Department of Computer Science
Series of Publications A
Report A-2003-1

■

Optimistic Concurrency Control Methods for Real-Time Database Systems

■

Jan Lindström

■

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-2003-1

Optimistic Concurrency Control Methods for Real-Time Database Systems

Jan Lindström

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XIII University Main Building, on January 25th, 2003, at 10 o'clock noon.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: jan.lindstrom@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/u/jplindst/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

ISSN 1238-8645
ISBN 952-10-0876-8 (paperback)
ISBN 952-10-0877-6 (PDF)
Computing Reviews (1998) Classification: C.3,D.4.7,H.2.4,J.7
Helsinki 2003
Helsinki University Printing House

Optimistic Concurrency Control Methods for Real-Time Database Systems

Jan Lindström

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
jan.lindstrom@cs.Helsinki.FI, http://www.cs.Helsinki.FI/jan.lindstrom/

PhD Thesis, Series of Publications A, Report A-2003-1
Helsinki, January 2003, pages
ISSN 1238-8645
ISBN 952-10-0876-8 (paperback)
ISBN 952-10-0877-6 (PDF)

Abstract

Many real-world applications contain time-constrained access to data as well as access to data that has temporal validity. For example, consider a telephone switching system, network management, navigation systems, stock trading, and command and control systems. These applications require gathering data from the environment, processing information in the context of information obtained in the past, and contributing *timely* response. Hence, these applications need a real-time database system, i.e. database system where transactions are associated with deadlines on their completion times.

Concurrency control is one of the main issues in the studies of real-time database systems. Many real-time concurrency control methods considered in the literature are based on pessimistic two-phase locking (2PL), where transaction acquires a lock before database operation and waits for the lock if it cannot be granted. However, 2PL has some inherent problems such as the possibility of deadlocks and unpredictable blocking time. These problems appear to be serious in real-time systems since real-time transactions need to meet their timing constraints, in addition to consistency requirements.

Optimistic concurrency control methods have the attractive properties of being non-blocking and deadlock-free. These properties make them especially attractive for real-time database systems. Because conflict resolution between the transactions is delayed until a transaction is near to its completion, there will be more information available on making the conflict resolution. Optimistic methods have the problem of unnecessary restarts and heavy restart overhead because some near-to-complete transactions have to be restarted. Therefore, the major concern in designing optimistic concurrency control methods is to design methods that minimize the number of transactions to be restarted.

This thesis shows that some of the well-known previous methods include unnecessary restart problems. A method to reduce these unnecessary restarts is proposed. This method is based on selecting a commit timestamp as near to the validation time as possible and a new method to resolve conflicts by adjusting the serialization order dynamically amongst the conflicting transactions after the validation is successful. This method maintains serializability or, more precisely, strict serializability. We show that many unnecessary restarts can be avoided efficiently and avoiding unnecessary restarts is an efficient approach for improving the performance and predictability of concurrency control methods for main-memory database systems beyond the current state-of-the-art.

Additionally, methods to incorporate information about the timing constraints of transactions in the conflict resolution is proposed. We show that priority cognizance is not a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems. The results show that the proposed methods offer better chances for critical transactions to complete before their deadlines.

Finally, the work identifies a need for adaptive and integrated concurrency control methods in real-time database systems. Therefore, a new optimistic concurrency control method is presented where conflict resolution is based on adaptation to the current workload. This method is shown to produce correct results and was experimentally tested. The performance of the proposed method is shown to be superior to previous approaches.

The feasibility of the proposed methods have been experimentally tested using a prototype of a main-memory real-time database system for telecommunications with a telecommunication service workload. The results show that optimistic methods can be used in this kind of environment.

Computing Reviews (1998) Categories and Subject Descriptors:

C.3	Real-time systems
D.4.7	Real-time systems
H.2.4	Concurrency - Transaction Processing
J.7	Real time

General Terms:
Real-Time Databases, Transaction Processing, Concurrency Control

Additional Key Words and Phrases:
Real-Time Systems, Real-Time Scheduling

Acknowledges

I am grateful to my supervisor Professor Kimmo Raatikainen for his guidance and support during the preparation of this work. I want to thank the current and former heads of the department for providing me with excellent working conditions.

I have been privileged to work with many excellent scientist, of whom I would like to mention especially Professor Timo Alanko, Tiina Niklander, Vilho Raatikka, Heikki Lokki and Professor Pertti Saurola. Working with them was inspiring and developed me both as a person and a scientists. I want to thank them for the experiences of working with them even when it gave me a good excuse to delay this work and do something more interesting.

I would like to thank Marina Kurtén for correcting the grammar of this thesis. All remaining errors are solely mine. Also, I want to thank all the unmentioned workmates at the Department of Computer Science and Finnish Museum of Natural History. I am thankful to the Helsinki Graduate School of Computer Science and Engineering for courses and meetings that provided valuable contacts as well as interesting insights into related research areas. Nokia Foundation have to be acknowledged for providing me with the necessary financial support in the form of Nokia Scholarships 2001–2002.

I am extremely indebted to my dear friends Matti Luukkainen, Wilhelmiina Hämäläinen, Hanna Kervinen, Jyrki Havia, Jaakko Kurhila, Anni Rytkönen, and Markku Vähäaho. Countless, delightful, and inspiring discussions with them were irreplaceable. I would like to thank my friends for many delightful moments I have experienced. Friends are like angels who taught me to fly, when I was swamped by life and its troubles. Life was sometimes hard with worries and there was some tension, but I always knew that I can go to them for support. Moreover, they showed me that the most important in the life is to care for your friends even when all else around you crashes. Finally, I especially want to thank all of them for their warm and encouraging words that made me continue with this thesis when I was ready to abandon it, and rekindle enthusiasm at the moment when my motivation was at stake. Without their crucial support this thesis would not have been completed. You are all important to me, and I care for you and love you as my friends.

I would like to thank my parents Pekka and Terttu for their guidance and support and my brother Marco for many pleasant moments.

My very special thanks go to Saara Teinilä, who came into my life at the moment when it was disintegrating and everything seemed futile. Her encouraging words and tender kisses and hugs gave me back my faith in the future.

jääkylmästä yössä harhaillut
kuuman kosteassa keskipäivässä hikoillut
lumisessa sohjossa tarponut
meren sinisessä sylissä uinut
karussa kalliossa kiipeillyt
pöllön kynnet päässä ihmetellyt
vihreässä metsässä eksynyt
pimeässä tuvassa pelännyt
jäätävässä sateessa palellut
kaipuun tunteessa matkustellut
repivässä yksinäisyydessä hajonnut
märässä suossa uponnut
kerran rakastuneena
kerran oikein tehneenä
elämä edessä

Contents

Chapter 1

Introduction

Real-time computing is one of the most difficult and challenging areas in computing. It is also of great importance, since real-time software is essential to all ultra-reliable and safety critical applications. Real-time systems play a critical role in modern life, ranging from domestic appliances to industrial robots, from industrial process control to advanced avionics and from computer games to telecommunication systems. These applications involve real-time tasks, which often carry significant penalties in terms of cost and loss of life in the event of failure. Every day these systems provide us with important services. When we drive, they control the engine and brakes of our car and control traffic lights. When we fly, they schedule and monitor the takeoff and landing of our plane, make it fly, maintain its flight path.

In recent years, a lot of research work has been devoted to the design of database systems for real-time applications. Databases are useful in real-time applications because they combine several features that facilitate (1) the description of data, (2) the maintenance of correctness and integrity of the data, (3) efficient access to the data, and (4) the correct executions of query and transactions in spite of concurrency and failures [81].

A real-time database system (RTDBS) is usually defined as a database system where transactions are associated with deadlines on their completion times. In addition, some of the data items in a real-time database are associated with temporal constraints on their validity. In order to commit a real-time transaction, the transaction has to be completed before its deadline, and all of its accessed data items must be valid up to its commit time. Otherwise, the benefits of the results will be seriously decreased. In many cases, any deadline violation of a critical real-time transaction may cause disaster.

Traditional databases, hereafter referred to as databases, deal with persistent data. Transactions access this data while preserving its consistency. The goal of transaction and query processing methods chosen in databases is to get a good throughput or response time. In contrast, *real-time database systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. The important difference is that the goal of real-time database systems is to meet the time constraints of the transactions.

One of the most important points to remember here is that real-time does not only mean fast [97]. Furthermore, real-time does not mean timing constraints that are in nanoseconds or microseconds. Real-time means the need to manage *explicit* time constraints in a predictable manner, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks and transactions [98].

As a sample application let us consider a database system for telecommunication applications called *Telecommunication Database System* in more detail [6]. Recent developments in networking and switching technologies have increased the data intensity of telecommunications systems and services. This can be seen in many areas of telecommunications including network management, service management, and service provisioning. For example, in the area of network management the complexity of modern networks leads to a large amount of data on network topology, configuration, equipment settings, and so on. In the area of service management there are customer subscriptions, the registration of customers, and service usage (e.g. call detail records) that lead to large databases.

The integration of network control, management, and administration also leads to a situation where database technology becomes an integral part of the core network. The combination of vast amounts of data, real-time constraints, and the necessity of high availability creates challenges for many aspects of database technology including distributed databases, database transaction processing, storage and query optimization. The performance, reliability, and availability requirements of data access operations are demanding. Thousands of retrievals must be executed in a second and the allowed down time is only a few seconds per year.

A telecommunication database system must offer real-time access to data [41, 42]. This is due to the fact that most read requests are for service programs that have exact time limits. If the database cannot give a response within a specific time limit, it is better not to waste resources and hence abort the request. As a result of this, the request management policy should favor predictable response times instead of high throughput. The best alternative is that the database can guarantee that all requests are replied to within a specific time interval. In telecommunications a typical time limit for a read request is around 50ms. Most of read requests must be served in that time [42]. For updates, the time limits are not as strict. It is better to finish an update even at a later time than to abort the request. In this work strict consistency and atomicity is required in updates.

Telecommunication database system services consist of two very different kind of semantics: service control and service management. *Service control* denote services for customers [39]. Service control transactions have quite strict deadlines and their arrival rate can be high (about 7000 transactions/second/call-area), but most service control transactions have read-only semantics. In transaction scheduling, service control transactions can be expressed as firm deadline transactions. *Service management* denote possible management services for customer and network administration [39]. Service management transactions have opposite characteristics. They are long updates which write many objects. A strict consistency and atomicity is required for service management transactions. However, they do not have explicit deadline requirements. Thus, service management transactions can be expressed as soft real-time transactions (i.e. transactions which have some value even after deadline) or non-realtime transactions (i.e. transactions without deadlines). In this thesis we will concentrate to service control transactions.

The requirements of the telecommunications database architectures originate in the following areas [48]: real-time access to data, fault tolerance, distribution, object orientation, efficiency, flexibility, multiple interfaces, security and compatibility [6, 82, 100]. In summary, Telecommunication Databases are real-time systems that contain rich data and transaction semantics, which can be used to design better methods for concurrency control, recovery, and scheduling. They have data with varying consistency criteria, recovery criteria, access patterns, and durability needs. This can potentially lead to development of various consistency and correctness criteria that will improve the performance and predictability of such systems.

Concurrency control is one of the main issues in the studies of real-time database systems. With a strict consistency requirement defined by serializability [7], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [18]. Two-phase locking has been studied extensively in traditional database systems and is being widely used in commercial databases. In recent years, various real-time concurrency control methods have been proposed for the single-site RTDBS by modifying 2PL (e.g. [4, 5, 34, 37, 58, 78, 95]). However, 2PL has some inherent problems such as the possibility of deadlocks as well as long and unpredictable blocking times. These problems appear to be serious in real-time transaction processing since real-time transactions need to meet their timing constraints, in addition to consistency requirements [83].

Optimistic concurrency control methods [26, 50] are especially attractive for real-time database systems because they are non-blocking and deadlock-free. Therefore, in recent years, numerous optimistic concurrency control methods have been proposed for real-time databases (e.g. [15, 17, 33, 53, 54, 63]). Although optimistic approaches have been shown to be better than locking methods for real-time database systems [28, 29], they have the problem of unnecessary restarts and heavy restart overhead. Unnecessary restart occurs when a transaction fails it's validation phase and is restarted even when history is serializable. Transaction is restarted only if it has enough time remaining for meeting its deadline. Otherwise, all transactions changes are rolled back. Heavy restart overhead is due to the late conflict detection that increases the restart overhead since some near-to-complete transactions have to be restarted because of failed validation. However, because the conflict resolution between the transactions is delayed until a transaction is near its completion, there will be more information available in making the conflict resolution.

This thesis examines optimistic concurrency control methods in the real-time database systems. An optimistic concurrency control method for real-time database systems should be predictable and respect timing constraints as well as maintain database consistency. Therefore, optimistic concurrency control methods should not restart unnecessary transactions, because transaction restart causes waste of resources and unpredictability.

Priority cognizant concurrency control methods based on the optimistic methods have not been widely studied. Because priority cognizance is important to offer better support for transaction timing constraints as well as predictability, the major concern in designing real-time optimistic concurrency control methods is not only to incorporate information about the timing constraints of transactions for conflict resolution but also to design methods that minimize the number of transactions to be restarted. Therefore, in this thesis we focus on two research questions: how to avoid many unnecessary restarts and how to integrate priority cognizance in the optimistic concurrency control method. Our theses are following:

- Many unnecessary restarts should and can be avoided efficiently.

- Our hypothesis is that priority cognizance is not a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems beyond the current state-of-the-art. If they would be, some of the previous attempts at designing priority cognizant methods for disk-based real-time database systems would have shown better results.

- Our hypothesis is that integrated and adaptive conflict resolution is a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems beyond the current state-of-the-art.

This thesis is based on work done in the following original publications [71, 72, 70, 76, 75, 73, 74]. In the following the main contributions of this work are listed:

- We have developed a prototype real-time database system for telecommunications [73]. Author's contribution to this system is focused on design and implementation of the transaction processing and concurrency control. All the experiments have been done using this prototype system. The prototype presented in Section 7.2 is based on this publication.

- We have developed a benchmark for a distributed real-time database system in telecommunications [72]. The author's contribution focuses to design and implementation of the database schema and transactions. All the experiments have been done using modified version of this workload. The author has done all experiments presented in this thesis. The workload presented in Section 7.3 is based on this publication.

- The work identifies the problem of unnecessary restarts in some of the previous proposals of the optimistic concurrency control methods for real-time database systems [74]. Therefore, a new optimistic concurrency control method is presented. This method is shown to produce correct results and avoids the problem of unnecessary restarts found in many previous methods. The proposed method is experimentally tested and shown that its performance is superior to previous methods. The author is main contributor in this publication. Chapter 5 is based on this publication with extended experiment tests and a new workload compared to original publication.

- The work extends the previous work on optimistic concurrency control methods using transaction priorities in conflict resolution [70]. The extended method is shown to produce correct results and experimentally tested. The performance of the proposed method is shown to be superior to previous approaches. Section 6.1 is based on this publication with extended experiment tests and a new workload compared to original publication.

- The work identifies a need for use deadline-driven priority in the conflict resolution and presents two different methods to use deadline-driven priority in the conflict resolution [76, 75]. The proposed methods are experimentally tested. The author is main contributor in both publications. Section 5.2 is based on [75] and Section 5.3 on [76]. This work presents extended experiment tests with a new workload compared to original publications.

- The work identifies a need for adaptive and integrated concurrency control methods in real-time database systems [71]. Therefore, a new optimistic concurrency control method is presented where conflict resolution is based on adaptation to the current workload. This method is shown to produce correct results and experimentally tested. The performance of the proposed method is shown to be superior to previous approaches. Section 6 is based on this publication.

This thesis is organized as follows. Chapter 2 presents basic concepts in real-time systems and databases. Chapter 3 presents concurrency control in real-time databases. Chapter 4 presents a new optimistic concurrency control method to reduce unnecessary restarts. Chapter 5 presents a new optimistic concurrency control methods which uses transaction attributes in conflict resolution. Chapter 6 presents a new optimistic concurrency control method which uses adaptive and integrated conflict resolution method. Chapter 7 presents research using real-time databases on telecommunication applications and experiment results. Chapter 8 draws the conclusions of this thesis.

Chapter 2

Real-Time Systems and Databases

In 1981, a software error was the reason why a stationary robot moved to the edge of its operational area. A nearby worker was crushed to death [11]. This is only one example of the dangers of real-time systems.

In this chapter we will define the concept of real-time system, give some examples of real-time systems, and list some characteristics of real-time systems. Additionally, we will define the concept of real-time database system, give some examples of real-time database systems and some recent research, and list some characteristics of real-time database systems.

2.1 Real-Time Systems

The *Oxford Dictionary of Computing* gives the following definition for real-time system:

Any system in which the time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical word, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.

Here the timeliness is taken in the context of the total system. In [11] the following definition of real-time system is used:

A Real-time system is any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified delay.

A real-time system consists of a *controlling system* and a *controlled system* [83]. The controlled system is the environment with which the computer and its software interacts. The controlling system interacts with its environment based on the data read from various sensors, e.g., distance and speed sensors. It is crucial that the state of the environment, as observed by the controlling system, is consistent with the actual state of the environment to a high degree of accuracy. Otherwise, the actions of the controlling systems may be catastrophic. Hence, timely monitoring of the environment as well as timely processing of the information from the environment is necessary. In many cases the read data is processed to derive new data [19]. Therefore, the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced.

The *release time* of a task is the instant of time at which the task becomes available for execution. The task can be scheduled and executed at any time at or after its release time. The *deadline* of a task is the instant of time by which its execution is required to be completed. The *response time* of a task is the length of time from the release time of the job to the instant when it completes. A *relative deadline* is the maximum allowable response time of the task. The deadline of a task, sometimes called its *absolute deadline*, is equal to its release time plus its relative deadline. A *timing constraint* is a constraint enforced on the timing behavior of a task.

It is common to divide timing constraints into three types: **hard**, **firm** and **soft** (see Figure 2.1).

- *Hard* deadline tasks are those which may result in a catastrophe if the deadline is missed. One can say that a large negative *value* is imparted to the system if a hard deadline is missed. These are typically safety-critical activities, such as those that respond to life or environment-threatening emergency situations (e.g. [77, 55]).

- *Soft* deadline tasks have some value even after their deadlines. Typically, the value drops to zero at a certain point past the deadline (e.g. [35, 46]).

- *Firm* deadline tasks impart no value to the system once their deadlines expire, i.e., the value drops to zero at the deadline (e.g. [14, 30]).

Several scheduling paradigms emerge, depending on a) whether a system performs a schedulability analysis, b) if it does, whether it is done statically or dynamically, and c) whether the result of the analysis itself produces a schedule or plan according to which tasks are dispatched at run-time. Based on this the following classes of algorithms can be identified [84]:

- Static table-driven approaches: These perform a static schedulability analysis and the resulting schedule is used at run time to decide when a task must begin execution.

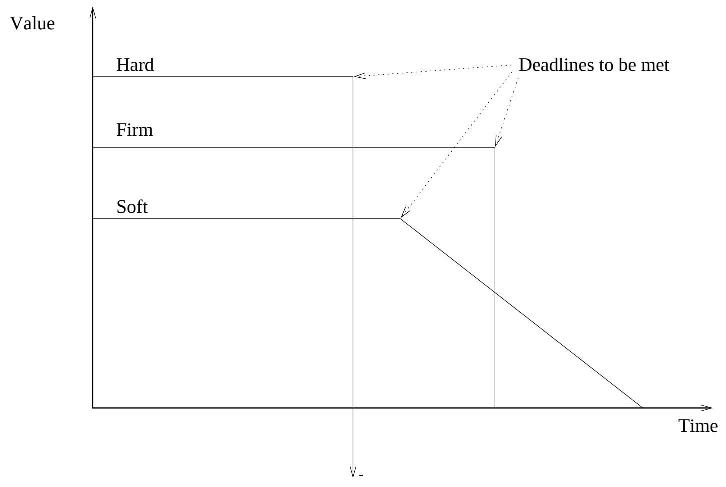


Figure 2.1: The deadline types.

- Static priority-driven preemptive approaches: These perform a static schedulability analysis but unlike the previous approach, no explicit schedule is constructed. At run time, tasks are executed using a highest priority first policy.
- Dynamic planning-based approaches: The feasibility is checked at run time, i.e., a dynamically arriving task is accepted for execution only if it is found feasible.
- Dynamic best effort approaches: The system tries to do its best to meet deadlines.

In the following we refer to a few well known scheduling algorithms.

In the *earliest deadline first* (EDF) [77] policy, the task with the earliest deadline has the highest priority. Other tasks will receive their priorities in descending deadline order. In the *least slack first* (LSF) [1] policy, the task with the shortest slack time is executed first. The slack time is an estimate of how long the execution of a task can be delayed and still meet its deadline. In the *highest value* (HV) [27] policy, task priorities are assigned according to the task value attribute. A survey of scheduling policies can be found in [1].

In a real-time system environment resource control may interfere with CPU scheduling [3]. When blocking is used to resolve a resource allocation, a *priority inversion* [2] event can occur if a higher priority task gets blocked by a lower priority task.

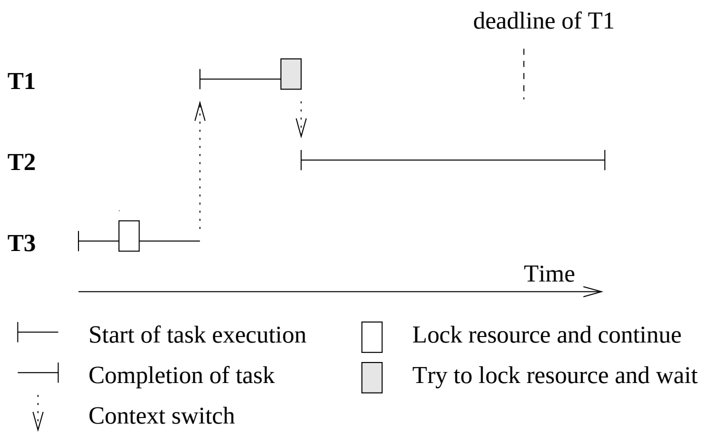


Figure 2.2: Priority inversion example.

Figure 2.2 illustrates an execution sequence, where a priority inversion occurs. Let us assume that priority of a task T_3 is 10, T_2 is 20, and T_1 is 30. The task T_3 executes and reserves a resource. The higher priority task T_1 pre-empts the task T_3 and tries to allocate a resource reserved by the task T_3. Then, the task T_2 becomes eligible and blocks T_3. Because T_3 cannot be executed the resource remains reserved suppressing T_1. Thus, T_1 misses its deadline due to the resource conflict.

In [88], a *priority inheritance* approach was proposed to address this problem. The basic idea of priority inheritance protocols is that when a task blocks one or more higher priority task the lower priority task inherits the highest priority among conflicting tasks.

Figure 2.3 illustrates how a priority inversion problem presented in figure 2.2 can be solved with the priority inheritance protocol. Again, task T_3 executes and reserves a resource, and a higher priority task T_1 tries to allocate the same resource. In the priority inheritance protocol task T_3 inherits the priority of T_1 and executes. Thus, task T_2 cannot pre-empt task T_3. When T_3 releases the resource, the priority of T_3 returns to the original level. Now T_1 can acquire the resource and complete before its deadline.

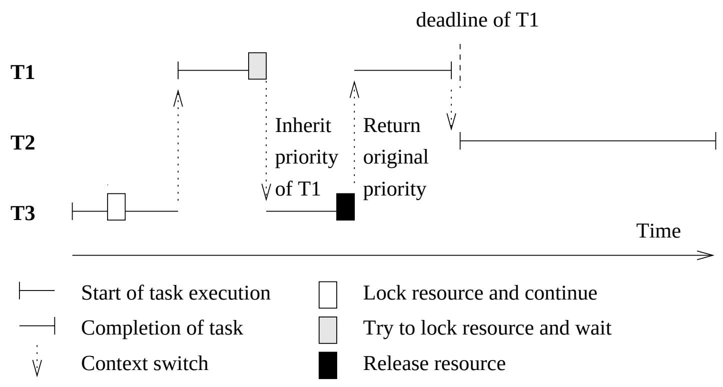


Figure 2.3: Priority inheritance example.

2.2 Real-Time Databases

A *database system* holds a set of named *data items*. Each data item has a *value* [7]. The values of the data items at any one time form the *state* of the database. In practice, a data item could be a word of the main memory, a page of a disk, etc. The size of the data contained in a data item is called a *granularity* of the data item. In this work we assume that data items are atomic i.e. a whole data item is accessed as one unit. Therefore, the granularity of the data item in this work is an object which has an identity (object identity, OID). A discussion of subobjects is out of the scope of this work.

