Predictability and Consistency in Real-time Transaction Processing

Approval Sheet

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

[illegible]

Young-Kuk Kim, Author

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

[illegible]

Sang H. Son, Dissertation Advisor

Alfred C Weaver

Alfred C. Weaver, Committee Chairman

Wm. A. Wulf

William A. Wulf

[illegible]

Jörg Liebeherr

K. Preston White

K. Preston White, Jr.

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean, School of Engineering and Applied Science

May 1995

PREDICTABILITY AND CONSISTENCY IN REAL-TIME TRANSACTION PROCESSING

A Dissertation
Presented to
the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment
of the Requirements for the Degree

Doctor of Philosophy
in
Computer Science

by

Young-Kuk Kim
May 1995

Dedicated with love and appreciation to my parents

Acknowledgments

I would like to thank everyone who has, in some way, contributed to this work. I would not have completed it without their support, guidance and encouragement.

First and foremost, I must thank Professor Sang H. Son, my advisor, for his patience and invaluable help in the development and completion of this thesis. The members of my examining committee, Alf Weaver, William Wulf, Jörg Liebeherr and K. Preston White, provided many insightful comments and devoted much time to ensure that this work measures up to the standards of the University of Virginia and the scientific community.

I would like to thank many of my fellow graduate students. Juhnyoung Lee, Ying-feng Oh, Bert Dempsey and Tim Strayer were always available for discussions and reviews of my work. I also thank the former and current members of the Real-Time Database Systems Lab, Marc Poris, Carmen Iannacone, David George, Matt Lehr, Anthony Williams and Stuart Shih. It has been my great pleasure to work with them. It has been a great privilege for me to have such wonderful officemates as Sally McKee, Erik Cota-Robles, Chris Oliver, Rob Hines and Gabriel Ferrer, to name a few. They were there to provide amusement and relief whenever needed and were always happy to answer my questions on English usage during my thesis preparation.

I also would like to thank every staff member and secretary at the Department of Computer Science for their invaluable support. In particular, Mark Smith, Gina Bull, Ray Lubinsky, Ann Bailey, Ginny Hilton and Brenda Lynch deserve my sincere thanks.

Finally, I wish to thank my family for all their love and encouragement. My wife Sohwa always has been a constant source of encouragement and understanding during many trying days of this research. I am also grateful to my parents in Korea. Their steadfast encouragement was perhaps the single most influential factor in my education since childhood. It is to them that this dissertation is dedicated.

Abstract

A real-time database system (RTDBS) can be defined as a database system where transactions are associated with real-time constraints typically in the form of deadlines. The system must process transactions so as to both meet the deadlines and maintain the data consistency. Previous research effort in this field has been focused on scheduling transactions with soft or firm deadlines under the conventional transaction model and database system architecture which cannot support predictable real-time transaction processing.

In this thesis, we provide a framework to realize predictable real-time transaction processing, satisfying both timing and consistency constraints of a real-time database system. First, we classify data objects and transactions found in typical real-time database applications, considering their distinct characteristics and requirements. Each type of real-time data objects has its own correctness criteria, different from the conventional one. Real-time transactions are categorized, according to their timing constraints, arrival patterns, data access patterns, availability of data and run-time requirements, and accessed data type. Our model is a superset of conventional models; it includes both hard and soft real-time transactions, and supports the temporal consistency as well as the logical consistency of the database.

Second, we develop an integrated transaction processing scheme that extends a fixed-priority-based *task* scheduling framework for mixed task sets into a *transaction* processing environment, combining it with *best-effort* real-time transaction scheduling algorithms. This scheme provides *predictability* for a RTDBS in the sense that under our transaction processing scheme it is guaranteed that every application in the system will achieve its own performance goals. Along with the transaction processing scheme, a temporal consistency enforcement scheme called *Static Temporal Consistency Enforcement* (STCE) is introduced.

In the scheme, the temporal consistency requirements of database are transformed into timing constraints of transactions. Thus, as long as the timing constraints of transactions are satisfied in the system, the temporal consistency requirements are automatically achieved.

Third, in order to synchronize the transactions' concurrent accesses to the data objects and maintain the logical consistency of the database, we provide a concurrency control and conflict resolution scheme called *Semantic-based Optimistic Concurrency Control* (SOCC) for our RTDBS model. It is *semantic-based* in that it can utilize the available semantic information about different classes of transactions to make more efficient control decisions, consequently increasing the concurrency level of the system. In SOCC, however, serializable schedules are not always achieved for every class of transactions, since meeting their timing constraints is sometimes more important than maintaining logical consistency of some types of data objects in RTDBS.

Our system allows application developers to specify multiple guarantee levels for different applications. We perform a simulation study to measure the cost of these guarantees realized in our integrated transaction processing scheme. The results show that the higher level of guarantee requires more system resources and therefore causes more non-guaranteed transactions to miss their deadlines.

Our RTDBS model relies on a deterministic subsystem environment. We propose a deterministic computing structure for the model, consisting of a real-time transaction server, a memory-resident data object manager, and a real-time microkernel. The proposed architecture can eliminate sources of unpredictable behavior in the system related to dynamic I/O, such as buffer management, dynamic paging, and disk scheduling. We develop a real-time database system testbed called StarBase, which is currently supporting only firm real-time transactions. The practical issues involved in implementing our RTDBS model and the integrated transaction scheduling scheme on the StarBase platform are discussed.

Finally, we give some thoughts on query processing and recovery mechanisms in the context of memory-resident RTDBS.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As our society becomes more integrated with computer technology, information processing for human activities necessitates computing that responds to requests in *real-time* rather than just with *best-effort*. Many computer systems are now being used to monitor and control physical devices and large complex systems that must have predictable and timely behaviors. We call such systems *real-time systems*. Real-time computing is emerging as an important discipline and an open research area in computer science and engineering. The growing importance of real-time computing in a diverse number of applications such as defense systems, industrial automation, aerospace and medical applications, has resulted in an increased research effort in this area [69].

Real-time systems often require database management services to store large amounts of time-sensitive data and to manipulate the data within given timing constraints. For example, an aerospace tracking system must update its trackfile database continuously with incoming track data from radar, retrieve the current state of a specific track from the database, and perform some image processing to display the track's movement on a graphics map. All these tasks have their own timing requirements.

Neither current real-time systems nor conventional database management systems, however, support this kind of application adequately. Real-time systems do take temporal specifications into account, but they do not consider consistency constraints of the data objects they use. On the other hand, conventional database systems do not stress the notion of timing constraints or deadlines with respect to transactions. The performance goal of conventional database systems is usually expressed in terms of minimizing *average* response time, which is relatively unimportant to real-time systems. These inadequacies of conventional systems for time-critical applications introduce the need for *real-time database systems.*

A **real-time database system** (RTDBS) can be defined as a database system where transactions are associated with explicit timing constraints, such as the worst-case response time requirements (i.e., *deadlines*) and the maximum temporal distance requirements between the accessed data objects. The correctness of a real-time databases system depends not only upon the logical results but also upon the time at which the results are produced. Transactions in the system must be scheduled in such a way that they can be completed before their corresponding deadlines expire as well as satisfy database consistency constraints.

In the last few years, the area of real-time databases has attracted attention from researchers in both real-time systems and database systems fields. The motivation of the database researchers has been to bring to bear many of the ideas of database technology to solve problems in managing data in real-time systems. Real-time system researchers have been attracted by the opportunity that real-time database systems can provide for building a complex real-time system with lots of time-sensitive data [57]. Although some of the necessary research has been done, many issues remain, especially seamless integration of real-time systems and the state-of-the-art database systems technology.

The design and implementation of RTDBS introduces several interesting problems. For example, what is an appropriate model for real-time transactions and data? What language constructs can be used to specify real-time constraints? What is the best concurrency control scheme that handles real-time constraints and the importance of transactions? Is serializability a useful correctness criterion for RTDBS? In this thesis, we focus on two issues, *predictability* and *consistency*, which are fundamental to real-time transaction processing, but sometimes require conflicting actions. To ensure consistency, we may have to block certain transactions. Blocking these transactions, however, may cause unpredictable transaction execution and may lead to the violation of timing constraints.

In the following sections, we discuss the characteristics of data and transactions in

RTDBS and the requirements of RTDBS, especially focusing on predictability and consistency issues. In this context, we identify the limitations of the previous research efforts in this field, set the goal of our research, and outline our approach to achieving this goal.

1.1 Characteristics of Real-Time Database Systems

Real-time systems in general try to meet the timing constraints of individual *tasks*, but may ignore data consistency problems. Tasks in real-time systems and transactions in RTDBS are similar abstractions in the sense that both are units of work as well as units of scheduling. However, tasks and transactions are different computational concepts, and their differences affect how they should be processed. In real-time task scheduling, it is usually assumed that all tasks are preemptable. Preemption of a transaction that uses a file resource in an exclusive mode of writing, however, may result in subsequent transactions reading inconsistent information. In addition, while the run-time behavior of a task is statically predictable, the behavior of a transaction is dynamic, making it difficult to predict its execution time with accuracy.

Conventional database systems are not used in real-time applications, because of their poor performance and their lack of predictability. In conventional database systems, the response time of a transaction is often affected by the slow and unpredictable disk access delay. Since real-time systems are often used in safety-critical applications, an unpredictable system can do more harm than good under abnormal conditions. There are other reasons why traditional database systems may have unpredictable performance. For example, to ensure the data consistency, traditional database systems often block certain transactions from reading or updating data if these data are locked by other transactions. It is difficult for a transaction to predict how long the delay will be since there may be cascaded blockings when the blocking transactions themselves are blocked by other transactions. Consequently, the response time of a transaction in conventional database systems is often unpredictable.

In the following subsections, we further investigate the characteristics of data and transactions in RTDBS.

1.1.1 Real-Time Data

Since real-time systems are used to monitor and to control physical devices, they need to store a large amount of information about their environments. Such information includes input data from devices as well as system and machine states. In addition, many embedded systems must also store the system execution history for maintenance or error recovery purposes. Some systems may also keep track of system statistics like average system load or average device temperature. Depending on the applications, real-time systems may have to handle multi-media information like audio (for sonar devices), graphics (for radar devices), and images (for robots). Since systems are constantly recording information, data must have their temporal attributes recorded. Also, some input devices may be subject to noise degradation and need to record the quality of the attributes along with the data.

Often a significant portion of a real-time database is highly perishable in the sense that it may contribute to a mission only if it is used in time. In addition to deadlines, therefore, other kinds of timing constraints could be associated with data in RTDBS. For example, each sensor input could be indexed by the time at which it was taken. Once entered into the database, data may become out-of-date if it is not updated within a certain period of time. To quantify this notion of "age," data may be associated with a *valid lifespan*. Data outside its valid lifespan does not represent the current state. What occurs when a transaction attempts to access data outside its valid lifespan depends on the semantics of data and the particular system requirements.

1.1.2 Real-Time Transactions

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 affected by the violation of timing constraints. For a hard deadline application, missing a deadline is equivalent to a catastrophe. In general, a large negative value is imparted to the system if a hard deadline is missed. For firm or soft deadline applications, however, missing deadlines leads to a performance penalty but does not lead to catastrophic results.

This categorization has also been applied to real-time database systems: *hard* deadline transactions are those which may result in a catastrophe if the deadline is missed and *soft* deadline transactions have some value even after their deadlines.[1]

However, there has been no research work which includes both types of transactions. Most previous work assumes only soft or firm deadline transactions in a real-time database system [1, 23, 25, 28, 52]. In such systems, the transaction scheduler is usually supposed to have no idea about a transaction's computing time and resource requirement in advance. Their justification can be stated as follows [70, 25]:

> Database systems for efficiently supporting hard deadline real-time applications, where all transaction deadlines have to be met, appear infeasible. This is because there is usually a large variance between the average case and worst case execution time of a transaction. The large variance is due to transaction's 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.

We agree that, under the conventional database management system's architecture and operating system environment, it is almost impossible to estimate the worst case behavior of a transaction correctly. Some real-time transaction scheduling algorithms [1, 59] that utilize *a priori* knowledge about transactions are unrealistic.

However, there are many potential applications of hard real-time database systems in the real world, such as flight control systems and missile guidance systems. For those applications, a real-time database system must provide mechanisms to minimize the execution time variance of a transaction, making the system's behavior predictable. There has been some work dealing with hard deadline transactions [59, 66, 4, 51], but impractical assumptions have often been made (for example, all transactions are periodic and their

[1] *Firm* deadline transactions are special cases of soft deadline transactions which have no value after their deadlines.

worst-case execution times are given) or different correctness and performance criteria for hard real-time transactions have been used in each work.

We believe that this conventional categorization of real-time applications is not enough for real-time database applications. Some of the terms used in conventional real-time systems should be redefined or clarified in the context of RTDBS, since the latter have quite different characteristics from the former. First of all, the term “hard” should have different semantics in RTDBS from that in conventional real-time systems, since RTDBS must satisfy not only the timing constraints of transactions but also the consistency constraints of real-time data. For instance, what if a real-time transaction meets its hard deadline but has lost the validity of the accessed data (i.e., the result of the transaction does not meet the given consistency constraints of real-time data)?

In this research, we define a *hard real-time transaction* as a transaction that has a hard response-time requirement and strict temporal data consistency constraints. The RTDBS must guarantee that both timing and consistency requirements of a hard real-time transaction are always met, since a failure to meet those hard requirements will lead to a system failure.

However, it would be extremely difficult, if not impossible, to dynamically satisfy both requirements at the same time. One possible approach to this problem is to associate temporal data consistency constraints with timing constraints. That is, one can determine the deadline of a real-time transaction so that once the deadline is met, the temporal data consistency is maintained. Note that in most research work on real-time transaction processing, deadlines are usually determined by considering only execution-time and/or response-time requirements.

A *soft/firm real-time transaction* is defined as a transaction which does not have critical timing constraints but has less or no value if it does not meet those constraints. But it must still maintain data consistency constraints.

Furthermore, there may be some real-time database applications which cannot be put into either hard or soft/firm real-time transaction category. Consider a transaction that has a critical response-time requirement but cannot be guaranteed to meet the deadline due to its indeterministic data access behavior or unknown data requirement. This kind of transaction can be regarded as neither hard nor soft. We call such transactions *critical* real-time transactions. To accommodate these transactions, we introduce the concept of *guarantee level* for real-time transactions. The guarantee level of a real-time transaction is determined by the degree of criticality of its timing constraints and has a value between 0 and 1. If the guarantee level of a transaction is x, the system should meet its timing constraints with $100 * x$ % probability. According to this definition, the guarantee levels of hard and soft real-time transactions are 1 and 0, respectively.

1.1.3 Real-Time Database Applications

A *real-time database system* is often defined as a database system where transactions are associated with real-time constraints, typically in the form of deadlines. However, with this definition, it is not clear what its applications are.

Unlike conventional general-purpose computing systems, it is extremely difficult, if not impossible, to develop general-purpose real-time computing systems that can be used for all kinds of time-critical applications, since each real-time application has different characteristics and performance requirements. As a consequence, a real-time system is often designed and configured for a specific type of application to achieve the desired performance. Otherwise, the system must be adaptable, providing a variety of options to process different kinds of applications.

There are two major categories of applications for which a real-time database system can be used: first, real-time process control systems which manage large amounts of real-time data, and second, information management systems in which at least some transactions have deadlines. Each type of application has totally different characteristics from the other. For example, a real-time transaction in a process control system often has hard timing constraints, accesses highly perishable and predefined set of data objects, requires only simple database functions, and arrives with a fixed period, while a real-time transaction in an information management system usually has a soft deadline, may request highly complex queries, and arrives aperiodically.

Even though there has been a considerable amount of research work done in the real-time database area so far, no research work deals with both types of transactions in a single application. Some of them assume process control systems having highly perishable data and hard deadline transactions as their target applications. Others consider full-fledged information management systems supporting only soft or firm deadline transactions. As the applications of real-time systems getting large and complex, more systems need to support both hard and soft real-time constraints in an integrated manner.

The real-time database systems model to be presented in the following chapter supports various types of real-time transactions, including both hard and soft real-time transactions.

1.2 Requirements of Real-Time Database Systems

1.2.1 Predictability and Timeliness

Real-time computing is not equivalent to *fast* computing [69]. There are more important properties of RTDBS than speed: *timeliness*, i.e., the ability to produce expected results early or at the right time, and *predictability*, i.e., the ability to function as deterministically as necessary to satisfy system specifications, including timing constraints. Fast computing which is busy doing the wrong activity at the wrong time is not helpful for real-time computing. Fast is helpful in meeting stringent timing constraints, but fast alone does not guarantee timeliness and predictability.

Since the performance requirements may be different for each class of real-time applications, the term *predictability* should be interpreted in a specific context. A hard real-time system must be predictable in the sense that we should be able to know beforehand whether its tasks will complete before their deadlines. This prediction will be possible only if we know the worst-case execution time of a task and its data and resource needs. However, in real-time database applications, it is not always possible to get such information in advance, since, unlike the conventional real-time applications, there are more factors that contribute to the unpredictability of transaction execution in database systems, such as interactions with indeterministic subsystems, data dependence of transaction execution, data and resource conflicts among transactions, and conventional recovery mechanisms [55].

Consequently, we cannot give this kind of predictability to all real-time transactions. Instead, we should provide a different level of guarantee to each class of real-time transactions. For example, for a group of real-time transactions, we should be able to *predict* what percentage of the transactions will meet the deadlines statistically.

Predictability is also important for soft real-time transactions, albeit to a lesser degree. For instance, if the execution-time requirement of a transaction is available in advance, then under some scheduling policies designed for on-line scheduling, a system may provide an earlier feedback on whether the transaction can be completed before its deadline. This allows the system to discard *infeasible* transactions (i.e., transactions which may not complete before their deadlines) even before they begin execution so that wasted computations, aborts, and restarts can be avoided [1, 39].

1.2.2 Correctness Criteria

Another limitation of current real-time transaction scheduling algorithms is that most of them rely on serializability to preserve the logical consistency of the database, but they fail to address how to maintain temporal consistency of real-time data.

To facilitate more timely executions of transactions to meet their deadlines, we may extend the definition of correctness in database systems. Since real-time systems are used to respond to external stimuli (e.g., in combat systems) or to control physical devices (e.g., in auto-pilot systems), a timely and useful result is much more desirable than a serializable but out-of-date response. As long as the result of a transaction is consistent with the situations of the real world, whether or not the database is internally consistent may not be important to the application. Depending on the semantics and requirements of data and transactions, a RTDBS may apply different correctness criteria under various situations.

1.2.3 Operating System and Architectural Support

Most research efforts on real-time database systems have concentrated on developing and evaluating real-time transaction scheduling algorithms, including priority assignment, disk I/O scheduling, concurrency control, and conflict resolution schemes. The primary goal of those research efforts is to minimize the deadline miss ratio of transactions [1, 39, 7, 23, 26, 25, 29, 28, 52]. However, less attention has been paid to architectural and operating system aspects of the system which support the predictable behavior of a real-time transaction. Without adequate support from the underlying subsystems, none of the scheduling algorithms can guarantee predictable transaction performance. The major difficulty is the lack of a reasonable paradigm for cooperation between real-time operating systems and real-time database management systems. The result is a duplication of some common services, not only leading to a degradation in system performance, but also making the system even more unpredictable. Clearly, this is intolerable for real-time applications. Real-time database building blocks must be integrated with the real-time operating system kernel and other run-time environment building blocks in order to avoid wasteful duplication and provide predictable services.

1.3 Research Objective and Approach

Our research is motivated by the limitations of the previous real-time database systems research as described above and intended to address some of the important issues on a RT-DBS's development which have not been investigated thoroughly in the previous research. The goal is to provide a framework for predictable real-time transaction processing which also maintains consistency of real-time data objects.

Our approach to achieving this goal is as follows: First, we analyze the characteristics of data and transactions in typical real-time database applications and categorize them into several classes, specifying different assumptions and requirements for each class. Second, we develop an integrated real-time transaction processing scheme which can guarantee the given performance requirements of each real-time application and maintain the consistency requirements of a real-time database.

Note that our model depends heavily on the deterministic behavior of transaction processing. In our framework, we try to eliminate or avoid the sources of unpredictability in a database system, which have made a hard deadline guarantee infeasible. What we need is to make the execution time of a transaction (pure computation and data access time) *deterministic* either by minimizing the variance between its worst-case and average-case execution times or by removing the sources of the variance. We observe that hard real-time database systems become feasible only when the worst-case execution times of hard real-time transactions are available. If the desired level of predictability can be achieved, the performance of soft or firm real-time transactions can also be significantly improved.

In the following, we identify a number of factors which contribute to the unpredictable execution of transactions and present our approaches to dealing with the problems.

Interaction between DBMS and the subsystem

Transactions interact with the operating system and I/O subsystem in unpredictable ways. Since transactions require concurrency control, commit protocols, recovery protocols, buffers, and access to disks, it is virtually impossible to predict the response time of a transaction under the conventional database system architecture. Also, response time is usually long due to these complicated protocols and disk access times. Unpredictability occurs from operating system features such as paging, working sets, dynamic adjusting of priorities, disk scheduling algorithms, buffering schemes, and blocking over resource contention.

Furthermore, many of the functions of a DBMS (e.g., scheduling and resource management) are usually considered parts of an operating system. Real-time applications cannot afford the duplication of these functions. For example, a DBMS application written in Ada might find its execution controlled by the operating system process scheduler, the Ada runtime task scheduler, the DBMS transaction scheduler, the DBMS resource (lock) manager, the operating system virtual memory manager, the DBMS buffer manager, the disk scheduler, and the DBMS recovery (log) manager.

Our approach to the above problem is to utilize a real-time microkernel architecture combined with memory resident real-time data objects. In contrast to the traditional monolithic operating system kernel, a microkernel provides system servers with generic services independent of a particular operating system, including (real-time) scheduling of one or more processors, memory management, and a simple IPC (Interprocess Communication) facility [20, 21, 76, 73].

This combination of elementary services forms a standard base which can support all other system-specific functions. These system-specific functions can then be configured into appropriate system servers managing other physical and logical resources of a computer system, such as real-time memory objects, devices, and high-level communication services. Such an architecture contributes to make the behavior of each service component of a real-time database system more predictable and analyzable.

As semiconductor memory becomes cheaper and chip densities increase, it becomes feasible to store larger and larger databases in memory. Memory residency of database is especially important for hard real-time applications where transactions have to complete by their specified deadlines. Using main memory as the primary repository of the database eliminates the problems due to dynamic I/O, such as buffer management, dynamic paging, and disk scheduling, and offers considerably improved performance over conventional disk-based DBMS.

However, structures and algorithms designed for conventional main memory databases must be reconsidered for real-time databases, since their performance goal (minimizing the average response time) is different from that of real-time database systems (minimizing the worst-case execution time).

Data dependence of transaction execution

Since a transaction's execution path can depend on the state of the data items it accesses, it may not be possible in general to predict the behavior of a transaction in advance. However, in some hard real-time environments, we can assume "canned" transactions and queries whose read/write sets can be predicted or predeclared beforehand. Since a real-time database is generally used in a closed loop situation where the environment being controlled closes the loop, the data items accessed by a transaction are likely to be known *a priori* once its functionality is known. Furthermore, with the support of a specialized hardware (e.g., content addressable memory), there is a potential to develop a predictable database access and query processing mechanism regardless of the state of the database.

In case the above statements are not valid for some critical transactions, at best we can provide a partial guarantee on their timing constraints.

Data and resource conflicts among transactions

Since a typical transaction dynamically acquires the data items it needs, it may be forced to wait until a data item is released by other transactions currently using it. Similarly, a transaction may be forced to wait for other resources, such as the CPU and I/O devices, to become available. While both of these blocking problems have their counterparts in real-time systems, the problems are exacerbated in real-time database systems due to data consistency requirements. Consider a database that employs a strict two phase locking protocol for concurrency control. In this case a transaction may wait for an unbounded amount of time, when it attempts to acquire a data item. The cumulative delays can be very long and unpredictable, given the possibility of deadlocks and restarts.

Conflict avoiding data access protocols and pre-allocation of resources can reduce the effects of this problem. Many such protocols have been developed in the context of real-time systems but they do not apply directly to real-time database systems. However, we can prevent the unbounded blocking of a transaction (e.g., priority ceiling protocol [59]), assuming that *a priori* knowledge about transactions is available, which is often the case in hard real-time applications, or avoid the blocking by allowing non-serializable schedule, which is acceptable for some types of real-time transactions.

Conventional recovery mechanisms

In conventional database systems, recovery mechanisms after failure often depend on logging and checkpointing. However, those actions often conflict with normal transaction processing, making a transaction's response time unpredictable. Also, transaction aborts and the resulting rollbacks and restarts not only increase the total execution time for the involved transaction, but also affect other ongoing transactions. This kind of backward recovery mechanism should not be used in hard real-time applications. Fortunately, we observe that it is meaningless to blindly roll back to an earlier consistent state of a real-time database since the prior values in the database could be out-of-date and useless anyway. Abnormal termination of a real-time transaction should not always require rolling back the changes.

We do not consider the failure situation and the database recovery issues in our RTDBS model, to be presented in Chapter 3, but the issues related with the real-time recovery in memory-resident database systems will be discussed in Chapter 7.

1.4 Contributions of the Thesis

The major contributions of this thesis can be summarized as follows:

A new RTDBS model. Our model classifies real-time data and transactions, considering their attributes and the application semantics and requirements. It includes both hard and soft real-time transactions, and supports the temporal consistency as well as the logical consistency of a database. The conventional RTDBS models used in the previous researches are only subsets of our model.

Predictable transaction processing. We have extended a fixed-priority-based method to jointly schedule both hard periodic tasks and soft aperiodic tasks into a transaction processing environment, and integrated it with a *best-effort* real-time transaction scheduling algorithm to come up with a predictable transaction processing scheme. Under this scheme, three different levels of performance can be specified and achieved in one system, in terms of guarantee on the timing constraints, depending on the characteristics and requirements of transactions:

- total guarantee (or 100% guarantee)
- partial guarantee (or statistical guarantee)
- no guarantee (or best effort)

This is not the case in the conventional RTDBS, where users only can express the relative importance of a transaction in terms of the priority and the scheduling algorithms only can make a best effort with no guarantee on its individual timing constraints.

Static temporal consistency enforcement. In order to maintain temporal consistency of real-time data and transactions, we have developed a systematic scheme to determine some attributes of real-time transactions, such as periods, deadlines, and priorities. Under this scheme, the temporal consistency requirements of RTDBS are always fulfilled as long as the deadlines of the related transactions are always met. Other approaches are different from ours in that they do not provide a guarantee on temporal consistency, but just make a best effort to minimize the number of temporal consistency violations.

Semantic-based concurrency control. In real-time databases, *serializable schedules* are not always required for transactions to maintain consistency of the database. Utilizing the inherent semantic information about transactions in specific classes, a real-time transaction manager can make different control decisions for different classes of transactions, in order to maximize the level of concurrency while maintaining both timing and consistency constraints of the system. We have developed a semantic-based concurrency control and conflict resolution scheme under our real-time database model. Comparing to concurrency control algorithms being used in conventional database systems, our scheme can make more efficient control decisions when conflicts occur, since it can make use of certain semantic knowledge about transactions which is given *a priori*.

Cost and performance evaluation. We have evaluated the cost and performance of our integrated real-time transaction processing system, comparing to the conventional best-effort system, through a simulation study. Especially, we illustrated the cost of timing constraint guarantee, the cost of temporal consistency guarantee, and the cost of logical consistency guarantee in our system, under the various situations.

Deterministic subsystem model. To support our RTDBS model, we suggest a deterministic computing environment for a real-time database system, utilizing a memory-resident real-time data object abstraction and a real-time microkernel architecture. The proposed architecture eliminates sources of unpredictable behavior in the system related to dynamic I/O, such as buffer management, dynamic paging, and disk scheduling.

Real-time query processing and recovery mechanism. In the context of memory-resident real-time databases, query processing and recovery issues have been discussed and several possible approaches to the problems have been proposed, but without evaluation.

1.5 Organization of the Thesis

The remainder of this thesis is organized as follows.

In Chapter 2, we review the current status of real-time database systems research and related areas, and discuss the problems of the current technology used in traditional database management systems in the context of a real-time database system.

In Chapter 3, we introduce a new model for real-time data and transactions. We come up with three types of data objects and five classes of transactions in RTDBS, based on their characteristics and requirements.

In Chapter 4, we develop a predictable transaction processing scheme under the proposed model, which satisfies the temporal consistency requirements of real-time data objects as well as the response-time requirements of real-time transactions.

In Chapter 5, our model and the supporting schemes to achieve predictable real-time transaction processing are evaluated by simulation. The main result is that the guarantee on hard real-time transactions can be achieved at the reasonable expense of soft real-time transactions.

In Chapter 6, we discuss some practical issues which may be encountered when a real-time database system is developed based on our model and our transaction processing algorithms are to be implemented on top of it. The current structure and future extensions of the StarBase system, an experimental real-time database system testbed, are also presented.

In Chapter 7, we give some lights on other issues in RTDBS, such as real-time query processing and recovery, which are important but have not been addressed in the previous RTDBS research.

Finally, in Chapter 8, we summarize the results of our research and discuss how this work has contributed to the field of real-time transaction processing. Also, we explore future directions in which our research can be extended.

Chapter 2

Background and Related Work

Real-time database systems require research efforts on various aspects of real-time systems and database management systems. However, since this research field has received attention only in the last few years, research work done so far does not address all the issues necessary for developing a practical real-time database system.

In this chapter, we review the current status of real-time task and transaction scheduling research, as well as other related areas, and discuss the problems of the current technology used in traditional database management systems in the context of real-time database support.

2.1 Real-Time Task Scheduling

Over the last decade, scheduling methods have been introduced which allow for the design of real-time systems with predictable timing correctness. Moreover, these methods have become sufficiently advanced so that many practical problems associated with these systems have been addressed successfully. The most complete theoretical results have been for the situation in which the system must process a significant number of periodic tasks, for example, tasks associated with monitoring in control systems. For this case, there are two popular approaches: (1) static or fixed-priority algorithms, including the rate-monotonic and deadline-monotonic algorithms [53, 3] and (2) dynamic priority algorithms, including the earliest deadline algorithm [53]. Both approaches are becoming increasingly well developed, although at the present time the static priority theory is much more complete. A summary of the results available on fixed-priority scheduling can be found in review articles by Burns [8] and Lehoczky [47].

Given the success of fixed priority scheduling methods, it is natural to attempt to extend this theory to solve other important problems that arise in real-time systems. This includes problems such as scheduling hard deadline sporadic tasks and simultaneously scheduling hard deadline periodic tasks along with soft deadline aperiodic tasks. In this section, we review recent development in scheduling theory designed to determine a fixed-priority-based method to accommodate sporadic tasks and jointly schedule tasks with both hard and soft time constraints.

Scheduling Sporadic Tasks

Non-periodic tasks are those whose releases are not periodic in nature. Such tasks can be subdivided into two categories [8]: *aperiodic* and *sporadic*. The difference between these categories lies in the nature of their release frequencies. Aperiodic tasks are those whose release frequency is unbounded. In the extreme, this could lead to an arbitrarily large number of simultaneously active tasks. Sporadic tasks are those that have a maximum frequency such that only one instance of a particular sporadic task can be active at a time.

When a static scheduling algorithm is employed, it is difficult to introduce non-periodic task executions into the schedule: it is not known before the system is run when non-periodic tasks will be released. More difficulties arise when attempting to guarantee the deadlines of those tasks. It is clearly impossible to guarantee the deadlines of aperiodic tasks as there could be an arbitrarily large number of them active at any time. Deadlines of sporadic tasks can be guaranteed since it is possible, by means of maximum release frequency, to define the maximum workload they place upon the system.

One approach is available to guarantee the deadlines of sporadic tasks without resorting to the introduction of polling servers within the existing deadline-monotonic theory. Consider the timing characteristics of a sporadic task τ_s, illustrated in Figure 2.1. The minimum time difference between successive releases of τ_s is the minimum inter-arrival time M_s. This occurs between the first two releases of τ_s. At this point, τ_s is behaving exactly like a

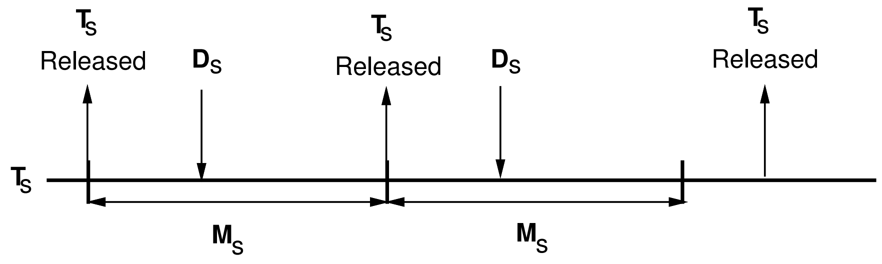


Figure 2.1: Sporadic Arrivals of a Task

periodic task with period M_s: the sporadic is being released at its maximum frequency and so is imposing its maximum workload. When the releases do not occur at the maximum rate (between the second and third releases in Figure 2.1) τ_s behaves like a periodic task that is intermittently activated and then laid dormant. The workload imposed by the sporadic is at a maximum when the task is released, but falls when the next release occurs after greater than M_s time units have elapsed.

In the worst-case the τ_s behaves exactly like a periodic task with period M_s and deadline D_s where $D_s \leq M_s$. The characteristic of this behavior is that a maximum of one release of the task can occur in any interval $[t,\, t + M_s]$ where release time t is at least M_s time units after the previous release of the task. This implies that to guarantee the deadline of the sporadic task the computation time must be available within the interval $[t,\, t + D_s]$ noting that the deadline will be at least M_s after the previous deadline of the sporadic. This is exactly the guarantee given by the original deadline-monotonic scheduling theory.

For schedulability purposes only, the sporadic task can be regarded as a periodic task whose period is equal to M_s. However, we note that since the task is sporadic, the actual release times of the task will not be periodic, but successive releases will be separated by no less than M_s time units.

For the schedulability tests given in [8] to be effective for this task set, it is assumed that at some instant all tasks, both periodic and sporadic, are released simultaneously (i.e., a *critical instant*). If the deadline of the sporadic can be guaranteed for the release at a critical instant then all subsequent deadlines are guaranteed. No limitations on the combination of periodic and sporadic tasks are imposed by this scheme. Indeed, the approach is optimal for a fixed-priority scheduling since sporadic tasks are treated in exactly the same manner as periodic tasks. To improve the responsiveness of sporadic tasks, their deadlines can be reduced to the point at which the system becomes unschedulable.

Scheduling Both Hard and Soft Deadline Tasks

Analysis of fixed-priority preemptive scheduling has provided a sound theoretical basis for designing predictable hard real-time systems. Within this framework, a number of approaches have been developed for scheduling mixed task sets. The simplest and perhaps least effective of these is to execute soft deadline tasks at a lower priority level than any of those with hard deadlines. This effectively relegates the soft tasks to background processing. Alternatively, soft tasks may be run at a higher priority under the control of a pseudo hard real-time server task, such as a simple polling server.

A polling server is a periodic task with a fixed priority level (usually the highest) and an execution capacity. The capacity of the server is calculated off-line and is normally set to the maximum possible, such that the hard task set, including server, is schedulable. At run-time, the polling server is released periodically and its capacity is used to service soft real-time tasks. Once this capacity has been exhausted, execution is suspended until it can be replenished at the server's next release.

The polling server will usually significantly improve the response times of soft tasks over background processing. However, if the ready soft tasks exceed the capacity of the server, then some of them will have to wait until its next release, leading to potentially long response times. Conversely, no soft tasks may be ready when the server is released, wasting its high priority capacity.

The latter drawback is avoided by the Priority Exchange [68] and Deferrable/Sporadic Server [67] algorithms. These are all based on similar principles to the polling server. However, they are able to preserve capacity if no soft tasks are pending when they are released. Due to this property, they are termed “bandwidth preserving algorithms”. The three algorithms differ in the ways in which the capacity of the server is preserved and replenished and in the schedulability analysis needed to determine their maximum capacity.

