DEPARTMENT OF COMPUTER SCIENCE

SERIES OF PUBLICATIONS C

REPORT C-2001-9

Optimistic Concurrency Control Methods for Real-Time Database Systems

Jan Lindström

UNIVERSITY OF HELSINKI

FINLAND

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/

Licentiate Thesis, Series of Publications C, Report C-2001-9

Helsinki, Feb 2001, 98 pages

Abstract

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

Contents

Chapter 1

Introduction

Numerous real-word applications contain time-constrained access to data as well as access to data that has temporal validity. Consider for example a telephone switching system, network management, navigation systems, stock trading, and command and control systems. Moreover consider the following tasks within these environments: looking up the ”800 directory”, obstacle detection and avoidance, radar tracking and recognition of objects. All of these entail gathering data from the environment, processing information in the context of information obtained in the past, and contributing *timely* response. Another characteristic of these examples is that they entail processing both temporal data, which loses its validity after a certain time intervals, as well as historical data.

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 approaches chosen in databases is to get a good throughput or response time. In contrast, *real-time systems* can also deal with temporal data, i.e., data that becomes outdated after a certain time. Due to the temporal character of the data and the response-time requirements forced by the environment, tasks in real-time systems have time constraints, e.g. periods or deadlines. The important difference is that the goal of real-time systems is to meet the time constraints of the tasks.

One of the most important points to remember here is that real-time does not just mean fast [78]. 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 fashion, that is, to use time-cognizant methods to deal with deadlines or periodicity constraints associated with tasks. 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 transaction execution in spite of concurrency and failures [65].

As a sample application let us consider database system for telecommunication applications called *Network Database System* in more detail. Recent developments in network and switching technologies have increased the data intensity of telecommunications systems and services. This is clearly 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. Until now, the database research community has only paid little attention to data management in telecommunications and has contributed little beyond core database technology. The challenges facing telecom data management are now at a point where the database research community can and should become deeply involved. 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 network database system must offer real-time access to data [33, 34]. This is due to the fact that most read requests are for logic 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 with the cost of less throughput. The best alternative is that the database can guarantee that all requests are replied to within a specific time interval. The average time limit for a read request is around 50ms. About 90% of all read requests must be served in that time. 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.

Network database system services consist of two very different kind of semantics: service

provision services and service management services. *Service provision services* define possible extra services for customers [32]. Service provision transactions have quite strict deadlines and their arrival rate can be high (about 7000 transactions/second), but most service provision transactions have read-only semantics. In transaction scheduling, service provision transactions can be expressed as firm deadline transactions. *Service management services* defines possible management services for customer and network administration [32]. 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.

The requirements of the telecommunications database architectures originate in the following areas [38]: real-time access to data, fault tolerance, distribution, object orientation, efficiency, flexibility, multiple interfaces, security and compatibility [7, 66, 80]. In summary, Network Services Databases is a soft/firm real-time system that contains rich data and transaction semantics which can be exploited to design better methods for concurrency control, recovery, and scheduling. It has 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 [8], most real-time concurrency control schemes considered in the literature are based on two-phase locking (2PL) [15]. 2PL 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. [5, 6, 27, 31, 46, 58, 75]). 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 [68].

Optimistic concurrency control methods [20, 41] 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. [13, 14, 26, 42, 43, 49]). Although optimistic approaches have been shown to be better than locking methods for real-time database systems [23, 24] , they have the

problem of unnecessary restarts and heavy restart overhead. This is due to the late conflict detection that increases the restart overhead since some near-to-complete transactions have to be restarted. Because 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. Therefore, this thesis proposes a method to reduce unnecessary restarts.

Priority-cognizant concurrency control methods based on the optimistic methods have not been widely studied. OCC-APR [13, 14] and its variants attempt to reduce the number of restarts by exploiting and extending the capabilities of priority-insensitive methods. However, experiments and analyses of results indicate that only minor advantages were obtained by incorporating priority cognizance in the validation-phase conflict resolution of optimistic concurrency control methods in firm real-time databases.

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 not only to incorporate information about the importance or criticalness of transactions for conflict resolution but also to design methods that minimize the number of transactions to be restarted.

This thesis focusses on concurrency control in the real-time database systems. A concurrency control method for real-time database systems should be predictable and respect timing constraints as well as maintain database consistency. Therefore, the concurrency control method should not restart unnecessary transactions. Transaction restart causes waste of resources and causes unpredictability.

Clearly some services defined in the IN CS-1 are more important than others [16]. For example, calls to emergency number are most important. If the network is full when a call to an emergency number comes to the switch, one of the normal calls is disconnected. Similarly as in a full network situation emergency calls should not be blocked because of a data conflict in the database. Therefore, new methods are needed for taking importance or criticality of the transactions into account in concurrency control. The main contributions of this thesis are the following:

- To identify an unnecessary restart problem in the OCC-TI (Optimistic Concurrency Control with Timestamp Intervals) [48] optimistic concurrency control method [51]. To present a solution to this problem and demonstrate that the solution will produce a correct result. Additionally, to propose two extensions to the basic conflict resolution method used in the OCC-TI. The proposed OCC-TI method introduces solution to

the unnecessary restart problem and uses priority based conflict resolution. The proposed method is demonstrated to produce correct results and the feasibility of proposed method in practice is tested [51].

- To present a method to reduce the number of transaction restarts and propose a new optimistic concurrency control method, called OCC-DATI (Optimistic Concurrency Control with Dynamic Adjustment of Serialization Order using Timestamp Intervals) [54]. The proposed method is demonstrated to produce a correct results and the feasibility of the proposed method in practice is tested by experiments.

- To propose a new optimistic concurrency control method called OCC-PDATI (Optimistic Concurrency Control using the Importance of the Transactions and Dynamic Adjustment of Serialization Order) [55], which uses information about the criticality of the transactions in the conflict resolution. The main idea behind this method is to offer better chances for critical transactions to complete according to their deadlines. This is achieved restarting transaction with lower criticality if the critical transaction should be restarted because of a data conflict. The proposed method is demonstrated to produce the correct results and the feasibility of the proposed method in practice is tested.

- To propose a new optimistic concurrency control method called OCC-RTDATI (Optimistic Concurrency Control with Real-Time Serializability and Dynamic Adjustment of Serialization Order) [56], which uses criticality of transactions in the conflict resolution. The main idea behind this method is to offer better chances for critical transactions to complete according to their deadlines. This is achieved by restarting transaction with a lower criticality if a critical transaction conflicts with a transaction with lower criticality. The proposed method is demonstrated to produce a correct results and the feasibility of proposed method in practice is tested.

- A *Real-Time Object-Oriented Database Architecture for Intelligent Networks* (RODAIN) [53, 39, 60] project group in which author was member have developed a prototype real-time database system to test the proposed methods in practice. RODAIN is an architecture for a real-time, object-oriented, fault-tolerant, and distributed database management system. RODAIN consists of a main-memory database, priority based real-time scheduling and optimistic concurrency control.

This thesis is organized as follows. Basic concepts in the real-time databases are presented in Chapter 2. Transaction execution on the real-time databases is discussed in Chapter 3.

Classification of the concurrency control methods for real-time databases and recent work on optimistic methods are presented in Chapter 4. The proposed optimistic methods are presented in Chapter 5. Chapter 6 presents RODAIN architecture and experiment configuration and results. Finally, Chapter 7 summarizes the thesis.

Chapter 2

Real-Time Database Systems

A real-time system consists of a *controlling system* and a *controlled system* [68]. 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 essential that the state of the environment is consistent with the actual state of the environment to a high degree of accuracy. Otherwise, the actions of the controlling systems may be disastrous. 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 [17].

This chapter discusses the characteristics of data and characteristics of transactions in real-time database systems.

2.1 Data and Consistency

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 surface because of the need to make data available to the controlling system for its decision-making activities [18]. The need to maintain consistency between the actual state of the environment and the state as reflected by the contents of the database leads to the notion of *temporal consistency*. Temporal consistency has two components [73]:

- *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*: Different data items that are used to derive new data must be temporally consistent with 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 [68]. 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. These timestamps are used in the concurrency control method to ensure that the transaction reads only from committed transactions and writes after the latest committed write. Formally,

Definition 2.1.1 *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 form a relative consistency set R. Each such set R is associated with a *relative validity interval*. Assume that $d \in R$. d has a correct state if and only if [68]:

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 $(current\_time - d_{observationtime}) \leq d_{absolutevalidityinterval}$.
   - Data items are relatively consistent if and only if $\forall d' \in R | d_{timestamp} - d'_{timestamp} | \leq R_{relativevalidityinterval}$.

2.2 Transactions in Real-Time Database System

In this section, transactions are characterized along three dimensions; the manner 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].

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.

Definition 2.2.1 *A transaction T_i is partial order with an ordering relation $\prec_i$ where [8]:*

1. $T_i \subseteq \{r_i[x], w_i[x] \mid$ *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 operation are executed to the same data item, then the ordering relation defines the order between these operations.

A *real-time transaction* is a transaction with additional real-time attributes. We have added additional attributes for a *real-time transaction*. These attributes are used by the real-time scheduling algorithm and concurrency control method. 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 used 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. Following values coded to numbers are used:

  1. **soft**: The transaction is not essential but should be completed if the execution history is serializable.
  2. **firm**: The transaction is critical and should not be restarted if there is data conflict with the transaction with soft criticality.
  3. **hard**: Transaction is very critical and should not be restarted even if there is data conflict with the transaction with soft or firm criticality.

- The timestamp interval (TI) [48] defines the possible time interval when the transactions can be committed, is denoted as $TI(T_i)$, and initialized as $[0, \infty[$.

2.3 Characterization of Real-Time Transactions

The real-time database system apply all three types of transactions discussed in the database literature [7]:

- *Write-only transactions* obtain the state of the environment and write into the database.
- *Update transactions* derive a new data item and store it in the database.
- *Read-only transactions* read data from the database and transmit that data or derived actions based on that data to the controlling system.

The above classification can be used to tailor the appropriate concurrency control methods [37]. 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.

2.4 Problems in the Real-Time Database System

Transaction processing and concurrency control in a real-time database system should be based on priority and criticalness of the transactions [79]. 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 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 [48].

Chapter 3

Transaction Processing in the Real-Time Database System

This section presents several characteristics of transaction and query processing. Transactions and queries have time constraints attached to them and there are different effects of not satisfying those constraints [68]. A key issue in transaction processing is *predictability* [77]. If a real-time transaction misses its deadline, it can have catastrophic consequences. Therefore, it is necessary to able to *predict* that such transactions will complete before their deadlines. This prediction will be possible only if the worst-case execution time of a transaction and the data and resource needs of the transaction is known.

In a database system, several sources of unpredictability exist [68]:

- Dependence of the transaction's execution sequence on data values
- Data and resource conflicts
- Dynamic paging and I/O
- Transactions abort and the resulting rollbacks and restarts
- Communication delays and site failures on distributed databases

Because a transaction's execution path can depend on the values of the data items it accessed, it may not be possible to predict the worst-case execution time of the transaction. Similarly,

it is better to avoid using unbounded loops and recursive or dynamically created data structures in real-time transactions. Dynamic paging and I/O unpredictability can be solved by using main memory databases [2]. Additionally, I/O unpredictability can be decreased using deadlines and priority-driven I/O controllers (e.g. [4, 76]).

Transaction rollbacks also reduce predictability. Therefore, it is advisable to allow a transaction to write only to its own memory area and after the transaction is guaranteed to commit write the transaction's changes to the database [48].

3.1 Scheduling Real-Time Transactions

A *transaction scheduling policy* defines how priorities are assigned to individual transactions [1]. The goal of transaction scheduling is that as many transactions as possible will meet their deadlines. Numerous transaction scheduling policies are defined in the literature. Only a few examples are quoted here.

Transactions in a real-time database can often be expressed as *tasks* in a real-time system [1]. Scheduling involves the allocation of resources and time to tasks in such a way that certain performance requirements are met. A typical real-time system consists of several tasks, which must be executed concurrently. Each task has a *value*, which is gained to the system if a computation finishes in a specific time. Each task also has a *deadline*, which indicates a time limit, when a result of the computing becomes useless.

In this chapter, the terms *hard*, *soft*, and *firm* are used to categorize the transactions [1]. This categorization tells the *value* imparted to the system when a transaction meets its deadline. In systems which use priority-driven scheduling algorithms, value and deadline are used to derive the priority [22, 83].

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 [9]. Therefore, the criticality of the transactions is used in place of the deadline in choosing the appropriate value to priority. This avoid the dilemma of priority scheduling, yet integrates

criticality and deadline so that not only the more critical transactions meet their deadlines. The overall goal is to maximize the net worth of the executed transactions to the system.

Whereas arbitrary types of value functions can be associated with transactions [10, 25, 35], the following simple functions occur more often (see also Figure 3.1):

- *Hard* deadline transactions 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. [57, 44]).
- *Soft* deadline transactions have some value even after their deadlines. Typically, the value drops to zero at a certain point past the deadline (e.g. [28, 36]).
- *Firm* deadline transactions impart no value to the system once their deadlines expire, i.e., the value drops to zero at the deadline (e.g. [12, 25]).

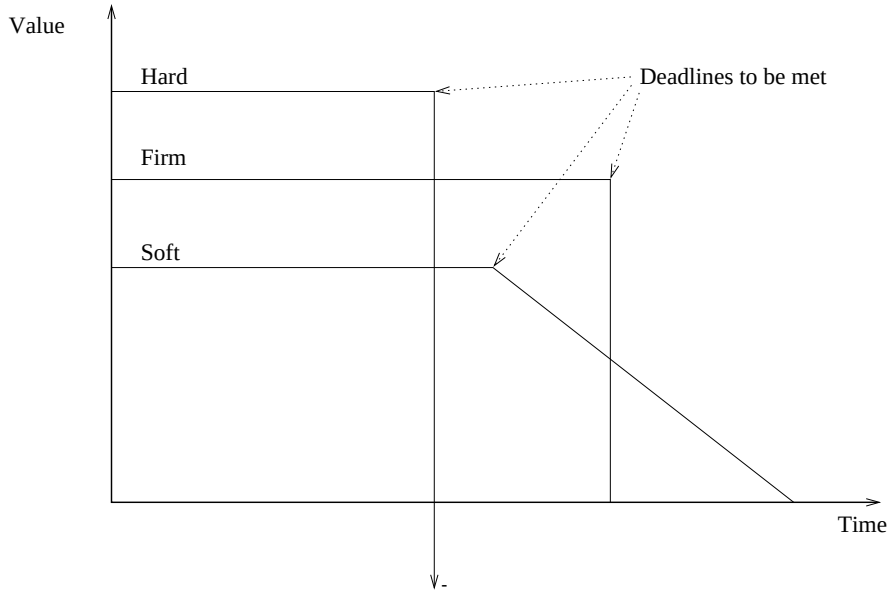


Figure 3.1: The deadline types.

3.2 Scheduling Paradigms

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 [69]:

- 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.
- 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 *earliest deadline first* (EDF) [57] policy, the transaction with the earliest deadline has the highest priority. Other transactions will receive their priorities in descending deadline order. In the *least slack first* (LSF) [1] policy, the transaction with the shortest slack time is executed first. The slack time is an estimate of how long the execution of a transaction can be delayed and still meet its deadline. In the *highest value* (HV) [22] policy, transaction priorities are assigned according to the transaction value attribute. A survey of transaction scheduling policies can be found in [1].