A *database system* (DBS) is a collection of hardware and software components that support commands to access the database, called *database operations*. The most important operations we will consider are read (denoted as r) and write (denoted as w). $r[x]$ returns the value stored in data item x without changing the state of the x. In this thesis x is an atomic instance of the object. $w[x]$ changes the value of x overwriting the old value. In this thesis the whole object is overwritten even in case when a small portion of the x is changed.

We will use the abbreviation DBS, instead of the more conventional DBMS, to point that a DBS in our sense may be much less than an integrated database management system. For example, it may only be a simple main-memory system with transaction management capabilities. The DBS executes each operation *atomically*. This means that the DBS behaves as if executing operations *sequentially* i.e. one at a time.

The DBS also supports *transaction operations*: begin, commit, and abort (denoted as b,c, and a). A transaction program reports to the DBS that it is about to begin executing a new transaction by issuing the operation begin. It marks the termination of the transaction by issuing either the operation commit or the operation abort. By issuing a commit, the transaction program reports to the DBS that the transaction has terminated normally and all of its effects should be made permanent. By issuing an abort the transaction program marks the DBS that the transaction has terminated abnormally and all of its effects should be destroyed.

To reason about transactions and about the correctness of the management algorithms, it is necessary to define the concept formally. For the simplicity of the exposition, it is assumed that each transaction reads and writes a data item at most once. From now on the abbreviations *r, w, a*, and *c* are used for the read, write, abort, and commit operations, respectively. For simplicity of exposition, we assume that each transaction reads and writes a data item once at the most.

Definition 2.1 A *transaction* T_i is partial order with an ordering relation $\prec_i$ where [7]:

1. $T_i \subseteq \{r_i[x], w_i[x] \mid \text{ x is a data item } \} \cup \{a_i, c_i\}$;
2. $a_i \in T_i$ if and only if $c_i \notin T_i$;
3. if t is c_i or a_i, for any other operation $p \in T_i$, $p \prec_i t$; and
4. if $r_i[x], w_i[x] \in T_i$, then either $r_i[x] \prec_i w_i[x]$ or $w_i[x] \prec_i r_i[x]$.

□

Informally, (1) a transaction is a subset of read, write and abort or commit operations. (2) If the transaction executes an abort operation, then the transaction is not executing a commit operation. (3) If a certain operation t is abort or commit then the ordering relation defines that for all other operations, precede operation t in the execution of the transaction. (4) If both read and write operations are executed to the same data item, then the ordering relation defines the order between these operations. In this thesis we do not consider other operations to the database system. Consideration how to transform SQL or other data management languages to these operations or directly use them as implementation method is out of the scope of this thesis. The interested reader can consult [105].

After the DBS executes a transaction's commit (or abort) operation, the transaction is said to be *committed* (or *aborted*). A transaction that has issued its begin operation but is not yet committed or aborted is called *active.* A transaction is *uncommitted* if it is aborted or active. If a transaction is aborted, the transaction can be *restarted.* In this thesis a restarted transaction starts its execution from the first command in the transaction program and reissues all database operations as new operations. We assume that each transaction is self-contained, meaning that it performs its computation without any direct communication with other transactions or users [7].

A major aim in developing a database system is to allow several users to access shared data concurrently [7]. Concurrent access is easy if all users are only reading data, because there is no way for them to interfere with one another. However, when two or more users are accessing the database concurrently and at least one is updating data, there may be interference that can cause inconsistencies [79].

Although two transactions may be correct in themselves, the interleaving of operations may produce an incorrect result, thus risking the integrity and consistency of the database [7, 22, 79]. The **ACID** properties of a transaction that all transactions should have are [22]:

- **Atomicity**: A transaction's changes to the state are atomic: either all happen or none happen.

- **Consistency**: A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. The formal transaction maintains the consistency.

- **Isolation**: Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

- **Durability**: Results of the committed transaction are persistent until other committed transaction possibly changes them.

The aim of concurrency control methods are to schedule transactions in such a way as to avoid any interference [7]. One obvious solution would be to allow only one transaction to execute at a time. However, the goal of a multiuser database system is also to maximize the degree of concurrency or parallelism in the system, so that transactions that can execute without interfering with one another can run parallelly [79].

When two or more transactions execute concurrently, their database operations execute in an interleaved way [7]. Therefore, operations from one transaction may execute between two operations from another transaction. This interleaving can cause transactions to behave incorrectly. Therefore, interleaved transaction execution can lead to an inconsistent database state. To avoid this and other problems the interleaving between transactions must be controlled [7]. We say that there is a *conflict* between two transactions if both transactions operate on the same data item and at least one of the operations is write. Execution interleaving can be modelled using a history. Formally, let $T = \{T_1, T_2, ..., T_n\}$ be a set of transactions.

Definition 2.2 A *complete history* H over T is a partial order with ordering relation $\prec_H$ where

1. $H = \bigcup_{i=1}^{n} T_i$;
2. $\prec_H \supseteq \bigcup_{i=1}^{n} \prec_i$; and
3. for any two conflicting operations $p, q \in H$, then either $p \prec_H q$ or $q \prec_H p$.

□

Condition (1) says that the execution represented by H involves precisely the operations submitted by $T_1, T_2, ..., T_n$. Condition (2) says that the execution honors all operation orderings specified within each transaction. Finally, condition (3) says that the ordering is determined by every pair of conflicting operations. A *history* is simply a prefix of a complete history.

One method to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial [7]. A complete history H is *serial* if, for every two transactions T_i and T_j that appear in H, either all operations of T_i appear before all operations of T_j or vice versa. Thus, a serial history represents an execution in which there is no interleaving of the operations of different transactions. A history H is *serializable* if its committed projection, $C(H)$, is equivalent to a serial history H. $C(H)$ is a complete history and it is not an arbitrarily chosen complete history. If H represents the execution so far, it is really only the committed transactions whose execution the database management system has unconditionally guaranteed. All other transactions may be aborted.

We can extend the class of correct executions to include executions that have the same effect as serial executions [7]. Such executions are called serializable. Execution is *serializable* [7] if it produces the same output and has the same effect on the database state as some serial execution of the same transactions. Because serial executions are correct and each serializable execution has the same effect as a serial execution, serializable executions are correct, too [79].

We can determine whether a history is serializable by analyzing a graph derived from the history called a serialization graph. The *serialization graph* for H, denoted $SG(H)$, is a directed graph whose edges are all $T_i \rightarrow T_j (i \neq j)$ such that one of T_i's operations precedes and conflicts with one of T_j's operations in H. Therefore, a history H is serializable if and only if $SG(H)$ is acyclic.

To ensure correctness in the presence of failures, the execution of transactions should be not only serializable but also recoverable, avoid cascading aborts, and be strict [7]. An execution is *recoverable* if each transaction commits after the commitment of all other transactions from which it read. An execution *avoids cascading aborts* if the transaction read only those values that are written by committed transactions or by the transaction itself. An execution is *strict* if the transaction reads or overwrites a data item after the transaction that previously wrote into it terminates either by aborting or by committing [7].

In our study of concurrency control, we need a model of the internal structure of a DBS [7]. In our model, a DBS consists of three modules (see 2.4). In this thesis we do not need the cache manager described in the [7] model, because we assume main-memory database system to be used. Therefore, we have integrated the recovery manager and cache manager found in [7] to one data manager module (DM). A *transaction manager* performs any required preprocessing of database and transaction operations it receives from the transactions (see Definition 2.1). A *scheduler* controls the relative order in which database and transaction operations are executed and transaction commitment and abortion. A *data manager* operates directly on the database.

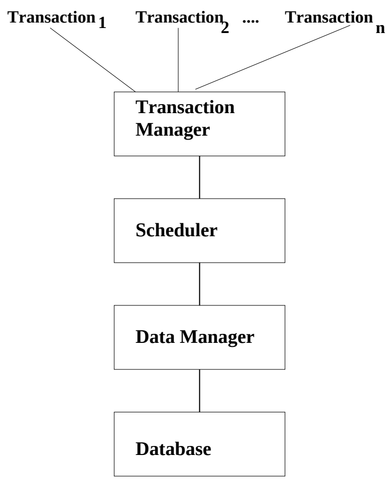


Figure 2.4: Centralized Database System.

In this thesis we concentrate on activities of the scheduler. A scheduler is a program or collection of programs that controls the concurrent execution of transactions. It makes use of this control by restricting the order in which the data manager executes the reads, writes, commits, and aborts of different transactions. The goal of the scheduler is to order these operations so that the resulting execution is serializable and recoverable. It may also ensure that the execution avoids cascading aborts or is strict. To execute a database operation, a transaction passes that operation to the scheduler. After receiving the operation, the scheduler can take one of three actions [7]:

- Execute: It can pass the operation to DM and wait for a result. When DM finishes executing the operation, it informs the scheduler. Additionally, if the operation is a read, the DM returns the value(s) it read, which the scheduler relays back to the transaction. In this thesis, the transaction waits until the scheduler peports the result of the operation.

- Reject: It can refuse to precess the operation, in which case it tells the transaction that its operation has been rejected. This causes the transaction to abort.

- Delay: It can delay the operation by placing it in a queue internal to the scheduler. Later, it can remove the operation from the queue and either execute it or reject it. While the operation is being delayed, the scheduler is free to schedule other operations.

There are many ways in which the schedulers can be classified [7]. One obvious classification criterion is the mode of database distribution. Some schedulers that have been proposed require a fully replicated database, while others can operate on partially replicated or partitioned databases. The schedulers can also be classified according to network topology. The most common classification criterion however is the synchronization primitive, i.e. those methods that are based on mutually exclusive access to shared data and those that attempt to order the execution of the transactions according to a set of rules [111]. There are two possible views: the pessimistic view that many transactions will conflict with each other, or the optimistic view that not too many transactions will conflict with each other. Pessimistic methods synchronize the concurrent execution of transactions early in their execution and optimistic methods delay the synchronization of transactions until their terminations [105]. Therefore, the basic classification is as follows:

- Pessimistic Methods
  - Timestamp Ordering Methods [78, 104]
  - Serialization Graph Testing [7]
  - Locking Methods [89, 34, 5, 57, 32, 58]
- Optimistic Methods
  - Backward Validation Methods [26]
  - Forward Validation Methods [28, 53, 54, 107, 64]
  - Serialization Graph Testing [7, 78, 66]
  - Hybrid Methods [65]
- Hybrid Methods [33, 94, 108, 21, 56]

In the locking-based methods, the synchronization of transactions is acquired by using physical or logical locks on some portion or granule of the database. The timestamp ordering method involves organizing the execution order of transactions so that they maintain mutual and internal consistency. This ordering is maintained by assigning timestamps to both the transactions and the data that are stored in the database [111].

The state of a conventional non-versioning database represent the state of a system at a single moment of time. Although the contents of the database change as new information is added, these changes are viewed as modification to the state. The current contents of the database may be viewed as a snapshot of the system. Additionally, conventional DBSs provide no guarantee of transaction completion times.

In this thesis we consider the database research area real-time database systems for completing database operations within time constraints. Another research area where time is an important part of the database is temporal databases. A good survey of temporal databases can be found e.g. in [96, 49, 110]. In this thesis we do not discuss temporal databases.

A real-time database is a database in which transactions have deadlines or timing constraints. Real-time databases are commonly used in real-time computing applications that require timely access to data. And, usually, the definition of timeliness is not quantified; for some applications it is milliseconds, and for others it is minutes [97, 98]. There are several surveys on real-time databases [63, 83, 109, 110].

In the past two decades, the research in real-time database systems (RTDBS) has received a lot of attention [52, 47, 81, 83, 90]. It consists of two different important areas in computer science: real-time systems and database systems. As tasks in conventional real-time systems, transactions in RTDBS are usually associated with time constraints, e.g. deadlines. On the other hand, RTDBS must maintain a database for useful information, support the manipulation of the database, and process transactions. Typically, these application systems need predictable response times, and they often have to process various kinds of queries in a timely manner. Contrary to traditional database systems, RTDBS must not only maintain database integrity but also meet the urgency of transaction executions [98].

In addition to the timing constraints that originate from the need to continuously track the environment, timing correctness requirements in a real-time database system also appear because of the need to make data available to the controlling system for its decision-making activities [20]. The need to maintain a consistency between the actual state of the environment and the state as returned by the contents of the database leads to the concept of *temporal consistency.* Temporal consistency has two components [92]:

- *Absolute consistency*: Data is only valid between absolute points in time. This is due to the need to keep the database consistent with the environment.

- *Relative consistency*: among the data used to derive other data. This arises from the need to produce the derived data from sources close to each other. This requires that a set of data items used to derive a new data item form a *relative consistency set* R.

Data item d is *temporally consistent* if and only if d is absolutely consistent and relatively consistent [83]. Every data item in the real-time database consists of the current state of the object (i.e. current value stored in that data item), and two timestamps. These timestamps represent the time when this data item was last accessed by the committed transaction. Formally,

Definition 2.3 a *Data item* in the real-time database is denoted by $d : (value, RTS, WTS, avi)$, where d_{value} denotes the current state of d, d_{RTS} denotes when the last committed transaction has read the current state of d, d_{WTS} denotes when the last committed transaction has written d, i.e., when the observation relating to d was made, and d_{avi} denotes d's absolute validity interval, i.e., the length of the time interval following R_{WTS} during which d is considered to have absolute validity. □

A set of data items used to derive a new data item forms a relative consistency set R. Each such set R is associated with a *relative validity interval* denoted R_{rvi}. Assume that $d \in R$. d has a correct state if and only if [83]:

1. d_{value} is logically consistent, i.e., satisfies all integrity constraints.
2. d is temporally consistent:
   - Data item $d \in R$ is absolutely consistent if and only if $(current\_time - d_{WTS}) \leq d_{avi}$.
   - Data items are relatively consistent if and only if $\forall d' \in R |d_{WTS} - d'_{WTS}| \leq R_{rvi}$.

In this thesis we do not consider temporal data or temporal constraints. A good book on temporal databases can be found in [101]. A discussion on integration of temporal and real-time database systems can be found from [85]. Finally, temporal consistency maintainance is discussed in [106].

Several research angles emerge from real-time databases: real-time concurrency control (e.g. [51, 16, 61]), buffer management (e.g. [13]), disk scheduling (e.g. [45, 12]), system failure and recovery (e.g. [91]), overload management (e.g. [25, 14]), security (e.g. [44, 24, 93], and distibuted real-time database systems (e.g. [31, 59, 58, 57, 67, 23, 68]). In this thesis we will focus on concurrency control because it is one of the main research areas in the real-time database systems.

Transactions can be characterized along three dimensions; the fashion in which data is used by transactions, the nature of time constraints, and the significance of executing a transaction by its deadline, or more precisely, the consequence of missing specified time constraints [1].

A *real-time transaction* is a transaction (see Definition 2.1) with additional real-time attributes. These attributes are used by the real-time scheduling algorithms and concurrency control methods. Consider the abstract database model presented in Figure 2.4. Additional real-time attributes can be used at all levels and these attributes are passed as a parameters in operations. Additional attributes are the following:

- The deadline is a timing constraint associated with the transaction denoted by $deadline(T_i)$. The developer assigns a value for the deadline based on an estimate or experimentally measured value of worst case execution time.

- The priority is a scheduling priority for the transactions calculated by a scheduling algorithm (EDF in this thesis) and is based on the deadline and arrival time. This attribute is denoted by $priority(T_i)$.

- The criticality of the transaction attribute is denoted by $criticality(T_i)$. The criticality attribute is assigned by the developer and is static and the same for all instances of the same transaction class.

- The deadline-driven conflict priority denoted by $cpriority(T_i)$ is calculated from the deadline and the criticality of the transaction. This value is calculated when the transaction enters the system and remains the same in the execution of the transaction. The following values coded to numbers are used:

    1. **Normal**: The transaction is not essential but should be completed if the execution history is serializable.
    2. **Medium**: The transaction is critical and should not be restarted if there is data conflict with the transaction with lower conflict priority.
    3. **Critical**: The transaction is very critical and a transaction with lower conflict priority is *always* restarted if there is conflict with the transaction with critical conflict priority.

This model is correct in real-time databases covered in this thesis, because additional real-time transaction attributes do not affect the definition of the history or serializability. Real-time databases might restrict possible serializable histories so that higher priority transactions have preference over other transactions. But this does not affect correctness.

Real-time information can be used to tailor the appropriate concurrency control methods [47]. Some transaction-time constraints come from temporal consistency requirements and some come from requirements imposed on system reaction time. The former typically take the form of periodicity requirements. Transactions can also be distinguished based on the effect of missing a transaction's deadline.

Transaction processing and concurrency control in a real-time database system should be based on priority and criticalness of the transactions [99]. Traditional methods for transaction processing and concurrency control used in a real-time environment would cause some unwanted behavior. Below, the four typified problems are characterized and priority is used to denote either scheduling priority or criticality of the transaction:

- **wasted restart:** A wasted restart occurs if a higher priority transaction aborts a lower priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted wait:** A wasted wait occurs if a lower priority transaction waits for the commit of a higher priority transaction and later the higher priority transaction is discarded when it misses its deadline.
- **wasted execution:** A wasted execution occurs when a lower priority transaction in the validation phase is restarted due to a conflicting higher priority transaction which has not finished yet.
- **unnecessary restart:** An unnecessary restart occurs when a transaction in the validation phase is restarted even when history would be serializable.

Traditional pessimistic two-phase locking methods suffer from the problem of wasted restart and wasted wait. Optimistic methods suffer the problems of wasted execution and unnecessary restart [62].

Chapter 3

Concurrency Control in Real-Time Databases

A *Real-Time Database System* (RTDBS) processes transactions with timing constraints such as deadlines [83]. Its primary performance criterion is timeliness, not average response time or throughput. The scheduling of transactions is driven by priority order. Given these challenges, considerable research has recently been devoted to designing concurrency control methods for RTDBSs and to evaluating their performance (e.g. [2, 29, 33, 36, 56, 53] Most of these methods are based on one of the two basic concurrency control mechanisms: *locking* [18] or *optimistic concurrency control* (OCC) [50].

In real-time database systems transactions are scheduled according to their timing constraints. Task scheduler assigns a priority for a task based on its timing constraints or criticality or both [83]. Therefore, high priority transactions are executed before lower priority transactions. This is true only if a high priority transaction has some database operation ready for execution. If no operation from a higher priority transaction is ready for execution, then an operation from a lower priority transaction is allowed to execute its database operation. Therefore, the operation of the higher priority transaction may conflict with the already executed operation of the lower priority transaction. In non-pre-emptive methods a higher priority transaction must wait for the release of the resource. This is the priority inversion problem presented earlier. Therefore, data conflicts in concurrency control should also be based on transaction priorities or criticalness or both. Hence, numerous traditional concurrency control methods have been extended to the real-time database systems.

In the following sections recent related work on locking and optimistic methods for real-time databases are presented. Of optimistic methods two well-known methods are presented in detail. These methods are used as reference methods when comparing the proposed methods. Finally, an evaluation of the optimistic methods is presented.

3.1 Locking Methods in Real-Time Databases

In this section we present some well known pessimistic concurrency control methods. Most of these methods are based on 2PL.

2PL High Priority In the 2PL-HP (2PL High Priority) concurrency control method [2, 4, 34] conflicts are resolved in favor of the higher priority transactions. If the priority of the lock requester is higher than the priority of the lock holder, the lock holder is aborted, rolled back and restarted. The lock is granted to this requester and the requester can continue its execution. If the priority of the lock requester is lower than that of the lock holder, the requesting transaction blocks to wait for the lock holder to finish and release its locks. High Priority concurrency control may lead to the cascading blocking problem, a deadlock situation, and priority inversion.

2PL Wait Promote In 2PL-WP (2PL Wait Promote) [3, 34] the analysis of concurrency control method is developed from [2]. The mechanism presented uses shared and exclusive locks. Shared locks permit multiple concurrent readers. A new definition is made - the priority of a data object, which is defined to be the highest priority of all the transactions holding a lock on the data object. If the data object is not locked, its priority is undefined.

A transaction can join in the read group of an object if and only if its priority is higher than the maximum priority of all transactions in the write group of an object. Thus, conflicts arise from incompatibility of locking modes as usual. Particular attention is given to conflicts that lead to priority inversions. A priority inversion occurs when a transaction of high priority requests and blocks for an object which has lesser priority. This means that all the lock holders have lesser priority than the requesting transaction. This same method is also called 2PL-PI (2PL Priority Inheritance) [34].

2PL Conditional Priority Inheritance Sometimes High Priority may be too strict policy. If the lock holding transaction T_h can finish in the time that the lock requesting transaction T_r can afford to wait, that is within the slack time of T_r, and let T_h proceed to execution and T_r wait for the completion of T_h. This policy is called 2PL-CR (2PL Conditional Restart) or 2PL-CPI (2PL Conditional Priority Inheritance) [34].

Priority Ceiling Protocol [87, 88] the focus is to minimize the duration of blocking to at most one lower priority task and prevent the formation of deadlocks. A real-time database can often be decomposed into sets of database objects that can be modeled as atomic data sets. For example, two radar stations track an aircraft representing the local view in data objects O_1 and O_2. These objects might include e.g. the current location, velocity, etc. Each of these objects forms an atomic data set, because the consistency constraints can be checked and validated locally. The notion of atomic data sets is especially useful for tracking multiple targets.

A simple locking method for elementary transactions is the two-phase locking method; a transaction cannot release any lock on any atomic data set unless it has obtained all the locks on that atomic data set. Once it has released its locks it cannot obtain new locks on the same atomic data set, however, it can obtain new locks on different data sets. The theory of modular concurrency control permits an elementary transaction to hold locks across atomic data sets. This increases the duration of locking and decreases preemptibility. In this study transactions do not hold locks across atomic data sets.

Priority Ceiling Protocol minimizes the duration of blocking to at most one elementary lower priority task and prevents the formation of deadlocks [87, 88]. The idea is that when a new higher priority transaction preempts a running transaction its priority must exceed the priorities of all preempted transactions, taking the priority inheritance protocol into consideration. If this condition cannot be met, the new transaction is suspended and the blocking transaction inherits the priority of the highest transaction it blocks.

The priority ceiling of a data object is the priority of the highest priority transaction that may lock this object [87, 88]. A new transaction can preempt a lock-holding transaction if and only if its priority is higher than the priority ceilings of all the data objects locked by the lock-holding transaction. If this condition is not satisfied, the new transaction will wait and the lock-holding transaction inherits the priority of the highest transaction that it blocks. The lock-holder continues its execution, and when it releases the locks, its original priority is resumed. All blocked transactions are awaked, and the one with the highest priority will start its execution.

The fact that the priority of the new lock-requesting transaction must be higher than the priority ceiling of all the data objects that it accesses, prevents the formation of a potential deadlock. The fact that the lock-requesting transaction is blocked only at most the execution time of one lower priority transaction guarantees, the formation of blocking chains is not possible [87, 88].

Read/Write Priority Ceiling The Priority Ceiling Protocol is further advanced in [89], where the Read/Write Priority Ceiling Protocol is introduced. It contains two basic ideas. The first idea is the notion of priority inheritance. The second idea is a total priority ordering of active transactions. A transaction is said to be active if it has started but not completed its execution. Thus, a transaction can execute or wait caused by a preemption in the middle of its execution. Total priority ordering requires that each active transaction execute at a higher priority level than the active lower priority transaction, taking priority inheritance and read/write semantics into consideration.

3.2 Optimistic Methods in Real-Time Databases

Optimistic Concurrency Control (OCC) [26, 50], is based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without delays. When a transaction desires to commit, a check is performed to determine whether a conflict has occurred. Therefore, there are three phases to an optimistic concurrency control method:

- *Read phase*: The transaction reads the values of all data items it needs from the database and stores them in local variables. Concurrency control scheduler stores identity of these data items to a read set. However, writes are applied only to local copies of the data items kept in the transaction workspace. Concurrency control scheduler stores identity of all written data items to a write set.

- *Validation phase*: The validation phase ensures that all the committed transactions have executed in a serializable fashion. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. For a transaction that has writes, the validation consists of determining whether the current transaction has executed serializable way.

- *Write phase*: This follows the successful validation phase for transactions including write operations. During the write phase, all changes made by the transaction are permanently stored into the database.

In the following we introduce some well known optimistic methods for real-time database systems.

