*ELECTRONIC* **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

**Rainer Manthey and Viacheslav Wolfengagen (Eds)**

# Advances in Databases and Information Systems 1997

Proceedings of the First East-European Symposium on Advances in Databases and Information Systems, (ADBIS'97), St Petersburg, 2-5 September 1997

# Concurrency Control Protocol for Nested Transactions in Real-Time Databases

Ekaterina Pavlova, Igor Nekrestyanov

Published in collaboration with the
British Computer Society





ISBN: 3-540-76227-2

# Concurrency Control Protocol for Nested Transactions in Real-Time Databases

Ekaterina Pavlova
St. Petersburg University
St. Petersburg, Russia

Igor Nekrestyanov*
St. Petersburg University
St. Petersburg, Russia

**Abstract**

In this paper we consider real-time concurrency control for the nested transaction model. We analyze problems that have pure optimistic and pessimistic approaches. As the solution we propose a hybrid concurrency control algorithm which acts as an optimistic for transactions from different transaction trees and as pessimistic inside a single transaction tree.

## 1 Introduction

Real-Time Database System (RTDBSs) are vital to a wide range of operations. As computers have become faster and more powerful, and their use more widespread, such databases have grown larger and more critical. Real-time database operations are characterized by their time constrained access to data, and access to data has temporary validity. Due to these unique characteristics, the performance goals of RTDBSs are different from conventional database systems, and RTDBSs require new transaction management techniques to satisfy the performance goals.

In the RTDBSs, the primary performance criterion is the timeliness level rather than the average response time or throughput, and scheduling of transactions is driven by priority rather than fairness considerations [12]. Depending on the kind of deadlines, RTDBS are divided into three groups – hard, firm and soft deadline systems. Taking into account these differences, considerable research has recently been devoted to designing concurrency control for RTDBSs and to evaluating their performance [11, 15, 14, 16].

Most of these studies assume the flat transaction model and use serializability [5] for maintaining data consistency. They also consider time constraints associated only with transactions. During the last few years, research has been done that attempted to use different assumptions. Some works were devoted to the systems where not just transactions, but also data has time constrains [18, 19]. Some attempts were made to use other consistency criteria like $\epsilon$-serializability [17].

Another direction of research is to design concurrency protocols for transactions with possible dependencies. This means that in the case of abort of one of the set of depended transactions, all others might be aborted too. Examples are the transaction chains [4] and transaction model for handling composite events [8] that are widely used in active database systems [3].

We consider the case of nested transaction model proposed by Moss [10]. There are some different nested transaction types depending on the types of possible lock modes, relationships between relatives locks, lock inheritance, parent reaction to transaction abort. A good taxonomy of the nested model variations can be found at [6]. We do not focus on specific variants and consider the general case with the only one limitation – all subtransaction are essential. This mean that the parent could commit only if all of its subtransactions have been committed.

*This work was partially supported by the Russian Foundation for Basic Research under grant 95-01-00636, and UrbanSoft Ltd. under contract 35/96

In general there are two reasons why concurrency control may cause decreasing of performance. One is the wasted time, and other is wasted work. And there are two approaches to concurrency control – optimistic and pessimistic. The first prevents conflicts at the expense of wasted time, and the second gives chance to all transactions at the expense of wasted work. In RTDBS, the optimistic approach is usually better than the pessimistic as shown [7].

The classical protocol for nested models in the conventional database is the two phase locking protocol [5] (or two phase commit for distributed system), which is certainly pessimistic. In real-time systems it is tempting to use the optimistic approach, but as we found pure optimistic approach has some limitations in the case of nested model due to its specifics. The problem is the wasted work during parallel execution of conflicting dependent transactions. The parallel execution of such transactions could not give any advantages to any transaction because their updates become permanent only when the enclosing top-level transaction commits. Our idea is to build a hybrid protocol to address this problem.

The rest of this paper is organized as follows: the next section briefly describes optimistic and pessimistic approaches and highlights their weak places. We describe kinds of nested models and specifics of model under the consideration in the Section 3. Thus we will show the problems of using the pure optimistic or pessimistic concurrency control for the nested model. The next section introduces our hybrid approach. We discuss it at Section 6. In the last section we conclude and summarize our results.

# 2 Real-Time Concurrency Control Protocols

The most important characteristic of the concurrency control protocol is performance. In conventional database systems performance is usually measured as the number of transactions per second. In real-time databases, performance depends on many other criteria, which are related to real-time. Some of these criteria are the number of transactions that missed their deadlines, average tardy time, etc. Due to new goals of optimization the algorithms that are used in the conventional database systems do not show best results. Recently a lot of research that study possible protocols and compare different approaches are being studied. Here we briefly cite their results.