3.3 A Hybrid Scheduling Algorithm

Most real-time scheduling algorithms offer either a real-time behavior to transactions or fairness between transactions [9]. These aspects often conflict with each other. To fulfill telecommunication database requirements, conflicting transaction types must be scheduled simultaneously in the same database. Short real-time transactions must be completed due to their deadlines as well as long normal transactions must get enough resources to complete.

Therefore, a new scheduling algorithm, called *FN-EDF* [53], is designed to support simultaneous execution of both firm real-time and non-realtime transactions. Similar work has been done in [82]. In this method, firm deadline transactions are scheduled according to the EDF scheduling policy [57]. The FN-EDF algorithm guarantees that non-realtime transactions will receive a prespecified amount of execution time.

The FN-EDF algorithm periodically samples the execution times of all transactions and of non-realtime transactions. The operating system scheduling priority of a non-realtime transaction is adjusted if its fraction of execution time is either above or below the prespecified target value. For each class of non-realtime transactions the target value is given as a system parameter, which can be changed while the system is running.

When a firm real-time transaction is started, a priority is assigned to it using the EDF method. When a non-realtime transaction is started, the transaction is assigned an initial priority at the lower end of the priority range.

In implementation, the scheduling of transaction processes is based on the FIFO scheduling policy provided by the Chorus operating system [64]. A continuous range of priorities is reserved for transactions. For firm deadline transactions the priority is assigned once and subsequent transactions receive priorities based on previous assignments. For non-realtime transactions priorities are assigned independently. When a non-realtime transaction is started, it receives the lowest priority of the priority range. During adjustment phases the priority is raised until the transaction has received the deferred fraction of execution time.

It is assumed that non-realtime transactions are relatively long when compared to transactions with a firm deadline. The priority adjustment procedure guarantees that any non-realtime transaction will receive the prespecified fraction in the long run. However, when a new non-realtime transaction is started, the priority is usually adjusted several times before it reaches a stable value. A slow start method is used to prevent a new non-realtime transaction to over-consume computing resources at the beginning.

An important aspect in priority calculation of non-realtime transactions is, how fine is the time granularity provided by the underlying operating system. In telecommunications the firm deadline transactions are typically very short. Therefore, time granularity that is too coarse may lead to inaccurate results in calculation of the fraction of execution time. In addition, the length of the sampling period is important. When the sampling period is too short, the sampling process increases the system overhead but the calculated fractions may also be inaccurate. If the sampling period is too long, there is a possibility that firm transactions do not meet their deadline due to delayed priority correction.

Prototype database system is built over Chorus real-time micro-kernel [64] on the Pentium processor. Implementation can offer a CPU time at the granularity of approximately 1 millisecond. Sampling period of 5 seconds is used, which offers enough calculation resolution but the transaction execution time is prolonged due to the slow start scheme.

3.4 Priority Inversion

In a real-time database environment resource control may interfere with CPU scheduling [3]. When blocking is used to resolve a resource allocation such as in 2PL [15], a *priority inversion* [2] event can occur if a higher priority transaction gets blocked by a lower priority transaction.

Figure 3.2 illustrates an execution sequence, where a priority inversion occurs. A task T_3 executes and reserves a resource. A higher priority task T_1 pre-empts task T_3 and tries to allocate a resource reserved by task T_3. Then, 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.

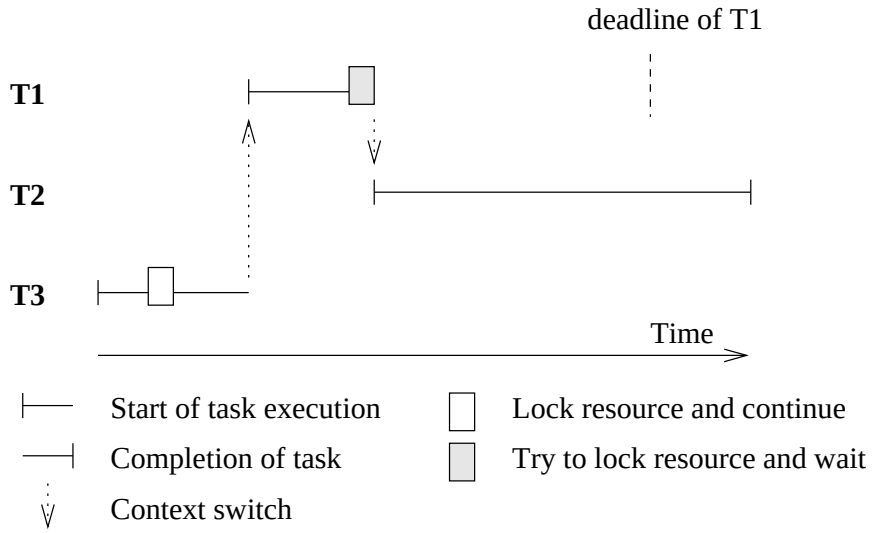


Figure 3.2: Priority inversion example.

In [71], 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 transaction inherits the highest priority among conflicting transactions.

Figure 3.3 illustrates, how a priority inversion problem presented in figure 3.2 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.

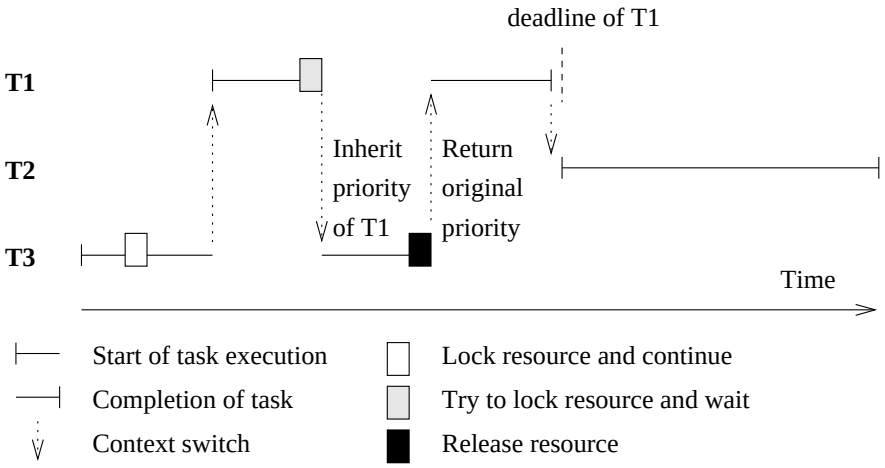


Figure 3.3: Priority inheritance example.

Chapter 4

Concurrency Control in Database Systems

A major aim in developing a database system is to allow many users to access shared data concurrently [8]. Concurrent access is easy if all users are only reading data, because there is no way that they can interfere with one another. However, when two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies [63].

Although two transactions may be correct in themselves, the interleaving of operations may produce an incorrect result, thus compromising the integrity and consistency of the database [8, 19, 63]. The **ACID** properties of a transaction that all transactions should have are [21]:

- **Atomicity**: A transaction is an atomic unit that is either performed completely or not performed at all.
- **Consistency**: A transaction must transform the database from one consistent state to another consistent state.
- **Isolation**: Transactions execute independently of each other. Therefore, the partial effects of incomplete transaction should not be visible to other transactions.
- **Durability**; The effect of a successfully completed and committed transaction is permanently stored in the database and must not be lost because of a later failure.

The aim of a concurrency control method is to schedule transactions in such a way as to avoid any interference [8]. 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 parallel [63].

4.1 Correctness and Serializability

When two or more transactions execute concurrently, their database operations execute in an interleaved way [8]. This means that operations from one transaction may execute in 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 [8].

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 [8]. Therefore, an execution is *serial* if for every pair of transactions all of the operations of one transaction execute before any of the operations of the other. Serial executions are correct because each transaction is assumed to be correct and transactions that execute serially cannot interfere with one another [63].

We can extend the class of correct executions to include executions that have the same effect as serial executions [8]. Such executions are called serializable. Execution is *serializable* 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 [63].

To ensure correctness in the presence of failures, the execution of transactions should not be only serializable but also recoverable, avoid cascading aborts, and strict [8]. 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 [8].

4.2 Classification of Concurrency Control Methods

A scheduler is a program or collection of programs that controls the concurrent execution of transactions [8]. The scheduler restricts the order in which transactions operations are executed. The goal of the scheduler is to order these operations so that the out-coming execution is serializable and recoverable. The scheduler may also ensure that the execution avoids cascading aborts or execution is strict [8].

There are many ways in which the schedulers can be classified [8]. One obvious classification criterion is the mode of database distribution. Some schedulers 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. 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. 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. Therefore, the basic classification is as follows [62]:

- **Pessimistic Methods**
  - Locking Methods
  - Timestamp Ordering Methods
  - Hybrid Methods
- **Optimistic Methods**
  - Locking Methods
  - Timestamp Ordering Methods

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 innter consistency. This ordering is maintained by assigning timestamps to both the transactions and the data times that are stored in the database [62].

4.3 Concurrency Control in Real-Time Databases

A *Real-Time Database Systems* (RTDBS) processes transactions with timing constraints such as deadlines [68]. 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, 24, 26, 29, 45, 42, 59]). Most of these methods are based on one of the two basic concurrency control mechanisms: *locking* [15] or *optimistic concurrency control* (OCC) [41].

In real-time systems transactions are scheduled according to their priorities [68]. 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 traditional 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 and related work in this area is presented.

4.4 Locking Methods in Real-Time Databases

2PL High Priority In the 2PL-HP (2PL High Priority) concurrency control method [2, 5, 27] 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 and a deadlock situation.

2PL Wait Promote In 2PL-WP (2PL Wait Promote) [3, 27] the analysis of concurrency control method is enhanced 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 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 care 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) [27].

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_h 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) [27].

Priority Ceiling Protocol In Priority Ceiling Protocol [70, 71] the aim is to minimize the duration of blocking to at most one elementary 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 [70, 71]. 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 [70, 71]. A new transaction can preempt a lock-holding transaction 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 alerted, 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 [70, 71].

Read/Write Priority Ceiling The Priority Ceiling Protocol is further worked out in [72], 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.

4.5 Optimistic Methods in Real-Time Databases

Optimistic Concurrency Control (OCC) [20, 41], is based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without delays to ensure

serializability. When a transaction wishes to commit, a check is performed to determine whether a conflict has occurred. 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. In some methods updates are applied to a local copy of the data and announced to the database system by an operation named *pre-write*.
- *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 updates, the validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained.
- *Write phase*: This follows the successful validation phase for update transactions. During the write phase, all changes made by the transaction are permanently stored into the database.

Broadcast Commit For RTDBSs, a variant of the classical concurrency control method is needed. In Broadcast Commit, OPT-BC [24], 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 already committed transactions since the committing transaction would have been restarted in the event of a conflict. This means that 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 chances of meeting the transaction deadlines [24].

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 [24]. 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 chances 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 [24]. 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, like in the conventional optimistic methods [24]. 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 the missing of the deadline of the restarted higher priority transaction [23].

OCCL-CMT In [26] a lock-based optimistic concurrency control method, OCCL-CMT, is proposed. The general conflict resolution policy works exactly as in the OPT-BC. The validating transaction is committed and all the other conflicting transactions are aborted. The physical implementation of OCCL-CMT is based on locking. The system maintains read and write locking sets for each transaction and a systemwide lock table, which is shared by all concurrently running transactions. Read-phase (R-lock) and validation-phase lock (V-lock) modes are used. An R-lock is incompatible with a V-lock and a V-lock is incompatible with another V-lock. If the R-lock is not granted, the transaction is blocked in the read phase of its execution. In the Validation phase, the resolution policy applied determines which of the transactions conflicting on V-locks are aborted. In OCCL-CMT [26], all other transactions in conflict with the validating transaction are restarted.

OPT-SACRIFICE In the OPT-SACRIFICE [23] method, when a transaction reaches its validation stage, 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. First, 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. Second, if priority fluctuations are allowed, there may be the mutual restarts problem between a pair of transactions. These two drawbacks are analogous to those in the 2PL-HP method [23].

OPT-WAIT and WAIT-X When a transaction reaches its validation stage, it checks if any of the concurrently running other transactions have a higher priority. In the OPT-WAIT [23] 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 [23].

The WAIT-50 [23] method is an extension of the OPT-WAIT - it contains the priority wait mechanism from 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 [23].

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.

In [26] a lock based WAIT-50 concurrency control method, OCCL-PW, is presented. The physical implementation of this method uses locks. If the priority of the validating transaction is not highest among the conflicting transactions, the validating transaction waits if at least 50% of the conflicting transactions have higher priority.

4.6 Validation Methods

The validation phase ensures that all the committed transactions have executed in a serializable fashion [41]. Every validation scheme uses the following principles to ensure serializability. If a transaction T_i is before transaction T_j in the serialization graph, the following two conditions must be satisfied [48]:

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

Generally, condition 1 is automatically ensured in most optimistic methods because I/O operations in the write phase are required to be done sequentially in the critical section [48]. 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 [48]:

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

In *Backward Validation* [20], the validating transaction is checked for conflicts against (recently) committed transactions. Data conflicts are detected by comparing the read set of the validating transaction and the write set 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 [41] is based on this validation process.

In *Forward Validation* [20], 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 [20]. Most of the proposed optimistic methods are based on forward validation, because it is easy to implement.

In real-time database systems, data conflicts should be resolved in favor of the higher priority transactions. Forward Validation provides flexibility for conflict resolution [20]. Either the validating transaction or the conflicting active transactions may be chosen to restart. Therefore it is preferable for the real-time database systems. 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.

They incorporate different priority conflict resolution methods in the validation phase of a transaction.

The major performance problem with optimistic concurrency control methods are the heavy restart overheads, wasting a large amount of resources [48]. Sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occured, even though it has not done so in actual execution [74].

Forward Validation (OCC-FV) [20] 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 [48]. Therefore, one important mechanism to improve the performance of an optimistic concurrency control method is to reduce the number of restarted transactions. In traditional optimistic concurrency control methods many transactions are unnecessarily restarted. To give an example, let us 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 [20], 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 [8].

One way to reduce the number of transaction restarts is to dynamically adjust the serialization order of the conflicting transactions [48]. Such methods are called *dynamic adjustment of the serialization order* [48]. 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 can be dynamically defined.

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 [48, 50, 74]:

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).

4.7 OCC-TI

The OCC-TI [48, 47] 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 $[0, \infty[$, 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 (figure 4.1) 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 (figure 4.1). If another transaction has read or written the object outside the timestamp interval, the transaction must be restarted.

```
read(T_i, D_i)
{
    TI(T_i) = TI(T_i) ∩ [WTS(D_i), ∞[ ;

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

    read(D_i);
}

pre-write(T_i, D_i)
{
    TI(T_i) = TI(T_i) ∩ [WTS(D_i), ∞[ ∩ [RTS(D_i), ∞[ ;

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

Figure 4.1: Read algorithm and pre-write algorithm for the OCC-TI.

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 [48].

At the beginning of the validation phase (figure 4.2), 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 $TS(T_v)$ [48, 47]. The timestamp intervals of all active concurrently running and conflicting transactions must be adjusted so as 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 $TS(T_v)$ and in the active transaction $TS(T_a)$, the timestamp interval of the $TS(T_a)$ is adjusted.

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

     for ∀ D_i ∈ ( RS(T_v) ∪ WS(T_v) )
     {
          for ∀ T_a ∈ active_conflicting_transactions()
          {
               if D_i ∈ ( WS(T_a) ∩ RS(T_v) ) then
                    TI(T_a) = TI(T_a) ∩ [TS(T_v), ∞[ ;

               if D_i ∈ ( RS(T_a) ∩ WS(T_v) ) then
                    TI(T_a) = TI(T_a) ∩ [0, TS(T_v) − 1];

               if D_i ∈ ( WS(T_a) ∩ WS(T_v) ) then
                    TI(T_a) = TI(T_a) ∩ [TS(T_v), ∞[ ;

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

          if D_i ∈ RS(T_v) then RTS(D_i) = max(RTS(D_i), TS(T_v));

          if D_i ∈ WS(T_v) then WTS(D_i) = max(WTS(D_i), TS(T_v));
     }

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

Figure 4.2: Validation algorithm for the OCC-TI.

4.8 OCC-DA

OCC-DA [42] is based on the Forward Validation scheme [20]. 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 committed transactions which have been defined after the transaction in the serialization order.

When the validating transaction T_v comes to 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 (figure 4.3). 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

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.