In general, all three offer improved responsiveness over the polling approach. However, there are still disadvantages with these more complex server algorithms. They are unable to make use of slack time which maybe present due to the often favorable phasing of periodic tasks (i.e., not being the worst case). Further, they tend to degrade to providing essentially the same performance as the polling server at high loads. The Deferrable and Sporadic Servers are also unable to reclaim spare capacity gained when, for example, hard tasks require less than their worst-case execution time. This spare capacity, termed *gain time*, can however be reclaimed by the Extended Priority Exchange algorithm.

Slack Stealing Algorithms

The static slack stealing algorithm of Lehoczky and Ramos-Thuel [48] suffers from none of these disadvantages. It is optimal in the sense that it minimizes the response times of soft tasks among all algorithms which meet all hard periodic task deadlines. The slack stealer services soft requests by making any spare processing time available as soon as possible. In doing so, it effectively steals slack from the hard deadline periodic tasks. A means of determining the maximum amount of slack which may be stolen, without jeopardizing the hard timing constraints, is therefore key to the operation of the algorithm.

In [48], Lehoczky and Ramos-Thuel describe how the slack available can be found. This is done by mapping out the processor schedule for the hard periodic tasks over their *hyperperiod* (the least common multiple of task periods). The mapping is then inspected to determine the slack present between the deadline on one invocation of a task and the next. The values found are stored in a table. At run time, a set of counters are used to keep track of the slack which may be stolen at each priority level. These counters are decremented depending on which tasks, if any, are executing and updated by reference to the table, at the completion of each task. Whenever the counters indicate that there is slack available at *all* priority levels, soft tasks may be executed at the highest priority level.

Unfortunately, the need to map out the hyperperiod restricts the applicability of this optimal algorithm: Slack can only be stolen from hard deadline tasks which are strictly periodic and have no release jitter [2] or synchronization. Realistically, it is also limited to task sets with a manageably short hyperperiod. This is a significant restriction, as even modest task sets (e.g., 10 tasks) may have very long hyperperiods.

The limitations inherent in the optimal static algorithm are addressed by the dynamic slack stealer developed by Davis *et al* [14]. By virtue of computing slack at run time, the optimal dynamic algorithm is applicable to a more general class of scheduling problems, including task sets which contain hard deadline sporadics and tasks which exhibit release jitter and synchronization. Further, the dynamic algorithm is able to improve the response times of soft tasks by exploiting run-time information about hard task execution requirements, blocking and context switch times [13]. Unfortunately, the execution time overhead of the optimal dynamic algorithm is such that it is infeasible in practice.

Approximate methods of determining slack presented by Davis [12] address the space and time complexity problems inherent in the optimal algorithms. These approximations form the basis of various approximate slack stealing algorithms which offer close to optimal performance with practical utility. These algorithms, to be described in Appendix A, have been integrated into our real-time transaction processing environment and their performances have been compared with background processing and optimal slack stealing method in Chapter 5.

2.2 Real-Time Transaction Scheduling

Scheduling transactions with timing constraints is a more complicated 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 in real-time database systems. We can categorize the studies in this area into the following three classes, according to their assumptions on real-time transaction model:

- Transactions arrive sporadically with unpredictable arrival times and resource requirements. That is, no *a priori* knowledge of transactions is available to the scheduler. This transaction model is usually used in soft or firm real-time system environment.

- Transactions arrive in the same way as the above, but some *a priori* knowledge of transactions can be utilized by the scheduler. That is, the data requirements of each transaction are still unknown but a worst case execution time is available to the scheduler. This transaction model is also used for real-time applications with soft or firm deadlines.

- Transactions arrive periodically with known invocation times. Furthermore, data requirement and worst case execution time of each transaction are known in advance to scheduler. This model is usually assumed for hard real-time transactions. In the case of aperiodic transactions, by making use of the smallest separation time between two incarnations of an aperiodic transaction, they can be viewed just like periodic transactions. Thus, all hard real-time transactions are regarded as *periodic* in this model.

Each real-time transaction scheduling algorithm employs different priority assignment, concurrency control, and conflict resolution schemes based on one of the above transaction models.

Hard Deadline Transaction Scheduling

By the definition of *hard* deadlines, all transactions with hard deadlines must meet their timing constraints. Since dynamically managed transactions cannot provide such guarantees, the data and processing resources as well as computation time needed by such transactions must be guaranteed to be made available when necessary. We thus have to know about the transaction's invocation time, its resource requirement, and the worst case execution time in advance. This requires that many restrictions be placed on the structure and characteristics of real-time transactions.

Only a few real-time transaction scheduling algorithms using this class of transaction model have been proposed so far. Sha *et al* proposed algorithms for scheduling a fixed set of periodic transactions with hard deadlines [59, 61]. Their model assumes that transaction priorities and resource requirements are known *a priori*. The rate-monotonic algorithm is used for determining transaction priority and scheduling the CPU. A priority ceiling protocol based on locking is used for concurrency control. The priority ceiling algorithm appears to have a promise for a hard real-time environment since it prevents deadlock formation and strictly bounds transaction blocking times. The price, however, is *a priori* knowledge about data to be accessed by real-time transactions.

This condition appears to be too restrictive and even unrealistic in traditional database systems where data access is random. Moreover, the scheme becomes extremely conservative with respect to the degree of concurrency if transactions can access any data objects in the database. Nevertheless, we can hardly imagine a hard real-time system with unpredictable transaction behavior. Our approach is to make transaction's behavior predictable by providing a new real-time database model which allows non-serializable transaction executions and partial guarantees on timing constraints.

Soft or Firm Deadline Transaction Scheduling

Most of the real-time transaction scheduling algorithms assume that the transaction scheduler is supposed to have no idea about transaction's computing time and resource requirement in advance, which is the case of soft or firm deadline applications. Priority of a transaction is thus assigned based on its timing constraint (i.e., deadline) and/or value, without considering information about its runtime behavior. Also, conflict resolution schemes used in real-time concurrency control protocols do not utilize such information. Consequently, they cannot guarantee that each transaction will complete by its deadline, but try to minimize the deadline miss ratio of transactions or to maximize the total value of transactions completed by their deadlines, when transactions have different values.

This class of algorithms are *High Priority* (or *Priority Abort*) [29, 1], *Priority Inheritance* (or *Wait Promote*) [30, 28, 1] protocols with the two-phase locking scheme, *OPT-BC*, *OPT-SACRIFICE*, *OPT-WAIT*, *WAIT-50* protocols [24, 23] with optimistic concurrency control techniques, and hybrid concurrency control algorithms [52, 63].

Some soft or firm real-time transaction scheduling algorithms utilize the worst case execution times of transactions, without the knowledge of data requirements. Intuitively, this information can help the scheduler to determine eligibility of transactions, reducing the wasted CPU time and recovery overhead due to aborted transactions. In this class of algorithms, *Least Slack* priority assignment policy can be used, combined with conflict resolution policies such as *Conditional Restart* [1] and *Conditional Priority Inheritance* [30, 28] under the two phase locking concurrency control scheme.

However, some of the performance results for those algorithms are contradictory and there is no consensus on which are the best algorithms. For example, it is reported in [1] that concurrency control policies that combine blocking with priority inheritance (Wait Promote, Conditional Restart) generally perform better than a policy that aborts lower priority transactions in order to favor higher priority ones (High Priority), whereas the opposite to this result is reported in [30]. The reason for this may lie in the fact that they used different environments and operating ranges to evaluate the algorithms. The latter group used an experimental testbed for their performance results and was unable to program the disk device driver to implement priority scheduling of I/O requests, while performance evaluation of the former group was done via simulation and did incorporate priority-based disk scheduling.

Furthermore, the well-known debate regarding the performance of locking based concurrency control protocols versus optimistic ones has surfaced in the field of real-time database systems. One group of researchers has reported an optimistic concurrency control algorithm that can generally perform better than locking based protocol [24]. A different group of researchers has reported the opposite [31]. The small differences in transaction models and the large differences in evaluation techniques make it difficult to resolve these competing claims.

Although it would certainly be fruitful to develop more algorithms and evaluate them for real-time transaction scheduling, we feel that a more integrated system and transaction model is needed to support advancement in this field. Our approach is to establish a new real-time data and transaction model which accommodates a broad range of real-time database applications and to develop a predictable transaction processing algorithm for this model which is capable of achieving multiple levels of guarantee on timing constraints.

2.3 Temporal Consistency

The *consistency* of a conventional database, and the *correctness* of its transactions, are defined purely over the values stored in data objects (*logical* consistency). While it may also apply to a hard real-time database, there is an additional requirement concerning time (*temporal* consistency). The concept of "temporal consistency" suggests that the age of data should be taken into account in making scheduling decisions. Specifying bounds on the *start* time of one transaction relative to the *stop* time of one or more transactions is a new form of timing constraint [50]. Temporal consistency can reference either the absolute age of the data read by a transaction, or the age of each data item relative to the age of every other data item in the read set of a transaction [66].

Not much work has been reported on schedulers that preserve both logical and temporal consistency of real-time database at the same time. All the real-time transaction scheduling algorithms mentioned in the previous section use "serializability" as the only database correctness criteria.

Liu and Song [66] apply temporal consistency along with serializability as a criterion for correctness. This is consistent with the observation that the correct operation interleavings for real-time transactions are those serializable interleavings which meet their timing constraints. However, although this notion of temporal consistency was used to judge the effectiveness of scheduling algorithms, their multiversion concurrency control algorithms do not guarantee the temporal consistency of real-time data objects. Audsley *et al* [4] characterize real-time data objects and derive some temporal consistency requirements, but they do not show how a transaction scheduler can realize such requirements.

Different from the above approaches, in our research different consistency criteria are applied for different types of data objects. For example, for some data objects whose values are rapidly changing over time, the logical consistency may not be required, but the temporal consistency must be maintained. This allows non-serializable (thus non-blocking) accesses to these data objects, making a predictable maintenance of the temporal consistency requirements feasible.

2.4 Memory Resident Database Systems

Since data can be accessed directly in memory, *memory resident database systems* (MRDSs) can provide much better response times and transaction throughputs, as compared to conventional disk resident database systems (DRDSs). Furthermore, the avoidance of disk I/O operations and buffer management functions provides a potential for predictable transaction processing. This is especially important for real-time applications. Since memory prices are steadily dropping, and memory sizes are growing, memory residence of a real-time database becomes less of a restriction.

During the 1980's, a good deal of research investigated the effects of the availability of very large main memories [15, 45, 44, 46, 58, 17]. The early work considered effects on query processing strategies, data structures, and failure recovery mechanisms when a substantial percentage of the database could fit into the DBMS buffer pool. Other researchers assume that the entire database can be made main memory resident.

The research cited above suggests that algorithms specialized for memory resident databases offer considerably improved performance over conventional DBMS with very large buffer pools. Much of the work in memory resident databases concentrate on the recovery aspects of the system [46, 58, 22, 35], since in MRDSs the recovery becomes more complex than disk-based database recovery mainly due to the volatility of main memory and the elimination of the separation of data storage and data processing location. To minimize the possible interference between normal transaction processing and recovery actions like logging and checkpointing, small amounts of stable main memory and a dedicated recovery processor are usually assumed in MRDSs.

However, even the most efficient memory resident database management mechanisms proposed so far are not readily applicable to real-time database systems because they do not consider timing constraints of transactions. Access methods, query processing, and recovery in memory resident database systems must be reconsidered in the context of real-time system in order to provide predictable execution behavior. Some of these issues will be addressed in Chapter 7.

2.5 Operating System Support for Database Management

There are three major areas where the operating system (OS) supports a database management system: *persistent data management*, *buffer and memory management*, and *transaction support*.

Databases store and manage persistent data which survive past the execution of the program that manipulates them. The traditional manner in which operating systems have dealt with persistent data is by means of file systems. Even as the traditional means of supporting secondary storage of data, current file systems are not suitable for DBMS which have specific requirements regarding both logical structure of files and their physical storage. The major issue regarding the file system is the management of memory buffers in accessing data files. Some operating systems are moving toward providing file system services through virtual memory [77, 16]. In this architecture, when a file is opened, it is mapped into a virtual memory segment and then read by referencing a page in virtual memory, which is then loaded by the operating system. Similarly, a write causes a virtual memory page to become dirty and be eventually written to disk.

If virtual memory file systems are successful, then buffer management code can be removed from DBMSs, resulting in one less problem to worry about. However, in this case, OS must provide an interface with which a DBMS can give some information about its data access behavior to the virtual memory manager, since the general-purpose algorithm like LRU replacement fails to perform adequately in a number of cases. For example, in a nested-loop join, the *buffer page*[1] that contains one data page of the outer relation is the least recently used if a replacement page is needed to bring in the last data page of the inner relation. However, if the outer relation buffer page is replaced, it will be read in again immediately, causing a *buffer fault*[2] for this and every page reference from then on.

Traditionally, DBMSs have contained a significant amount of code to provide transaction management services (concurrency control and crash recovery). However, such services are only available to DBMS users and not to general clients of the operating system such as mail programs and text editors. There has been considerable interest in providing transactions as an operating system service [71, 72, 16, 18], and several OSs have done exactly that [27, 38, 11]. Moreover, if a user wishes to have a cross-system transaction (i.e., one involving both a DBMS and another subsystem) then it is easily supported with an OS transaction manager but is nearly impossible with a DBMS-supplied one.

Current attempts to implement these services in the conventional operating systems (e.g., Camelot [18]) are not adequate for real-time transaction processing systems. First, the access to the virtual memory mapped data objects is unpredictable, since a page in a data object sometimes resides in main memory buffer and sometimes not, depending on the OS paging mechanism. Second, the recovery mechanisms used in OS supported transaction managers still rely on the conventional logging and checkpointing algorithms, and use backward recovery procedure when restarted after a failure. Even the most efficient recovery mechanism used in a conventional DBMS is not appropriate to hard real-time database systems. The support for recoverable update of real-time data objects, not relying on backward recovery, is required for real-time transaction processing.

One possible solution is to build a deterministic service interface for real-time data objects on top of the real-time microkernel, such as ARTS [74], Real-Time Mach [76], or

[1]The buffer space is divided into pages of the same size, called *buffer pages*.

[2]When a process attempts to read from a file, the buffer manager first searches the buffer pages for the specific data page. If the search is unsuccessful, a *buffer fault* occurs which is serviced by bringing the data page from secondary storage and loading into a buffer page.

CHORUS [20], and then develop predictable database management functions (e.g., transaction manager, recovery manager, etc.) based on this interface. Note that making transaction execution times predictable through an adequate architecture and OS support does not guarantee that the deadline of a transaction will be met. It is the scheduling mechanism of a real-time database management system that utilizes such information and guarantees both consistency and timing constraints.

Chapter 3

Real-Time Database System Model

Most real-time database scheduling algorithms have been developed and evaluated under almost the same workload and operating environment model used in conventional database systems [1, 25, 28, 64]. That is, transactions are assumed to arrive in a Poisson stream at a specified mean rate. Each transaction consists of a random sequence of pages to be read, a subset of which are updated. In addition, a conventional disk-based database environment is assumed. The general approach is to utilize existing concurrency control protocols, especially two-phase locking, and to apply time-critical transaction scheduling methods that favor more urgent transactions [62]. While this model is suited to some real-time database applications with soft or firm deadlines (e.g., airline reservation system, telephone directory service system, etc.), typical hard real-time database applications do not fit into this model, since they require predictable execution of transactions and semantically consistent data which may not satisfy serializability.

3.1 Real-Time Data Object Model

In our model, a real-time database consists of a set of data objects representing the state of an external world controlled by a real-time system. There are two types of data objects in a RTDBS: *continuous* and *discrete*.

Continuous data objects are related to external objects that are continuously changing in time. There are two types of continuous data objects: one is an *image object* whose value is obtained directly from a sensor and the other is a *derived object* whose value is computed from the values of other data objects with a regular period. Discrete data objects are static in the sense that their values do not become obsolete as time passes, and they remain valid until update transactions change the values.

Different from non-real-time data objects found in traditional databases, continuous data objects are related with the following additional attributes:

- A *timestamp* tells when the current value of the data object was obtained.
- An *absolute validity duration* is the length of time during which the current value of the data object is considered to be valid. The value of a continuous data object x achieves *absolute temporal consistency* (or *external consistency*) only when $t_{now} - t_x \leq avd_x$, where t_{now} is the current time, t_x is the timestamp of x, and avd_x is the absolute validity duration of x.
- A *relative validity duration* is associated with a set of data objects Σ_y used to derive a new data object y. Such a set Σ_y has a *relative temporal consistency* when the timestamp difference (or *temporal distance*) between the data object y and any data object in the set is not greater than the relative validity duration rvd_y. The value of a derived object y has *temporal consistency* only when all the values of data objects in Σ_y are externally consistent and Σ_y satisfies relative temporal consistency.

A continuous data object is in a *correct state* if and only if the value of the object satisfies both absolute and relative temporal consistency, while a discrete data object is in a correct state as long as the value of the object is logically consistent (i.e., satisfies all integrity constraints).

Observe that there is only one writer for each continuous data object and that its value can be used as long as it maintains temporal consistency. Thus, serializability and recoverability of transactions, on which most conventional databases depend to maintain their correctness, may not be necessary for these kinds of data objects.

Let's denote that a real-time database $\mathcal{R}$ consists of the following data objects:

1. A set of image objects $X = \{x_1, x_2, \ldots, x_n\}$,

2. A set of derived objects $Y = \{y_1, y_2, \ldots, y_m\}$, and

3. A set of discrete data objects $Z = \{z_1, z_2, \ldots, z_l\}$.

A set of data objects which is used to compute the value of a derived object y is denoted as $\Sigma_y = \{\sigma_1, \sigma_2, \ldots, \sigma_k\},\ \sigma_i \in X \cup Y \cup Z,\ 1 \leq i \leq k$.

We will use this notations throughout the thesis.

3.2 Real-Time Transaction Model

Generally, a real-time transaction τ has the following attributes:

1. Arrival time (a_τ)
2. Periodicity: a period (P_τ) if periodic, or a minimum inter-arrival time (M_τ) if sporadic
3. Timing constraints: Deadline (D_τ)
4. Priority (p_τ)
5. Execution time requirement (C_τ)
6. Data requirement: Read set (RS_τ) and Write set (WS_τ)
7. Criticalness (w_τ)
8. Value function ($v_\tau(t)$)

Based on the values of the above attributes, the availability of the information, and other semantics of the transactions, a real-time transaction τ can be characterized as follows:

1. Implication of missing deadline D_τ: *hard*, *critical*, or *soft* (*firm*) real-time
2. Arrival pattern: *periodic*, *sporadic*, or *aperiodic*
3. Data access pattern: predefined (*write-only*, *read-only*, or *update*) or *random*

4. Data requirement: *known* or *unknown*
5. Runtime requirement (pure processor and data access time): *known* or *unknown*
6. Accessed data type: *continuous*, *discrete*, or *both*

We believe that if a RTDBS utilizes the unique characteristics of real-time data and transactions, it can make more efficient decisions in processing transactions and improve overall system performance. Considering the above characterization of real-time data and transactions, there are hundreds of possible transaction classes. However, some of them are infeasible (e.g., a hard real-time transaction with random arrival pattern, random data access set, and unknown execution time), and others can be grouped together to be processed differently. In our model, a typical real-time database application consists of the following classes of transactions:

Class I Transactions

This class includes all the *hard periodic* real-time transactions whose data and computation requirements are supposed to be available in advance. Furthermore, Class I transactions write only continuous data objects which require temporal consistency as their sole correctness criteria. It is thus feasible to guarantee their hard timing constraints using an appropriate scheduling algorithm.

This class can be further divided into three subclasses according to the semantic information of transactions:

Class IA Transactions. A transaction in this class is responsible for maintaining the *absolute temporal consistency* of the database by writing a sampled value of an external object to the corresponding image object with a regular interval (i.e., $WS_\tau \subset X$). It is a *write-only* (i.e., $RS_\tau = \emptyset$) transaction. We assume that the transaction is the only writer to the corresponding image object (*single-writer* property). Thus, there is no *write-write conflict*[1] between a Class IA transaction and any other transactions.

[1]a conflict between any two transactions τ_x and τ_y if $WS_{\tau_x} \cap WS_{\tau_y} \neq \emptyset$

Class IB Transactions. Transactions of this class read some data objects (mainly, continuous data objects), compute new values of derived objects, and write them to the database (i.e., $RS_\tau \subset X \cup Y \cup Z, \;\; WS_\tau \subset Y$). They do not conflict with other Class I transactions, since there can be only one writer for a derived object. Note that in order to strictly maintain the correctness of the derived objects written by a Class IB transaction, the values of data objects read by the transaction must be in a correct state until the completion of the next instance of the transaction. However, we may be able to relax this condition to be less conservative, depending on the semantics of a specific application.

Class IC Transactions. Class IC transactions periodically retrieve the values of data objects (i.e., $RS_\tau \subset X \cup Y \cup Z$) and send either some control decisions to actuators or the retrieved data to display monitors. They are *read-only* transactions (i.e., $WS_\tau = \emptyset$) with hard deadlines. They have different validity requirements from Class IB transactions: the values of data objects read by a Class IC transaction must be in a correct state until the transaction is completed.

Class II Transactions

Transactions of this class are *read-only* transactions with some critical timing constraints. Their timing constraints come from response-time requirements of the transactions, not from the attributes of data. Different from the Class IC transactions, they are not necessarily *periodic*, and their run-time estimates and read sets are not always available in advance. Also, their read sets may contain some discrete data objects which require serializable accesses. For this reason, we cannot always guarantee that a Class II transaction will meet its deadlines. This is the transaction class in which each transaction should have a different *guarantee level* as its performance requirement.

The main idea to achieve the specified guarantee level of a Class II transaction is to reduce the sources of unpredictability down to one dimension, making the transaction execution time a function of only one variable (i.e., an indeterministic transaction attribute such as data requirement). We assume that each transaction of this class has a specified minimum inter-arrival time M_τ (i.e., *sporadic*) and a known *selectivity distribution* S_τ. If an appropriate execution-time budget for a Class II transaction is given, the specified guarantee level can be achieved under the scheduling scheme to be presented in the next chapter.

Note that a Class II transaction can be either a hard or a soft real-time transaction depending on its required guarantee level (i.e., hard real-time if the guarantee level is 1, and soft real-time if it is 0). Determining the guarantee level of a Class II transaction is entirely application-dependent.

Class III Transactions

All real-time transactions not belonging to any of the above classes can be categorized in this group. They have either soft (Class IIIA) or firm (Class IIIB) deadlines, their data and run-time requirements are not always known, and they can access both continuous and discrete data objects. Since data conflicts between Class III and Class II transactions can occur, an appropriate resolution scheme is required.

In fact, Class III transactions can be further divided into several classes and processed differently. For example, *a priori* knowledge of the attributes of a transaction is sometimes available for some soft real-time transactions and should be utilized to improve the system performance. However, we decide not to further categorize soft real-time transactions but to concentrate on Class I and Class II transactions. Much work has been already done for Class III transactions [1, 25, 28, 64]. Moreover, all Class III transactions are supposed to have the same level of importance (non-critical) and do not require an individual performance guarantee. Non-real-time conventional transactions which do not have any timing constraints and access only discrete data objects also can be included in this class as if they have *infinite* deadlines.

An interesting question to ask at this point is whether database consistency is guaranteed at all times while different classes of transactions are scheduled by different scheduling algorithms. In fact, the database may be in an inconsistent state during certain intervals of time. It is because transactions of Class IB, Class IC, and Class II are executed separately from transactions of Class IA, even though they depend on the values written by Class IA transactions. This decoupling among different classes of hard deadline transactions is the key in eliminating conflicts among hard deadline transactions. However, even though data objects may have inconsistent values at times, their values are sufficiently up-to-date to satisfy temporal consistency requirements specified by validity durations. In hard real-time database systems, guaranteeing temporal consistency requirements is far more critical than satisfying the conventional notion of serializability.

Our classification of real-time transactions is summarized in Table 3.1. There may be some exceptions in this classification: for example, aperiodic update transactions with hard deadlines and unknown data and run-time requirements. We exclude these cases from our consideration because it is not feasible to guarantee their timing constraints, and therefore they should not be hard real-time transactions. However, sometimes it is possible to transform a transaction which does not belong to any of the above classes into a set of related transactions in our classification category. For example, if there is a highly-critical transaction which must react to an aperiodic event within a given hard deadline, its function can be implemented by two Class I transactions: one is to periodically update a flag which indicates the arrival of the event (Class IA) and the other is to periodically read the flag and make an appropriate action if the flag is on (the event occurs).

Most real-time database research uses the models which include only a subset of the above classes (e.g., {Class I} [61, 4, 66] or {Class III} [1, 25, 28, 64]), and never discriminate among transactions in the system. However, in practice, all kinds of transactions can coexist in one system.

Consider a medical information system as an example: Class IA transactions are transactions which update the dynamic physical status of a critical patient from the sensor devices, such as blood pressure, heart rate, and body temperature. Transactions that write derived information from the raw data about the patient's physical status are Class IB transactions. Class IC transactions may include the transactions monitoring the physical status of the patient to provide information to life support devices. A decision-making transaction issued by a surgeon during a critical operation on a patient can be regarded as a Class II transaction. It may access not only the patient's current physical status but

Table 3.1: Classification of Real-Time Transactions

Class / Property	Class I			Class II	Class III
	A	B	C		
Timing constraints	Hard			Critical	Soft or firm
Arrival pattern	Periodic			Sporadic	Aperiodic
Data access pattern	Write only	Update	Read only	Read only	No restriction
Data requirement	Known			Unknown	Unknown
Runtime requirement	Known			Unknown	Unknown
Updated data type	Image	Derived	N/A	N/A	Discrete
Correctness criteria	Temporal consistency			Both	Logical consistency
Transaction schedule	Non-serializable			Both	Serializable
Performance goal	100% guarantee			Statistical guarantee	No guarantee, but best-effort

also his or her medical history. Conventional record-keeping transactions on patient data, such as retrieving and updating their weights and heights, can be classified as Class III transactions.

Chapter 4

Predictable Transaction Processing

Our model for a RTDBS supports all the transaction classes discussed in the previous chapter. The performance goal of such a RTDBS is first to guarantee all the hard timing constraints of Class I transactions, to achieve the specified guarantee levels of Class II transactions, and finally, to minimize the deadline miss ratio of Class III transactions (or maximize the total values of the completed transactions if a value function can be defined for each transaction). At the same time, all the consistency requirements implicated in the real-time database and transactions must be fulfilled.

To achieve this performance goal, it is necessary to apply different transaction scheduling and concurrency control algorithms for each class of transactions. In this chapter, we present a framework to achieving predictable real-time transaction processing which includes a method to maintaining database consistency statically and an integrated scheduling scheme to satisfying the performance requirement of each class of transactions.

4.1 Maintaining Temporal Consistency

There are two possible approaches to maintaining temporal consistency of real-time data objects: one is *static* and the other is *dynamic*. In a static approach, temporal consistency requirements are transformed into timing constraints of transactions. The system then has only to provide a guarantee on the timing constraints, since as long as the corresponding transactions meet their deadlines, the temporal consistency of the data objects accessed by the transactions is automatically maintained [33, 51, 54]. In a dynamic approach, the system keeps checking the temporal consistency at run time and tries to meet them dynamically, by either using multiple versions of data objects [66] or delaying some transactions in favor of more urgent transactions in terms of temporal consistency enforcement at the specific moment [36].

In this section, we present a static approach which we call Static Temporal Consistency Enforcement (STCE) scheme for our real-time database system model. We decided not to go for a dynamic approach, since it involves significant run-time overheads and may conflict with other scheduling mechanisms which are responsible for enforcing the timing constraints of transactions. More importantly, our real-time database system model presented in the previous chapter is designed for a static approach in mind. Our static approach is different from other static ones in that it provides a total guarantee on temporal consistency. In other static approaches, temporal consistency is not guaranteed, but just its violation ratio is tried to be minimized.

Throughout this section, we assume that the transactions are scheduled under a fixed-priority scheduling framework, called *deadline-monotonic* scheduling scheme [3], where priorities assigned to processes are inversely proportional to the length of the deadline. Thus, the process with the shortest deadline is assigned the highest priority and the longest deadline process is assigned the lowest priority. This priority ordering defaults to a *rate-monotonic* ordering [53] when the period equals to the deadline.

Class IA Transactions

A Class IA transaction τ_x is responsible for maintaining the absolute temporal consistency of an image object x. To achieve this, its period must satisfy the following condition:

$$avd_x \ \geq \ P_x + D_x, \tag{4.1}$$

where avd_x is the absolute validity duration of x, P_x is the period of τ_x, and D_x is the deadline of τ_x. This is because the worst-case next update time for an image object which is written at the beginning of a certain period is the deadline of the next period (Figure 4.1).

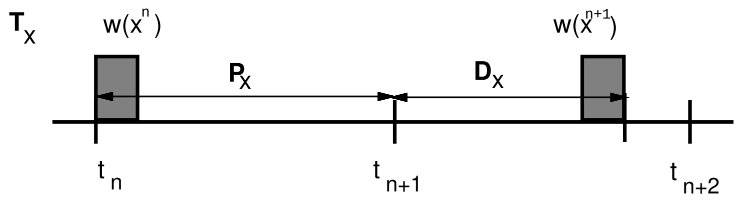


Figure 4.1: The Worst-Case Scenario for a Class IA Transaction

If the deadline of a transaction is the end of the period, the value of the period must be less than or equal to the half of the absolute validity duration of the related image object to maintain its absolute temporal consistency.

Class IB Transactions

The value of a derived object y is correct only while the value of each data object σ_i in Σ_y is correct (i.e., temporally consistent) and y maintains its correctness until the next instance of the transaction updates the value. Unfortunately, it is extremely difficult, if not impossible, to ensure this dynamically. We approach this problem by giving some restrictions on the attributes of transactions under a fixed-priority scheduling framework.

Assuming that the first periods of all transactions begin at the same time (*in phase*), we find a sufficient condition to maintain the temporal consistency of a derived object:

A derived object y always has a correct value if each transaction τ_{σ_i} which writes a data object in Σ_y, always meets the deadlines and satisfies the following condition:

$$P_y = P_{\sigma_i} = D_{\sigma_i} \ \leq \ \min\{rvd_y, \ 0.5 * \min_j(avd_{\sigma_j})\} \tag{4.2}$$

for all $\sigma_i \in \Sigma_y \cap (X \cup Y)$, where rvd_y is the relative validity duration of Σ_y. Higher priorities must be assigned to all τ_{σ_i} than that of τ_y so that all σ_is are written before τ_y and τ_y always reads the most recent value of σ_i. Applying these conditions, Σ_y maintains its relative temporal consistency and each σ_i read by τ_y will be valid until the next update of y.

However, since this condition is very restrictive, we investigate how to relax this condition, considering the semantics of a specific application. Suppose the given application allows the following temporal consistency criteria of derived objects:

1. The values of σ_i in Σ_y have only to be valid until the completion of τ_y.
2. The derived object y has its own absolute validity duration avd_y.
3. Σ_y must satisfy the relative temporal consistency requirement (i.e., temporal distance $d(y, x_i) \leq rvd_y$ for any $x_i \in \Sigma_y \cap (X \cup Y)$).

Using this kind of semantic information, we can derive the following conditions for the period and deadline of a Class IB transaction in order to maintain the temporal consistency of derived objects under the deadline-monotonic scheduling framework:

$$P_{x_i} + D_{x_i} \leq avd_{x_i}, \tag{4.3}$$

$$P_y + D_y \leq avd_y, \tag{4.4}$$

$$D_y \leq D_{x_i}, \text{ and} \tag{4.5}$$

$$P_{x_i} + D_{x_i} \leq rvd_y \tag{4.6}$$

for all $x_i \in \Sigma_y \cap (X \cup Y)$. Here, if $D_{x_i} = D_y$, then a higher priority must be assigned to τ_y than that of τ_{x_i} so that τ_y reads x_i before the next update of x_i by τ_{x_i}. Note that unlike Condition (4.2), all P_{x_i}, D_{x_i}, P_y, and D_y do not have to be the same as long as D_y is less than or equal to the smallest D_{x_i}.

As long as the above conditions hold, the temporal consistency requirements related with a derived object are always satisfied. We can justify this statement as follows. First of all, let's consider the two possible cases of a Class IB transaction's read operation ($r(x_i)$ by τ_y in Figure 4.2).

Case I: τ_y reads x_i *after* the current instance of τ_{x_i} updates x_i (i.e., writes x_i^n).

In this case, the value x_i^n will be valid at least until the time $t_{n+1} + D_{x_i}$ from Condition (4.3) and the completion of τ_y comes before that time since $D_y \leq D_{x_i}$ from Condition (4.5). Thus, x_i^n, the value read by τ_y, will be valid until τ_y completes.

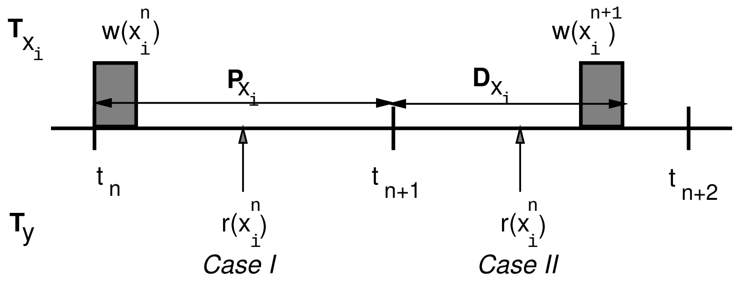


Figure 4.2: Two Possible Cases of a Class IB Transaction's Read Operation

Case II: τ_y reads x_i *before* the current instance of τ_{x_i} updates x_i (i.e., writes x_i^{n+1}).

In this case, the value x_i^n read by τ_y will be valid at least until the time $t_{n+1}+D_{x_i}$ from Condition (4.3). Furthermore, τ_y should complete before that time (i.e., the deadline of τ_{x_i}), since the priority of τ_y is higher than the priority of τ_{x_i} from Condition (4.5). Thus, x_i^n, the value read by τ_y will be valid until the completion of τ_y.

The above two cases cover all possible situations for the transaction τ_y and it is shown that the absolute temporal consistency of Σ_y is maintained in each case if the conditions (4.3) and (4.5) hold. The absolute temporal consistency of y is guaranteed by Condition (4.4). Therefore, the *absolute* temporal consistency involved in a derived object is always maintained under the given conditions.

Now, let's suppose that the arrival time of τ_y is a_y. Then, the oldest possible timestamp of x_i read by τ_y is $a_y+R_y-\max_{\forall x_i\in\Sigma_y\cap(X\cup Y)}(P_{x_i}+D_{x_i})$ (R_y is the worst-case response time of τ_y), since x_i must be valid until the completion of τ_y. Also, the latest possible timestamp of y is a_y+R_y. Thus, the maximum temporal distance between the data object y and any data object in $\Sigma_y\cap(X\cup Y)$ is

$$\max_{\forall x_i\in\Sigma_y\cap(X\cup Y)}(P_{x_i}+D_{x_i}).$$

According to the definition, Σ_y satisfies its *relative* temporal consistency requirement if this distance is less than or equal to rvd_y (i.e., if Condition (4.6) holds).

From the above, we just show that all the temporal consistency requirements for a derived object can be satisfied if the conditions (4.3) through (4.6) retain.

Class IC Transactions

Since a Class IC transaction is read-only, the values of data objects read by the transaction have only to be in a correct state until the transaction finishes.

Suppose $O_\tau = \{o_i\}$ is a set of data objects that are read by a Class IC transaction τ, rvd_τ is a relative validity duration of the set, and τ_{o_i} is responsible for updating the data object o_i. The set O_τ must satisfy its relative temporal consistency requirement (i.e., temporal distance between any two data objects in O_τ must be less than or equal to rvd_τ) as well as the absolute temporal consistency of each data object in the set. The following conditions are sufficient to maintain the above temporal consistency requirements by a Class IC transaction, assuming that the first periods of all transactions begin at the same time (*in phase*):

$$P_{o_i} \ = \ D_{o_i} \ \leq \ \min\{rvd_\tau,\ 0.5 * \min_j(avd_{o_j})\}, \tag{4.7}$$

$$P_\tau \ = \ n * P_{o_i}, \ \ n = 1,\ 2,\ 3,\ \ldots,\ \text{and} \tag{4.8}$$

$$D_\tau \ \leq \ 2 * P_{o_i} \tag{4.9}$$

for all $o_i \in O_\tau \cap (X \cup Y)$, where rvd_τ is the relative validity duration of O_τ and τ has a lower priority than any τ_{o_i} to guarantee that it reads the most recent values of o_is. With these conditions, O_τ satisfies its relative temporal consistency, and each o_i read by τ will be valid at least until τ finishes (in the worst-case, the deadline of τ).

