

A Transaction Processing Technique in Real-Time Object-Oriented Databases

Woochun Jun

Dept. of Computer Education Seoul National University of Education Seoul, Korea

Summary

In this paper, transaction processing technique is presented for real-time object-oriented databases (RTOODBs). At first, a RTOODB model is proposed. The proposed model is based on imprecise computation and an affected set of attributes so that it can provide trade-off between temporal consistency and logical consistency. Besides temporal consistency, more concurrency is also can be achieved and deadlock due to lock escalation can be reduced. Also, a real-time priority-based conflict resolution scheme is proposed. The proposed scheme has following advantages: It aborts earlier transactions whose deadlines cannot be met. It also avoids chained blocking and deadlock. In additions, if one of lock requester and lock holder must miss its deadline due to conflict access mode, the one with less work done so far is selected as a victim so that system resource is not wasted.

Keywords:

Real-time, object-oriented, transaction processing

1. Introduction

Interests in RTOODBs have been growing for time-critical object-oriented database (OODB) applications such as stock-trading system. Many works have been done for real-time transaction processing techniques in conventional databases [1,2,3,4]. But, researches for transaction processing in RTOODB need to be focused. Especially, only a few transaction processing techniques for RTOODB have been proposed [5,6].

In this paper, a transaction processing technique for RTOODBs is proposed. The technique includes a RTOODB model and a real-time priority-based conflict resolution scheme. The model is based on imprecise computation [7] and affected set of attributes [8]. The proposed model can provide trade-off between temporal consistency and logical consistency. It can also provide higher concurrency among methods and reduce deadlock by adopting run time information.

In existing conflict-resolution schemes for real-time database, they try to reduce or avoid *priority inversion* problem, where a high priority transaction is blocked by one or more lower priority transactions due to conflict access on resources, as follows: if a high priority transaction requests an access to data object which is

locked by a lower priority transaction with conflict lock mode, the higher priority transaction aborts a lower priority transaction or is blocked as long as the lower priority transaction can finish within slack time of the higher priority transaction. But, when a lower priority transaction requests an access to data object which is locked by a higher priority transaction with conflict lock mode, the lower priority transaction is always blocked regardless of its deadline so that it may miss deadline and waste work done so far.

In this work, we consider the situation which blocking a lock requesting lower priority transaction misses its deadline so that we try to reduce the number of transactions missing deadline and reduce wastes on system resource and time

This paper is organized as follows. In Section 2, problem statement for this research is presented. In Section 3, the proposed RTOODB model is presented. Also, the proposed real-time conflict resolution scheme is presented in Section 4. Finally, conclusions and further works are discussed in Section 5.

2. Problem Statement

Several RTOODB models have been proposed in the literature [5,6,9,10,11]. The imprecise computation concept was introduced in RTOODB model in [6]. The imprecise computation is to give the user an approximate result of acceptable quality whenever the system cannot produce the exact result in time. Also, it is based on the argument, "in real-time systems, it is better to produce a partial result before the deadline instead of the complete result after deadline." [7]. But, the authors in [6] do not provide how to construct methods from imprecise computation concept.

In this work, a formal way to construct methods and commutativity tables among methods is presented based on imprecise computation.

Many priority-based conflict resolution schemes are proposed [2,3,4]. Two representative approaches are priority abort scheme [2] and priority inheritance scheme [3], although many variations are proposed from two schemes. In priority abort scheme, whenever a high

priority lock requesting transaction conflicts with lock holding lower priority transaction, the high priority transaction aborts the lower priority transaction. If a lower priority lock requesting transaction conflicts with high priority lock holding transaction, the lower priority transaction is blocked. It can eliminate the priority inversion problem completely. But, it may cause unnecessary aborts since, in some case, the lock requesting transaction can wait until the lock holder is finished. On the other hand, in priority inheritance scheme, a lock requesting transaction is always blocked whenever it encounters conflict with lock holding transaction regardless of its priority. But, if lock requesting transaction has higher priority, its priority is inherited into lower priority lock holding transaction. So the lock holding transaction with the increased priority can run faster and release resources earlier for lock requesting transaction. This scheme does not waste any system resources since there is no abort under any case. But, it does not deal with the priority inversion problem properly.