```
part_one(T_v)
{
    if  SOT(T_v) ≠ ∞  then
    {
        for  ∀ D_p ∈ RS(T_v)
        {
            if  TR(T_v, D_p) > SOT(T_v)  then
                restart(T_v);
        }

        for  ∀ D_p ∈ WS(T_v)
        {
            if  SOT(T_v) < RTS(D_p)  or  SOT(T_v) < WTS(D_p)  then
                restart(T_v);
        }
    }
}
```

Figure 4.3: The first part of the validation algorithm for OCC-DA.

The purpose of part two of the validation phase is to detect read-write conflicts between the active transactions and the validating transactions (figure 4.4). The write set of T_v is compared with the read sets of the active transactions 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.

```
part_two(T_v)
{
     BTlist(T_v) = ∅ ;

     for ∀ T_j ∈ ATS(T_v)
     {
          for ∀ D_p ∈ WS(T_v)
          {
               if D_p ∈ RS(T_j) then
                    BTlist(T_v) = BTlist(T_v) ∪ T_j ;
          }
     }
}
```

Figure 4.4: The second part of the validation algorithm for OCC-DA.

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 (figure 4.5). 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 has serious conflict with T_v. In conflict resolution transactions for restart are selected based on priorities.

```
part_tree(T_v)
{
     for ∀ T_j ∈ BTS(T_v) ∪ BTlist(T_v)
     {
          for ∀ D_p ∈ RS(T_v)
          {
               if D_p ∈ WS(T_j) then
                    conflict_resolution(T_v, T_j);
          }

          for ∀ D_p ∈ WS(T_v)
          {
               if D_p ∈ WS(T_j) then
                    conflict_resolution(T_v, T_j);
          }
     }
}
```

Figure 4.5: The third part of the validation algorithm for OCC-DA.

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 (figure 4.6).

```
part_four(T_v)
{
    if  SOT(T_v) = ∞  then
        SOT(T_v) = validation_time ;

    for  ∀ T_j ∈ BTlist(T_v)
        SOT(T_j) = SOT(T_v) − ε ; //infinitesimal quantity

    for  ∀ D_p ∈ RS(T_v)
        RTS(D_p) = SOT(T_v) ;

    for  ∀ D_p ∈ WS(T_v)
        WTS(D_p) = SOT(T_v) ;

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

Figure 4.6: The fourth part of the validation algorithm for OCC-DA.

4.9 Evaluation

As presented earlier, the major performance problem with optimistic concurrency control methods are the heavy restart overhead, wasting a large amount of resources [48]. Sometimes the validation process using the read sets and write sets erroneously concludes that a nonserializable execution has occured, even though it has not in actual execution [74].

As presented earlier, OCC-BC restarts all active conflicting transactions. OPT-SACRIFICE restarts the validating transaction if at least one of the conflicting transactions has a higher priority. OPT-WAIT-X family restarts the active conflicting transactions. Therefore, 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 [48] which was presented in Section 4.7. The same validation method but different implementation is used in the OCC-DA method [42]. 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 [48] 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 [48]. 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 [42] have shown that OCC-DA outperforms OCC-TI. OCC-DA can be extended to use Thomas's write rule [81] and this extension is presented in [43].

However, the algorithms provided for OCC-TI in [48, 47] do not seem to fully resolve the unnecessary restart problem. The problem with the existing algorithm is best described by the example given below (a similar example can be found in [51]). Prewrite operation of the transaction T_i to the data item x is marked as $pw_i[x]$.

Example 4.9.1 *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: $r_1[x]r_2[x]pw_1[x]c_1$.

Transaction T_1 executes $r_1[x]$, which causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [0, \infty[\cap [100, \infty[= [100, \infty[$. 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 $pw_1[x]$, which causes the timestamp interval of the transaction to be forward adjusted to $TI(T_1) = [100, \infty[\cap [100, \infty[\cap [100, \infty[= [100, \infty[$. 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 $TI(T_1) = [100, \infty[\cap [0, 99] = []$. The timestamp interval is shut out, and T_1 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 4.9.2 *Let all objects timestamp be initialized as 100. Consider transactions T_1, T_2, T_3, T_4, and history H_1:*

T_1: $r_1[x]$ $w_1[x]$ c_1
T_2: $r_2[x]$ $r_2[y]$ $w_2[y]$ c_2
T_3: $r_3[z]$ $w_3[y]$ c_3
T_4: $r_4[z]$ $w_4[z]$ c_4
H_1: $r_1[x]$ $r_4[z]$ $pw_4[z]$ c_4 $r_2[x]$ $r_3[z]$ $r_2[y]$ $pw_3[y]$ $pw_2[y]$ c_3 c_1 c_2.

All operations before the first commit operation are executed successfully. Let us consider the execution of the c_1 operation. Assume that transaction T_1 arrives to its validation phase at time 600. In this case the set $BTS(T_1)$ is empty and the set $ATS(T_1) = \{T_2, T_3, T_4\}$. Because $x \in WS(T_1) \cap RS(T_2)$, the active conflicting transaction T_2 is added to the backward set, i.e. $BTlist(T_1) = \{T_2\}$. In the part three of the OCC-DA validation algorithm there are no detectable conflicts. In the part four of the OCC-DA validation algorithm $SOT(T_1) = 600$ and $SOT(T_2) = 599$ are set. The validation of the transaction T_1 is completed and the write set of the validating transaction (i.e. $WS(T_1)$) is committed to the database.

Let us consider then the execution of the c_4 operation. Assume that transaction T_4 arrives to its validation phase at time 700. In this case the set $BTS(T_4) = \{T_2\}$ and the set $ATS(T_4) = \{T_3\}$. Because $z \in WS(T_4) \cap RS(T_3)$, the active conflicting transaction T_3 is added to the backward set, i.e. $BTlist(T_4) = \{T_3\}$. In the part three of the OCC-DA validation algorithm there are no detectable conflicts. In the part four of the OCC-DA validation algorithm $SOT(T_4) = 700$ and $SOT(T_4) = 699$ are set. The validation of the transaction T_4 is completed and the write set of the validating transaction (i.e. $WS(T_4)$) is committed to the database.

Let us consider then the execution of the c_3 operation. Assume that transaction T_3 arrives to its validation phase at time 800. In this case the set $ATS(T_3) = \{T_2\}$ and the set $BTS(T_3) = \emptyset$. Because $y \in WS(T_3) \cap RS(T_2)$, the active conflicting transaction T_2 is added to the backward set, i.e. $BTlist(T_3) = \{T_2\}$. In the part three of the OCC-DA validation algorithm a conflict is detected, because $y \in WS(T_3) \cap WS(T_3)$. Therefore, the transaction T_2 or the transaction T_3 must be restarted. However this restart is unnecessary, because serialization graph $SG(H_1)$ is acyclic, that is, history H_1 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, an unnecessary restart problems found in OCC-TI and OCC-DA are not desirable.

Therefore, in the next chapter will propose a new optimistic concurrency control method to reduce number of unnecessary restarts. Additionally, methods to incorporate priority or criticality information to conflict resolution are presented.

Chapter 5

Proposed Optimistic Concurrency Control Methods

This chapter proposes new optimistic concurrency control methods for firm real-time database systems. As discussed earlier, the major problem with optimistic concurrency control methods is unnecessary restarts.

Therefore, the first section presents a solution to unnecessary restart problem found in OCC-TI (Optimistic Concurrency Control with Timestamp Intervals) [48] and demonstrate that proposed solution will produce the correct result. Additionally, two extensions to the basic conflict resolution method used in OCC-TI will be proposed. The proposed OCC-TI method introduces a solution to the unnecessary restart problem and uses priority-based conflict resolution. The proposed method is demonstrated to produce the correct result and tested by experiments. This proposal is based on earlier work presented in [51].

The second section presents an optimistic concurrency control method called OCC-DATI (Optimistic Concurrency Control with Dynamic Adjustment of Serialization Order using Timestamp Intervals) [54] to reduce the number of transaction restarts. The presented method is demonstrated to solve unnecessary restart problems found in OCC-TI and OCC-DA. The proposed method is demonstrated to produce a serializable history and the proposed method is tested in practice.

Traditional real-time concurrency control methods use priority in the conflict resolution of the validation phase check. As presented earlier transactions with very short deadlines might not be very critical. Therefore, two optimistic concurrency control methods are proposed

where criticality of the transactions is used in the conflict resolution of the validation phase check.

Firstly, a new optimistic concurrency control method will be proposed, called OCC-PDATI (Optimistic Concurrency Control using Importance of the Transactions and Dynamic Adjustment of Serialization Order) [52], which uses information about the criticality of the transactions in the conflict resolution. The main idea behind this method is to offer better changes for critical transactions to complete according to their deadlines. This is achieved by restarting the transaction with lower criticality if a critical transaction should be restarted because of data conflict. The proposed method is demonstrated to produce the correct result and the feasibility of the proposed method in practice is tested by experiments.

Secondly, a new optimistic concurrency control method will be proposed, called OCC-RTDATI (Optimistic Concurrency Control with Real-Time Serializability and Dynamic Adjustment of Serialization Order) [56], which uses criticality of the transactions in the conflict resolution. The main idea behind this method is to offer better changes for critical transactions to complete according to their deadlines. This is achieved by restarting transaction with the lower criticality if the critical transaction conflicts with the transaction with lower criticality. The proposed method is demonstrated to produce the correct result and the feasibility of the proposed method in practice is tested.

5.1 Revised OCC-TI

Firstly, as showed in section 4.7 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 OCC-TI algorithm. An extension to OCC-TI algorithm is proposed in this section to solve these problems. This thesis includes the following extensions to 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 continuous. Formally,

Definition 5.1.1 *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:*

$\forall x \forall y((x \in TI(T_i) \wedge y \in RTI_n(T_i)) \wedge$
$\nexists z(z \in ([0, \infty[\backslash (TI(T_i) \cup RTI_n(T_i)) \ \wedge \ (x < z < y) \ \vee (y < z < x)))$. $\square$

Example 5.1.1 *Let $TI(T_1) = [100, 1000]$, $RTI_1(T_1) = [0, 100]$, and $RTI_2(T_1) = [1002, 2000]$. Using definition 5.1.1, 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 $\exists z(z \in [o, \infty[\ \wedge \ (1000 < z < 1002))$ e.g. $z = 1001$. Thus the removed timestamp interval $RTI_2(T_1)$ cannot be rollbacked. When checking $RTI_1(T_1)$ for rollbacking it can see that definition 5.1.1 holds and we can rollback the removed timestamp interval* $\square$.

The next definition 5.1.2 shows how dynamically adjusted timestamp intervals can be rollbacked.

Definition 5.1.2 *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 5.1.2 *Let $TI(T_1) = [100, 1000]$ and $RTI_1(T_1) = [0, 100]$. Then rollbacking is done with*

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

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 5.1.3 *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 5.1.1. Using definition 5.1.2 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 larger timestamp interval:

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

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 removed timestamp intervals. Therefore, only two timestamp intervals are actually stored in the data structure. The current timestamp interval value of the active transaction and temporal timestamp interval value of the active transaction during the validation phase of the another transaction. The temporal timestamp interval value is copied to actual value when the validating transaction is guaranteed to commit. Thus, no unnecessary rollbacking is done.

5.1.2 Prioritized Dynamic Adjustment of Serialization Order

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. 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 changes 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 chances 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 higher priority transaction would become empty. In the forward ordering final timestamp can be moved backward if there is space in the timestamp interval of the validating transaction. Again, it is checked if 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 5.1.4 *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*:*

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

Example 5.1.5 *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:*

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

5.1.3 Revised OCC-TI Algorithm

In this section the validation algorithm for extended OCC-TI is presented. OCC-TI is extended with a new final timestamp selection method and priority-depended 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 (Figure 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 ∈ 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 ∀ D_i ∈ ( RS(T_v) ∪ WS(T_v)){

        /* Iterate conflicting active transactions */
        for ∀ T_a ∈ active_conflicting_transactions() {
            if (D_i ∈ ( RS(T_v) ∩ WS(T_a)))
                forward_adjustment(T_a, T_v, adjusted);

            if (D_i ∈ ( WS(T_v) ∩ RS(T_a)))
                backward_adjustment(T_a, T_v, adjusted);

            if (D_i ∈ ( WS(T_v) ∩ WS(T_a)))
                forward_adjustment(T_a, T_v, adjusted);

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

        // Adjust conflicting transactions
        for ( ∀ T_a ∈ adjusted)
        {
            TI(T_a) = adjusted.pop(T_a);

            if (T_a.backward == true )
                TI(T_a) = TI(T_a) ∩ [0, TS(T_v) − 1];

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

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

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

Figure 5.1: Revised validation algorithm for the OCC-TI.

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

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

  if ( priority(T_v) >= priority(T_a))
      TI = TI ∩ [TS(T_v), ∞];
  else {
       if (((min(TI(T_v)) + TS(T_v)) / 2) ∈ 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 ∩ [TS(T_v), ∞];
       }
       else {
            if ( TS(T_v) > max(TI)) restart(T_v);

            TI = TI ∩ [TS(T_v), ∞];
       }
  }

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

}

backward_adjustment(T_a, T_v, adjusted) {
  if (T_a ∈ 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)});

}
```

Figure 5.2: Backward and Forward adjustment for the reviced OCC-TI.

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.

Having described the basic concepts and the algorithm, the correctness of the algorithm is now proven. To prove that a history H produced by revised OCC-TI is serializable, it must be proven that the serialization graph for H, denoted by $SG(H)$, is acyclic [8].

Lemma 5.1.1 *Let T_1 and T_2 be committed transactions in a history H produced by the revised OCC-TI algorithm. If there is an edge $T_1 \rightarrow T_2$ in $SG(H)$, then $TS(T_1) < TS(T_2)$.*

Proof: *Since there is an edge, $T_1 \rightarrow T_2$ in $SG(H)$, 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 implies that T_1 commits before T_2 reaches its validation phase since $r_1[x]$ is not affected by $w_2[x]$. For $w_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $RTS(x)$ that is equal to or greater than $TS(T_1)$. That is, $TS(T_1) \leq RTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.*

2. *$w_1[x] \rightarrow r_2[x]$: This case implies that the write phase of T_1 finishes before $r_2[x]$ is executed in T_2's read phase. For $r_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. That is, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.*

3. *$w_1[x] \rightarrow w_2[x]$: This case implies that the write phase of T_1 finishes before $w_2[x]$ is executed in T_2's write phase. For $w_2[x]$, OCC-TI adjusts $TI(T_2)$ to follow $WTS(x)$, which is equal to or greater than $TS(T_1)$. That is, $TS(T_1) \leq WTS(x) < TS(T_2)$. Therefore, $TS(T_1) < TS(T_2)$.* □

Theorem 5.1.1 *Every history generated by the revised OCC-TI algorithm is serializable.*

Proof: *Let H denote any history generated by the revised OCC-TI algorithm. Suppose, by way of contradiction, that $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 5.1.1 $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. This is a contradiction. Therefore no cycle can exist in $SG(H)$ and thus the algorithm produces only serializable histories.* □

The same example history as in Example 4.9.1 in Section 4.9, which caused unnecessary restart in the original OCC-TI. Using the same example, it is shown here how the revised OCC-TI produces a serializable history and avoids unnecessary restart.

Example 5.1.6 *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 revised OCC-TI produces serializable histories and avoids the unnecessary restart problem found in the original OCC-TI algorithm □.

5.2 OCC-DATI

This section presents an optimistic concurrency control method named OCC-DATI [54]. OCC-DATI is based on forward validation [20]. 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 [49]. 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 [42] 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 serialization method is proposed, called *deferred dynamic adjustment of serialization order*. In the deferred dynamic adjustment of serialization order all adjustments of timestamp interval are done to temporal variables. The timestamp interval 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. OCC-TI and OCC-DA both use dynamic adjustment but they make unnecessary adjustments when the validating transaction is aborted.

OCC-DATI offers greater changes to successfully validate transactions 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.

With the new method, the number of transaction restarts is smaller than with OCC-BC, OPT-WAIT, or WAIT-50 [23, 24, 26], 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 the OCC-DA in worst case. Therefore, the overhead for supporting dynamic adjustment is much smaller in OCC-DATI than the one in OCC-DA [42].

The OCC-DATI method resolves conflicts using the timestamp intervals [48] 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 (Figure 5.3), 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 $TI(T_v)$.