Broadcast Commit For RTDBSs, a variant of the classical concurrency control method is needed. In Broadcast Commit, OPT-BC [29], when a transaction commits, it notifies other running transactions that conflict with it. These transactions are restarted immediately. There is no need to check a conflict with committed transactions since the committing transaction would have been restarted in the event of a conflict. Therefore, a validating transaction is always guaranteed to commit. The broadcast commit method detects the conflicts earlier than the conventional concurrency control mechanism, resulting in earlier restarts, which increases the possibility of meeting the transaction deadlines [29].

The main reason for the good performance of locking in a conventional DBMS is that the blocking-based conflict resolution policy results in conservation of resources, while the optimistic method with its restart-based conflict resolution policy wastes more resources [29]. But in a RTDBS environment, where conflict resolution is based on transaction priorities, the OPT-BC policy effectively prevents the execution of a low priority transaction that conflicts with a higher priority transaction, thus decreasing the possibility of further conflicts and the waste of resources is reduced. Conversely, 2PL-HP loses some of the basic 2PL blocking factor due to the partially restart-based nature of the High Priority scheme.

The delayed conflict resolution of optimistic methods aids in making better decisions since more information about the conflicting transactions is available at this stage [29]. Compared to 2PL-HP, a transaction could be restarted by, or wait for, another transaction which is later discarded. Such restarts or waits are useless and cause performance degradation. OPT-BC guarantees the commit and thus the completion of each transaction that reaches the validation stage. Only validating transactions can cause the restart of other transactions, thus, all restarts generated by the OPT-BC method are useful.

First of all, OPT-BC has a bias against long transactions (i.e. long transactions are more likely to be aborted if there is conflicts), like in the conventional optimistic methods [29]. Second, as the priority information is not used in the conflict resolution, a committing lower priority transaction can restart a higher priority transaction very close to its validation stage, which will cause missing the deadline of the restarted higher priority transaction [28].

OPT-SACRIFICE In the OPT-SACRIFICE [28] method, when a transaction reaches its validation phase, it checks for conflicts with other concurrently running transactions. If conflicts are detected and at least one of the conflicting transactions has a higher priority, then the validating transaction is restarted, i.e. sacrificed in favor of the higher priority transaction. Although this method prefers high priority transactions, it has two potential problems. Firstly, if a higher priority transaction causes a lower priority transaction to be restarted, but fails in meeting its deadline, the restart was useless. This degrades the performance. Secondly, if priority fluctuations are allowed, there may be the mutual restarts problem between a pair of transactions (i.e. both transactions are aborted). These two drawbacks are analogous to those in the 2PL-HP method [28].

OPT-WAIT and WAIT-X When a transaction reaches its validation phase, it checks if any of the concurrently running other transactions have a higher priority. In the OPT-WAIT [28] case the validating transaction is made to wait, giving the higher priority transactions a chance to make their deadlines first. While a transaction is waiting, it is possible that it will be restarted due to the commit of one of the higher priority transactions. Note that the waiting transaction does not necessarily have to be restarted. Under the broadcast commit scheme a validating transaction is said to conflict with another transaction, if the intersection of the write set of the validating transaction and the read set of the conflicting transaction is not empty. This result does not imply that the intersection of the write set of the conflicting transaction and the read set of the validating transaction is not empty either [28].

The WAIT-50 [28] method is an extension of the OPT-WAIT - it contains the priority wait mechanism from the OPT-WAIT method and a wait control mechanism. This mechanism monitors transaction conflict states and dynamically decides when and for how long a low priority transaction should be made to wait for the higher priority transactions. In WAIT-50, a simple 50 percent rule is used - a validating transaction is made to wait while half or more of its conflict set is composed of transactions with higher priority. The aim of the wait control mechanism is to detect when the beneficial effects of waiting are outweighed by its drawbacks [28].

We can view OPT-BC, OPT-WAIT and WAIT-50 as being special cases of a general WAIT-X method, where X is the cutoff percentage of the conflict set composed of higher priority transactions. For these methods X takes the values infinite, 0 and 50 respectively.

3.3 Validation Methods

The validation phase ensures that all the committed transactions have executed in a serializable fashion [50]. Most of the validation methods use the following principles to ensure serializability. If a transaction T_i is before transaction T_j in the serialization graph(i.e. $T_i \prec T_j$), the following two conditions must be satisfied [62]:

1. No overwriting. The writes of T_i should not overwrite the writes of T_j.
2. No read dependency. The writes of T_j should not affect the read phase of T_i.

Generally, condition 1 is automatically ensured in most optimistic methods because I/O operations in the write phase are required to be done sequentially in the critical section [62]. Thus most validation schemes consider only condition 2. During the write phase, all changes made by the transaction are permanently installed into the database. To design an efficient real-time optimistic concurrency control method, three issues have to be considered [62]:

1. which validation scheme should be used to detect conflicts between transactions;
2. how to minimize the number of transaction restarts; and
3. how to select a transaction or transactions to restart when a nonserializable execution is detected.

In *Backward Validation* [26], the validating transaction is checked for conflicts against (recently) committed transactions. Conflicts are detected by comparing the read set of the validating transaction and the write sets of the committed transactions. If the validating transaction has a data conflict with any committed transactions, it will be restarted. The classical optimistic method in [50] is based on this validation process.

In *Forward Validation* [26], the validating transaction is checked for conflicts against other active transactions. Data conflicts are detected by comparing the write set of the validating transaction and the read set of the active transactions. If an active transaction has read an object that has been concurrently written by the validating transaction, the values of the object used by the transactions are not consistent. Such a data conflict can be resolved by restarting either the validating transaction or the conflicting transactions in the read phase. Optimistic methods based on this validation process are studied in [26]. Most of the proposed optimistic methods are based on Forward Validation.

Forward Validation is preferable for the real-time database systems because Forward Validation provides flexibility for conflict resolution [26]. Either the validating transaction or the conflicting active transactions may be chosen to restart. In addition to this flexibility, Forward Validation has the advantage of early detection and resolution of data conflicts. In recent years, the use of optimistic methods for concurrency control in real-time databases has received more and more attention. Different real-time optimistic methods have been proposed.

Forward Validation (OCC-FV) [26] is based on the assumption that the serialization order of transactions is determined by the arriving order of the transactions at the validation phase. Thus the validating transaction, if not restarted, always precedes concurrently running active transactions in the serialization order. A validation process based on this assumption can cause restarts that are not necessary to ensure data consistency. These restarts should be avoided.

The major performance problem with optimistic concurrency control methods is the late restart [62]. Sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occurred, even though it has not done so in actual execution [94] (see Example 3.1). Therefore, one important mechanism to improve the performance of an optimistic concurrency control method is to reduce the number of restarted transactions.

Example 3.1 Consider the following transactions T_1, T_2 and history H_1:
T_1: $r_1[x]c_1$
T_2: $w_2[x]c_2$
H_1: $r_1[x]w_2[x]c_2$

Based on the OCC-FV method [26], T_1 has to be restarted. However, this is not necessary, because when T_1 is allowed to commit such as:

$H_2 : r_1[x]w_2[x]c_2\mathbf{c_1}$,

then the schedule of H_2 is equivalent to the serialization order $T_1 \rightarrow T_2$ as the actual write of T_2 is performed after its validation and after the read of T_1. There is no cycle in their serialization graph and H_2 is serializable. □

One way to reduce the number of transaction restarts is to dynamically adjust the serialization order of the conflicting transactions [62]. Such methods are called *dynamic adjustment of the serialization order* [62]. When data conflicts between the validating transaction and active transactions are detected in the validation phase, there is no need to restart conflicting active transactions immediately. Instead, a serialization order of these transactions can be dynamically defined.

Definition 3.2 Suppose there is a validating transaction T_v and a set of active transactions $T_j (j = 1, 2, ..., n)$. There are three possible types of data conflicts which can cause a serialization order between T_v and T_j [62, 69, 94]:

1. $RS(T_v) \cap WS(T_j) \neq \emptyset$ (read-write conflict)
   A read-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ so that the read of T_v cannot be affected by T_j's write. This type of serialization adjustment is called *forward ordering* or *forward adjustment.*

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ (write-read conflict)
   A write-read conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_j \rightarrow T_v$. It means that the read phase of T_j is placed before the write of T_v. This type of serialization adjustment is called *backward ordering* or *backward adjustment.*

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ (write-write conflict)
   A write-write conflict between T_v and T_j can be resolved by adjusting the serialization order between T_v and T_j as $T_v \rightarrow T_j$ such that the write of T_v cannot overwrite T_j's write (forward ordering).

$\square$

3.4 OCC-TI

This section presents perhaps the most well-known optimistic method Optimistic Concurrency Control with Timestamp Intervals (OCC-TI).

The OCC-TI [62, 60] method resolves conflicts using the timestamp intervals of the transactions. Every transaction must be executed within a specific time slot. When an access conflict occurs, it is resolved using the read and write sets of the transaction together with the allocated time slot. Time slots are adjusted when a transaction commits.

In this method, every transaction in the read phase is assigned a timestamp interval (TI). This timestamp interval is used to record a temporary serialization order during the execution of the transaction. At the start of the execution, the timestamp interval of the transaction is initialized as

```
read($T_i, D_i$)
{
    $TI(T_i) = TI(T_i) \cap [WTS(D_i), \infty[$ ;

     if $TI(T_i) = []$ then
        restart($T_i$);

    read($D_i$);
}

pre-write($T_i, D_i$)
{
    $TI(T_i) = TI(T_i) \cap [WTS(D_i), \infty[ \cap [RTS(D_i), \infty[$ ;

     if $TI(T_i) = []$ then
        restart($T_i$);
}
```

Algorithm 3.1: Read algorithm and pre-write algorithm for the OCC-TI.

$[0, \infty[$, i.e., the entire range of timestamp space. Whenever the serialization order of the transaction is changed by its data operation or the validation of other transactions, its timestamp interval is adjusted to represent the dependencies.

In the read phase when a read operation is executed, the write timestamp (WTS) of the object accessed is verified against the timestamp interval allocated to the transaction. If another transaction has written the object outside the timestamp interval, the transaction must be restarted. In the read algorithm (Algorithm 3.1) D_i is the object to be read, T_i is the transaction reading the object, $TI(T_i)$ is the timestamp interval allocated to the transaction, $WTS(D_i)$ is the write timestamp of the object, and $RTS(D_i)$ is the read timestamp of the object. This algorithm is executed for all objects read in the transaction.

In the read phase when a write operation is executed, the modification is made to a local copy of the object. A *pre-write* operation is used to verify read and write timestamps of the written object against the timestamp interval allocated to the transaction (Algorithm 3.1). If another transaction has read or written the object outside the timestamp interval, the transaction must be restarted.

The noticeable point of OCC-TI is that unlike other optimistic methods, it does not depend on the assumption of the serialization order of transactions as being the same as the arriving order in the validation phase [62].

However, as we can see OCC-TI is not exactly optimistic, because some conflict checking is already done in the read phase. The idea behind this is to detect nonserializable executions as soon as possible and restart nonserializable transaction early. This will prevent unnecessary execution of the nonserializable transaction to its validation phase, thus avoiding resource wasting.

At the validation phase (Algorithm 3.2), the final timestamp of the validated transaction is determined from the timestamp interval allocated to the transaction. In this method, the minimum value of $TI(T_v)$ is selected as the timestamp $TS(T_v)$ [62, 60]. The timestamp intervals of all active concurrently running and conflicting transactions must be adjusted to reflect the serialization order. Any transaction whose timestamp interval becomes empty must be restarted. The adjustment of timestamp intervals of active transactions iterates through the readset and the writeset. When access has been made to the same objects both in the validating transaction T_v and in the active transaction T_a, the timestamp interval of the $TI(T_a)$ is adjusted.

3.5 OCC-DA

In this section we present another well known optimistic concurrency control method Optimistic Concurrency Control with Dynamic Adjustment (OCC-DA).

OCC-DA [53] is based on the Forward Validation scheme [26]. The number of transaction restarts is reduced by using dynamic adjustment of the serialization order. This is supported with the use of a dynamic timestamp assignment scheme. Conflict checking is performed at the validation phase of a transaction. No adjustment of the timestamps is necessary in case of data conflicts in the read phase. In OCC-DA the serialization order of committed transactions may be different from their commit order.

In OCC-DA for each transaction, T_i, there is a timestamp called *serialization order timestamp* $SOT(T_i)$ to indicate its serialization order relative to other transactions. Initially, the value of $SOT(T_i)$ is set to be ∞. If the value of $SOT(T_i)$ is other than ∞, it means that T_i has been backward adjusted before a committed transaction.

If T_i has been backward adjusted, $SOT(T_i)$ will also be used to detect whether T_i has accessed any invalid data item. This is done by comparing its timestamp with the timestamps of the committed transactions which have read or written the same data item. A data item in its read set and write set is invalidated if the data item has been updated by other

```
occti_validate($T_v$)
{
    $TS(T_v) = min(TI(T_v))$;

    for $\forall D_i \in ( RS(T_v) \cup WS(T_v) )$
    {
        for $\forall T_a \in active\_conflicting\_transactions()$
        {
            if $D_i \in ( WS(T_a) \cap RS(T_v) )$ then
                $TI(T_a) = TI(T_a) \cap [TS(T_v), \infty[$ ;

            if $D_i \in ( RS(T_a) \cap WS(T_v) )$ then
                $TI(T_a) = TI(T_a) \cap [0, TS(T_v) - 1]$;

            if $D_i \in ( WS(T_a) \cap WS(T_v) )$ then
                $TI(T_a) = TI(T_a) \cap [TS(T_v), \infty[$ ;

            if $TI(T_a) = []$ then
                restart($T_a$);
        }

        if $D_i \in RS(T_v)$ then $RTS(D_i) = max(RTS(D_i), TS(T_v))$;

        if $D_i \in WS(T_v)$ then $WTS(D_i) = max(WTS(D_i), TS(T_v))$;
    }

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

Algorithm 3.2: Validation algorithm for the OCC-TI.

committed transactions which have been defined after the transaction in the serialization order.

When the validating transaction T_v comes to the validation, the set of active transactions, T_i, whose serialization order timestamp, $SOT(T_v) \geq SOT(T_i)$ are collected to set $ATS(T_v)$. The set of active transactions, T_j, whose serialization order timestamp, $SOT(T_j) < SOT(T_v)$ are collected to set $BTS(T_v)$. In read phase $TR(T_v, D_p)$ is set to be $WTS(D_p)$ of the read data item D_p.

The first part of the validation phase is used only for those validating transactions which have been backward adjusted (Algorithm 3.3). It is to check whether:

1. all the read operations of T_v have been read from the committed transactions T_c whose $SOT(T_c) < SOT(T_v)$, and

```
part_one($T_v$)
{
    if  $SOT(T_v) \neq \infty$  then
    {
        for  $\forall D_p \in RS(T_v)$
        {
            if  $TR(T_v, D_p) > SOT(T_v)$  then
                restart($T_v$);
        }

        for  $\forall D_p \in WS(T_v)$
        {
            if  $SOT(T_v) < RTS(D_p)$  or  $SOT(T_v) < WTS(D_p)$  then
                restart($T_v$);
        }
    }
}
```

Algorithm 3.3: The first part of the validation algorithm for OCC-DA.

2. whether T_v's write is invalidated. This is done by comparing $SOT(T_v)$ with $WTS(D_p)$ and $RTS(D_p)$ of the data item D_p in T_v's write set or read set.

The purpose of part two of the validation phase is to detect read-write conflicts between the active transactions and the validating transactions (Algorithm 3.4). The write set of the validating transaction T_v is compared with the read set of the active transaction T_j. The identity of the conflicting active transactions T_j are added to $BTlist(T_v)$ to indicate that T_j needs to be backward adjusted before T_v.

The third part of the validation phase is to detect whether a backward-adjusted transaction T_j also needs forward adjustment with respect to T_v (Algorithm 3.5). It compares the write set of T_j which is in $BTS(T_v)$ or in $BTlist(T_v)$ with the read set of T_v, and the write set of T_v with the write sets of T_j. If either one of them is not empty, T_j is in serious conflict with T_v. In conflict resolution, transactions for restart are selected based on priorities i.e. a lower priority transactions is restarted or an active transaction is restarted.

When the validating transaction reaches part four of the validation phase, it is guaranteed to commit. The purpose is to assign a final commitment timestamp to the validating transaction and to update the necessary timestamps of the data items (Algorithm 3.6).

```
part_two($T_v$)
{
    $BTlist(T_v) = \emptyset$ ;

    for $\forall T_j \in ATS(T_v)$
    {
        for $\forall D_p \in WS(T_v)$
        {
            if $D_p \in RS(T_j)$ then
                $BTlist(T_v) = BTlist(T_v) \cup T_j$ ;
        }
    }
}
```

Algorithm 3.4: The second part of the validation algorithm for OCC-DA.

```
part_tree($T_v$)
{
    for $\forall T_j \in BTS(T_v) \cup BTlist(T_v)$
    {
        for $\forall D_p \in RS(T_v)$
        {
            if $D_p \in WS(T_j)$ then
                conflict_resolution($T_v$,$T_j$);
        }

        for $\forall D_p \in WS(T_v)$
        {
            if $D_p \in WS(T_j)$ then
                conflict_resolution($T_v$,$T_j$);
        }
    }
}
```

Algorithm 3.5: The third part of the validation algorithm for OCC-DA.

3.6 Evaluation of Methods

As presented earlier (see Example 3.1), the major performance problem with optimistic concurrency control methods are the heavy restart overhead, wasting a large amount of resources [62]. Sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occurred, even though it has not in actual execution [94].

OCC-BC restarts all active conflicting transactions. OPT-SACRIFICE

```
part_four($T_v$)
{
    if  $SOT(T_v) = \infty$  then
        $SOT(T_v) = validation\_time$ ;

    for  $\forall\ T_j \in BTlist(T_v)$
        $SOT(T_j) = SOT(T_v) - \epsilon$ ; //infinitesimal quantity

    for  $\forall\ D_p \in RS(T_v)$
        $RTS(D_p) = SOT(T_v)$ ;

    for  $\forall\ D_p \in WS(T_v)$
        $WTS(D_p) = SOT(T_v)$ ;

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

Algorithm 3.6: The fourth part of the validation algorithm for OCC-DA.

restarts the validating transaction if at least one of the conflicting transactions has a higher priority. The OPT-WAIT-X family restarts the active conflicting transactions. Therefore, the OCC-BC, OPT-SACRIFICE, OPR-WAIT, and OPT-WAIT-X methods all unnecessary restarts transactions. Hence, a new validation method was proposed for the OCC-TI method in [62] which was presented in Section 3.4. The same validation method but different implementation is used in the OCC-DA method [53]. These methods are selected as compare methods because they are well known and shown to work very well in real-time database systems.

Performance studies in [62] have shown that under the policy that discards tardy transactions from the system, the optimistic methods outperform 2PL-HP [2]. OCC-TI does better than OPT-BC among the optimistic methods [62]. The performance difference between OPT-BC and OCC-TI becomes large especially when the probability of a data object read being updated is low, which is true in most actual database systems.

Performance studies in [53] have shown that OCC-DA outperforms OCC-TI. OCC-DA can be extended to use Thomas's write rule [102] and this extension is presented in [54].

However, the algorithms provided for OCC-TI in [62, 60] do not seem to fully resolve the unnecessary restart problem. The problem with the existing algorithm is best described by the example given below.

Example 3.3 Let $RTS(x)$ and $WTS(x)$ be initialized as 100. Consider transactions T_1, T_2, and history H_1:

T_1: $r_1[x]w_1[x]c_1$
T_2: $r_2[x]w_2[y]c_2$
H_1: $r_1[x]r_2[x]w_1[x]c_1w_2[y]c_2$.

Transaction T_1 executes $r_1[x]$, which causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [0,\infty[\cap [100,\infty[= [100,\infty[$. T_2 then executes a read operation on the same object, which causes the timestamp interval of the transaction to be forward adjusted similarly. T_1 then executes $w_1[x]$. This operation is executed inside the OCC-TI scheduler using pre-write operation and no operation is generated in the output history. Write operation is delayed until the transaction is successfully validated. This is similar as in pessimistic 2PL using delayed writes. Pre-write operation causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [100,\infty[\cap [100,\infty[\cap [100,\infty[= [100,\infty[$. T_1 starts the validation, and the final (commit) timestamp is selected to be $TS(T_1) = min([100,\infty[) = 100$. Because there is one read-write conflict between the validating transaction T_1 and the active transaction T_2, the timestamp interval of the active transaction must be adjusted. Thus $TI(T_2) = [100,\infty[\cap [0,99] = []$. The timestamp interval is shut out, and T_2 must be restarted. However this restart is unnecessary, because serialization graph $SG(H_1)$ is acyclic, that is, history H_1 is serializable. Taking the minimum as the commit timestamp $(TS(T_1))$ was not a good choice here.

□

Similarly, OCC-DA also unnecessarily restarts transactions. The problem with the existing algorithm is best described by the example given below.

Example 3.4 Let all objects timestamp be initialized as 100. Consider transactions T_3, T_4, T_5, and history H_2:
T_3: $r_3[x]\ w_3[x]\ c_3$
T_4: $r_4[x]\ w_4[y]\ c_4$
T_5: $r_5[y]\ w_5[z]\ c_5$
H_2: $r_3[x]\ r_4[x]\ r_5[y]\ w_3[x]\ w_4[y]\ w_5[z]\ c_3\ c_5\ c_4$

All operations before the first commit operation are executed successfully. Let us consider the execution of the c_3 operation. Assume that transaction T_3 arrives to its validation phase at time 600. In this case the set $BTS(T_3)$ is empty and the set $ATS(T_3) = \{T_4, T_5\}$. Because $x \in WS(T_3) \cap RS(T_5)$, the active conflicting transaction T_4 is added to the backward set, i.e. $BTlist(T_3) = \{T_4\}$. In part three of the OCC-DA validation algorithm there are no detectable conflicts. In part four of the OCC-DA validation algorithm $SOT(T_3) = 600$ and $SOT(T_4) = 599$ are set. The validation of the transaction T_3 is completed and the write set of the validating transaction (i.e. $WS(T_3)$) is committed to the database.

Let us consider then the execution of the c_5 operation. Assume that transaction T_5 arrives at its validation phase at time 700. In this case the set $BTS(T_5) = \{T_4\}$ and the set $ATS(T_5) = \emptyset$. In part three of the OCC-DA validation algorithm a conflict is detected, because $y \in RS(T_5) \cap WS(T_4)$. Therefore, the transaction T_3 or the transaction T_5 must be restarted. However this restart is unnecessary, because serialization graph $SG(H_2)$ is acyclic, that is, history H_2 is serializable. □

As the conflict resolution between the transactions is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. However, the problem with optimistic concurrency control methods is the late conflict detection, which makes the restart overhead heavy as some near-to-complete transactions have to be restarted. Therefore, the major concern in the design of real-time optimistic concurrency control methods is not only to incorporate priority information for the conflict resolution but also to design methods to minimize the number of transaction restarts. Hence, unnecessary restart problems found in OCC-TI is not desirable. Unnecessary restart found in OCC-DA is not so important because situation explained in the example is not very common.

3.7 Priority Cognizance

Priority cognizant concurrency control methods based on the optimistic methods have not been widely studied. Researchers have speculated that priority cognizant optimistic concurrency control methods, if designed well, could outperform priority insensitive ones in real-time database systems. In this section we survey the work done in this research question. The original OCC-TI method has no real-time properties and is not priority cognizant. However, already in [61], priority cognizant versions of the OCC-TI method was presented. In the proposed methods, if a nonserializable history is found then several different priority-based conflict resolution schemes were proposed:

- Priority abort [33]: When a transaction reaches its validation phase, it is aborted if its priority is less than that of all the conflicting transaction; otherwise, it commits and all the conflicting transactions are restarted.

- Priority sacrifice [28]: When a transaction reaches its validation phase, it is aborted if there exists one or more conflicting transaction with higher priority; otherwise it commits and all the conflicting transactions are restarted. This is the same as the OPT-SACRIFICE [28] method presented earlier.

- Priority wait [28]: When a transaction reaches its validation phase, if there exist one or more conflicting transactions with higher priority, it waits for those transactions to complete. This is the same as the OPT-WAIT [28] method presented earlier.

- Wait-50 [28]: A validating transaction is made to wait as long as more than half of the conflicting transactions have higher priorities; otherwise it commits and all the conflicting transactions are restarted. This is the same as the WAIT-50 [28] method presented earlier.