In this work, a new priority-based conflict resolution scheme is proposed. It is to provide more flexible scheduling so that it tries to reduce priority inversion. Especially, unlike existing works, lower priority lock requesting transaction can abort high priority lock holding transaction only if both transactions can meet their deadlines by doing so.

3. RTOODB Model

3.1. Basic Idea

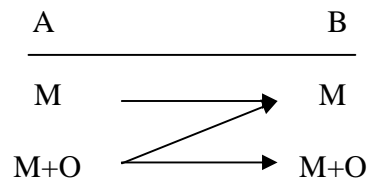
The proposed model is based on [8]. In [8], in order to decide commutativity of two methods, they use the notion of *affected set of attributes*. In their work, each method has DAV (Direct Access Vector). A DAV represents *most restrictive access mode* (among N (Null), R (Read), and W (Write)) for each attribute when the method is executed at run time. Their idea is as follows: Two methods commute if their corresponding DAVs commute. In turn, Two DAVs commute if their access mode is compatible for each attribute within two methods. Thus, it can have fine concurrency than mere read and write on entire instance. That is, for example, as long as two instance write methods modify disjoint set of attributes on same instance, two methods can run in parallel. Also, since an access mode of a DAV takes most restrictive access mode of attribute in DAVs of methods defined in that method, thus it reduces deadlocks due to lock escalation. Lock escalation may occur if a method M1 needs less restrictive mode on some attributes but later it requires more restrictive mode by nested method invocation. But, it has following disadvantages: 1) it is too

conservative since DAV of each method represents worst case mode on each attribute, resulting in less concurrency 2) For changes on method content, DAV should be reconstructed. Thus, for continuously evolving schemas, it takes many overheads.

In this work, the author's aim is to rectify disadvantages of scheme in [8] and provide trade-off between temporal consistency and logical consistency. Especially, the objectives are as follows: a) in determining commutativity of methods, make it less conservative. That is, give more concurrency. b) For changes on method content, try to reduce overhead on reconstruction of DAV. That is, divide each method into some parts and each part has its own DAV so that, even though some part is modified, we do not have to reconstruct DAVs of all parts. c) Provide flexibility in real-time scheduling so that partial result before deadline is better than complete result after deadline.

In the proposed work, the author adopts the concept of *imprecise computation* [7]. The idea is to divide each method into a *mandatory part* and an *optional part*. A *mandatory part* (M part) is required portion for acceptable result and must be computed. The *optional part* (O part) refines the result and can be left unfinished. The O part depends on the M part and becomes ready for execution when M part is finished. It can be terminated at any time after M part is finished. In scheduling this *imprecise computation*, two options are used. First, as long as a transaction can finish their work within deadline, then it finishes every method execution with M and O part. Otherwise, it finishes only M part of a method [7].

It is assumed that each method is invoked with either M part or (M+O) part. That is, an O part cannot be invoked independently without corresponding M part. Also, even though (M+O) part is requested, but after run time, only M part is invoked and gets a lock. The following gives table



Where A represents parts in which a lock requester T can have and also B represents parts in which T can have, after execution of method. Note that arrow from M+O to M means that M+O part is requested after start of execution, only M part is chosen. This is due to depending on 1) run-time result of M part and/or 2) deadline requirement of transaction.

For example, consider three methods M1, M2 and M3 on some object, say, O. The following is commutativity table, without considering M part and O part. Note that ‘Y’ and ‘N’ represent *commute* and *not commute*, respectively.

	M1	M2	M3
M1	N	N	N
M2	N	Y	N
M3	N	N	N

Fig. (a). Commutativity table in [8]

If we consider M part and O part, then we may have the following commutativity tables.

