Experimental performance evaluation of RODAIN concurrency control and scheduling

Jan Lindström

Helsinki 16th March 1998

University of Helsinki
Department of Computer Science
Master of Science Thesis

Contents

1 Introduction

Databases will have a dominant role in future telecommunication services. Databases will hold the information needed in operations and management of telecommunication services and networks. The performance, reliability, and availability requirements of data access operations are hard. The requirements of the telecommunications database architectures originate in the following areas: real-time access to data, fault tolerance, distribution, object orientation, efficinency, flexibility, multiple interfaces, security and compatibility [Ahn94, Raa94, TR96].

Real-Time Object-Oriented Database Architecture for Intelligent Networks (RODAIN) is an architecture for a real-time, object-oriented, fault-tolerant, and distributed database management system. The RODAIN architecture is designed to fulfill the requirements derived from the most important telecommunications standards including *Intelligent Network* (IN) [GRKK93], *Telecommunications Management Network* (TMN) [ITU92c], and *Telecommunication Information Networking Architecture* (TINA) [AKS93].

The most challenging issue in designing a real-time transaction processing system for telecommunications is the handling of transactions having very different characteristics. A telecommunication database system should be able to support 1) short simple read transactions, 2) short simple updating transactions, and 3) very long and complex updating transactions in the same real-time database system. The RODAIN Database is designed to meet those constraints. To support heterogeneous transaction characteristics, modifications must be done into existing transaction scheduling and concurrency control mechanisms.

In telecommunication applications, database transactions have several different characteristics [RKMT95]. In a RODAIN database system two transaction types have been studied: service provision and service management transactions. Service provision transactions are used to retrieve information about characteristics of the called subscriber, the calling telephone line, and update information of a single subscriber. Service management transactions are used to update database contents widely, for example when adding new users or configuring user profiles.

Service provision type transactions are short reads or updates, which affect usually a few database objects. Typically the object is fetched based on its key attribute value. Transaction atomicity and consistency requirements can sometimes be relaxed [TR96]. However, service provision transactions have quite strict deadlines and their arrival rate can be high, but most service provision transactions have read-only semantics. In existing service provision applications, the write propability is approximately 1 to 10 percent [TR96]. In RODAIN transaction scheduling, service provision transactions are expressed as firm deadline transactions.

Service management transactions have opposite characteristics. They are long updates which write many objects. A strict serializability, consistency and atomicy is required for service management transactions. However, they do not have explicit deadline requirements. Thus, service management transactions are expressed as non-realtime transactions.

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

In traditional databases correctness is well defined as serializability and equally for all transactions. However, strict serializability as the correctness criterion is not always the most suitable one in real-time databases, in which correctness requirements may vary from one type of transactions to another. Some data may have temporal behavior, some data can be read although it is already written but not yet committed, and some data must be guarded by strict serializability. These conflicting requirements must be solved through using a special purpose concurrency control scheme.

A heterogeneous transaction behavior introduces problems also to concurrency control. In traditional databases, database correctness is well defined and homogeneous between transactions. However, a demand for strict database correctness is not applicable for real-time databases, where correctness requirements can be heterogenous. Some of the data may have a temporal behavior, some data can be read although it is written but not committed, and some

of the data must be guarded with a strict serializability. These conflicting requirements must be solved with a special concurrency control scheme.

In RODAIN the object-oriented approach was chosen because an object model for real-time transactions can be used to provide the information needed in the concurrency control and real-time scheduling of heterogeneous transactions. Object orientation is derived from general telecommunications standards. Object orientation is a general trend in other standards, and it is also naturally adaptable to databases [ABD+92]. When object orientation is used, it is easy to encapsulate real-world data to database objects. Also general object-oriented tools, such as object methods and inheritance, make data modelling easy and straightforward.

Multiple interfaces to the database are a necessity in telecommunications. A telecommunications database has different types of users all of which have a different view to the data. In telecommunications, the data views differ not only in data but also in metadata. Thus some users might want to see data in database as an X.500 Directory [ITU92a] while others want to see it as as X.700 Management Information Tree [ITU92b]. It is possible that not all metadata transformations are possible in the object data model.

Finally, a telecommunications database should be compatible with other object standards, especially with standards that deal with object exchange between different entities in the telecommunications network. Currently the hot items are TINA object exchange [AKS93] and OMG CORBA [OMG92]. In particular, a telecommunications database management system should be able to support CORBA interfaces.

In this paper we focus on transaction scheduling and concurrency control. We will propose a new scheduling policy and relaxed concurrency control to solve challenges defined above. Implemented scheduling policy and concurrency control protocols are tested with real transactions in Intelligent Networks framework.

This thesis is organized as follows. In section 2 we briefly summarize the RODAIN Database architecture, data model, process structure, and transaction processing. In section 3 we review real-time transaction scheduling. On section 4 we review principles of optimistic concurrency control and in section 5 we present classical optimistic protocols, two new protocols in detail

and we propose two new protocols. Section 6 presents experiment configuration and results. Finally section 7 summarizes the thesis.

2 Overview of RODAIN Real-Time Database System

This section is based on work published in [KNPR97, NKR97]. The RODAIN DBMS architecture consists of a set of autonomous RODAIN Database Nodes that interact with each other. Each Database Node may communicate with one or more applications, and an application may communicate with one or more database nodes. The RODAIN Database Node consists of Database Primary Node, Database Mirror Node, and a Secondary Storage Subsystem; see Figure 1.

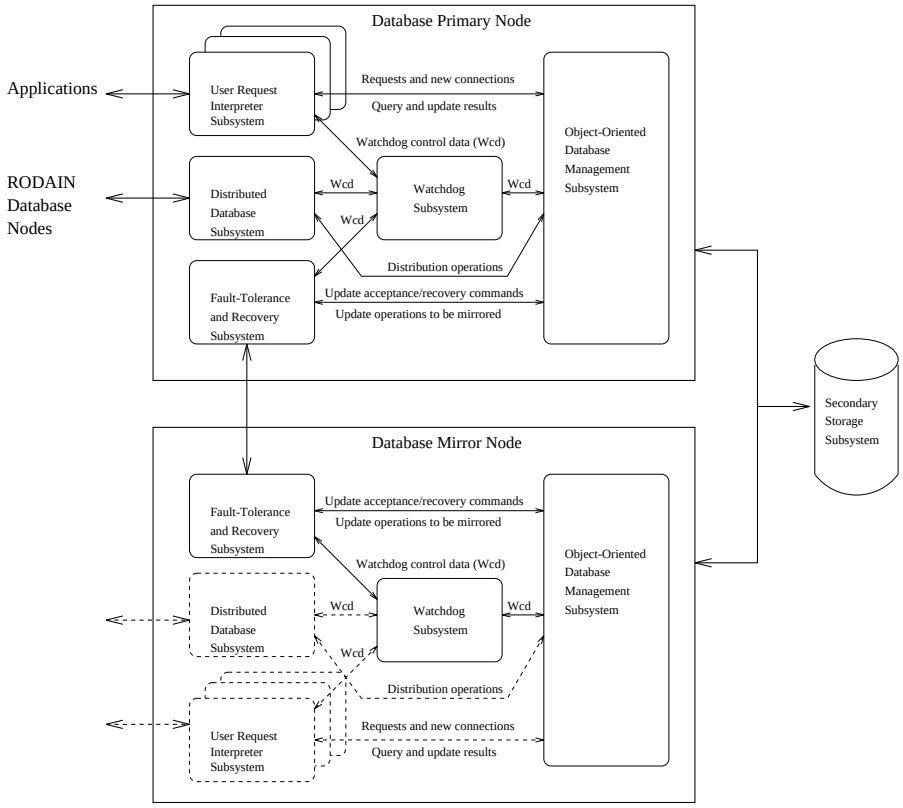


Figure 1: Structure of a RODAIN Database Node.

At each time one of the nodes, called *Database Primary Node*, executes the application requests. The other node, called *Database Mirror Node*, maintains a copy of the main memory

database. The mirror also takes care of storing the transaction log into the Secondary Storage Subsystem.

The *User Request Interpreter Subsystem* (URIS) is the entry point to the RODAIN Database. All application requests arrive through an URIS. The Distributed Database Subsystem is used in communication with other autonomous RODAIN Database Nodes. The *Fault-Tolerance and Recovery Subsystem* (FTRS) handles the communication between the primary and mirror node. The FTRS also takes care of all issues related to fault tolerance. The Watchdog Subsystem controls the other subsystems in the node. Its responsibility is to detect any malfunction, fault, or failure in other subsystems. The *Object-Oriented Database Management Subsystem* (OO-DBMS) is the database engine. It takes care of the data manipulation and maintains the main memory database.

The RODAIN data model is a true superset of the ODMG 2.0 data model [Cat97]. The concepts of real-time extensions are expressed as additions to the ODMG 2.0 type hierarcy 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 [KR96].

2.1 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 2. Below we briefly summarize the basic concepts of processes that are relevant when transactions are processed.