Also it was noted that the final timestamp $TS(T_v)$ is determined so that the order induced by the final timestamp should not destroy the serialization order constructed by the already committed transactions [61]. But no methods to select the final timestamp were proposed. Finally, the paper did not present any performance results.

Very similar work is later done in [16]. They also extend OCC-TI with OPT-SACRIFICE [28], OPT-WAIT [28], WAIT-50 [28], and a new conservative sacrifice method. In conservative sacrifice, the validating transaction is restarted only if all transactions in the conflict set have a higher priority.

Additionally, a new optimistic concurrency control method OCC-APR (Optimistic Concurrency Control - Adaptive PRiority) was presented. OCC-APR is an extension to the original OCC-TI method. OCC-APR identifies how much time a validating transaction has left to its deadline and restarts it only if sufficient time is left. This is done by storing the number of data accesses, average CPU time, and service time and comparing these with the deadline. Restart is done only if a nonserializable history is found. Again, no methods to select the final timestamp is presented. Therefore, the unnecessary restart problem in the original OCC-TI can affect the results presented in [16].

In [16] is postulated that critical condition that validating transaction should have large number of transactions in conflict set to make priority cognizant conflict resolution to work. Surprisingly, it turns out that it is very difficult for priority cognizance to work as the above condition does not, usually, hold. This is because restarting a validating transaction would only make a difference if there were significant numbers of transactions that were likely to commit in the conflict set. This condition will not usually hold. Performance of several concurrency control methods were analyzed across a wide range of resource contention and system loading parameters [16]. All results show that in disk-based real-time database systems, priority cognizance does not seem to be a good approach for improving the performance of real-time concurrency control methods beyond the current state-of-the-art.

Clearly, it takes too much resources to store and maintain the number of data accesses, average CPU time, and service time and comparing the required restart time to the deadline. Other extensions in [16] to OCC-TI are too conservative, i.e. too eagerly restarting conflicting transactions. Additionally, the unnecessary restart problem found in the original OCC-TI can have a significant affect on the results. Finally, is the priority cognizance viable approach for main-memory real-time database systems is still an open research question. We will study this open research question later in this thesis.

Chapter 4

OCC-DATI

As we have seen, one of the main problems with the optimistic methods is unnecessary restarts. This is due to the late conflict detection that increases the restart overhead since some near-to-completed transactions have to be restarted. Therefore, this chapter proposes a new optimistic concurrency control method which tries to avoid unnecessary restarts. This work is based on earlier work presented in [74].

This chapter presents an optimistic concurrency control method named Optimistic Concurrency Control with Dynamic Adjustment using Timestamp Intervals (OCC-DATI) [74]. OCC-DATI is based on forward validation [26]. The number of transaction restarts is reduced by dynamic adjustment of the serialization order which is supported by similar timestamp intervals as in OCC-TI [63]. Unlike the OCC-TI method, all checking is performed at the validation phase of each transaction. There is no need to check for conflicts while a transaction is still in its read phase. As the conflict resolution between the transactions in OCC-DATI is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict. OCC-DATI also has a new final timestamp selection method compared to OCC-TI.

OCC-DATI differs from OCC-DA [53] in several ways. Timestamp intervals have been adopted as the method to implement dynamic adjustment of the serialization order instead of dynamic timestamp assignment as used in OCC-DA. Timestamp intervals allow transactions to be both forward and backward adjusted. As presented earlier, the dynamic timestamp method used in OCC-DA does not allow the transaction to be both forward and backward adjusted. Therefore, the validation method used in OCC-DATI allows more concurrency than the validation method used in OCC-DA.

Additionally, a new dynamic adjustment of the serialization method is proposed, called *deferred dynamic adjustment of serialization order.* In the deferred dynamic adjustment of serialization order all adjustments of a timestamp interval are done to temporal variables. The timestamp intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If a validating transaction is aborted no adjustments are done. Adjustment of the conflicting transaction would be unnecessary since no conflict is present in the history after the abortion of the validating transaction. Unnecessary adjustments may later cause unnecessary restarts. Because the database contains only values from committed transactions, no other transaction can see changes made by the aborted transaction. Therefore, this method avoids cascading aborts and is recoverable.

The OCC-DATI method resolves conflicts using the timestamp intervals [62] of the transactions. Every transaction must be executed within a specific time interval. When an access conflict occurs, it is resolved using the read and write sets of the conflicting transactions together with the allocated time interval. The timestamp interval is adjusted when a transaction validates. In OCC-DATI every transaction is assigned a timestamp interval (TI). At the start of the transaction, the timestamp interval of the transaction is initialized as $[0, \infty[$, i.e., the entire range of timestamp space. This timestamp interval is used to record a temporary serialization order during the validation of the transaction.

At the beginning of the validation (Algorithm 4.1), the final timestamp of the validating transaction $TS(T_v)$ is determined from the timestamp interval allocated to the transaction T_v. The timestamp intervals of all other concurrently running and conflicting transactions must be adjusted to reflect the serialization order. The final validation timestamp $TS(T_v)$ of the validating transaction T_v is set to be the current timestamp, if it belongs to the timestamp interval $TI(T_v)$, otherwise $TS(T_v)$ is set to be the maximum value of $TI(T_v)$.

The adjustment of timestamp intervals iterates through the read set (RS) and write set (WS) of the validating transaction. First it is checked that the validating transaction has read from the committed transactions. This is done by checking the object's read and write timestamp. These values are fetched when the first read and write to the current object is made. Then the set of active conflicting transactions is iterated. When access has been made to the same objects both in the validating transaction and in the active transaction, the temporal time interval of the active transaction is adjusted. Thus, the deferred dynamic adjustment of the serialization order is used.

```
occdati_validate($T_v$)
{
  // Select final timestamp for the transaction
  $TS(T_v) = min(validation\_time, max(TI(T_v)))$;
  // Iterate for all objects read/written
  for ( $\forall D_i \in (RS(T_v) \cup WS(T_v))$ )
  {
    if ($D_i \in RS(T_v)$) // read from committed transactions
      $TI(T_v) = TI(T_v) \cap [WTS(D_i), \infty[$ ;
    if ($D_i \in WS(T_v)$) // write after committed transactions
      $TI(T_v) = TI(T_v) \cap [WTS(D_i), \infty[ \cap [RTS(D_i), \infty[$ ;
    // if timestamp interval is empty, then restart the validating transaction
    if ($TI(T_v) == []$) restart($T_v$);

    // Conflict checking and timestamp interval calculation
    for ( $\forall T_a \in active\_conflicting\_transactions()$ )
    {
      if ($D_i \in (RS(T_v) \cap WS(T_a))$)
        forward_adjustment($T_a, T_v, adjusted$);

      if ($D_i \in (WS(T_v) \cap RS(T_a))$)
        backward_adjustment($T_a, T_v, adjusted$);

      if ($D_i \in (WS(T_v) \cap WS(T_a))$)
        forward_adjustment($T_a, T_v, adjusted$);
    }
  }

  // Adjust conflicting transactions
  for ( $\forall T_a \in adjusted$)
  {
    $TI(T_a) = adjusted.pop(T_a)$;

    if ($TI(T_a) == []$) restart($T_a$);
  }

  // Update object timestamps
  for ( $\forall D_i \in (RS(T_v) \cup WS(T_v))$ )
  {
    if ($D_i \in RS(T_v)$)
      $RTS(D_i) = max(RTS(D_i), TS(T_v))$;
    if ($D_i \in WS(T_v)$)
      $WTS(D_i) = max(WTS(D_i), TS(T_v))$;
  }

  commit $T_v$ to database;
}
```

Algorithm 4.1: Validation algorithm for the OCC-DATI.

Identities of adjusted transactions are inserted to *adjusted* data structure. Time intervals of all conflicting active transactions are adjusted after the validating transaction is guaranteed to commit. If the validating transaction is aborted no adjustments are done. Non-serializable execution is detected when the timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted.

Finally, the current read and write timestamps of the accessed objects are updated and changes to the database are committed. This process is in the critical section.

Algorithm 4.2 shows a implementation of the dynamic adjustment of serialization order using timestamp intervals.

```
forward_adjustment($T_a$, $T_v$, $adjusted$)
{
  // Find the current value of the timestamp interval
   if ($T_a \in adjusted$)
     $TI = adjusted.pop(T_a)$;
   else
     $TI = TI(T_a)$;

  // Forward adjustment
  $TI = TI \cap [TS(T_v)+1, \infty[$ ;
  // Store the current value of the timestamp interval
  $adjusted.push(\{(T_a, TI)\})$;
}

backward_adjustment($T_a$, $T_v$, $adjusted$)
{
  // Find the current value of the timestamp interval
   if ($T_a \in adjusted$)
     $TI = adjusted.pop(T_a)$;
   else
     $TI = TI(T_a)$;

  // Backward adjustment
  $TI = TI \cap [0, TS(T_v)-1]$ ;
  // Store the current value of the timestamp interval
  $adjusted.push(\{(T_a, TI)\})$;
}
```

Algorithm 4.2: Backward and Forward adjustment algorithms for the OCC-DATI.

OCC-DATI offers greater changes to successfully validate transactions resulting in both less waste of resources and a smaller number of restarts. This is because OCC-TI and OCC-DA both use dynamic adjustment but they make unnecessary adjustments when the validating transaction is aborted. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines.

With the new method, the number of transaction restarts is smaller than with OCC-BC, OPT-WAIT, or WAIT-50 [28, 29, 33], because the serialization order of the conflicting transactions is adjusted dynamically. The read and write set of the validating transaction is iterated only twice in the OCC-DATI. In contrast, the read set is iterated three times and the write set four times in OCC-DA in the worst case. Therefore, the overhead for supporting dynamic adjustment is much smaller in OCC-DATI than the one in OCC-DA [53].

Firstly, let us consider Example 3.3 as in Section 3.6, which was used to show unnecessary restart problem in the OCC-TI.

Example 4.1 Consider transactions T_1, T_2, and history H_1:
T_1: $r_1[x]w_1[x]c_1$
T_2: $r_2[x]w_2[y]c_2$
H_1: $r_1[x]r_2[x]w_1[x]c_1$.

In this example T_1 reaches the validation phase first and has a write-read conflict with T_2. Therefore, T_2 must precede T_1 in the serialization history in order to avoid an unnecessary restart. Let $RTS(x)$, $WTS(x)$, $RTS(y)$, and $WTS(y)$ be initialized as 100. Assume that transaction T_1 arrives at its validation phase at time 1000. The OCC-DATI algorithm sets $TS(T_1)$ as 1000. The validating transaction T_1 is first forward adjusted to $[1000, \infty[$. Transaction T_2 has read object x. Therefore T_2's time interval is adjusted to $[0, 999]$ using backward adjustment. Transaction T_1 updates object x timestamps and commits. Thus, the OCC-DATI algorithm produces a serializable history as well as avoiding the unnecessary restart problem found in OCC-TI. □

Secondly, let us consider Example 3.1 as in Section 3.2, which was used to show unnecessary restart problem in the OCC-FV.

Example 4.2 Consider the following transactions T_6, T_7 and history H_3:
T_6: $r_6[x]$ c_6
T_7: $w_7[x]$ c_7
H_4: $r_6[x]$ $w_7[x]$ c_7 c_6

All operations before the first commit operation are executed successfully. Let $RTS(x)$, $WTS(x)$ be initialized as 100. Let us consider the execution of the c_7 operation. Assume that transaction T_7 arrives to its validation phase at time 600. Therefore, $TS(T_7) = 600$. Because $X \in WS(T_7)$, the validating transaction T_7 is forward adjusted to be $TI(T_2) = TI(T_2) \cap [WTS(x), \infty[\cap = [100, \infty[$. Because $x \in WS(T_7) \cap RS(T_6)$, the active transaction T_6 is backward adjusted to be $TI(T_6) = TI(T_6) \cap [0, TS(T_7) - 1] = [0, 599]$. The validation of the transaction T_7 is completed and the write set of the validating transaction (i.e. $WS(T_7)$) is committed to the database. Let us consider the execution of the c_6 operation. Assume that transaction T_6 arrives to its validation phase at time 700. Therefore, $TS(T_1) = 599$. Because $X \in RS(T_6)$, the validating transaction T_6 is forward adjusted to be $TI(T_6) = TI(T_6) \cap [WTS(x), \infty[\cap[RTS(x), \infty[= [100, 599]$. No other conflicts are detected and the validation of the transaction T_6 is completed and the write set of the validating transaction (i.e. $WS(T_6)$) is committed to the database. $\square$

Previous examples show that the OCC-DATI method avoids the unnecessary restart problems found in OCC-TI and OCC-FV. But these examples do not demonstrate that the proposed OCC-DATI method produces only serializable histories. Having described the basic concepts and the algorithm, now the correctness of the algorithm is proven. To prove that a history H produced by OCC-DATI is serializable, it must be proven that the serialization graph for H, denoted by $SG(H)$, is acyclic [7]. Therefore, the following Lemma demonstrates that if there is conflict between the validating transaction and the active transaction then there is a total order between these transactions. This total order is set to the final timestamp of the transactions, i.e. $TS(T_i)$.

Lemma 4.3 Let T_1 and T_2 be transactions in a history H produced by the OCC-DATI algorithm and $SG(H)$ serialization graph. If there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $TS(T_1) < TS(T_2)$.

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

1. $r_1[x] \rightarrow w_2[x]$: This case means that T_1 reads old value of the data item x since $r_1[x]$ is not affected by $w_2[x]$. If the transaction T_1 arrives to the validation before the transaction T_2, OCC-DATI adjusts T_2 by the forward adjustment. Thus,$TS(T_2) = min(max([TS(T_1) + 1, \infty[), \text{current time}) > TS(T_1)$. If the transaction T_2 arrives to the validation before the transaction T_1, OCC-DATI adjust T_1 by the backward adjustment. Thus, $TS(T_1) = min(max([0, TS(T_2) - 1]), \text{current time}) = TS(T_2) - 1 < TS(T_2)$.

Therefore, $TS(T_1) < TS(T_2)$. Both transactions can not enter to the validation in the same time, because the validation is in the critical section.

2. $w_1[x] \rightarrow r_2[x]$: This case means that the write phase of T_1 finishes before $r_2[x]$ executes in T_2's read phase. This is because, T_2 has seen the new value of the data item x and the write phase is in the critical section and done in a atomic way. For $r_2[x]$, OCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

3. $w_1[x] \rightarrow w_2[x]$: This case means that the write phase of T_1 finishes before $w_2[x]$ executes in T_2's write phase. This is because the write phase is in the critical section and done in a atomic way. For $w_2[x]$, OCC-DATI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. Thus, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.

□

To show that every history generated by the OCC-DATI algorithm is serializable, it is assumed that the algorithm will produce a cycle in the serialization graph. This case is shown to cause contradiction in the following theorem.

Theorem 4.4 Every history generated by the OCC-DATI algorithm is serializable.

Proof: Let H denote any history generated by the OCC-DATI algorithm and $SG(H)$ its serialization graph. Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 4.3 $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. By simple induction this leads to $TS(T_1) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the OCC-DATI algorithm produces only serializable histories. □

Chapter 5

Attributes in Conflict Resolution

Priority-cognizant concurrency control methods based on the optimistic methods have not been widely studied. Because time cognizance is important to offer better support for timing constraints as well as predictability, the major concern in designing real-time optimistic concurrency control methods is to incorporate information about the timing constraints of transactions for conflict resolution.

Firstly, this chapter extends the original OCC-TI conflict resolution method with transaction deadline-driven priority information. In conflict resolution higher priority transactions are favored. This work is based on earlier work presented in [70].

Secondly, this chapter extends the OCC-DATI conflict resolution method with deadline-driven priority information. In conflict resolution higher deadline-driven priority transactions are favored. This work is based on earlier work presented in [75].

Thirdly, this chapter further extends the OCC-DATI conflict resolution method with an even more pessimistic view. In conflict resolution a lower deadline-driven priority transaction is **always** restarted if it conflicts with the higher deadline-driven priority transaction. This work is based on earlier work presented in [76].

The idea behind these extensions is to see whether priority or criticality of the transaction can be used in conflict resolution and what kind of effects it has on overall performance and performance when only high priority or critical transactions are examined.

5.1 Revised OCC-TI

Firstly, as showed in section 3.4 the original OCC-TI does not fully solve the unnecessary restart problem. Therefore, in this section a solution to this problem is proposed.

Secondly, there are no real-time properties in the original OCC-TI algorithm. An extension to the OCC-TI algorithm is proposed in this section to solve these problems. This thesis includes the following extensions to the OCC-TI:

1. Rollbackable Dynamic Adjustment of Serialization Order
2. Prioritized Dynamic Adjustment of Serialization Order

In the first extension the undoing of dynamic adjustments done to an active transaction when the adjustment was unnecessary will be attempted. For example, if the validating transaction aborts then all dynamic adjustment to other conflicting active transactions were unnecessary. In the second extension priorities are taken into account before using dynamic adjustment.

5.1.1 Rollbackable Dynamic Adjustment

Let $TI(T_i)$ denote the timestamp interval for transaction T_i and let $RTI_n(T_i), n = 1, ..k, k \in \mathbb{N}$ denote the n:th removed timestamp interval from transaction T_i. One modification to the timestamp interval can be rollbacked if the current timestamp interval and the removed timestamp interval are adjacent. Formally,

Definition 5.1 The timestamp interval $TI(T_i)$ of the transaction T_i is **rollbackable** with the removed timestamp interval $RTI_n(T_i), n = k, .., 1, k \in \mathbb{N}$ if and only if:

$$\forall x \forall y ((x \in TI(T_i) \land y \in RTI_n(T_i)) \land$$
$$\nexists z (z \in ([0, \infty[\setminus (TI(T_i) \cup RTI_n(T_i)) \ \land \ (x < z < y) \ \lor (y < z < x))). \qquad \square$$

Example 5.2 Let $TI(T_1) = [100, 1000]$, $RTI_1(T_1) = [0, 100]$, and $RTI_2(T_1) = [1002, 2000]$. Using definition 5.1, the first removed timestamp interval to be checked for rollbacking is $RTI_2(T_1)$. If it is set $x = 1000$ and $y = 1002$ then clearly $\exists z (z \in [o, \infty[\ \land \ (1000 < z < 1002))$ e.g. $z = 1001$. Thus the removed timestamp interval $RTI_2(T_1)$ cannot be rollbacked. When checking $RTI_1(T_1)$ for rollbacking we can see that definition 5.1 holds and we can rollback the removed timestamp interval. $\square$

The next definition 5.3 shows how dynamically adjusted timestamp intervals can be rollbacked.

Definition 5.3 The timestamp interval $TI(T_i)$ of transaction T_i is **rollbacked** with removed timestamp interval $RTI_n(T_i), n \in \mathbb{N}$ calculating the new timestamp interval :

$$TI(T_i) = TI(T_i) \cup RTI_n(T_i).$$

□

Example 5.4 Let $TI(T_1) = [100, 1000]$ and $RTI_1(T_1) = [0, 100]$. Then rollbacking is done with

$$TI(T_1) = [100, 1000] \cup [0, 100] = [0, 1000]\ .$$

□

Removed timestamp intervals should be rollbacked in descending order thus $\forall n(n = k, .., 1, k \in \mathbb{N})$. This ensures that the resulting timestamp interval is as big as possible. The following example shows what happens if rollbacking is not done in descending order.

Example 5.5 Let $TI(T_1) = [200, 1000]$, $RTI_1(T_1) = [0, 100]$, and $RTI_2(T_1) = [100, 200]$. These removed timestamp intervals are rollbackable using definition 5.1. Using definition 5.3 in ascending order, the result would be:

$$TI(T_1) = [200, 1000] \cup [0, 100] = [200, 1000] \cup [100, 200] = [100, 1000].$$

But if rollbacking is done in descending order the result would be a larger timestamp interval:

$$TI(T_1) = [200, 1000] \cup [100, 200] = [100, 1000] \cup [0, 100] = [0, 1000].$$

□

This method, while important, needs additional data structure to store removed timestamp intervals and in case of rollbacking quite expensive iteration of the data structure holding the removed timestamp intervals. Therefore, only two timestamp intervals are actually stored. The current timestamp interval value of the active transaction and temporal timestamp interval value of the active transaction during the validation phase of another transaction. The temporal timestamp interval value is copied to the actual value when the validating transaction is guaranteed to commit. Thus, no unnecessary rollbacking is done.

5.1.2 Prioritized Dynamic Adjustment

In this section a priority-dependent extension to dynamic adjustment of the serialization order is presented. In real-time database systems, the conflict resolution should take into account the priority of the transactions. This is especially true in the case of heterogeneous transactions (i.e. transactions with different importance). Some transactions are more important or valuable than others. The dynamic adjustment in the validation phase should be done in favor of a higher priority transaction. Here, a method will be presented that tries to make more room for the higher priority transaction to commit in its timestamp interval. This offers the high priority transaction better chances to commit before its deadline and meeting timing constraints.

A *Prioritized Dynamic Adjustment of the Serialization Order* (PDASO) implemented with timestamp intervals creates partial order between transactions based on conflicts and priorities.

If a validating transaction has higher priority than an active conflicting transaction, forward adjustment is correct. If the validating transaction has lower priority than the active conflicting transaction, the higher priority transaction should be favored. This is supported by reducing the timestamp interval of the validating transaction and selecting a new final timestamp earlier in the timestamp interval. Normally the current time or maximum value from the timestamp interval is selected. But now the middle point is selected. This offers greater changes for a higher priority transaction to commit in its timestamp interval. If the middle point cannot be selected, the validating transaction is restarted. This is wasted execution, but it is required to ensure the execution of the transaction of higher priority.

If a validating transaction has higher priority than an active conflicting transaction, backward adjustment is correct. If the validating transaction has lower priority than the active conflicting transaction, then backward adjustment is done if the active transaction is not aborted in the backward adjustment. Otherwise, the validating transaction is restarted. This is wasted execution, but it is required to ensure the execution of the transaction of higher priority.

However, in backward adjustment the validating transaction cannot be moved to the future to obtain more space for the higher priority transaction. One can only check if the timestamp interval of the lower priority transaction would become empty. In the forward ordering the final timestamp can be moved backward if there is space in the timestamp interval of the validating transaction. Again, it is checked whether the timestamp interval of the higher priority transaction would shut out. Aborting the validating transaction when the timestamp interval of the higher priority transaction shuts out has been chosen. Thus, this algorithm favors the higher priority transactions and might waste resources aborting near to complete transactions.

Example 5.6 Let $TI(T_1) = [100, 1000]$, $TS(T_1) = 1000$, and $TI(T_2) = [0, \infty[$. Let $pri(T_1) < pri(T_2)$. Assume that there is a read-write conflict between the transactions in the history. Dynamic adjustment solves this conflict by forward adjusting the active transaction T_2:

$$TS(T_1) = (100 + 1000)/2 = 550$$
$$TI(T_2) = [0, \infty[\ \cap\ [550, \infty] = [550, \infty[e.$$

□

Example 5.7 Let $TI(T_1) = [100, 1000]$, $TS(T_1) = 1000$, and $TI(T_2) = [1200, \infty[$. Let $pri(T_1) < pri(T_2)$. Assume that there is a write-read conflict between the transactions. Using the backward adjustment validating transaction must be aborted because:

$$TI(T_2) = [1200, \infty[\ \cap\ [0, 1000] = \emptyset.$$

□

5.1.3 Revised OCC-TI Algorithm

In this section the validation algorithm for extended OCC-TI is presented. We will call this method OCC-PTI (Optimistic Concurrency Control with Prioritized Timestamp Intervals). OCC-TI is extended with a new final timestamp selection method and priority-dependent conflict resolution. Final (commit) timestamp $TS(T_v)$ should be selected in such a way that room is left for backward adjustment. A new validation algorithm is proposed where the commit timestamp is selected differently. In the revised validation algorithm for OCC-TI (Algorithm 5.1) the final timestamp $TS(T_v)$ is set as the validation time if it belongs to the time interval of T_v or maximum value from the time interval otherwise. Additionally, the original OCC-TI is extended to use prioritized dynamic adjustment of the serialization order.

```
occti_validate($T_v$) {
    /* Select final (commit) timestamp */
    if ($validation\_time \in TI(T_v)$)
        $TS(T_v) = validation\_time$;
    else $TS(T_v) = max(TI(T_v))$;

    /* Iterate read and write sets of the validating transaction */
    for $\forall D_i \in ( RS(T_v) \cup WS(T_v))${

        /* Iterate conflicting active transactions */
        for $\forall T_a \in active\_conflicting\_transactions()$ {
            if ($D_i \in ( RS(T_v) \cap WS(T_a))$)
                forward_adjustment($T_a, T_v, adjusted$);

            if ($D_i \in ( WS(T_v) \cap RS(T_a))$)
                backward_adjustment($T_a, T_v, adjusted$);

            if ($D_i \in ( WS(T_v) \cap WS(T_a))$)
                forward_adjustment($T_a, T_v, adjusted$);

            if ($TI(T_a) = []$) restart($T_a$);
        }

        // Adjust conflicting transactions
        for ( $\forall T_a \in adjusted$)
        {
            $TI(T_a) = adjusted.pop(T_a)$;

            if ($T_a.backward == true$ )
                $TI(T_a) = TI(T_a) \cap [0, TS(T_v) - 1]$;

            if ($TI(T_a) == []$) restart($T_a$);
        }

        /* Update RTS and WTS timestamps */
        if ($D_i \in RS(T_v)$)
            $RTS(D_i) = max(RTS(D_i), TS(T_v))$;

        if ($D_i \in WS(T_v)$)
            $WTS(D_i) = max(WTS(D_i), TS(T_v))$;
    }
    commit $WS(T_v)$ to database;
}
```

Algorithm 5.1: Validation algorithm for the OCC-PTI.