(M+O) part			
	M1	M2	M3
M1	N	N	N
M2	N	Y	N
M3	N	N	N

Fig. (b). The proposed scheme

M part			
	M1	M2	M3
M1	N	N	Y
M2	Y	Y	Y
M3	N	N	N

Fig. (c). The proposed scheme

M part			
	M1	M2	M3
M1	N	Y	Y
M2	Y	Y	Y
M3	Y	Y	Y

Fig. (d). The proposed scheme

In above figures, Fig. (a) is corresponding to Fig. (b). That is, both of lock holding method and requesting method has (M+O) part. In the proposed work, two extra tables are necessary for method commutativity.

3.2. RTOODB Model

In this Section, transaction model, method model and object model are introduced.

1) Transaction Model

A transaction is characterized $\langle trans-id, O, Exec, Prio \rangle$

tran-id : a unique transaction identifier

O : a set of operations representing the implementation of the transaction. These operations may include method invocations (MI), commit or abort statement, and statements for conditional branching, looping, and reads/writes on local variables. MIs are the forms $\langle Mn, Mand + Opt/Mand \rangle$, where *Mn* is method name, *Mand+Opt/Mand* is a Boolean field in which an invoking transaction requires *mandatory+optional part* or *mandatory part* of the method. In the proposed model, each method's computation has two parts, *mandatory part* and *optional part*

Exec : worst case execution time estimate of the transaction.

Prio : priority based on criticalness and/or deadline

2) Method model

A method *m* consists of $\langle Nm, Arg, OP, Exec, Mand+Opt/Mand \rangle$

Nm : name of the method

Arg : arguments of the method

OP : a set of operations representing the implementation of the method. These operations include statements for conditional branching, looping, I/O, and reads and writes to an attribute's value. Also, a method can call another method during its execution (i.e., nested method invocation). *OP* consists of two parts : *mandatory part* and *optional part*

Exec : worst case execution time of the method, $Exec = Mand-Exec + Opt-Exec$ where *Mand-Exec* and *Opt-Exec* is worst case execution time estimate of *mandatory part* and *optional part*.

Mand+Opt/Mand : *Mand+Opt/Mand* is a Boolean field, which is set/reset by an invoking transaction, representing whether an invoking transaction requires *mandatory+optional part* or *mandatory part* only.

3) Object model and commutativity tables

Each object has the following structure $\langle O\text{-id}, A, M \rangle$

O-id: unique object identifier

A: a set of attributes

M: a set of methods

For methods in an object, three commutativity tables, M+O-to-M+O, M+O-to-M, M-to-M, are constructed.

4. Priority-based Conflict Resolution Scheme

4.1. Basic Idea

The proposed scheme has the following characteristics.

A. Earlier Aborting

When a transaction should be blocked and will miss its deadline due to its blocking time, abort itself. Thus, release its resource and other blocked transactions earlier [1].

B. How to reduce priority inversion

B.1. A high priority lock requesting transaction aborts a blocked low priority lock holding transaction.

B.2. A lock requesting transaction with inherited priority cannot be blocked again, that is, abort conflicting lock holding transaction or abort itself.

This characteristic results in no deadlock.

The reason doing B.1 is as follows: when a higher priority transaction T is blocked by a blocked lower priority transaction T1, the blocking delay of the T may be unpredictable due to the reason such as T1 is waiting lock held by another transaction T2. That is, the *chained blocking* should be avoided. In this case, we abort blocked lower priority transaction T1.

For example, T1 has a lock on A and waiting for lock on B (since B is locked by another transaction). Later, T with higher priority than T1 comes and request a lock on A. Then, T aborts T1.

The reason doing B.2 is as follows: when a transaction T has inherited a higher priority, then it is supposed to be finished without delay. Otherwise, transaction(s) blocked by T may lose deadline(s). Also, if T is blocked, then it may result in deadlock. The following gives deadlock.