All concurrency control protocols may be divided into two groups – optimistic and pessimistic approaches.

## 2.1 Pessimistic Approach

The main feature of the pessimistic approach is to prevent possible conflicts. Transactions get access to data only if this will not cause possible conflict situation later. If it is not possible immediately the transaction should wait until it will become possible.

Most of pessimistic algorithms are based on locks. The classical pessimistic algorithm is the widely used two phase locking (2PL) [5]. Real-time systems usually use modified versions of 2PL, such as 2PL-HP, 2PL-WP, 2PL-PC [1, 2, 13]. These protocol modifications take into consideration real-time aspects, such as transaction priorities. According to 2PL-HP, conflict situations are resolved in favor of the transaction with higher priority. When a transaction requests a lock on an object held by an other transaction in a conflicting lock mode, if the requesters priority is higher than that of all the lock holders, the holders are restarted and the requester is granted the lock; if the requesters priority is lower, it waits for the lock holders to release the lock [1]. If all transactions have different priorities such algorithm will also prevent deadlocks.

To summarize, the pessimistic approach tries to conserve resources by minimizing the amount of potentially useless work. But this causes not full usage of available resources.

## 2.2 Optimistic Approach

In case of optimistic approach transaction execution consists of three phases: read, validation, write. During the read phase a transactions work in parallel without any verification and write to their own local space. The

validation phase is the check-up of existing conflicts. After a successful validation it is possible to commit the transaction and copy its local space to the database.

Depending on the kind of examination during validation phase all optimistic protocols can be divided into two classes – forward validation [9] and backward validation [11] protocols. Forward validation commits finished transaction and aborts still working that conflict with it. Backward validation checks for conflicts of finishing transaction with committed transactions and abort transaction if such conflicts exist. It doesn't check for conflicts with still working transactions.

To summarize, according to the optimistic approach the load of the system is maximized, but part of work is useless.

### 2.3 Optimistic vs Pessimistic

Some research on experimental and theoretical comparison of both described approaches has been done. As was shown in conventional database management systems, pessimistic approach outperform optimistic one [7, 14].

In the context of the real-time databases, preventing possible conflicts is the weak point of the pessimistic approach. It causes useless transaction idle time and as a result increases the number of transactions that missed their deadlines. The wait is useless if the transaction that caused this wait will abort or will miss its deadline. Another negative point is that transaction aborts happen more often in real-time than in the conventional case due to aborts because of missed deadlines. Aborts cause rollbacks, which are not easy operations for pessimistic approaches.

From this point of view the optimistic approach looks more advantageous because it tries to execute all transactions simultaneously, which gives a better chance for one of the conflicting transaction to commit. Negative side of this is necessity of time consuming validation phase and embezzlement of resources for parallel execution and restart of transactions from which only one can be committed.

Relative estimations for these two approaches in the case of the flat transaction model in RTDBS's show that in most of situations optimistic approach outperform pessimistic. Especially if rules for transactions that missed their deadline are strict – firm and hard [7, 15].

## 3 Transaction model

### 3.1 Nested model

In this model each transaction may initiate any number of subtransactions, and each of these subtransactions may initiate any number of its own subtransactions. Any subtransaction starts after its parent's start and should terminate before parent's termination. A transaction is not allowed to commit until all its children have terminated. However, if child fails, its parent is not required to abort. Instead, the parent is allowed to perform its own recovery. In order to meet its goal, the parent may:

1. ignore the condition.
2. choose to retry the subtransaction.
3. initiate another subtransaction (a *contingency subtransaction*) that implements an alternative action.
4. abort (then its goal cannot be accomplished within a reasonable amount of time).

The ACID-properties are fulfilled for top-level transactions, while only a subset of them are valid for subtransactions. Subtransactions appear atomic to the surrounding transactions and may commit or abort independently. If the concurrency control scheme introduced by Moss [10] is applied, isolated execution is guaranted for subtransactions. The durability of the effects of a committed subtransaction depends on the outcome of its superiors – even if it commits, abort of one of its superiors will undo its effects. Updates of

| Transaction | arrival time | exec time | read | write |
|---|---|---|---|---|
| $A$ | 0 | 15 | — | $x_1$ |
| $A1$ | 10 | 15 | $x_3$ | $x_2$ |
| $A2$ | 15 | 20 | $x_2$ | $x_3$ |
| $B$ | 25 | 15 | $x_1$ | — |

Figure 1: Summary of considered example

a subtransaction become permanent only when the enclosing top-level transaction commits. The consistency property for subtransactions seems to be too restrictive as sometimes a parent transaction needs the results of several child transactions to perform some consistency preserving actions.