```
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 ( ∀ D_i ∈ (RS(T_v) ∪ WS(T_v) ))
  {
    if (D_i ∈ RS(T_v)) // read from committed transactions
      TI(T_v) = TI(T_v) ∩ [WTS(D_i), ∞[ ;
    if (D_i ∈ WS(T_v)) // write after committed transactions
      TI(T_v) = TI(T_v) ∩ [WTS(D_i), ∞[ ∩ [RTS(D_i), ∞[ ;
    // if timestamp interval is empty, then restart the validat-
ing transaction
    if (TI(T_v) == []) restart(T_v);

    // Conflict checking and timestamp interval calculation
    for ( ∀ T_a ∈ active_conflicting_transactions() )
    {
      if (D_i ∈ (RS(T_v) ∩ WS(T_a)))
        forward_adjustment(T_a, T_v, adjusted);

      if (D_i ∈ (WS(T_v) ∩ RS(T_a)))
        backward_adjustment(T_a, T_v, adjusted);

      if (D_i ∈ (WS(T_v) ∩ WS(T_a)))
        forward_adjustment(T_a, T_v, adjusted);
    }
  }

  // Adjust conflicting transactions
  for ( ∀ T_a ∈ adjusted)
  {
    TI(T_a) = adjusted.pop(T_a);

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

  // Update object timestamps
  for ( ∀ D_i ∈ (RS(T_v) ∪ WS(T_v) ))
  {
    if (D_i ∈ RS(T_v))
      RTS(D_i) = max(RTS(D_i), TS(T_v));
    if (D_i ∈ WS(T_v))
      WTS(D_i) = max(WTS(D_i), TS(T_v));
  }

  commit T_v to database;
}
```

Figure 5.3: Validation algorithm for the OCC-DATI.

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.

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.

Figure 5.4 shows a sketch of implementing 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 ∈ adjusted)
    TI = adjusted.pop(T_a);
  else
    TI = TI(T_a);

  // Forward adjustment
  TI = TI ∩ [TS(T_v) + 1, ∞[ ;
  // 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 ∈ adjusted)
    TI = adjusted.pop(T_a);
  else
    TI = TI(T_a);

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

Figure 5.4: Backward and Forward adjustment algorithms for the OCC-DATI.

Firstly, let us consider the Example 4.9.1 as in Section 4.9 which was used to show unnecessary restart problem in OCC-TI.

Example 5.2.1 *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 to its validation phase at time 1000. 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,

OCC-DATI algorithm produces a serializable history as well as avoiding the unnecessary restart problem found in OCC-TI. □

Secondly, let us consider the Example 4.9.2 as in Section 4.9 which was used to show unnecessary restart problem in OCC-DA.

Example 5.2.2 *Let all object timestamp be initialized as 100. Consider transactions* T_1, T_2, T_3, T_4, *and history* H_1*:*

T_1: $r_1[x]$ $w_1[x]$ c_1
T_2: $r_2[x]$ $r_2[y]$ $w_2[y]$ c_2
T_3: $r_3[z]$ $w_3[y]$ c_3
T_4: $r_4[z]$ $w_4[z]$ c_4

H_1: $r_1[x]$ $r_4[z]$ $pw_4[z]$ c_4 $r_2[x]$ $r_3[z]$ $r_2[y]$ $pw_3[y]$ $pw_2[y]$ c_3 c_1 c_2.

All operations before the first commit operation are executed successfully. Let us consider the execution of the c_1 *operation. At this point* $TI(T_1) = TI(T_2) = TI(T_3) = TI(T_4) = [0, \infty[$. *Assume that transaction* T_1 *arrives to its validation phase at time 600. Because* $TI(T_1) = [0, \infty[$, *the final timestamp is the validating time, i.e.* $TS(T_1) = 600$.

Because $x \in RS(T_1) \cap WS(T_1)$, *the validating transaction* T_1 *is forward adjusted to be* $TI(T_1) = TI(T_1) \cap [WTS(x), \infty[\cap [RTS(x), \infty[= [100, \infty[$. *Because* $x \in WS(T_1) \cap RS(T_2)$, *the active transaction* T_2 *is backward adjusted to be* $TI(T_2) = TI(T_2) \cap [0, TS(T_1) - 1] = [0, 599]$. *The validation of the transaction* T_1 *is completed and the write set of the validating transaction (i.e.* $WS(T_1)$*) is committed to the database.*

Let us consider then the execution of the c_4 *operation. Assume that transaction* T_4 *arrives to its validation phase at time 700. Because* $TI(T_4) = [0, \infty[$, *the final timestamp is the validating time, i.e.* $TS(T_4) = 700$. *Because* $z \in RS(T_4) \cap WS(T_4)$, *the validating transaction* T_4 *is forward adjusted to be* $TI(T_4) = TI(T_4) \cap [WTS(z), \infty[\cap [RTS(z), \infty[= [100, \infty[$. *Because* $z \in WS(T_4) \cap RS(T_3)$, *the active transaction* T_3 *is backward adjusted to be* $TI(T_3) = TI(T_3) \cap [0, TS(T_4) - 1] = [0, 699]$. *The validation of the transaction* T_4 *is completed and the write set of the validating transaction (i.e.* $WS(T_4)$*) is committed to the database.*

Let us consider then the execution of the c_2 operation. Assume that transaction T_2 arrives to its validation phase at time 800. Because $TI(T_2) = [0, 599]$, the final timestamp is the maximum value of the timestamp interval, i.e. $TS(T_2) = 599$. Because $y \in RS(T_2) \cap WS(T_2)$, the validating transaction T_2 is forward adjusted to be $TI(T_1) = TI(T_1) \cap [WTS(x), \infty[\cap[RTS(x), \infty[= [100, \infty[$. The validating transaction T_3 have not observed the change of $WTS(x)$, and $RTS(x)$ because these values where fetched before the changes. Because $y \in WS(T_2) \cap WS(T_3)$, the active transaction T_3 is forward adjusted to be $TI(T_3) = TI(T_3) \cap [TS(T_2) + 1, \infty[= [600, 699]$. The validation of the transaction T_2 is completed and the write set of the validating transaction (i.e. $WS(T_2)$) is committed to the database.

Finally, let us consider the execution of the c_3 operation. Assume that transaction T_3 arrives to its validation phase at time 900. Because $TI(T_2) = [600, 699]$, the final timestamp is the maximum value of the timestamp interval, i.e. $TS(T_3) = 699$. The validating transaction T_3 have not observed the change of $WTS(z)$, $RTS(y)$, and $WTS(y)$ because these values where fetched before the changes. Therefore, timestamp interval of the validating transaction is not empty and no conflicts are detectable. 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. Thus, OCC-DATI algorithm produces a serializable history as well as avoiding the unnecessary restart problem found in OCC-DA. □

Previous examples show that OCC-DATI method avoids unnecessary restart problems found in OCC-TI and OCC-DA. 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-DA is serializable, it must be proven that the serialization graph for H, denoted by $SG(H)$, is acyclic [8]. Therefore, following Lemma demonstrate 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 5.2.1 *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 *commits before* T_2 *reaches its validation phase since* $r_1[x]$ *is not affected by* $w_2[x]$. *For* $w_2[x]$, *OCC-DATI adjusts* $TI(T_2)$ *to follow* $RTS(x)$ *that is equal to or greater than* $TS(T_1)$. *Thus,* $TS(T_1) \leq RTS(x) < TS(T_2)$. *Therefore,* $TS(T_1) < TS(T_2)$.

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. 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. 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 algorithm will produce cycle in the serialization graph. This case is shown to cause contradiction in following theorem.

Theorem 5.2.1 *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 5.2.1* $TS(T_1) < TS(T_2) < ... < TS(T_n) < TS(T_1)$. *By induction* $TS(T_1) < TS(T_1)$. *This is a contradiction. Therefore no cycle can exist in* $SG(H)$ *and thus the OCC-DATI algorithm produces only serializable histories.* □

5.3 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 equalized 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.

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 critical transaction were aborted, then the transaction of lower criticalness should be restarted.

This section proposes an optimistic concurrency control method called OCC-PDATI (Optimistic Concurrency Control using Importance of the Transaction and Dynamic Adjustment of Serialization Order). OCC-PDATI is based on forward validation [20] and the earlier optimistic method OCC-DATI [54]. The difference is in the conflict resolution. The conflict resolution of OCC-PDATI uses criticalness of the transaction found in transaction object attributes. This section outlines new parts of OCC-PDATI when compared to OCC-DATI.

A critical transaction should not be restarted because of data conflict with a transaction of low criticality. Greater changes for a critical transaction to complete before its deadline should be offered. Therefore, if dynamic adjustment of the serialization order would cause the critical transaction to be restarted, a conflicting active transaction of lower criticality 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 criticality 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.

Figure 5.5 depicts an implementation of a deferred dynamic adjustment of the serialization order using timestamp intervals and information about the criticality of the transactions.

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

   TI = TI(T_a) ∩ [TS(T_v) + 1, ∞[;

    if ( criticality(T_v) < criticality(T_a) )
       if ( TI == ∅ )
         restart(T_v); /* Validation ends here /*

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

backward_adjustment(T_a, T_v, adjusted)
{
   if (T_a ∈ adjusted)
     TI = adjusted.pop(T_a);
   else
     TI = TI(T_a);

   TI = TI(T_a) ∩ [0, TS(T_v) − 1]

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

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

Figure 5.5: Backward and Forward adjustment for the OCC-PDATI.

Theorem 5.3.1 *OCC-PDATI* $\subset$ *OCC-DATI* $\subseteq$ *SERIALIZABLE*

Proof: *It is easy to show that OCC-PDATI produces serializable histories with similar proof as in OCC-DATI case. This implies that OCC-PDATI* $\subseteq$ *OCC-DATI* $\subseteq$ *SERIALIZABLE. The true subset part is also trivial. Consider transactions* T_1 *and* T_2 *such that* $criticality(T_1) > criticality(T_2)$ *and history* $H = r_1[x]w_2[x]c_1$. *Because critical transactions may not be backward adjusted in the OCC-PDATI, the active conflicting lower criticality transaction* T_2 *is restarted. This history is serializable and can be executed using the method OCC-DATI. This implies the clause.* □

5.4 OCC-RTDATI

This section is based on earlier results presented in [56]. In real-time database systems, the conflict resolution of the transactions should be based on the criticality 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 criticality.

This section proposes an optimistic concurrency control method called OCC-RTDATI. OCC-RTDATI is based on forward validation [20] and the earlier optimistic method OCC-DATI [54]. 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 criticality of the transaction. A critical transaction should precede a transaction of lower criticality in the history. Therefore, the critical transaction should not be forward adjusted after a transaction of lower criticality. Thus, if this is the case a transaction of lower criticality 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 criticality should be restarted. This is wasted execution and unnecessary restart, which must be acceptable when critical transactions are favored.

Figure 5.6 depicts an implementation outline of conflict resolution for the OCC-RTDATI.

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

   if (T_a ∈ adjusted)
     TI = adjusted.pop(T_a);
   else
     TI = TI(T_a);

  TI = TI(T_a) ∩ [TS(T_v) + 1, ∞[;
  adjusted.push({(T_a, TI)});
}

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

   if (T_a ∈ adjusted)
     TI = adjusted.pop(T_a);
   else TI = TI(T_a);

  TI = TI(T_a) ∩ [0, TS(T_v) − 1]
  adjusted.push({(T_a, TI)});
}
```

Figure 5.6: Backward and Forward adjustment algorithms for the OCC-RTDATI.

Theorem 5.4.1 *OCC-RTDATI* $\subset$ *OCC-PDATI* $\subset$ *OCC-DATI* $\subseteq$ *SERIALIZABLE*

Proof: *It is easy to show that OCC-RTDATI produces serializable histories with similar proof as in the OCC-DATI case. This implies that OCC-RTDATI* $\subseteq$ *OCC-PDATI* $\subseteq$ *OCC-DATI* $\subseteq$ *SERIALIZABLE. The true subset part is also trivial. Consider transactions* T_1 *and* T_2 *such that* $criticality(T_1) > criticality(T_2)$ *and history* $H = r_1[x]w_2[x]c_1$. *Because critical transactions may not be backward adjusted, active conflicting lower criticality transaction* T_2 *is restarted. This history is serializable and can be executed using the method OCC-DATI. This implies that OCC-RTDATI* $\subseteq$ *OCC-PDATI* $\subset$ *OCC-DATI* $\subseteq$ *SERIALIZABLE. The last part is also trivial. Consider* T_1 *and* T_2 *such that* $criticality(T_1) > criticality(T_2)$ *and history* $H = w_1[x]r_2[x]c_2$. *Because critical transactions may not be forward adjusted in the OCC-RTDATI, the active conflicting lower criticality transaction* T_2 *is restarted. This history is serializable and can be executed using the method OCC-PDATI. This implies the clause.* □

5.5 Implementation Issues

A locking mechanism is used to implement an optimistic method as proposed in [26]. In the selected mechanism the system maintains a system-wide lock table to take care of book keeping data access by all concurrently executing transactions.

Generally, the validation process is carried out by checking the lock compatibility with the lock table. Such locking-based implementation of the validation test is efficient because its complexity does not depend on the number of active transactions. If a serious conflict is found between the validating transaction and an active transaction, the transaction having the lowest value of the *importance* attribute is selected to be restarted.

There are two possible implementations of the write phase: *serial validation-write* (OCCL-SVW) and *parallel validation-write* (OCCL-PVW) [26]. In serial validation-write the validation phase and write phase are in one critical section. Parallel validation-write separates the validation phase and the write phase into two critical sections, thus providing greater concurrency. In this study we use the parallel validation-write because implementation is based on a multiprocess server.