Assume that T, holding the lock on A, is blocked by T1 with Priority (T) > Priority (T1). Now, T1 with inherited priority of T is executing and requests a lock on A. If T1 is blocked again, it results in deadlock. In proposed scheme, all alternatives T can take are abort T1 or abort itself. Thus, there is no deadlock in the scheme.

C. How to protect lower priority lock holding transactions.

In order to prevent a low priority transaction, whose work is almost done from missing deadline when it encounters a blocked high priority transaction, we take the following steps: if a low priority transaction (Tr) has conflict with blocked high priority transaction (Th) and blocking itself may lose its deadline, abort Th only if either of one conditions hold: (1) restarting Th does not miss deadline (2) restarting Th miss deadline and the work performed ratio of Tr is bigger than that of Th.

Based on A, B and C, the basic principles are as follows.

1. Regardless of lock requester's priority, a lock requesting transaction aborts conflicting lock holding transaction only if either of the following two conditions is true.
 - a) If aborting lock holding transaction does not miss deadline and blocking lock requesting transaction miss its deadline
 - b) If blocking lock requesting transaction and restarting lock holding transaction miss their deadlines, select a transaction with the smaller workload performed so far as a victim.
- 2) Regardless of lock requester's priority, a lock requesting transaction with inherited priority cannot be blocked again, that is, abort the conflicting lock holding transaction or abort itself.
- 3) When a higher priority lock requesting transaction can abort a lower priority lock holder if restarting lock holder does not miss deadline.
- 4) A high lock requesting transaction always aborts blocked low priority lock holding transaction.

4.2. Conflict Resolution Algorithm

Whenever a transaction requests a lock, call conflict resolutions scheme *Resolve-Conflict* procedure. Also, The procedure first calls function *Decide-Lock-Conflict*, which is to determine if a lock requesting method can get a lock or not.

Function *Decide-Lock-Conflict*

//This function is to determine if a lock requesting transaction Tr. can get a lock or not. If a lock requesting

transaction can abort all conflicting transactions, then it returns T (True), otherwise F (False) //

Decide-Lock-Conflict \leq T;

For each conflicting transaction Th with lock requesting transaction Tr do

// Regardless of priority of Tr, we perform the following part //
 If blocking Tr miss its deadline and aborting Th does not miss deadline
 Then check next conflicting transaction
 // If it is True, Th can be aborted. Thus, continue to check next conflicting transactions //

Else if blocking Tr and restarting Th miss their deadline
 // We need to pick one of Tr or Th as a victim //
 Then if ratio of work done so far of Tr is greater than that of Th

Then check next conflicting transaction
 // T can abort Th with less damage //

Else Decide-Lock-Conflict \leq F

// Tr can not abort Th. Thus, stop conflict

resolution. It is

decided that Tr cannot get a lock //

End if

End if

// Depending on priority of Tr, different action can be taken //

Else if priority of Tr is greater than priority of Th

Then if (Th is blocked) OR (restarting Th does not miss deadline)

Then check next transaction

// If Th is blocked or restarting Th does not miss

deadline, Tr can abort Th //

Else Decide-Lock-Conflict \leq F; return //Tr

cannot abort Th//

End if

End if

End if

End do

End Function

The following is a conflict resolution scheme.

Procedure *Resolve-Conflict*

If *Decide-Lock-Conflict* = T

Then abort Th//abort all conflicting lock holding transaction and get a lock//

// If Tr cannot get a lock, under various conditions, Tr should be blocked or

abort itself. The following gives the conditions in which Tr should abort

itself //

Else if (Tr has inherited priority) OR

(Tr will miss its deadline due to blocking time

(FEASIBILITY

(Tr,Th)=F))

Then abort Tr

Else if priority of Tr is greater than priority of Th

Then priority of Th = priority of Tr;

//Th has inherited Tr's priority//

block Tr;

End if

End if

End if

End procedure

Function FEASIBILITY-TEST(Tr,Th)

// This function is used to test if a transaction to be blocked misses its deadline or not. If the transaction may miss deadline, F is returned, otherwise T is returned. //

If (remaining service of Tr) + (remaining service time of Th) \leq deadline of Tr