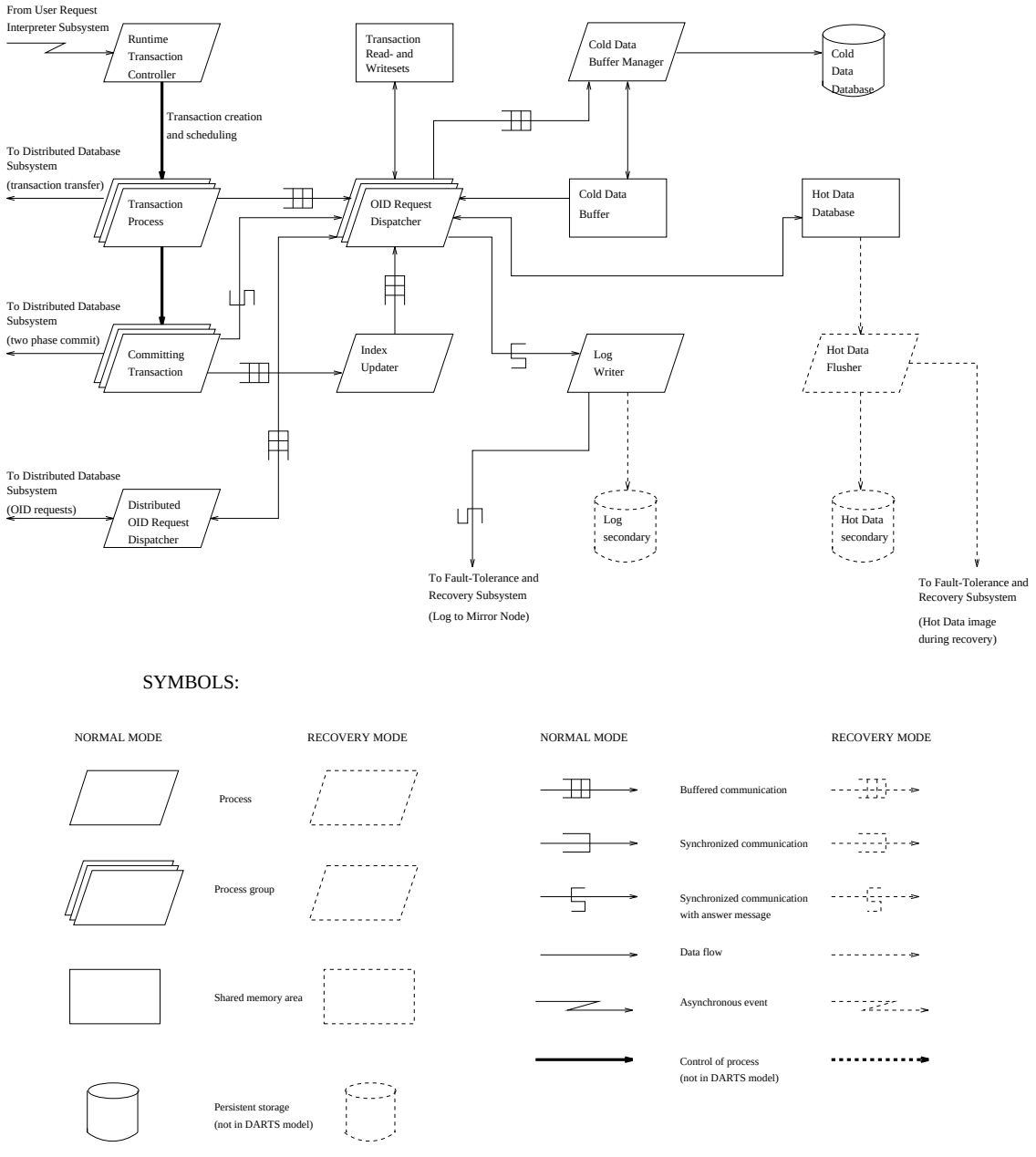


Figure 2: Processes in the RODAIN Database Primary Node.

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 han-

dling 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 for 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 preforms the validation of the transaction commit using read-set and the write-set of the transaction. If the validation is successful, all modified objects are written into the database and the transaction is commited.

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.

2.2 Transaction Processing

The transaction flow of a local transaction is presented in Figure 3. The flow is shown with numbers that are referred in the explanations below.

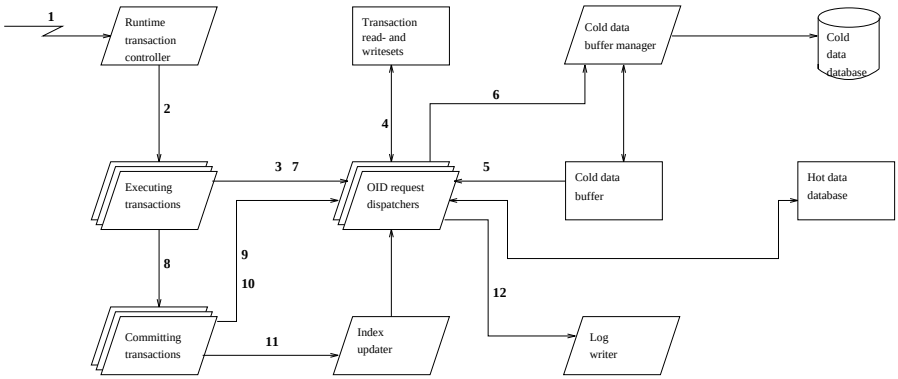


Figure 3: Flow of local Transaction.

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 in 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 to 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, that is 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 send to the application.

3 Real-Time Transaction Scheduling

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.

The deadlines can be divided into three types: *hard*, *soft*, and *firm* deadlines [AGM88a] (Figure 4). The hard deadline means, that a task may cause a very high negative value to the system, if the computation is not completed before the deadline. In a soft deadline, the computation has a decreasing value after the deadline, and the value may become zero ultimately. In the middle of these extremes, a firm deadline is defined: a task loses its value after the deadline, but no negative consequence will occur.

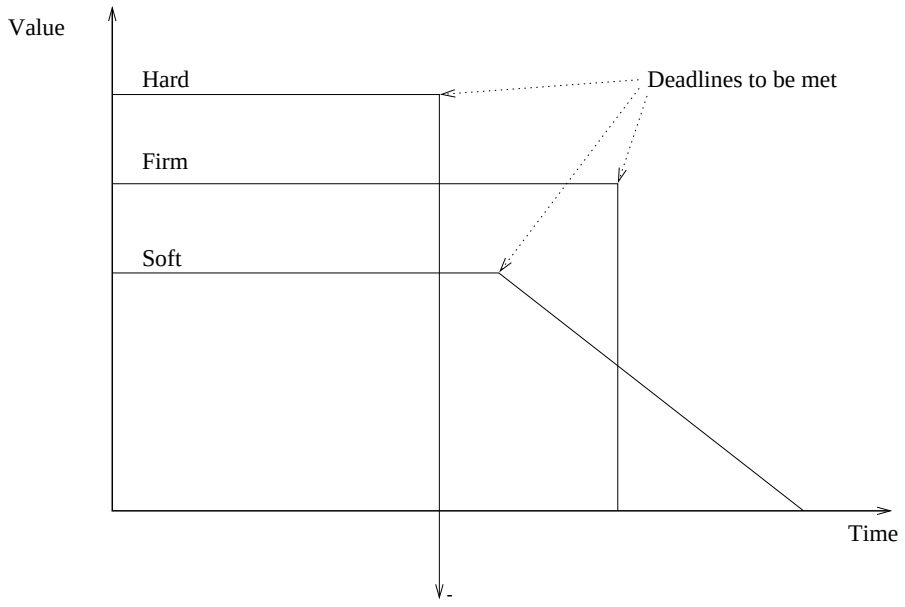


Figure 4: The deadline types.

Predictability is one of the primary issues in real-time systems [SR90]. Schedulability analysis or feasibility checking of the tasks of a real-time system has to be done to predict whether the tasks will meet their timing constraints.

3.1 Priority Inversion

In a real-time database environment conflict resolution of concurrency control may interfere with CPU scheduling. When blocking is used to resolve a conflict such as in 2PL [EGLT76], a *priority inversion* [AGM88b] phenomenon can occur if a higher priority transaction gets blocked by a lower priority transaction.

Figure 5 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 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 to run. Thus, T_1 misses its deadline due to the resource conflict.

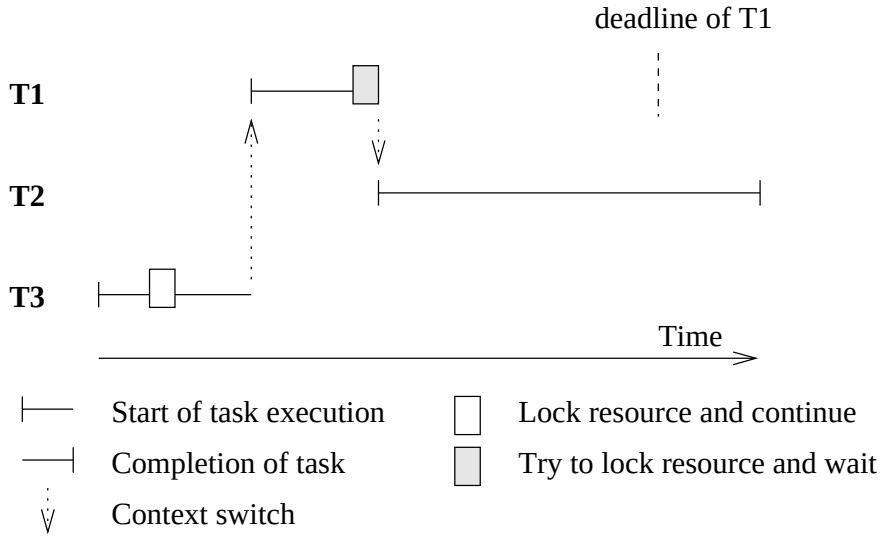


Figure 5: Priority inversion example.

In [SRL90], 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 6 illustrates, how a priority inversion problem presented in figure 5 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 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.

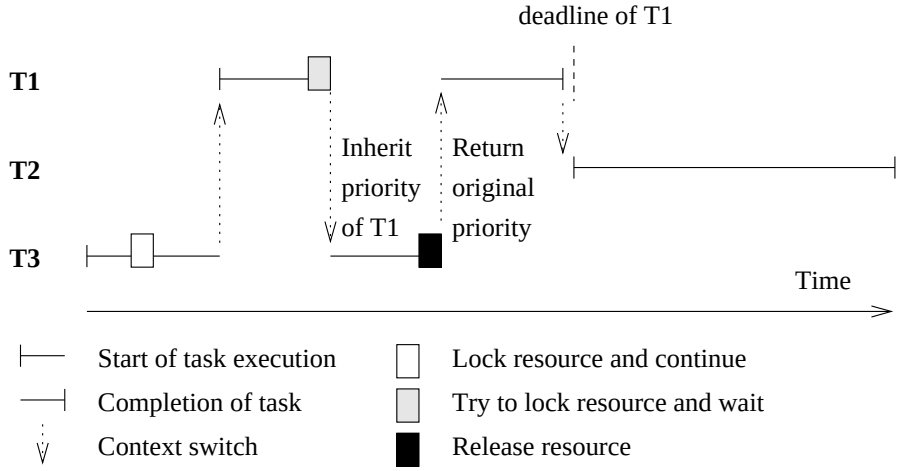


Figure 6: Priority inheritance example.

3.2 Sheduling paradigms

Transactions in a real-time database can often be expressed as *tasks* in a real-time system. A *transaction scheduling policy* defines how priorities are assigned to individual transactions. 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. Here we quote only a few examples.