Again, however, these conditions may be too restrictive for some applications. Alternative conditions for Class IC transactions similar to those of Class IB transactions can be given as follows:

$$P_{o_i} + \ D_{o_i} \ \leq \ avd_{o_i}, \tag{4.10}$$

$$D_\tau \ \leq \ D_{o_i},\ \text{and} \tag{4.11}$$

$$P_{o_i} + D_{o_i} - R_\tau \ \leq \ rvd_\tau \tag{4.12}$$

for all $o_i \in O_\tau \cap (X \cup Y)$, where R_τ is the worst-case response time of τ. Remind that transactions are supposed to be processed under the deadline-monotonic scheduling framework. In case $D_{o_i} = D_\tau$, τ must be assigned a higher priority than that of τ_{o_i} so that it reads o_i before the next update of o_i by τ_{o_i}.

The justification for the conditions (4.10) and (4.11) can be given similar to that of Class IB transactions. We can divide the τ's read operation on o_i into two possible cases: the one is the case that τ_y reads o_i *after* the current instance of τ_{o_i} updates o_i and the other is the case that τ reads o_i *before* the current instance of τ_{o_i} updates o_i. In the former case, the current value of o_i will be valid at least until the next deadline of τ_{o_i} from Condition (4.10) and the completion of τ comes before that time since $D_\tau \leq D_{o_i}$ from Condition (4.11). Thus, the value of o_i read by τ, will be valid until τ completes. In the latter case, the current value of o_i read by τ will be valid at least until the current deadline of τ_{o_i} from Condition (4.10). Furthermore, τ should complete before that time, since the priority of τ is higher than the priority of τ_{o_i} from Condition (4.11). Thus, the value read by τ will be valid until the completion of τ. From the above, we know that the absolute temporal consistency of O_τ is maintained for τ if the conditions(4.10) and (4.11) hold. Note that since a Class IC transaction writes no data object, there is no restriction on its period.

Condition (4.12) can be justified as follows: if the arrival time of τ is a_τ, the oldest possible timestamp of o_i read by τ is $a_\tau + R_\tau - \max_{\forall o_i \in O_\tau \cap (X \cup Y)}(P_{o_i} + D_{o_i})$, since o_i must be valid until the completion of τ, and the latest possible timestamp of o_i read by τ is less than a_τ, since τ has higher priority than any τ_{o_i} and thus τ_{o_i} cannot write o_i while τ is runnable. Therefore, the maximum temporal distance between any two data objects in $O_\tau \cap (X \cup Y)$ is

$$\max_{\forall o_i \in O_\tau \cap (X \cup Y)} (P_{o_i} + D_{o_i}) \; - \; R_\tau,$$

and if it is less than or equal to rvd_τ (i.e., if Condition (4.12) holds), O_τ satisfies its relative temporal consistency requirement.

In conclusion, all the temporal consistency requirements involved in a Class IC transaction can be achieved under the given conditions.

4.2 Integrated Transaction Scheduling

Since the STCE scheme described in the previous section transforms the temporal consistency constraints of a real-time database into timing constraints of transactions, the RTDBS now have only to concentrate on satisfying the timing constraints of the transactions, but does not have to care about temporal consistency of data objects at run time. In other words, with STCE the goal of a real-time database system (to meet both consistency and timing constraints) can be achieved as long as the transactions meet their timing constraints.

In this section, we present an integrated transaction scheduling scheme to achieve this goal. Since the transactions in each class defined in Chapter 3 have different performance requirements (i.e., different guarantee levels on their timing constraints), a distinct scheduling policy must be applied to each transaction class. Our approach to jointly scheduling all the classes of transactions is to utilize a fixed-priority preemptive scheduling framework for guaranteeing timing constraints of Class I and Class II transactions *statically* and a slack stealing algorithm to find spare capacity for *dynamically* scheduling Class III transactions. This approach requires to extend the base algorithms designed for real-time *task* scheduling to support real-time *transactions*, and integrate them into our RTDBS model.

Suppose that a real-time database application consists of a set of transactions

$$\mathcal{T} = \{T_I, T_{II}, T_{III}\},$$

where T_N is a set of Class N transactions, $N \in \{I, II, III\}$. We also denote the set of all guaranteed transactions $T_I \cup T_{II}$ as T_G . Each class of transactions in $\mathcal{T}$ should be scheduled to achieve its own performance goal as follows. We assume that the underlying subsystems provide deterministic services to the transactions and thus real-time data objects can be accessed with deterministic service time.

4.2.1 Guaranteed Scheduling

Total Guarantee on Class I Transactions

For all transactions in T_I, the computation and data requirements are known in advance. Also, there is no blocking due to data conflicts with other transactions (i.e., these transactions access only a fixed set of continuous data objects, and any values of the data objects can be used as long as they maintain temporal consistency), requiring no concurrency control. Therefore, total guarantee can be provided for this class of transactions under a fixed-priority scheduling algorithm.

In this research, we employ a *deadline-monotonic* approach for hard deadline guarantee, since it is more flexible than the rate-monotonic approach and easily extendible.[1] An extended schedulability test for our model under this approach will be presented later in this section.

Statistical Guarantee on Class II Transactions

If the worst-case execution time is available, hard sporadic transactions can be guaranteed to meet their deadlines under a deadline-monotonic scheduling framework, as shown in Section 2.1. Unfortunately, this is not the case for transactions in T_{II}: its execution time is not necessarily bounded.

However, if we assign an arbitrary execution time budget to a Class II transaction and enforce it at run time, it can be included in the deadline-monotonic scheduling framework and will always meet the deadline as long as the actual execution time does not exceed the given budget. In order to meet its required guarantee level based on this idea, we must determine an appropriate execution time budget for a Class II transaction.

Suppose that we can bound the time to fetch one instance of the data objects accessed by a Class II transaction τ (denoted as t_{fetch}). Then, the pure execution time of τ (denoted

[1]In deadline-monotonic approach, the deadline and period of a process do not have to be equal. Such a relaxation enables sporadic processes to be directly incorporated without alteration to the process model [3].

as C_τ) can be written as

$$C_\tau = t_{init} + (t_{fetch} + t_{comp}) * N * S_\tau + t_{close},$$

where N is the size of the database, S_τ is a random variable of the selectivity distribution of τ, and t_{init}, t_{comp}, and t_{close} are the transaction initialization time, the pure computation time of τ per data object, and the transaction closing time, respectively.

Since the cumulative distribution function for S_τ is known, we can get the probability that the worst-case execution time of τ, C_τ, is t_s:

$$Prob\,[C_\tau \leq t_s] \;=\; Prob\,[S_\tau \leq s] \;=\; p_s,$$

where $t_s = t_{init} + (t_{fetch} + t_{comp}) * N * s + t_{close}$.

We can claim that if the value t_s is used as the transaction's maximum execution time budget and the sporadic task scheduling scheme presented in Section 2.1 is employed, the transaction will meet its deadline with the probability no less than p_s (i.e., it will achieve the given guarantee level p_s).

The Class II transaction scheduling protocol can be summarized as follows:

1. Derive the selectivity s of a transaction τ in T_{II} from the given performance requirement (its *guarantee level*, p_s) and the selectivity distribution (S_τ), and then calculate the execution-time budget t_s using s.

2. Regard τ as a periodic transaction with the period M_τ (the minimum inter-arrival time of τ), the worst-case execution time t_s, and the deadline D_τ. Then, τ can be scheduled under the deadline-monotonic scheduling framework.

3. Keep track of the consumed run time by τ. If τ has spent the given execution-time budget t_s but is not completed yet, it must be treated as a Class III transaction with the highest priority until its deadline. In this way, overrunning Class II transactions never affect the other hard deadline transactions in T_G.

Table 4.1: Parameters and Notations

Notation	Description
P_j	Period of τ_j in T_I
C_j	Worst-case computation time of τ_j in T_I
D_i	Deadline of τ_i in T_G
M_s	Minimum inter-arrival time of τ_s in $hps(i)$
C_s	Worst-case computation time of τ_s in $hps(i)$
P_{clk}	Clock interrupt handler period
C_{clk}	Worst-case execution time of a clock interrupt handler
C_{int}	Fixed overhead associated with a clock interrupt
C_{QL}	Cost of moving one process between queues
C_{QS}	Additional cost (per transaction) of moving
	more than one transaction at a time
$hpp(i)$	The set of higher-priority transactions than τ_i in T_I
$hps(i)$	The set of higher-priority transactions than τ_i in T_{II}
sih	Set of all sporadic interrupt handlers
M_h	Period of a sporadic interrupt handler in sih
	associated with a transaction τ_s in $hps(i)$
C_{IH}	Computing cost of a sporadic interrupt handler

Extended Schedulability Analysis

The schedulability tests for the deadline-monotonic scheduling scheme presented in [3] does not include the possible system overheads involved in the underlying kernel mechanism, such as clock interrupt-driven scheduling, context switching, and sporadic interrupt handling. A more realistic off-line schedulability analysis method must be designed for transactions in T_G, considering the system's operating environment.

In the following, we present our extended schedulability analysis based on the analysis in [9, 10]. The parameters and notations used in the analysis are summarized in Table 4.1.

In our RTDBS model, transactions are supposed to be scheduled by a *timer-driven* *scheduler* [32] (i.e., the scheduler is invoked by a regular timing interrupt with a period denoted by P_{clk}). The scheduler maintains two queues of tasks: one called the *delay queue*, containing a list of tasks ordered by next arrival time, and one called the *run queue*, containing a list of runnable tasks, ordered by priority. When the scheduler is invoked, it removes any tasks where the arrival time is less than the current time, and places these tasks in the run queue. The scheduler then dispatches the highest priority task. In the worst-case, the scheduler must take all the tasks from the delay queue and place them in the run queue. The computational overheads due to the execution of the scheduler must be included in the analysis.

Suppose that the transactions in $T_G(=\{\tau_1, \tau_2, \ldots, \tau_n\})$ are ordered according to priority with τ_1 having the highest priority and τ_n having the lowest. Then, for each τ_i, the following relationship holds:

$$R_i = C_i + B_i + I_i + IS_i + IH_i, \tag{4.13}$$

where R_i is the worst-case response time of τ_i, C_i is the worst-case execution time of τ_i, B_i is the worst-case blocking time of τ_i, I_i is the interference that τ_i experiences from higher-priority transactions in T_G, IS_i is the computational overheads due to the scheduler, and IH_i is the sporadic interrupt handler overheads. Then, the schedulability of a transaction τ_i can be assessed by comparing the worst-case response time R_i with the deadline:

$$R_i \le D_i.$$

Each term in Equation (4.13) can be determined as follows:

$$\begin{aligned}
I_i &= \sum_{\forall j \in hpp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil C_j + \sum_{\forall s \in hps(i)} \left\lceil \frac{R_i}{M_s} \right\rceil C_s, \\
IS_i &= K * C_{int} + \min(K, V) * C_{QL} \\
&\quad + \max(V - K, 0) * C_{QS}, \text{ and} \\
IH_i &= \sum_{\forall h \in sih} \left(C_{IH} + \left\lceil \frac{R_i}{M_h} \right\rceil C_{IH} \right),
\end{aligned}$$

where K is the maximum number of times the scheduler is invoked in a given interval $(0, R_i]$ and V is the maximum number of transactions moved from the delay queue to the run queue in the interval, which can be bounded by:

$$K = \left\lceil \frac{R_i}{P_{clk}} \right\rceil \quad \text{and} \quad V = \sum_{\forall j \in T_G} \left\lceil \frac{R_i}{P_j} \right\rceil .$$

The first and second term of the equation for I_i represent the interferences that τ_i experiences from higher-priority Class I and Class II transactions, respectively. If the number of movements from the delay queue to the run queue (V) is larger than the number of invocations of the timer-driven scheduler (K), then some of the invocations of the scheduler will take more than one transactions from the delay queue. Therefore, some transaction movements will not require time C_{QL}, but C_{QS} per transaction. Hence the costs of operating the scheduler can be bounded as:

$$\begin{aligned} C_{QL} * K \ + \ C_{QS} * (V - K) \qquad & \text{if } V \geq K \\ C_{QL} * V \qquad & \text{otherwise.} \end{aligned}$$

Adding on the costs of the clock interrupt processing, we can obtain the formula for IS_i as above.

Each sporadic Class II transaction has associated with an interrupt handler. However, it is not possible to simply add the cost of the interrupt handling into the cost of the sporadic. This is because the interrupt handler is allowed to execute twice in an interval smaller than M_i. To understand this, consider the sporadic executing immediately after an interrupt has occurred. This will turn interrupts back on again, and hence a second interrupt could occur soon after the previous one. However, at most 3 interrupt handling activities can occur in any interval $2M_i$. In general, $N+1$ interrupts can occur in N intervals. Thus, the cost of interrupt handling for Class II transactions (IH_i) can be given as above.

Note that transactions in T_G will not be blocked due to data contention with other transactions, but we may need B_i's to account for the blockings caused by unavoidable critical sections in the kernel and the servers (for example, meta-data manipulation like index).

In the above analysis, it is assumed that if a transaction completes, an interrupt is raised and the highest priority transaction in run queue can be immediately scheduled. However, it is not always supported by the operating system kernel. In some systems, a clock interrupt is the only scheduling point and thus a context switch cannot occur in the middle of a clock period due to a transaction completion. The amount of time between a transaction completion and the next clock interrupt must be wasted. If this is the case, the worst-case computation time of τ_i (C_i) used in the above equations must be replaced by C'_i:

$$\begin{aligned} C'_i &= (P_{clk} - C_{clk}) \left\lceil \frac{C_i}{P_{clk} - C_{clk}} \right\rceil, \\ C_{clk} &= C_{int} + C_{QL} + (n-1)C_{QS}, \end{aligned}$$

where C_{clk} is the worst-case execution time of the clock interrupt handler and n is the number of transactions in T_G.

4.2.2 Non-Guaranteed Scheduling

In the preceding subsection, we described how to provide offline guarantees for Class I and Class II transactions. It is noted that the provision of guarantees implies that computing resources are under-utilized at run time (i.e., spare capacity exists at run time). This spare capacity must be detected at run time and assigned to schedule non-guaranteed transactions (i.e., overrunning Class II transactions and Class III transactions), while maintaining the guarantees made offline regarding Class I and Class II transactions.

Exploiting Spare Capacity

Within the fixed-priority preemptive scheduling framework, a number of approaches have been developed for scheduling soft real-time tasks along with the guaranteed hard real-time tasks, as reviewed in Section 2.1. Our approach to scheduling non-guaranteed transactions is based on *dynamic slack stealing* approach reviewed in Appendix A, which is shown to have better flexibility and performance than other approaches, such as background processing, the Sporadic Server algorithm [67], the Extended Priority Exchange algorithm [68] and the optimal static slack stealing algorithm [48].

Different from the original dynamic slack stealing presented in [14], our algorithm accounts the scheduler overheads included in our extended schedulability analysis. That is, Equation (A.1) in Section A.2 must be modified by adding the following term $IS_{w_i}(t)$, which represents the clock interrupt handling overhead during the busy period, to the right-hand side of the equation:

$$IS_{w_i}(t) \;=\; K * C_{int} + \min(K, V) * C_{QL} + \max(V - K, 0) * C_{QS},$$

where

$$K = \left\lceil \frac{w_i^m(t)}{P_{clk}} \right\rceil \quad \text{and} \quad V \;=\; \sum_{\forall j \in T_G} \left\lceil \frac{w_i^m(t)}{P_j} \right\rceil .$$

In summary, the transaction processing protocol for non-guaranteed transactions under the timer-driven scheduling environment can be described as follows:

1. Maintain the available slack time counter, $S_i(t)$ for every priority level i during $[t, t + D_i(t))$, using one of the dynamic slack stealing methods.

2. At each scheduling point, if there is no guaranteed transactions runnable, schedule the highest priority non-guaranteed transaction.

3. If there exist both guaranteed and non-guaranteed transactions runnable at a scheduling point but slack time is available at priority level k and all lower levels (i.e.,

$$\min_{\forall j \in lp(k)} S_j(t) > 0,$$

   where k is the priority of the highest runnable guaranteed transactions), then the highest-priority non-guaranteed transaction is scheduled.

4. Among the non-guaranteed transactions, an overrunning Class II transaction has a higher priority than that of any other Class III transactions until its deadline is reached.

Best Effort Scheduling for Class III Transactions

Once spare capacity is available for Class III transaction processing, best effort must be made choose a right transaction to schedule so that the total values of the completed transactions can be maximized. If we assume that all Class III transactions have the same values when they complete, the obvious goal of the scheduler is to meet as many deadlines as possible. In this case, we can assign priorities to Class III transactions using one of the following methods: Earliest Deadline First (EDF) or Least Slack First (LSF) [1]. Otherwise, we need to define a *priority value function* for each transaction:

$$p_\tau(t) = f(v_\tau(t), t - a_\tau, D_\tau, c_\tau, C_\tau - c_\tau),$$

where c_τ is the consumed execution time budget for the current invocation of transaction τ.

Note that the soft deadline task model used in the original slack stealing studies [48, 14, 12] is different from our Class III transaction model in that its performance goal is to minimize their average response times. The schedulers do not care about the deadlines of soft deadline tasks, but just process them in FIFO order.

4.3 Semantic Concurrency Control and Conflict Resolution

As presented in Chapter 3, the real-time database model does not always require *serializable schedules* for transactions to maintain consistency of the database. Utilizing the inherent semantic information about transactions in specific classes, the real-time transaction manager can make different control decisions for different classes of transactions, in order to maximize the level of concurrency while maintaining both timing and consistency constraints of the system.

In this section, we present a semantic concurrency control and conflict resolution scheme under our real-time database model. Our scheme is based on an optimistic real-time concurrency control algorithm using *Precise Serialization* (OCC-PS) developed by Lee and Son [41]. According to the recent studies in [24, 23, 25], optimistic approach appears well-suited to real-time database systems. Especially, OCC-PS is shown to outperform other real-time optimistic concurrency control algorithms, and more importantly, it can be easily integrated with non-serializable transaction scheduling.

Under our Semantic Optimistic Concurrency Control (SOCC) algorithm, each class of transactions is managed as follows so that both consistency and timing constraints of the system can be satisfied.

Class I Transactions

Class I transactions write only continuous data objects which do not require serializable accesses as long as they are temporally consistent. There is no data conflict among them, and it is never necessary to block or abort transactions in this category, due to data contention.

Especially, no concurrency control is required for a Class IA transaction, since it is a *write-only* transaction on some image objects and it is the one and only writer to the objects. However, Class IB and IC transactions may experience *read-write* conflicts with Class III transactions, since they are allowed to read discrete data objects. Even in this case, the Class I transactions can continue without blocking, while the conflicting Class III transactions must wait until the conflicting Class I transactions commit. Note that the Class III transactions do not have to be aborted or restarted as long as they can feasibly meet the deadlines, since the Class I transactions are not going to write discrete data objects.

In conclusion, Class I transactions can bypass the validation phase of SOCC and always commit without blocking.

Class II Transactions

We can also claim that Class II transactions do not experience any blocking due to data contention, since:

- They never conflict with Class I transactions, since serializable access is not necessary for continuous data objects which can be shared by both classes of transactions. Class IB, Class IC, and Class II transactions may share some discrete data objects, but they do not conflict with each other, since the accesses are read-only.

- There is no conflict among Class II transactions, since they are *read-only* transactions.
- Even though a conflict occurs with a Class III transaction, the Class II transaction continues. The conflicting Class III transaction must wait until the Class II transaction commits.

Since the priority inversion problem due to shared data objects will not occur, no priority ceiling protocol (PCP)-based synchronization scheme [54] is necessary. Therefore, Class II transactions can also bypass the validation phase of SOCC and always commit.

Class III Transactions

As described above, if a Class III transaction conflicts with a Class I or Class II transaction, it just waits until the conflicting transaction commits as long as it is still feasible. However, if a conflict occurs between two Class III transactions, it should be resolved based on their priorities (e.g., *WAIT-50* [25] and *Feasible Sacrifice* [40]).

Class III transactions under the SOCC algorithm must go through the validation phase and read phase as detailed in Figure 4.3 and Figure 4.4, respectively. Note that as explained above, Class I and Class II transactions do not have to carry out these phases.

In OCC-PS, a concurrently running transaction, τ_{cr}, which has read data items updated by the committing transaction, τ_v, is not always restarted. Instead, such write-read conflicts[2] are resolved by dynamically placing τ_{cr} ahead of τ_v in execution history. That is, the order of τ_{cr} in history is dynamically arranged that it does not read from τ_v, and hence, τ_{cr} needs not restart. Such placement of τ_{cr} in execution history is called *backward ordering.* On the other hand, to resolve read-write[3] and write-write[4] conflicts, OCC-PS adjusts the serialization order as $\tau_v \rightarrow \tau_{cr}$. Again conflicts are resolved without restarts, because τ_{cr}'s writes do not affect the operations of τ_v. Such placement of τ_{cr} is referred to as *forward ordering.*

[2]conflicts between the transactions τ_v and τ_{cr} if $WS_{\tau_v} \cap RS_{\tau_{cr}} \neq \emptyset$
[3]conflicts between the transactions τ_v and τ_{cr} if $RS_{\tau_v} \cap WS_{\tau_{cr}} \neq \emptyset$
[4]conflicts between the transactions τ_v and τ_{cr} if $WS_{\tau_v} \cap WS_{\tau_{cr}} \neq \emptyset$

```
/* $\tau_v$: a validating Class III transaction */
/* $CR_{II}(\tau_v)$: a set of currently running Class II transactions for $\tau_v$ */
/* $CR_{III}(\tau_v)$: a set of currently running Class III transactions for $\tau_v$ */

Algorithm validate ($\tau_v$)
begin
  /* Check if any active Class II transaction conflicts */
  /* with this validating Class III transaction, $\tau_v$ */
  forall $\tau_{cr}$ in $CR_{II}(\tau_v)$ do
      if $WS(\tau_v) \cap RS(\tau_{cr}) \neq \emptyset$ then restart($\tau_v$);
  end

  /* Check if any active Class III transaction conflicts */
  /* with this validating Class III transaction, $\tau_v$ */
  forall $\tau_{cr}$ in $CR_{III}(\tau_v)$ do
      fore = back = FALSE;
      if $WS(\tau_v) \cap RS(\tau_{cr}) \neq \emptyset$ then back = TRUE;
      if $WS(\tau_v) \cap WS(\tau_{cr}) \neq \emptyset$ then fore = TRUE;
      if $RS(\tau_v) \cap WS(\tau_{cr}) \neq \emptyset$ then fore = TRUE;
      if fore == back == TRUE then restart($\tau_{cr}$);
      else if back == TRUE then insert $\tau_v$ to $RCC(\tau_{cr})$;
  end

  commit $WS(\tau_v)$ (except late-writes) to database;
end
```

Figure 4.3: Validation Phase of a Class III Transaction

```
/* $\tau$: a running Class III transaction */
/* $o_\tau[x]$: an operation of $\tau$ (either $r_\tau[x]$ or $w_\tau[x]$) */

Algorithm read_phase_check ($o_\tau[x]$);
begin
  forall $\tau_{rcc}$ in $RCC(\tau)$ do
      if $o_\tau[x] == r_\tau[x]$ then
          if $x$ in $WS(\tau_{rcc})$ then restart($\tau$);
      else   /* i.e., $o_\tau[x] == w_\tau[x]$ */
          if $x$ in $RS(\tau_{rcc})$ then restart($\tau$);
          if $x$ in $WS(\tau_{rcc})$ then mark as a late-write;
      end
  end
end
```

Figure 4.4: Read Phase Check in a Class III Transaction

In the OCC-PS algorithm, a running transaction τ, restarts only when it is involved in not only a write-read conflict with a validating transaction τ_v, but also involved in either a read-write or a write-write conflict with τ_v. In such a situation, the OCC-PS algorithm attempts to place τ both behind and in front of τ_v in execution history, which is clearly not feasible, and hence τ needs to be restarted. Note that such restarts are inevitable to ensure serializability.

In validation phase (Figure 4.3), OCC-PS employs two flags, *fore* and *back*, which indicate the current τ_{cr}'s place relative to τ_v in execution history. In read phase (Figure 4.4), the algorithm maintains for each running transaction τ, a set of *recently-committed transactions having a write-read conflict* with τ. Let $RCC(\tau)$ denote the set for τ. $RCC(\tau)$ contains only relevant transactions in serializing conflicts during τ's read phase. Those transactions are detected when a write-read conflict occurs between τ and a validating transaction. Note that the serialization order between τ and τ_{rcc} in $RCC(\tau)$ determined in the validation phase of τ_{rcc} is $\tau \rightarrow \tau_{rcc}$. If τ reads a data item updated by τ_{rcc}, τ has to restart, because τ's reading from τ_{rcc} results in the serialization order of $\tau_{rcc} \rightarrow \tau$. If τ writes on a data item that has been read by τ_{rcc}, τ also has to restart, because τ_{rcc} has not read the value written by τ, while their order in history is $\tau_{rcc} \rightarrow \tau$. Finally, if τ writes on a data item written by τ_{rcc}, instead of restarting τ, the OCC-PS algorithm applies the *Thomas' Write Rule* [6], i.e., guarantees serializability simply by not committing the write value of τ to database. (A late write value of τ can be discarded from the private workplace as soon as it is known that the write is late.)

Different from the original OCC-PS algorithm, our Semantic Concurrency Control (SOCC) algorithm makes use of certain semantic information of transactions during the validation phase. For example, when a validating Class III transaction τ_v checks if any active Class II transaction conflicts with itself, it does only one type of conflict (i.e., write-read conflict) since it is known Class II transaction is read-only. Furthermore, when a conflict is detected between τ_v and an active Class II transaction τ_{cr}, the validating transaction τ_v must be restarted, since τ_{cr} has more critical deadline than τ_v and should not be restarted. In this situation, OCC-PS inserts τ_v to $RCC(\tau_{cr})$ and immediately commits τ_v, so τ_{cr} should be restarted if it reads any data item in $WS(\tau_v)$ sometime later.

Chapter 5

Cost and Performance Evaluation

In this chapter, we compare the performance of our integrated real-time transaction scheduling algorithms to that of the various conventional algorithms through simulation study. We also explore how much it costs to guarantee the timing constraints of real-time transactions and their temporal consistency requirements under our system.

5.1 Simulation Model

This section outlines the structure and details of our simulation model and explains how each class of data objects and transactions in our model is generated synthetically for the experiments.

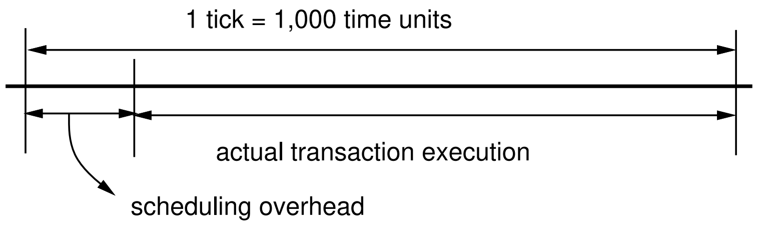
First of all, the assumptions we made in our simulation study are as follows:

- The system consists of a *single processor* and a large main memory. The database is memory resident. This assumption simplifies our model and provides a deterministic data access time for a transaction.
- Timer-driven scheduler (or tick scheduler), described in the previous chapter.
- A context switch cannot occur between the ticks, implying that the regular clock interrupt is the only scheduling point. Even though a transaction completes before the next tick, the remaining CPU time cannot be utilized by the highest priority waiting transaction.
- No failure situation is assumed, and hence no recovery procedure is provided.

- Only firm deadlines for Class III transactions. A Class III transaction whose deadline has expired (*tardy* transaction) will be discarded (aborted) at the earliest possible time. This is to avoid the complexity of scheduling tardy soft deadline transactions, which is beyond the scope of this research.

- Once released, each transaction is supposed to execute for its given worst-case execution time. That is, the gain time (difference between the worst-case execution time and the actual execution time of a transaction) is always 0 and cannot be reclaimed by the slack stealing algorithms.

We implement the *timer-driven* scheduler described in Section 4.2. One tick is a basic scheduling unit, which is called a *clock interrupt* in an actual system. However, unlike in the conventional simulation model, one tick is not the unit of simulation time. It is 1,000 time unit in our model, so that we can simulate the overheads due to tick scheduling, which are included in the extended deadline-monotonic schedulability analysis described in Section 4.2.



The other overheads, such as transaction release delays, slack calculation, and concurrency control, are also considered in our simulation model.

5.1.1 Workload Characteristics and Generation

To evaluate the cost and performance of our algorithms in a simulation program, we need to generate synthetic real-time databases and transaction workloads.

Real-Time Databases

The real-time database used in the simulation is generated according to the model described in Chapter 3. First, a given number of image objects are generated, each having an *absolute validity duration* (avd). A number of derived objects is then generated, each with a set of data objects that are used to compute the value of a derived object and its *relative validity duration* (rvd). Finally, the number of discrete data objects in the database is determined.

Real-Time Transactions

The transaction loads for Class I and Class II transactions are simulated using groups of transaction sets with utilization levels of 30, 50, and 70 %. The results presented in subsequent sections are the averages over a group of five transaction sets. Each transaction set is generated as follows. First, Class IA and Class IB transactions for each image and derived object are generated. Then, a number of Class IC transactions are generated, each with a randomly generated read set and the corresponding *relative validity duration.* The periods and deadlines of Class I transactions are assigned, constrained by the equations in Section 4.1. Execution times of Class I transactions are calculated, considering the transaction processing overhead and data object access time. The transaction set for Class I and Class II, generated as above, is then sorted into deadline monotonic priority order and go through the schedulability test presented in Section 4.2. The above steps should be repeated until a schedulable transaction set with a desired utilization level is obtained.

To simulate sporadic arrival of Class II transactions, each Class II transaction τ_i has the following probability of arrival per tick ($Prob^{arr}_{\tau_i}$) at time t:

$$Prob^{arr}_{\tau_i} = \begin{cases} 0 & \text{if } -l_i(t) < P_i \\ \frac{1}{P_i} & \text{otherwise} \end{cases},$$

where $l_i(t)$ is the time relative to t of the previous arrival of transaction τ_i.

The Class III transaction load is simulated by a pre-generated queue of transactions, each having predefined attributes, such as deadline, execution time, read set, and write

Table 5.1: System Parameters

Parameter Name	Description	Base value
`P_CLK (TICK)`	P_{clk} in Table 4.1	1000
`C_INT`	C_{int} in Table 4.1	2
`C_QL`	C_{QL} in Table 4.1	6
`C_QS`	C_{QS} in Table 4.1	3
`C_IH`	C_{IH} in Table 4.1	2
`INIT_TIME`	transaction initialization overhead	100
`CLOSE_TIME`	transaction closing overhead	100
`READ_TIME`	unit data object read time	10
`COMP_TIME`	unit data object processing time	5
`WRITE_TIME`	unit data object write time	20
`VALIDATION_TIME`	concurrency control overhead	100
`SLACK_CALC_COST`	slack calculation overhead	500
`SLACK_CALC_PERIOD`	slack calculation period	100

set. The value of each attribute is generated using an appropriate distribution function, for example, uniform, triangular or Pearson type V [37]. The arrival times of the Class III transactions follow an exponential random distribution over the test duration. The number of Class III transactions is varied to produce a range of total processor utilization levels (plotted on the x-axis of the graphs). That is, the workload generator generates transactions until the sum of execution times is not greater than the test duration multiplied by the target Class III utilization level. The mean inter-arrival time of Class III transactions is then the test duration divided by the number of generated Class III transactions.

5.1.2 Simulation Parameters

Table 5.1 shows the names and meanings of the parameters that control system resources. The values of `P_CLK`, `C_CLK`, `C_QL`, `C_QS`, and `C_IH` are fixed throughout the experiments as in the table, since their variation will affect the scheduling overhead of the system

Table 5.2: Workload Parameters

Parameter Name	Description	Base value
`util_ClassI`	utilization level of Class I transactions	–
`util_ClassII`	utilization level of Class II transactions	–
`util_ClassIII`	utilization level of Class III transactions	–
`MEAN_AVD`	average absolute validity duration	100000
`MEAN_RVD`	average relative validity duration	100000
`MEAN_TRANS_SIZE`	average transaction execution time	2500
`MEAN_PERIOD`	average transaction period	1000000
`MIN_SLACK`	minimum slack factor	2
`MAX_SLACK`	maximum slack factor	8
`WRITE_RATIO`	write probability of a Class III transaction	0.25

but will not change the relative performances of different scheduling algorithms. The parameters `INIT_TIME`, `CLOSE_TIME`, `READ_TIME`, `COMP_TIME`, `WRITE_TIME`, and `VALIDATION_TIME` are used to calculate the execution time of a transaction. For example, if a transaction τ updates N data objects, its execution time C_τ is `INIT_TIME` + N * (`READ_TIME` + `COMP_TIME` + `WRITE_TIME`) + `VALIDATION_TIME` + `CLOSE_TIME`. Since we use an optimistic approach, the cost of concurrency control mostly comes from the validation phase of a transaction processing and the parameter `VALIDATION_TIME` represents the cost. `SLACK_CALC_COST` is the overhead to calculate the slack at all priority levels in the HASS algorithm and `SLACK_CALC_PERIOD` is the period of the slack calculation in the PASS and HASS algorithms.

Table 5.2 summarizes the key parameters that characterize system workload and transactions. The sum of `util_ClassI` and `util_ClassII` will be one of the values between 0.3 and 0.7, and the value of `util_ClassIII` will be varied from 0.0 and 1.2 − `util_ClassI` − `util_ClassII` in each experiment. The parameters `MEAN_AVD`, `MEAN_RVD`, `MEAN_TRANS_SIZE`, and `MEAN_PERIOD` are used to generate random values of the corresponding transaction and data object attributes by a given distribution function (e.g., Pearson type V). The assignment of deadlines to Class II and Class III transactions is controlled by the parameters, MIN_SLACK and MAX_SLACK, which set a lower and upper bound, respectively, on a transaction's slack time. We use the following formula to assign a deadline to a transaction τ:

$$D_\tau = \text{uniform}(\texttt{MIN\_SLACK}, \texttt{MAX\_SLACK}) * C_\tau.$$

The base values for parameters shown in Tables 5.1 and 5.2 are not meant to model a specific real-time application. They were chosen to be reasonable for a wide range of actual database systems. The values of some parameters will be varied in each experiment, but the others are fixed throughout the base experiments. The effects of the parameters whose values are fixed in the base experiments will be explored in the separate experiments.

5.1.3 Performance Metrics

The key performance criteria used in this simulation study is the *deadline miss ratio* of Class III transactions. *Miss_Ratio* is calculated with the following equation:

$$\mathit{Miss\_Ratio} \;=\; \frac{\mathit{number\ of\ tardy\ jobs}}{\mathit{number\ of\ jobs\ arrived}}\ .$$

We use this criteria as it gives a measure of how early spare capacity can be made available and how efficiently the given spare capacity can be used for soft real-time transaction processing. The deadline miss ratio of Class I and II transactions must always be 0 in our scheduling framework, if our *off-line* schedulability analysis is correct, but it would not be the case for the conventional real-time transaction scheduling algorithms, where all the transactions are equally treated.

Other important performance metric is the *frequency of temporal consistency violation.* It must be 0 all the time if the static enforcement scheme presented in Chapter 4 is applied to the transaction set, but it would not always be 0 if that scheme is not employed.

