what is the probability that transaction B does not conflict with A ($B \not\twoheadrightarrow A$)? Assuming a uniform database access pattern and that *WriteSet* $\subseteq$ *ReadSet* for all transactions, and using *overlap* to refer to the cardinality of the set $ReadSet_A \cap ReadSet_B$, the probability $\Pi_{B \not\twoheadrightarrow A \,|\, A \rightarrow B}$ is given by the following expression

$$\Pi_{B \not\twoheadrightarrow A \,|\, A \rightarrow B} = \sum_{k=1}^{\max(overlap)} Pr(B \not\twoheadrightarrow A \mid k\ overlap)\ Pr(k\ overlap \mid A \rightarrow B)$$

which can be re-written as

$$\Pi_{B \not\twoheadrightarrow A \,|\, A \rightarrow B} \tag{C1}$$

$$= \frac{1}{Pr(A \rightarrow B)} \sum_{k=1}^{\max(overlap)} Pr(B \not\twoheadrightarrow A \mid k\ overlap)\ Pr(k\ overlap)\ Pr(A \rightarrow B \mid k\ overlap)$$

The terms in Equation (C1) can be evaluated as shown in Table C.1, where the symbols have the following meaning:

N	=	number of data items in the database
R_A	=	size of readset of transaction A
R_B	=	size of readset of transaction B
W_A	=	size of writeset of transaction A
W_B	=	size of writeset of transaction B

For the sake of simplicity, let us assume that every transaction has the same ratio between the sizes of its write set and read set, and denote it by f $(0 \leq f \leq 1)$. A sample plot of $\Pi_{B \not\twoheadrightarrow A \,|\, A \rightarrow B}$ as a function of f is shown in Figure C.1, for $N = 100$ and $N = 1000$, keeping the read set sizes of

$Pr(A \rightarrow B)$	$=$	$1 - \dfrac{\binom{N - W_A}{R_B}}{\binom{N}{R_B}}$
$Pr(B \not\twoheadrightarrow A \mid k\ overlap)$	$=$	$\dfrac{\binom{R_B - k}{W_B}}{\binom{R_B}{W_B}}$
$Pr(k\ overlap)$	$=$	$\dfrac{\binom{R_A}{k}\binom{N - R_A}{R_B - k}}{\binom{N}{R_B}}$
$Pr(A \rightarrow B \mid k\ overlap)$	$=$	$1 - \dfrac{\binom{R_A - k}{W_A}}{\binom{R_A}{W_A}}$
$\max(overlap)$	$=$	$\min(R_A, R_B)$

Table C.1: Term Evaluations

both A and B constant at 16. Note that for $N \gg (R_A, R_B)$, which is usually true since database sizes are typically much larger than transaction sizes, Equation (C1) reduces to

$$\Pi_{B \not\twoheadrightarrow A \mid A \rightarrow B} \approx 1 - f \tag{C2}$$

and this is evident in the $\Pi_{B \not\twoheadrightarrow A \mid A \rightarrow B}$ curve for $N = 1000$ in Figure C.1. This means that if the write fraction is 0.25, for example, then by waiting, we can resolve about 75% of the conflicts without restarting either transaction.

It should be noted that neither Equation (C1) nor Equation (C2) gives the *actual probability* with which conflicts will be reduced by waiting in our system – this is because some of the

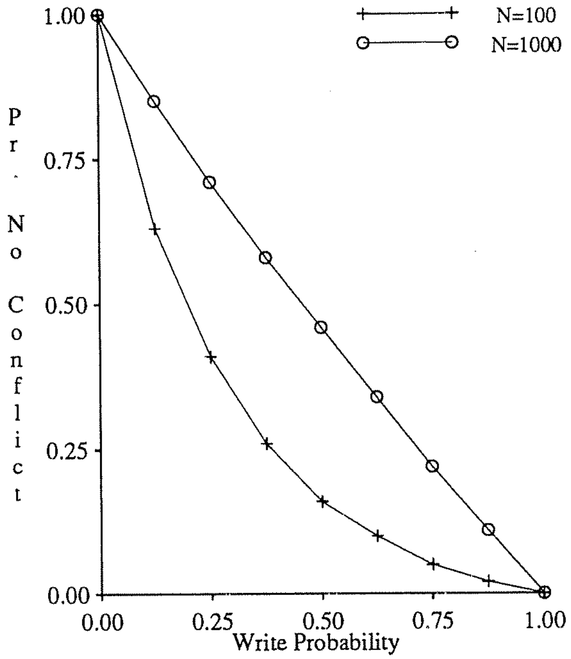

Figure C.1: Uni-directional conflict probabilities.

assumptions made in deriving the equations do not hold in our scenario. These assumptions include the facts that only conflicts with a single transaction are considered, and that transaction priorities are not taken into account. The equations provide a rough idea, however, of the extent to which waiting can be beneficial with respect to reducing data conflicts.