The adjustment of timestamp intervals (TI) iterates through the read set (RS) and write set (WS) of the validating transaction (T_v). First we check that the validating transaction has read from the committed transactions. This is done by checking the object's read timestamp (RTS) and write timestamp (WTS). These values are fetched when the first access to the current object is made. Then the algorithm iterates the set of active conflicting transactions. When access has been made to the same objects both in the validating transaction and in the active transaction (T_a), the timestamp interval (TI) of the active transaction is adjusted. Non-serializable execution is detected when the timestamp interval of an active transaction becomes empty. If the timestamp interval is empty the transaction is restarted. Finally, current read timestamps and write timestamps of accessed objects are updated and changes to the database are committed.

Algorithm 5.2 presents forward and backward adjustment algorithms for dynamic adjustment of the serialization order using timestamp intervals and priorities.

Backward and Forward adjustment algorithms create order between conflicting transaction timestamp intervals. The final (commit) timestamp is selected from the remaining timestamp interval of the validating transaction. Therefore, the final timestamps of the transactions create partial order between transactions.

Compared to method presented in [16], OCC-PTI uses an different final timestamp selection method and priorities are used in every conflict resolution case.

```
forward_adjustment($T_a, T_v, adjusted$) {
    if ($T_a \in adjusted$)
      $TI = adjusted.pop(T_a)$;
    else
      $TI = TI(T_a)$;

    if ( $priority(T_v) >= priority(T_a)$)
        $TI = TI \cap [TS(T_v), \infty]$;
    else {
         if ((($min(TI(T_v)) + TS(T_v)) / 2) \in TI(T_v)$) {
             $TS(T_v) = (min(TI(T_v)) + TS(T_v)) / 2$;

              if ( $TS(T_v) > max(TI)$) restart($T_v$);

             $TI = TI \cap [TS(T_v), \infty]$;
        }
         else {
              if ( $TS(T_v) > max(TI)$) restart($T_v$);

             $TI = TI \cap [TS(T_v), \infty]$;
        }
   }

   $adjusted.push(\{(T_a, TI)\})$;

}

backward_adjustment($T_a, T_v, adjusted$) {
    if ($T_a \in adjusted$)
      $TI = adjusted.pop(T_a)$;
    else
      $TI = TI(T_a)$;

    if ( $priority(T_v) >= priority(T_a)$)
        $T_a.backward = true$ ;
    else {
         if ( $TS(T_v) - 1 < min(TI)$) restart($T_v$);

        $T_a.backward = true$ ;
   }

   $adjusted.push(\{(T_a, TI)\})$;

}
```

Algorithm 5.2: Backward and Forward adjustment for the OCC-PTI.

Let us consider the same example history as in Example 3.3 in Section

3.6, which caused unnecessary restart in the original OCC-TI. Using the same example, it is shown here how the OCC-PTI produces a serializable history and avoids unnecessary restart.

Example 5.8 Let $RTS(x)$ and $WTS(x)$ be initialized as 100. Consider transactions T_1, T_2, and history H_1, where $pri(T_1) = pri(T_2)$:
T_1: $r_1[x]w_1[x]c_1$
T_2: $r_2[x]w_2[y]c_2$
H_1: $r_1[x]r_2[x]pw_1[x]c_1$.

Transaction T_1 executes $r_1[x]$, which causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [0, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_2 then executes a read operation on the same object, which causes the timestamp interval of the transaction to be forward adjusted similarly. Transaction T_1 then executes $pw_1[x]$, which causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [100, \infty[\cap [100, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_1 starts the validation at time 1000, and the final (commit) timestamp is selected to be $TS(T_1) = validation\_time = 1000$. Because there is one read-write conflict between the validating transaction T_1 and the active transaction T_2, the timestamp interval of the active transaction must be adjusted: $TI(T_1) = [100, \infty[\cap [0, 999] = [100, 999]$. Thus the timestamp interval is not empty, and revised OCC-TI has avoided unnecessary restart. Both transactions commit successfully. History H_1 is acyclic, that is, serializable. Therefore, the proposed OCC-PTI produces serializable histories (this can be proven with similar construct as was used in the OCC-DATI method) and avoids the unnecessary restart problem found in the original OCC-TI algorithm □

5.2 OCC-PDATI

In real-time database systems, the conflict resolution should take into account the criticality of the transactions. This is especially true in the case of heterogeneous transactions. Some transactions are more critical or valuable than others. Therefore, the goal of any real-time system should be to maximize the value or criticalness of the completed transactions. In most of the previous approaches the value or the criticalness of a transaction has been equalled to the scheduling priority of a transaction. Unfortunately, that is a very serious restriction if the target is to maximize the value or the criticalness of the completed transactions.

In this section we will use deadline-driven conflict priority. As mentioned in Section 2.2 deadline-driven conflict priority is based on the deadline and the criticality of the transaction. We will call this deadline-driven conflict priority as conflict priority in this section.

When data conflicts between the validating transaction and active transaction are detected in the validation phase, there is no need to restart conflicting active transactions immediately. Instead, a serialization order can be dynamically defined. However, if a higher conflict priority transaction is to be aborted because of conflict to lower conflict priority transaction, then the transaction of lower conflict priority should be restarted.

This section proposes an optimistic concurrency control method called OCC-PDATI (Optimistic Concurrency Control using Priority of the Transaction and Dynamic Adjustment of Serialization Order). OCC-PDATI is based on forward validation [26] and the earlier optimistic method OCC-DATI [74]. The difference is in the conflict resolution. The conflict resolution of OCC-PDATI uses conflict priority of the transaction found in transaction object attributes. This section outlines new parts of OCC-PDATI when compared to OCC-DATI.

A higher conflict priority transaction should not be restarted because of conflict with a transaction of lower conflict priority. Greater changes for a transaction to complete before its deadline should be offered. Therefore, if dynamic adjustment of the serialization order would cause the higher conflict priority transaction to be restarted, a conflicting active transaction of lower conflict priority is restarted. This is a wasted execution, but it is required to ensure the execution of the critical transactions.

The method must ensure serializable (or another correct order of) execution. Database operations of an active transaction is not known beforehand. These future reads or writes may lead to an empty timestamp interval if backward adjustment is used. Therefore, critical transactions should not be backward adjusted, but conflicting active transactions having lower conflict priority should be restarted. This is wasted execution and unnecessary restart, which must be acceptable when critical transactions are favored. Backward adjustment of a critical transaction is possible if the transaction is not to be restarted due to an empty timestamp. This, however, implies that the critical transaction should not be allowed to read or write new data objects that belong to a new database state after a backward adjustment. In other words, future read or write operations could reduce the timestamp interval of the transaction to empty.

Algorithm 5.3 depicts an implementation of a deferred dynamic adjustment of the serialization order using timestamp intervals and information about the criticality of the transactions.

Because the OCC-PDATI uses the same conflict detection method as the OCC-DATI, the same conflicts are found. Conflict resolution in the OCC-PDATI is based on same principles as in the OCC-DATI but OCC-PDATI restricts possible histories so that higher conflict priority transactions are favored in case of conflict. Therefore, OCC-PDATI produces serializable histories which youd be prooven with very similar proof as in the OCC-DATI's case. Because transactions see only values from committed transactions, OCC-PDATI avoids cascading aborts and is recoverable. Therefore, OCC-PDATI produces only strict serializable histories.

```
forward_adjustment($T_a$, $T_v$, $adjusted$)
{
   if ($T_a \in adjusted$)
     $TI = adjusted.pop(T_a)$;
   else
     $TI = TI(T_a)$;

  $TI = TI(T_a) \cap [TS(T_v) + 1, \infty[$;

   if ( $cpriority(T_v) < cpriority(T_a)$ )
     if ( $TI == \emptyset$ )
       $restart(T_v)$; /* Validation ends here /*

  $adjusted.push(\{(T_a, TI)\})$;
}

backward_adjustment($T_a$, $T_v$, $adjusted$)
{
   if ($T_a \in adjusted$)
     $TI = adjusted.pop(T_a)$;
   else
     $TI = TI(T_a)$;

  $TI = TI(T_a) \cap [0, TS(T_v) - 1]$

   if ( $cpriority(T_v) < cpriority(T_a)$ )
       $restart(T_v)$; /* Validation ends here /*

  $adjusted.push(\{(T_a, TI)\})$;
}
```

Algorithm 5.3: Backward and Forward adjustment for the OCC-PDATI.

5.3 OCC-RTDATI

This section is based on earlier results presented in [76]. In real-time database systems, the conflict resolution of the transactions should be based on the timing constraints of the transactions. Therefore, an optimistic concurrency control method has been developed where the decision about which transaction is to be restarted is based on the transaction priority.

In this section we will use deadline-driven conflict priority. As mentioned in Section 2.2 deadline-driven conflict priority is based on the deadline and the criticality of the transaction. We will call this deadline-driven conflict priority as conflict priority in this section.

This section proposes an optimistic concurrency control method called OCC-RTDATI. OCC-RTDATI is based on forward validation [26] and the earlier optimistic method OCC-DATI [74]. The validation protocol is the same in OCC-DATI and OCC-RTDATI. The difference is in the conflict resolution. This section outlines new parts of OCC-RTDATI when compared to OCC-DATI.

The order of the conflicting transactions should be based on the conflict priority of the transaction. A higher conflict priority transaction should precede a transaction of lower conflict priority in the history. Therefore, the higher conflict priority transaction should not be forward adjusted after a transaction of lower conflict priority. Thus, if this is the case a transaction of lower conflict priority is restarted. This is a wasted execution, but it is required to ensure the execution of the critical transaction.

Critical transactions should not be backward adjusted; instead, conflicting transactions having lower conflict priority should be restarted. This is wasted execution and unnecessary restart, which must be acceptable when critical transactions are favored.

Algorithm 5.4 depicts an implementation outline of the conflict resolution for the OCC-RTDATI.

Because the OCC-RTDATI uses the same conflict detection method as the OCC-DATI, the same conflicts are found. Conflict resolution in the OCC-RTDATI is based on same principles as in the OCC-DATI but OCC-RTDATI restricts possible histories so that higher conflict priority transactions are favored in case of conflict. Therefore, OCC-RTDATI produces serializable histories which youd be prooven with very similar proof as in the OCC-DATI's case. Because transactions see only values from committed transactions, OCC-RTDATI avoids cascading aborts and is recoverable. Therefore, OCC-RTDATI produces only strict serializable histories.

```
forward_adjustment($T_a, T_v, adjusted$)
{
    if ( $cpriority(T_v) < cpriority(T_a)$ )
        $restart(T_v)$; /* Validation ends here /*

    if ($T_a \in adjusted$)
      $TI = adjusted.pop(T_a)$;
    else
      $TI = TI(T_a)$;

   $TI = TI(T_a) \cap [TS(T_v)+1, \infty[$;
   $adjusted.push(\{(T_a, TI)\})$;
}

backward_adjustment($T_a, T_v, adjusted$)
{
    if ( $cpriority(T_v) < cpriority(T_a)$ )
        $restart(T_v)$; /* Validation ends here /*
    if ( $cpriority(T_v) > cpriority(T_a)$ )
        $restart(T_a)$; $return$;  /* Restart conflicting */

    if ($T_a \in adjusted$)
      $TI = adjusted.pop(T_a)$;
    else $TI = TI(T_a)$;

   $TI = TI(T_a) \cap [0, TS(T_v)-1]$
   $adjusted.push(\{(T_a, TI)\})$;
}
```

Algorithm 5.4: Backward and Forward adjustment algorithms for the OCC-RTDATI.

Chapter 6

Adaptive Conflict Resolution

Many methods proposed earlier do not include any real-time properties. Therefore, these methods are too "fair". A characteristic of most real-time scheduling algorithms is the use of priority-based scheduling [1]. Here transactions are assigned 'priorities', which are implicit or explicit functions of their deadlines or *criticality* or both. The criticality of a transaction is an indication of its level of importance. However, these two requirements sometimes conflict with each other. That is, transactions with very short deadlines might not be very critical, and vice versa [8].

In real-time systems transactions are scheduled according to their priorities [83]. Therefore, high priority transactions are executed before lower priority transactions. This is true only if the high priority transaction has some database operation ready for execution. If no operation from the higher priority transaction is ready for execution, then the operation from the lower priority transaction is allowed to execute its database operation. Therefore, the operation of the higher priority transaction may conflict with the already executed operation of the lower priority transaction. In traditional methods the higher priority transaction must wait for the release of resources. This is the priority inversion problem. Therefore, data conflicts in the concurrency control should also be based on transaction priorities or criticality or both.

Therefore, the conflict priority of the transaction is used in place of the deadline in choosing the appropriate conflict resolution method. This avoids the dilemma of the priority-based conflict resolution, yet integrates criticality and deadline so that, not only do the more critical transactions meet their deadlines, but the overall goal is to maximize the net worth (critical transactions worth more to system) of the executed transactions to the system. This work is based on work presented in [71].

Each instance of the transaction object has the attributes *priority*, *dead-*

line, *criticality*, and *conflict priority*. The developer assigns a value for the deadline based on the estimate or value measured through experiments of worst case execution time. The priority attribute is assigned by the real-time scheduler (EDF) and is based on the deadline and arrival time. The criticality attribute is assigned by the developer and is static and the same goes for all instances of the same transaction class. From criticality and deadline an deadline driven conflict priority is calculated. We have used the following values coded as numbers:

- **Normal**: The transaction is not essential but should be completed if the execution history is serializable. For this transaction class we use basic conflict resolution. This method is same as in OCC-DATI (See Chapter 4).
- **Medium**: The transaction is important and should not be restarted if there is a data conflict with the transaction with normal conflict priority. For this transaction class we use conflict resolution where a lower conflict priority transaction is restarted if an active conflicting higher conflict priority transaction would be restarted because of a data conflict. This method is same as in OCC-PDATI (See Section 5.2).
- **Critical**: The transaction is critical and should not be restarted even if there is data conflict with the transaction with normal or medium conflict priority. For this transaction class we use conflict resolution where a lower conflict priority transaction is *always* restarted if there is an active conflicting higher conflict priority transaction. This method is same as in OCC-RTDATI (See Section 5.3).

Let us assume that the conflict detection algorithm (see Algorithm 4.1) has found that a validating transaction T_v and active transaction T_1 are conflicting. Assume that the validating transaction has read the data item X and the active transaction has written the data item X. Assume that $cpriority(T_v) = cpriority(T_1) = normal$. In this case the conflict is resolved using normal forward adjustment.

Assume that the conflict detection algorithm finds another conflict between the validating transaction and an active transaction T_2. Assume that the validating transaction has read the data item X and the active transaction has written the data item X. Assume that $cpriority(T_v) = normal < cpriority(T_2) = medium$. In this case the conflict is resolved according to the medium forward adjustment. Therefore, the conflict resolution method is selected according to the criticality of the higher conflict priority transaction (see Figure 6.1).

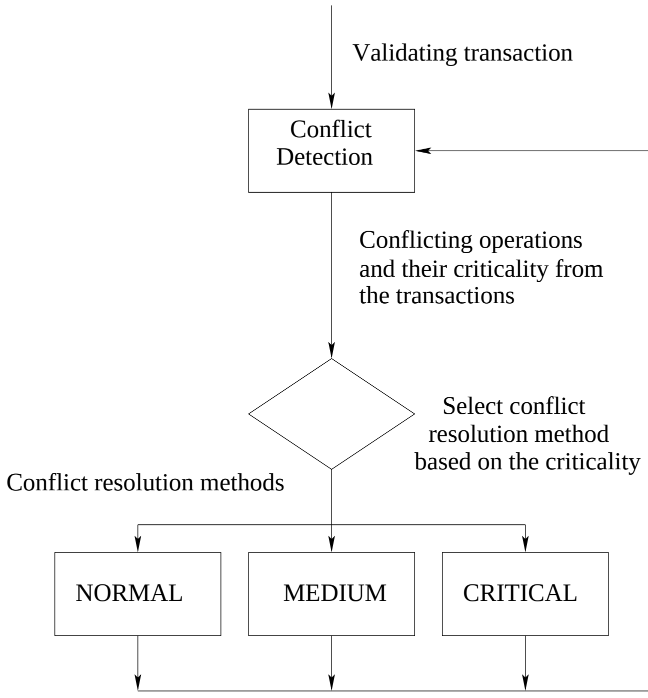

Validating transaction
Conflict Detection
Conflicting operations and their criticality from the transactions
Find the next conflict
Select conflict resolution method based on the criticality
Conflict resolution methods
NORMAL
MEDIUM
CRITICAL

Figure 6.1: Integrated Optimistic Concurrency Control Method.

Assume that the conflict detection algorithm finds another conflict between the validating transaction and an active transaction T_3. Assume that the validating transaction has read the data item X and the active transaction has written the data item X. Assume that $cpriority(T_v) = normal < cpriority(T_3) = critical$. In this case the conflict is resolved according to the critical forward adjustment.

Therefore, the proposed method dynamically adapts to different load situations. This is because the conflict resolution method dynamically selects a resolution method based on the conflict priority of the conflicting transactions. The proposed method offers three different conflict resolution methods. However, inside all the methods the conflict priority of the conflicting transactions are still taken into account. Therefore, the proposed method can offer better changes for the more critical transaction even if both conflicting transactions belong to the same conflict priority level. This is because one conflict priority level can be constructed from a large interval (i.e. the conflict priority boundaries can be any positive integer values).

This method is possible because all integrated methods use precisely the same conflict detection method (i.e. same method as in the OCC-DATI, see Chapter 4). Additionally, all integrated methods basically use the same conflict resolution method. The only difference between different conflict resolution methods is that the medium method and the critical method restart the conflicting lower conflict priority transaction if the validating transaction is more critical.

Finally, we present an integrated and adaptive method for conflict resolution. We will call this method OCC-IDATI. Below we present forward and backward adjustment algorithms, i.e. the conflict resolution method in the OCC-IDATI. In implementation, three different forward and backward adjustment algorithms are integrated to one forward and one backward adjustment algorithm. Similarly, conflict resolution method selection is integrated inside both algorithms (see Algorithm 6.1).

It is easy to show that OCC-IDATI produces serializable histories with similar proof as in the OCC-DATI's case. Because transactions see only values from committed transactions, OCC-IDATI avoids cascading aborts and is recoverable.

```
forward_adjustment($T_a, T_v, adjusted$)
{
  cpriority = max($cpriority(T_a), cpriority(T_v)$);

   if $(T_a \in adjusted)$
     $TI = adjusted.pop(T_a);$
   else
     $TI = TI(T_a);$

   if ( cpriority >= NORMAL_MIN && cpriority <= NORMAL_MAX )
       continue;
   if ( cpriority >= MEDIUM_MIN && cpriority <= MEDIUM_MAX ) {
        if ( $cpriority(T_v) < cpriority(T_a)$ )
          if ( $TI == \emptyset$ )
            $restart(T_v);$ /* Validation ends here /*
  }
   if ( cpriority >= CRITICAL_MIN && cpriority <= CRITICAL_MAX ) {
        if ( $cpriority(T_v) < cpriority(T_a)$ )
            $restart(T_v);$ /* Validation ends here /*
  }

  $TI = TI \cap [TS(T_v) + 1, \infty[$ ;
  $adjusted.push(\{(T_a, TI)\});$
}

backward_adjustment($T_a, T_v, adjusted$)
{
  cpriority = max($cpriority(T_a), cpriority(T_v)$);

   if $(T_a \in adjusted)$
     $TI = adjusted.pop(T_a);$
   else
     $TI = TI(T_a);$

   if ( cpriority >= NORMAL_MIN && cpriority <= NORMAL_MAX )
       continue;
   if ( cpriority >= MEDIUM_MIN && cpriority <= MEDIUM_MAX ) {
        if ( $cpriority(T_v) < cpriority(T_a)$ )
            $restart(T_v);$ /* Validation ends here /*
  }
   if ( cpriority >= CRITICAL_MIN && cpriority <= CRITICAL_MAX ) {
        if ( $cpriority(T_v) < cpriority(T_a)$ )
            $restart(T_v);$ /* Validation ends here /*
        if ( $cpriority(T_v) > cpriority(T_a)$ )
            $restart(T_a);$ $return;$
  }

  $TI = TI \cap [0, TS(T_v) - 1]$ ;
  $adjusted.push(\{(T_a, TI)\});$
}
```

Algorithm 6.1: Backward and Forward adjustment for the OCC-IDATI method.

Chapter 7

Experiments Using RTDB for Telecommunications

In this section we present requirements of telecommunication system and show that real-time databases can answer these requirements. This work is based on earlier work presented in [73]. From these requirements we have developed a real-time database system for telecommunications, the architecture of which is presented. This system is used for evaluating methods presented in this thesis. Additionally, we present a benchmark used in evaluating the proposed methods and the system. This benchmark will be workload used in the experiments. Finally, we present experiments done using proposed optimistic methods presented earlier.

7.1 Requirements

The telecommunication field has different services, which have different database needs. The intelligent network (IN) concept models its services in a traditional fixed-line network. The GSM services are wireless. Telecommunication Management Network (TMN) has its own requirements for the databases. This section concentrates on the requirements for distributed databases. Requirements for these databases are listed in [81, 86, 100].

The Intelligent Network (IN) [40] services do not necessarily require the support from a distributed database. A centralized database is enough to support them. The caller and called profiles can be fetched separately from their own databases at both ends of the call. Even the dialed number services (800 Service) does not require co-operation of multiple database nodes [43, 9]. According to [81], the intelligent network has databases for traffic data, service data and customer data. The requirements analysis shows that transactions in the IN system are commonly small (only a few operations per transaction) and require short response time. This creates problems for most commercial database systems.

The best-known IN architecture is probably the Datacycle architecture by AT&T [10]. It is based on special hardware that allows fast data access. A regular type IN database has been presented by Norwegian Telecom Research [38, 103]. Their architecture is a shared-nothing parallel database where parallel relational database nodes communicate with each other via an ATM network. The traditional IN services can be implemented without transactions accessing multiple databases.

Previous work has proposed that the consistency of some subsets of the telecommunication transactions can be relaxed [81]. However, we recommend using full consistency on all transactions in the telecommunication database because it is easier to support the full consistency with all transactions than the full consistency with a subset of transactions and a relaxed consistency with another subset. Relaxed consistency would require additional semantic information of the transactions. This information can originate only from the application developer. The application developer would need to include additional code in applications to maintain database consistency when integrity constraints are present. This would complicate and lengthen the development time.

7.2 RODAIN Architecture

The Real-Time Object-Oriented Database Architecture for Intelligent Networks (RODAIN) [73] is a real-time, highly-available, main-memory database server. RODAIN supports concurrently running real-time transactions using an optimistic concurrency control protocol with deferred write policy. Real-time transactions are scheduled according to their type, priority, mission criticality, or time criticality. In this section only those parts of RODAIN are discussed which are essential to this thesis and which were used in the experiments.

RODAIN is further divided into a set of subsystems (Figure 7.1) that perform the needed function. The database resides in the subsystem called HOT (Hot Object Storage). Subsystem OCC (Optimistic Concurrency Control Scheduler) takes care of the concurrency control, while subsystem ORD (Object Request Dispatcher) is simply a common entrance point to the core database system. TRP (Transaction processing) executes all transactions.