Chapter 6

Experiments

A set of experiments have been carried out in order to examine the feasibility of the algorithms in practice. This chapter describes the configuration of the test system, test database and transactions. Several test session have been done, the results of which are presented. It is examined how revisions made to OCC-TI effect on it's performance when compared with original OCC-TI. Revised OCC-TI is also compared to OCC-DA and OCC-DATI.

Also experiments with original OCC-TI, OCC-DA and proposed OCC-DATI will be presented. Additionally, experiments with OCC-DATI, OCC-PDATI, and OCC-RTDATI will be presented. Finally, some scalability test results are presented.

6.1 Database Clusters

In this section the architecture of the RODAIN (Real-Time Object-Oriented Database Architecture for Intelligent Networks) database system is presented. This section is based on work published in [39, 60, 53].

The RODAIN Database Architecture is a hierarchical distributed database architecture. The RODAIN Database Nodes are linked together to form database clusters. Furthermore, the database clusters are then connected together to allow access between different, possibly heterogeneous, distributed databases (Figure 6.1).

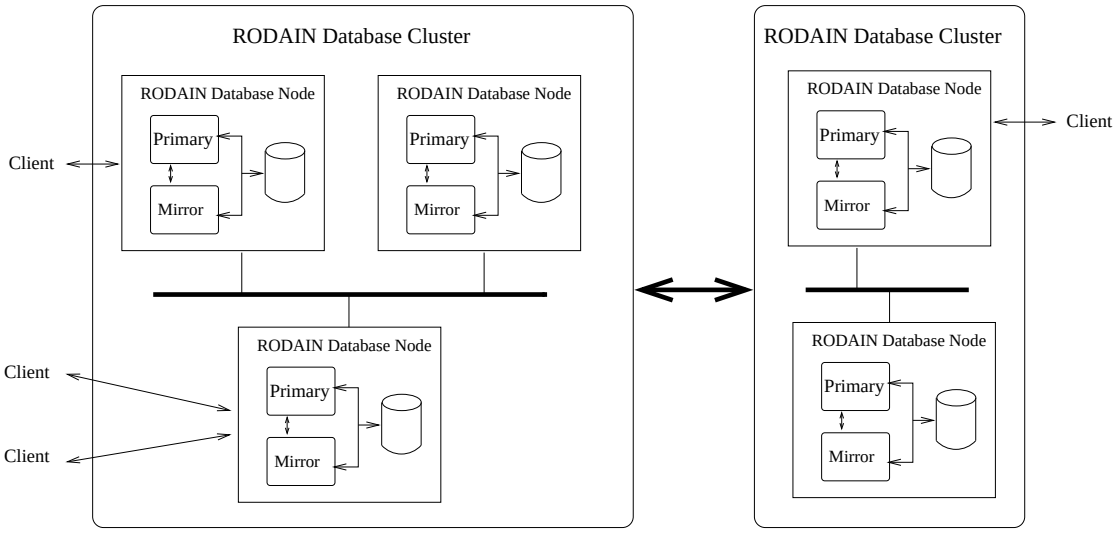


Figure 6.1: RODAIN Database Cluster.

RODAIN Database Nodes within one database cluster share common metadata. They also have access to a global dictionary in order to locate any data item. Different telecommunication applications as clients of RODAIN Database Cluster can access any of the Database Nodes. Each node serving a request can fully hide data distribution by providing access to all data items within the cluster. For most time-critical transactions the RODAIN Database offers the possibility to learn the fastest access point of each time-critical data item.

RODAIN Database Clusters do not necessarily share common metadata. Instead, schema translations may be needed when data items from a remote cluster are accessed. It should also be noted that no assumption of real-time behavior of the remote clusters can be made since the remote cluster usually belongs to a different administration domain. Therefore, we assume that the communication between clusters will be based on standard communication protocols and object access models like the ones used in CORBA [61]. In this way a remote cluster does not have to be a RODAIN Database Cluster. In fact a remote cluster can be any object or relational database.

The data in each Rodain Database Node is divided into two parts; each data item belongs to hot data or to cold data. The data items are stored in different databases in the Rodain Database Node; hot data is stored in a main-memory database and cold data is stored in a disk-based database. All updates in the hot data are done in the main memory. A transaction log of hot data is maintained to keep the database in a consistent state. A secondary copy of hot data is located in a mirror node and only a backup copy is maintained on the disk. Since cold data is stored in a disk-based database a hybrid data management method is used that

combines a main-memory database and a disk-based database.

RODAIN Database Nodes that form a RODAIN Database Cluster are real-time, highly-available, main-memory database servers. They support concurrently running real-time transactions using an optimistic concurrency control protocol with deferred write policy. They can also execute non-real-time transactions at the same time on the database. Real-time transactions are scheduled based on their type, priority, mission criticality, or time criticality.

6.2 Database Nodes

In order to increase the availability of the database each Rodain Database Node consists of two identical co-operative nodes. One of the nodes acts as the Database Primary Node and the other one, Database Mirror Node, is mirroring the Primary Node. Whenever necessary, i.e. when a failure occurs, the Primary and the Mirror Node can switch their roles. When there is only one node in function it is called a Transient Node. A Transient Node behaves like a Primary Node, but it is not accompanied by a Mirror Node and, therefore, it behaves like a stand-alone database system. The role of Transient Node is designed to be temporary and used only during the failure period of the other node.

The Database Primary Node and Mirror Node use a reliable shared Secondary Storage Subsystem (SSS) for permanently storing the cold data database, copies of the hot data database, and log information. The nodes themselves are further divided into a set of subsystems (Figure 6.2) that perform the needed function on both nodes. Below, the function of each subsystem will be shortly be summarized.

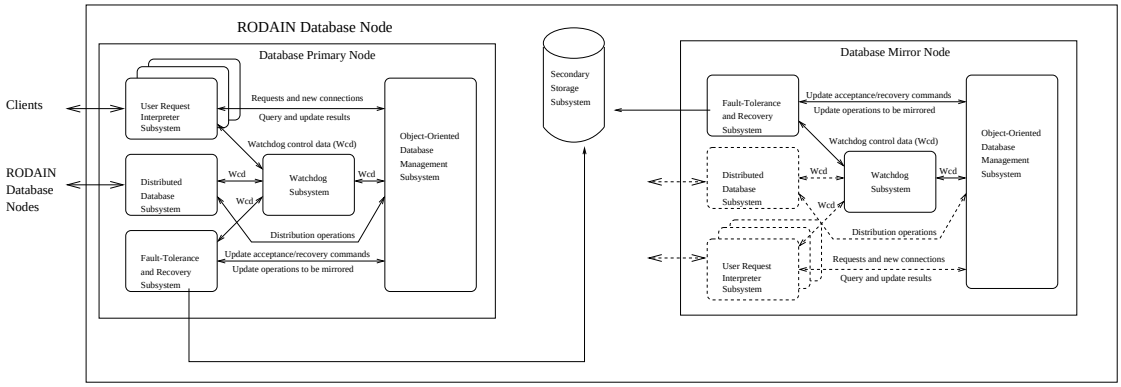


Figure 6.2: Rodain Database Node.

User Request Interpreter Subsystem. The Rodain Database Node can have multiple application interfaces. Each interface is handled by one specific User Request Interpreter Subsystem. It translates its own interface language into a common connection language that the database management subsystem understands. The URISs on the Primary Node are active and communicate with the clients. On the Mirror Node the URISs are not needed.

Distributed Database Subsystem. A Rodain Database Node may either be used as a standalone system or in co-operation with the other autonomous Rodain Database Nodes within one RODAIN Database Cluster. The database co-operation management in the Database Primary Node is left to the Distributed Database Subsystem (DDS). The Distributed Database Subsystem on the Mirror Node is passive or non-existent. It is activated when the Mirror Node becomes a new Primary or Transient Node.

Fault-Tolerance and Recovery Subsystem. The FTRS on both nodes controls communication between the Database Primary Node and the Database Mirror Node. It also co-operates with the local Watchdog Subsystem to support fault tolerance.

The FTRS on the Primary Node handles transaction logs and failure information. It sends transaction logs to the Mirror Node. It also monitors the Mirror Node. When it notices a failure it reports that to the Watchdog Subsystem for changing the role of the node to a Transient Node. On the Transient Node FTRS stores the logs directly to the disk on the SSS.

The FTRS on the Mirror Node receives the logs sent by its counterpart on the Primary Node. It then saves the logs to disk on Secondary Storage Subsystem and gives the needed update instructions to the local Database Management Subsystem. When it notices that the Primary Node has failed, it informs the local Watchdog Subsystem to start the role change.

Watchdog Subsystem. The Watchdog subsystem watches over the other local running subsystems both on the Primary and on the Mirror Node. It implements a subset of watchd [30] service. Upon a failure it recovers the node or the failed subsystem. Most subsystem failures need to be handled like the failure of the whole node. The failure of the whole node requires compensating operations on the other node. On Primary Node, when the failure of the Mirror Node is noticed, the WS controls the node change to the Transient Node. This change mostly affects the FTRS, which must start storing the logs to the disk. On the Mirror Node the failure of the Primary Node generates more work. In order to change the Mirror Node to the Transient Node the WS must activate passive subsystems such as URIS and DDS. The FTRS must change its functionality from receiving logs to saving them to the disk on SSS.

Object-Oriented Database Management Subsystem. The OO-DBMS is the main subsystem on both Primary Node and Mirror Node. It maintains both hot and cold databases. It maintains real-time constraints of transactions, database integrity, and concurrency control. It consists of a set of database processes, that use database services to resolve requests from other subsystems, and a set of manager services that implement database functionality. The Object-Oriented Database Management Subsystem needs the Distributed Database Subsystem, when it can not solve an object request on the local database.

The RODAIN data model is a true superset of the ODMG 2.0 data model [11]. The concepts of real-time extensions are expressed as additions to the ODMG 2.0 type hierarchy and as additional attributes and operations to persistent objects. Extensions are also induced to transaction management. The primary objective of the extensions is to provide sufficient information for scheduling and concurrency control so that the problems due to heterogeneous transactions can reasonably be solved. The RODAIN data model is presented in [40].

6.3 Architecture of the Database Management Subsystem

The Object-Oriented Database Management Subsystem implements the functionality of the RODAIN database. It takes care of storing objects, processing queries, and providing transaction services for URIS. The Database Primary Node executes transactions. The processes in the Database Primary Node are depicted in Figure 6.3. Below we briefly summarize the basic concepts of processes that are relevant when transactions are processed.

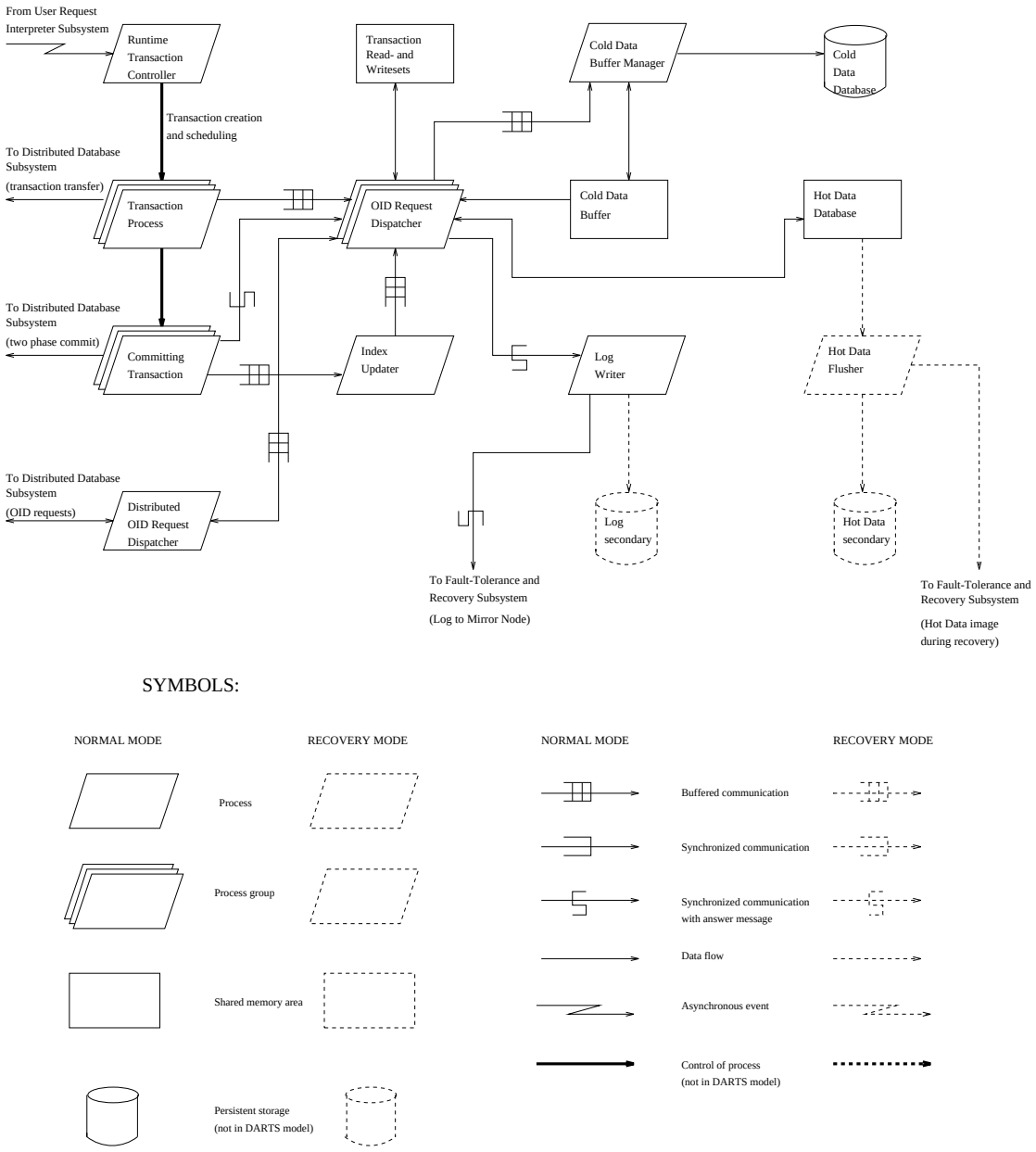


Figure 6.3: Processes in the RODAIN Database Primary Node.

The *Runtime Transaction Controller* (RTC) receives new transaction requests from applications through the URIS. The RTC allocates one *Transaction Process* from the pool of Transaction Processes to serve the incoming transaction. Based on the attribute values of the transaction instance the RTC assigns a priority to the Transaction Process. In the transaction scheduling the RTC adjusts the priorities of each transaction based on the scheduling policy. The RTC also handles transactions that have missed their deadlines according to the over-deadline handling scheme.

A Transaction Process is invoked to execute the request sent by an application. A request is either a request to access an object or an invocation of a method in an object. Transaction Process offers the functionality specified in the RODAIN data model, query optimization, and exploitation of indices in queries. The methods in any object are executed in the memory space of the Transaction Process. A Transaction Process also keeps track of inserted, deleted, and modified objects.

An *OID Request Dispatcher* (ORD) is a process that receives object read and write requests from Transaction Process and executes them. ORD also takes care of validating and committing a transaction. When the transaction has been successfully validated, it enters its committing phase. In this phase, all modifications - insertion, updating and deletion of persistent objects - are written to the database. Log records are also generated by the ORD.

A *Committing Transaction* is a Transaction Process that has entered the commit phase. A Committing Transaction performs the validation of the transaction commit using the read-set and write-set of the transaction. If the validation is successful, all modified objects are written into the database and the transaction is committed.