//If this is false, Tr will miss deadline due to blocking time. Thus, it is better to abort Tr earlier//

Then FEASIBILITY-TEST := T

Else FEASIBILITY-TEST := F

End if

End Function

5. Discussion and Further Work

In this paper, a real-time transaction processing technique is proposed. At first, a RTOODB model is presented. The model is based on imprecise computation. The model can provide more concurrency among methods and trade-off between logical consistency and temporal consistency. It can also avoid deadlocks due to lock escalation. But, in order to achieve these merits, extra work is needed to analyze source code of each method for deciding method commutativity and specify the mandatory part and the optional part for each method.

The real-time priority-based conflict resolution scheme is also proposed. The scheme is to try to reduce priority inversion in many ways. It aborts transactions earlier, whose deadlines can not be met. It also allows a low priority lock requesting transaction to get a lock as long as both lock holder and lock requester can meet their deadlines. In the meanwhile, if one of lock requester and lock holder must miss its deadline due to conflict access mode, the one with less system resource used is selected as a victim so that system resource is not wasted.

Currently the author is implementing the proposed real-time priority-based conflict resolution scheme in order to evaluate its performance with existing schemes. Also, the author is combining class hierarchy and class composition hierarchy, which are the important characteristics in OODBs, into the model.

References

- [1] K. Lam and W. Yau, "On Using Similarity for Transaction processing technique in Real-Time Database Systems", *The Journal of Systems and Software*, No. 43, 1998, pp. 223-232.
- [2] L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", *IEEE Trans. On Computers*, Vol. 39, No. 9, Sep. 1990, pp. 1175 – 1185.
- [3] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions", *ACM SIGMOD RECORD*, Vol. 17, No. 1, Mar. 1988, pp. 71 – 81.
- [4] O. Ulusoy and G. Belford, "Real-Time Transaction Scheduling in Database Systems", *Journal of Information System*, Vol. 18, No. 8, 1993, pp. 559 – 580.
- [5] L. Dipippo and V. Wolfe, "Object-Based Semantic Real-Time Transaction processing technique" *Proc. of Real-Time Systems Symposium*, 1993.
- [6] J. Lee, S. Son, and M. Lee, "Processing Real-Time Transactions in Object-Oriented Databases", *Tech. Report, Computer Science Dept. Univ. of Virginia*, Aug. 1994.
- [7] Jane W.S. Liu, et al, "Algorithms for scheduling Imprecise Computations", *IEEE Computer*, Vol. 24, No. 5, 1991, pp. 58 - 68.
- [8] C. Malta and J. Martinez, "Automating Fine Transaction processing technique in Object-Oriented Databases", *Proc. of the 9th Int. Conf. on Data Engineering*, Vienna, Austria, April, 1993, pp. 253 – 260.
- [9] Victor F. Wolfe and Lisa B. Cingiser, "Issues in Real-Time Object-Oriented Databases", *Proc. of IEEE Workshop on Real-Time Operating Systems and Software*, May, 1992
- [10] J. J. Prichard, Lisa Cingiser DiPippo, Joan Peckham, Victor F. Wolfe, "RTSORAC: Real-Time Object-Oriented Database Model", *Tech. Rep. Dept. of Computer Science, Univ. of Virginia*, 1994
- [11] Victor F. Wolfe and Lisa B. Cingiser, "Real-Time Object-Oriented Database Support For Intelligent Program Stock Trading", *Proc. of the 2nd Int. Conf. on Artificial Intelligence Applications on Wall Street*, Apr.1993.



Woochun Jun has been an a professor in Dept. of Computer Education at Seoul National University of Education , Seoul, Korea, since 1998.. His areas of interest include web-based instruction, mobile learning, web mining, semantic web. He holds a Ph.D. degree in Computer Science from University of Oklahoma, USA in 1997. He also received a Master's degree and BS degree in Computer Science from Sogang University, Seoul, Korea, in 1987 and 1985, respectively.