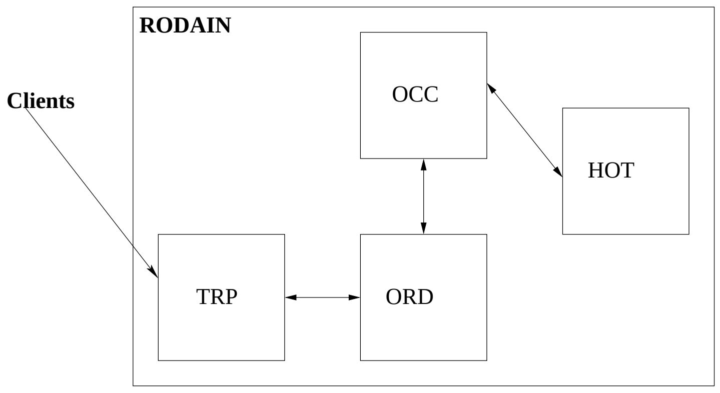


Figure 7.1: Subsystem Interfaces.

The threads in the TRP execute the transactions according to the requests and given parameters. The precoded real-time transactions get all their parameters in the requests and give their answers in the replies. Precoded real-time transactions are executed in the RODAIN server. A transaction starts when the TRP receives a transaction number and arguments to the transaction. TRP makes a feasibility test and if the transaction is found eligible for execution, the TRP assigns a priority to the transaction process. TRP implements the EDF scheduling policy and allocates the transaction process to execute the code of the transaction.

TRP is responsible for checking that transactions are scheduled in a feasible schedule. TRP controls, at the operation level, during the execution to the effect that all constraints are satisfied. TRP terminates successfully executed transactions or terminates rejected, aborted, and restarted transactions if some of the constraints are violated.

When the transaction requires database services, the TRP performs subroutine calls for these services. Complete structures of all accessed objects are known in this level, thus the transaction has free access to all properties of the object. When a transaction process needs access to an object in a database, it sends a request to an Object Request Dispatcher (ORD).

This request is forwarded to Optimistic Concurrency Control Scheduler (OCC). OCC maintains appropriate markings in the readset or in the writeset of the transaction.

At this point of the transaction flow the request can be either a read request or a write request of an object in the Hot Object Storage (HOT). When the request is a read request, the data is fetched from the main memory database (HOT). In the case of a write request no data is accessed.

When a transaction is going to commit, the transaction process is moved into the process group of committing transactions. The difference between a transaction process and a committing transaction is that each committing transaction always has a higher priority than any transaction process has.

The first step in the commit is validation, that is to certify whether or not concurrency conflicts have occurred. The validation is done by an Optimistic Concurrency Control Scheduler (OCC) that receives a validate request. During the validation process the OCC checks whether or not read-write or write-write conflicts exist. If a conflict exists, the conflict is solved either by adjusting the serialization order or by aborting the committing transaction or the conflicting transaction.

After a successful validation, the commit procedure continues. The next step in the commit procedure is to send a write request to the ORD in order to store the written objects permanently into the database. The communication between the Committing Transaction and the ORD is synchronous. The commit ends when all write requests have been processed successfully. The Transaction Process is returned into the pool of free Transaction Processes, and the result sent to the application.

The client requests arrive via TCP/IP over a network directly to the TRP, which contains threads to serve the clients. Each client may use the same connection for multiple transaction requests. No communication during the transaction execution is allowed between the transaction and the calling client. All transactions arrive at the RODAIN through a specific user subsystem. In these tests we have used a special user subsystem that receives the arriving transactions from an off-line generated test file (see Figure 7.2).

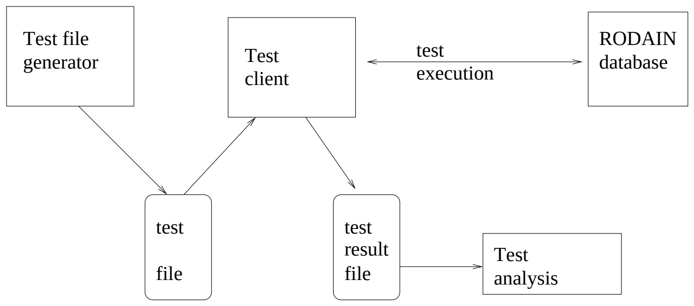


Figure 7.2: Architecture of the RODAIN Database Evaluation System.

Consider now the abstract database model presented in Chapter 2 Section 2.2 and especially modules presented in Figure 2.4. In RODAIN, TRP

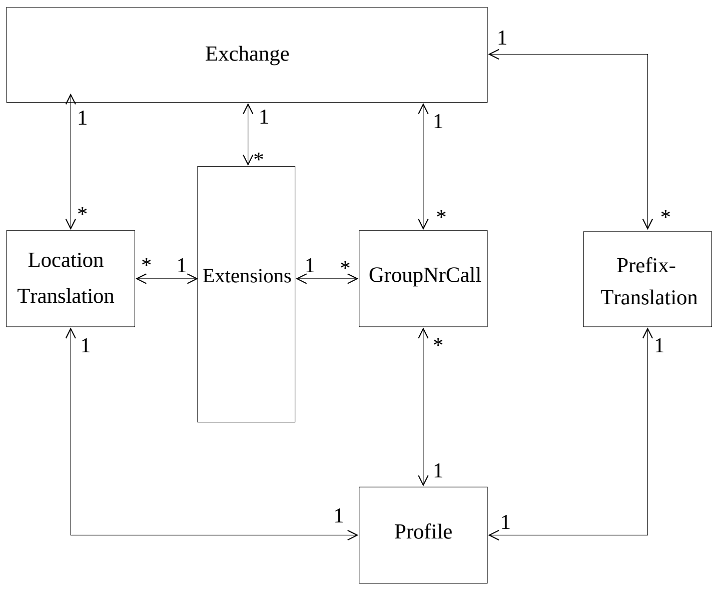


Figure 7.3: Schema of the benchmark database.

implements the transaction manager, OCC is the scheduler, ORD is the data manager, and HOT is the database. Therefore, RODAIN fully implements the abstract model presented earlier.

7.3 Benchmark Database

This benchmark models a hypothetical telecommunication operator. The network has multiple service providers. The service providers may belong to one operator or they may belong to multiple operators. Each service provider has its own database, but for this benchmark the databases are similar. The service provider has many customers, each with one or more subscriptions for different available services. The database represents the telecommunication services and billing information of each entity (service provider and service).

The components of the database are defined as consisting of five separate and individual classes: ServiceProvider, HomeProfile, VisitorProfile, ServiceInfo, and Subscriptions. The relationships among these classes are defined in the following UML-diagram (Figure 7.3). This diagram is a logical description of the classes.

Although the entity names in the model are collected from the wireless world, they can be used to model the IN services as well. The Virtual

Private Network as described in [43] maps to our model by simply changing the names of the entities. The simple read and write transactions are also similar in both worlds. In order for the transactions to represent a similar amount of work to all the systems, it is important that the records handled by the database servers, file systems, etc. are of the same size. Therefore, the records/rows must be stored in an uncompressed format.

The size of each item in the ServiceProvider class must be at least 100 bytes. The actual attributes of the class are not important for the benchmark itself. However, one of the attributes, called ProviderId, must uniquely identify the service provider and the information attached to it. In this benchmark we assume the ServiceProvider class (Table 7.1) to contain in any order or representation the attributes ProviderId, ProviderName and ProviderInfo.

Table 7.1: ServiceProvider class.

Data Attribute	Description
ProviderId	unique identifier across the ServiceProviders
ProviderName	Name of the ServiceProvider
ProviderInfo	Additional information of the provider

The ServiceInfo class contains the information of the available service at each service provider. In this benchmark the services must be uniquely identified over all services on all service providers. The benchmark concentrates only on the service price and its usage in the service. The size of each item in the ServiceInfo class must be at least 100 bytes. For the benchmark it must have the attributes (see Table 7.2) ServiceId, ServicePrice and ServiceName. The actual order of these attributes is not described.

Table 7.2: ServiceInfo class.

Data Attribute	Description
ServiceId	unique identifier across the range of Service
ServicePrice	Price of the service
ServiceName	Service name in uncompressed format

Each client of the system has one home service provider. The client information is located in the HomeProfile of that service provider. The size of the data item in the HomeProfile must be at least 100 bytes. The

HomeProfile (see Table 7.3) contains the client's subscriber identity (SubsId), which identifies the client over all clients in the whole system. The clients also have a local identity (ClientId), which usually is much smaller than the SubsId. This id is used to connect the client with her local subscribed services. The home profile must also contain the client's true phone number, the current roaming position as the service provider id, and the client address as the connection information to the client.

Table 7.3: HomeProfile class.

Data Attribute	Description
SubsId	unique identifier across the range of HomeProfiles
ClientId	Client identifier
PhoneNumber	Subscriber real phone number
CurPosition	Current position, i.e., provider identification
SubscriberAddress	Subscribers address
SubscriberInfo	Additional information of the subscriber

The service provider keeps information about current roaming users in VisitorProfile (see Table 7.4). The class must at the minimum contain the identification of the roaming user and the identification of the user's own service provider. The size of each item in the VisitorProfile is assumed to be small, only 16 bytes. To map the visitors with the services, the visitors are also attached with a temporary ClientId for the mapping.

Table 7.4: VisitorProfile class.

Data Attribute	Description
SubsId	Visitor identification, the same as in HomeProfile
ClientId	Client identification
HomeLocation	service providers ProviderId

The class Subscriptions (see Table 7.5) connects the subscribers and services together. It must have the identification to the user and the service. These identifications together identify each data item in this class. The size of the data item must be at least 50 bytes. In addition to the identification attributes SubServiceId and SubClientId, the class contains information specific to this subscription. The information is stored in SubType, SubValue, and SubName attributes.

Table 7.5: Subscriptions class.

Data Attribute	Description
SubClientId	Identification of the subscriber
SubServiceId	Identification of the service
SubType	Subscription type
SubValue	Connection information
SubName	Subscription name

The data item identifiers of the ServiceProvider, ServiceInfo, Subscriptions, HomeProfile, and VisitorProfile must not directly represent the physical disk addresses of the items or any offsets thereof. The applications may not reference records using relative record numbers since they are simply offsets from the beginning of a file. For each nominal configuration, the test must use the minimum database size given in Table 7.6.

Table 7.6: Database size.

Table/Class	Number of rows
ServiceProvider	2
Services	10
Subscriptions	50000
HomeProfile	30000
VisitorProfile	10000

The classes presented above are necessary in the databases of each service provider. The transactions represent the work performed when a customer uses some telecommunication services. The transactions are performed in the database of some service provider(s). This set of transactions presents the minimum that can be used for evaluating a database for telecommunication. Each transaction has its own purpose in the test set.

7.4 Local workload

In this section transactions and the workload used in the local database experiments is presented. These transactions and workload are used in the experiments presented in this thesis. The workload used in the experiments is presented using the model presented in Definition 2.1 (see Chapter 2).

Error checking is ommitted from this presentation to make transactions more readable. In RODAIN these transactions are implemented using precoded low-level interface transactions. Implementation is based on begin, read, write, commit, and abort operations.

The GetSubscriber transaction (see Algorithm 7.1) is used to search a specific subscriber in the database. It is mainly a simple local read-only transaction. These transactions require subscriber identification, i.e. a Sid attribute as an input variable.

```
int Transaction::GetSubscriber(OID sid)
{
    begin();
    HomeProfile = read(sid);
    commit();
    return Homeprofile.getPhoneNumber();
}
```

Algorithm 7.1: GetSubscriber transaction.

The UpdateSubscriber transaction (see Algorithm 7.2) is used to modify some of the subscriber data. This simple update transaction represents a large set of update transactions that access data in one class. The chosen update transaction updates the subscriber's name and number. It is actually the same as the Modify Subscriber Number service. This query also represents the Customer Management service in Intelligent Network Capability Set 1 (IN CS-1) [39].

```
void Transaction::UpdateSubscriber(OID sid,data1,data2)
{
    begin();
    HomeProfile = read(sid);
    HomeProfile.setSubscriberAddress(data1);
    HomeProfile.setSubscriberAddInfo(data2);
    write(sid,HomeProfile);
    commit();
}
```

Algorithm 7.2: UpdateSubscriber transaction.

The GetAccessData transaction (see Algorithm 7.3) can be used in three different scenarios. In the first scenario, the query is used to fetch the new destination number from the database in case the number given in the parameter is an abbreviated number (e.g. this query implements the Abbreviated Dialing service in IN). The query returns the destination number, which is active when the query is executed. The second scenario for the GetAccessData is to fetch the new destination number from the database in case the number given in the parameter is forwarded to another number (e.g. this query implements the Call Forwarding service). The query returns the destination number, which is active when the query is executed. Finally, the third possible scenario for the GetAccessData query is to fetch the new destination number from the database in case the number given in the parameter is a group number (e.g. this query implements the Universal Access Number service). When executed the query returns the current active destination number. All of these three scenarios in this database map to the same transaction. The actual phone number is fetched from the Subscription table.

```
int Transaction::GetAccessData(OID sid)
{
    begin();
    HomeProfile = read(sid);

     if ( HomeProfile == NULL )
    {
        VisitorProfile = read(sid);
        ClientId = VisitorProfile.getClientId();
    }
     else
        ClientId = HomeProfile.getClientId();

    Subscription = read(ClientId);
    commit();
    return Subscription.getSubValue();
}
```

Algorithm 7.3: GetAccessData transaction.

The SetAccessData transaction (see Algorithm 7.4) is used to insert a service subscription which can be an abbreviated number, a forwarded number or a universal access number.

In order for the transactions to represent a similar amount of work to all the systems, it is important that the fractions of different transactions are specified. Table 7.7 shows the fraction of the different transactions in the test load, their conflict priority and relative deadlines.

The actual workload depends on the write fraction used in the experiments. For example if the write fraction is 20%, then the work load contains 40% of FindSubscriber transactions, 40% of GetAccessData transactions, 10% of UpdateSubscriber transactions, and 10% SetAccessData transac-

```
int Transaction::SetAccessData(OID sid,id,type,value,name)
{
    begin();
    Subscription = new Subscription(sid,id,type,value,name);
    write(sid,Subscription);
    commit();
}
```

Algorithm 7.4: SetAccessData transaction.

Table 7.7: Transaction mix.

Transaction name	Fraction of all transactions	Conflict priority	Deadline
FindSubscriber	0..50%	Critical	50ms
UpdateSubscriber	0..50%	Normal	150ms
GetAccessData	0..50%	Medium	50ms
SetAccessData	0..100%	Normal	150ms

tions.

The workload in a test session consists of a variable mix of transactions. Fractions of each transaction type is a test parameter. Other test parameters include the arrival rate, assumed to be exponentially distributed (Table 7.8).

Table 7.8: Transaction test parameters

Parameter	Unit	Value	Description
ArrRate	trans/s	100–500	Average arrival rate of transactions
Deadline type		firm	All transactions are firm transactions
DbSize	num	90012	Number of objects in the database
NumTRP	num	20	Number of Transaction Processes

7.5 Experimental Setup

All experiments were executed in the RODAIN (See Section 7.2) prototype database running on a machine with one processor Pentium Pro 200MHz and 64 MB of main memory with the Chorus/ClassiX real-time operating system [80].

Every test session contains 10 000 transactions and is repeated at least 20 times. The reported values are the means of the repetitions. The experiments are based on the benchmark that models a hypothetical telecommunication operator. The network has multiple service providers. The service providers may belong to one operator or they may belong to multiple operators. Each service provider has its own database, but for this benchmark the databases are similar. The service provider has many customers, each with one or more subscriptions for different available services. The database represents the telecommunication services and billing information of each entity (service provider and service). The transactions represent the work performed when a customer uses some telecommunication services. The transactions are performed in the database of some service provider(s). This set of transactions presents the minimum that can be used for evaluating a database for telecommunication. Only firm real-time transactions are used in experiments. The database used in the experiments was presented in Section 7.3 and transactions and workload was presented in Section 7.4.

7.6 Experiments Using OCC-DATI

In this section we examine how well our OCC-DATI algorithm (see Chapter 4) performs when compared to the OCC-TI [62] and OCC-DA [53] algorithms. In the test the fraction of transactions which missed their deadline is measured, i.e. the miss ratio of the transactions is measured (see Definition 7.1).

Definition 7.1

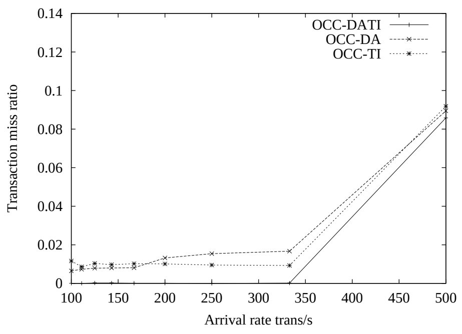
$$\text{Miss ratio} = \frac{\text{Number of transactions that missed their deadline}}{\text{Number of transactions}}$$

□

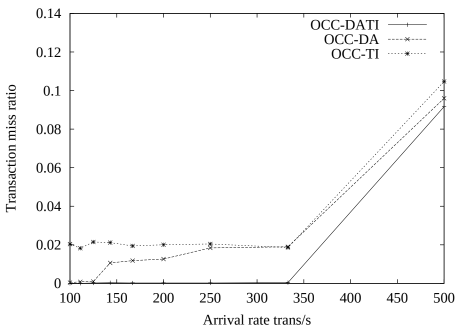
In the first experiments a fixed fraction of write transactions has been used, varying the arrival rate of the transactions from 100 to 500 transactions per second (Figure 7.4). From the experimental results one can find that the experimental system starts to become overloaded when the arrival rate of the transactions is 333 transactions per second.

From the experiments one can see that system becomes overloaded at 500 transactions per second. There is quite few missed transactions because of the concurrency control. This is because we have used very old and slow

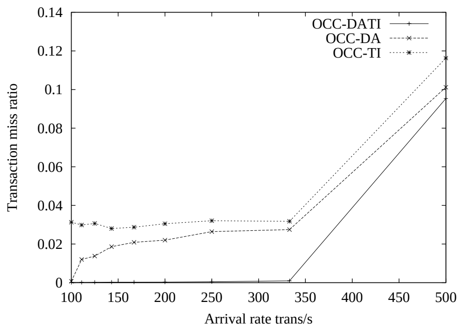
Pentium Pro 200MHz processor machines. Additionally, Chorus/ClassiX real-time operating system did not very well work on overloading situations. This is clearly characteristic of the used environment.



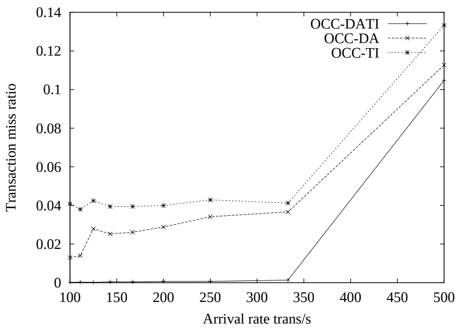
(a) writes 10%



(b) writes 20%



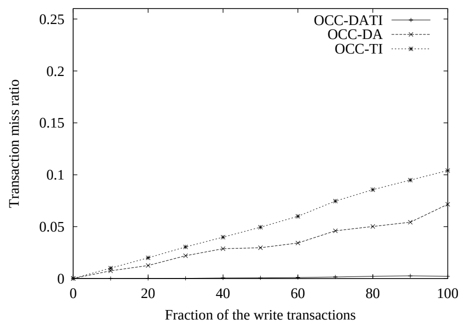
(c) writes 30%



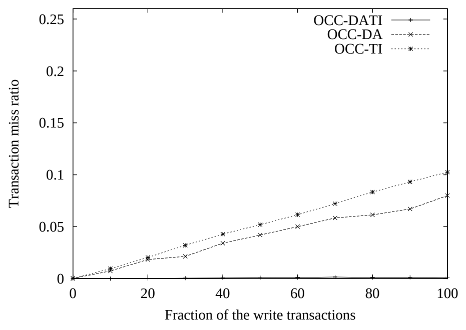
(d) writes 40%

Figure 7.4: Miss ratio of the OCC-DATI, OCC-TI, and OCC-DA methods compared when the arrival rate of the transactions is varied from 100 to 300 transactions per second and the fraction of write transactions is between 10% and 40%.

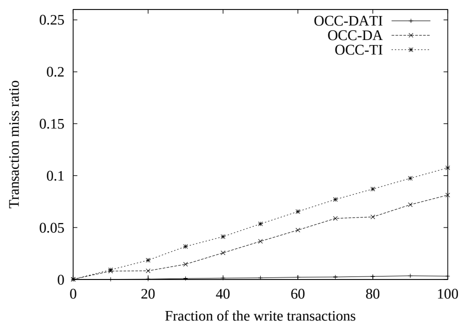
In the second experiments a fixed arrival rate of transactions has been used and the fraction of write transactions has been varied from 10% to 100%. The rest are read-only transactions. In Figure 7.5(a) the arrival rate of the transactions is 200 transactions per second, in Figure 7.5(b) the arrival rate of the transactions is 250 transactions per second, in Figure 7.5(c) the arrival rate of the transactions is 333 transactions per second, and in Figure 7.5(d) the arrival rate of the transactions is 500 transactions per second.



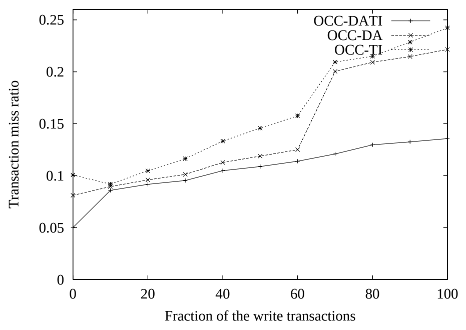
(a) arrival rate 200 trans/s



(b) arrival rate 250 trans/s



(c) arrival rate 333 trans/s



(d) arrival rate 500 trans/s

Figure 7.5: Miss ratio of the OCC-DATI, OCC-TI, and OCC-DA methods compared when the fraction of the write transactions is varied from 10% to 100% and the arrival rate of the transactions is between 200 and 500 transactions per second.

OCC-DATI clearly offers the best performance in all tests. This confirms that the overhead for supporting dynamic adjustment in OCC-DATI is smaller than the one in OCC-DA. This also confirms that the number of transaction restarts is smaller in OCC-DATI than OCC-TI or OCC-DA. The results clearly show how unnecessary restarts affect the performance of the OCC-TI. The results also confirm the conclusion in [53] that OCC-DA outperforms OCC-TI. The results [62] have already confirmed that OCC-TI outperforms the OCC-BC, OPT-WAIT and WAIT-50 algorithms. These results confirm that OCC-DATI also outperforms OCC-DA and OCC-TI when the arrival rate of the transactions is increased.

7.7 Experiments Using OCC-PTI

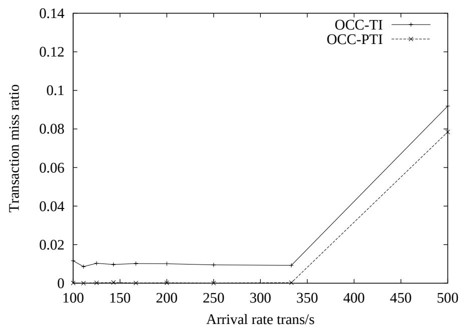
This section presents experiment results when the original OCC-TI (see Section 3.4) and the OCC-PTI (see Section 5.1.3) are compared. Performance is measured using Miss ratio measuring the fraction of transactions that missed their deadlines (see Definition 7.1) and a new Critmiss ratio is used. Critmiss ratio measures the fraction of FindSubscriber transactions that missed their deadlines.