The *Cold Data Buffer Manager* (CDBM) receives read and write requests to the cold data from ORD. The Log Writer handles log write commands. The *Hot Data Flusher* writes the contents of the main memory database into the Secondary Storage Subsystem.

6.4 Transaction Processing

The transaction flow of a local transaction is presented in Figure 6.4. The flow is shown with numbers that are referred to in the explanations below.

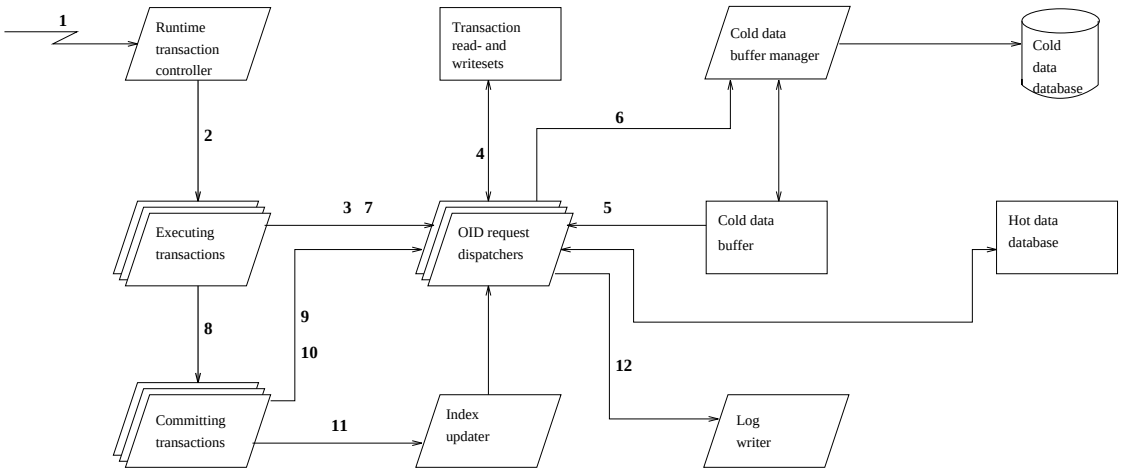


Figure 6.4: Flow of local Transaction.

A transaction starts when the Runtime Transaction Controller receives either an object method call or a *begin_transaction* primitive (1). If the transaction request is accepted, the Runtime Transaction Controller allocates a Transaction Process to execute the transaction and assigns a priority to the Transaction Process (2).

The allocated Transaction Process executes the code of the transaction. When the transaction requires database services, the Transaction Process performs subroutine calls for these services. Complete structures of all accessed objects are known on this level, thus the transaction has free access to all properties of the object. However, the security properties of the object and the classification of the transaction can put some restrictions on accessing the object and on performing operations.

When a Transaction Process needs to access an object in a database, it sends a request to an OID Request Dispatcher (3). The incoming request is accepted by a free OID Request Dispatcher process that starts servicing the request. Communication between the Transaction Process and the OID Request Dispatcher is buffered, so the Transaction Process can send multiple requests into the request queue. The requests may be executed in parallel if there are enough free OID Request Dispatchers. The maximum number of simultaneous requests is a property of the transaction.

At this point of the transaction flow the request can be either a read request or a prewrite request of an object either in the hot data or in the cold data. When the request is a read request, the data is fetched from the main memory database (hot data) or from the Cold Data Buffer (cold data) (5).

In the case of a prewrite request no data is accessed. Both read and prewrite requests lead to appropriate markings in the readset or in the writeset of the transaction (4). When an object in the hot data is accessed, the request is directly fulfilled from the main memory database (Hot Data Database). When an object in the cold data is accessed, the request is first tried to be resolved from the Cold Data Buffer. If the object is not in the buffer, the Cold Data Buffer Manager (CDBM) is requested (6). The CDBM fetches the object into the Cold Data Buffer from the Cold Data Database in the Secondary Storage Subsystem. After the request has been completed, the OID Request Dispatcher sends the result of the request back to the Transaction Process (7).

When a transaction is going to commit, the Transaction Process is moved into the process group of Committing Transactions (8). 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, i.e. to certify whether or not concurrency conflicts have occurred. The validation is done by an OID Request Dispatcher that receives a validate request (9).

During the validation process the OID Request Dispatcher checks whether or not read-write or write-write conflicts exist. If a conflict exists, the conflict is solved either by aborting the committing transaction or the conflicting transaction(s). The resolution depends on the properties of the conflicting transactions. If no conflict exists, the Committing Transaction is marked unabortable.

After a successful validation, the commit procedure continues. The next step in the commit procedure is to send a write request to the OID Request Dispatcher (10) in order to store the prewritten objects permanently into the database. In addition, index update requests are sent through the Index Updater (11). All write requests are passed also to the Log Writer which takes care of permanent storage of the transaction log (12). The communication between the Committing Transaction, the OID Request Dispatcher, and the Log Writer is synchronous. The commit ends when all write requests are successfully processed. The Transaction Process is returned into the pool of free Transaction Processes, and the result is sent to the application.

6.5 Configuration of Test System

The test data is generated off-line. The Transaction Generator reads the data and starts transactions. Transactions are executed by Transaction Processes and scheduled by the Runtime Transaction Controller. All data access is directed to the OID Request Dispatcher that maintains the main memory database and performs concurrency control. After the transaction is completed or aborted, the result status of the transaction is recorded by the Transaction Generator.

The computing system is a Pentium Pro 200 MHz with 64 MB of main memory. The operating system is Chorus/ClassiX. All processes are pre-installed prior to a test session. Every test session contains 10 000 transactions and is repeated at least 20 times. The reported values are means of the repeated runs.

The test database represents a typical Intelligent Network (IN) service, the database schema of which was provided by Telecom Finland. The size of the database is 20 000 objects. Test transactions include both service provision and service management operations. A service provision transaction accesses a few objects and there are no hot spots in the database. A service management transaction scans through the whole database and replaces an attribute value in a large proportion of object instances.

Those services that may affect resolving the physical number are briefly described. The purpose of each class and their relationships are also presented, which are used in resolving the physical number (see Figure 6.5). All these services are related to the redirection of a logical number. These services are:

- **Get Profile**: The Service Control Function finds the profile of the called number. With this the SCF determines the IN service needed to be executed. The class *Profile* includes all necessary information needed for customer management. The most simple transaction will be a read-only transaction *get_profile* that reads customer profile information. It gives the OID of the profile needed and receives the profile information. We use OID instead of the true number because the test database does not yet contain an index structure for secondary keys. Although the duration of the *get_profile* transaction is reduced by not accessing the index, the same reduction in operation affects all other transactions as well. Thus, it is still feasible to include this very simple transaction to the group of test transactions.

- **Customer Profile Management**: This service feature allows the subscriber to manage his or her service profile in real time; for example terminating destinations, announcements to be played, call distribution. The simplest update transaction, called *modify_profile*, is an update that changes one customer's profile information. This transaction implements a customer profile management service.

- **Resolve Abbreviated Number**: This service is an originating line feature that allows business subscribers to dial others in their company using short numbers (e.g. four digits).

  For example, the caller (A-party) dials the extension number of the callee (B-party) and the network connects the call. SDF translates the abbreviated number into an actual destination number. The profile for the originating line includes abbreviated numbers.

  The class *Exchange* includes all necessary information of switches in the system. Attribute ExchangeId presents the identity number of the exchange. Other attributes set maximum lengths for extension numbers, prefixes and for location prefixes.

  The class *PrefixTranslation* is used to give shorter prefix numbers for exchanges. A user may replace the real exchange number with a corresponding prefix number. This allows users to make ”local calls” to other exchanges. Attributes are used to map prefixes to exchange numbers. The relationship is a link to the exchange that has defined this prefix. One exchange may define several prefixes to different exchanges. Definitions, however, are valid only in the defining exchange.

  The read-only transaction, called *translate_number*, implements the redirection of the logical number. It gives the OID of the number to be translated and receives the translated number. The number translation is necessary at least in IN services: call forwarding, conference calls, free phone and televoting.

These are only a subset of the 28 targeted services defined in the *IN Capability Set 1 (IN CS-1)* [32]. A review of all services may be found in [67]. Figure 6.5 presents a test database with an IN schema. It provides data structures for exchanges and extensions, and their services used to reroute a call.

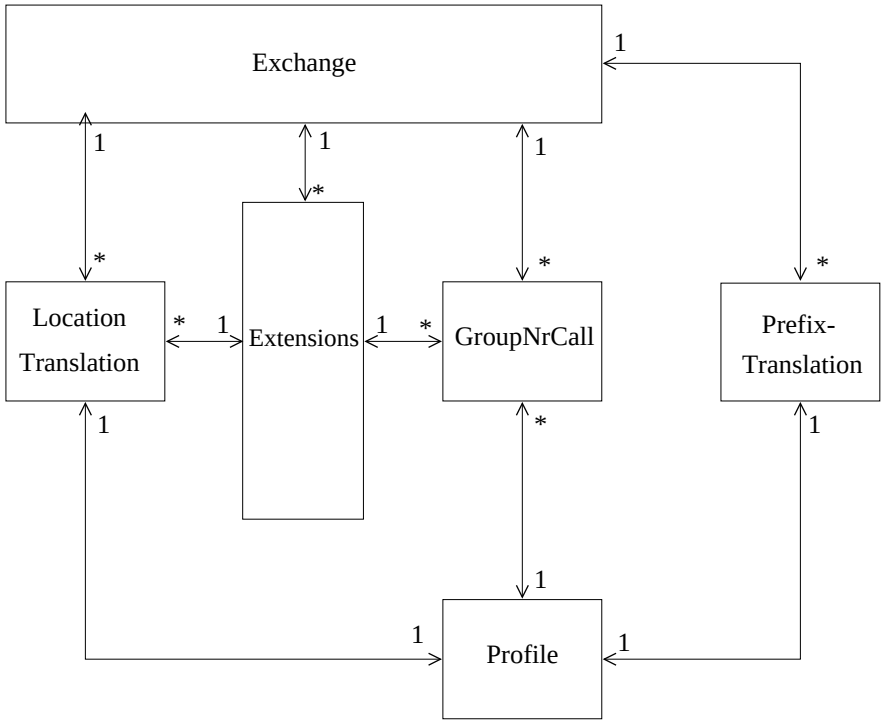


Figure 6.5: Object model schema of test database.

New transactions are accepted up to a pre-specified limit, which is the number of installed Transaction Processes. If there are no Transaction Processes available when a new transaction arrives, the transaction is aborted. Transactions are atomically validated. The order of the *criticality* attribute, which is used in selecting the transaction to be restarted in conflict resolution, is $R1 > W1 > T1$. If the conflicting transactions have the same criticality, then the validating transaction is favored. If the deadline of a transaction expires, the transaction is always aborted.

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, and the proportion of objects accessed in the service management transaction T1 (Table 6.1).

Table 6.1: Transaction test parameters

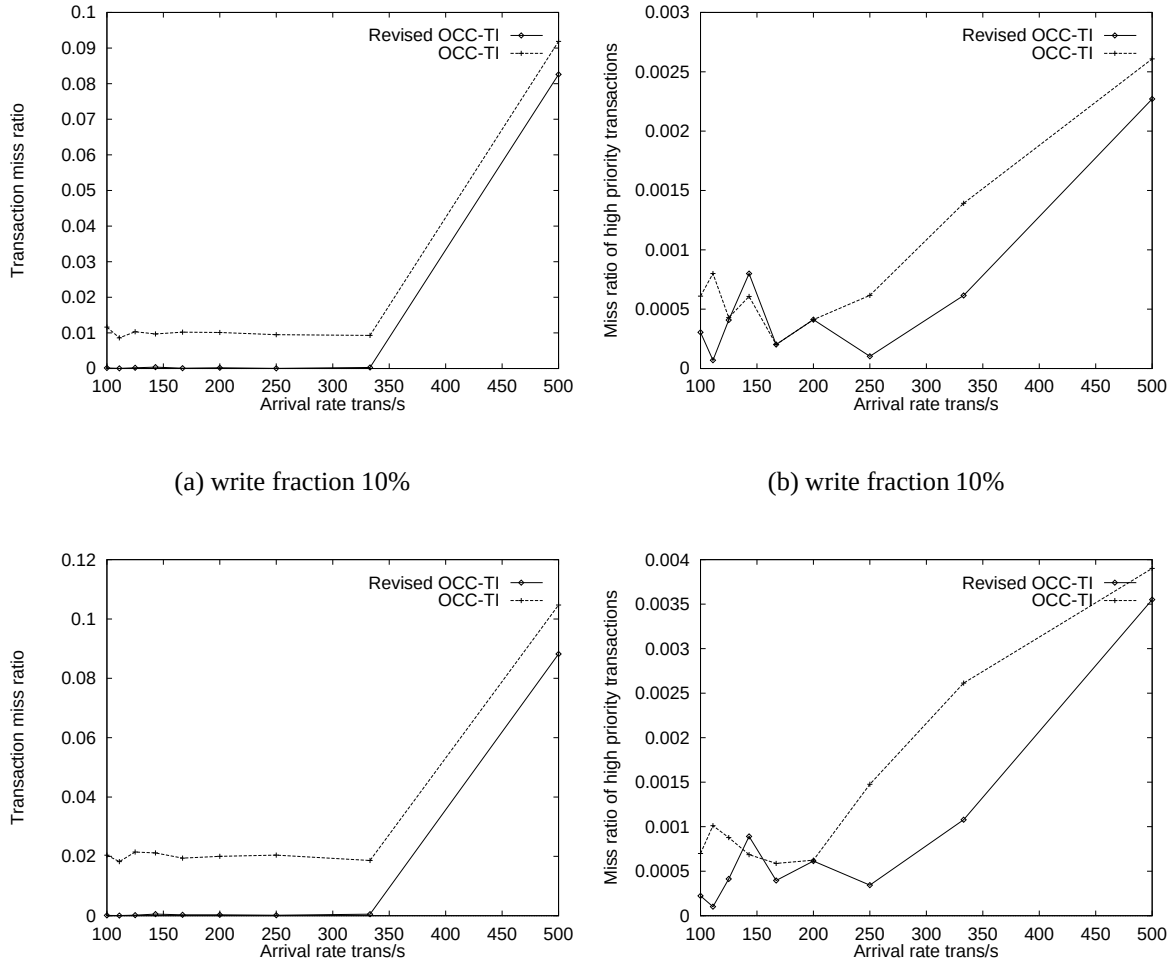
Parameter	Unit	Value	Description
ArrRate	trans/s	100–1000	Average arrival rate of transactions
Deadline	mSec	100	The deadline of a firm transaction
DbSize	num	20000	Number of objects in the database
NonrealFrc	%	5	Fraction of cpu guaranteed to non-realtime transactions
NumTRP	num	50–100	Number of Transaction Processes
WriteProp	%	0–100	Write probability for the service management transaction

Miss ratio performance metrics are reported. The miss-ratio characterizes the fraction of uncompleted transactions. A transaction may fail to complete due to a concurrency control conflict, due to exceeding its deadlines, or due to an overload in the system. The miss-ratio is defined as

$$\text{Miss Ratio} = \frac{\text{Number of uncompleted transactions}}{\text{Number of arrived transactions}} .$$

6.6 Experiments with OCC-TI and Revised OCC-TI

The experiment was run to compare the miss rates of the original OCC-TI and revised OCC-TI. In the experiments, the arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 6.6(a) the fraction of write transactions is 10%. In Figure 6.6(c) the fraction of write transactions is 20%. Figure 6.6 shows that the revised OCC-TI performs better than OCC-TI, especially when the arrival rate is high. This is because the revised OCC-TI does not suffer from the unnecessary restart problem. Finally, Figures 6.6(b) and 6.6(e) shows the miss ratio of transactions of high priority. This demonstrates how the Revised OCC-TI favors transactions of high priority. Revised OCC-TI clearly offers better chances for high priority transactions to complete according to their deadlines. The results clearly indicate that Revised OCC-TI meets the goal of favoring transactions of high priority.