### 3.2 Our Model

In general we consider the generic nested transaction model. We don't focus on such characteristics as open or closed model, because this is not important for us.

The specific feature of model under consideration is that all subtransactions are essential. We also assume that a parent transactions tries to restart its subtransactions in case they failure. The restart is possible if the deadline of this subtransaction (or one of its children) is not missed. In the opposite case the parent transaction should be aborted and hence all transaction tree will be aborted too, because all transactions are essential and one of them already missed it's deadline.

## 4 Motivations

In section 2.3 we argued why the optimistic approach is better than pessimistic for flat transactions, but the nested model requires additional consideration.

In this section we will discuss behavior of pure pessimistic and optimistic approaches. As the basis for discussion we will use the following example.

### 4.1 Example

We will consider scheduling of transactions on a two processor system. This example consists of two transaction trees. Every transaction is characterized by arrival and execution times[1], read and write sets.

The first transaction tree has parent transaction $A$. This transaction writes data element $x_1$ and starts two subtransaction $A1$ and $A2$. First of them reads $x_2$ and writes $x_3$. The second reads $x_3$ and writes $x_2$. Obviously $A1$ and $A2$ are conflicting and could not work in parallel. Suppose that $A$ came to the system at the time 0 and forked $A1$ and $A2$ after 10 and 15 time units respectively.

The second transaction tree consists of only one transaction $B$. This transaction only read $x_1$. Suppose that the priority of $B$ is higher than the priority of $A$[2].

This is summarized in the table 1.

In our example we assume that a processor scheduling procedure always selects the highest priority transaction for execution and transaction start to work on the first free processor.

[1] It is only processor time is expended on the transaction, not all time is spended by transaction in the system

[2] Due to deadlines or for some other reasons.

## 4.2 Pure pessimistic approach

As the pessimistic algorithm we consider a direct adaptation of the 2PL for nested model to the real-time case. The modification takes into consideration real-time aspects, such as transaction priorities. Let's consider the conflict resolution according to the *high priority* rule as it was proposed for 2PL-HP [1].

According such algorithm we get the schedule shown in the Figure 2.

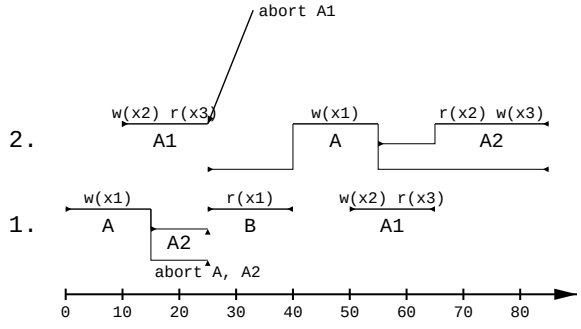


Figure 2: Schedule for pure pessimistic approach.

Transactions $A$[3] and $A1$ run until the appearance of $B$. Transaction $B$ needs to get a read lock for $x_1$ but a conflicting kind of lock is already held by $A$. Due to higher priority of $B$, according to the *high priority* rule, transaction $A$ should be restarted. This causes abort of $A1$ and $A2$ too.

After restart $A$ is forced to wait for the write lock of $x_1$ until the finish of $B$. Hence it could really start to work again only at the time 40.

The pessimistic approach also prevents parallel execution of $A1$ and $A2$ because they are conflicting.

Overall length of schedule in this case is 85 and it uses 95 processor time units.

## 4.3 Pure optimistic approach

Consider forward validation optimistic algorithm[4]. The schedule produced by it is shown at Figure 3.

According to this algorithm transaction $A1$ and $A2$ will work in parallel until the finish of $A1$. During the validation phase of $A1$ the conflict with $A2$ will be discovered and $A2$ will be aborted and restarted. After restart $A2$ will work in parallel with $B$. During the validation phase for $B$ it will be serialized before $A$.

Hence the overall length of schedule will be 45 and require 75 processor time units.

## 4.4 Discussion

Consider the situation when every transaction tree consists of the one transaction. In that case the nested model is simplified to usual flat transaction model. And, as it was explained in section 2.3, the optimistic approach is usually better.