Several scheduling paradigms emerge, depending on a) whether a system performs 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 we can identify the following classes of algorithms [RS94]:

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

In the *earliest deadline first* (EDF) [LL73, HLC91] 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) [AGM88a] policy, the transaction with the shortest slack time is executed first. The slack time is an estimate of how long we can delay the execution of transaction and still meet its deadline. In the *highest value* (HV) [HCL91] policy, transaction priorities are assigned acording to the transaction value attribute.

The *priority ceiling protocol with dynamic adjustment* (PCP-DA) [LSH97] uses a priority ceiling protocol with dynamic adjustment of serialization order to enhance the system schedulability in hard RTDBs. PCP-DA can avoid unnecessary blockings and is deadlock-free ensuring that a transaction can be blocked at most once by a lower priority transaction. PCP-DA provides a better schedulabitity condition compared to other protocols [LSH97]. A survey of transaction scheduling policies is found in [AGM88a].

3.3 Transaction Scheduling in RODAIN

The RODAIN scheduling algorithm, called *FN-EDF*, is designed to support simultaneous execution of both firm real-time and non-realtime transactions. In RODAIN, firm deadline transactions are scheduled according to the EDF scheduling policy [LL73, HLC91]. 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[1] 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 $(c = 1, \ldots, C)$ the target value is given as a system parameter γ_c, which can be changed while the system is running.

Let $a_i(t)$ be the cpu-time consumed and $b_i(t)$ be the number of database operations requested by transaction i in the time interval $[0, t]$. The execution time of transaction i at time t, $r_i(t)$, is defined as

$$r_i(t) = a_i(t) + \lambda b_i(t) \ , \tag{1}$$

where the constant λ characterizes the execution time of a database operation[2]. Thus, the execution time of all transactions in time interval $[0, t]$, $R(t)$, is

$$R(t) = \sum_i r_i(t) \ , \tag{2}$$

where the summation goes over all transactions being active in the time interval $[0, t]$.

Let $t_k \ , \ k = 1, 2, \ldots$, denote the periodic sampling times. Let a non-realtime transaction i be started during the sampling interval $[t_{i-1}, t_i]$. The fraction of execution time obtained by the transaction i, $F_i(t_k)$, is

$$F_i(t_k) = \frac{r_i(t_k)}{R(t_k) - R(t_{i-1})} \ , \ t_k \geq t_i \ . \tag{3}$$

[1] In the RODAIN prototype the operating system is Chorus ClassiX.

[2] The form of Equation 1 is due to the way we have implemented the RODAIN database kernel.

The target value, γ_i, to which $F_i(t_k)$ is compared, is defined as

$$\gamma_i = \frac{\gamma_c}{n_c(t_k, t_{i-1})} \tag{4}$$

where γ_c is the fraction of computing resources assigned to non-realtime transactions in class c and $n_c(t_k, t_{i-1})$ is the average number of active transactions in class c during the time interval $[t_{i-1}, t_k]$.

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.

3.4 Implementation Issues

In RODAIN, scheduling of transaction processes is based on the FIFO scheduling policy provided by the Chorus operating system [Pou94]. 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.

We have 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 in 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 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 also the calculated fractions may 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.

Our prototype database system is built over Chorus real-time micro-kernel [Pou94] on the Pentium processor. Our implementation can offer cpu time at the granularity of approximately 1 millisecond. We use a sampling period of 5 seconds, which offers enough calculation resolution but transaction execution time is prolonged due to the slow start scheme.

4 Optimistic Concurrency Control

In *Optimistic Concurrency Control* (OCC) [KR81], transactions are allowed to execute unhindered until they reach their commit point, at which time they are validated. The execution of a transaction consists of three phases, *read phase*, *validation phase*, and *write phase*.

The read phase is the normal execution of the transaction. Write operations are performed on private data copies in the local workspace of the transaction. This kind of operation is called *pre-write*. Identification of all read data items is stored in a *readset*. Identification of all pre-written data items is stored in a *writeset*. The readset and writeset are used to check conflicts against other active transactions.

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

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 algorithms because I/O operations in the write phase are required to be done sequentially in the critical section. 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 OCC protocol, three issues have to be considered:

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 there are conflicts.

4.1 Backward Validation

In *Backward Validation* [Här84], 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 algorithm in [KR81] is based on this validation process.

Let T_v be the validating transaction and T_c $(c = 1, 2, ..., n, c \neq v)$ be the committed transactions, which have been active during the read phase of T_v. Let $RS(T)$ and $WS(T)$ denote the read set and the write set of the transaction T, respectively. The backward validation protocol is presented in Figure 7.

```
validate(T_v)
{
    valid = true;
    for T_c (c = 1, 2, ..., n)
    {
        if WS(T_c) ∩ RS(T_v) ≠ ∅ then
            valid = false;
        if not valid then
            break;
    }

    if valid then
        commit WS(T_v) to the database;
    else
        restart(T_v);
}
```

Figure 7: Backward Validation algorithm.

4.2 Forward Validation

In *Forward Validation* [Här84], 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 algorithms based on this validation process are studied in [Här84].

Let $T_a(a = 1, 2, ..., n, a \neq v)$ be the conflicting transactions in their read phase. The forward validation protocol is presented in Figure 8.

In the real-time database systems, data conflicts should be resolved in favor of the higher priority transactions. Forward Validation provides flexibility for conflict resolution. 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.

```
validate(T_v)
{
    valid = true;
    for T_a  (a = 1, 2, ...n)
    {
        if  RS(T_a) ∩ WS(T_v) ≠ ∅  then
            valid = false;
        if not valid then
            break;
    }

    if valid then
        commit WS(T_v) to the database;
    else
        conflict_resolution(T_v);
}
```

Figure 8: Forward Validation algorithm.

4.3 Unnecessary restarts

The major performance problem with OCC protocols are the heavy restart overheads, wasting a large amount of resources. 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 [SLL92].

Forward Validation (OCC-FV) [Här84] 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 incur restarts that are not necessary to ensure data consistency. These restarts should be avoided.

Let $r_i[x]$ and $w_i[x]$ denote the read and write operations, on data object x by transaction T_i, and let v_i and c_i denote the validation and commit of transaction T_i, respectively. Consider transactions T_1 and T_2:

$T_1 : r_1[x]w_1[x]v_1c_1$
$T_2 : r_2[x]w_2[y]v_2c_2$

and suppose they execute as follows:

$H_1 = r_2[x]w_2[y]r_1[x]w_1[x]v_1$

Based on the OCC-FV [Här84], T_2 has to be restarted. However, it is not necessary. Because if T_2 is allowed to commit such as:

$H_1 = r_2[x]w_2[y]r_1[x]w_1[x]v_1c_1\mathbf{v_2c_2}$,

the history, H_1, is equivalent to the serialization order of $T_2 \rightarrow T_1$. There is no cycle in the history H_1. Therefore the execution is serializable □.

We refer such a restart of T_2 in the forward validation as an *unnecessary restart*. Also we refer to transactions having both write-write and write-read conflicts with the validating transaction as *irreconcilably conflicting*, while transactions having only write-read conflicts as *reconcilably conflicting* [LS93].

4.4 Relaxing Serializability

Traditional concurrency control methods use serializability [EGLT76] as the correctness criterion when transactions are concurrently executed. However, in real-time database systems strict serializability is not always needed because it restricts simultaneous access and induces unnecessary overhead to the system. Real-time data is often temporal and neither serializing concurrency control nor full support for failure recovery is not required because of the overhead [SZ88]. This has lead to ideas such as *atomic set-wise serializability* [SLJ88], *external versus internal consistency* [Lin89], *epsilon serializability* [RP96], and *similarity* [KM93]. Graham [H.92] have argued that none are as obviously correct, nor as obviously implementable, as serializability. Graham [H.93] have presented a method suporting atomicity seemingly without any mechanism to do so. This mechanism does not seem to fit our system.

Due to the semantic properties of telecommunications applications, the correctness criterion can be relaxed to so called *semantic based serializability*. In the RODAIN database system we decompose the semantics based serializability into two part. Firstly, we define a new

temporal serializability criterion called τ-*serializability* [RKMT95], which allows old data to be read unless the data is too old. Secondly, we use a semantic conflict resolution model that introduces explicit rules, which are used to relax serializability of transactions. The first method reduces the number of read-write conflicts whereas the second one reduces the number of write-write conflicts. We also present how the OCC-DA concurrency control algorithm [LLH95a] is modified to support our extensions.

We use a correctness criterion called τ-serializability to reduce read-write conflicts. Suppose that transaction T_A updates the data item x and gets a write lock on x at time t_a. Later transaction T_B wants to read the data item x. Let t_b be the time when T_B requests the read lock on x. In τ-serializability the two locks do not conflict if $t_a + \min(\tau_b, \tau_x) > t_b$. The tolerance $\min(\tau_b, \tau_x)$ specifies how long the old value is useful, which may depend both on data semantics (τ_x) and on application semantics (τ_b).

Assume that $RS(T_i)$ is the readset of transaction T_i, $WTS(D_p)$ is the write timestamp of an object D_p, and $ROTS(T_i, D_p)$ is the read operation timestamp of the object D_p by transaction T_i. Then the τ-serializability is formally defined as

$$\text{Transaction } T_i \text{ is } \tau\text{-serializable} \iff \forall D_p \in RS(T_i) : \; WTS(D_p) - ROTS(T_i, D_p) \leq \min(\tau_p, \tau_i)\,. \tag{5}$$

4.5 Semantic Conflict Resolution

Garcia-Molina [GM83] defines a *semantically consistent schedule* to be a schedule that transforms the database into a consistent state. Transactions are classified into semantic types based on their database operations. For each semantic type, a *compatibility set* is defined in order to identify which other semantic types are compatible. When a transaction requests a lock of an object that is locked by another transaction, the transaction processing system checks whether or not the semantic type of the requesting transaction is in the compatibility set of the transaction holding the lock. If the requesting transaction belongs to the compatibility set, the

lock is granted, otherwise the requesting transaction must “wait” to gain access to the object.

In RODAIN we have adopted the model of semantic-based concurrency control presented in [Sin88, LP96]. We use two semantic levels for write operations: *update* and *replace* semantics. Update semantics is like a write operation in traditional databases. Replace semantics is used when the new value of an object does not depend on any value of any object in the database. This semantics is like a blind write operation but the transaction can perform a read operation before replacing the object. In RODAIN the semantic model is specified as an attribute of the transaction. Thus, all write operations in the same transaction have the same semantics.