Definition 7.2

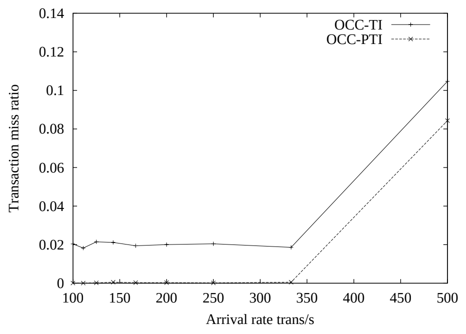
$$\text{Critmiss ratio} = \frac{\text{Number of FindSubscriber trans. that missed their deadlines}}{\text{Number of FindSubscriber transactions}}$$

□

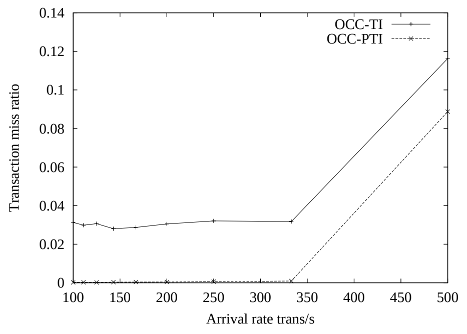
In the first experiments, the arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 7.6(a) the fraction of write transactions is 10%. In Figure 7.6(c) the fraction of write transactions is 20%. In Figure 7.6(a) the fraction of write transactions is 30%. In Figure 7.6(c) the fraction of write transactions is 40%. Figure 7.6 shows that the OCC-PTI performs better than OCC-TI, especially when the fraction of the write transactions increases. This is because the OCC-PTI does not suffer from the unnecessary restart problem. Figure 7.6 also clearly shows that the experimental system becomes overloaded when the arrival rate of the transactions is more than 333 transactions per second.



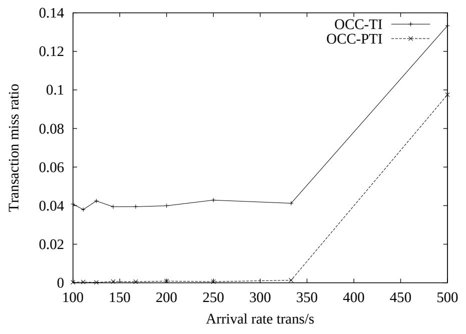
(a) write fraction 10%



(b) write fraction 20%



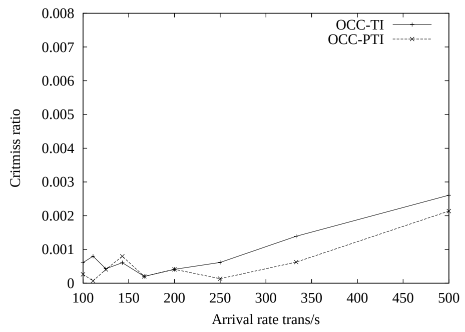
(c) write fraction 30%



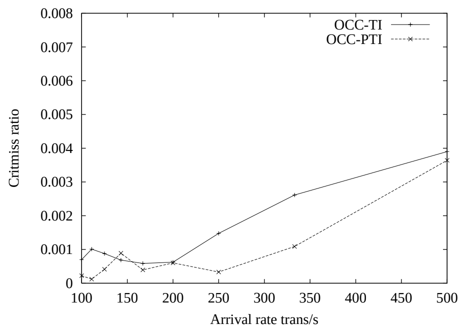
(d) write fraction 40%

Figure 7.6: Miss ratio of the OCC-TI and OCC-PTI compared when the arrival rate of the transactions is varied from 100 to 500 transactions per second and the fraction of write transactions is between 10% and 40%.

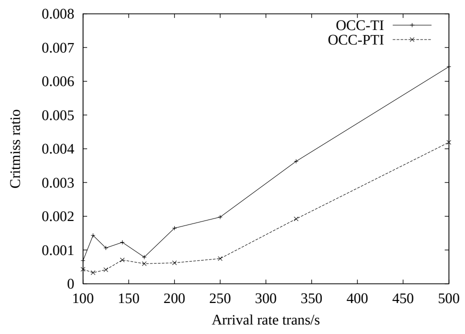
Figures 7.7(a) to 7.6(d) show the critmiss ratio of transactions. This demonstrates how the OCC-PTI favors transactions of high priority. The OCC-PTI clearly offers better chances for high priority transactions to complete according to their deadlines. The results clearly indicate that the OCC-PTI meets the goal of favoring transactions of high priority.



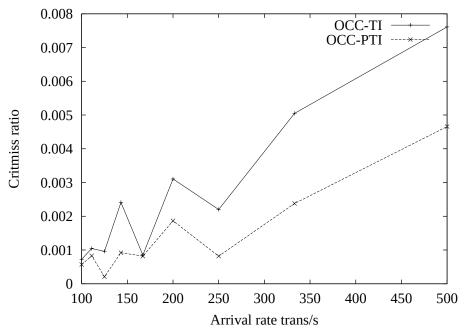
(a) write fraction 10%



(b) write fraction 20%



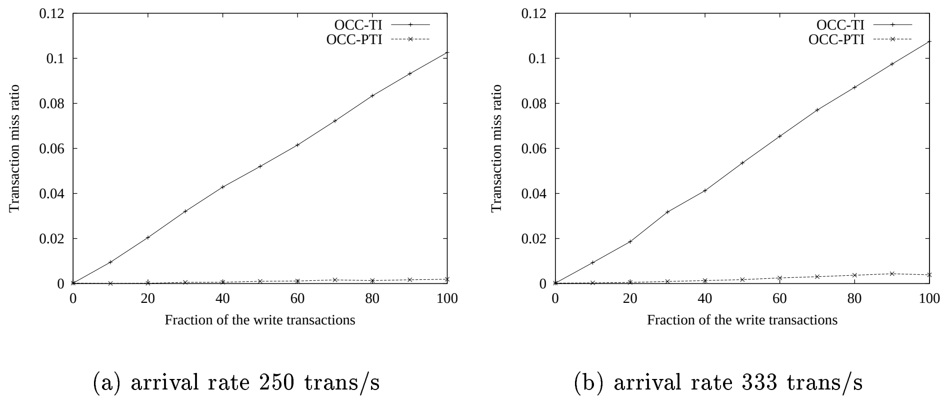
(c) write fraction 30%



(d) write fraction 40%

Figure 7.7: Critmiss ratio of the OCC-TI and OCC-PTI compared when the arrival rate of the transactions is varied from 100 to 500 transactions per second and the fraction of write transactions is between 10% and 40%.

In the next experiments, the arrival rate of the transactions is fixed and the write fraction of the transactions is varied from 10% to 100%. Figure 7.8 shows that the OCC-PTI performs better than the original OCC-TI, especially when the write fraction rate is high. This is because the OCC-PTI does not suffer from the unnecessary restart problem.



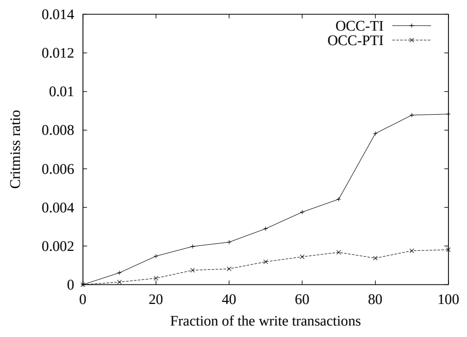
(a) arrival rate 250 trans/s

(b) arrival rate 333 trans/s

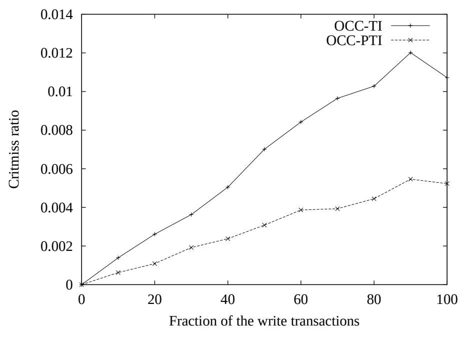
Figure 7.8: Miss ratio of the OCC-TI and OCC-PTI compared when the fraction of the write transactions is varied from 10% to 100% and the arrival rate of the transactions is 250 and 333 transactions per second.

In the next experiments, the arrival rate of the transactions is fixed and the write fraction of the transactions is varied from 10% to 100% and the critmiss ratio is measured. Figure 7.9 shows that the OCC-PTI performs better than the original OCC-TI, especially when the write fraction rate is high. This is because the OCC-PTI does not suffer from the unnecessary restart problem. Again, OCC-PTI clearly offers better chances for high priority transactions to complete according to their deadlines.

In the final experiments, results from the OCC-DATI method are included. The arrival rate of the transactions is varied from 100 to 500 transactions per second. The performance of the OCC-PTI is similar to OCC-DATI. Figure 7.10 shows miss ratio measurements and Figure 7.11 shows critmiss ratio measurements. Experiments clearly show that priority cognizance is not a feasible approach for improving the performance of real-time concurrency control methods beyond the current state of the art as noted also in [16].

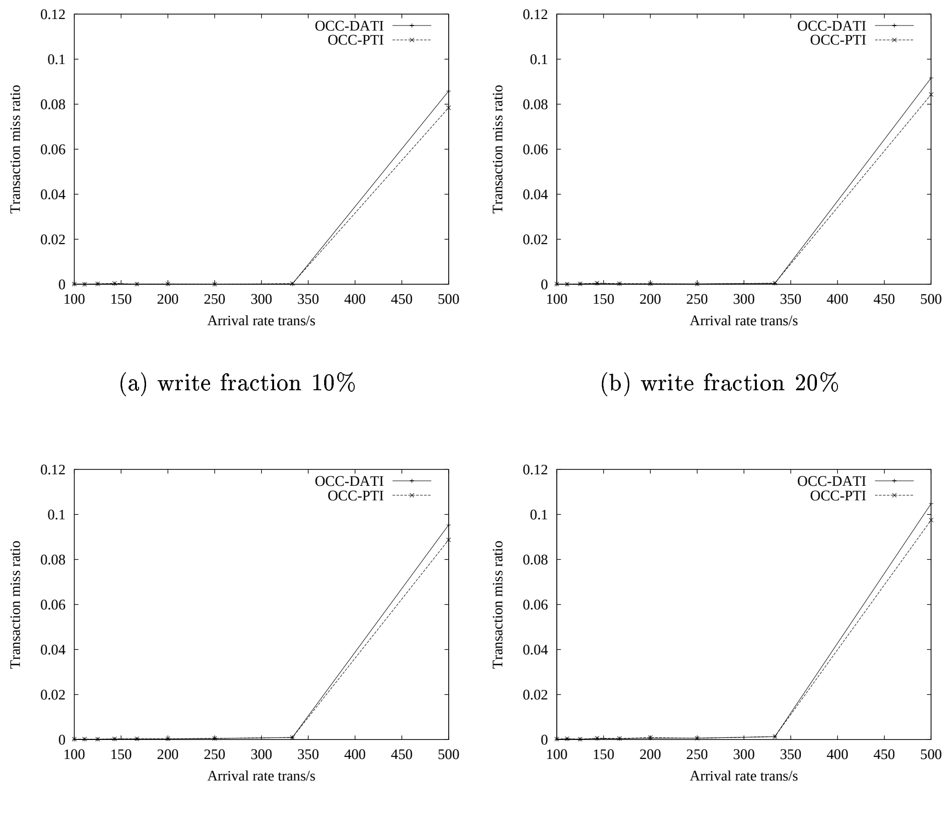


(a) arrival rate 250 trans/s



(b) arrival rate 333 trans/s

Figure 7.9: Critmiss ratio of the OCC-TI and OCC-PTI compared when the fraction of the write transactions is varied from 10% to 100% and the arrival rate of the transactions is 250 and 333 transactions per second.



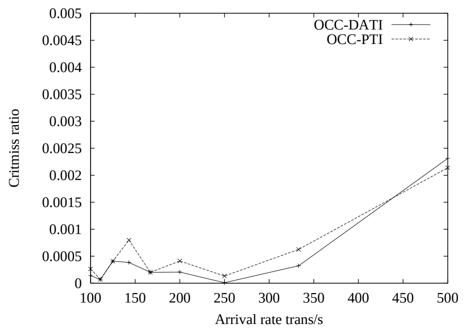
(a) write fraction 10%

(b) write fraction 20%

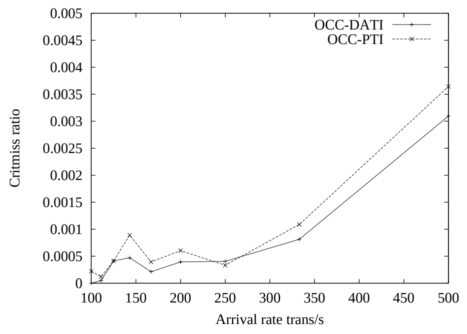
(c) write fraction 30%

(d) write fraction 40%

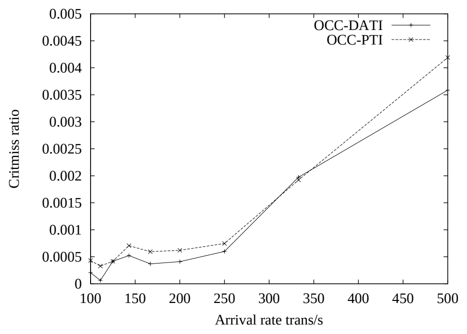
Figure 7.10: Miss ratio of the OCC-DATI and OCC-PTI compared when the arrival rate of the transactions is varied from 100 to 500 transactions per second and the fraction of write transactions is between 10% and 40%.



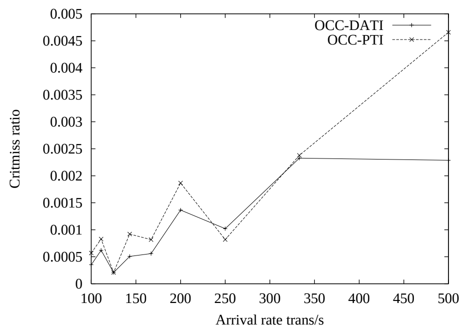
(a) write fraction 10%



(b) write fraction 20%



(c) write fraction 30%

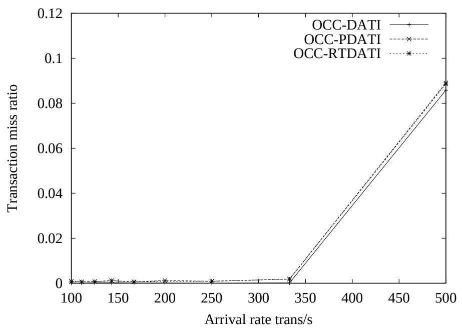


(d) write fraction 40%

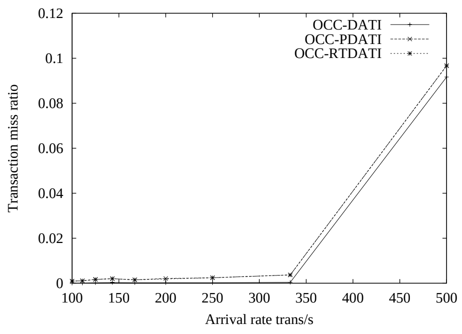
Figure 7.11: Critmiss ratio of the OCC-DATI and OCC-PTI compared when the arrival rate of the transactions is varied from 100 to 500 transactions per second and the fraction of write transactions is between 10% and 40%.

7.8 Experiments Using OCC-PDATI and OCC-RTDATI

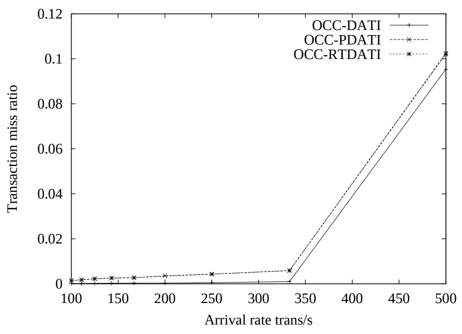
In this section results from experiments with OCC-DATI (see Chapter 4), OCC-PDATI (see Section 5.2) and OCC-RTDATI (see Section 5.3) methods are presented. Performance is measured using the Miss ratio measuring the fraction of transactions that missed their deadlines (see Definition 7.1) and the Critmiss ratio, which measures the fraction of FindSubscriber transactions that missed their deadlines (see Definition 7.2).



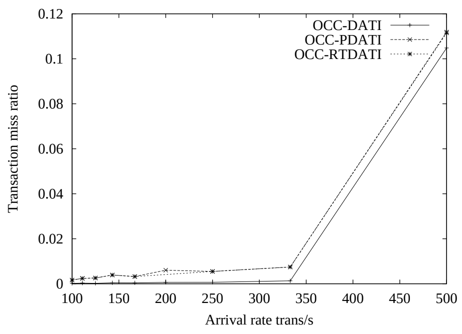
(a) writes 10%



(b) writes 20%



(c) writes 30%



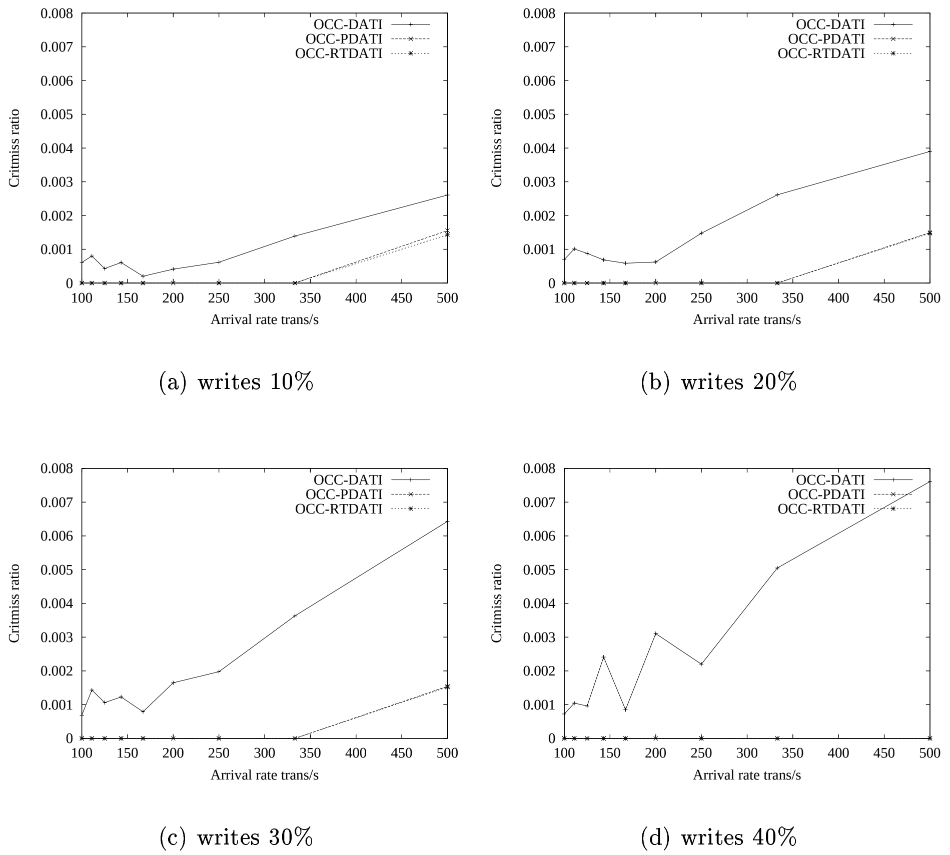
(d) writes 40%

Figure 7.12: Miss ratio of the OCC-DATI, OCC-RTDATI, and OCC-PDATI compared when the arrival rate of the transactions is varied from 100 to 300 transactions per second and the fraction of write transactions is between 10% and 40%.

In the first experiments a fixed fraction of write transactions has been used and the arrival rate of the transactions have been varied from 100 to 500 transactions per second (Figure 7.12).

As expected, there is some overhead when information about the importance of the transaction is used in dynamic adjustment of the serialization order. As Figure 7.12 indicates, the overhead using additional information from the transactions is quite low. The miss ratio of the transactions when using the OCC-PDATI or OCC-RTDATI algorithm is only slightly higher than in the OCC-DATI.

Finally, Figure 7.13 shows the miss ratio of transactions of high importance. Figure 7.13 demonstrates how the OCC-PDATI and OCC-RTDATI favor transactions of high importance. OCC-PDATI and OCC-RTDATI clearly offer better chances for high priority transactions to complete according to their deadlines. The results clearly indicate that OCC-PDATI and OCC-RTDATI meet the goal of favoring transactions of high importance.



(a) writes 10%

(b) writes 20%

(c) writes 30%

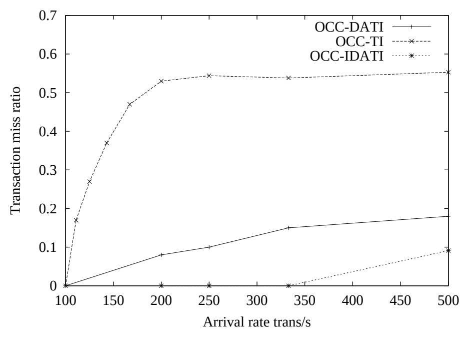
(d) writes 40%

Figure 7.13: Critmiss ratio of the OCC-DATI, OCC-RTDATI, and OCC-PDATI compared with transactions of high importance when the arrival rate of the transactions is varied from 100 to 300 transactions per second and the fraction of write transactions is between 10% and 40%.

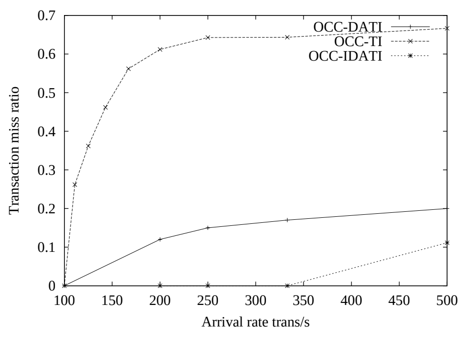
7.9 Experiments Using OCC-IDATI

In the experiments, we examined how well our OCC-IDATI (see Chapter 6) method performs when compared to the OCC-DATI method [74] and to the OCC-TI method [62]. The performance is measured using Miss ratio measuring fraction of transactions that missed their deadlines (see Definition 7.1) and the Critmiss ratio, which measures the fraction of FindSubscriber transactions that missed their deadlines (see Definition 7.2).

In the first set of experiments, a fixed fraction of write transactions was used. The arrival rate of transactions was the varying parameter. OCC-IDATI clearly offers the best performance in all tests (see Figure 7.14). This confirms that the overhead for supporting dynamic adjustment in OCC-IDATI is smaller than the one in OCC-TI and in OCC-DATI.



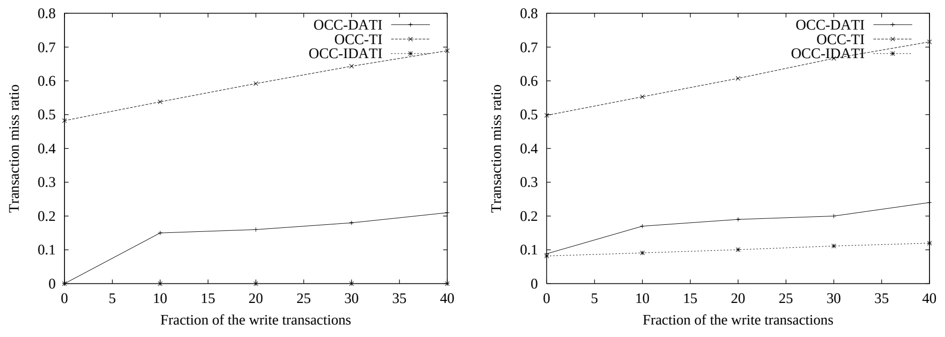
(a) 10% writes



(b) 30% writes

Figure 7.14: Miss ratio of the OCC-IDATI, OCC-DATI and OCC-TI compared when the arrival rate of the transactions is varied from 100 to 500 and the fraction of the write transactions is 10% and 30%.

Figure 7.15 shows the miss-ratio transactions when the transaction's write fraction is varied. Figure 7.15 demonstrates how the OCC-IDATI favors critical transactions (i.e. transactions with high conflict priority). OCC-IDATI clearly offers better chances for critical transactions to complete according to their deadlines. The results clearly indicate that OCC-IDATI meets the goal of favoring critical transactions.



(a) 333 trans/s (b) 500 trans/s

Figure 7.15: Miss ratio of the OCC-IDATI, OCC-DATI and OCC-TI compared when the fraction of the write transactions is varied from 10% to 40% and the arrival rate of the transactions is 333 and 500 transactions per second.

7.10 Summary

Firstly, experiment results show that the unnecessary restart problem found in OCC-TI greatly decreases the performance of OCC-TI. OCC-DATI is clearly superior to OCC-TI in all tests. This is because OCC-DATI avoids unnecessary restart problem found in OCC-TI. Additionally, OCC-DATI outperforms OCC-DA because it makes validation tests more efficiently than OCC-DA. This is also confirmed by the performance tests. Therefore, many unnecessary restarts should and can be avoided efficiently.