The example under consideration clearly demonstrates this. The main problem of classical real-time pessimistic approaches is useless restarts. The situation is even worse for the case of the nested model due to useless aborts of the whole transaction trees. In our example transaction $B$ causes abort of tree $A$ because of *potential* conflict. Although in reality $B$ could be serialized before $A$ (or $B$ could be aborted due to missed deadline or internal reasons).

[3] Transaction $A$ start to work on processor $\#2$ because at the time 25 this processor is free.

[4] For this example backward validation algorithm has worse results.

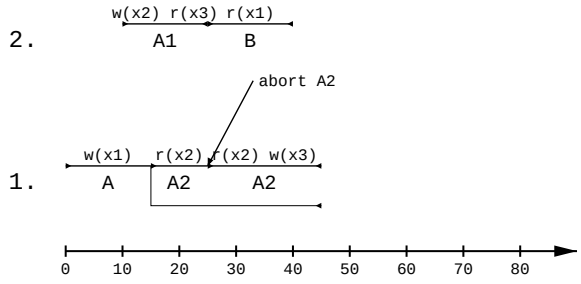


Figure 3: Schedule for pure optimistic approach.

Although optimistic algorithm shown better results for this example it also has some problems. Let's look at transactions from the one transaction tree. The model under consideration requires that all these transactions should be committed or all should be aborted[5]. If we execute two conflicting transactions from the same transaction tree in parallel then we will need to restart one of them[6]. But all updates made by committed transaction should not become visible for the outside transactions until the other one will be committed, too. Hence the time of commit will be the same as in the case of sequential execution. But in the last case we save resources and give better chances to another transactions from the same and other transaction trees. Therefore there is no reason to run conflicting transactions from the same tree in parallel. This problem is demonstrated by restart of $A2$ due to parallel work with $A1$ in Figure 2. The processor time wasted by $A2$ could be used by another transaction with lower priority and, therefore, cause overall increase of performance.

If we consider the case of only one transaction tree then the usage of the pessimistic approach avoids wasted resources because it prevents parallel execution of conflicting transactions. The pessimistic algorithm in our example demonstrates this behavior during the time interval $[40, 85]$. Therefore in such situations the pessimistic approach has the advantage.

On the other hand, parallel execution of transactions from different transactions trees gives them an extra chance to meet their deadline.

Summarizing the above discussion it is evident that applying both of these pure approaches to the nested model has its own merits and demerits. The idea is to use an intermediate hybrid strategy, which uses an optimistic approach for transactions from different trees, and a pessimistic one within one tree.

# 5 Hybrid Concurrency Control for the Nested model

## 5.1 Basic Concepts

The idea of the algorithm is that the transaction tree is serialized with another transaction using the optimistic algorithm, but transactions within the same tree are serialized using the pessimistic algorithm.

We will try to serialize transaction trees using the optimistic approach just as if they were flat transactions. For distinguishing the role of the transaction tree as the unit controlled by optimistic algorithm we will call it – *global transaction*[7]. Note that the transaction tree can consist of one transaction.

[5] Because all transactions are essential.

[6] Actually this is true not only for this case. In general it is forbidden to commit more than one transaction from the the set of conflicting transaction worked in parallel.

[7] This has nothing in common with global transaction in multidatabase systems.

If the optimistic algorithm is used then each transaction has its own local workspace. A transaction reads from the DB and writes to its local workspace. Before propagating updates to the DB, a validation test is performed to check database consistency. If the result of the test is positive, updates of the transaction become permanent and the transaction commits.

In our case we use local workspaces for every transaction tree. That's why updates of all transactions from the same tree are propagated to the DB simultaneously. To get access to data, a transaction should request a lock for copy of this data in the local workspace of its transaction tree. Lock handling is controlled using the pessimistic algorithm. From the point of view of this algorithm, the local workspace of the controlled transaction is the database.

## 5.2 The Algorithm

We describe our algorithm by explaining its behavior in common situations:

- **Start of the new global transaction**

  When a new global transaction arrives to the system it is registered and get its own local workspace. This is done according to optimistic approach but further management of this global transaction is controlled by the pessimistic protocol. Consequently from the point of view of the pessimistic protocol, the local workspace is the database and there is only one nested transaction which works with this database.

- **Start of the new subtransaction**

  A new subtransaction is registered according to the rules of pessimistic protocol as in the conventional case. There is no need to do anything in the context of the optimistic protocol.