Four different kinds of locks are used in the semantic concurrency control of RODAIN: *read*, *update*, *replace*, and *validate*. A read lock is granted for read access. Update and replace locks are used in write operations depending on selected semantics of the transaction. A validate lock is used to guarantee that a transaction already validated is not disturbed during its write phase. The compatibility table of locks is presented in table 1.

Table 1: Lock Compatibilities Used in RODAIN

Validating	Active transaction locks			
Transaction locks	Validate	Update	Replace	Read
Read	abort	adjust	adjust	OK
Update	abort	adjust	adjust	adjust
Replace	abort	adjust	OK	adjust

Because an optimistic concurrency control method is used in RODAIN, all locks are granted immediately. All conflicts are checked in the validation phase. If a conflict is detected between the validating transaction and an active transaction, the active transaction is adjusted using dynamic adjustment of the serialization order.

5 Optimistic Concurrency Control protocols

In recent years, the use of optimistic schemes for concurrency control in real-time optimistic protocols has received more and more attention. Different real-time optimistic protocols have been proposed. They incorporate different priority conflict resolution methods in the validation phase of a transaction. In this paper we briefly present some popular optimistic protocols and two optimistic protocols in detail.

OPT-BC [HCL90b] extends classical optimistic protocol with the *Broadcast Commit* [MN82, Rob87] protocol. In the Broadcast Commit protocol the transaction notifies other running transactions that conflict with it. The conflicting transactions are restarted.

OPT-SACRIFICE [HCL90a] is an optimistic protocol which uses a priority-driven abort for conflict resolution. When a transaction reaches its validation phase, the algorithm checks for conflicts with currently executing transactions. If conflicts are detected and at least one of the transactions in the conflict set is a higher priority transaction, then the validating transaction is restarted. It is *sacrificed* in an effort to help the higher priority transactions make their deadlines.

OPT-WAIT [HCL90a] algorithm incorporates a *priority wait* mechanism. When a transaction reaches its validation phase, if its priority is not the highest among the conflicting transactions, it waits for the conflicting transactions with higher priority to complete. This gives the higher priority transactions a chance to make their deadlines first. If the transaction finally commits after waiting for some time, it causes all its conflicting transactions with lower priority to be restarted. This transaction restart problem becomes worse with the possibility of chained blocking, which may cause cascaded aborts [LS96].

The WAIT-50 [HCL90a] algorithm is an extension of the OPT-WAIT. It incorporates a *wait control* mechanism. This mechanism monitors transaction conflict states and dynamically decides when, and for how long, a low priority transaction should be made to wait for its conflicting higher priority transactions. In WAIT-50 scheme, a validating transaction is made to wait as long as more than half the transactions that conflict with it have higher priorities.

Otherwise it commits and all the conflicting transactions are restarted. Performance studies in [HCL90a] have shown WAIT-50 to provide significant performance gains over OPT-BC, OPT-SACRIFICE and OPT-WAIT.

5.1 OCC-TI

The OCC-TI [LS93, Lee94] protocol resolves conflicts using time 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 protocol, every transaction in the read phase is assigned a timestamp interval (TI). This interval is used to record a temporary serialization order induced 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 induced 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 time interval allocated to the transaction. If another transaction has written the object outside the time interval, the transaction must be restarted. In the read algorithm (figure 9) D_i is the object to be read, T_i is the transaction reading the object, $TI(T_i)$ is the time 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.

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 time interval allocated to the transaction (figure 9). If another transaction has read or written the object outside the time 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 9: Read algorithm and pre-write algorithm for OCC-TI.

The noticeable point of OCC-TI is that unlike other optimistic algorithms, it does not depend on the assumption of the serialization order of transactions as being the same as the validation phase arriving order. OCC-TI records the serialization order induced precisely and uses restarts only when necessary. This means that OCC-TI uses dynamic adjustment of serialization order. Let us examine how the serialization order is adjusted between the validating transaction and a concurrently active transaction for the three possible types of conflict:

1. $RS(T_v) \cap WS(T_j) \neq \emptyset$ (read-write conflict)

   A read-write conflict between T_j and T_v 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*.

   Example: $TI(T_1) = [0, \infty[\ \cap \ [100, \infty[\ = [100, \infty[$.

2. $WS(T_v) \cap RS(T_j) \neq \emptyset$ (write-read conflict)

   A write-read conflict between T_j and T_v 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*.

Example: $TI(T_1) = [0, \infty[\ \cap \ [0, 100] \ = \ [0, 100]$.

3. $WS(T_v) \cap WS(T_j) \neq \emptyset$ (write-write conflict)

   A write-write conflict between T_j and T_v 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).

At the beginning of the validation (figure 10), the final timestamp of the validated transaction is determined from the timestamp interval allocated to the transaction. In this algorithm, the minimum value of $TI(T_v)$ is selected as the timestamp $TS(T_v)$ [LS93, Lee94]. The timestamp intervals of all 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 a running transaction iterates through the readset and writeset. When access has been made to the same objects both in the validating transaction $TS(T_v)$ and in another active transaction $TS(T_a)$, the time interval of the $TS(T_a)$ is adjusted. If $TS(T_a)$ has made a conflicting operation in an adjusted time interval, the transaction must be restarted.

Performance studies in [LS93] have shown that under the policy that discards tardy transactions from the system, the optimistic algorithms outperform 2PL-HP [AGM88b]. OCC-TI does better than OPT-BC among the optimistic algorithms. 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.

```
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 10: Validation algorithm for OCC-TI.

5.2 OCC-DA

OCC-DA [LLH95a] is based on the Forward Validation scheme. 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.

One way to reduce the number of transaction restart is to dynamically adjust the serialization order of the conflicting transactions. When some data conflict with the validating transaction is detected, there is no need to restart the conflicting transaction immediately. Instead, a serialization order can be dynamically defined.

OCC-DA uses similar dynamic adjustment of serialization order as in OCC-TI. A write-read conflict (see section 5.1 item 1) is adjusted using *forward adjustment*. Read-write conflict (see item 2) is adjusted using *backward adjustment*. Write-write conflict (see item 3) is adjusted using forward adjustment.

To support dynamic adjustment of the serialization order in OCC-DA, a dynamic timestamp assignment method is used. 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 T_v comes to validation, the sets 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 transac-

tions, 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 read data item D_p.

The first part of the validation test is used only for those validating transactions which have been backward adjusted (figure 11). 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(Tv)
{
    if  SOT(Tv) ≠ ∞  then
    {
        for  ∀ Dp ∈ RS(Tv)
        {
            if  TR(Tv, Dp) > SOT(Tv)  then
                restart(Tv);
        }

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

Figure 11: First part of the validation algorithm for OCC-DA.

The purpose of part two of the test is to detect read-write conflicts between the active transactions and the validating transactions (figure 12). 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 12: Second part of the validation algorithm for OCC-DA.

The third part of the test is to detect whether a backward-adjusted transaction T_j also needs forward adjustment with respect to T_v (figure 13). 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 we select transactions for restart 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 13: Third part of the validation algorithm for OCC-DA.

When the validating transaction reaches part four of the test, 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 14).

```
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 14: Fourth part of the validation algorithm for OCC-DA.

Performance studies in [LLH95a] have shown that OCC-DA outperforms OCC-TI. OCC-DA can be extended to use Thomas's write rule [Tho79] and this extension is presented in [LLH95b].

5.3 Revised OCC-TI

The algorithms provided for OCC-TI in [LS93, Lee94] do not seem to fully resolve the unnecessary restart problem. The problem with the existing algoritm is best described by the example given below. Let $RTS(x)$ and $WTS(x)$ be initialized as 100. Consider transactions T_1, T_2, and history H_1:

T_1: $r_1[x]w_1[x]v_1c_1$

T_2: $r_2[x]v_2c_2$

$H_1 = r_1[x]r_2[x]w_1[x]v_1c_1$.

Then the OCC-TI algorithm executes the history H_1 as follows. Transaction T_1 executes

$r_1[x]$, which causes the time interval of the transaction to be forward adjusted to be $TI(T_1) = [0, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_2 then executes a read operation on the same object, which causes the time interval of the transaction to be similarl forward adjusted.

Transaction T_1 then executes $w_1[x]$, which causes the time interval of the transaction to be forward adjusted to be $TI(T_1) = [100, \infty[\cap [100, \infty[\cap [100, \infty[= [100, \infty[$. Transaction T_1 starts the validation, and the final timestamp is selected to be $TS(T_1) = min([100, \infty[) = 100$.

Because we have one read-write conflict between the validating transaction T_1 and the active transaction T_2, the time interval of the active transaction must be adjusted: Thus $TI(T_1) = [100, \infty[\cap [0, 99] = []$. [3] Thus the time interval is shut out, and T_1 must be restarted. Howewer this restart is unnecessary, because history H_1 is acyclic, that is serializable. Taking the minimum as the commit timestamp ($TS(T_1)$) was not a good choice here.

We should select the commit timestamp $TS(T_v)$ in such a way that room is left for backward adjustment. We propose a new validation algorithm where the commit timestamp is selected differently. In our revised validation algorithm for OCC-TI (Figure 15) we set $TS(T_v)$ as the validation time if it belongs to the time interval of T_v, or maximum value from the time interval otherwise.