5.2 Experiments and Results

The simulation program is written in C++. For each experiment, we ran the simulation with the same parameter values for at least 5 different random number seeds. Each run continues for up to 2,000,000,000 time units (= 2,000,000 ticks = 20,000 sec, if TICK = 10 ms). For each run, the statistics gathered during the first and the last 5% of the simulation time (1,000 sec) were discarded in order to collect the stabilized system state after initial and before terminal transient situation.

The experiments conducted for this study were designed to investigate the impact of our real-time database model, static enforcement of temporal consistency, integrated transaction scheduling, and semantic concurrency control scheme, compared to the conventional approaches. Hence, the following system configurations were implemented and tested for experiments:

- A conventional scheduling policy for soft/firm real-time transactions. We call this Blind policy, since it is not aware of each transaction's distinct semantics and requirements, although they are available in advance. That is, under the Blind scheduling policy, hard and soft deadlines cannot be discriminated, and all transactions are scheduled solely based on their arrival time (FCFS) or priorities (EDF, LSF).
- Our integrated scheduling scheme (Guaranteed) with the following variations:
    1. *Spare-Capacity Finding* (SCF) policy –
       a policy to exploit the spare capacity for Class III transaction execution in a fixed-priority preemptive system, in the context of our framework:
        (a) Background processing (Background) – the simplest and perhaps least effective approach is to execute Class III transactions at a lower priority level than any Class I and Class II transactions.
        (b) Optimal slack stealing (Optimal), described in Section A.3.
        (c) Approximate slack stealing (PASS, HASS), described in Section A.4.

2. *Class III scheduling* (CIII) policy – priority assignment and concurrency control algorithms for Class III transactions:
   (a) FCFS without concurrency control (FCFS/NO-CC)
   (b) EDF with concurrency control (EDF/CC)
   (c) LSF with concurrency control (LSF/CC)

We also investigate the effects of changes in certain parameters, such as write ratio (conflict ratio) and slack calculation overhead, on overall performance of the system.

Note that, since our research intention is not to compare the conventional transaction scheduling and concurrency control algorithms, but to investigate the impacts of their integration into our framework, we do not implement many of them, but choose some of the algorithms which are considered the best (and also practical in our framework) in each category. For example, we use the semantic concurrency control scheme based on OCC-PS (SOCC), described in Chapter 4, as the only concurrency control and conflict resolution policy in our simulation study, and do not consider any other policies like two-phase locking (2PL) based ones. Their performances are already investigated in other work, as mentioned in Chapter 2.

5.2.1 No Class I and II, but all Class III transactions

In this experiment there is no Class I and Class II transaction load. Since there are only Class III transactions in the system, this can reproduce the results of the past research on soft/firm real-time transaction scheduling.

Figure 5.1 shows the results for the three different priority assignment policies under SOCC. As reported in [1], EDF and LSF perform better than FCFS. At lower load settings, EDF perform close to LSF. As the load increases, the performance margin of EDF over FCFS narrows and LSF becomes the best policy. Obviously, this is because EDF assigns high priorities to transactions which have missed or are about to miss their deadlines. Nevertheless, we will not use LSF for Class III transactions in the following experiments, since most of the transactions in this class do not have the runtime estimates.

5.2.2 Cost of Timing Constraint Guarantee

The following experiments are performed to evaluate the cost of timing constraint guarantee for Class I and Class II transactions and the performance of the various SCF policies in our system.

Effect of Spare-Capacity Finding policy

Figures 5.2 through 5.7 show the results of several SCF scheduling policies under the EDF/SOCC Class III scheduling policy at three different utilization levels for Class I and II transactions. For comparison purposes, we include the result of the conventional real-time transaction scheduling policy (Blind) at each graph.

As seen in Figures 5.2, 5.4 and 5.6, under Blind some of the Class I transactions miss their deadlines, while under our integrated scheduling scheme (Guaranteed) all the Class I transactions meet their deadlines. Note that the deadline miss ratio of a Class II transaction is supposed to be not greater than its given guarantee ratio. It is achieved under Guaranteed, but not under Blind. In fact, the actual deadline miss ratio of Class II transactions is much lower than required (1.0 − guarantee level). In Figures 5.2 and 5.4, all the Class II transactions whose guarantee levels are 99% meet their deadlines, but in Figure 5.6 96% of Class II transactions whose guarantee levels are 80% meet their deadlines under Guaranteed. This is because a Class II transaction whose execution time budget is expired is processed in slack time, at the expense of Class III transactions.

We also observe that the performances of the Class I and Class II transactions under Guaranteed are not affected by the Class III transaction loads; however, this is not the case under Blind.

Figures 5.3, 5.5 and 5.7 show the deadline miss ratio of Class III transactions under various scheduling policies. At all utilization level of Class I and Class II transactions, the Blind scheduling shows the better performance for Class III transactions than any other Guaranteed schemes, but at the expense of Class I and Class II transactions. Class III transactions under Guaranteed schemes miss more deadlines than under Blind, since they are scheduled using only spare capacity after the off-line guarantee on Class I and Class II transactions. We thus consider this performance margin between Blind and Guaranteed as the cost of guarantee on the timing constraints of Class I and II transactions. For example, the cost of guarantee of HASS at 70% of total utilization in Figure 5.5 is about 0.18 (i.e., 18% more Class III deadline miss). We can see that the cost of guarantee increases as the Class I and Class II transaction load gets higher (from 30% to 70%). Obviously, there are more chances to meet the deadlines of Class III transactions as more slack times are available for them at low utilization level of Class I and Class II transactions.

Among the different SCF policies, Background performs the worst and Optimal slack stealing shows the best performance at all transaction loads, as expected. The performance degradation of PASS over Optimal is due to its periodic calculation of slack, and the performance of HASS suffers comparing to PASS, since it includes the slack calculation overhead. We also observe that the relative performances among slack stealing algorithms are not much varied at different Class I and Class II transaction loads.

Effect of Priority Assignment Policy

Figures 5.8, 5.9, and 5.10 show the results of different priority assignment policies for Class III transactions. In all cases, FCFS performs the worst and LSF performs the best. The performance margin between FCFS and EDF gets smaller as the total utilization level gets higher under the Blind policy, since EDF gets worse more rapidly. However, it does not seem to be the case under our Guaranteed policy. That is, EDF shows relatively better performance under Guaranteed than under Blind.

The relative cost of guarantee seems to be similar among different Class III scheduling policies at all Class I and Class II transaction load, except one case: Class III transactions may meet more deadlines under the Guaranteed policy than the Blind policy (no cost of guarantee) if they are scheduled by FCFS and the combined load of Class I and Class II transaction is low (Figure 5.8).

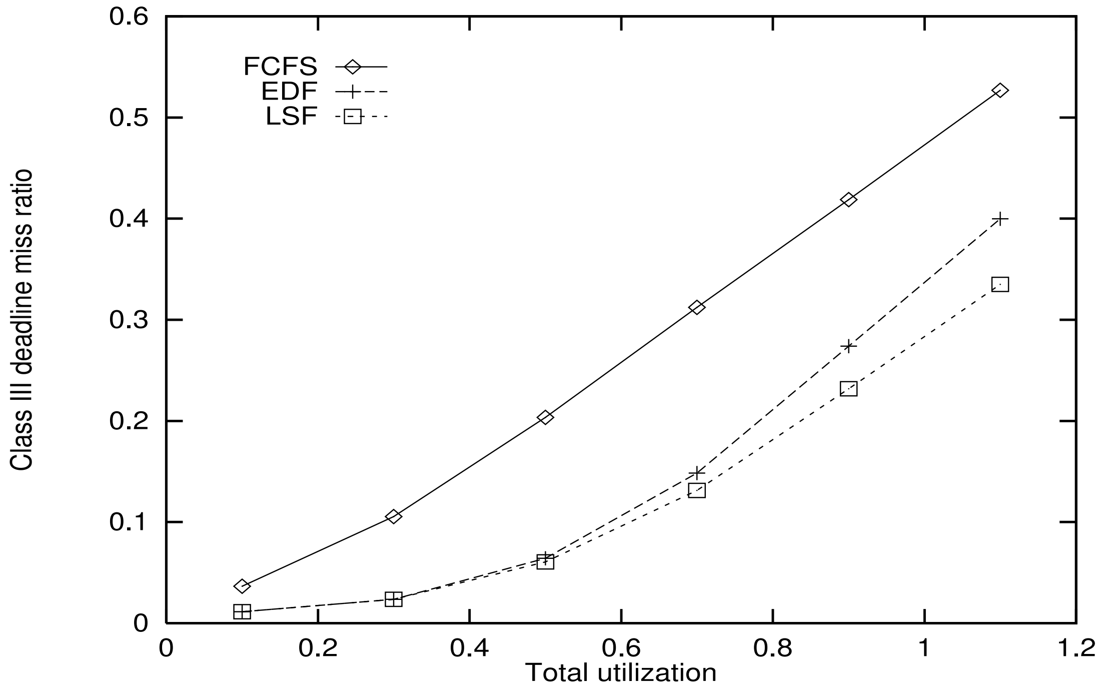


Figure 5.1: No Class I and II, Only Class III Transactions, SOCC

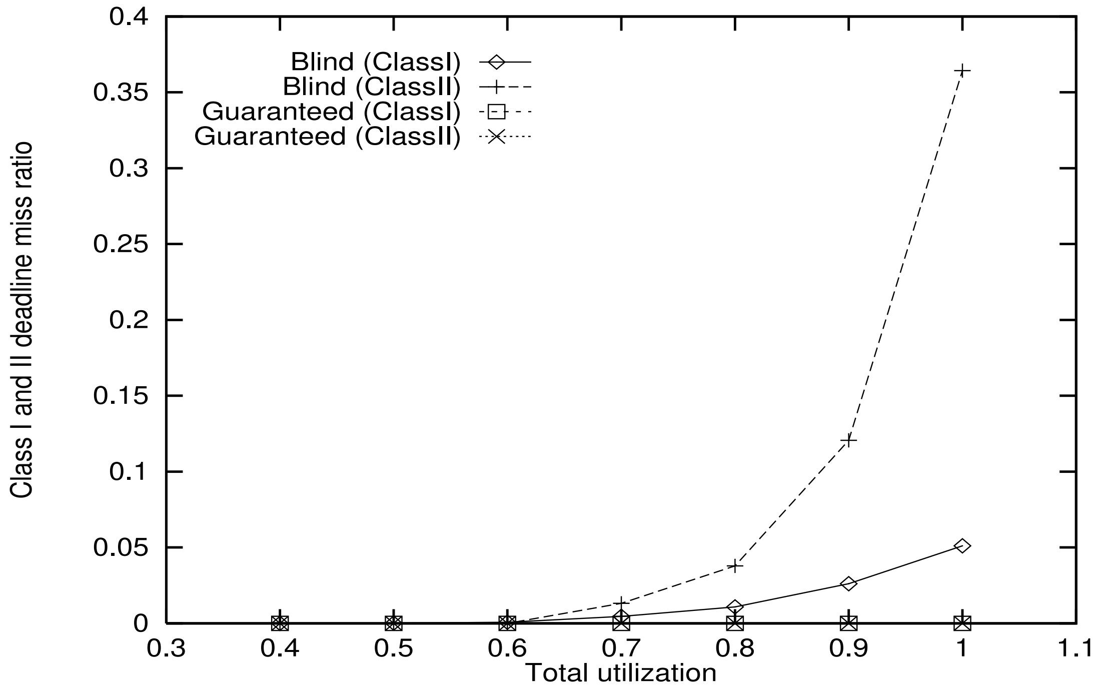


Figure 5.2: 30% Class I and II, 99% Class II Guarantee Level, EDF

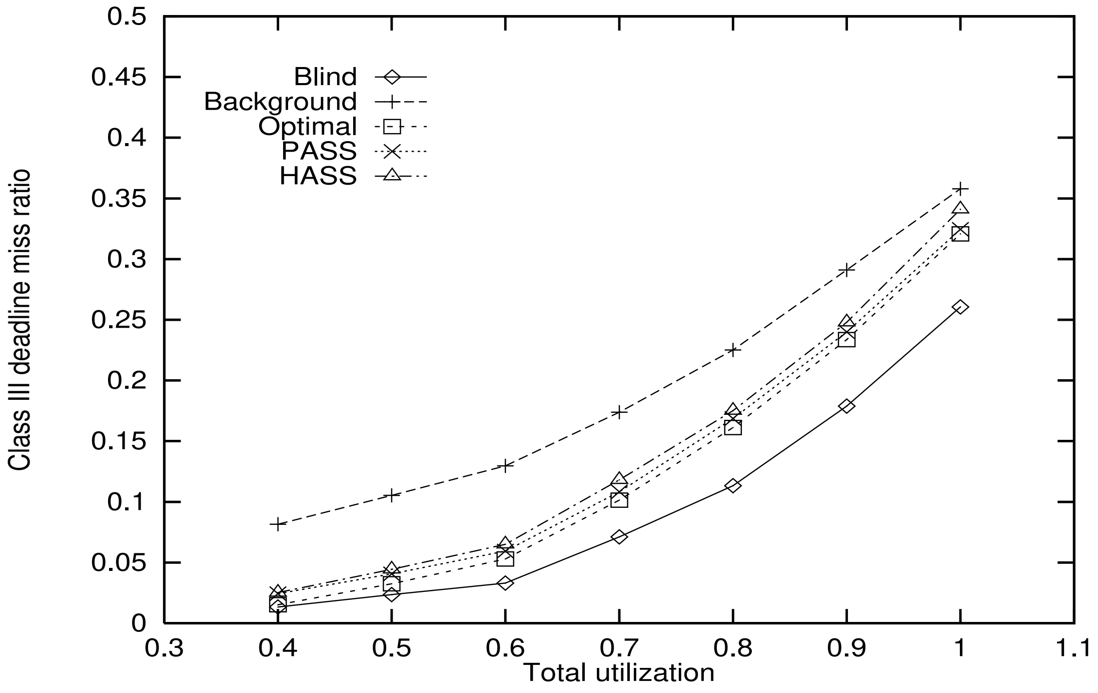


Figure 5.3: 30% Class I and II Utilization, EDF

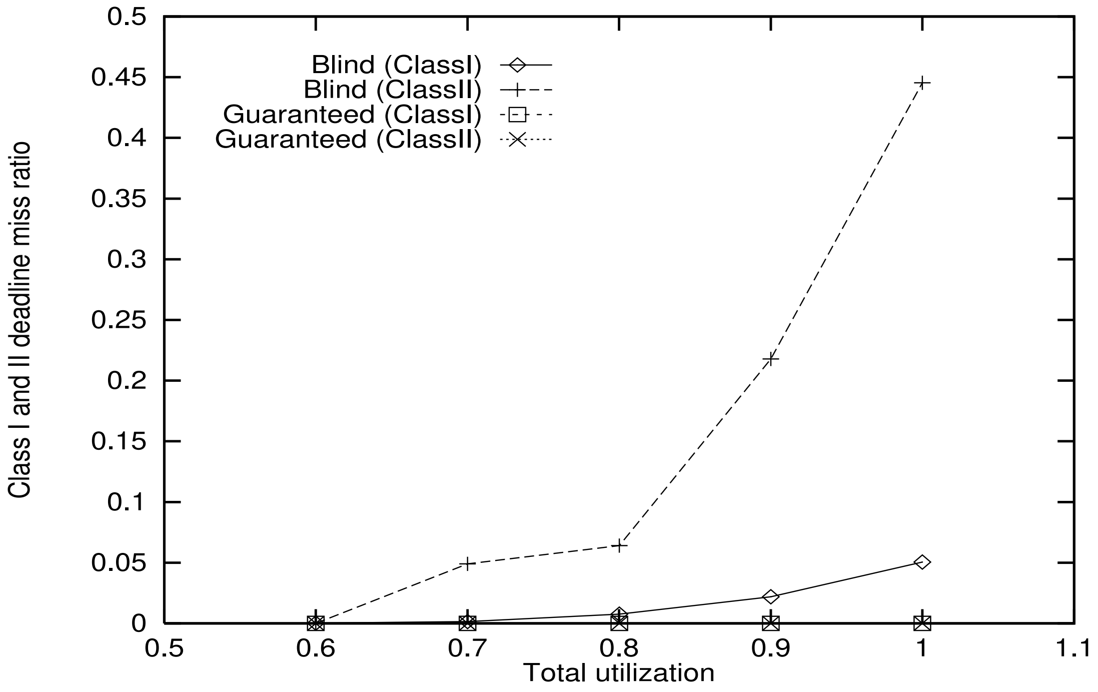


Figure 5.4: 50% Class I and II, 99% Class II Guarantee Level, EDF

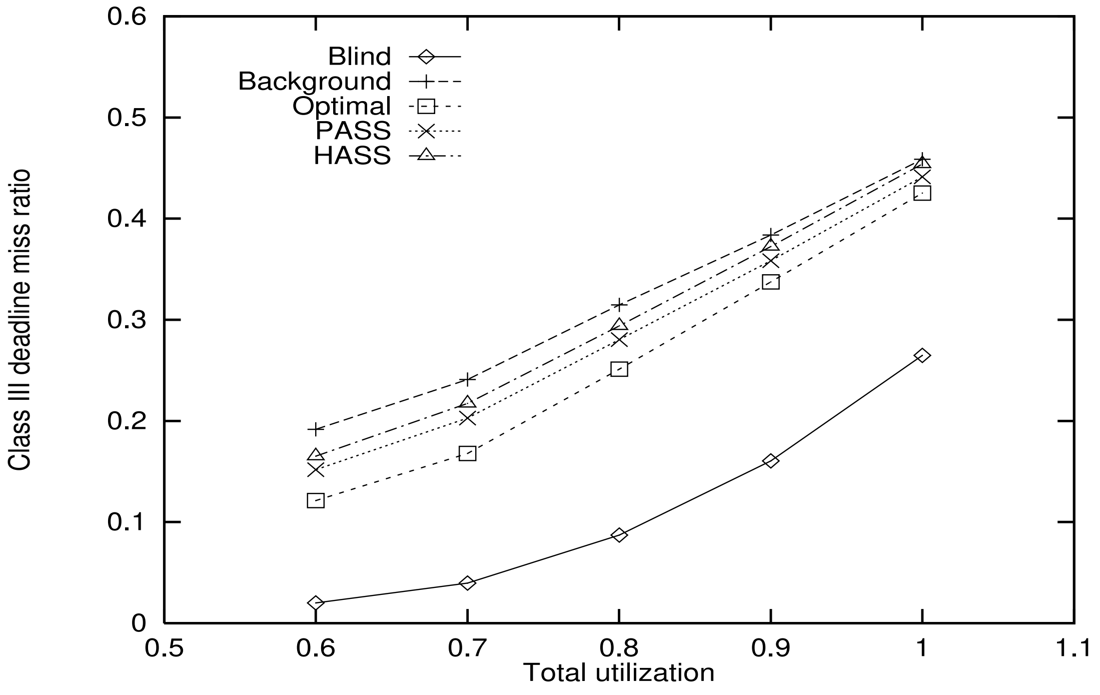


Figure 5.5: 50% Class I and II Utilization, EDF

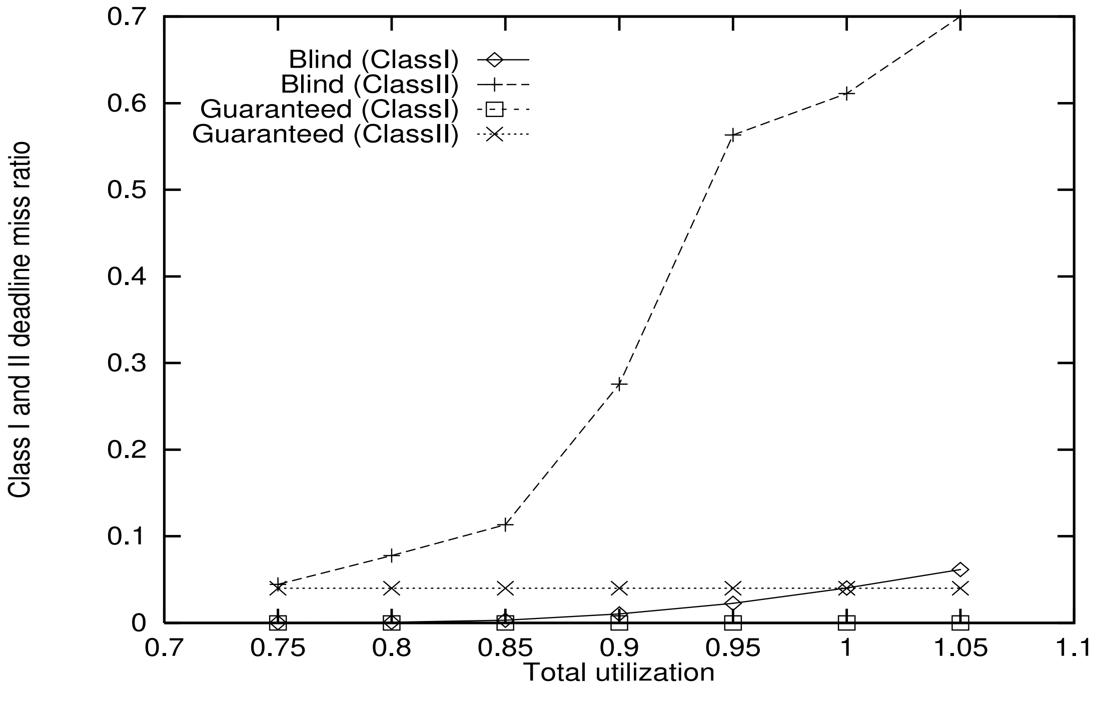


Figure 5.6: 70% Class I and II, 80% Class II Guarantee Level, EDF

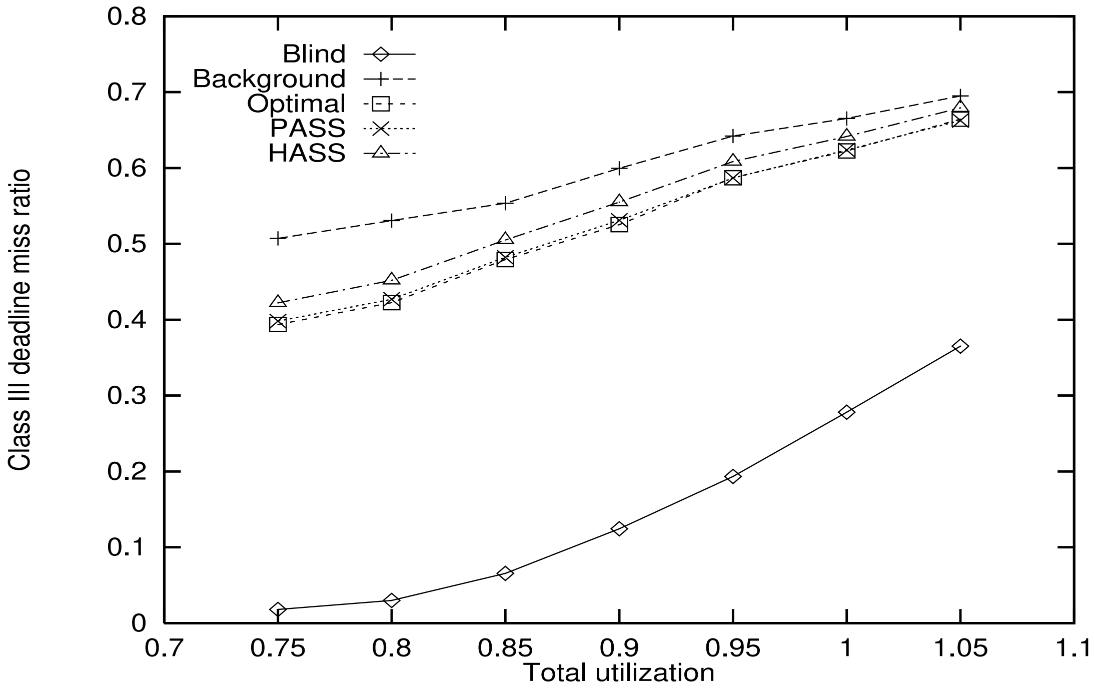


Figure 5.7: 70% Class I and II Utilization, EDF

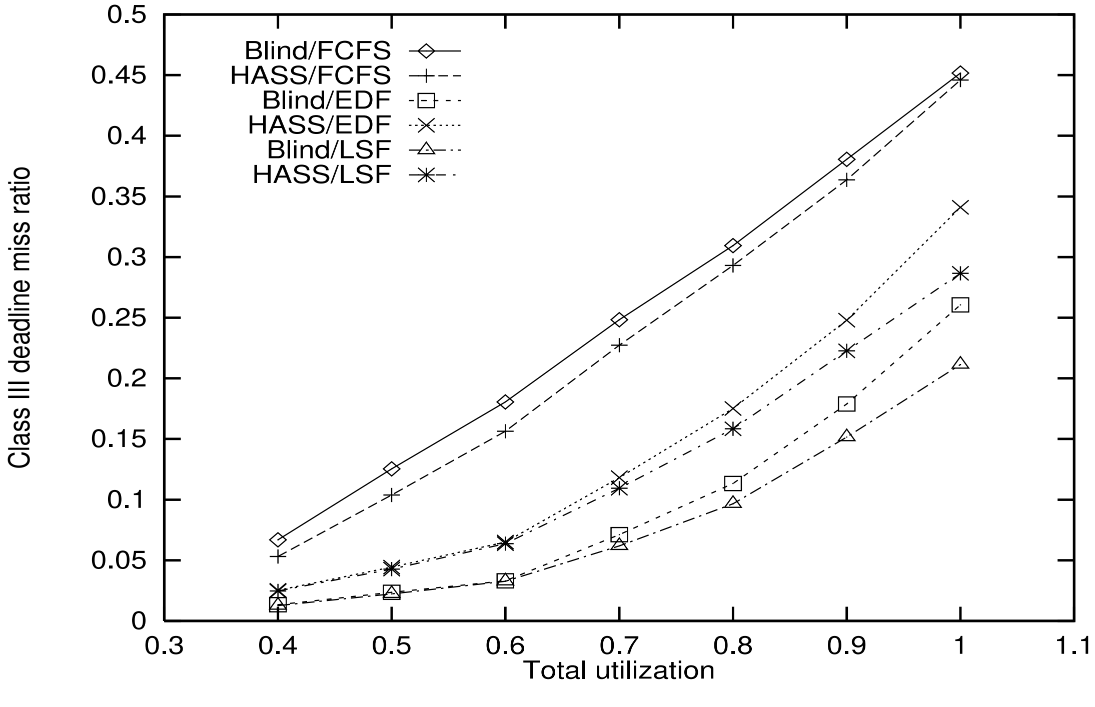


Figure 5.8: 30% Class I and II Utilization

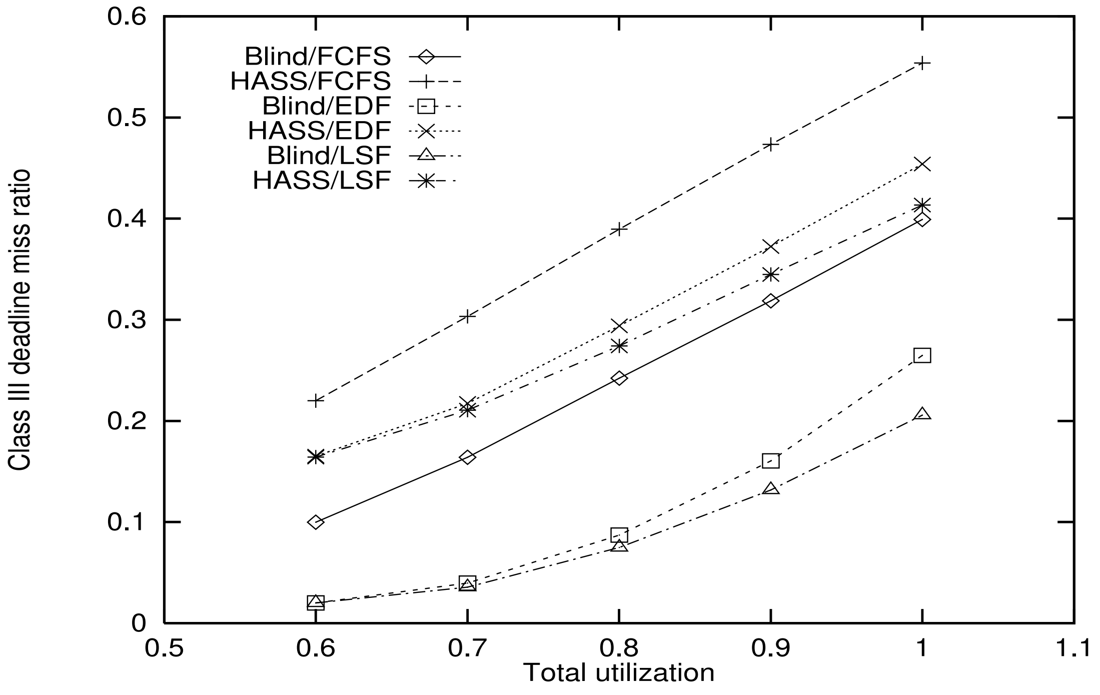


Figure 5.9: 50% Class I and II Utilization

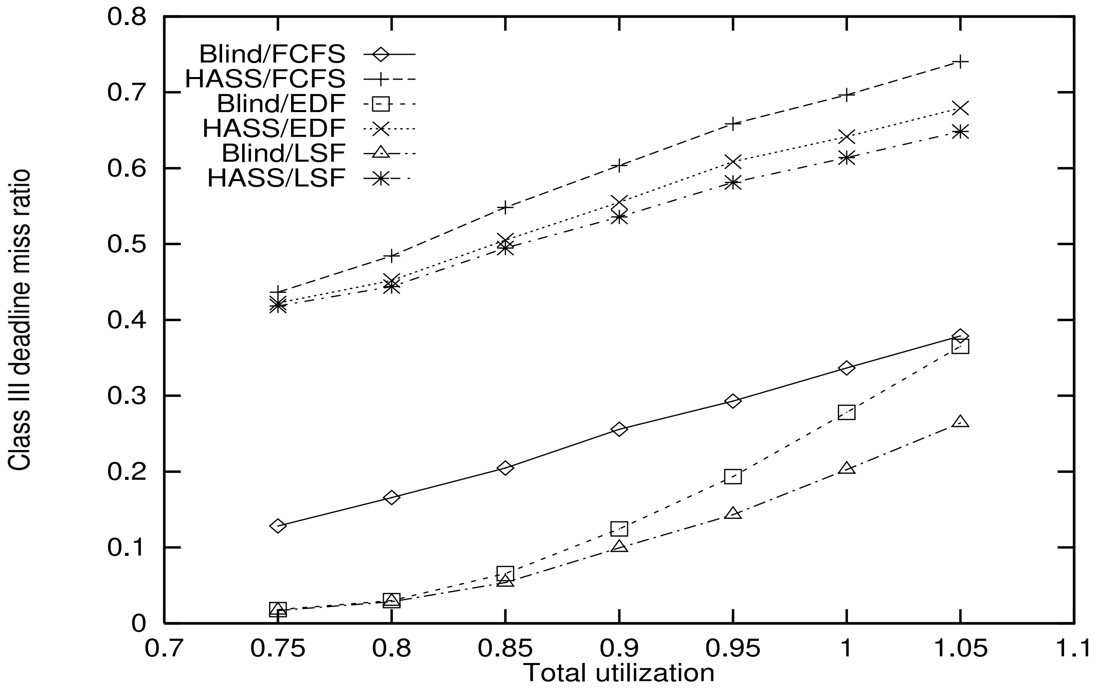


Figure 5.10: 70% Class I and II Utilization

5.2.3 Cost of Temporal Consistency Guarantee

As we discussed in Section 4.1, enforcing the temporal consistency requirements of real-time database statically may result in higher load of Class I transactions and consequently greater deadline miss ratio of Class III transactions. In order to evaluate the cost of our static temporal consistency enforcement scheme (STCE) at various situations, we conducted experiments on three different test applications: the utilization levels of the original sets of Class I and Class II transactions (No STCE) are 9.5%, 30.2%, and 35.6%, and those of the modified sets according to STCE (STCE) are 19.1%, 31.2%, and 42.7%, respectively. That is, the increased utilization level of Class I and Class II transactions due to the STCE scheme varies at each test case. In all three experiments, we use EDF/SOCC as the CIII policy and HASS as the SCF scheme.

As seen in Figures 5.14, 5.15, and 5.16, the temporal consistency is always maintained by the static temporal consistency enforcement scheme under our integrated scheduling mechanism (STCE/Guaranteed). However, if we don't apply STCE to the given set of Class I and Class II transactions (No STCE/Blind and No STCE/Guaranteed), the significant number of transactions will violate the temporal consistency requirements. Even though the transaction set satisfies the STCE conditions, the temporal consistency cannot be guaranteed if one of the Guaranteed scheduling policies is not used (STCE/Blind).

We can also observe that the bigger increment of the Class I and II load due to STCE results in the higher violation ratio of temporal consistency. In Figure 5.14 (Figure 5.15), the Class I and II utilization increases by about 10% (1%), and then about 70% (15%) of Class I transactions violate the temporal consistency requirements. The Class III transaction load does not affect the temporal consistency violation ratio.

In Figures 5.11, 5.12, and 5.13, the performance difference between STCE/Guaranteed and No STCE/Guaranteed represents the cost of the static temporal consistency guarantee. The total cost of guarantee on timing constraints and temporal consistency requirements is shown as the performance margin between STCE/Guaranteed and No STCE/Blind in the graphs.

Unlike the temporal consistency violation ratio, the performance of Class III transactions does not seem to be affected by the amount of the increased Class I and II utilization, but it looks sensitive to the combined utilization level of Class I and II transactions. As the utilization level of Class I and II transactions gets higher, the cost of guarantee increases.

5.2.4 Effects of Various Parameter Settings

In the previous experiments, we used the fixed values for parameters SLACK_CALC_COST, SLACK_CALC_PERIOD, VALIDATION_TIME, and WRITE_RATIO, as in Tables 5.1 and 5.2. In the following experiments, we investigate the impacts of these parameters on the overall system performance. The combined utilization of Class I and II transactions is set to 50% throughout the following experiments.

Varying Slack Calculation Overhead

We did not perform the experiments for PASS with the various SLACK_CALC_PERIOD values, since the impact of this parameter has been already studied in [12]. Typically, to achieve close to optimal performance, the slack calculation period (SLACK_CALC_PERIOD) in the PASS algorithm needs to be the same order of magnitude as the shortest minimum inter-arrival time of any Class I or II transaction. Increasing the period of PASS results in increased Class III deadline miss ratio. With a period close to the longest minimum inter-arrival time of any Class I or Class II transaction, performance may be little better than the Background processing.

Figures 5.17, 5.18, and 5.19 show the results of the HASS algorithms with the various SLACK_CALC_PERIOD values under the EDF/SOCC Class III scheduling policy. Since under the HASS algorithm the slack calculation is only performed in slack time, the performance of Class III transactions suffers at short SLACK_CALC_PERIOD values (e.g., less than 50 ticks). As illustrated in the graphs, we can hardly determine a single SLACK_CALC_PERIOD value which performs the best at all transaction loads. It heavily depends on the characteristics of the given Class I and II transaction set. Fortunately, the performances of HASS with the reasonably long SLACK_CALC_PERIOD values are comparable to each other.

Cost of Serializability

We can use the results of the experiment where concurrency control was turned off to understand how the enforcement of serializable schedules affects performance in terms of missed deadlines. Figure 5.20 shows the performance of serialized (CC) and unserialized (No CC) versions for EDF. The unserialized versions perform better than the serialized version for each algorithms. This performance difference represents the cost of serializability (i.e., cost of logical consistency) and shows the efficiency of the concurrency control algorithm.

Increasing Conflicts

In this experiment we varied the value of `WRITE_RATIO` from 0.0 to 1.0 in increments of 0.25. Since transactions access the same number of objects, the probability of conflict gets higher when `WRITE_RATIO` increases. Thus we can see how our semantic concurrency control (SOCC) algorithm performs as the number of conflicts changes.