- **Read operation**

  A transaction should get a lock on replica of data in the local workspace before reading it. If this replica doesn't exist, then data is copied to the local workspace from the real database and then the transaction gets the lock. Reading from the DB to local workspace is registered in the context of the optimistic protocol as action from the corresponding global transaction.

  If such replica already exists, then the request for a lock is processed according to pessimistic protocol without participation of optimistic.

- **Write operation**

  In case of data update the actions are analogous to the previous situation. The difference is that the transaction sets write lock instead read.

  All updates of a transaction are saved to the local workspace and become accessible by other transactions from the same tree after the successful commit of this subtransaction.

- **Subtransaction commit**

  The commit of a subtransaction is performed according to rules of pessimistic protocol for nested transaction model.

- **Global transaction commit**

  The commit of a top-level transaction causes an attempt to commit corresponding global transaction in the optimistic context. For this purpose management is returned to the optimistic protocol which tries to commit this global transaction. The validation phase is processed the same way as in case of flat transactions.

  During this phase a validation test of the transaction is performed to check whether updates made by the global transaction preserve database consistency. If the result of the test is positive, the transaction becomes permanent and the transaction commits. Otherwise, the transaction is aborted and restarted if possible.

- **Subtransaction abort**

  All changes of failed subtransaction are rolled back using the standard mechanisms of the specific pessimistic protocol. The parent tries to retry this transaction. If this is not possible because of missed deadline then parent should abort itself.

  If a top-level transaction is aborted then this causes an abort of corresponding global transactions in the optimistic context.

- **Global transaction abort**

  The optimistic protocol handles this situation according to its own rules. Usually it only frees the local workspace and unregisters this transaction.

### 5.3 Implementation Issues

The implementation of this technique may be based on interaction of two kinds of transaction managers. One of them uses optimistic strategy and the other one – pessimistic.

All the new transaction trees are initially controlled by the optimistic manager, which further passes management of this tree to the pessimistic manager. This pessimistic manager controls the execution of transactions from this tree until the commit of whole tree.

It is possible to have a separate pessimistic manager for each transaction tree or the same pessimistic manager which will control all of the trees. In any case it will work with every tree separately.

Also it is possible to have simultaneously different kinds of pessimistic managers. The decision of which one should be used is made by the optimistic manager according to the type of new transaction. It seems tempting to use specific protocols for handling read-only, flat and nested transactions but this requires additional research.

Because the interaction of two different transaction managers may lead to new performance problems it is possible to have one optimistic manager which will to handle locks of transactions from the transaction tree itself. This means that when transaction from tree starts to work it locks data in corresponding mode[8] (if it is possible), reads data from the DB (or from local workspace) and writes data to local workspace.

## 6 Discussion

At first glance, it seems that the proposed algorithm reduces the overall degree of concurrency because it doesn't allow inter-leaf of transaction trees at the serialization order. This means that if $A_1A_2$ are subtransaction from one tree and $B_1B_2$ - from another tree, serialization order $A_1B_1A_2B_2$ is not permitted. But actually this order isn't permitted by most other algorithms to avoid cascading aborts.

Therefore our algorithm does not restrict the set of possible serialization orders in comparison with other algorithms. At the same time it allows saving of resources without wasting extra time (see section 4). Hence this algorithm looks advantageous for real-time systems.

Let's look to the schedule produced by our algorithm for the example from section 4.1. We will use forward validation algorithm as optimistic one. The schedule is shown at the Figure 4. The result has the same length as in the case of pure optimistic approach but requires fewer processor time units (65 instead 75).

The weak point of the proposed algorithm is the restart of the tree after discovering of the conflict. Actually in most cases there is no need to restart the whole transaction tree and it is possible to eliminate conflict by restarting only some currently active subtransactions[9] of the tree in the case of early conflict discovering. But restarts of subtransactions require their rollbacks which are expensive for the pessimistic approach and sometimes it seems cheaper to restart a whole tree instead of performing a huge rollback. This problem still requires additional work on quantitative analysis.

[8] There is lock table for each transaction tree.

[9] This subtransactions do not necessary conflict themselves, but their committed children do. And restart of such transaction cause rollbacks and restarts of actually conflicted.

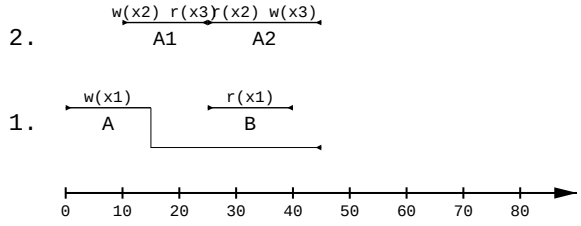


Figure 4: Schedule for hybrid approach.