Secondly, from the analysis and experiments, we conclude that priority cognizance is not a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems beyond the current state-of-the-art.

Priority cognizance is a viable approach for main-memory real-time database systems if critical transactions (i.e. transactions with high conflict priority) should be favored against less critical transactions. In this case the overall performance of the real-time system does not significantly suffer or improve but the predictability of critical transactions clearly improve. Using priority cognizance in conflict resolution can offer better changes for high priority transactions to complete according to their deadlines.

Finally, from the analysis and experiments, we conclude that it is possible to use integrated and dynamic adaptation in conflict resolution. Thus, developer can affect conflict resolution in the optimistic concurrency control method for main-memory real-time database systems. Therefore, adaptation is a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems.

Chapter 8

Conclusions

Database performance is an important aspect of database usability. The performance of a database system depends not only on the database architecture and algorithms, but also on the platform the database is running on.

Real-time databases are needed when the database requests must be served within given time limits. The database is then designed to support the timely execution on all levels of the database architecture. Especially, it provides transaction scheduling based on priorities, deadlines, or criticality of the transactions. Telecommunication is an example of an application area that has database requirements that require a real-time database or at least time-cognizant database.

Several advantages can be gained when real-time and object orientation are combined. Data objects can have attributes that specify the correctness criteria. Operations can have attributes that tell the resource consumptions of the operations. Transactions are also either transient or persistent objects. They can have attributes that specify the priority, deadline, criticality, and correctness criterion, among other things.

Although the optimistic approach has been shown to be better than locking methods for RTDBSs, it has the problems of unnecessary restarts and heavy restart overhead. This thesis has proposed an optimistic concurrency control method called OCC-DATI. It has several advantages over the other concurrency control methods. The method maintains all the nice properties with forward validation, a high degree of concurrency, freedom from deadlock, and early detection and resolution of conflicts, resulting in both less wasted resources and a smaller number of restarts. All of these are important to the performance of RTDBSs and contribute to greater chances of meeting transaction deadlines.

An efficient method was designed to adjust the serialization order dynamically amongst the conflicting transactions to reduce the number of transaction restarts. Compared with other optimistic concurrency control methods that use dynamic serialization order adjustment, the method presented in this thesis is much more efficient and its overhead is smaller. There is no need to check for conflicts while a transaction is still in its read phase. All the checking is performed in the validation phase. As the conflict resolution between the transactions in OCC-DATI is delayed until a transaction is near completion, there will be more information available for making the choice in resolving the conflict.

This thesis also proposes two different methods to take transaction conflict priority into account in the conflict resolution of the validation phase. These methods are called OCC-PDATI and OCC-RTDATI. When compared with the OCC-DATI method that uses dynamic serialization order adjustment, the OCC-PDATI method offers the same efficiency and the overhead is only slightly larger. The most important feature of the OCC-PDATI is that it clearly offers better chances for the transactions of higher conflict priority to complete before their deadlines when compared to the OCC-DATI. The results clearly indicate that OCC-PDATI meets the goal of favoring transactions of high criticality. Similar results were obtained when OCC-RTDATI was compared to OCC-DATI.

Finally, this thesis also proposes the integrated and adaptive optimistic concurrency control method OCC-IDATI. This method dynamically adapts to different load situations, because the conflict resolution method dynamically selects a resolution method based on the conflict priority of the conflicting transactions. The proposed method offers three different conflict resolution methods. Performance tests show that OCC-IDATI is a viable approach for improving the performance and predictability of real-time concurrency control methods for main-memory real-time database systems beyond the current state-of-the-art.

These ideas have been implemented as a part of the prototype version of the RODAIN real-time database system. The most important feature in the future versions of the RODAIN architecture is *real-time*. A real-time transaction that has an explicit deadline is suitable for telecommunications use. The writer wants to support two kinds of real-time transactions: soft transactions, which may continue execution after the deadline at lower priority; and firm transactions that are terminated when the deadline is not met. How these different timing constraints are integrated to real-time concurrency control is still an open question.

The experiments are based on the benchmark that models a hypothetical telecommunication operator. The network has multiple service providers. The service providers may belong to one operator or they may belong to multiple operators. Each service provider has its own database, but for this benchmark the databases are similar. The service provider has many customers, each with one or more subscriptions for different available services. The database represents the telecommunication services and billing information of each entity (service provider and service). The transactions represent the work performed when a customer uses some telecommunication services. The transactions are performed in the database of some service provider(s). This set of transactions presents the minimum that can be used for evaluating a database for telecommunication. Only firm real-time transactions are used in experiments.

From the experiments one can see that the system becomes overloaded at 500 transactions per second. There is quite a few missed transactions because of the concurrency control. This is because we have used very old and slow Pention Pro 200MHz processor machines. Additionally, Chorus/ClassiX real-time operating system did not work very well in overloading situations. This is clearly characteristic of the environment used. We are currently working on porting the prototype system to the Linux operating system.

The experiments indicated that the validation time of the transactions becomes crucial. To increase database throughput on high-arrival rate levels, the validation time must be shortened. Another possibility is to increase concurrency in the validation process. The third possibility is to use temporal object versioning. This is induced from the semantics of the telecommunication services. An object can be inserted into the database to become valid at the later time. Insertion causes no conflict and thus the validation of inserted objects is a short process.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, March 1988.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Proceedings of the 14th VLDB Conference*, pages 1–12. Morgan Kaufmann, 1988.

[3] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB Conference*, pages 385–396. Morgan Kaufmann, 1989.

[4] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database Systems*, 17(3):513–560, September 1992.

[5] D. Agrawal, A. E. Abbadi, and R. Jeffers. Using delayed commitment in locking protocols for real-time databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 104–113. ACM Press, 1992.

[6] I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23(2):37–43, June 1994.

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

[8] S. R. Biyabani, J. A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pages 487–507. IEEE Computer Society Press, 1988.

[9] J. O. Boese and A. A. Hood. Service control point — database for 800 service. In *Proceedings of GLOBECOM'86*, December 1986.

[10] T. F. Bowen, G. Gopal, G. Herman, and W. Mansfield Jr. A scale database architecture for network services. *IEEE Communications Magazine*, 29(1):52–59, January 1991.

[11] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison-Wesley, 2001.

[12] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance evaluation of two new disk scheduling algorithms for real-time systems. *The Journal of Real-Time Systems*, 3(3):307–336, September 1991.

[13] A. Datta and S. Mukherjee. Buffer management in real-time active database systems. In *Real-Time Database Systems - Architecture and Techniques*, pages 77–96. Kluwer Academic Publishers, 2001.

[14] A. Datta, S. Mukherjee, P. Konana, I. Viguier, and A. Bajaj. Multiclass transaction scheduling and overload management in firm real-time database systems. *Information Systems*, 21(1):29–54, March 1996.

[15] A. Datta and S. H. Son. A study of concurrency control in real-time active database systems. Tech. report, Department of MIS, University of Arizona, Tucson, 1996.

[16] A. Datta, S. H. Son, and V. Kumar. Is a bird in the hand worth more than two in the bush? Limitations of priority cognizance in conflict resolution for firm real-time database systems. *IEEE Transactions on Computers*, 49(5):482–502, May 2000.

[17] A. Datta, I. R. Viguier, S. H Son, and V. Kumar. A study of priority cognizance in conflict resolution for firm real time database systems. In *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, pages 167–180. Kluwer Academic Publishers, 1997.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[19] M. H. Graham. Issues in real-time data management. *The Journal of Real-Time Systems*, 4:185–202, 1992.

[20] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proceedings of the 14th IEEE Real-time Systems Symposium*, pages 56–65, 1993.

[21] P. Graham and K. Barker. Effective optimistic concurrency control in multiversion object bases. *Lecture Notes in Computer Science*, 858:313–323, 1994.

[22] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[23] U. Halici and A. Dogac. An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17(7):712–724, July 1991.

[24] H. Han, S. Park, and C. Park. A concurrency control protocol for read-only transactions in real-time secure database systems. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 458–462. IEEE Computer Society Press, 2000.

[25] J. Hansson and H. Son, S. Overload management in RTDBS. In *Real-Time Database Systems - Architecture and Techniques*, pages 125–140. Kluwer Academic Publishers, 2001.

[26] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[27] J. Haritsa, M. Carey, and M. Livny. Value-based scheduling in real-time database systems. Tech. Rep. CS-TR-91-1024, University of Winconsin, Madison, 1991.

[28] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 94–103. IEEE Computer Society Press, 1990.

[29] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM Press, 1990.

[30] J. R. Haritsa, M. J. Carey, and M. Livny. Data access scheduling in firm real-time database systems. *The Journal of Real-Time Systems*, 4(2):203–241, June 1992.

[31] J. R. Haritsa, K. Ramamritham, and R. Gupta. Real-time commit processing. In *Real-Time Database Systems - Architecture and Techniques*, pages 227–244. Kluwer Academic Publishers, 2001.

[32] D. Hong, S. Chakravarthy, and T. Johnson. Locking based concurrency control for integrated real-time database systems. In *Proceedings of the First International Workshop on Real-Time Databases: Issues and Applications*, March 7-8, 1996.

[33] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB Conference*, pages 35–46, September 1991. Morgan Kaufmann.

[34] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. On using priority inheritance in real-time databases. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, pages 210–221, 1991. IEEE Computer Society Press.

[35] J. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley, and B. Purimetla. Priority inheritance in soft real-time databases. *The Journal of Real-Time Systems*, 4(2):243–268, June 1992.

[36] J. Huang, J. A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 144–153. IEEE Computer Society Press, December 1989.

[37] S.-L. Hung and K.-Y. Lam. Locking protocols for concurrency control in real-time database systems. *ACM SIGMOD Record*, 21(4):22–27, December 1992.

[38] S. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. In *Proceedings of the 21th VLDB Conference*, pages 469–477. Morgan Kaufmann, 1995.

[39] ITU. *Introduction to Intelligent Network Capability Set 1. Recommendation Q.1211.* ITU, International Telecommunications Union, Geneva, Switzerland, 1993.

[40] ITU. *Q-Series Intelligent Network Recommendation Overview. Recommendation Q.1200.* ITU, International Telecommunications Union, Geneva, Switzerland, 1993.

[41] ITU. *Distributed Functional Plane for Intelligent Network CS-1. Recommendation Q.1214.* ITU, International Telecommunications Union, Geneva, Switzerland, 1994.

[42] ITU. *Draft Q.1224 Recommendation IN CS-2 DFP Architecture.* ITU, International Telecommunications Union, Geneva, Switzerland, 1996.

[43] M. Jarke and M. Nicola. Telecommunication databases – applications and performance analysis. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 1819, pages 1–15, 1999.

[44] B.-S. Jeong, D. Kim, and S. Lee. Optimistic secure real-time concurrency control using multiple data version. In *Lecture Notes in Computer Science*, volume 1985, 2001.

[45] B. Kao and R. Cheng. Disk scheduling. In *Real-Time Database Systems - Architecture and Techniques*, pages 97–108. Kluwer Academic Publishers, 2001.

[46] B. Kao and H. Garcia-Molina. Deadline assigment in a distributed soft real-time system. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437. IEEE Computer Society Press, 1993.

[47] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 463–486. Prentice Hall, 1995.

[48] R. Kerboul, J.-M. Pageot, and V. Robin. Database requirements for intelligent network: How to customize mechanisms to implement policies. In *Proceedings of the 4th TINA Workshop*, volume 2, pages 35–46, September 1993.

[49] N. Kline. An update of the temporal database bibliography. *ACM Sigmod Record*, 22(4):66–80, 1993.

[50] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[51] T.-W. Kuo and K.-Y. Lam. Conservative and optimistic protocols. In *Real-Time Database Systems - Architecture and Techniques*, pages 29–44. Kluwer Academic Publishers, 2001.

[52] T.-W. Kuo and K.-Y. Lam. Real-time database systems: An overview of system characteristics and issues. In *Real-Time Database Systems - Architecture and Techniques*, pages 3–8. Kluwer Academic Publishers, 2001.

[53] K.-W. Lam, K.-Y. Lam, and S. Hung. An efficient real-time optimistic concurrency control protocol. In *Proceedings of the First International Workshop on Active and Real-Time Database Systems*, pages 209–225. Springer, 1995.

[54] K.-W. Lam, K.-Y. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Technology and Application Symposium*, pages 174–179. IEEE Computer Society Press, 1995.

[55] K.-W. Lam, V. Lee, S.-L. Hung, and K.-Y. Lam. An augmented priority ceiling protocol for hard real-time systems. *Journal of Computing and Information, Special Issue: Proceedings of Eighth International Conference of Computing and Information*, 2(1):849–866, June 1996.

[56] K.-W. Lam, S. H. Son, and S. Hung. A priority ceiling protocol with dynamic adjustment of serialization order. In *Preceedings of the 13th IEEE Conference on Data Engineering.* IEEE Computer Society Press, 1997.

[57] K.-Y. Lam and S.-L. Hung. Concurrency control for time-constrained transactions in distributed databases systems. *The Computer Journal*, 38(9):704–716, 1995.

[58] K.-Y. Lam, S.-L. Hung, and S. H. Son. On using real-time static locking protocols for distributed real-time databases. *The Journal of Real-Time Systems*, 13(2):141–166, September 1997.

[59] K.-Y. Lam and T.-W. Kuo. Mobile distributed real-time database systems. In *Real-Time Database Systems - Architecture and Techniques*, pages 245–258. Kluwer Academic Publishers, 2001.

[60] J. Lee. *Concurrency Control Algorithms for Real-Time Database Systems.* PhD thesis, Faculty of the School of Engineering and Applied Science, University of Virginia, January 1994.

[61] J. Lee and S. H. Son. An optimistic concurrency control protocol for real-time database systems. In *3rd International Conference on*

Database Systems for Advanced Applications (DASFAA), pages 387–394, 1993.

[62] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium*, pages 66–75. IEEE Computer Society Press, 1993.

[63] J. Lee and S. H. Son. Performance of concurrency control algorithms for real-time database systems. In V. Kumar, editor, *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, pages 429–460. Prentice-Hall, 1996.

[64] U. Lee and B. Hwang. Optimistic concurrency control based on timestamp interval for broadcast environment. In *Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002*, Lecture Notes in Computer Science,vol 2435, 2002.

[65] V. C. S. Lee and K-W. Lam. Optimistic concurrency control in broadcast environments: Looking forward at the server and backward at the clients. In H. V. Leong, W-C. Lee, B. Li, and L. Yin, editors, *First International Conference on Mobile Data Access*, Lecture Notes in Computer Science, vol 1748, pages 97–106. Springer Verlag, 1999.

[66] V. C. S. Lee and K.-W. Lam. Conflict free transaction scheduling using serialization graph for real-time databases. *The Journal of Systems and Software*, 55(1):57–65, November 2000.

[67] V. C. S. Lee, K-Y. Lam, and S-L. Hung. Virtual deadline assignment in distributed real-time database systems. In *Second International Workshop on Real-Time Computing Systems and Applications*, 1995.

[68] K.-J. Lin and M.-J. Lin. Enhancing availability in distributed real-time databases. *ACM SIGMOD Record*, 17(1):34–43, March 1988.

[69] K.-J. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 104–112. IEEE Computer Society Press, 1990.

[70] J. Lindström. Extensions to optimistic concurrency control with time intervals. In *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications*, pages 108–115. IEEE Computer Society Press, 2000.

[71] J. Lindström. Integrated and adaptive optimistic concurrency control method for real-time databases. In *Proceedings of the 8th International Conference on Real-Time Computing Systems and Applications*, pages 143–151, 2002.

[72] J. Lindström and T. Niklander. Benchmark for real-time database system for telecommunications. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 2209, 2001.

[73] J. Lindström, T. Niklander, P. Porkka, and K. Raatikainen. A distributed real-time main-memory database for telecommunication. In *Databases in Telecommunications*, Lecture Notes in Computer Science, vol 1819, pages 158–173, 1999.

[74] J. Lindström and K. Raatikainen. Dynamic adjustment of serialization order using timestamp intervals in real-time databases. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, pages 13–20. IEEE Computer Society Press, 1999.

[75] J. Lindström and K. Raatikainen. Using importance of transactions and optimistic concurrency control in firm real-time databases. In *Proceedings of 7th International Conference on Real-Time Computing Systems and Applications*, pages 463–467. IEEE Computer Society Press, 2000.

[76] J. Lindström and K. Raatikainen. Using real-time serializability and optimistic concurrency control in firm real-time databases. In *Proceedings of the 4th IEEE International Baltic Workshop on DB and IS BalticDB&IS'2000, May 1-5*, pages 25–37, 2000.

[77] 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, January 1973.

[78] K. Marzullo. Concurrency control for transactions with priorities. Tech. Report TR 89-996, Department of Computer Science, Cornell University, Ithaca, NY, May 1989.

[79] C. Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, 1986.

[80] Dick Pountain. The Chorus microkernel. *Byte*, pages 131–138, January 1994.

[81] B. Purimetla, R. M. Sivasankaran, K. Ramamritham, and J. A. Stankovic. Real-time databases: Issues and applications. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 487–507. Prentice Hall, 1996.

[82] K. Raatikainen. Database access in intelligent networks. In *Proceedings of the IFIP TC6 Workshop on Intelligent Networks*, pages 163–183, Lappeenranta, Finland, 1994. Lappeenranta University of Technology.

[83] K. Ramamritham. Real-time databases. *Distributed and Parallel Databases*, 1:199–226, April 1993.

[84] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[85] R. Ramamritham, R. M. Sivasankaran, J. A. Stankovic, D. F. Towsley, and M. Xiong. Integrating temporal, real-time, and active databases. *SIGMOD Record*, 25(1):8–12, 1996.

[86] M. Ronström. *Design and Modelling of a Parallel Data Server for Telecom Applications.* PhD thesis, Linköping University, Department of Computer and Information Science, 1997.

[87] L. Sha, R. Rajkumar, and J. P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record*, 17(1):82–98, March 1988.

[88] 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.

[89] L. Sha, R. Rajkumar, S. H. Son, and C.-H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, 40(7):793–800, January 1991.

[90] M. Singhal. Issues and approaches to design of real-time database systems. *ACM SIGMOD Record*, 17(1):19–33, March 1988.

[91] M. Sivasankaram, R. and J. Ramamritham, K. an Stankovic. System failure and recovery. In *Real-Time Database Systems - Architecture and Techniques*, pages 109–124. Kluwer Academic Publishers, 2001.

[92] R. Snodgrass and I. Ahn. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[93] S. Son, D. Rasikan, and B. Thuraisingham. Improving timeliness in real-time secure database systems. *SIGMOD Record*, 25(1):25–33, 1996.

[94] S. H. Son, J. Lee, and Y. Lin. Hybrid protocols using dynamic adjustment of serialization order for real-time concurrency control. *The Journal of Real-Time Systems*, 4(2):269–276, June 1992.

[95] S. H. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *Proceedings of the 8th International Conference on Data Engineering*, pages 527–534. IEEE Computer Society Press, 1992.

[96] M. D. Soo. Bibliography on temporal databases. *ACM Sigmod Record*, 20(1):14–23, 1991.

[97] J. A. Stankovic. Misconceptions about real-time computing. *IEEE Computer Magazine*, 21(10):10–19, October 1988.

[98] J. A. Stankovic, S. H. Son, and J. Hansson. Misconceptions about real-time databases. *IEEE Computer*, 32(6):29–36, June 1999.

[99] J. A. Stankovic and W. Zhao. On real-time transactions. *ACM SIGMOD Record*, 17(1):4–18, March 1988.

[100] J. Taina and K. Raatikainen. Experimental real-time object-oriented database architecture for intelligent networks. *Engineering Intelligent Systems*, 4(3):57–63, September 1996.

[101] A. Tansel, J. Clifford, S. Jojodia, A. Segev, and R. (ed.) Snodgrass. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1994.

[102] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[103] Ø. Torbjørnsen, S. Hvasshovd, and Y. Kim. Towards real-time performance in a scalable, continuously available telecom DBMS. In *Proceedings of the First Int. Workshop on Real-Time Databases*, pages 22–29. Morgan Kaufmann, 1996.

[104] Ö. Ulusoy and G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, 18(6):559–580, 1993.

[105] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[106] M. Xiong, J. A. Stankovic, R. Ramamritham, D. F. Towsley, and R. M. Sivasankaran. Maintaining temporal consistency: Issues and algorithms. In *RTDB*, pages 1–6, 1996.

[107] I. Yoon and S. Park. Enhancement of alternative version concurrency control using dynamic adjustment of serialization order. In *Proceedings of the Second International Workshop on Real-Time Databases: Issues and Applications*, September 18-19, 1997.

[108] P. S. Yu and D. M. Dias. Analysis of hybrid concurrency control for a high data contention environment. *IEEE Transactions on Software Engineering*, SE-18(2):118–129, February 1992.

[109] P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son. On real-time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.

[110] G. Özsoyoglu and T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[111] M. T. Özsu and P. Valduriez. *Principles of Distributed Database System.* Prentice Hall, second edition, 1999.

A-1992-4 A. Valmari: Alleviating state explosion during verification of behavioural equivalence. 57 pp.

A-1992-5 P. Floréen: Computational complexity problems in neural associative memories. 128 + 8 pp. (Ph.D. thesis).

A-1992-6 P. Kilpeläinen: Tree matching problems with applications to structured text databases. 110 pp. (Ph.D. thesis).

A-1993-1 E. Ukkonen: On-line construction of suffix-trees. 15 pp.

A-1993-2 Alois P. Heinz: Efficient implementation of a neural net α-β-evaluator. 13 pp.

A-1994-1 J. Eloranta: Minimal transition systems with respect to divergence preserving behavioural equivalences. 162 pp. (Ph.D. thesis).

A-1994-2 K. Pohjonen (toim./ed.): Tietojenkäsittelyopin laitoksen julkaisut 1992–93 – Publications from the Department of Computer Science 1992–93. 58 s./pp.

A-1994-3 T. Kujala & M. Tienari (eds.): Computer Science at the University of Helsinki 1993. 95 pp.

A-1994-4 P. Floréen & P. Orponen: Complexity issues in discrete Hopfield networks. 54 pp.

A-1995-1 P. Myllymäki: Mapping Bayesian networks to stochastic neural networks: a foundation for hybrid Bayesian-neural systems. 93 pp. (Ph.D. thesis).

A-1996-1 R. Kaivola: Equivalences, preorders and compositional verification for linear time temporal logic and concurrent systems. 185 pp. (Ph.D. thesis).

A-1996-2 T. Elomaa: Tools and techniques for decision tree learning. 140 pp. (Ph.D. thesis).

A-1996-3 J. Tarhio & M. Tienari (eds.): Computer Science at the University of Helsinki 1996. 89 pp.

A-1996-4 H. Ahonen: Generating grammars for structured documents using grammatical inference methods. 107 pp. (Ph.D. thesis).

A-1996-5 H. Toivonen: Discovery of frequent patterns in large data collections. 116 pp. (Ph.D. thesis).

A-1997-1 H. Tirri: Plausible prediction by Bayesian inference. 158 pp. (Ph.D. thesis).

A-1997-2 G. Lindén: Structured document transformations. 122 pp. (Ph.D. thesis).

A-1997-3 M. Nykänen: Querying string databases with modal logic. 150 pp. (Ph.D. thesis).

A-1997-4 E. Sutinen, J. Tarhio, S.-P. Lahtinen, A.-P. Tuovinen, E. Rautama & V. Meisalo: Eliot – an algorithm animation environment. 49 pp.

A-1998-1 G. Lindén & M. Tienari (eds.): Computer Science at the University of Helsinki 1998. 112 pp.

A-1998-2 L. Kutvonen: Trading services in open distributed environments. 231 + 6 pp. (Ph.D. thesis).

A-1998-3 E. Sutinen: Approximate pattern matching with the q-gram family. 116 pp. (Ph.D. thesis).

A-1999-1 M. Klemettinen: A knowledge discovery methodology for telecommunication network alarm databases. 137 pp. (Ph.D. thesis).

A-1999-2 J. Puustjärvi: Transactional workflows. 104 pp. (Ph.D. thesis).

A-1999-3 G. Lindén & E. Ukkonen (eds.): Department of Computer Science: annual report 1998. 55 pp.

A-1999-4 J. Kärkkäinen: Repetition-based text indexes. 106 pp. (Ph.D. thesis).

A-2000-1 P. Moen: Attribute, event sequence, and event type similarity notions for data mining. 190+9 pp. (Ph.D. thesis).