The experiment results show that at low `WRITE_RATIO` (Figure 5.21) EDF performs much better than FCFS. As `WRITE_RATIO` increases, however, the performance margin between FCFS and EDF gets smaller (Figures 5.22 and 5.23), and finally FCFS starts to perform better than EDF at an extreme `WRITE_RATIO` value (Figure 5.24). This is because the number of restarts (and thus the number of aborts) increases as the probability of conflicts grows under EDF, while FCFS (i.e., serial execution) does not suffer from the conflicts.

One noteworthy observation is that the performance margin between FCFS and EDF under our Guaranteed scheduling (i.e., HASS/FCFS vs HASS/EDF) is more rapidly closing than under the conventional Blind scheduling (i.e,. Blind/FCFS vs Blind/EDF), as `WRITE_RATIO` increases.

Increasing Concurrency Control Overhead

In order to see how the overhead of our SOCC algorithm affects the overall system performance, we varied the value of `VALIDATION_TIME` from 0 to 1,000 (10ms), since the validation phase is the main source of overhead in optimistic concurrency control algorithms. Thus, in

Table 5.3: Utilization Increment due to CC Overhead

Total utilization (Class III utilization)	0.6 (0.1)	0.7 (0.2)	0.8 (0.3)	0.9 (0.4)	1.0 (0.5)
`VALIDATION_TIME` = 100 (1ms)	0.005	0.01	0.015	0.02	0.024
`VALIDATION_TIME` = 500 (5ms)	0.055	0.05	0.075	0.1	0.12
`VALIDATION_TIME` = 1000 (10ms)	0.05	0.1	0.015	0.2	0.24

this experiment, we can see how the performance of EDF/SOCC suffers as the concurrency control (CC) overhead increases, comparing to that of FCFS without concurrency control overhead.

Figures 5.25, 5.26, 5.27, and 5.28 show the results of the experiment at four different CC overhead levels. Beware that these graphs may be misleading, since the lines for EDF contain the CC overheads. That is, with the same set of Class III transactions, the CPU utilization of the transactions under EDF/SOCC is greater than that of under FCFS without any such overhead. We must consider this utilization increment when we read the graphs.

Since the unit overhead (`VALIDATION_TIME`) is involved in each Class III transaction, the utilization increment due to CC overhead can be calculated as `VALIDATION_TIME` divided by the mean inter-arrival time of Class III transactions at the given utilization level. Table 5.3 shows some of the calculated values at each `VALIDATION_TIME` value.

For example, in Figure 5.27, the Class III deadline miss ratio at the total utilization 0.9 under HASS/EDF is 0.35 and the utilization increment due to CC overhead is 0.1. Thus, the comparable total utilization under HASS/FCFS is 0.8 (= 0.9 − 0.1), and the Class III deadline miss ratio at this level is 0.29 (FCFS performs better than EDF!).

If we read the graphs in this way, we can observe that at low CC overhead (= 100, 5% of the mean execution time of a Class III transaction), EDF performs better than FCFS under our Guaranteed scheduling algorithms (HASS in this experiment). However, as the CC overhead increases the performance of EDF suffers and at some point (= 500, 25% of the mean execution time of a Class III transaction), it starts to perform worse than that of FCFS. In other words, we can conclude that there is no benefit from the priority-based Class III scheduling algorithm (EDF) if the concurrency control overhead is too high.

One interesting observation to note is that EDF always performs better than FCFS under the conventional Blind scheduling regardless of the CC overhead.