The early discovering of conflicts allows to reduce amount of transaction processing and work for restart. It also provides a better chance for transactions that should be restarted to meet their deadlines. From this point of view it is better to use forward validation algorithm instead backward as optimistic one.

If currently used pessimistic protocol has troubles with deadlocks then some mechanisms for their detection are used. This selection of specific method of the deadlock detection depends on the concrete situation but we would like to mention that in our case deadlocks can only occur among transactions from the same transaction tree. Therefore the check for deadlocks will not be very complex due to the limited amount of participating transactions.

In general we don't restrict the set of possible concurrency control protocols which can be used. This gives extra functionality to select the best algorithms depending on the concrete situation. It is also possible to use some pessimistic algorithms at same time for different kinds of transaction. For this, the optimistic manager should have some preliminary information about possible kinds of transactions.

# 7 Conclusion and Future Work

In this paper we study concurrency control in the nested transaction model in real-time systems. Considered model has only one restriction comparing with the general case – all subtransaction are essential. The a nalysis of pure pessimistic and optimistic concurrency approaches highlights their weak sides. Thus, a hybrid algorithm was proposed in this paper.

The proposed algorithm acts as optimistic for transactions from different transaction trees and as pessimistic for transactions inside one tree. We are not restricting the types of optimistic and pessimistic algorithms which can be used allowing selection of the best alternatives according to the specific features of the system.

Additional work is required on investigation of the problem of partial restarts and on quantitative analysis of costs of both approaches – with and without partial restarts. Additional work is required for the general case if not all subtransactions are essential.

# References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. In *Procedings of the 14th VLDB Conference*, August 1988.

[2] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Procedings of the 15th VLDB Conference*, August 1989.

[3] S.F. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring. Deeds towards a distributed and active real-time database systems. *ACM Sigmod Record*, 15(1):38–40, March 1996.

[4] Erdogan Dogdu. Scheduling adaptive transactions in real-time database systems. In *Proceedings of the 7th International Conference and Workshop on Database and Expert Systems Applications (DEXA"96)*, Zurich, Switzerland, September 1996.

[5] K. Eswaran, J. Gray, R. Lorie, and I. Traiger. The notions of consistency and predicate locks in a database system. In *ACM*, November 1976.

[6] Theo Harder and Kurt Rothermel. Concurrency control issues in nested transactions. *VLDB*, 2(1):39–74, January 1993.

[7] Lee Juhnyoung and H. Son Sang. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Prentice-Hall, 1996.

[8] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and M. Sakkinen. A transaction model for handling composite events. In *Proceedings of the Third International Workshop on Advances in Databases and Information Systems (ADBIS"96)*, 5-7262-0064-0, pages 116–125, Russia, Moscow, September 1996.

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

[10] J.E.B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT Press, Cambridge, MA, 1985.

[11] P.Krzyżagórski and T.Morzy. Optimistic concurrency control algorithm with dynamic serialization adjustment for firm deadline real-time database systems. In *Proceedings of the Second International Workshop on Advances in Databases and Information Systems (ADBIS"95)*, volume 1, pages 21–28, jun 1995.

[12] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(1), 1992.

[13] L. Sha, R. Rajkumar, and J.P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD Record*, March 1988.

[14] O. Ulusoy. Analysis of concurrency control protocols for real-time database systems. Technical Report BU-CEIS-9514, Bilkent University, 1995.

[15] O. Ulusoy. Research issues in real-time database systems. *Information Sciences*, 87(1-3), November 1995.

[16] O. Ulusoy and G.G. Belford. A performance evaluation model for distributed real-time database systems. *International Journal of Modeling and Simulation*, 15(2), 1995.

[17] K. Wu, P. Yu, and C. Pu. Divergence control for epsilon serializability. *Proceedings of the 8th IEEE International Conference on Data Engineering*, February 1992.

[18] Ming Xiong, Raju Sivasankaran, Jack Stankovic, Krithi Ramamritham, and Don Towsley. Scheduling transactions with temporal constraints: Exploiting data semantics. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, December 1996.

[19] Ming Xiong, Raju Sivasankaran, Jack Stankovic, Don Towsley, and Rajendran Sivasankaran. Maintaining temporal consistency: Issues and algorithms. In *The First International Workshop on Real-Time Databases*, California, March 1996.