(a) write fraction 10%

(b) write fraction 10%

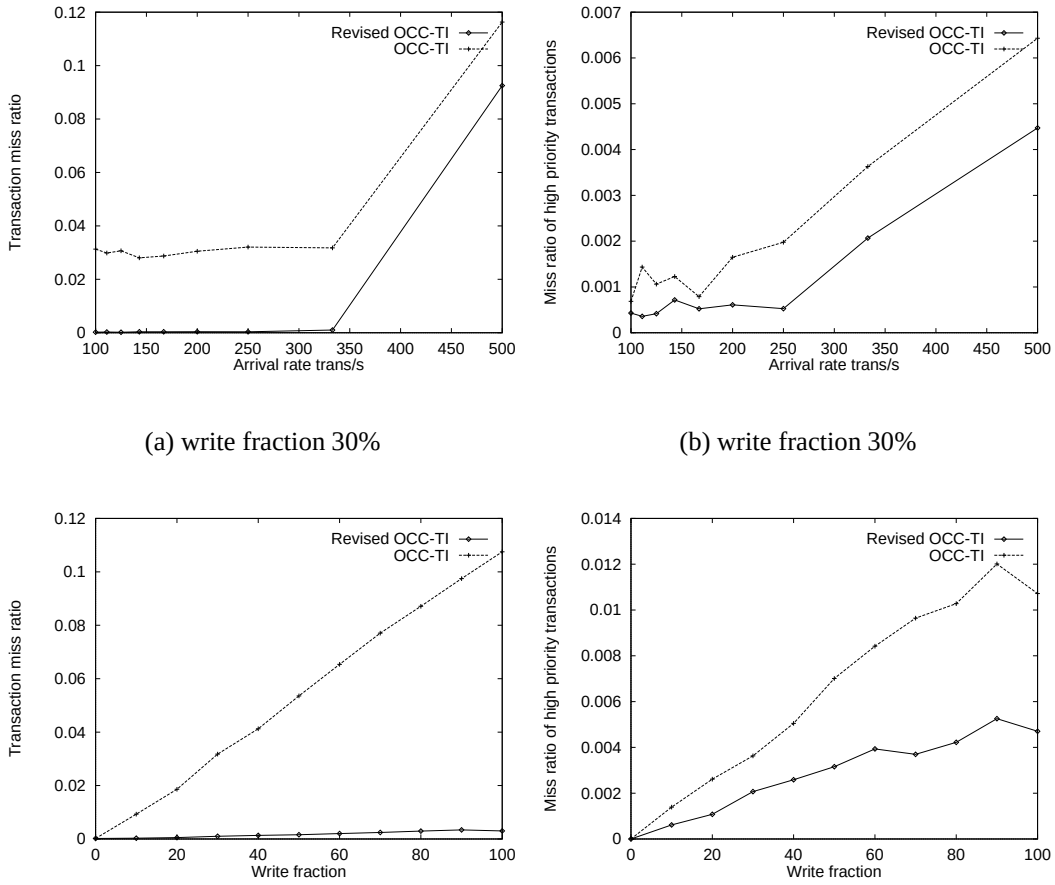
(c) write fraction 20%

(d) write fraction 20%

Figure 6.6: Revised OCC-TI compared to OCC-TI.

In the next experiments, the arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 6.7(a) the fraction of write transactions is 30%. Figure 6.7(b) shows the miss ratio of transactions of high priority when the fraction of write transactions is 30%. Figure 6.7 shows that the revised OCC-TI performs better than the original OCC-TI, especially when the arrival rate is high. This is because the revised OCC-TI does not suffer from the unnecessary restart problem. In Figure 6.7(c) the fraction of write transactions have been varied from 10% to 100% and the arrival rate of the transactions have been fixed to 333 transactions per second. Figure 6.7(d) shows the miss ratio of transactions of high priority. The results demonstrate how the revised OCC-TI outperforms original OCC-TI especially when a fraction of write transactions increases and how OCC-TI favors transactions of high priority. The revised OCC-TI clearly offers better chances for transactions to complete ac-

cording to their deadlines.



(a) write fraction 30%

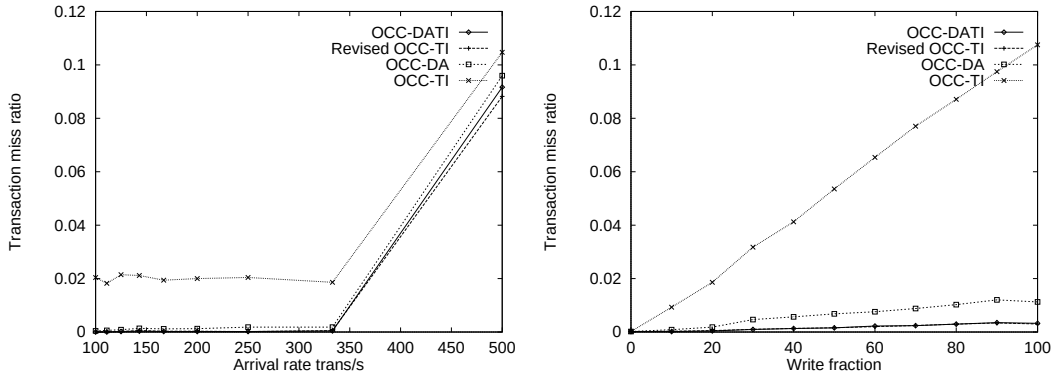
(b) write fraction 30%

(c) arrival rate 333 trans/s

(d) arrival rate 333 trans/s

Figure 6.7: Revised OCC-TI compared to OCC-TI.

In the final experiments, results from OCC-DA and OCC-DATI methods are included. The arrival rate of the transactions is varied from 100 to 500 transactions per second. In Figure 6.8(a) the fraction of write transactions is 20%. The performance of the revised OCC-TI is similar to OCC-DA and OCC-DATI. In Figure 6.8(b) the fraction of write transactions have been varied from 10% to 100% and the arrival rate of the transactions have been fixed to 333 transactions per second. This shows that the performance of the revised OCC-TI is even better than OCC-DA and the same as OCC-DATI. Therefore, the revised OCC-TI is well suited for a firm real-time database system where transactions are heterogeneous.

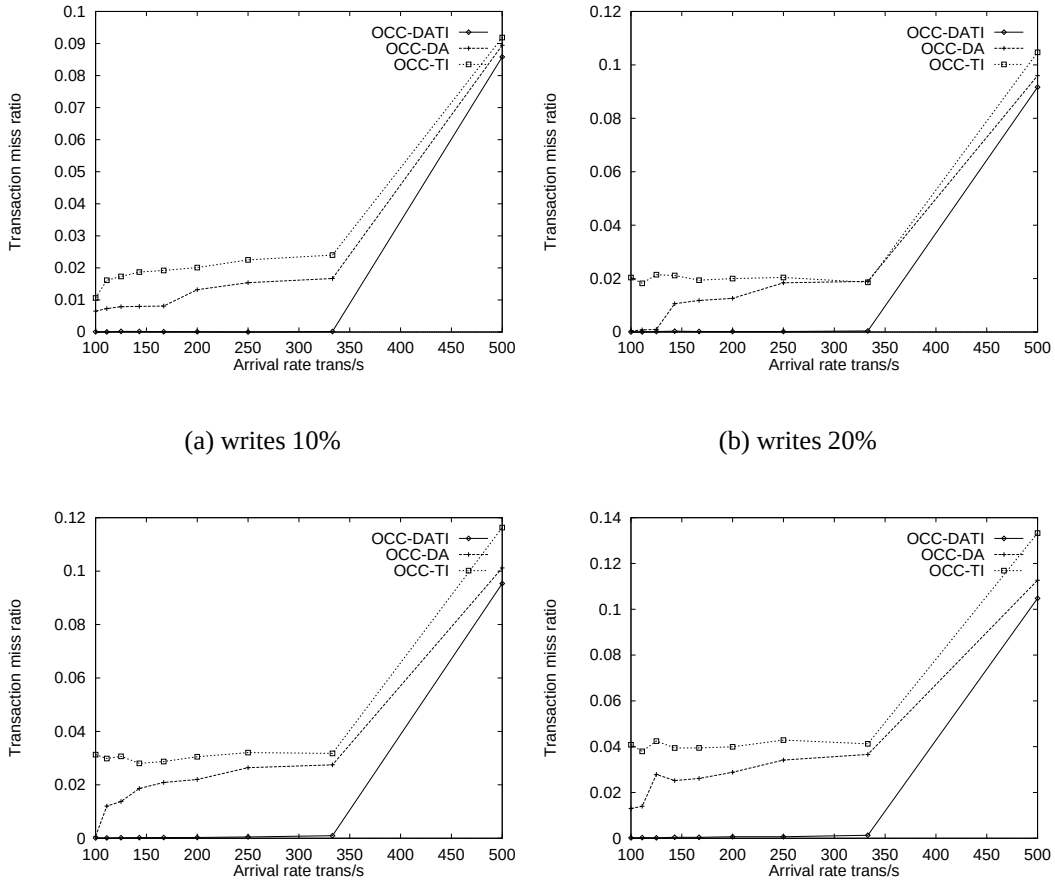


(a) write fraction 30% (b) arrival rate 333 trans/s

Figure 6.8: Optimistic concurrency control methods compared.

6.7 Experiments with OCC-TI, OCC-DA and OCC-DATI

In this sections first experiments a fixed fraction of write transactions have been used, varying the arrival rate of the transactions from 100 to 500 transactions per second (Figure 6.9).



(a) writes 10%

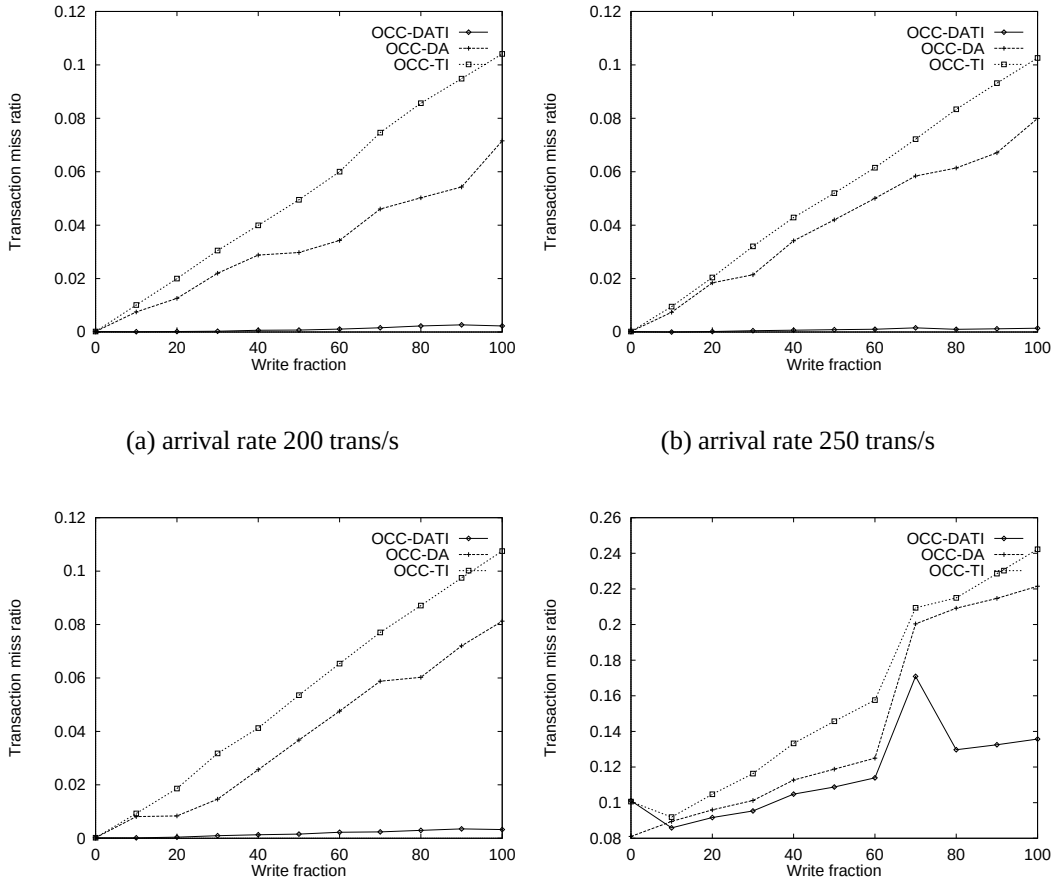
(b) writes 20%

(c) writes 30%

(d) writes 40%

Figure 6.9: OCC-DATI, OCC-TI, and OCC-DA 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 have been used and the fraction of write transactions (W1) have been varied from 10% to 100%. The rest are read transactions (R1). In Figure 6.10(a) the arrival rate of the transactions is 200 transactions per second, in Figure 6.10(b) the arrival rate of the transactions is 250 transactions per second, in Figure 6.10(c) the arrival rate of the transactions is 333 transactions per second, and in Figure 6.10(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

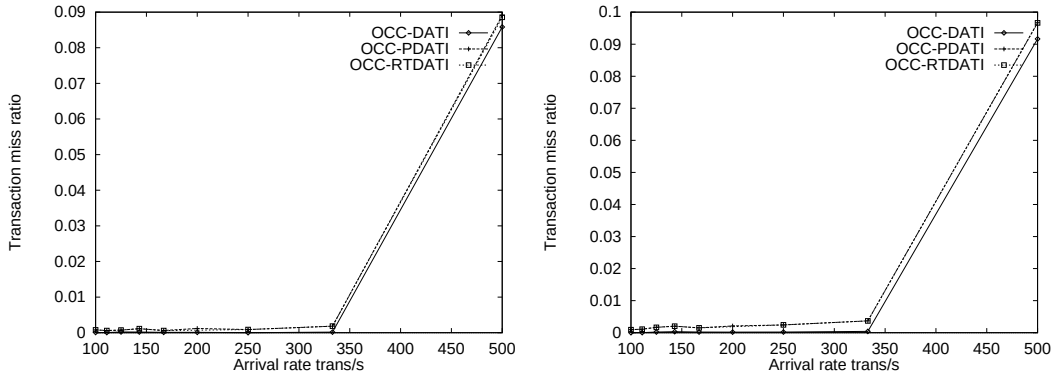
(d) arrival rate 500 trans/s

Figure 6.10: OCC-DATI, OCC-TI, and OCC-DA compared when the fraction of the write transactions is varied from 10% to 40% 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 [42] that OCC-DA outperform OCC-TI. The results [48] have already confirmed that OCC-TI outperforms 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.