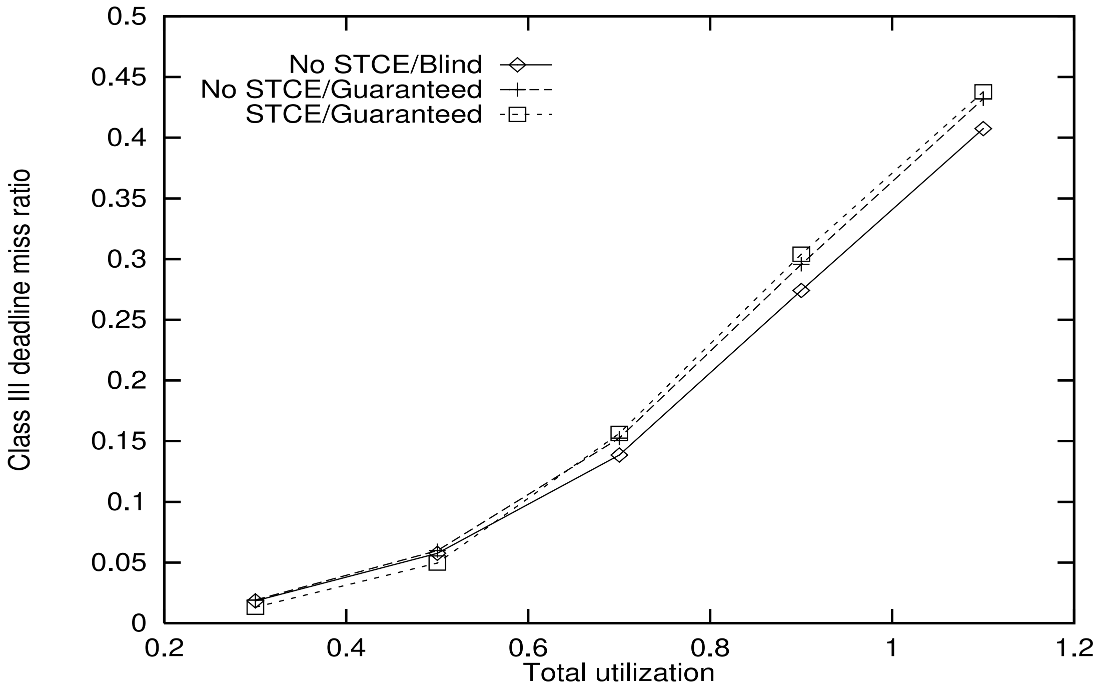


Figure 5.11: Combined Load of Class I and II: No STCE=9.5%, STCE=19.1%

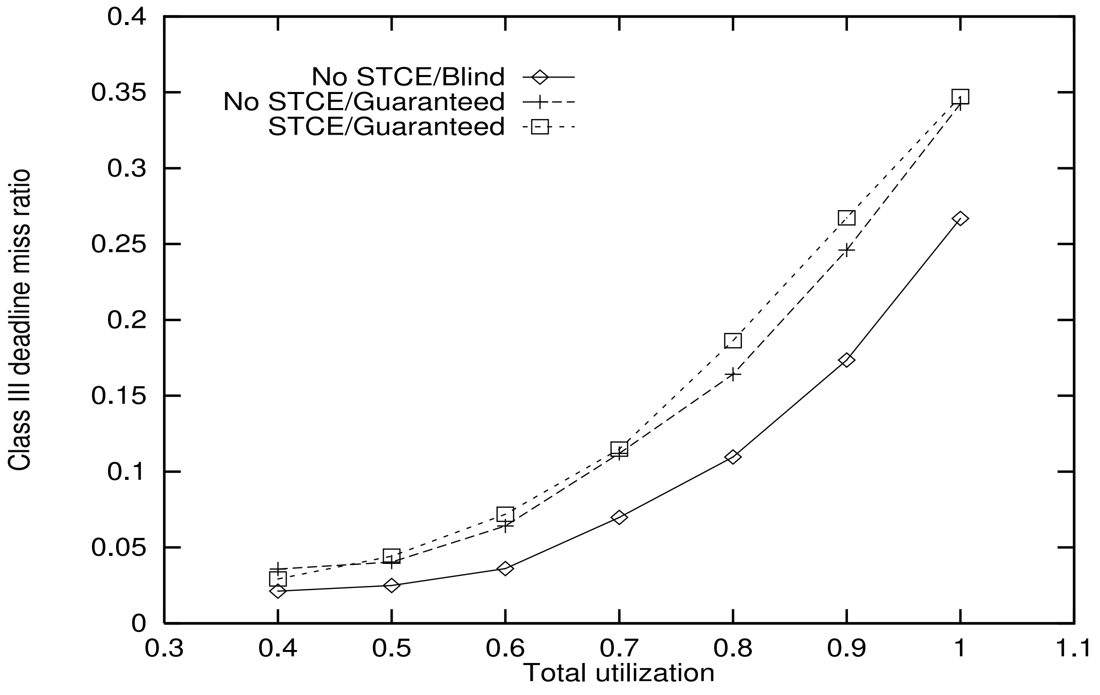


Figure 5.12: Combined Load of Class I and II: No STCE=30.2%, STCE=31.2%

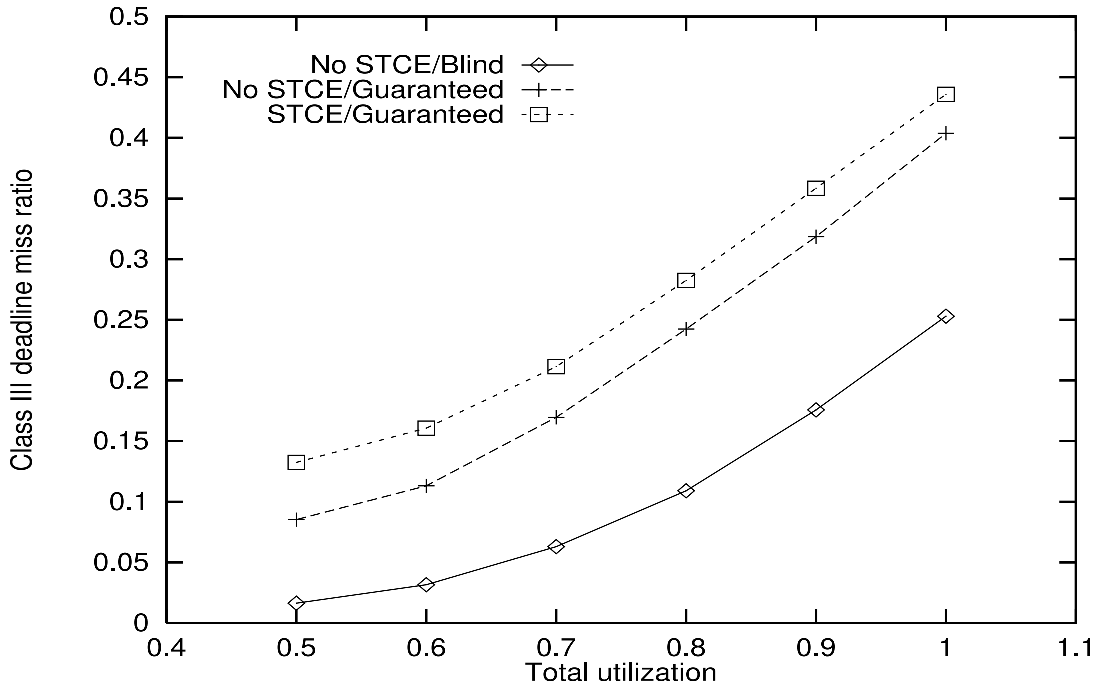


Figure 5.13: Combined Load of Class I and II: No STCE=35.6%, STCE=42.7%

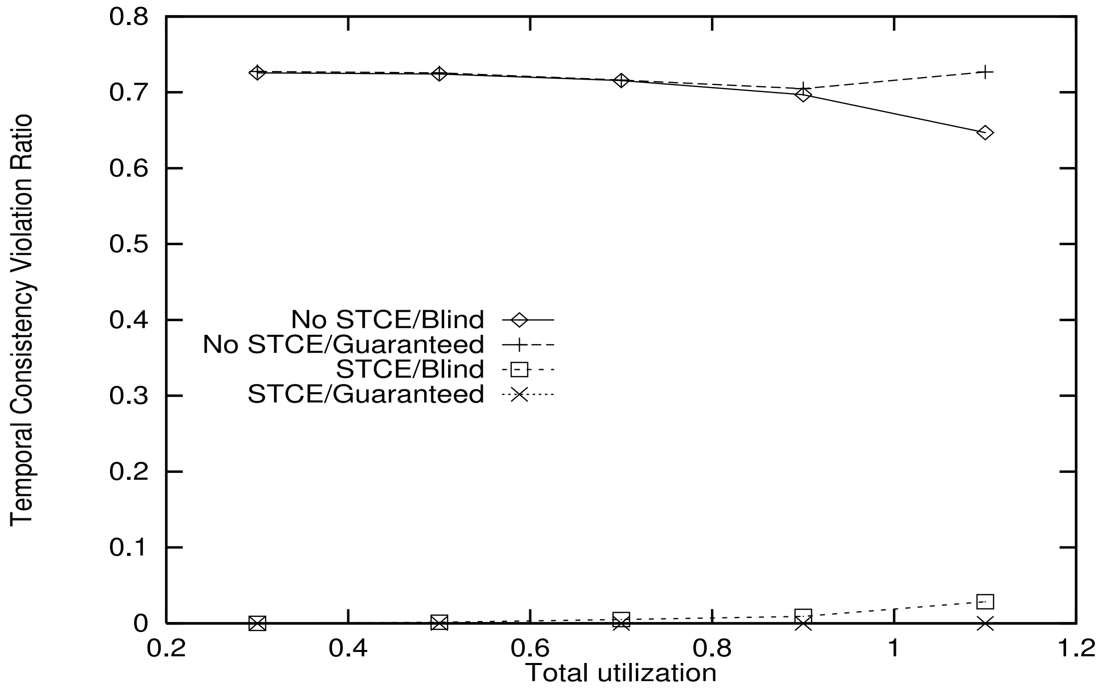


Figure 5.14: Combined Load of Class I and II: No STCE=9.5%, STCE=19.1%

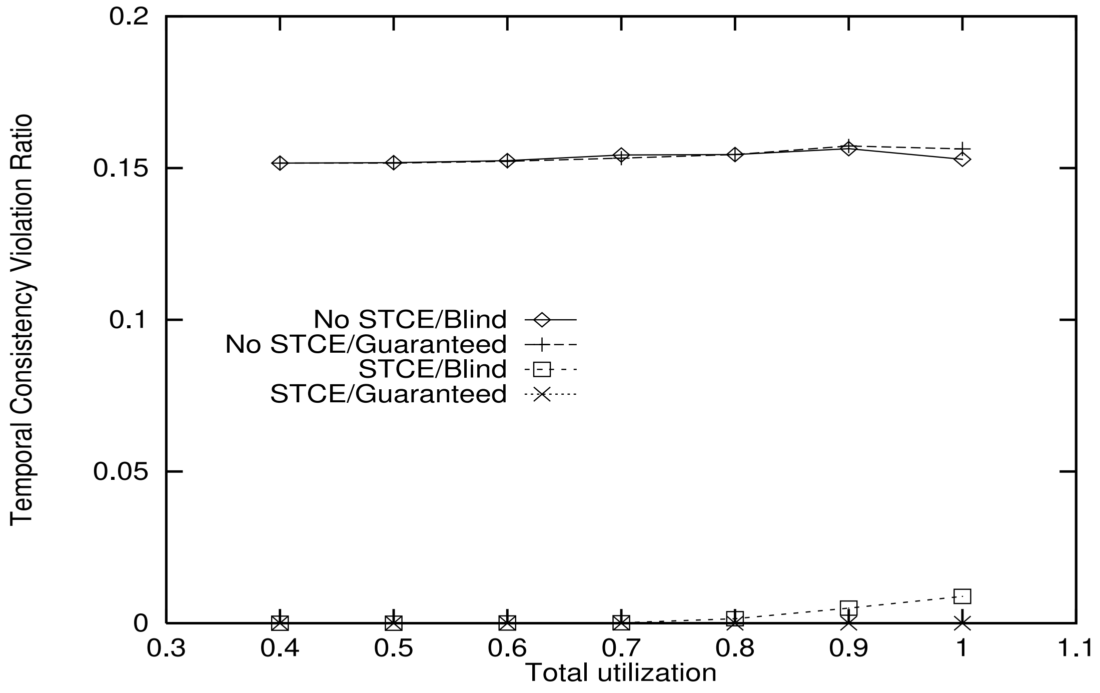


Figure 5.15: Combined Load of Class I and II: No STCE=30.2%, STCE=31.2%

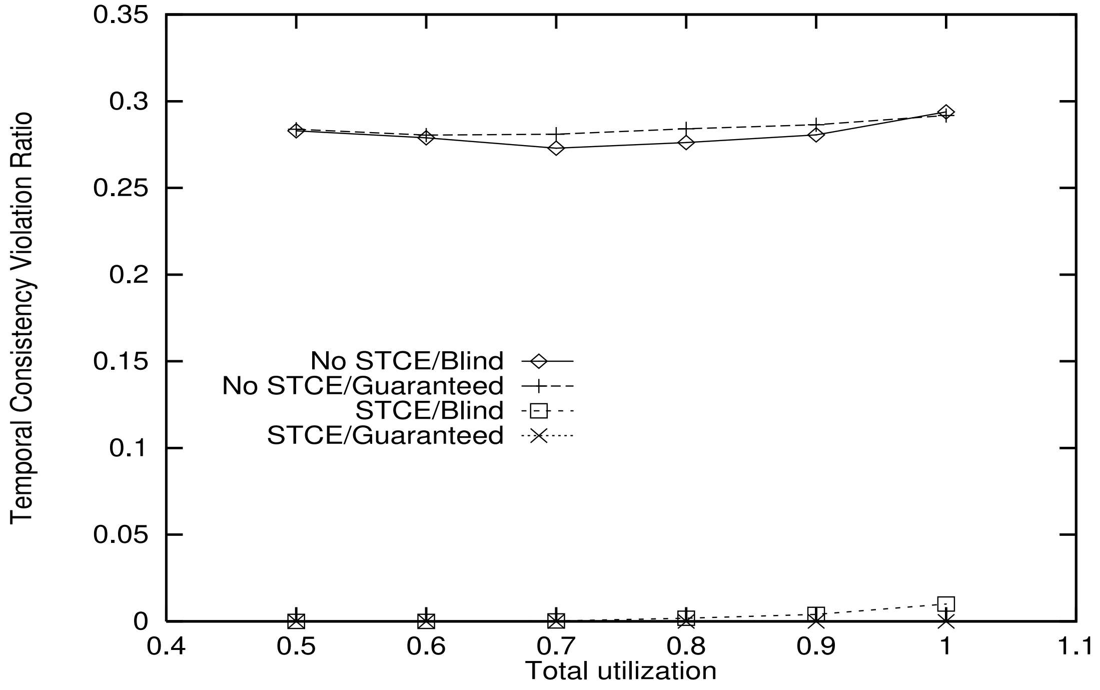


Figure 5.16: Combined Load of Class I and II: No STCE=35.6%, STCE=42.7%

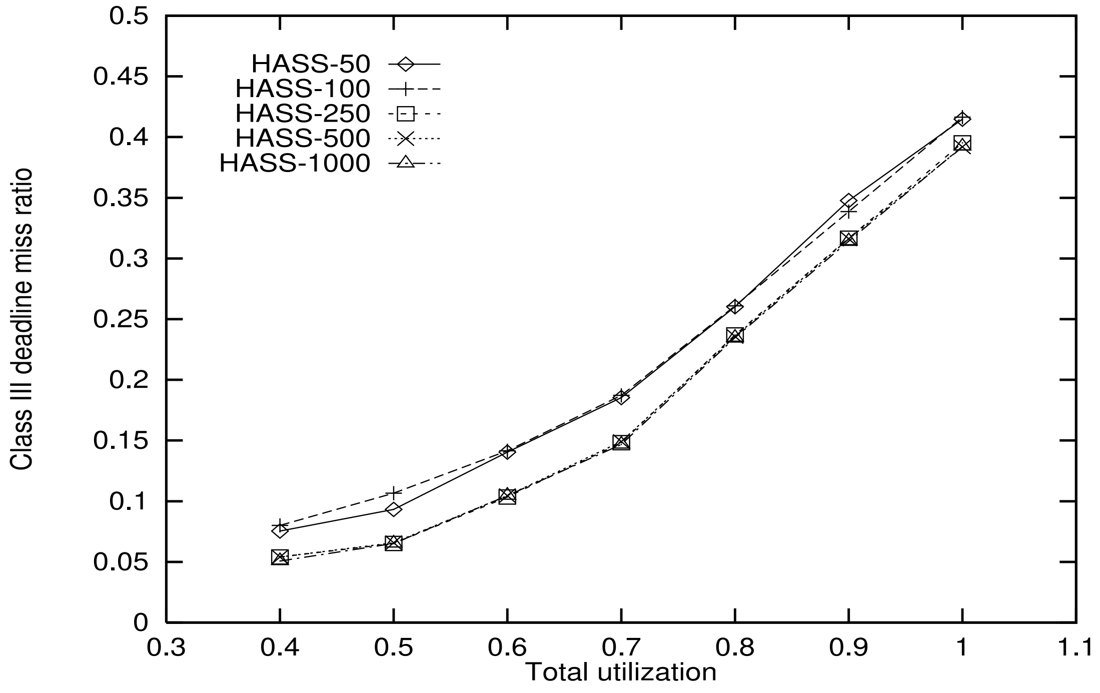


Figure 5.17: 30% Class I and II Utilization, EDF/SOCC

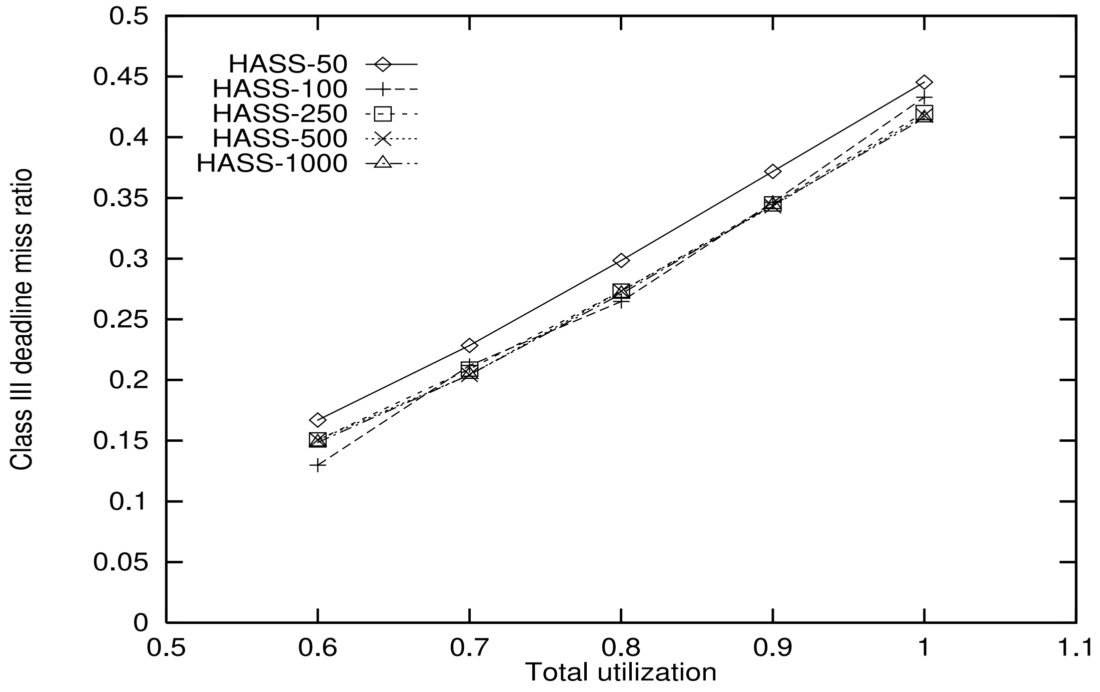


Figure 5.18: 50% Class I and II Utilization, EDF/SOCC

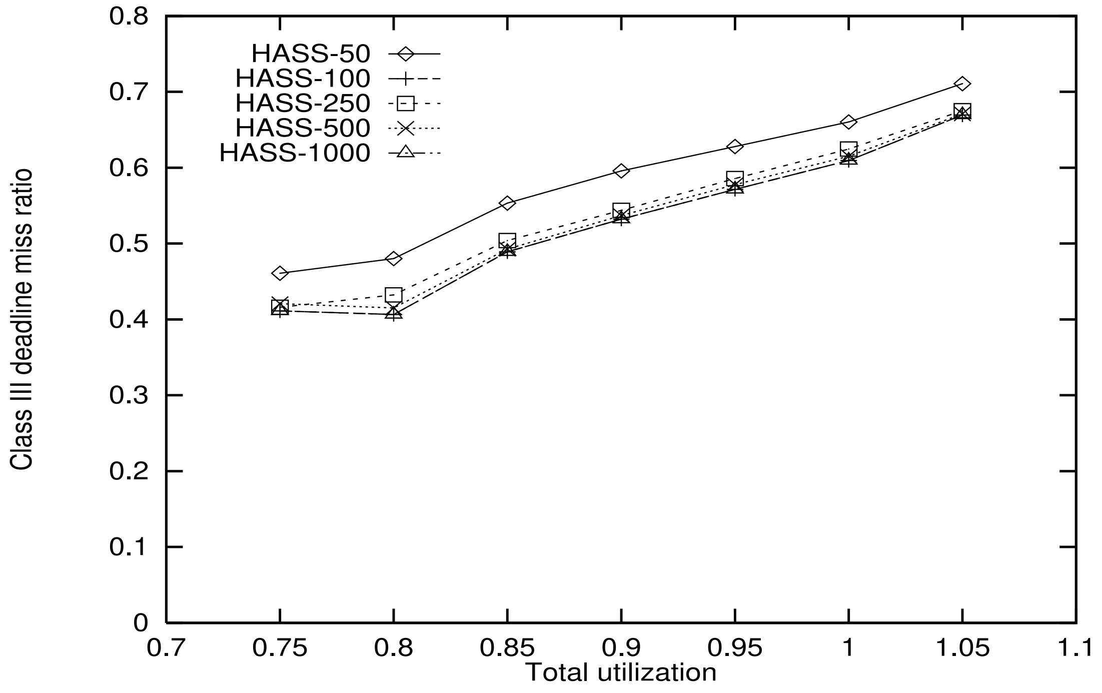


Figure 5.19: 70% Class I and II Utilization, EDF/SOCC

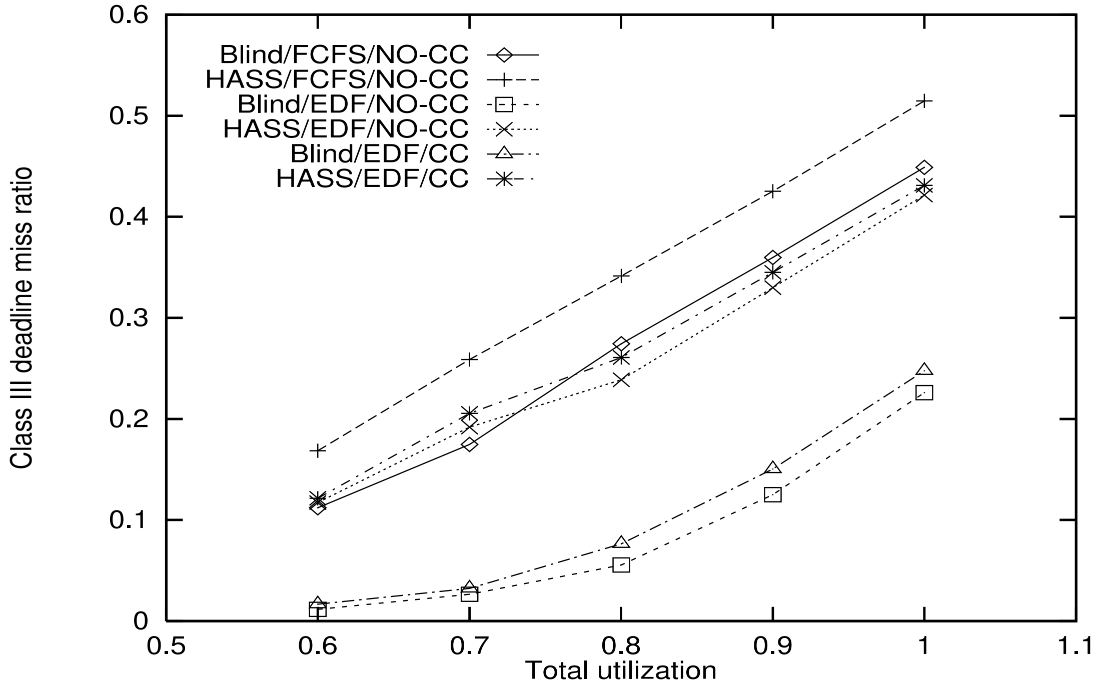


Figure 5.20: 50% Class I and II Utilization, No CC vs CC

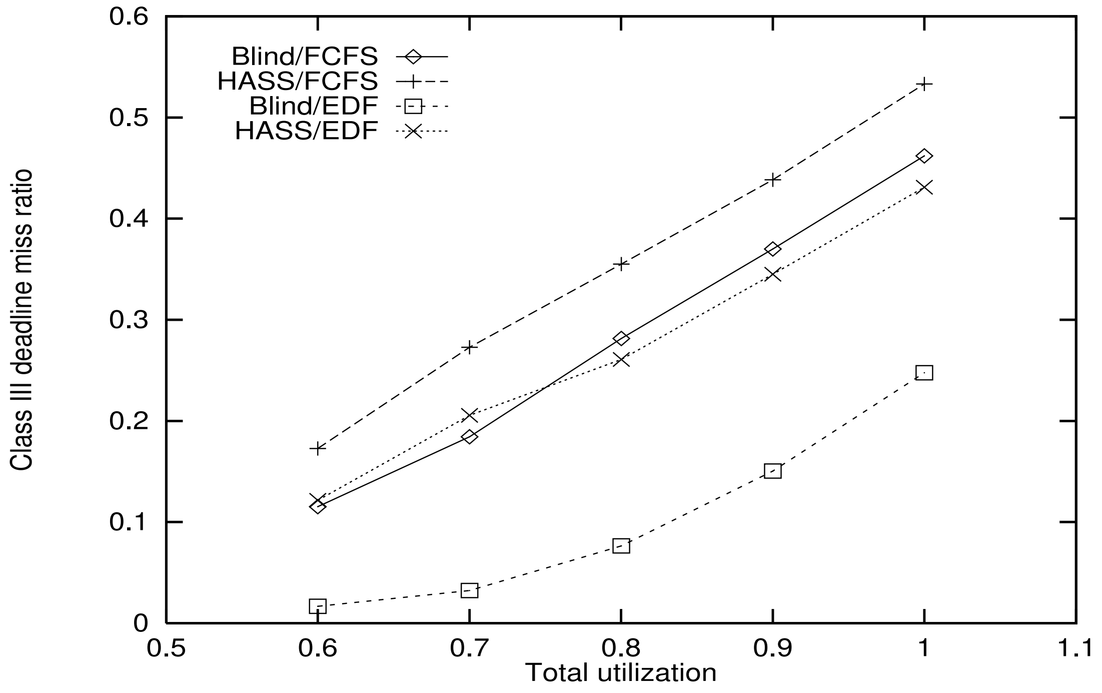


Figure 5.21: 50% Class I and II Utilization, `WRITE_RATIO` = 0.25

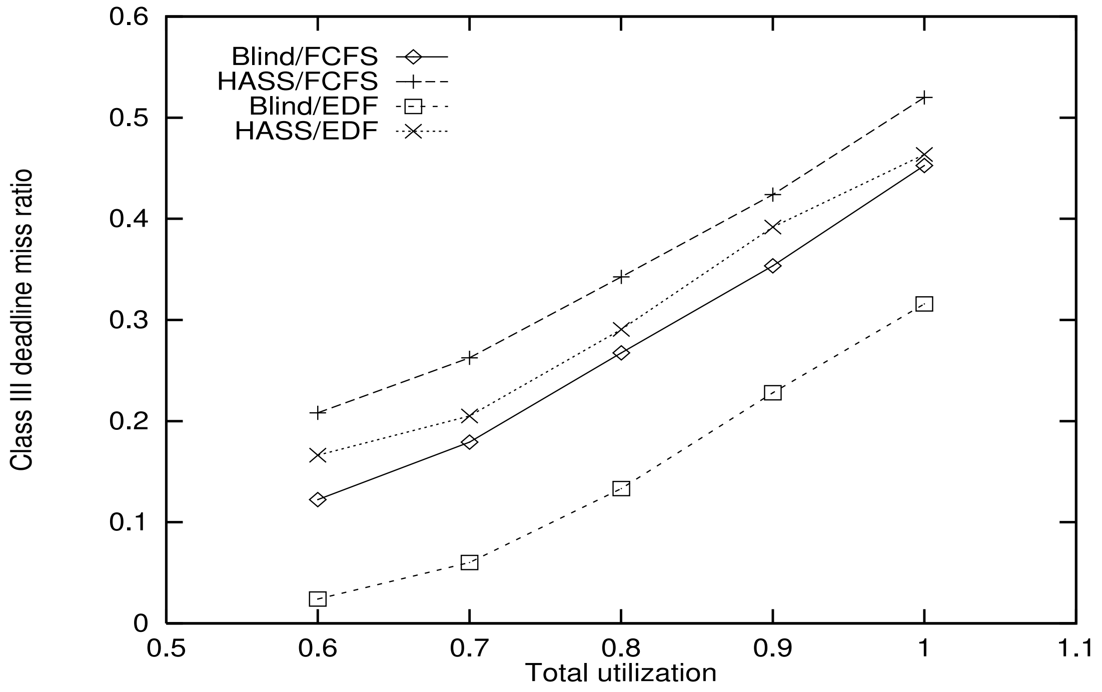


Figure 5.22: 50% Class I and II Utilization, `WRITE_RATIO` = 0.5

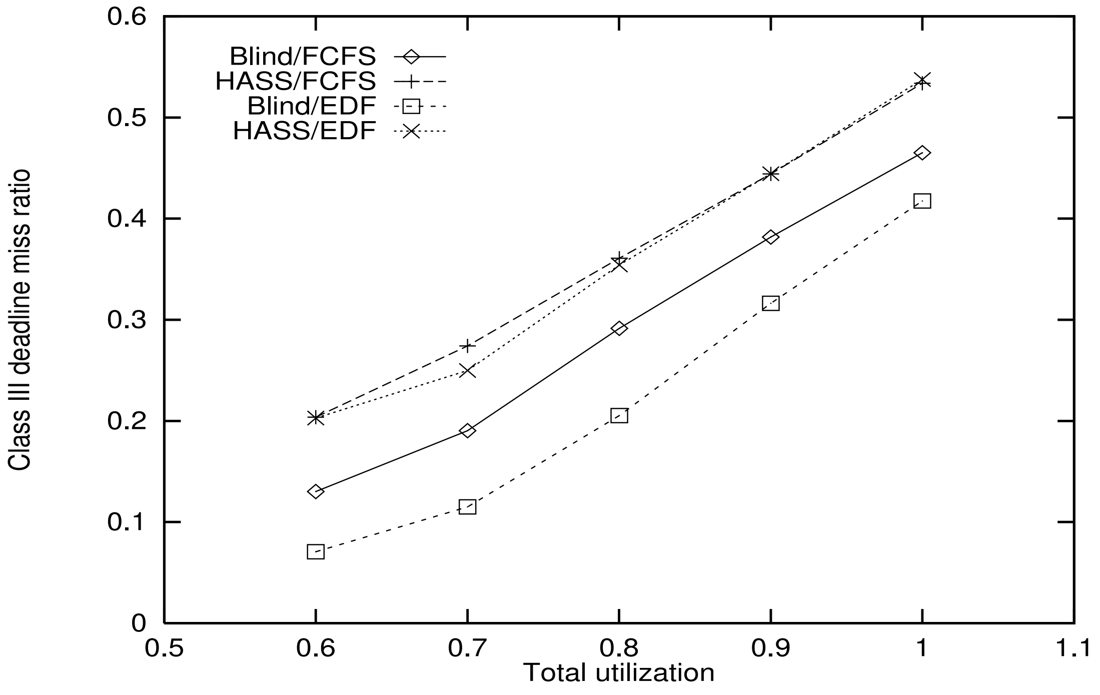


Figure 5.23: 50% Class I and II Utilization, `WRITE_RATIO` = 0.75

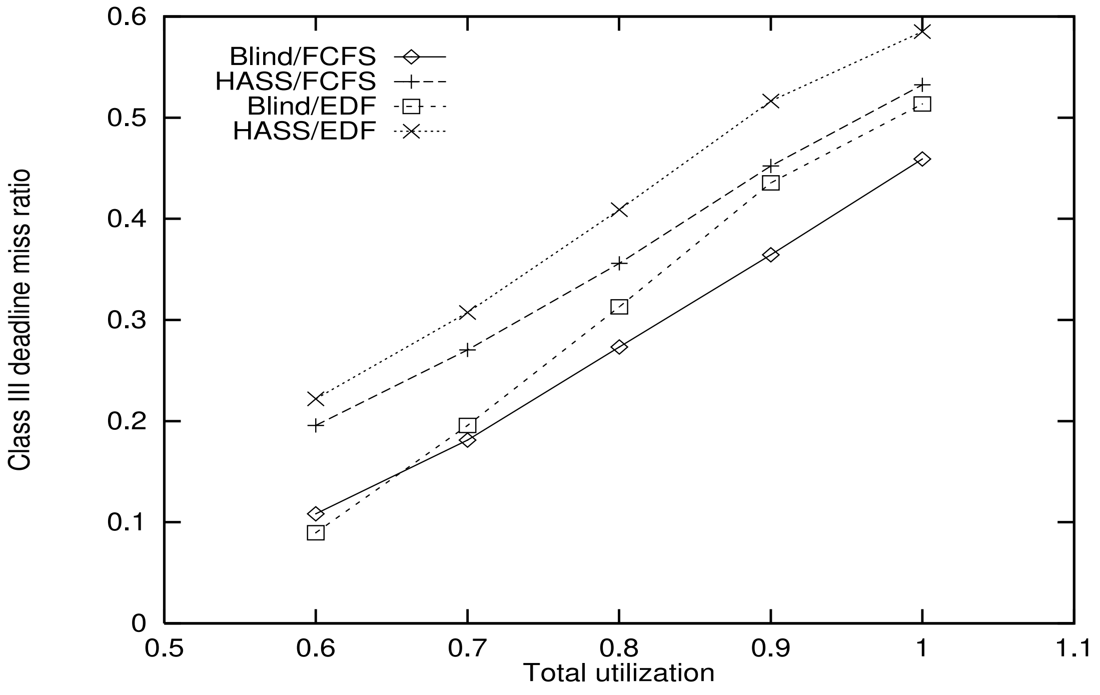


Figure 5.24: 50% Class I and II Utilization, `WRITE_RATIO` = 1.0

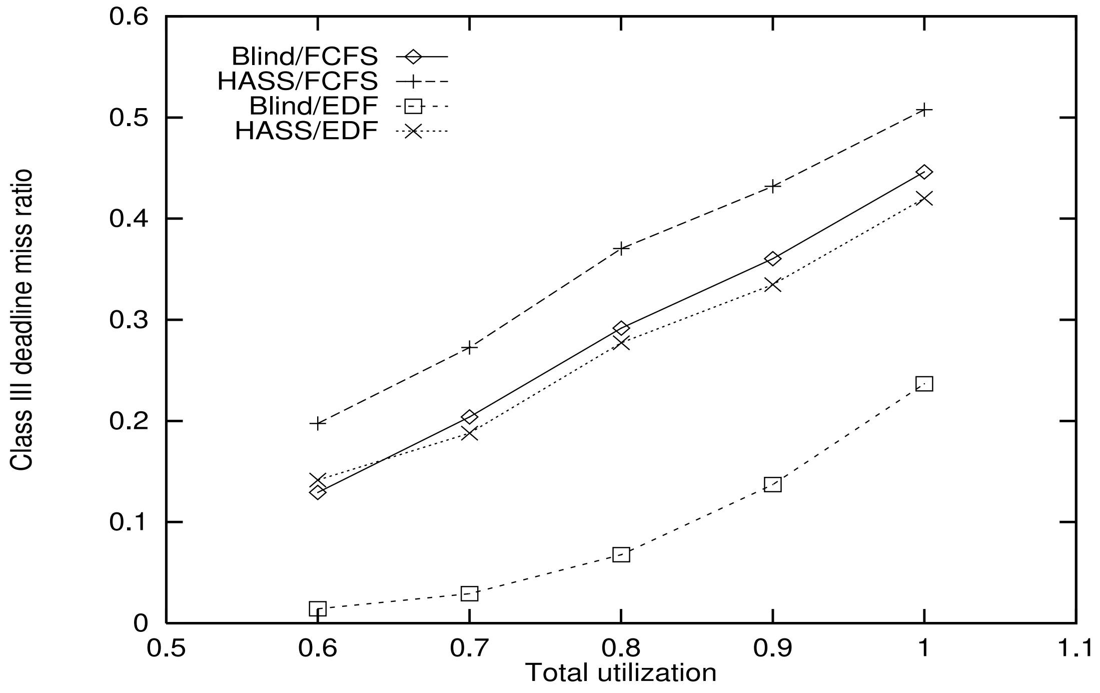


Figure 5.25: 50% Class I and II Utilization, CC Overhead = 0

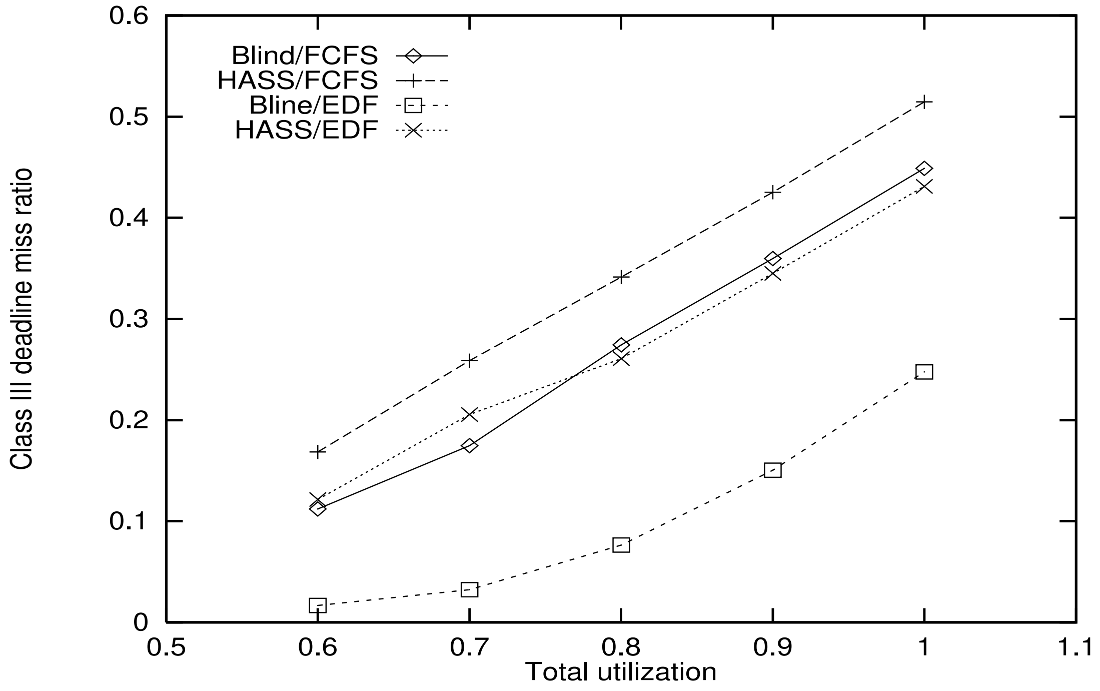


Figure 5.26: 50% Class I and II Utilization, CC Overhead = 100

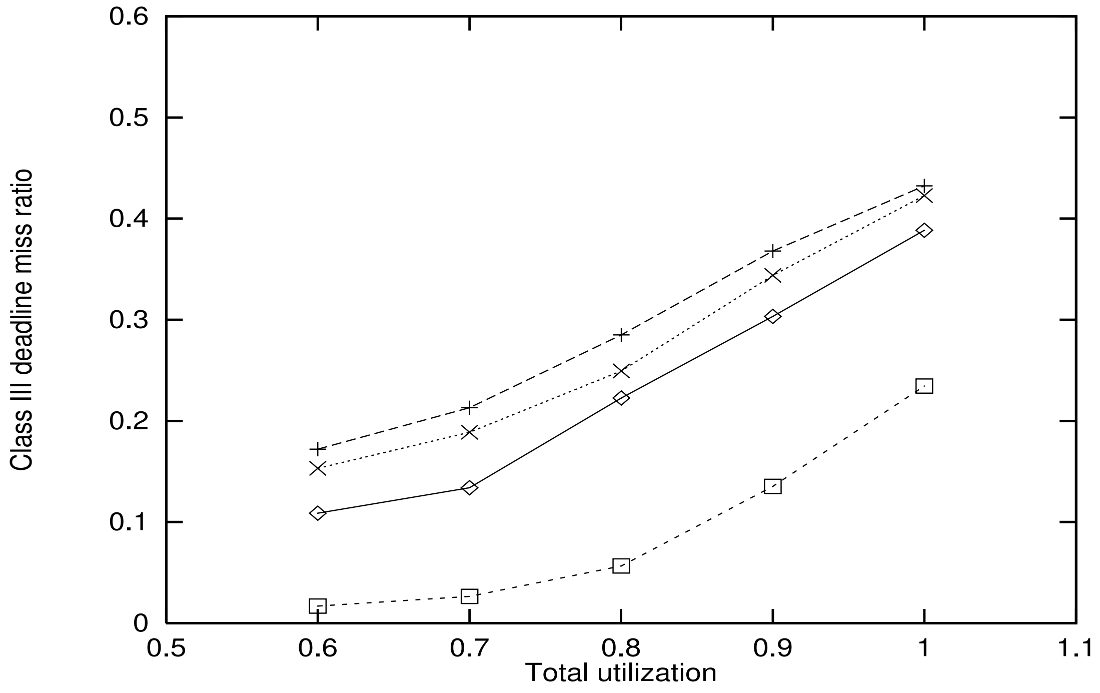


Figure 5.27: 50% Class I and II Utilization, CC Overhead = 500

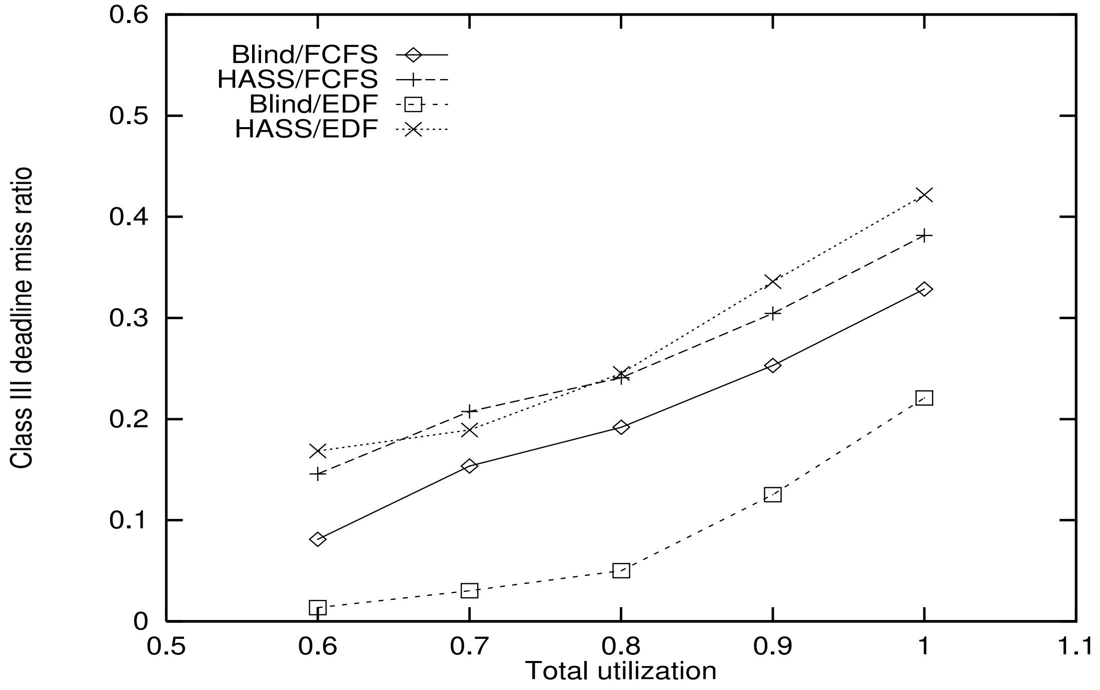


Figure 5.28: 50% Class I and II Utilization, CC Overhead = 1000

5.3 Summary

Our simulation results have illustrated the tradeoffs between the conventional best effort transaction scheduling algorithms and our integrated transaction processing schemes at various workloads and parameter changes. It is difficult to draw any definite conclusions, but at least we believe the following statements hold from our simulation study.

- Under our integrated transaction scheduling schemes (Guaranteed), the required performance level of each class of transactions can be achieved:
  - All the deadlines of Class I transactions are met, as guaranteed in the *off-line* schedulability test.
  - Statistical guarantee for Class II transactions is achieved. That is, the chances that Class II transactions meet their deadlines are not lower than the given guarantee levels. In fact, Class II transactions can meet much more deadlines than required, since they utilize the slack time in case of overrunning.
- *Cost of guarantee on timing constraints*

  Class III transactions miss more deadlines under Guaranteed than under the conventional best-effort scheduling (Blind), where some Class I and II transactions miss their deadlines and sometimes violate the temporal consistency requirements. This cost of guarantee varies, depending on how to find the spare-capacity for Class III transaction processing (SCF), how to assign priorities to Class III transactions, and how to resolve the conflicts among concurrently-running transactions (CIII).
  - Among the SCF policies we studied, Optimal slack stealing minimizes the cost. However, considering the implementation overhead, HASS may be a more realistic choice.
  - Among the CIII policies, LSF/SOCC seems to be the best choice. However, since LSF requires run-time estimates for Class III transactions which are not available all the time, EDF/SOCC may be a more reasonable option. In fact, EDF/SOCC shows relatively better performance under Guaranteed than under Blind.

    - Priority-based scheduling with concurrency control for Class III transactions seems beneficial even in memory-resident database environment, as long as the data conflict ratio and concurrency control overhead are reasonable.

- Under our static temporal consistency enforcement scheme (STCE), combined with our integrated transaction scheduling (Guaranteed), temporal consistency requirements of data and transactions are always fulfilled.

- *Cost of guarantee on temporal consistency requirements*
  STCE causes higher utilization level of Class I transactions than necessary, and consequently higher deadline miss ratio of Class III transactions. Furthermore, a modified transaction set according to STCE has more chances to fail the off-line schedulability test than the original transaction set, due to utilization level increase.

Chapter 6

Implementation Issues

Our RTDBS model and transaction processing schemes presented in previous chapters assumed a deterministic subsystem support. That is, they are viable only when the underlying subsystems provide deterministic services to transactions. In this chapter, we discuss how such a platform can be provided, making use of the current real-time microkernel technology and memory-resident databases. We also investigate how our experimental real-time database testbed, called StarBase [49], which is currently supporting only firm deadline transactions, can be extended to provide predictable transaction processing.

First, we propose a conceptual structure for achieving a deterministic system. Then, we briefly describe the current status of StarBase. Finally, we discuss the practical issues involved in implementing our RTDBS model and the integrated transaction scheduling scheme on top of the StarBase platform.

6.1 Deterministic Subsystem Structure

A database system must operate in the context of available operating system services. In other words, database operations need to be coherent with the operating system, because correct functioning and timing behavior of database management systems depends on the services of the underlying operating system. Because our real-time database model depends on the predictable execution time of database access operations, we need to have deterministic operating system service times. Unfortunately, conventional operating system services are not adequate for real-time transaction processing as mentioned in Section 2.5.

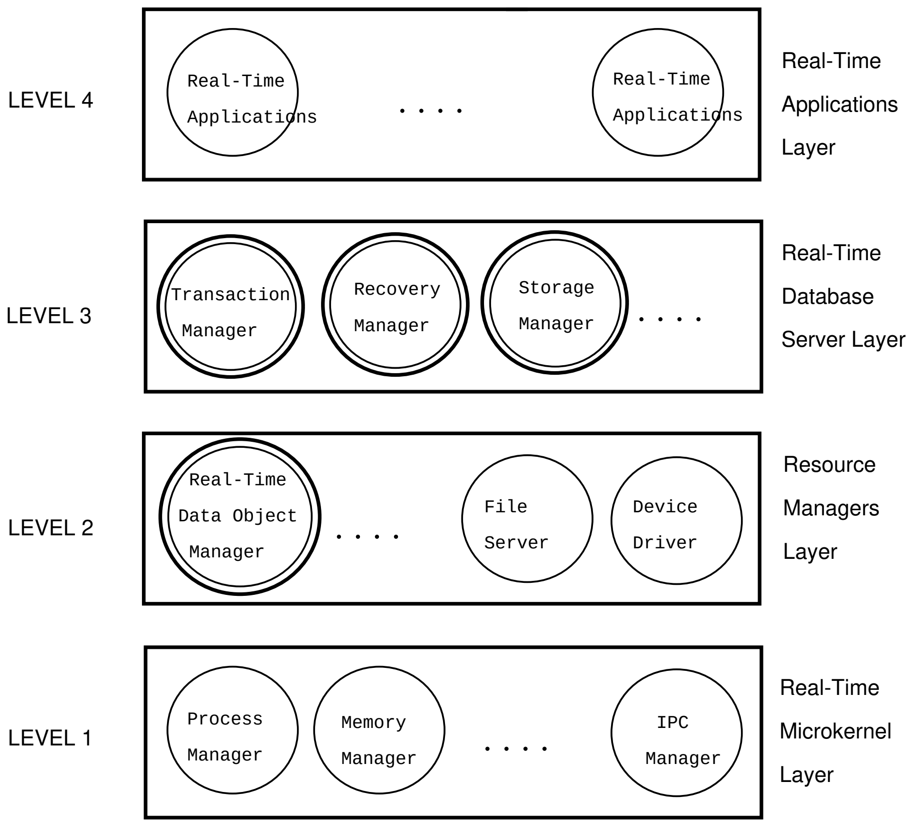


Figure 6.1: Conceptual Structure of a Real-Time Database System

Our approach is to build a deterministic service interface for real-time data objects on top of the real-time microkernel, and then develop predictable database management functions based on this interface. This is basically a layered approach to organizing real-time database systems (as shown in Figure 6.1).

The lowest level (Level 1) implements the microkernel itself, supporting a process manager, a memory manager, and an IPC manager. Level 2 implements specialized resource managers, such as device drivers and a real-time data object manager, which replaces an unpredictable conventional file system and buffer manager. At Level 3, we have a real-time database server, which consists of a transaction manager, a recovery manager, and a storage manager. The real-time data object manager provides the real-time database server with an interface which guarantees predictable data object access time. Having this

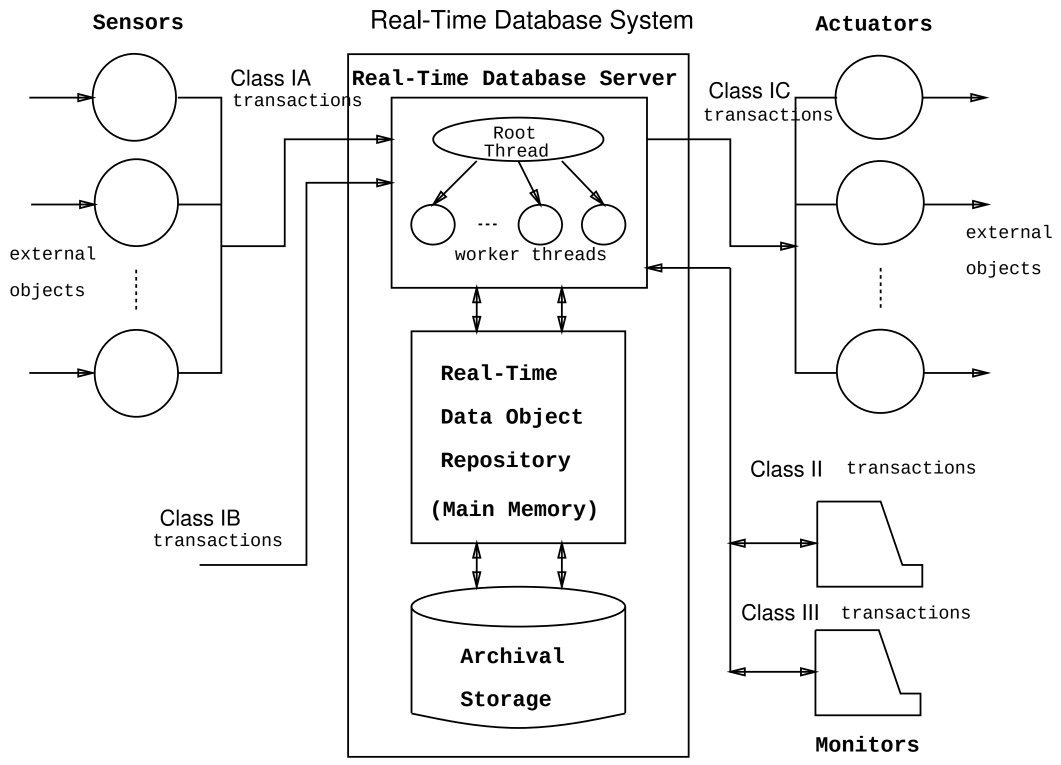


Figure 6.2: Physical Structure of a Real-Time Database System

deterministic kernel, we can support predictable real-time transaction capability using the proposed model.

The overall architecture of the target real-time database system are shown in Figure 6.2. The heart of the real-time database system is a real-time database server. It receives all types of transactions from sensors and monitors, processes them through multiple worker threads interacting with the real-time data object manager, and sends some results to actuators and monitors.

6.1.1 The Real-Time Database Server

Real-time transactions are processed by a multi-threaded real-time database server: each periodic transaction (Class I) has a dedicated worker thread, and sporadic or aperiodic transactions (Class II or Class III) are processed through one or more worker threads dynamically allocated by the root thread. Since each thread in this system is a schedulable entity, we can implement a real-time transaction scheduling algorithm under this architecture.

The Real-Time Database Server Layer (Level 3) consists of the following logical components (Figure 6.1):

- **Transaction Manager** implements the integrated real-time transaction scheduling and concurrency control schemes described in Chapter 4. It maintains all the necessary information of real-time transactions submitted in the system and makes appropriate scheduling decisions.

- **Recovery Manager** is responsible for abort and restart of Class III transactions. As noted earlier, the other types of transactions do not require this kind of conventional recovery scheme.

- **Storage Manager** supports an adaptive storage scheme for real-time data objects. It guarantees memory residence of time-critical hard real-time data objects (generally, continuous data objects) and archives less time-critical data objects (generally, discrete data objects) into archival storage without interfering with normal processing of hard real-time transactions.

6.1.2 Interface of the Real-Time Data Object Manager

Conventional file system abstraction and virtual memory management system are not appropriate as a hard real-time database objects repository since they are disk based and rely on unpredictable paging mechanisms. Even the most advanced recoverable virtual memory management systems such as Camelot [18] do not support hard real-time applications well, mainly due to their conventional rollback recovery mechanisms. We must develop a new abstraction for real-time data objects, giving a predictable behavior, and define a set of interface functions to access them.

We propose a new abstraction for real-time data objects which provides the following interface for the real-time database server:

- Create and delete a data object (**create_object**, **delete_object**)
- Open and close a data object with commit or abort flag (**open_object**, **close_object**)
- Map a data object into the user's address space (**map_object**)
- Read and write a data object (**read_object**, **write_object**)
- Guarantee memory residence of a time-critical data object (**pin_object**)
- Mark the non-time-critical part of the database, making it available for underlying operating system kernel's page replacement mechanism (**unpin_object**)
- Lock and unlock a data object (**lock_object**, **unlock_object**)

This real-time data object abstraction is implemented at Level 2 (Resource Managers Layer) in Figure 6.1 as a *Real-Time Database Object Manager*.

The **create_object** operation provides the semantic information of the real-time data object to the system (e.g., data type, temporal property, etc.) and this information is utilized by Storage Manager to determine the residency of the data object. Also, the **pin_object** and **unpin_object** operations are used by Storage Manager to communicate with the underlying memory manager. Furthermore, the other data object access operations must work adaptively for different types of data objects. For example, the **write_object** operation to a continuous data object does not make any log, but the operation to a discrete data object include some type of logging actions for transaction recovery.

To provide the transaction concepts, such as serializability and recoverability, for Class II and III transactions, and also implement the proposed optimistic concurrency control scheme, we adopt the *intentions list* approach, and thus **write_object** operation appends an entry to a transaction's intentions list and actual write to the data object occurs when the transaction is committed together with **close_object** operation.

6.2 StarBase – The Current Status

The StarBase RTDBS is an attempt to merge conventional DBMS functionality with real-time technology. StarBase supports the relational database model and understands a simple SQL-like query language. The RTDBS maintains a centralized server to which local or remote clients submit transactions. Transactions may execute concurrently and serializability is the correctness criterion. In addition to this conventional functionality, StarBase seeks to minimize the number of high-priority transactions which miss their deadlines. StarBase uses no *a priori* information about transaction workload and discards tardy transactions at their deadline points. In order to realize many of these real-time goals, StarBase is constructed on top of RT-Mach, a real-time operating system developed by Carnegie Mellon University [76]. StarBase differs from previous RTDBS work [1, 25, 28], in that a) it relies on a real-time operating system which provides priority-based scheduling and time-based synchronization, and b) it deals explicitly with data contention and deadline handling in addition to transaction scheduling, the traditional focus of simulation studies.

6.2.1 Database Overview

The StarBase system is organized as a multi-threaded server (Figure 6.3). It is assumed that database clients are physically disparate from the server, so they pass messages to communicate with the database server. Transaction requests are sent via RT-Mach's Inter-Process Communication (IPC) mechanism and are queued at the server's service port. RT-Mach provides a naming service with which StarBase registers its service port during initialization. Clients look up the service port by querying the name server with StarBase's well-known name.

When a request enters service, a *transaction manager* thread of execution is charged

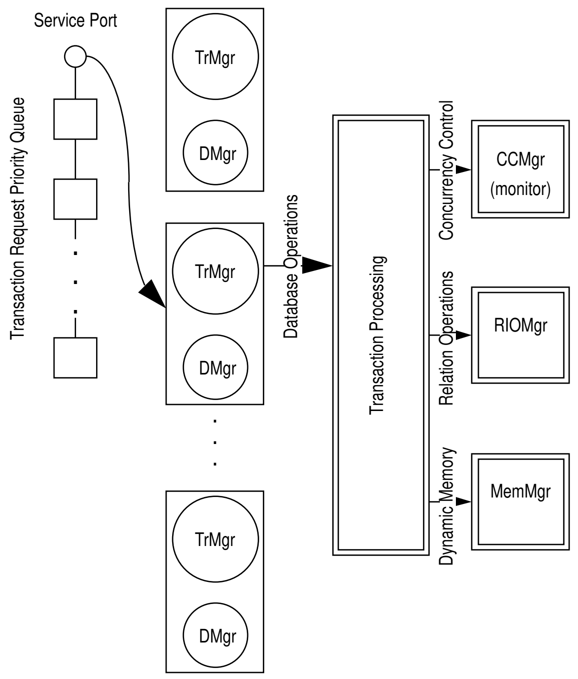


Figure 6.3: StarBase Server Architecture

with ensuring it is properly processed. The transaction manager executes the appropriate operations (e.g., read, write) as dictated by the content of the request. At the start of transaction processing, the transaction manager starts a *deadline manager* thread to enforce the transaction's deadline. A transaction needs certain resources to execute, including mechanisms to acquire memory, read and write data from relations, and ensure that data remains consistent. StarBase's three resource managers provide these services: the Small Memory Manager (MemMgr), the Relation I/O Manager (RIOMgr), and the Concurrency Controller (CCMgr). Each resource manager must ensure that transactions access their resources in a consistent and orderly fashion. To prevent mayhem, two of the resource managers are organized as *monitors* to synchronize the actions of different transactions. The services of the RIOMgr, however, are explicitly synchronized by the CCMgr.

StarBase uses *optimistic concurrency control* to ensure data consistency, allowing transactions to proceed unhindered until they are ready to apply their updates to the database.

6.2.2 Resource Contention and Transaction Scheduling

The StarBase system is highly reliant on its native operating system, RT-Mach, to provide the priority-cognizant services necessary for real-time resource scheduling. RT-Mach's services in turn are based on two major ideas (among others) which have been developed to ensure the allocation of resources to more important tasks in real-time systems. Those ideas are *priority-based CPU scheduling* [53] and the *Basic Priority Inheritance Protocol* (BPI) [60] for non-preemptible resources. With both ideas, tasks to be performed are ranked by their relative priorities (a function of their criticality and/or feasibility), and the highest priority tasks are granted access to the resource in question. RT-Mach provides several priority-based scheduling regimes, including Fixed Priority, Earliest Deadline First, Rate Monotonic, and Deadline Monotonic. RT-Mach's real-time thread model [76] distinguishes real-time threads of execution from ordinary ones, requiring the explicit specification of timing constraints and criticality on a per-thread basis. The timing and priority information is then used as input to the RT-Mach scheduler. RT-Mach also has striven to implement priority-based resource scheduling through its interprocess communication (RT-IPC) [34] and thread synchronization (RT-Sync) [75] facilities. RT-Mach implements BPI itself as a combination of priority queuing and priority inheritance. When a thread blocks on a mutex variable or when a message cannot be immediately received because all potential receivers are busy, RT-Mach queues the waiting thread or message in priority order and then boosts the priority of the thread inside the critical section or the priority of one of the potential receivers in accordance with the BPI protocol [60].

StarBase employs RT-Mach's priority-based CPU and BPI resource scheduling in several ways: to determine the transaction service order, to provide high-priority transactions the means to progress faster than low-priority transactions, and to provide priority-consistent access to facilities such as the Small Memory Manager and Concurrency Controller. For purposes of uniformity, StarBase adopts the same data type that RT-Mach uses to convey priorities, facilitating the straightforward translation of StarBase to RT-Mach priorities. Since the priority data type, `rt_priority_t`, includes a wide range of criticality and timing information, major changes in scheduling policy (e.g., Fixed Priority to Earliest Deadline First) are reduced to simple changes in the functions which compare priorities (e.g., changing the comparison of criticalities to one of deadlines) without any change in the client/server interface. StarBase itself must make priority-based decisions (e.g., concurrency control), so its priority-based comparisons involve priorities expressed as `rt_priority_t`-typed values. Of course, which policy is most appropriate differs from application to application, so the policy is to be used is left as a compile-time constant. Naturally, StarBase must use a consistent transaction scheduling policy across all of its priority-based decisions.

Transaction Service Order

Since performance ultimately degrades as the number of threads of execution in a system increase, and lazy allocation of resources adds unpredictability to the system, StarBase maintains only a fixed number of preallocated transaction manager threads. At the same time, since the StarBase RTDBS has no *a priori* knowledge of transaction workload, more transactions may be submitted to the RTDBS than it can handle at any given time. In order to throttle the flow in such circumstances, StarBase needs a mechanism to decide which requests to admit into service, and RT-Mach's RT-IPC facilities do just that in a convenient and priority-cognizant manner. To submit a transaction to the StarBase server, a client places the transaction instructions and priority information into a message and uses RT-IPC to send the message to the database server. Since RT-IPC queues incoming messages in priority order, the next available transaction manager receives the next highest priority unreceived message. Requests are therefore served in priority order and only the highest priority outstanding requests are in service at any given time. If a high priority transaction request cannot be serviced immediately because all transaction manager threads are busy serving some lower priority requests, RT-IPC's priority inheritance expedites one or more of the transaction managers so that the high priority request enters service at a time bounded by the minimum of the in-service transaction deadlines.

Transaction Progress

Once transactions enter service, StarBase needs to ensure that high priority transactions progress as quickly as possible. Since transactions require real-time execution, StarBase creates one real-time thread for each transaction manager and relies on RT-Mach's real-time CPU scheduling to schedule them. Transaction manager priorities are not specified explicitly by StarBase, however. Each obtains the correct priority assignment automatically upon receipt of a new transaction via RT-IPC's priority handoff mechanism [34].

Memory Manager

Transactions, depending on the nature of their operations, require some dynamic allocation of memory during their execution. StarBase maintains a Small Memory Manager to allocate and manage dynamic memory. Since transaction managers of different priorities may attempt to use it simultaneously, entry into the Small Memory Manager is guarded by a real-time mutex variable to avoid the priority inversion problem and to ensure the heap is accessed in mutual exclusion. To provide (relatively) predictable access to memory allocated through the manager, the heap is *wired* so that it cannot be paged out of physical memory.

Concurrency Controller

Although the StarBase concurrency controller is responsible for resolving contention at a higher level (i.e., data contention), it still relies on RT-Mach to provide basic synchronization and avoid the priority inversion problem. In particular, the concurrency controller must keep its own data structures consistent and ensure that transaction commits occur without interference. As such the concurrency controller is organized as a monitor, with a single real-time mutex variable for the monitor lock, and one real-time condition variable for each transaction manager.

To resolve data conflicts, StarBase uses a concurrency control implementation which draws heavily from the work of two research groups. First, Haritsa reasoned that optimistic concurrency control can outperform lock-based algorithms in a firm real-time setting [25]. He then developed a real-time optimistic concurrency control method, WAIT-X(S), which he found empirically superior, over a wide range of resource availability and system workload levels, to a previously proposed real-time lock-based concurrency control method called 2PL-HP [25]. Second, Lee *et al* devised an improvement to the conflict detection of optimistic concurrency control in general (Precise Serialization) [41], which StarBase integrates with Haritsa's WAIT-X(S). Detailed descriptions on their implementation for StarBase can be found in [49].

6.2.3 Summary

We outlines the architecture to support firm deadline transactions assuming no *a priori* knowledge of transaction workload characteristics. Unlike previous simulation studies, StarBase uses a real-time operating system to provide basic real-time functionality and deals with issues beyond transaction scheduling: resource contention, data contention, and enforcing deadlines. Issues of resource contention are dealt with by employing priority-based CPU and resource scheduling provided by the underlying real-time operating system. Issues of data contention are dealt with by use of a priority-cognizant concurrency control algorithm with a special conflict-detection scheme, called Precise Serialization, to reduce the number of aborts. Issues of deadline-handling are dealt with by constructing deadline handlers which synchronize with the start and end of a transaction and which do not interfere with its execution until the deadline expires.

The next step is to extend these solutions to the situation in which transaction characteristics are at least partially specified beforehand. With prior knowledge, a RTDBS can preallocate resources and arrange transaction schedules to minimize conflicts, resulting in more predictable service. Execution time estimates and off-line analysis can be used to increase system-wide predictability. Temporal consistency is also a matter to be explored. Once the basic, real-time, POSIX-compliant functionality needed to support our RTDBS model has been established, StarBase can be ported to other platforms. These issues will be discussed in the following section.

6.3 Predictable Transaction Processing in StarBase

As described in the previous section, StarBase supports only Class III transactions of our RTDBS model. This is partly because the current subsystem structure for StarBase lacks some functionality to implement deterministic services on the database server. In this section, we discuss how the current StarBase structure can be extended to provide guaranteed scheduling for Class I and Class II transactions, and investigate what functionalities must be supported by the underlying real-time operating system kernel to implement our integrated transaction processing scheme on StarBase.

Real-Time Server Support

Real-Time Mach provides two computational environments for real-time applications: Unix and Real-Time Server. Unix environment enables application programmers to compile, test, and debug using UNIX tools such as `cc`, `make` and `gdb` without rebooting machines. The environment is appropriate for debugging logical aspects of programs and makes the speed of developing programs very fast. This is the main reason why the current StarBase system has been implemented using the environment. However, the Unix server in RT-Mach (UX server) do not provide bounded response time. This is fine with the current StarBase system, since it deals with only firm deadline transactions in which bounded response time is not absolutely needed.

More critical real-time applications require deterministic computing environment. Real-Time Server (RTS) is a run-time environment for such applications on RT-Mach. Since RTS is implemented as a real-time server, the response time can be bounded. RTS provides a memory-resident file system. Files are allocated as a continuous memory block. Files can be mapped into a task's address space and shared among tasks. RTS also provides a simple task management facility which allows creation, suspension, and termination of tasks with low and bounded overhead. To support guaranteed transaction processing for Class I and Class II, StarBase should be implemented on such deterministic environment as RTS.