[3] If a and b are bounds in time interval as [a,b] and if b $<$ a then we define [a,b] = [] i.e. we do not allow undefined time intervals.

```
occti_validate(T_v)
{
    if validation_time ∈ TI(T_v) then
        TS(T_v) = validation_time;
    else
        TS(T_v) = max(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 15: Revised validation algorithm for OCC-TI.

This algorithm will generate a serializable history. Let H be the history generated by the algorithm, and let $SG(H)$ be the serialization graph. Now, if there is an edge $T_1 \rightarrow T_2$ in $SG(H)$ then the revised algorithm will always result in $TS(T_1) < TS(T_2)$ (Lemma 1 in [LS93]), thus satisfying the serializability theorem in [LS93].

Konana et al. [KLR97] offered an example history, which is claimed not to be serializable in the original OCC-TI algorithm. Using the same example, we show how our algorithm produces a serializable history and avoids an unnecessary restart. Consider transactions T_1, T_2 and T_3 and history H_1:

$T_1 : r_1[x]w_1[x]r_1[y]w_1[y]v_1$

$T_2 : r_2[x]w_2[x]...v_2$

$T_3 : r_3[y]...v_3$

$H_1 : r_1[x]w_1[x]r_2[x]r_3[y]w_2[x]r_1[y]w_1[y]v_1$

In this example T_1 reaches the validation phase first and has a *reconcilable conflict* with T_3 and an *irreconcilable conflict* with T_2 [LS93]. Therefore, it is required that transaction T_2 be restarted. Similary, T_3 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. Then all transactions are forward adjusted to $[100, \infty[$ during the read phase. Assume that transaction T_1 arrives to its validation phase at time 1000. Our revised OCC-TI algorithm sets $TS(T_1) as 1000$. Then all the conflicting transactions are adjusted. T_2's time interval is first adjusted to $[1000, \infty[$ using forward adjustment, and then to be $[1000, \infty[\cap [0, 999] = []$ using backward adjustment. Thus the time interval of T_2 is shut out and the transaction is restarted. Transaction T_3's time interval is adjusted to $[100, 999]$ using backward adjustment. Thus the revised OCC-TI algorithm produces a serializable history as well as avoids the unnecessary restart problem.

5.4 Revised OCC-DA (OCC-τDA)

In RODAIN we selected to use the OCC-DA algorithm [LLH95a] with Thomas's write rule [Tho79] as presented in [LLH95b] for concurrency control because it minimizes transaction restarts and decreases write-write conflicts. We modified the OCC-DA algorithm to exploit τ-serializability and semantic conflict resolution to decrease read-write conflicts.

In order to support τ-serializability we modified OCC-DA so that read operations are allowed to be τ time-units late without dynamic adjustment of the serialization order or conflict resolution. Semantic conflict resolution is used in the modified OCC-DA to decrease write-write conflicts between the validating transaction and active transactions. This modification can be seen in conflict resolution which differs from the original OCC-DA scheme. In conflict resolution we use the *behavior* and *importance* attributes of transaction defined in the RO-

DAIN data model [KNPR97]. The *behavior* attribute specifies which write-semantics (update or replace) the transaction has. The *importance* attribute is used to select which transaction is restarted when a conflict occurs. The conflict resolution algorithm is presented in figure 16. The modified OCC-τDA validation algorithm is presented in figure 17.

```
conflict_resolution(T_v, T_a)
{
     if Transaction_Behavior(T_v) != Replace  and
       Transaction_Behavior(T_a) != Replace  then
     {
          if Transaction_Importance(T_v) >=
          Transaction_Importance(T_a)  then
               restart(T_a);
          else
               restart(T_v);
     }
}
```

Figure 16: Conflict Resolution Algorithm Used in RODAIN

```
occ-τda_validate(T_v)
{
    BTlist(T_v) = ∅ ;

    SOT = min(validation_time, SOT(T_v));

    for ∀ D_p ∈ ( RS(T_v) ∪ WS(T_v) )
    {
        if D_p ∈ RS(T_v) and
        TR(T_v, D_p) − min(τ(T_v), τ(D_p)) > SOT(T_v) then
            restart(T_v);

        if SOT(T_v) < RTS(D_p) − min(τ(T_v), τ(D_p)) then
            restart(T_v);

        for ∀ T_a ∈ active_conflicting_transactions(T_v)
        {
            if SOT(T_a) ≥ SOT(T_v) then
            {
                if D_p ∈ ( WS(T_v) ∩ RS(T_a) ) and
                   SOT − ROTS(T_a) > min(τ(T_v), τ(T_a)) then
                      BTlist(T_v) = BTlist(T_v) ∪ T_a ;

                if D_p ∈ ( (RS(T_a) ∩ WS(T_a) ) ∩ ( RS(T_v) ∩ WS(T_v) ) ) and
                   SOT − ROTS(T_a) > min(τ(T_v), τ(T_a)) then
                     conflict_resolution(T_v, T_a);
            }
            else if D_p ∈ ( WS(T_a) ∩ ( (RS(T_v) ∪ WS(T_v)) ) ) then
                     conflict_resolution(T_v, T_a);
        }
    }

    SOT = min(validation_time, SOT(T_v));

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

    for ∀ D_p ∈ RS(T_v) ∪ WS(T_v)
    {
        if D_p ∈ RS(T_v) and SOT(T_v) > RTS(D_p) then
            RTS(D_p) = SOT(T_v) ;