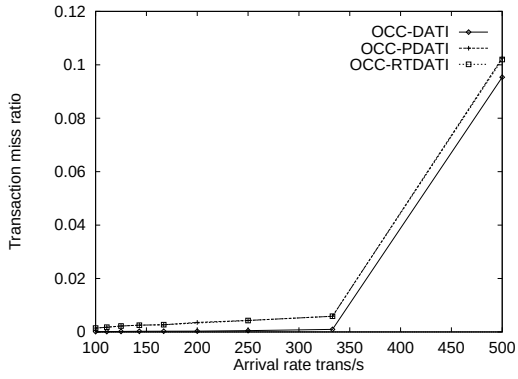
6.8 Experiments with OCC-DATI, OCC-PDATI and OCC-RTDATI

In this section results from experiments with OCC-DATI, OCC-PDATI and OCC-RTDATI methods are presented. In the first experiments a fixed fraction of write transactions have been used and the arrival rate of the transactions have been varied from 100 to 500 transactions per second (Figure 6.11).

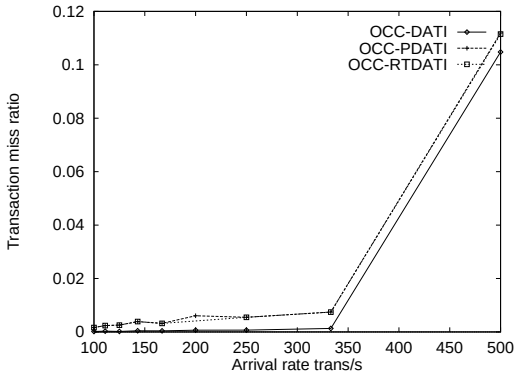


(a) writes 10%

(b) writes 20%



(c) writes 30%



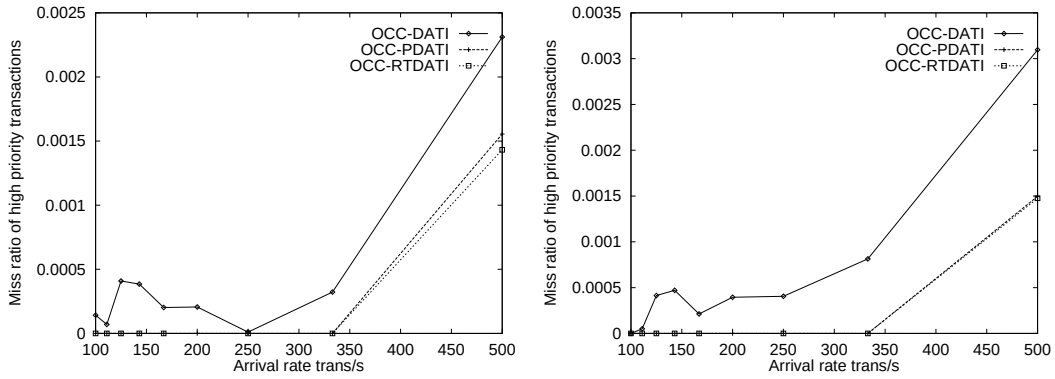
(d) writes 40%

Figure 6.11: 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%.

As expected, there is some overhead when information about the importance of the transaction is used in dynamic adjustment of serialization order. As Figure 6.11 indicates, the overhead using additional information from the transactions is quite low. The miss ratio of

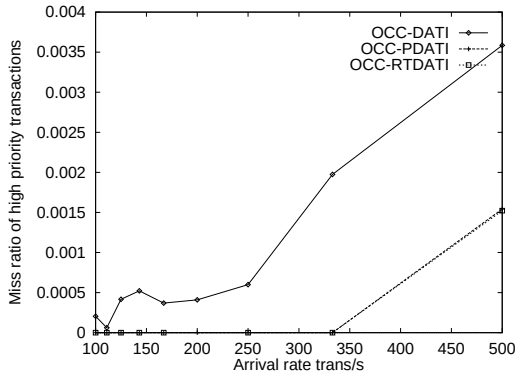
the transactions when using the OCC-PDATI or OCC-RTDATI algorithm is only slightly higher than in the OCC-DATI.

Finally, Figure 6.12 shows the miss ratio of transactions of high importance. Figure 6.12 demonstrates how the OCC-PDATI and OCC-RTDATI favors transactions of high importance. OCC-PDATI and OCC-RTDATI clearly offers 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.

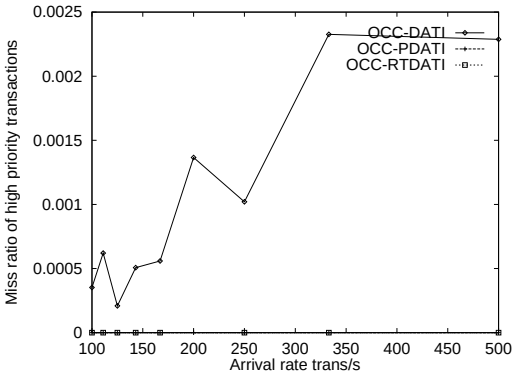


(a) writes 10%

(b) writes 20%



(c) writes 30%



(d) writes 40%

Figure 6.12: 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%.

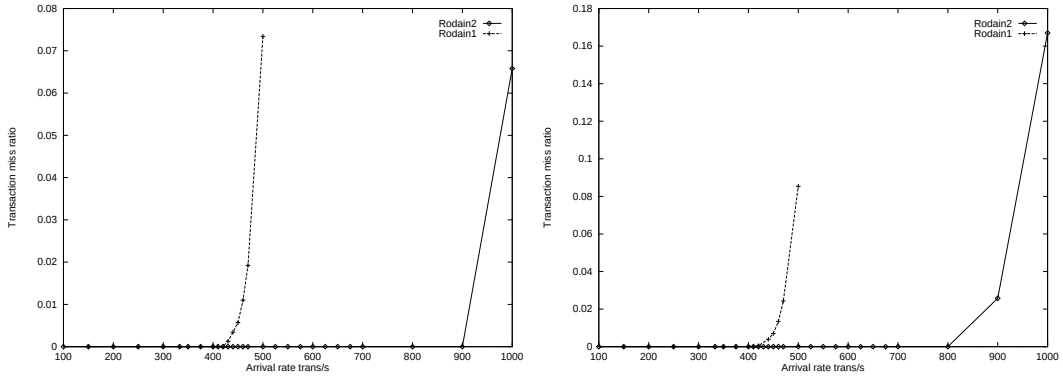
6.9 Other experiments

This section presents results from experiments which have been executed using two different systems. The goal of this experiment is to study what kind of effects there are when the hardware changes. Two different hardware have been used. Detailed information can be found in Table 6.2.

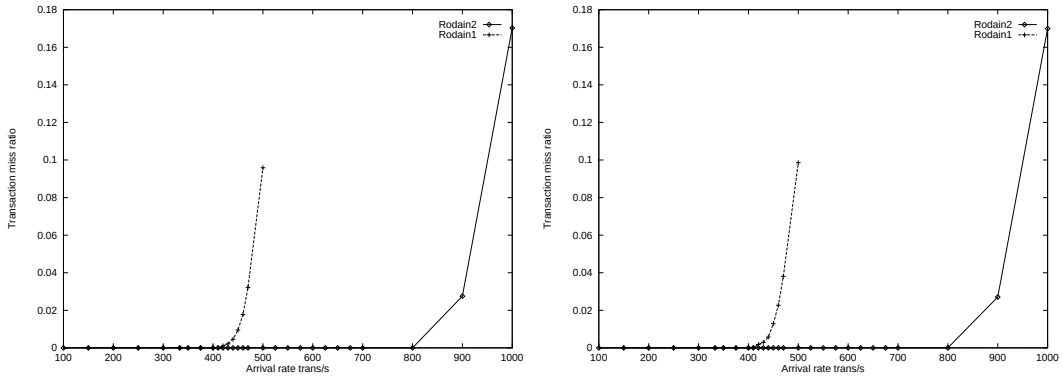
Table 6.2: System Specifications

Parameter	Rodain1	Rodain2
Processor	Intel Pentium Pro 200 MHz	Intel Pentium III 600 MHz
Motherboard	Intel VS440FX	ABIT BX6 2.0
Second level cache	256KB	256KB
Bus speed	66MHz	100 MHz
Memory	2 * 32MT SIM 32bit 72pin JED	512MT SDRAM 168pin DIMM

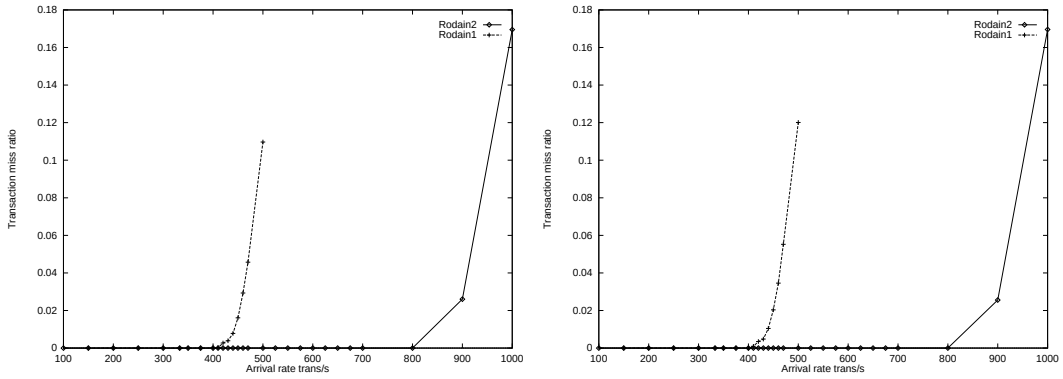
Figure 6.13 shows results from hardware experiments. These results clearly indicate that when the processor speed is tripled then the overall system performance is doubled. The Rodain1 system becomes overloaded when the arrival rate of transactions is 400 transactions per second. The Rodain2 system becomes overloaded when the arrival rate of transactions is 800 transactions per second. These results clearly indicate that the RODAIN system is easily scaled using faster hardware.



(a) wp0 (b) wp10



(c) wp20 (d) wp30



(e) wp40 (f) wp50

Figure 6.13: Hardware effects.

Chapter 7

Conclusions

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 objects either transient or persistent. 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 paper 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, 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 has also proposed two different method to take transaction criticalness 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 high criticality 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.

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, that may continue execution after the deadline at lower priority; and firm transactions that are terminated when the deadline is not met.

The experiments indicated that the validation time of the transactions becomes crucial. To increase database throughput in 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.

Bibliography

[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, Los Angeles, California, USA, 1988. Morgan Kaufmann.

[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, Amsterdam, The Netherlands, 1989. Morgan Kaufmann.

[4] R. Abbott and H. Garcia-Molina. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 113–124, Lake Buena Vista, Florida, USA, 1990. IEEE Computer Society Press.

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

[6] 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.

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

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

[9] 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, Huntsville, Alabama, USA, 1988. IEEE Computer Society Press.

[10] A. P. Buchmann, D. R. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: A framework for integrating real-time scheduling and concurrency control. In *Proceedings of the 5th International Conference on Data Engineering*, pages 470–480, Los Angeles, California, USA, 1989. IEEE Computer Society Press.

[11] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kauffmann, 1997.

[12] 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.

[13] 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.

[14] 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, Burlington, Vermont, USA, 1997. Kluwer Academic Publishers.

[15] 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.

[16] J. J. Garrahan, P. A. Russo, K. Kitami, and R. Kung. Intelligent network overview. *IEEE Communications Magazine*, 31(3):30–36, March 1993.

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

[18] 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.

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

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

[21] T. Härder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–318, December 1983.

[22] 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.

[23] 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, Lake Buena Vista, Florida, USA, 1990. IEEE Computer Society Press.

[24] 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, Nashville, Tennessee, 1990. ACM Press.

[25] 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.

[26] 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, Barcelona, Catalonia, Spain, September 1991. Morgan Kaufmann.

[27] 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, San Antonio, Texas, USA, 1991. IEEE Computer Society Press.

[28] 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.

[29] 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, Santa Monica, California, USA, December 1989. IEEE Computer Society Press.

[30] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *The 23rd International Symposium on Fault-Tolerant Computing*, pages 2–9, Toulouse, France, 1993. IEEE Computer Society Press.

[31] 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.

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

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

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

[35] E. D. Jensen, C. D. Locke, and Tokuda. H. A time-driven scheduling model for real-time systems. In *Proceedings of the 5th IEEE Real-Time Systems Symposium*, pages 112–122, San Diego, California, USA, 1985. IEEE Computer Society Press.

[36] 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, Pittsburgh, PA, USA, 1993. IEEE Computer Society Press.

[37] 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.

[38] 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.

[39] J. Kiviniemi, T. Niklander, P. Porkka, and K. Raatikainen. Transaction processing in the RODAIN real-time database system. In A. Bestavros and V. Fay-Wolfe, editors, *Real-Time Database and Information Systems*, pages 355–375, London, 1997. Kluwer Academic Publishers.

[40] J. Kiviniemi and K. Raatikainen. Object oriented data model for telecommunications. Report C-1996-75, University of Helsinki, Dept. of Computer Science, Helsinki, Finland, October 1996.

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

[42] 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, Skövde, Sweden, 1995. Springer.

[43] 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, Chigago, Illinois, 1995. IEEE Computer Society Press.

[44] 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.

[45] 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*, Birmingham, UK, 1997. IEEE Computer Society Press.

[46] 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.

[47] 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.

[48] 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, Raleigh-Durham, NC, USA, 1993. IEEE Computer Society Press.

[49] 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.

[50] 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, Los Alamitos, Calif., 1990. IEEE Computer Society Press.

[51] 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, Cheju Island, South Korea, 2000. IEEE Computer Society Press.

[52] J. Lindström. Using optimistic concurrency control in firm real-time databases. *HeCSE Winter School 2000: Next Millennium Computing, ISBN 951-22-4940-5*, January 2000.

[53] 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, 1819, pages 158–173, Edinburgh, UK, Co-located with VLDB-99, 1999.

[54] 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, Hong Kong, China, 1999. IEEE Computer Society Press.

[55] 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, Cheju Island, South Korea, 2000. IEEE Computer Society Press.

[56] 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, Vilnius, Lithuania, 2000.

[57] 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.

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

[59] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[60] T. Niklander, J. Kiviniemi, and K. Raatikainen. A real-time database for future telecommunication services. In D. Gaïti, editor, *Intelligent Networks and Intelligence in Networks*, pages 413–430, Paris, France, 1997. Chapman & Hall.

[61] OMG. *CORBAservices: Common Object Service Specification*. OMG Document, December 1998. John Wiley & Sons, 1998.

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

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

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

[65] 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.

[66] 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.

[67] K. Raatikainen. Information aspects of services and service features in intelligent network capability set 1. Report C-1994-45, University of Helsinki, Dept. of Computer Science, Helsinki, Finland, September 1994.

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

[69] 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.

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

[71] 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.

[72] 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.

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

[74] 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.

[75] 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, Tempe, Arizona, USA, 1992. IEEE Computer Society Press.

[76] B. Sprunt, D. Kirj, and L. Sha. Priority-driven, preemptive i/o controllers for real-time systems. In *Proceedings of the International Symposium on Computer Architecture*, pages 152–159, Honolulu, Hawaii, USA, 1988. ACM.

[77] J. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *The Journal of Real-Time Systems*, 2:247–254, 1990.

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

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

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

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

[82] S. Thomas, S. Seshadri, and J. R. Haritsa. Integrating standard transactions in firm real-time database systems. *Information Systems*, 21(1):3–28, 1996.

[83] S-M. Tseng, Y. H. Chin, and W-P. Yang. Scheduling real-time transactions with dynamic values: A performance evaluation. In *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications*, Tokio, Japan, 1995. IEEE Computer Society Press.