Class I and Class II Transaction Support

As described in the previous section, in the current StarBase system, only a fixed number of real-time threads (*worker threads*) are maintained for transaction managers for Class III transactions. Class III transactions arrive at the server aperiodically from separate client tasks and are queued at a priority queue. One of the available worker threads is allocated to run the transaction manager for a Class III transaction.

A Class I transaction, however, can be implemented as a dedicated periodic real-time thread on the server, since its behavior is predefined. In this way, we can avoid message passing delay between the server and client and thread switching overhead. For each Class II transaction, a dedicated aperiodic real-time thread can be preallocated. It does not require a delay queue, since only one instance of a Class II transaction can exist at the same time. The transaction manager for a Class II transaction must be able to keep track of the consumed run time by the transaction and mark it as "overrunning" when it spends the given execution-time budget without completion. The overrunning Class II transactions can be processed using slack time as explained in Section 4.2.

Class III Transaction Support

One additional real-time thread is required to support Class III transactions in our RTDBS model: a *Slack Stealer*. It maintains a slack table for every priority level either statically or dynamically, depending on the slack stealing algorithm used. For example, if the HASS algorithm is employed, a periodic real-time thread is invoked with a specified interval, generates a Class III slack stealing transaction, and places it at the front of the Class III transaction queue (i.e., it has the highest priority among Class III transactions). The slack stealer then runs at the next available slack time and updates the slack table.

The global transaction scheduler always looks up the slack table, and it schedules the highest-priority Class III transactions whenever slack time is available and no overrunning Class II transactions exist.

Run-Time Profiling

The integrated transaction processing strategy for our RTDBS model must

1. accurately measure the computation time consumed by each transaction, and
2. be able to control the execution of Class III transactions which attempt to overrun their predefined execution-time budget.

The first requirement demands precise performance monitoring software which typical operating systems do not provide. Operating systems usually accumulate usage statistics for each process by sampling during regular clock interrupts [42], but this information is not very precise over short intervals. Furthermore, the execution behavior of the monitored program must be independent of the sampling period. A more precise mechanism would measure durations between context switches and would account for interrupt processing time and other "system overhead".

Even if the system can accurately measure capacity consumption on a per-process basis, other problems arise. Usage statistics in traditional operating systems consist of system-level usage time and user-level time for each process [42]. For monolithic operating systems, this approach is sufficient, but for microkernel systems where operating system services are offered by different user-level servers, the usage statistics of an activity cannot found in the usage statistics of a single process. An activity may request services of several different operating system servers such as a name server, a file server, or an I/O server. To maintain an accurate picture of the activity's capacity consumption, the cost of these services must be charged to the activity itself rather than to the individual servers.

The second requirement, that the scheduler be able to control the execution of Class III transactions which attempt to overrun their reserved capacity, means that the system must accurately measure the processor usage of each Class III transaction and that a mechanism must exist to notify the scheduler when a program exceeds its computation time for the current invocation. Whenever the scheduler dispatches a thread for a Class III transaction, it computes the maximum duration of time the thread could execute based on its allocation and its usage so far, and it sets a timer to expire after this duration.

Chapter 7

Query Processing and Recovery

Our RTDBS research presented in preceding chapters made some assumptions on transactions and the operating environment. We regarded a transaction as a sequence of read and/or write operations on data objects, but have not considered the computational aspects of transactions, such as query processing. We also assumed that there would be no failure situation in our RTDBS and hence no recovery mechanism were provided. These issues, however, may need to be addressed in designing a *real* RTDBS application. The conventional approaches cannot be used directly in our RTDBS environment, since they are designed to fit in the conventional disk-based non-real-time DBMS environment.

Also with the memory-resident database environment, different performance metrics and cost formulas must be applied for our RTDBS model, requiring new directions in algorithms and data structures for query processing and recovery. In this chapter, we present our ideas for query processing and recovery mechanisms in the context of the memory-resident RTDBS.

7.1 Query Processing on Memory-Resident Databases

A number of different indexing methods are applicable for use in database systems. In [45] an indexing structure for use in memory-resident databases, called the *T Tree* was introduced which was reported to perform significantly better than the existing indexing methods for memory-resident data retrieval. Memory-resident indexes are designed to reduce the overall computation time while minimizing the amount of memory needed. Since relations are stored in memory it is not necessary for the indexes to contain the actual data, pointers to the data can be used instead. StarBase can support both disk and in-memory relations but is geared towards the memory-resident relations. The T Tree was selected as the indexing method because of its greater performance for in-memory data [19].

7.1.1 T Tree Index Structure

The T Tree is a new balanced tree structure that evolved from AVL and B Trees both of which have certain merits for use in main memory. The AVL tree which has the advantage of being fast since the binary search is intrinsic to the tree structure, has the disadvantage of its poor storage utilization. The B trees which have the advantage of better storage utilization however do not have the binary tree structure thus limiting the speed of access. The T Tree is a binary tree and each node in it contains many elements thus providing good storage utilization.

Associated with the T Tree is a minimum and a maximum count. Internal nodes keep their occupancy in this range. The minimum and the maximum counts differ slightly—one or two data items—which is sufficient to significantly reduce the tree rotations. As an example of an operation, to insert an element into a T Tree, the node that bounds the insert value is searched. If found the element is inserted there. If the insertion causes an overflow, the minimum element of that node is transferred to a leaf node, becoming the new greatest lower bound for the node it used to occupy. If no bounding node can be found, then the leaf node where the search ended is the node where the insert value goes. If the leaf node is full, a new leaf is added and the tree is rebalanced.

7.1.2 The Join Algorithms

Of the relational operations required to study, *join* is the most crucial one owing to the time complexity of the process. The parameters that were variable in the study undertaken by [45] were:

- The relation cardinality ($|R|$).

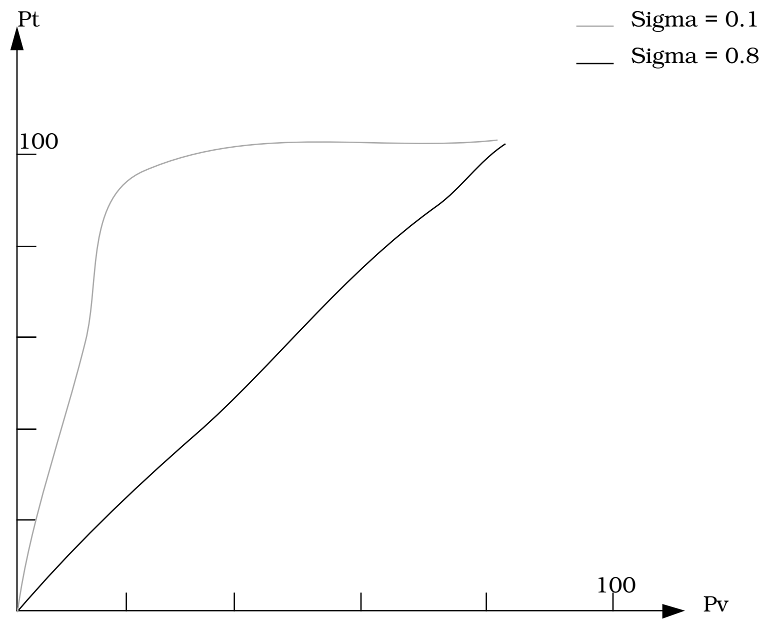


Figure 7.1: Distribution of Duplicate Values

- The number of join column duplicate values (as a percentage of $|R|$).
- The semijoin selectivity – that is the number of values in the larger relation that participate in the join expressed as a percentage of the larger relation.

To get a variable number of duplicates, a specified number of unique values were generated and then the occurrences of each of these was determined using a random sampling procedure with a variable standard deviation (as shown in the Figure 7.1). P_t denotes the percentage tuples and P_v denotes the percentage values. In order to get a variable semijoin activity, a smaller relation was built with a specified number of values from the larger relation.

The results that were obtained from the study of the various algorithms were based on the following cases ($|R1|$ denotes the outer and $|R2|$ the inner relation);

- *Vary Cardinality*: Vary the sizes of the relation with $|R1| = |R2|$, no duplicates and a semijoin activity of 100%.

- *Vary Inner Cardinality*: Vary the size of $R2$ ($|R2| = 1$ to 100% of $|R1|$) with $|R1| = 30,000$, no duplicates and the semijoin selectivity of 100%.

- *Vary Outer Cardinality*: Very the size of $|R1|$ ($|R1| = 1$ to 100% of $|R2|$) with $|R2| = 30,000$, no duplicates and the semijoin selectivity of 100%.

- *Vary Duplicate Percentage (skewed)*: Vary the duplicate percentage of both relations from 0 to 100% with $|R1| = |R2| = 20,000$, a semijoin selectivity of 100% and a skewed (standard deviation of 0.1) duplicate distribution.

- *Vary Duplicate Percentage (uniform)*: Vary the duplicate percentage of both relations from 0 to 100% with $|R1| = |R2| = 20,000$, a semijoin selectivity of 100% and a uniform (standard deviation of 0.8) duplicate distribution.

- *Vary Semijoin Selectivity*: Vary the semijoin selectivity from 0 to 100% with $|R1| = |R2| = 30,000$ and a duplicate percentage of 50% with a uniform duplicate distribution.

In the following, we mention the various algorithms that were taken into consideration. We describe them briefly before we discuss the most efficient[1] among them in detail. The algorithms that were considered are:

- Nested Loops
- Hash Join
- Tree Join
- Sort Merge
- Tree Merge

[1]By the most *efficient*, we mean that it provides not only the best average-case performance but also the minimum variance between the average- and the worst-case behavior. The algorithms used in real-time systems need to be deterministic, as well as being fast.

The *pure nested loops* join is an $O(N * N)$ algorithm. It uses one relation as the outer, scanning each of its tuples once. For each outer tuple, the entire inner relation is scanned looking for tuples with a matching join column value. As is evident this is the most naive and also a very inefficient process of performing the join operation.

The Hash Join and Tree Join algorithms are similar but they each use an index to limit the number of tuples that have to be scanned in the inner relation. The *Hash join* builds a Chain bucket hash index on the join column of the inner relation and then it uses this index to find matching tuples during the join. The *Tree join* algorithm uses the tree index on the inner relation to find the matching tuples. It may be noted that tree join is a viable solution only if the T Tree index is already existing, if not the cost of building one would certainly be greater than the hash join algorithm. It may be pointed out that in the study undertaken, the cost of building hash table index was always included as the likelihood of a hash table existing is relatively less.

The Merge Join algorithm was implemented using two index structures, an array index and a T Tree index and the two deviants so obtained are the Sort-Merge and the Tree-Merge algorithms. For the *sort-merge* algorithm, array indexes were built on both relations and then sorted. The sort was done using quicksort with an insertion sort for subarrays of ten elements or less. For the *tree merge* tests, T Tree indices were built on the join columns of each relation and then a merge join was performed using these indices. It may be noted that the graphs shown do not include the time taken for the construction of the T Trees as Tree Merge is a viable solution only if the indices already exist. A study undertaken in [44] reports that the arrays can be built and sorted in sixty percent of the time it would take to build the trees.

Based on the performance curves presented in [45] we conclude that if the T Tree has been built on the join columns in both the columns then Tree Merge is the best algorithm. If the tree indices is not built on the join columns of the two relations Hash join is the most efficient algorithm. We describe the two protocols in greater details.

Tree Merge

If T Tree indices are on the join columns of the two relations, then a merge join is performed using these indices. To compute a merge join, a pointer is associated with each relation. These pointers initially assigned to the smallest data item. As the algorithm proceeds, the pointers are moved successively to the next element in the sorted order. A group of tuples of one relation with the same value on the join attributes is read. Then the corresponding tuples of the other relation are read. Since the pointers progress in sorted order, tuples with the same value on the join attributes are in consecutive order. This allows us to read each tuple only once.

We present the algorithm in Figure 7.2. The functions that have been used in this algorithm are *smallest_element*s which return a pointer to the node that has the smallest data item corresponding to the column by which it has been indexed. This is a very straight forward process as we are basically looking for the leftmost leaf. The function *next_element*(cur_item) returns the pointer to the node which contains the data item with a value which is following the value of `cur_item`. The value following the `cur_item` may be the same as the value of the `cur_item`. Considering the present implementation of T Trees in StarBase this function of finding the next element in the sorted can very easily be incorporated. Finally, the function, *join_tuple* is the most basic join function that takes two tuples and performs a join operation.

Hash Join

If the T Tree indices are not on the join columns, the Tree Merge algorithm is not a viable solution and Hash Join is the most efficient solution. The Hash Join builds a Chain Bucket Hash index on the join column of the inner relation and then uses this index to find matching tuples during the join. Chained Bucket Hashing is a static structure used both in memory and on disk. It is very fast because it is a static structure – it never has to reorganize its data. This however requires that the size of the hash be appropriately estimated. Tests reported in [45] show that the *optimal table size* for a given number of relations has an

```
Algorithm Tree_Merge (*R1, *R2)
begin
  S1 = smallest_element(struct tnode *R1);
  S2 = smallest_element(struct tnode *R2);
  while S1–>val != NULL and S2–>val != NULL do
      /* find the nodes in the tree that have the same value on the join column */
      while S1–>value != S2–>value do
          if S1–>value < S2–>value then
              next_element(S1);
          else if S2–>val < S1–>val then
              next_element(S2);
          end
      end
      join_tuple(S1, S2);
      /* Join the remaining group of tuples with the same value of join */
      S1_restore = S1;
      S2_restore = S2;
      do begin
          S2 = S2_restore;
          while S1_restore–>val == next_element(S2)–>val do
              S2 = next_element(S2);
              join_tuple(S1, S2);
          end
          S1 = next_element(S1);
      until S1_restore–>val != S1–>val end
  end
end
```

Figure 7.2: Tree Merge Algorithm

```
Algorithm Hash_Join (*R1, *R2)
begin
  /* Initialize P1 to point to the first tuple of R1. */
  while P1 != NULL do
      /* proceed for all the tuples in R1 */
      Bucket = Hash_function(P1->val);
      while (P2 = scan_bucket(Bucket)) != NULL do
          if P2->val == P1->val then
              join_tuple(P1, P2);
          end
      end
      P1 = next_tuple(P1);
      reset_bucket(Bucket);
  end
end
```

Figure 7.3: Hash Join Algorithm

average of only two items per hash value.

We present the algorithm for the hash join protocol in Figure 7.3. Here *Hash_function* is the function that is used to determine the appropriate bucket. The function *scan_bucket* takes as an argument the bucket number and scans through the bucket to return pointer to next node present. Since the pointer progresses successively through the list of nodes in the bucket, after the entire bucket has been scanned the pointer is made to point to the first node using the *reset_bucket* function. The function *join_tuple* is very similar to the one described in the Tree Merge algorithm and takes two tuples and performs the join operation on them and returns the attributes that are desired.

7.1.3 Hybrid Join Approach

As has been pointed out in the discussion of the Tree Merge algorithm the Tree Merge algorithm performs better than the other algorithm if the tree indices are on the join column. If not the performance of the other algorithms is better than the Tree Merge. As is reported in [44], arrays can be built and sorted in 60% of the time to build trees, and also arrays can be scanned in about 60% of the time taken to scan a tree.

With the above argument in mind, we have explored the best algorithm given that the tree indices do not exist on the join column and we observe that Hash join performs the best in that scenario. However since StarBase has the T Tree as the index structure it may be a better idea if the T Tree structure could be benefited in some way instead of creating a new index structure that would entail additional expense. With this in mind we explore the possibility of an alternative structure.

In the Hash join approach we observe that in one of the relations the tuples are scanned only once while in the second relation (on whose data items hash is created), the tuples are accessed multiple times. Hence an improvement would be to hash that relation that has fewer number of tuples, so that the elements of the larger relation are accessed only once. Hence for the relation with larger number of tuples the tree is maintained and the tree is scanned. The hash function is applied to the data item of the join column and then the hash bucket is searched to look for the existence of tuples with the same data value on the join column. It may be noted that the tree may be scanned in any order, need not be in a sorted order.

We do not envision any mechanism of avoiding the creation of a hash bucket to take care of the additional increase in the space without trading off the performance of the join operation. However we suggest a mechanism where the process of join may be expedited in the case where the tree is indexed on one of the multiple join columns, *by taking advantage of the T Tree structure that would already be existing.* This is accomplished by the observation that the entire T Tree would be scanned while the chained hash bucket is being created and the T Tree structure renders some ordering to the data elements.

If the T Tree is indexed on one of the multiple join columns, we note that the tuples can very efficiently be scanned in the sorted order (of the indexed column). The following algorithm can be used to effectively create the hash buckets and perform the join operation.

Let us call the relation with large number of tuples (and hence the tree with greater number of nodes) L, and the relation with lesser number of tuples as S. Initially, assign pointers to the node that has the smallest data item of the join column(s) that are indexed. For instance, if the join be on columns A, B, C and D and the index be on A, B and G, then assign pointers to the smallest data item of the A and B column. As the algorithm proceeds the pointers are incremented to the next node. If the value of the data items in the two trees match, a hash function is applied on the data values of the columns that are not in the index, for instance in this example on C and D, and a hash bucket is created. If the two data value do not match the pointers are incremented and no hash is created.

Algorithmically, it may be expressed as in Figure 7.4. The algorithm is used to assign buckets to the appropriate elements of the smaller relation. The functions *smallest_element* and *next_element* are similar to the ones used in the previous algorithm except that here only those indexed keys that are present in the join column are considered while determining the order. The function *create_bucket* is used to determine the bucket to which the node/element under consideration should be added to. The algorithm that is used to determine the appropriate bucket while performing the join operation is similar to the above algorithm and is presented in Figure 7.5.

Here `rest_val` refers to the keys of the join column barring the ones that are present in the index. The function *next_bucket_element* is used to determine the next element present in the bucket. The remaining functions are similar to the ones in the preceding sections.

It may be observed that the performance gain obtained is as a result of two factors. Firstly, the hash function need to be applied to a fewer number of keys and the performance gain may be substantial if the function is complex. The second factor leading to a performance gain is the fact that while creating the hash buckets we are simultaneously eliminating out the tuples that do not have any chance to be present in the join operation. Hence saving the time for creating a bucket, the space associated and finally the reduced

```
Algorithm Hybrid_Join_1 (*L, *S)
begin

  S1 = smallest_element(struct node *L);
  S2 = smallest_element(struct node *S);

  while S1 != NULL do
      /* perform until all the elements of the bigger relations are accounted for */
      while S1–>val >= S2–>val do
          if S1–>val == S2–>val then
              create_bucket(S2–>val);
          end
          S2 = next_element(S2);
      end

      S1_restore_val = S2–>val;

      /* skip all the elements in the larger relations which have been accounted for
      or those which do not have a corresponding join element in the smaller relation */
      while S1_restore_val =< S1–>val do
          S1 = next_element(S1);
      end
  end

end
```

Figure 7.4: Hybrid Join Algorithm (1)

```
Algorithm Hybrid_Join_2 (*L, *S)
begin

  S1 = smallest_element(struct node *L);
  S2 = smallest_element(struct node *S);

  while S1 != NULL do
      while S1–>val >= S2–>val do
          if S1–>val == S2–>val then
              find_bucket(S2–>val);
          end
          while S1–>rest_val != S2–>rest_val or End_Of_Bucket do
              S2 = next_bucket_element;
          end
          if S1–>rest_val == S2–>rest_val then
              join_tuple(S1, S2);
          end
          S2 = next_element(S2);
      end

      S1_restore_val = S2–>val;

      /* skip all the elements in the larger relations which have been accounted for
      or those which do not have a corresponding join element in the smaller relation */
      while S1_restore_val =< S1_val do
          S1 = next_element(S1);
      end
  end

end
```

Figure 7.5: Hybrid Join Algorithm (2)

search time (fewer elements would mean lesser search time).

7.1.4 Summary

The motivation of this work was to find the algorithm that would be the most efficient one in the scenario where T Tree indexing is present. We have determined that Tree-Merge is the most optimal solution if the indices of the T Tree are on the join column. If not then Hash Join is the most optimal algorithm as the cost of building a T Tree on the join columns is very high. However, if space be a constraint then we propose a hybrid algorithm where the number of elements that have to be put into the hash and hence the number of elements that have to be searched is reduced and hence is beneficial in terms of both performance and storage space.

7.2 Recovery in Memory-Resident Real-Time Databases

Memory-resident data can mean large gains for database systems, since much of a transaction's lifetime is spent waiting to access data on disks. The increased performance is, however, not without its problems. In this section, we will concern ourselves with the problem of recovery in memory-resident databases. Specifically, we will be concerned with three subtasks of the recovery process – logging, checkpointing and restart after failure. We use the term *checkpointing* to refer to the maintenance of a copy of the memory-resident database on secondary storage. *Logging* is the maintenance of a sequential record of transaction activity that has occurred since the most recent checkpoint. In case of a failure, a new up-to-date database can be created by the *Restart* procedure, which reads in the secondary copy (from the most recent checkpoint) and brings that up-to-date using the information in the log.

Recovery in memory-resident databases is different from that in disk-based systems for the following reasons:

- In disk-based systems, I/O operations from disk to memory are the main criteria for determining which recovery technique is the best. In memory-resident database systems, processor costs are the most critical.

- The cost of recovery management, relative to the cost of executing a transaction will be much greater in a memory resident system than in a disk-based one.
- In disk-based systems, transaction processing may be halted while a checkpoint is taken. This is reasonable if the amount of main memory is small, since transactions will be halted only for a short time. Memory-resident databases will have much more primary memory and therefore, checkpointing and transaction processing must be concurrent.

Recovery mechanisms directly conflict with the real-time needs of a database. In disk-based databases, normal transaction processing is affected during the process of logging, when the log items of a transaction are written to stable storage or when a checkpoint is taken. As a result, valuable time is lost, which could otherwise have been used towards meeting the deadlines of currently executing transactions. Logging and checkpointing techniques that do not interfere with normal transaction processing are desirable. In conventional databases, when a failure occurs, a Restart procedure restores the database to some previous consistent state. This approach, however, is unsatisfactory for a real-time database, since data items may have temporal constraints. Rolling back might leave the database in a consistent state with respect to the correctness criteria, but the database may be inconsistent with respect to the temporal constraints. Some sort of forward recovery mechanisms may be necessary after a failure. Also, restart usually takes time of the order of half an hour to one hour, which means that all normal transaction processing has to be blocked for that period of time. This is totally unacceptable in a real- time system.

Memory-resident databases have been suggested as a good model for a real-time database, mainly because of their faster and deterministic database access time. Recovery in memory-resident databases is, however, the primary barrier to the use of memory-resident databases for real-time systems, mainly due to the reasons discussed in the previous paragraph (the issues raised are applicable even to memory-resident databases, since recovery mechanisms need substantial I/O with the disk).

As the basis of our solution to the problem of recovery in memory-resident real-time databases, we draw from two different approaches. The first, a memory-resident database design proposed in [46], uses a dedicated recovery processor to handle all recovery tasks that need disk access. The second is a slack stealing framework discussed in Section 4.2.2 and Appendix A. Our aim is to integrate these two approaches and provide a recovery mechanism that guarantees a certain level of deadline-cognizance.

In the following subsections, we will discuss the memory-resident database recovery approach proposed in [46], which will be integrated with the slack stealing approach, and present a composite recovery scheme. Further, we will briefly discuss other possible approaches to recovery.

7.2.1 A Memory-Resident DBMS Structure

In [43], a complete architecture for a memory-resident relational DBMS is presented, with a new index structure, new recovery methods and possibly new concurrency control approaches. As opposed to most previous work in memory-resident database recovery, where the database is regarded as a single entity, recovery is done at the relation or index level, providing a form of *demand recovery.*

The memory-resident recovery component as shown in Figure 7.6, consists of two independent processors, a main processor and a recovery processor; a portion of main memory is assumed to be stable, comprising two different log components, a Stable Log Buffer and a Stable Log Tail. The two processors have logically different functions. The main CPU is responsible for transaction processing, while the recovery CPU manages logging, checkpointing operations and archive storage. The two processors run independently and communicate through a buffer area in the Stable Log Buffer. Main memory is assumed to be organized as a sequence of partitions. All recovery actions – logging, checkpointing and restart – will be performed at the partition level.

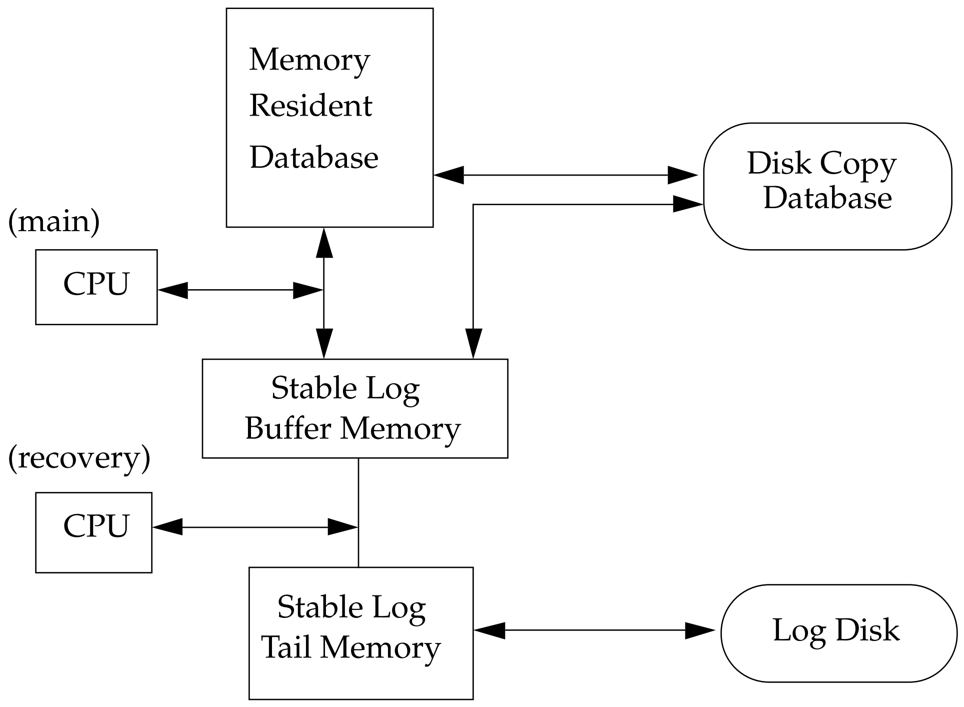


Figure 7.6: Recovery Mechanism Architecture for a Memory-Resident DBMS

Logging

The main CPU performs normal transaction processing. When a transaction completes, it writes its REDO log records to the Stable Log Buffer. This is the only contribution of the main CPU to the logging process. The recovery transaction, running on the recovery CPU, reads records in the Stable Log Buffer and places them in bins in the Stable Log Tail according to their partition address. Each partition having outstanding log information is represented in the Stable Log Tail by such a partition bin. The partition bin pages are written to disk when they become full.

Checkpointing

The main purpose of a checkpointing operation is to bound the log space for partitions by writing to disk those partitions that have a predefined number of log records (maintained in a variable update-count for each partition). Its secondary purpose is to reclaim the log space of partitions that have to be checkpointed because of age. When the recovery manager (running on the recovery CPU) decides that a partition has to be checkpointed, either due to update count or due to age, the following steps are taken:

- The recovery CPU issues a checkpoint request containing a partition address and a status flag in the Stable Log Buffer.
- The transaction manager, running on the main CPU, checks the checkpoint request queue in the Stable Log Buffer between transactions. For each partition checkpoint request that it finds, it starts a checkpoint transaction to read the specified partition from the database and write it to the checkpoint disk. It also sets the checkpoint status flag to *in-progress.*
- The checkpoint transaction sets a read lock on the partition's relation or waits until it is granted. This is necessary to ensure that the partition in a *transaction-consistent* state.
- The checkpoint transaction then allocates a block of memory large enough to hold the partition, copies the partition into that memory and releases the read lock. The checkpoint transaction then writes the partition to disk and commits. The status of the check- point operation is changed to *finished.*
- The recovery manager then flushes the partition's remaining log information from the Stable Log Buffer to the log disk. This step has to be taken, since this information is needed to recover from media failure.

The only problem with this approach is that the main processor is responsible for copying the partition to disk. This can be avoided if the recovery processor had access to the checkpoint disk and if the Stable Log Buffer were large enough to hold a number of partitions at a time. The main processor could copy the partition to be checkpointed to the Stable Log Buffer and the recovery processor could then copy the partition to disk. In this way, regular transaction processing being suspended, while the main processor performs a disk copy could be avoided.

Restart

Restoring the database after recovery from a failure involves reading in the previously checkpointed version of all partitions into main memory and applying the log information to the checkpointed version to bring it back to a consistent state. Restart usually takes time of the order of hours to complete. To amortize interference of the Restart process with normal transaction processing, a "recovery on demand" approach is proposed in [46].

During its initialization phase, each transaction declares the set of relations and indices that it will need. The transaction manager checks the relation catalog to see if they are memory resident. If they are not, it initiates a set of recovery transactions to recover them, one per partition. A recovery transaction for a partition reads the partition's checkpoint copy from the checkpoint disk and issues a request to the recovery CPU to read the partition's log records and place them in the Stable Log Buffer. Once the partition and the log records are available, the log records are applied to the partition to bring it up to its state preceding the crash. Then, between regular transactions, a system transaction issues recovery transactions at a lower priority for partitions that have not yet been recovered and that have not been requested by regular transactions.

7.2.2 An Integrated Recovery Scheme

As discussed earlier, a failure recovery scheme must ensure minimal interference of the recovery procedures with normal transaction processing. In addition, recovery operations must not be responsible for transaction deadline misses, or at least the deadline miss percentage must be minimized. In this subsection, we propose a method to integrate the recovery methods discussed in [46] with the slack stealing framework described in Section 4.2.2, to introduce a certain level of deadline cognizance into the recovery techniques.

As seen in the previous section, the logging process does not result in any perceptible overhead for the main CPU. All that the main CPU has to do is to copy the log records for each transaction to the Stable Log Buffer. The copy to disk is taken care of by the recovery processor. As a result, the time taken for logging can be neglected. Similarly, even during checkpointing, the main CPU is concerned only with the copy of the concerned partition to the Stable Log Buffer and so time taken for its execution can be ignored.

The Restart procedures are time-critical operations, since the database system cannot function until failure recovery is completed. Normal transaction processing cannot continue until the database has been brought back to a consistent state by the Restart operation. In the preceding section, a "recovery on demand" approach was explained. To integrate this technique into our real-time transaction processing framework presented in Section 4.2, each recovery action can be treated as a Class III aperiodic task. The deadline of this aperiodic task, D_r, is set by the following expression:

$$D_r \; = \; D_i - c_i - k,$$

where D_i is the deadline of transaction τ_i that "demanded" the recovery action, c_i is the worst-case execution time of the remaining actions in τ_i after the recovery action is demanded, and k is a constant safety factor. Also, if $D_r - t_{cur} < C_r$, where t_{cur} is the current time and C_r is the worst-case execution time of the recovery action, then this means that the time taken for recovery plus the time for completion of execution of the transaction is greater than the deadline of the transaction and hence the transaction can be discarded.

Each recovery task also has a priority associated with it. In most cases, the priority of the recovery action is the same as the priority of the transaction that demanded it. However, it could also be assigned a higher priority depending on how important the relation(s) that it is recovering happens to be, where importance is measured by the number of transactions (and their priorities) that would be needing the recovered relation(s) in the near future. A similar priority assignment approach can be adopted for recovery tasks that recover relations that have not yet been requested, which would normally be executed at the lowest priority.

There are a number of limitations to this approach:

- For regular actions in a transaction, estimation of worst-case execution time is not very difficult, because all data required is in random access main memory. However, for a recovery task, interaction with the disk is involved and estimation of execution times of actions involving dynamic I/O from/to disk would result in a very pessimistic estimate. As a result, the schedulability analysis is also too pessimistic and certain transactions may be rejected even though their corresponding recovery actions would actually meet their deadlines.

- Secondly, the above approach of slack stealing for recovery actions is based on the assumption that enough slack time is available for the recovery tasks to run. This assumption may not be valid immediately after failure. This is because, there are a number of transactions that have been blocked during the failure period and a majority of them would be close to their deadlines when the system comes up. Since the deadline-monotonic approach is used, these transactions would have a high priority leaving very little slack for recovery actions.

- The equation above is valid only if $t_{cur} + C_r + c_i$ is not greater than D_i (i.e., the time taken for recovery plus the time taken to complete the rest of τ_i must not exceed the deadline of τ_i. Since the worst-case execution time estimate is bound to be pessimistic, this problem should certainly be considered serious.

A number of heuristics can be used to decrease the deadline miss percentage of transactions immediately after recovery. First of all, immediately after the system is powered up after a failure, there exist a number of transactions whose remaining worst-case execution time is greater than their deadline. These transactions can be immediately discarded. Secondly, to further reduce interference of Restart with normal transaction processing, only the minimum number of relations as "demanded" by a transaction are recovered. This reduces demanded recovery time for each transaction to a minimum. Thirdly, if the workload level is very high, then one could consider discarding some lower priority transactions and using the time gained to perform critical recovery operations that recover relations which would be used later by higher priority transactions.

7.2.3 Other Recovery Techniques

There are a few limitations to the recovery scheme discussed in the previous sections. First of all, the scheme requires the presence of a dedicated recovery processor. Although this is not an unfair assumption in a real-time system where cost is a less important issue than the meeting of timing constraints, there might exist systems where this assumption is not valid.

Secondly, the system does not take into account the tradeoff between time spent on checkpointing and the time spent on Restart. The greater the degree of consistency of the checkpoint, the lesser the work done by Restart and vice versa. To minimize the time spent on checkpointing, the scheme discussed provides no guarantees on the consistency on the checkpoint. This means that Restart has to do a lot of work to bring the database back to a consistent state. Again, this is not an unfair policy, since typically, checkpoints are many and failures are few. However, if this were not true, then more consistent checkpoints are desirable. In this subsection, we will discuss some alternative approaches to the recovery actions in memory-resident databases.

Logging

In the event that a separate recovery processor is not available, the following logging technique can be used:

- When a transaction commits, it writes its log records to a stable portion of main memory.
- A separate recovery transaction for the transaction is generated, whose task is to copy the log records to disk. This transaction is assigned the lowest existing priority and can be modeled as a Class III aperiodic task in our RTDBS model.
- If slack time is not available for the recovery task, the priority of the logging task is gradually increased with increasing time.

Checkpointing

In this subsection, two approaches to checkpointing in a memory-resident database that guarantee a certain degree of consistency are presented – Black/White Policies [56] and an interference-free checkpoint mechanism surveyed in [65]. In the Black/White policy, the database is colored “read” or “not read yet” (by the checkpointer). A transaction can continue processing as long as it modifies only information that has already been read by the checkpointing procedure or has not been read yet. Transactions that have written “read” data must wait before writing “not read yet” data until after the checkpoint process has read it. Transactions that have modified “not read yet” data and desire to write “read” data are aborted.

A second approach to designing checkpointing mechanisms that do not interfere with transaction processing is presented by Son [65]. The principle here is to create a separate state of the system such that the checkpointing mechanism can view a consistent state that could result by running to completion all the transactions that are in progress when the checkpoint begins, instead of viewing a consistent state that actually occurs by suspending further transaction execution. If main memory were big enough to hold the snapshot, then the snapshot could be placed in main memory itself. As and when the snapshot reaches a consistent state, it is copied to disk. If a recovery processor were present, then this copy could be taken care of by the recovery processor, but if this were not the case, then the copy to disk would have to be done by the main processor, eating into time that could be used for normal transaction processing. This approach is ideal for systems where the average transaction size is small; i.e., the snapshot reaches a consistent state very quickly. However if transactions are long, it takes a long time to complete the checkpoint and if a failure occurs during the checkpoint, the transactions whose effects have not been reflected in the snapshot must be re-executed, wasting all the resources used for the initial execution of the transaction.