        if D_p ∈ WS(T_v) and SOT(T_v) > WTS(D_p)
        {
            WTS(D_p) = SOT(T_v) ;
            install D_p to database;
        }
    }
}
```

Figure 17: Validation Algorithm in OCC-τDA

5.5 Implementation Issues

We use a locking mechanism to implement an optimistic protocol as proposed in [HSRT91]. 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, we select the transaction having the lowest value of the *importance* attribute to be restarted.

There are two possible implementations of the write phase: *serial validation-write* (OCCL-SVW) and *parallel validation-write* (OCCL-PVW) [HSRT91]. In serial validation-write validation phase and write phase is in one critical section. Parallel validation-write separates the validation phase and the write phase into two critical sections, thus providing greater concurrency. In RODAIN we use the parallel validation-write because our implementation is based on a multiprocess server.

6 Experimental Results

We have carried out a set of experiments in order to examine the feasibility of our algorithms in practice. All experiments were executed in the RODAIN prototype database running on a Pentium PC with the Chorus/ClassiX operating system. In particular, we examined:

1. How well the FN-EDF can provide computing resources to non-realtime transactions?
2. Which of optimistic algorithms studied provides the best overall performance ?
3. How well does our OCC-τDA algorithm perform when compared to the OCC-DA algorithm?

6.1 Configuration of Test System

The test environment is a reduced subset of the RODAIN architecture. The environment includes all necessary parts of the database kernel. However, the writing of log records was simulated as a constant delay. The test system contains four functional parts illustrated in figure 18.

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.

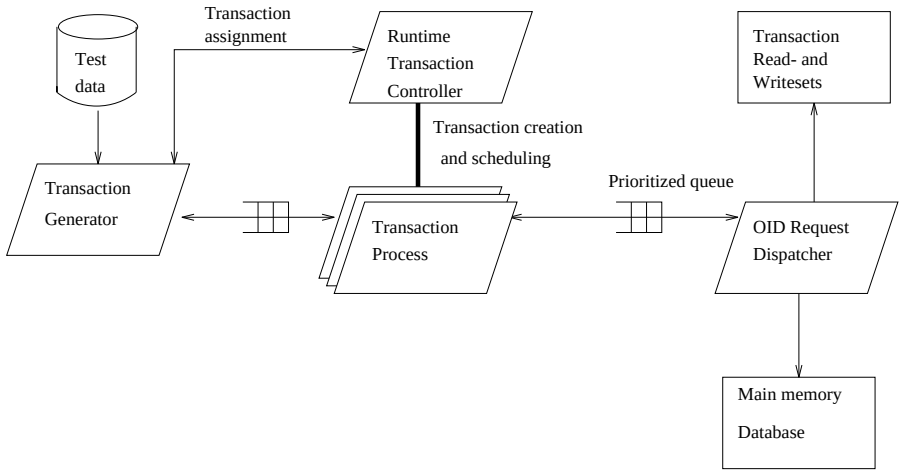


Figure 18: Processes in the Test Environment

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. Reported values are means of the repeated runs.

6.2 Test Database and Transactions

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.

We used three different types of transactions named R1, W1, and T1. Transaction R1 is a read-only service provision transaction that reads two objects and commits. Transaction W1 is an update service provision transaction that reads two objects, updates them and commits. Transaction T1 is a service management transaction with the replace semantics. It reads a specified fraction of objects and replaces them.

New transactions are accepted up to a prespecified limit, which is the number of installed Transaction Processes. If there are no Transaction Process available when a new transaction arrives, the transaction is aborted. Transactions are atomically validated. The order of the *importance* attribute, which is used in selecting the transaction to be restarted in conflict resolution, is $T1 > W1 > R1$. If the conflicting transactions have the same importance, 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 2).

Table 2: 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
Tau	mSec	10000	How old data a transaction can read
WriteProp	%	0–100	Write probability for the service management transaction

We report two performance metrics: abort/commit -ratio and miss-ratio. The abort/commit -ratio is used to figure out how a concurrency control scheme behaves when the workload increases. It is defined as

$$\text{AbortCommitRatio} = \frac{\text{Number of transactions aborted by concurrency control}}{\text{Number of completed transactions}} .$$

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 overload in the system. The miss-ratio is defined as

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

6.3 Scheduling Policy Experiments

The FN-EDF scheduling policy was designed to guarantee a prespecified fraction of computing resources to non-realtime transactions. In our scheduling policy experiments one T1 transaction with the read-only semantics is repeatedly executed. The transaction reads 50%

of the objects in the database and then commits. The workload also contains R1 transactions, the arrival rate of which is varied from 100 to 1000 transactions per second. The target value for the fraction of computing resources that non-realtime transactions (T1) should receive was 5%.

Figure 19 shows the number of completed T1 transactions. The reference method is pure EDF. The figure clearly indicates that the FN-EDF provides a stable fraction of computing resources to non-realtime transaction even when the system is highly loaded.

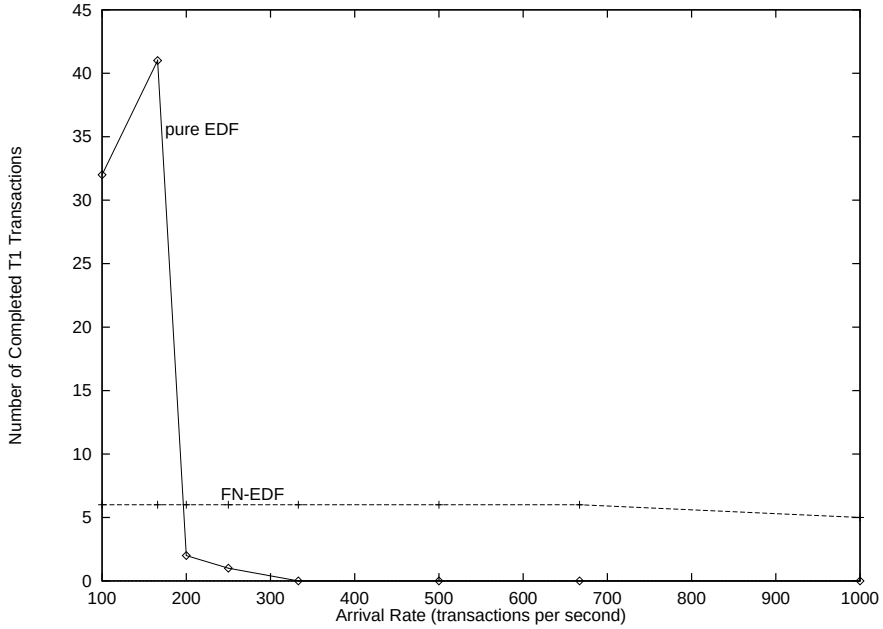
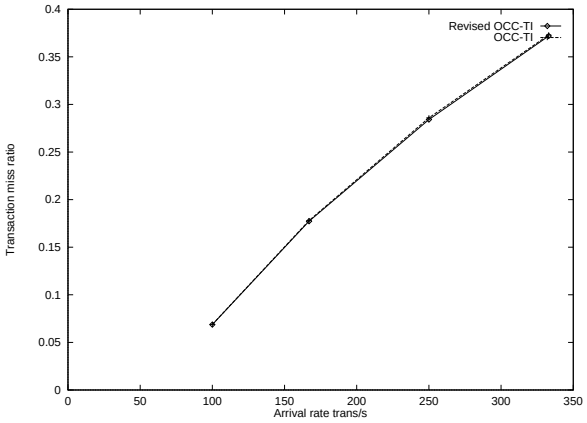


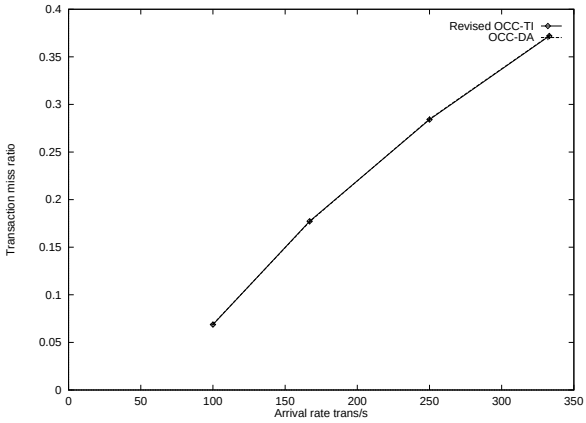
Figure 19: Transaction scheduling experiment

6.4 Service Provision Experiments

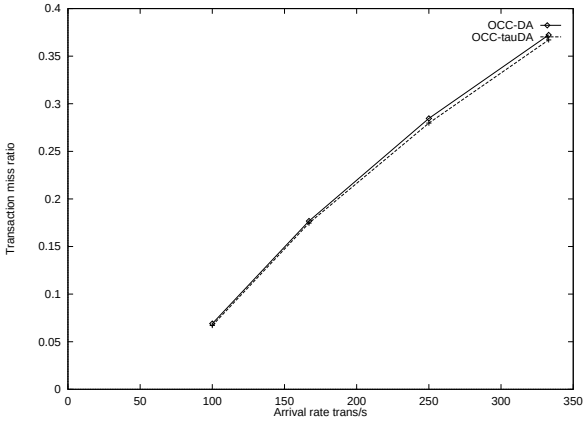
The first series of our test contained 60% of W1 and 40% of R1 transactions with 10mSec of think time before commit. Thus only service provision transactions were included. From Figure 20 we can see that using miss-ratio metrics there are only very small differences between the algorithms. A large percentage of transactions are rejected, because of the small number of available transaction processes (50). The rejection percentage covers algorithm differences. To see real difference between algorithm we calculated new metric abort/commit -ratio.



(a) OCC-TI and revised OCC-TI



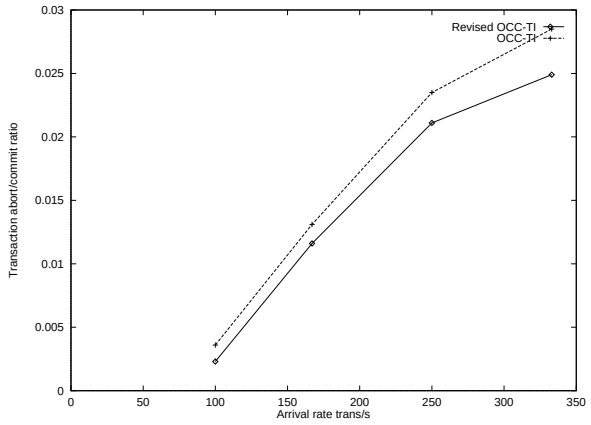
(b) Revised OCC-TI and OCC-DA



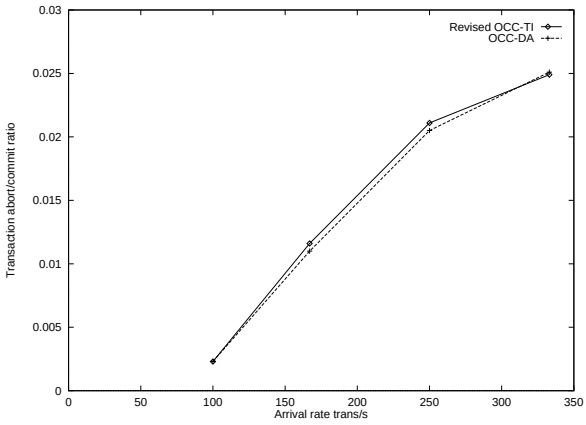
(c) OCC-DA and OCC-τDA

Figure 20: OCC algorithms compared with miss-ratio metrics.

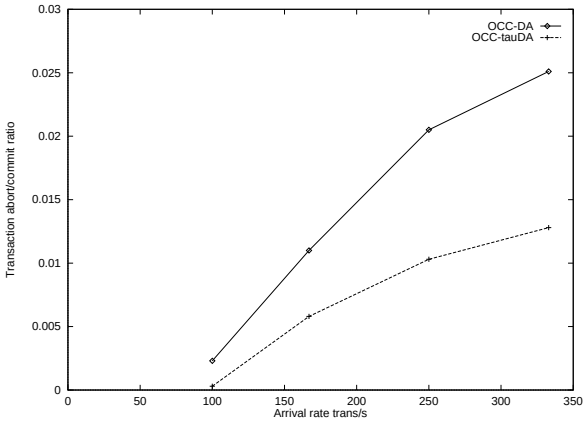
Figure 21 plots abort/commit -ratio metrics for the same transactions and algorithms as in Figure 20. From Figure 21 (a) we can see that the revised OCC-TI clearly outperforms OCC-TI. This is because the revised OCC-TI efectively avoids unnecessary restarts. From Figure 21 (b) we can see that there is only a small difference between the revised OCC-TI and OCC-DA. From Figure 21 (c) we can see that OCC-τDA clearly outperforms OCC-DA.



(a) OCC-TI and revised OCC-TI



(b) Revised OCC-TI and OCC-DA



(c) OCC-DA and OCC-τDA

Figure 21: OCC algorithms compared with abort/commit -ratio metrics.

In the second series of our test we varied the fraction of write transactions (W1) and the arrival rate of transactions was fixed to 250 transactions per second. As a result we wanted to know how the algorithms behaved in the case of a service provision type workload (1-10% write transactions) and how the percentage of write transactions affects the to performance of the

algorithms. Figure 22 shows the transaction abort/commit ratios for the revised OCC-TI and OCC-TI in the case of various percentages of write transactions. Figure 22 clearly shows that an increase in the fraction of write transactions increases the probability of transaction abort. The difference between the revised OCC-TI and OCC-TI is small when the fraction of write transactions is small. The difference between the revised OCC-TI and OCC-TI is biggest when the fraction of write transactions is 50%, and the difference decreases when the fraction of write transactions increases.

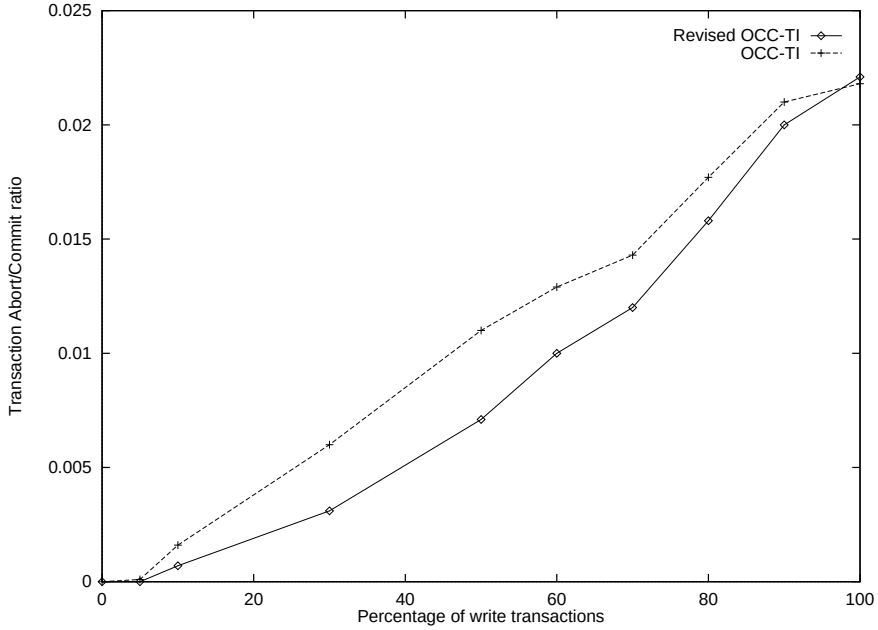


Figure 22: Comparison between revised OCC-TI and OCC-TI

Figure 23 shows the transaction abort/commit ratios the revised OCC-TI and OCC-DA for various percentages of write transactions. Figure 23 clearly shows that the difference between the revised OCC-TI and OCC-DA is small for all percentages.

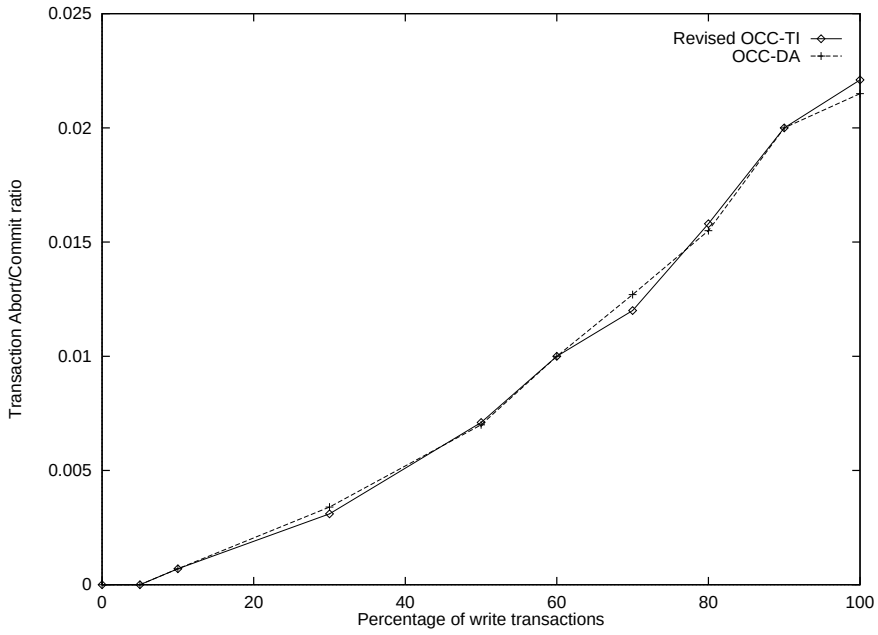


Figure 23: Comparison between revised OCC-TI and OCC-DA

Figure 24 shows the transaction abort/commit ratios for the revised OCC-DA and OCC-τDA for various percentages of write transactions. Figure 24 clearly shows that the difference between the revised OCC-DA and OCC-τDA increases when the fraction of write transactions increases. In OCC-τDA the abort/commit -ratio remains zero due to Thomas's write rule. In the future we have to decide if we can use Thomas's write rule in service provision type transactions. Thomas's write rule cannot be used if in some situations old writes must be installed to the database.

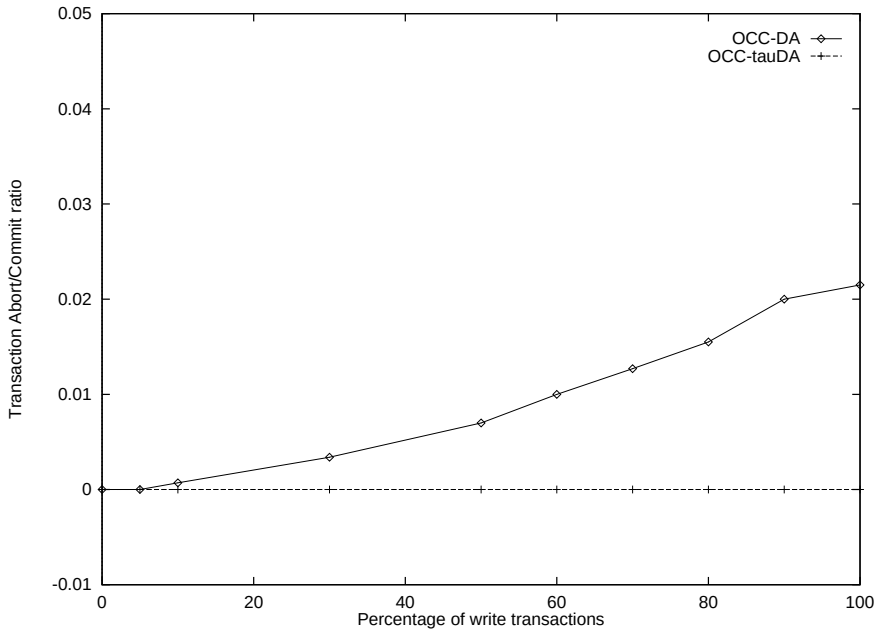


Figure 24: Comparison between OCC-DA and OCC-τDA

6.5 Service Management Experiments

A high data contention introduced by non-real time service management transactions often raises a high number of concurrency control aborts in traditional optimistic concurrency control protocols. In addition, the validation time of a transaction must be bounded in order to guarantee that real-time transactions will meet their deadlines. In these concurrency control experiments we examined how the execution of non-realtime service management transactions affects the database throughput. The fractions of transactions were: R1 89.8%, W1 10.0%, and T1 0.02%.

In the first experiment we compared the abort/commit -ratio of OCC-DA and OCC-τDA. Figure 25 indicates that OCC-τDA performs better than OCC-DA, especially when the arrival rate is high.

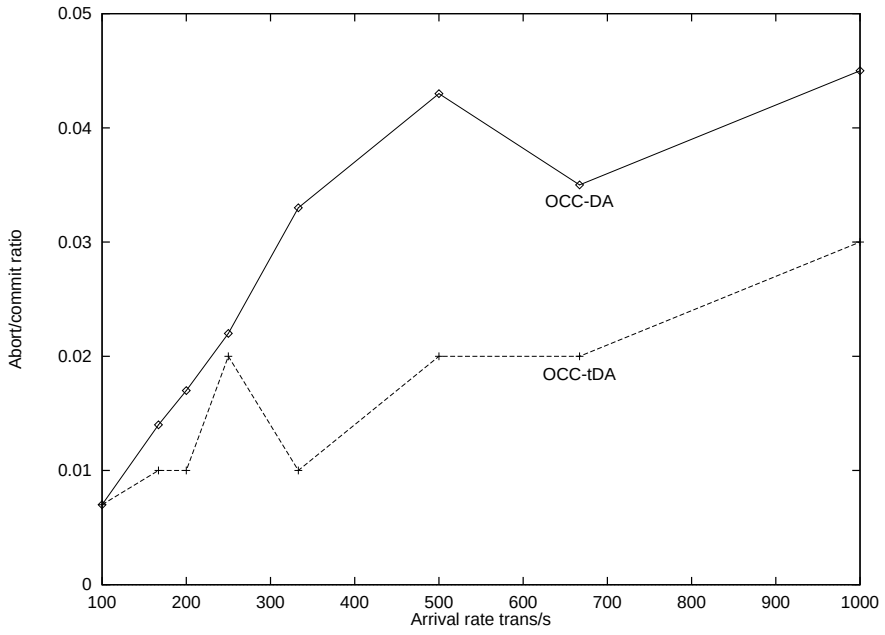


Figure 25: Comparison between OCC-DA and OCC-τDA

Figure 26 shows the miss-ratio for OCC-DA and OCC-τDA. As a reference we also show the miss-ratio for OCC-τDA without T1 transactions. The figure clearly indicates that the validation phase of massive updating transactions (T1) has a significant impact on the miss-ratio and hence to the system throughput. In this experiment the OCC-τDA does not show significant performance gains over OCC-DA. This is due to the fact that the proportion of write transactions (W1) is quite low and OCC-DA performs well when the write/read -ratio is low. It should also be noted that in this experiment the overload management policy dominates the result when the arrival rate is high.

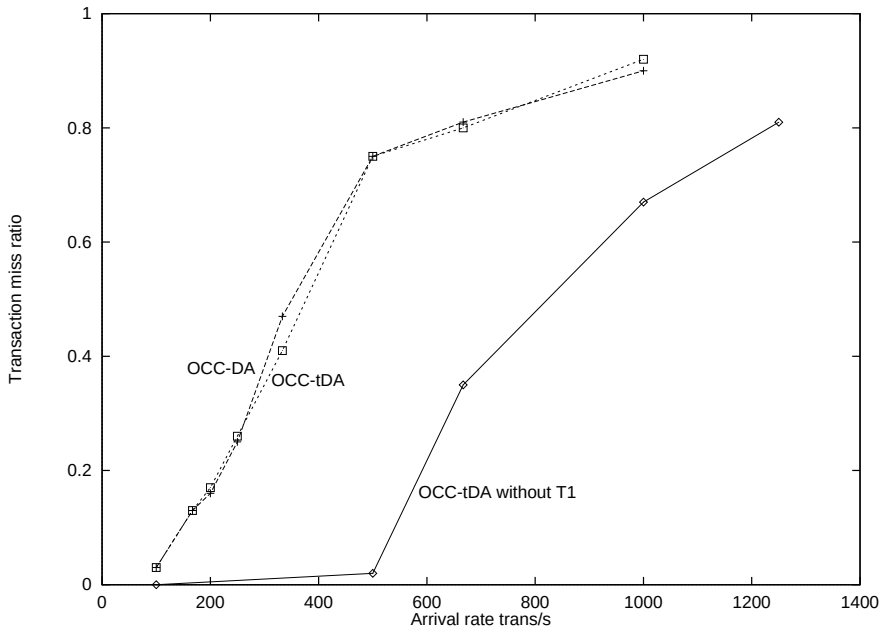
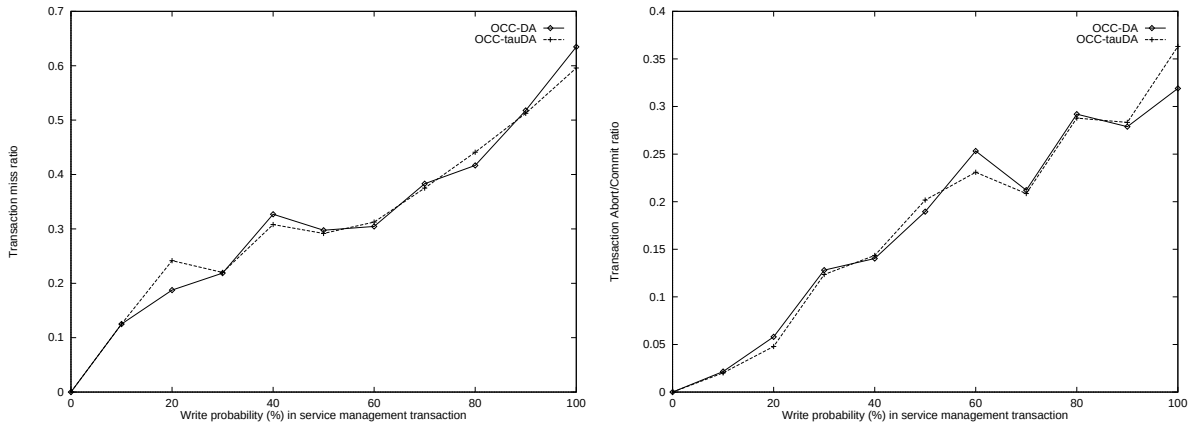


Figure 26: Impact of Validating Massive Transactions

Initially we planned to use the OCC-TI algorithm [LS93] because its validation process is shorter than in OCC-DA. Unfortunately, we found that the OCC-TI algorithm was not sufficient to our requirements because long transactions were always aborted in their read phase.

Figure 27 shows the transaction miss ratios as a function of data contention. In our experiment the arrival rate of transactions is fixed to 200 transactions a second. The test parameter is the write probability of transaction T1, varied from 0 to 100 per cent. Figure 27 clearly shows that an increase in the data contention due to the service management transaction increases the probability of transaction aborts. The main reason to this is the increase in the validation time of the management transaction.



(a) Miss ratio (b) Abort/Commit -ratio

Figure 27: OCC-DA and OCC-τDA compared.

Figure 28 shows fraction of completed service management transactions. In our experiment the arrival rate of transactions is fixed to 200 transactions a second. The test parameter is the write probability of transaction T1, varied from 0 to 100 per cent. Figure 28 clearly shows that an increase in the data contention due to the service management transaction increases the probability of service management transaction aborts.

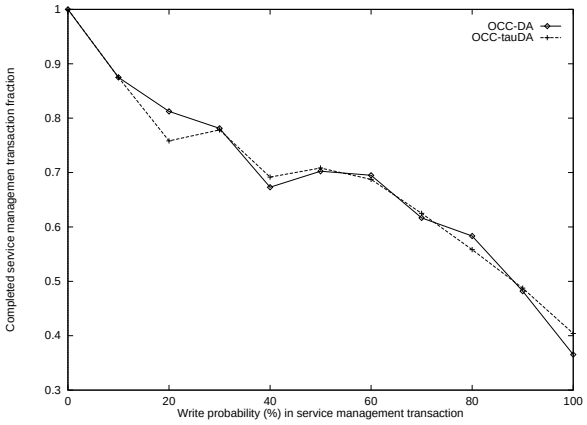


Figure 28: Impact of data contention.

7 Conclusions

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. We want 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 database architecture should be flexible due to different telecommunications needs. We believe that the required functionality is the easiest to implement with an object-oriented approach. An open question is if a full object-oriented architecture is needed. Currently we believe that we can use a reduced architecture that is more suitable for real-time databases.

Another interesting area is correctness in service provision transaction databases. In real-time applications concurrent transactions do not necessarily need serializability to produce a correct result. In many situations the result of a transaction can be regarded as correct as long as the result corresponds to the real-world situation presented in the database even if the transaction does not serialize with other transactions. Temporal consistency is often more important than the internal consistency obtained through serializability. Temporal consistency requires that data values used by a transaction have existed about the same time.

In telecommunication there are several database operations that need not to serialize. However, especially some service management transactions need serializability. An interesting correctness criteria, which we call τ-serializability, can be used to reduce the number of serialization conflicts. In τ-serializability a transaction is allowed to read old data as long as the update is not older than a predefined time.

We can gain several advantages when we combine real-time and object orientation. 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, correctness criterion, among other things. However, we must remember that the combining

real-time and object-oriented databases is an almost unexplored research area.

OCC-TI [LS93, Lee94] uses a novel approach to minimize the number of transaction restarts and wasted resources, thus providing better performance relative to OCC-FV. It eliminates transaction restarts for some category of conflicts. This makes OCC-TI a promising candidate for RTDBs by incorporating time sensitive conflict resolution. While the motivation behind OCC-TI is extremely important, the algorithms provided in [LS93, Lee94] suffer from a serious problem in that unnecessary restarts are caused. We provided a simple solution to this problem by changing the way how validation timestamps are chosen. The revised OCC-TI concurrency control protocol gains significant advantage over OCC-TI. The revised protocol was also studied in service provision workload, but we found it unsuitable for service management transactions. If OCC-TI is used, service management transactions should use versioning.

We also proposed extensions to the OCC-DA concurrency control algorithm. These extension included τ-serializability and semantic conflict resolution. The OCC-τDA concurrency control protocol gains no significant advantage over the OCC-DA in service provision workload. In this workload, conflicts are rare and OCC-DA behaves well. When more conflicts are in the workload, the OCC-τDA protocol gains significant advantage over OCC-DA.