Summary

As is clear from the issues involved in checkpointing, there is a tradeoff between the time spent on checkpointing (the level of consistency that a checkpoint guarantees) and the time spent by Restart. If the checkpointing method guarantees consistency of the checkpoint, then Restart reads less log information that it would have to if a fuzzy checkpointing scheme [22] were used. In the presence of a secondary processor, the method of choice seems to be the non-interfering checkpoint method [65], since the overhead involved in generating a commit consistent checkpoint is taken care of by the recovery processor. A factor to be kept in mind is the frequency of failure in the system. If the failures are very frequent, then Restart would have to be optimized; i.e., commit consistent checkpoints are desirable. However, if failures are few and far between, then a more efficient checkpointing mechanism is desirable. In such a case the partition-level checkpointing scheme [46] would be most appropriate.

7.2.4 Temporal Constraints

Typically, a real time system consists of a controlling system and a controlled system. The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of temporal consistency. Usually the purpose of most periodic transactions, like Class IA and IB transactions in our model, is to keep individual data items up to date with respect to the state of the environment. Temporal constraints are the main reason why traditional backward recovery methods are considered inappropriate in a real-time database system. The very idea of rolling back to a previous consistent state seems to go against the goal of maintaining the state of the environment, since the only consistent state is the current state of the environment. Unfortunately, designing a forward recovery mechanism is not easy and very few results are available in the literature. The difficulty is mainly because in most cases, it requires a perfect understanding of the application semantics and in some cases it is just not possible without the execution of a new transaction called a compensating transaction to determine the current value of the temporally inconsistent data and update it.

Two things need to be done to handle the problem of temporally inconsistent data – on failure recovery, the temporally inconsistent data must be marked “invalid” and secondly, critical “invalid” data must be brought back to a temporally consistent state. The first part is easy – on recovery, if a data item does not satisfy a temporal constraint, just mark it as invalid. All transactions that try to access this data item are either blocked until the data item is restored to a consistent state or are aborted and restarted. The second part requires a slightly more sophisticated mechanism. Specifically, the concept of active databases could be put to good use. Triggers could be associated with critical data. Triggers could be used to fire off transactions to update the data items, once their temporal consistency is violated. The priority of such transactions could be fixed based on the importance of the data item (whether it could possibly be accessed by a hard deadline transaction) and the likelihood of the data item being read in the near future. The main drawback is that these update transactions are eating into an already scarce resource – time. The priority of these transactions must, therefore, be fixed very carefully, so that other important transactions in the workload do not miss their deadline.

There seems to be no general solution to the problem of preserving temporal constraints after recovery, but depending on the application various approaches are possible. Firstly, it might be the case that some data in the database is time-dependent; i.e., its value at the current instant can be extrapolated from its value at some previous instant. One could also visualize a situation in which some data returned by the database system is better than no data, i.e., instead of returning temporally inconsistent data, an approximation to the current value of the data could be computed, based on the condition of the other parameters in the environment. These solutions, however, vary depending on the application semantics.

7.2.5 Future Research Issues

In all of the previous sections, we have only discussed heuristics, i.e., methods that incorporate a certain level of "deadline cognizance" into the recovery techniques, but which do not provide any guarantees on the meeting of timing constraints of transactions in the workload. We have discussed how interference of the Restart procedure with normal transaction execution can be reduced through a "recovery on demand" approach, but we have not provided any rigorous estimates of the level of interference (the fraction of time taken away from normal transaction processing), a characterization of the number of transactions that would miss their deadlines after failure recovery, given the workload level, the deadlines of each transaction in the workload, number of relations that have to be recovered and the worst-case execution time of the recovery process.

We have seen how the presence of a separate recovery processor has provided substantial improvement in response time. This is especially true with logging and checkpointing, where the recovery process is independent of normal transaction processing, where the main processor can continue to process transactions with near negligible overhead for the recovery operation. For Restart, however, an additional processor did not provide that great a speedup, mainly because normal transaction processing has to be suspended while the recovery operation is processed. In this case also, some speedup can be obtained if the recovery of the relation is delegated to the recovery processor, the transaction that "demanded" the recovery operation is suspended and some other transaction that does not immediately require recovery of a relation is processed.

Finally, a general framework is required, where for each data item or groups of data items, temporal constraints, possible future values of the data item, depending on the state of the environment, limitations on such a calculation can be specified. In addition, a formal framework can be specified for defining the semantics of a transaction, steps that need to be taken for forward recovery, in the event of a failure, etc.

Chapter 8

Conclusions

Transactions with soft or firm deadlines can be processed successfully by using *time-cognizant* transaction scheduling algorithms that make no special assumptions on data and transaction semantics. This is because the performance goal of the scheduler is not to guarantee timing constraints of individual transactions, but rather to make a best effort to minimize the deadline miss ratio of transactions (or to maximize the total value of finished transactions when transactions have different values) under the given processing capability.

However, if there exist some hard deadline transactions in a real-time database application, the scheduling algorithm must guarantee that all the hard deadline transactions will complete by their deadlines and then make a best effort for the remaining soft or firm deadline transactions. This goal cannot be achieved without the support of a deterministic subsystem and *a priori* analysis of its data and transactions. Furthermore, RTDBS must satisfy the temporal consistency requirements of real-time data, which are sometimes more important than the logical consistency requirements of data in real-time databases.

We observe that no transaction scheduling algorithms proposed so far completely satisfy all these requirements even though several papers in the real-time database field have pointed them out [66, 4, 50, 5]. The goal of our research is to investigate a comprehensive model for real-time data and transactions which can be applicable to a broad range of real-time database applications, and to develop a predictable transaction processing scheme for the proposed model which can satisfy the individual performance constraints of each class of transactions.

8.1 Summary

Our research carried out in this thesis can be summarized as follows:

- We classified data objects and transactions found in typical real-time database applications, considering their different characteristics and requirements. Each type of real-time data objects has its own correctness criteria, different from the conventional one. Real-time transactions are categorized, according to their timing constraints, arrival patterns, data access patterns, availability of data and run-time requirements, and accessed data type. Our model is a superset of the conventional models; it includes both hard and soft real-time transactions, and supports the temporal consistency as well as the logical consistency of a database.

- Since each class of transactions has different performance requirements and data access constraints, it should be processed by a distinct scheduling mechanism and also its processing results must be predictable. Our integrated transaction processing scheme extends a fixed-priority-based *task* scheduling framework for mixed task sets into a *transaction* processing environment, combining with *best-effort* real-time transaction scheduling algorithms. It provides *predictability* for a RTDBS in the sense that under our transaction processing scheme it is guaranteed that every application in the system will achieve its own performance goals.

- We developed a scheme called STCE that can enforce temporal consistency of database, one of the most important requirements of RTDBS, statically. In STCE, temporal consistency requirements are transformed into timing constraints of transactions. Thus, as long as the timing constraints of transactions are satisfied in the system, temporal consistency of the system follows automatically.

- To synchronize the transactions' concurrent accesses to the data objects and maintain logical consistency of the database, we provided a concurrency control and conflict resolution scheme called SOCC for our RTDBS model. It is *semantic-based* in the context that it can utilize the available semantic information about different classes of transactions to make more efficient control decisions, consequently increasing the concurrency level of the system. In SOCC, however, serializable schedules are not always achieved for every class of transactions, since meeting their timing constraints is sometimes more important than maintaining the logical consistency of some types of data objects in a RTDBS.

- Our system allows the application developers to specify multiple guarantee levels for different applications. We performed a simulation study to identify the costs of these guarantees realized in our integrated transaction processing scheme. As expected, we found that the higher level of guarantee requires more system resources and therefore costs more non-guaranteed transactions.

- To support our RTDBS model, we designed a deterministic computing structure for a real-time database system, utilizing a memory-resident real-time data object abstraction and a real-time microkernel architecture. The proposed architecture eliminates sources of unpredictable behavior of the system related with dynamic I/O such as buffer management, dynamic paging, and disk scheduling. We developed a real-time database server called StarBase, which is currently supporting only firm real-time transactions, running on the top of the Real-Time Mach microkernel.

- We discussed issues related with query processing and recovery in memory-resident real-time database environment, and suggested some solutions to the problems. The proposed approaches extend the existing algorithms for the conventional memory-resident databases into our RTDBS model.

8.2 Future Work

Our research work has many possible future directions. First, our RTDBS model can be extended to accommodate more complicated applications in various computing environments:

- Some types of real-time transactions can hardly be included in our current RTDBS model. For example, there can be a hard real-time aperiodic transaction which updates some discrete data objects. We excluded such cases from our model regarding them as infeasible, but in a future more complex real-time system it may become feasible. Scheduling models and concurrency control mechanisms should be extended to adapt such cases.

- The transaction characteristics of each class, listed in Table 3.1, are just the minimum requirements for a transaction to be classified into the class. For example, for a Class III transaction its data and run-time requirements may be known in advance. Also, each Class III transaction can have a different value function. The current transaction scheduler, which assumes only firm deadline equal-valued Class III transactions, can be augmented to utilize such additional information and consequently maximize the total value of the system.

- As microprocessor technology matures and high performance microprocessors are available at low cost, it becomes feasible to construct a real-time system in a multiprocessor environment. Since our current model assumes a single processor environment, it must be changed to be applied to such settings. For example, if a data acquisition or slack stealing task is executed in a dedicated processor, the transaction model and the related supporting mechanisms also need to be reconsidered.

- Multimedia applications may need a real-time database support. Our memory-resident data object model is well-suited for process control applications, but may have some limitations to support a large amount of multimedia data. Furthermore, a multimedia database system may have much different kinds of timing and consistency constraints on its data and transactions.

Second, some parts of our approach may need further elaboration. For instance,

- Our STCE scheme may be too conservative. We may find less conservative conditions to guarantee temporal consistency of continuous data objects and develop approaches to relax these conditions according to the semantics of the given application.

- In this study, we used approximate methods based on dynamic slack stealing algorithms to find spare capacity for non-guaranteed transaction processing. However, in some system environments, a static approach may be more appropriate. For example, dynamic slack calculation costs may not be justified in some applications. It would be interesting to compare the performance of dynamic methods with that of static schemes in various situations.

Finally, it would be necessary to investigate how diverse real world real-time database applications can be specified under our model and how our integrated transaction processing scheme can be implemented to support the applications on top of a specific operating system kernel. First of all, the right real-time operating system kernel platform capable of supporting our RTDBS model should be identified. The current StarBase testbed can be used for this purpose if the underlying RT-Mach kernel becomes mature enough to fully satisfy the basic kernel requirements of our model. Once a kernel platform is decided and a real-time database server is implemented as proposed in Chapter 6, the schedulability analysis formula presented in Section 4.2 may need to be adjusted to reflect the specific system overheads involved in the kernel. It would be interesting to evaluate the cost and performance of our real-time transaction processing schemes using practical workloads on an actual system platform and compare them with our simulation results which are obtained using synthetic workloads.

Appendix A

Slack Stealing Algorithms for Fixed-Priority Preemptive Systems

Recent research for jointly scheduling transactions with both hard and soft time constraints has focused on the development of optimal slack stealing algorithms. In this appendix, we will review the dynamic slack stealing algorithms developed by Davis *et al* [14, 12].

In Section A.1, we outline the computational model and assumptions used. Section A.2 presents analysis of the maximum amount of slack which may be stolen at each priority level. This is used as the basis for the optimal dynamic algorithm described in Section A.3. In Section A.4, we discuss the implementation of two approximate slack stealing algorithms which are used in our simulation study.

A.1 Computational Model and Assumptions

In fixed-priority preemptive scheduling, each transaction is assigned a unique priority. Then at run time, the processor is allocated to the highest-priority runnable transaction. Each transaction is assumed to have a base priority i, in the range 1 to n where n is the number of transactions. 1 is the highest priority level and n the lowest. $hp(i)$ is used to denote the set of transactions with a higher base priority than i. Similarly, $lp(i)$ denotes the set of transactions with priority i or lower. The set of all periodic and sporadic transactions (i.e., in our model described in Chapter 3, Class I and Class II transactions) is denoted by T_G. Each sporadic transaction gives rise to an infinite sequence of invocation requests, separated by a period P_i. Thus periodic transactions may be viewed as a special case of sporadics. Each invocation of transaction τ_i performs an amount of computation between 0 and C_i (its bounded worst-case execution time) and has a deadline D_i measured relative to the time of the request.

In subsequent sections, the following assumptions apply:

- The transaction set T_G is assumed to be schedulable using fixed-priority preemptive dispatching with a priority ordering determined by some means such as deadline-monotonic priority assignment.
- Transactions cannot voluntarily suspend themselves.
- Transaction deadlines are assumed to be less than or equal to their periods.
- Transactions do not exhibit blocking or release jitter,[1] and context switch times are zero.

A.2 Schedulability Analysis

In this section, we determine the maximum amount of processing time which may be stolen from an invocation of a guaranteed transaction without causing its deadline to be missed. The analysis stems from considering the schedulability of each transaction τ_i in T_G at some arbitrary time t. It is assumed that at time t, the following data is available via the operating system, (typically derived from data stored in a process control block):

$l_i(t)$	The time at which τ_i was last released.
$x_i(t)$	The earliest possible next release of τ_i. Typically, $x_i(t) = l_i(t) + P_i$.
$d_i(t)$	The next deadline on an invocation of τ_i.
$c_i(t)$	The remaining execution time budget for the current invocation of τ_i.

[1]When a transaction is subject to a bounded delay between its arrival and release, it is said to exhibit release jitter [2].

Note that if the current invocation of τ_i is complete, then $d_i(t) = x_i(t) + D_i$, i.e., $d_i(t)$ is the deadline following the next release. $c_i(t)$ can be found by subtracting the execution time used from the worst-case execution time, C_i. Note that if τ_i is complete at time t and thus awaiting release, then $c_i(t) = 0$.

We now focus on finding the maximum amount of slack time, $S_i^{max}(t)$, which may be stolen at priority level i, during the interval $[t, t + d_i(t))$, while guaranteeing that τ_i meets its deadline. Note that $S_i^{max}(t)$ may not actually be available for non-guaranteed transaction processing due to the constraints on transactions in T_G with priorities lower than that of τ_i. To guarantee that τ_i will meet its deadline, we need to analyze the worst-case scenario from time t onwards. It is therefore assumed that all transactions τ_j are re-invoked at their earliest possible next release $x_j(t)$ and subsequently with a period of P_j.

In attempting to determine the maximum guaranteed slack, $S_i^{max}(t)$, it is instructive to view the interval $[t,\, t + d_i(t))$ as comprising a number of level i busy and idle periods. Any level i idle time between the completion of τ_i and its deadline could be swapped for τ_i computation without causing the deadline to be missed. Hence the maximum slack which may be stolen is equal to the total level i idle time in the interval. This result is used to calculate $S_i^{max}(t)$.

The method for finding the level i idle time relies on two equations: one determines $w_i(t)$, the length of a level i busy period with starts at time t, and the other determines the length of a level i idle period given its start time. By combining these two equations, $S_i^{max}(t)$ can be found by iterating over the interval $[t, t + d_i(t))$, totaling up all the idle time.

The first equation is derived using techniques given in [2]; two components determine the extent of busy period:

1. The level i or higher priority processing outstanding at time t.

2. The level i or higher priority processing released during the busy period.

The second component implies a recursive definition. As the processing released increases monotonically with the length of busy period, a recurrence relation can be used to find

$w_i(t)$:[2]

$$w_i^{m+1}(t) = S_i(t) + \sum_{\forall j \in hp(i) \cup i} \left(c_j(t) + \left\lceil \frac{w_i^m(t) - x_j(t)}{P_j} \right\rceil_0 C_j \right) \tag{A.1}$$

The term $S_i(t)$ represents level i non-guaranteed transaction processing released at time t and executing in slack time.

The recurrence relation begins with $w_i^0(t) = 0$ and ends when $w_i^{m+1}(t) = w_i^m(t)$ or $w_i^{m+1}(t) > d_i(t)$. Proof of convergence follows from the analysis of similar recurrence relations by Audsley *et al* [2]. The final value of $w_i(t)$ defines the length of the busy period. Alternatively, we may view $t + w_i(t)$ as defining the start of a level i idle period.

Given the start of a level i idle period, within the interval $[t, t + d_i(t))$, the end of the idle time, which may be converted to slack, occurs either at the next release of a transaction of priority i or higher or at the end of the interval. The second equation gives the length, $v_i(t, w_i(t))$, of the level i idle period.

$$v_i(t, w_i(t)) = \min \left(\begin{array}{l} (d_i(t) - w_i(t))_0, \\ \\ \min_{\forall j \in hp(i) \cup i} \left(\left\lceil \frac{w_i(t) - x_j(t)}{P_j} \right\rceil_0 P_j + x_j(t) - w_i(t) \right) \end{array} \right) \tag{A.2}$$

Combining equations (A.1) and (A.2), the method for determining the maximum slack, $S_i^{max}(t)$, proceeds as follows:

1. The slack which may be stolen, $S_i(t)$, is initially set to zero.
2. Equation (A.1) is used to compute the end of a busy period in the interval $[t, t + d_i(t))$.
3. The end of the busy period is used as the start of an idle period by Equation (A.2) which returns the length of contiguous idle time.
4. The slack processing, $S_i(t)$ is incremented by the amount of idle time found in step 3.
5. If the deadline on τ_i has been reached, the maximum slack which can be stolen is given by $S_i(t)$. Otherwise, steps 2 to 5 are repeated.

This method can be implemented as shown in Figure A.1.

[2]Note, $(x)_0$ is notational shorthand for $\max(x, 0)$, i.e., the minimum value of $(x)_0$ is zero.

```
/* Determine the maximum level $i$ slack at time $t$ */
/* Note: $\varepsilon$, set to the granularity of time, is a mathematical device */
/*         used to force the recurrence relation to continue. */

Algorithm find_slack ($i$, $t$, $S_i^{max}(t)$)
begin
  $S_i(t) = 0$;
  $w_i^{m+1}(t) = 0$;
  while $w_i^{m+1}(t) \leq d_i(t)$ do
      $w_i^m(t) = w_i^{m+1}(t)$;
      $w_i^{m+1}(t) = S_i(t) + \sum_{\forall j \in hp(i) \cup i} \left( c_j(t) + \left\lceil \frac{w_i^m(t) - x_j(t)}{P_j} \right\rceil_0 C_j \right)$;
      if $w_i^{m+1}(t) == w_i^m(t)$ then
          $S_i(t) = S_i(t) + v_i(t, w_i(t))$;
          $w_i^{m+1}(t) = w_i^{m+1}(t) + v_i(t, w_i(t)) + \varepsilon$;
      end
  end
  $S_i^{max}(t) = S_i(t)$;
end
```

Figure A.1: Algorithm for Determining the Level i Slack

We note that the algorithm in Figure A.1 may be used to calculate the maximum slack which may be stolen from a transaction τ_i which is either periodic or sporadic.

A.3 Optimal Dynamic Slack Stealing Algorithm

The previous analysis is used as the basis for a dynamic slack stealing algorithm. Non-guaranteed transactions need to be executed as soon as possible, while the deadlines of all guaranteed transactions are still met. In the case of strictly periodic transactions, this can be achieved by serving non-guaranteed transactions at highest priority, when there is slack available at *all* priority levels. However, when hard sporadic transactions are considered, there are problems with this approach.

Suppose, at time t there are non-guaranteed transactions pending and the highest-priority runnable transactions in T_G is τ_k. Further, suppose a sporadic transaction with priority higher than k has zero slack (say $D = C$) and could arrive at any time. This sporadic transaction may never arrive, preventing slack from ever being available at *all* priority levels.

To avoid the above problem, a different criteria is used for determining when non-guaranteed transactions may execute. The analysis guarantees that, provided level k non-guaranteed transaction processing is limited to $S_k^{max}(t)$ in the interval $[t,\, t + d_k(t))$ then transactions in T_G at priority levels k and higher will meet their deadlines. As the deadlines of *all* guaranteed transactions in T_G must be met, non-guaranteed transaction processing is only permissible at priority k while there is slack present at priority level k and *all* lower levels:

$$\min_{\forall i \in lp(k)} S_i(t) > 0 \tag{A.3}$$

Note, for completeness, when there are no hard transactions runnable, it is regarded that there is infinite slack available at priority level $n+1$. Provided Inequality (A.3) is true, non-guaranteed transactions can execute at priority level k in preference to the guaranteed transaction τ_k.

Using the above result, the dynamic slack stealing algorithm is formulated as follows:

Whenever there are non-guaranteed transactions pending, the algorithm in Figure A.1 is used to find the slack available at each priority level lower than or equal to k, where k is the priority level of the highest-priority runnable transaction in T_G. Inequality (A.3) is then used to determine if non-guaranteed transaction processing can proceed immediately in preference to τ_k. This dynamic slack stealing algorithm has been proved *optimal* in [14].

Note that the dynamic algorithm, described above, potentially requires the slack at each priority level to be re-computed at each time increment. To reduce the run-time overheads of this algorithm, it needs to examine how the slack at some later time t' $(t' > t)$ may be derived from the set of lower bounds on slack at time t. If the processor serviced non-guaranteed transactions or was idle between t and t' then slack is consumed at all priority levels:

$$\forall \tau_j \in T_G : S_i(t') = S_i(t) - (t' - t) \tag{A.4}$$

Whereas, if the processor was busy with the guaranteed transaction τ_j, then slack is consumed at all priority levels higher than j:

$$\forall \tau_i \in hp(j) : S_i(t') = S_i(t) - (t' - t) \tag{A.5}$$

The above equations represent a generic set of methods for maintaining the slack at each priority level. These methods accurately maintain the level i slack provided that τ_i does not complete during the interval $[t, t')$ and that all transactions τ_j of priority i or higher are released at their earliest next release and periodically thereafter. If these conditions do not hold, then the above equations still maintain valid lower bounds on the slack available at each priority level, although the degree of pessimism in these bounds is potentially increased. Hence, for strictly periodic hard transaction sets (i.e., a set of Class II transactions $T_{II} = \emptyset$), optimal slack scheduling can be achieved by re-calculating the exact level i slack each time transaction τ_i completes, while using equations (A.4) and (A.5) to maintain the slack counters at other times. In contrast, for transaction sets containing sporadics (i.e., $T_{II} \neq \emptyset$), optimal slack scheduling is only possible if the exact slack at all priority levels is recalculated every clock tick.

A.4 Approximate Slack Stealing Algorithms

In this section, approximations to the optimal dynamic slack stealing algorithm is discussed. The aim is to produce slack stealing algorithms which are efficient enough for run-time usage.

For hard deadline periodic transaction sets, the slack available at priority level i only increases when τ_i completes. Optimal slack stealing can be achieved by using the algorithm in Figure A.1 to calculate $S_i^{max}(t)$ at each completion of τ_i. Equations (A.4) and (A.5) are then used to keep track of the slack available at other times. The overhead of computing the slack can be reduced by *a priori* calculation of the least additional slack, S_i^{add}, which becomes available at *every* completion of τ_i. This enables a less pessimistic initial value for $S_i(t)$ to be used in algorithm of Figure A.1, thus reducing computation.

The least additional level i slack is generated when τ_i completes as late as possible (i.e., at its deadline) and the subsequent invocation is subject to the maximum interference from higher priority transactions. Maximum interference occurs when all higher priority transactions are released at the above deadline. This is effectively a critical instant for τ_i. Thus S_i^{add} is equivalent to the level i idle time in the interval $[0, P_i)$ where time 0 is a critical instant. The algorithm in Figure A.1 may be used to calculate the value of S_i^{add} offline.

Periodic Approximate Slack Stealing Algorithm

The Periodic Approximate Slack Stealing (PASS) algorithm combines both the static and dynamic methods of calculating slack. The static method is to increment the available level i slack by S_i^{add} at each completion of τ_i, while the dynamic approximation is to periodically re-evaluate the slack available at every priority level. Recall that to maintain the exact slack at all priority levels in a system containing sporadic transactions potentially requires that the slack at every priority level be re-evaluate every clock tick. Clearly this is infeasible in practice. However, the PASS algorithm approximates to this optimal approach. Varying the period of the PASS algorithm enables the overhead of slack calculation to be traded off against a decrease in the performance of non-guaranteed transactions. A very short period minimizes the deadline miss ratio of non-guaranteed transactions at the expense of a large overhead. While a very long period minimizes the overhead and increases the deadline miss ratio. This trade off has been further examined in [12].

Hyperperiod Approximate Slack Stealing Algorithm

The overheads of the PASS algorithm are clearly dependent on its period and the cardinality of the transaction set T_G. Consider the performance of the PASS algorithm when the calculation of slack is performed on the same processor which is executing the transactions. This requires the addition of a special high-priority transaction which performs the necessary slack calculations. Unfortunately, the presence of such a transaction may render the guaranteed transaction set infeasible. The PASS algorithm should be modified so that the calculation of slack has no effect on the processing capacity available for guaranteed transactions. The new algorithm is called the Hyperperiod Approximate Slack Stealing (HASS) algorithm.

Under the HASS algorithm, the dynamic calculation of slack is only ever performed in slack time. A special soft transaction SC is used to calculate the slack at all priority levels. Upon release, SC is placed at the head of the queue of non-guaranteed transactions awaiting execution. When there is slack available, SC executes re-calculating the slack at each priority level. Finally, when SC completes at time t, its next release is set for $t + P_{HASS}$, where P_{HASS} is the period of the HASS algorithm. This approach guarantees that the calculation of slack cannot interfere with the execution of guaranteed transactions while ensuring that it is performed as promptly as possible.

Bibliography

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, September 1993.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, May 1991.

[4] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Absolute and Relative Temporal Constraints in Hard Real-Time Databases. In *Proceedings of 1992 IEEE EuroMicro Workshop on Real Time Systems*, February 1992.

[5] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Data Consistency in Hard Real-Time Systems. Technical Report YCS203, Department of Computer Science, University of York, March 1992.

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

[7] A. Buchmann et al. Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control. In *Proceedings of the 5th International Conference on Data Engineering.* IEEE, February 1989.

[8] A. Burns. Scheduling Hard Real-Time Systems: A Review. *Software Engineering Journal*, 6(3):116–128, 1991.

[9] A. Burns and A. J. Wellings. Implementing Analysable Hard Real-time Sporadic Tasks in Ada 9X. Technical Report YCS209, Department of Computer Science, University of York, September 1993.

[10] A. Burns, A. J. Wellings, and A. D. Hutcheon. The Impact of an Ada Runtime System's Performance Charactersitics on Scheduling Models. In *Proceedings of the 12th Ada Europe Conference, LNCS 688*, pages 240–248. Springer-Verlag, 1993.

[11] P. Dasgupta and R. J. LeBlanc Jr. Clouds: A Support Architecture for Fault Tolerant, Distributed Systems. Technical report, School of Information and Computer Science, Georgia Institute of Technology, 1985.

[12] R. I. Davis. Approximate Slack Stealing Algorithms for Fixed Priority Preemptive Systems. Technical Report YCS217, Department of Computer Science, University of York, November 1993.

[13] R. I. Davis. Scheduling Slack Time in Fixed Priority Preemptive Systems. Technical Report YCS216, Department of Computer Science, University of York, November 1993.

[14] R. I. Davis, K. W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 222–231, Raleigh-Durham, NC, December 1993.

[15] D. J. DeWitt et al. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM SIGMOD Conference*, June 1984.

[16] H. Diel et al. Data Management Facilities of an Operating System Kernel. In *Proc. ACM SIGMOD Conference*, pages 58–69, Boston, June 1984.

[17] M. H. Eich. MARS: The Design of a Main Memory Database Machine. In *Proc. of the International Workshop on Database Machines*, October 1987.

[18] J. L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, February 1989. Also available as technical report CMU-CS-89-115.

[19] David W. George. Implementation of indexing and concurrency control mechanisms in a real time database. Master's thesis, Department of Computer Science, University of Virginia, February 1993.

[20] Michel Gien. Micro-kernel Architecture – Key to Modern Systems Design. *Unix Review*, November 1990.

[21] M. Guillemont. Microkernel Design Yields Real Time in a Distributed Environment. *Computer Technology Review*, pages 13–19, Winter 1990.

[22] R. B. Hagmann. A Crash Recovery Scheme for a Memory-Resident Database System. *IEEE Transactions on Computers*, C-35(9):839–843, September 1986.

[23] J. Haritsa, M. Carey, and M. Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 94–103, Orlando, FL, December 1990.

[24] J. Haritsa, M. Carey, and M. Livny. On Being Optimistic About Real-Time Constraints. In *Proceedings of the ACM Symposium on Principles of Database Systems*, April 1990.

[25] J. R. Haritsa. *Transaction Scheduling in Firm Real-Time Database Systems*. PhD thesis, University of Wisconsin–Madison, August 1991.

[26] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proceedings of the 12th Real-Time Systems Symposium*, pages 232–242, December 1991.

[27] R. Haskin et al. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, February 1988.

[28] J. Huang. *Real-Time Transaction Processing: Design, Implementation, and Performance Evaluation.* PhD thesis, University of Massachusetts at Amherst, May 1991.

[29] J. Huang, J. A. Stankovic, et al. Experimental Evaluation of Real-Time Transaction Processing. In *Proceedings of the 10th Real-Time Systems Symposium*, Santa Monica, CA, December 1989.

[30] J. Huang, J. A. Stankovic, et al. On Using Priority Inheritance in Real-Time Databases. Technical Report COINS TR 90-121, University of Massachusetts, November 1990.

[31] J. Huang, J. A. Stankovic, et al. Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes. Technical Report COINS TR 91-16, University of Massachusetts, January 1991.

[32] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and Analysis of Fixed Priority Schedulers. *IEEE Transactions on Software Engineering*, 19(9), September 1993.

[33] Young-Kuk Kim and Sang H. Son. An Approach Towards Predictable Real-Time Transaction Processing. In *Proceedings of the 5th Euromicro Workshop on Real-Time Systems*, pages 70–75, Oulu, Finland, June 1993.

[34] T. Kitayama, T. Nakajima, and H. Tokuda. RT-IPC: An IPC Extension for Real-Time Mach. Technical report, Carnegie-Mellon University, August 1993.

[35] V. Kumar and A. Burger. Performance Measurement of Some Main Memory Database Recovery Algorithms. In *IEEE Transactions on Knowledge and Data Engineering*, pages 436–443, 1991.

[36] Tei-Wei Kuo and Aloysius K. Mok. SSP: A Semantics-Based Protocol for Real-Time Data Access. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 76–86, Raleigh-Durham, NC, December 1993.

[37] Averill M. Law and W. David Kelton. *Simulation Modeling & Analysis.* McGraw-Hill, Inc., 1991.

[38] E. F. Lazowska et al. The Architecture of the Eden System. In *Proceedings of the 8th Symposium on Operating System Principles*, pages 148–159, Pacific Grove, Calif., December 1981. ACM.

[39] J. Lee and S. H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proceedings of the 14th Real-Time Systems Symposium*, pages 66–75, Raleigh-Durham, NC, December 1993.

[40] J. Lee and S. H. Son. Deadline-Sensitive Conflict Resolution for Real-Time Optimistic Concurrency Control. submitted for publication, 1994.

[41] J. Lee and S. H. Son. Precise Serialization for an Optimistic Concurrency Control Algorithm. submitted for publication, 1994.

[42] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, Reading, MA, 1989.

[43] T. J. Lehman. *Design and Performance Evaluation of a Main Memory Database System.* PhD thesis, University of Wisconsin–Madison, August 1986.

[44] T. J. Lehman and M. J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proc. 12th Conf. on Very Large Data Bases*, pages 294–303, Kyoto, Japan, August 1986.

[45] T. J. Lehman and M. J. Carey. Query Processing in Main Memory Database Management Systems. In *Proc. ACM SIGMOD Conference*, pages 239–250, Washington D.C., May 1986.

[46] T. J. Lehman and M. J. Carey. A Recovery Algorithm for a High-Performance Memory-Resident Database System. In *Proc. ACM SIGMOD Conference*, pages 104–117, San Francisco, CA, May 1987.

[47] J. P. Lehoczky. Real-Time Resource Managenment Techniques. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 1011–1020. John Wiley and Sons, New York, 1994.

[48] J. P. Lehoczky and S. Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. In *Proceedings of the 13th Real-Time Systems Symposium*, pages 110–123, Phoenix, AZ, December 1992.

[49] Matthew R. Lehr and Sang H. Son. Managing Contention and Timing Constraints in a Real-Time Database System. Technical Report CS-94-19, University of Virginia, May 1994.

[50] K.-J. Lin. Consistency Issues in Real-Time Database Systems. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989.

[51] K.-J. Lin, F. Jahanian, A. Jhingran, and C. D. Locke. A Model of Hard Real-Time Transaction Systems. Technical Report RC No. 17515, IBM T. J. Watson Research Center, January 1992.

[52] Y. Lin and S. H. Son. Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 94–103, Orlando, FL, December 1990.

[53] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[54] H. Nakazato. *Issues on Synchronization and Scheduling Tasks in Real-Time Database Systems.* PhD thesis, University of Illinois at Urbana-Champaign, January 1993. Also available as UIUCDCS-R-93-1786.

[55] P. E. O'Neil, K. Ramamritham, and C. Pu. Towards Predictable Transaction Executions in Real-Time Database Systems. Technical Report CS-TR-92-35, University of Massachusetts at Amherst, 1992.

[56] Calton Pu. On-the-Fly, Incremental, Consistent Reading of Entire Databases. *Algorithmica*, 1(4):271–287, December 1986.

[57] Krithi Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(1), 1992.

[58] K. Salem and H. Garcia-Molina. System M: A Transaction Processing Testbed for Memory Resident Data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[59] L. Sha, R. Rajkumar, and J. Lehoczky. Concurrency Control for Distributed Real-Time Databases. *ACM SIGMOD Record*, 17(1):82–98, March 1988.

[60] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[61] L. Sha, R. Rajkumar, S. H. Son, and C. Chang. A Real-Time Locking Protocol. *IEEE Transactions on Computers*, 40(7), July 1991.

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

[63] S. H. Son, J. Lee, and Y. Lin. Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control. *Journal of Real-Time Systems*, 4(3):269–276, September 1992.

[64] S. H. Son, S. Park, and Y. Lin. An Integrated Real-Time Locking Protocol. In *Proceedings of the 8th IEEE International Conference on Data Engineering*, pages 527–534, Phoenix, AZ, February 1992.

[65] Sang H. Son. An Adaptive Checkpointing Scheme for Distributed Databases with Mixed Types of Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), December 1989.

[66] X. Song and J. Liu. Performance of Multiversion Concurrency Control Algorithms in Maintaining Temporal Consistency. In *Proceedings of the IEEE 14th Annual International Computer Software and Applications Conference (COMPSAC)*, October 1990.

[67] B. Sprunt. *Aperiodic Task Scheduling for Real-Time Systems.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, August 1990.

[68] B. Sprunt, J. Lehoczky, and L. Sha. Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm. In *Proceedings of the 9th Real-Time Systems Symposium*, pages 251–258, December 1988.

[69] J. Stankovic. Real-Time Computing Systems: The Next Generation. Technical Report TR-88-06, University of Massachusetts, Amherst, January 1988. Also available as Misconceptioins about Real-Time Computing, *IEEE Computer*, October 1988.

[70] J. A. Stankovic and W. Zhao. On Real-Time Transactions. *ACM SIGMOD Record*, 17(1):4–18, March 1988.

[71] M. Stonebraker. Virtual Memory Transaction Management. *ACM Operating Systems Review*, 18(2):8–16, April 1984.

[72] M. Stonebraker, D. DuBourdieux, and W. Edwards. Problems in Supporting Database Transactions in an Operating System Transaction Manager. *ACM Operating Systems Review*, 19(1):6–14, January 1985.

[73] H. Tokuda. RT-Thread Model for Real-Time Mach. In *IEEE Workshop on Real-Time Operating Systems and Software*, May 1991.

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

[75] H. Tokuda and T. Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of the Second USENIX Mach Workshop*, October 1991.

[76] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proceedings of the USENIX 1990 Mach Workshop*, October 1990.

[77] I. L. Traiger. Virtual Memory Management for Data Base Systems. *ACM Operating Systems Review*, 16(4):26–48, October 1982.