The experiments indicated that the validation time of the service management transaction 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 service management transactions. An object can be inserted into the database to become valid at the later time. Insertion causes no conflict and thus validation of inserted objects is a short process.

References

[ABD+92] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonic. The object-oriented database system manifesto. In F. Bancilhon, C. Delobel, and P. Kannellakis, editors, *Building an Object-Oriented Database System - the Story of O2*, pages 3–20. Morgan Kaufman publishers, San Francisco, Calif., 1992.

[AGM88a] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions. *ACM SIGMOD Record*, 17(1):71–81, March 1988.

[AGM88b] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In F. Bancilhon and D. J. DeWitt, editors, *Proceedings of the 14th VLDB Conference*, pages 1–12, San Mateo, Calif., 1988. Morgan Kaufmann.

[Ahn94] I. Ahn. Database issues in telecommunications network management. *ACM SIGMOD Record*, 23:37–43, 1994.

[AKS93] M. Appeldorn, R. Kung, and R. Saracco. Tmn + in = tina. *IEEE Communications Magazine*, 31(3):78–85, March 1993.

[Cat97] R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kauffmann, San Francisco, Calif., draft of revision 2.0 edition, 1997.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[GM83] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[GRKK93] J. J. Garrahan, P. A. Russo, K. Kitami, and R. Kung. Intelligent network overview. *IEEE Communications Magazine*, 31(3):30–36, March 1993.

[H.92] Graham M. H. Issues in real-time data management. *The Journal of Real-Time Systems*, 4:185–202, 1992.

[H.93] Graham M. H. How to get serializability for real-time transactions without having to pay for it. In *Real-time System Symposium*, pages 56–65, 1993.

[Här84] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984.

[HCL90a] J. R. Haritsa, M. J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th Real-Time Symposium*, pages 94–103, Los Alamitos, Calif., 1990. IEEE, IEEE Computer Society Press.

[HCL90b] J. R. Haritsa, M. J. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 331–343. ACM, ACM Press, 1990.

[HCL91] 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.

[HLC91] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *Proceedings of the 12th Real-Time Symposium*, pages 232–242, Los Alamitos, Calif., 1991. IEEE, IEEE Computer Society Press.

[HSRT91] J. Huang, J. A. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the 17th VLDB Conference*, pages 35–46, San Mateo, Calif., September 1991. Morgan Kaufmann.

[ITU92a] ITU. *Information Technology – Open Systems Interconnection – The Directory: Abstract Service Definition. Recommendation X.511*. ITU, International Telecommunications Union, Geneva, Switzerland, 1992.

[ITU92b] ITU. *Management framework for Open Systems Interconnection (OSI) for CCITT applications. Recommendation X.700*. ITU, International Telecommunications Union, Geneva, Switzerland, 1992.

[ITU92c] ITU. *Principles for a Telecommunications Management Network. Recommendation M.3010*. ITU, International Telecommunications Union, Geneva, Switzerland, 1992.

[KLR97] P. Konana, J. Lee, and S. Ram. Updating timestamp interval for dynamic adjustment of serialization order in optimistic concurrency control-time interval (occti) protocol. *Information Processing Letters*, 63:189–193, 1997.

[KM93] T. Kuo and A. K. Mok. Application semantics and concurrency control of real-time data-insensive applications. In *Proceedings of Real-Time System Symposium*, pages 76–86, 1993.

[KNPR97] 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.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[KR96] 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.

[Lee94] 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.

[Lin89] K.-J. Lin. Consistency issues in real-time database systems. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 654–661, 1989.

[LL73] 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.

[LLH95a] K. Lam, K. 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, June 1995. Springer.

[LLH95b] K. Lam, K. Lam, and S. Hung. Real-time optimistic concurrency control protocol with dynamic adjustment of serialization order. In *Proceedings of IEEE Real-Time Technology and Application Symposium*, pages 174–179, Chigago, Ilinois, May 1995.

[LP96] K.-J. Lin and C. Peng. Enhancing external consistency in real-time transactions. *ACM SIGMOD Record*, 25(1):26–28, March 1996.

[LS93] J. Lee and S. H. Son. Using dynamic adjustment of serialization order for real-time database systems. In *Proceedings Real-Time Systems Symposium*, pages 66–75, Los Alamitos, Calif., 1993. IEEE, IEEE Computer Society Press.

[LS96] Y. Lee and S. 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.

[LSH97] K. Lam, S. Son, and S. Hung. A priority ceiling protocol with dynamic adjustment of serialization order. In *13th IEEE Conf. on Data Engineering (ICDE'97)*, Birmingham, UK, April 1997.

[MN82] D. Menasce and T. Nakanishi. Optimistic versus pessimistic concurrency control mechanisms in database management systems. *Information Systems*, 7(1):13–27, 1982.

[NKR97] 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.

[OMG92] OMG. *The Common Object Request Broker: Architecture and Specification*. Number Number 91.12.1. Revision 1.1 in OMG Document. John Wiley & Sons, New York, N.Y., 1992.

[Pou94] Dick Pountain. The chorus microkernel. *Byte*, pages 131–138, January 1994.

[Raa94] K. E. E. Raatikainen. Database access in intelligent networks. In O. Martikainen and J. Harju, editors, *Proceedings of IFIP TC6 Workshop on Intelligent Networks*, pages 163–183, Lappeenranta, Finland, August 1994. Lappeenranta University of Technology.

[RKMT95] K. Raatikainen, T. Karttunen, O. Martikainen, and J. Taina. Evaluation of database architectures for intelligent networks. In *Proceedings of the 7th World Telecommunication Forum (Telecom 95), Technology Summit, Volume 2*, pages 549–553, Geneva, Switzerland, September 1995. ITU.

[Rob87] J. Robinson. *Design of Concurrency Controls for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, December 1987.

[RP96] K. Ramamritham and C. Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), December 1996.

[RS94] 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.

[Sin88] M. Singhal. Issues and approaches to design of real-time database systems. *ACM SIGMOD Record*, 17(1):19–33, March 1988.

[SLJ88] L. Sha, J. P. Lehoczky, and E. D. Jensen. Modular concurrency control and failure recovery. *IEEE Transactions on Computers*, 37(2):146–159, February 1988.

[SLL92] S. H. Son, J. Lee, and Y. Lee. 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.

[SR90] J. Stankovic and K. Ramamritham. Editorial: What is predictability for real-time systems? *Real-Time Systems*, 2:247–254, 1990.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, C-39(9):1175–1185, September 1990.

[SZ88] J. A. Stankovic and W. Zhao. On real-time transactions. *ACM SIGMOD Record*, 17(1):4–18, March 1988.

[Tho79] R. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[TR96] J. Taina and K. Raatikainen. Experimental real-time object-oriented database architecture for intelligent networks. *Engineering Intelligent Systems*, 4(3):57–63, September 1996.